Framework de Desarrollo de Aplicaciones IoT sobre Dispositivos tipo Gateway en Smart Campus UIS

Alvaro Steven Mejia Ortiz

Santiago José Meneses Cáceres

Trabajo de Grado para optar al título de Ingeniero de Sistemas

Director

Gabriel Rodrigo Pedraza Ferreira

Doctor en Ciencias de la Computación

Codirector

Daniel Felipe Jaimes Blanco

Ingeniero de Sistemas

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingeniería de Sistemas e Informática

Bucaramanga

2026

Agradecimientos

Este trabajo no es únicamente el resultado de un proceso académico, sino la suma de aprendizajes, esfuerzos y acompañamientos que se consolidaron a lo largo de distintas etapas de mi vida personal y profesional.

Agradezco en primer lugar a la Universidad Industrial de Santander, por brindar el espacio académico, los recursos y el entorno necesarios para mi formación como ingeniero de sistemas. A los docentes que hicieron parte de este proceso, por aportar no solo conocimiento técnico, sino también criterio, disciplina y una visión responsable del ejercicio profesional.

Expreso un agradecimiento especial a mi director de proyecto, por la orientación, el acompañamiento y la confianza depositada en este trabajo. Sus observaciones y recomendaciones permitieron encaminar el proyecto hacia una propuesta coherente, viable y bien fundamentada, aportando claridad en momentos clave de su desarrollo.

Agradezco también a las personas con las que compartí espacios académicos y laborales durante este proceso, quienes, a través de conversaciones, discusiones técnicas y experiencias compartidas, influyeron de manera directa o indirecta en la concepción y evolución de este proyecto.

En el ámbito personal, agradezco de manera especial a mis padres y a mi familia, por el apoyo constante a lo largo de mi proceso académico y personal. Su acompañamiento, paciencia y respaldo fueron fundamentales para sostener este camino, especialmente en los momentos de mayor exigencia y dificultad. Más allá del apoyo material, su confianza y comprensión contribuyeron de manera significativa a que pudiera avanzar y culminar este proyecto.

De manera profundamente especial, quiero agradecer a mi tía, quien fue una presencia esencial en mi vida y un apoyo emocional constante. Su confianza en mí, su disposición para

escuchar y sus palabras de aliento marcaron momentos clave de mi proceso personal y académico. En etapas de cansancio, duda o incertidumbre, fue una de las personas que me brindó validación y tranquilidad, recordándome que era capaz de seguir adelante. Aunque ya no esté físicamente, su recuerdo y la huella que dejó en mí siguen presentes. Este trabajo también es, en parte, resultado de lo que ella creyó de mí cuando más lo necesitaba.

Finalmente, me agradezco a mí mismo por la constancia, la disciplina y la capacidad de continuar incluso cuando el camino se hizo más difícil. Este proyecto representa no solo un logro académico, sino también un proceso de crecimiento personal, resiliencia y reafirmación del propósito profesional que deseo seguir construyendo.

Alvaro Steven Mejia Ortiz.

Agradecimientos

Este trabajo de grado es el resultado de un proceso académico que estuvo acompañado de esfuerzo y aprendizaje, tanto en el ámbito académico como en el personal.

En primer lugar, agradezco a Dios por brindarme la fortaleza, la sabiduría y la perseverancia necesarias para culminar esta etapa de mi formación profesional y personal.

Agradezco a la Universidad Industrial de Santander por brindarme la oportunidad de formarme como ingeniero de sistemas, así como al director y al codirector del proyecto por el acompañamiento y las observaciones realizadas durante el desarrollo de este trabajo.

Agradezco a mis padres por su apoyo constante, su esfuerzo y su dedicación a lo largo de todo mi proceso de formación, los cuales fueron fundamentales para alcanzar este logro académico. A mis hermanos, por su apoyo y por estar presentes a lo largo de esta etapa.

Finalmente, agradezco a mi novia por su apoyo, paciencia y acompañamiento durante esta etapa de mi vida, que fueron muy importantes para poder sacar adelante este proceso.

Santiago José Meneses Cáceres

TABLA DE CONTENIDO

Lista de Tablas

Lista de Figuras

Lista de Apéndices

Los apéndices están adjuntos y puede visualizarlos en la base de datos de la biblioteca UIS

Resumen

Título: Framework de desarrollo de aplicaciones IoT sobre dispositivos tipo gateway en Smart Campus UIS

Autores: Alvaro Steven Mejia Ortiz y Santiago José Meneses Cáceres

Palabras clave: Rust, Framework de software, Extensibilidad, Flexibilidad arquitectónica, Smart Campus UIS, IoT Gateway.

Descripción: El desarrollo de soluciones para el Internet de las Cosas (IoT) en entornos heterogéneos, como el Smart Campus UIS, demanda arquitecturas de software que trasciendan la implementación de casos de uso específicos. Este trabajo presenta un framework modular diseñado en Rust, cuya propuesta de valor radica principalmente en la flexibilidad y extensibilidad de su arquitectura para dispositivos tipo gateway. A diferencia de soluciones monolíticas, el framework emplea un sistema de abstracción de hardware y una estructura de plugins que permite la integración dinámica de nuevos protocolos y sensores sin alterar el núcleo del sistema.

La implementación en Rust garantiza la seguridad de memoria y un manejo eficiente de la concurrencia, fundamentales para la robustez de la computación en el borde (edge). Para validar estas propiedades, se desplegó una infraestructura sobre Raspberry Pi integrando diversos sensores de monitoreo ambiental (DHT22, DHT11, Ds18b20 y MH-RD), demostrando la capacidad del framework para desacoplar la lógica de adquisición de datos de la lógica de comunicación. Se concluye que el diseño modular no solo agiliza el desarrollo actual en el Smart Campus, sino que asegura la evolución del sistema ante futuras tecnologías de sensado y conectividad, proporcionando una base técnica escalable y de alto rendimiento.

---

* Trabajo de grado

** Facultad de Ingenierías Fisicomecánicas. Escuela de Ingeniería de Sistemas e Informática. Director: Gabriel Rodrigo Pedraza Ferreira. Codirector: Daniel Felipe Jaimes Blanco.

Abstract

Title: Development Framework for IoT Applications on Gateway Devices in Smart Campus UIS

Authors: Alvaro Steven Mejia Ortiz y Santiago José Meneses Cáceres

Keywords: Rust, Software Framework, Extensibility, Architectural Flexibility, Smart Campus UIS, IoT Gateway.

Description: The development of Internet of Things (IoT) solutions in heterogeneous environments, such as the Smart Campus UIS, demands software architectures that go beyond the implementation of specific use cases. This work presents a modular framework designed in Rust, whose value proposition lies primarily in the flexibility and extensibility of its architecture for gateway devices. Unlike monolithic solutions, the framework employs a hardware abstraction system and a plugin-based structure that enables the dynamic integration of new communication protocols and sensor types without altering the system core.

The implementation in Rust guarantees memory safety and efficient concurrency management, both of which are fundamental to the robustness and reliability required by edge computing. To validate these properties, an infrastructure was deployed on Raspberry Pi devices, integrating various environmental monitoring sensors (DHT22, DHT11, DS18B20, and MH-RD), demonstrating the framework's ability to decouple data acquisition logic from communication logic. It is concluded that this modular design not only streamlines current development within the Smart Campus but also ensures the system's continued evolution as new sensing and connectivity technologies emerge, providing a scalable and high-performance technical foundation.

---

* Undergraduate thesis

** Faculty of Physicomechanical Engineering. School of Systems and Computer Engineering. Advisor: Gabriel Rodrigo Pedraza Ferreira. Co-advisor: Daniel Felipe Jaimes Blanco.

Introducción

El internet de las cosas (IoT) ha transformado significativamente la forma en que los sistemas interactúan con el entorno, desde aplicaciones en el hogar y la salud, hasta soluciones industriales y urbanas. Este panorama ha impulsado la necesidad de enfoques de desarrollo más eficientes, seguros y adaptables, especialmente en dispositivos gateway. Estos son clave, ya que actúan como puentes entre sensores distribuidos y la nube. A pesar de su papel estratégico, los gateways enfrentan desafíos considerables debido a sus limitaciones en memoria, la necesidad de operar con bajo consumo energético y la obligación de manejar múltiples protocolos de comunicación simultáneamente. Estos factores configuran un entorno de desarrollo complejo, especialmente cuando se requieren altos niveles de confiabilidad y seguridad en la transmisión y procesamiento de datos.

Ante estas complejidades, los desarrolladores suelen recurrir a plataformas que buscan simplificar el proceso, como Mongoose OS y MicroPython son bastante populares para desarrollar aplicaciones IoT. Son una excelente opción cuando se necesita algo rápido y sencillo, pero tienen limitaciones cuando se busca construir sistemas más complejos y robustos. Mongoose OS, por ejemplo, introduce un overhead de memoria significativo en dispositivos con recursos limitados, mientras que MicroPython sacrifica eficiencia para facilitar el desarrollo, lo que impacta el rendimiento en tareas críticas (Plauska, Liutkevičius, & Janavičiūtė, 2023).

Lo anterior evidencia la necesidad de un entorno de desarrollo que no solo sea ligero y seguro, sino también lo suficientemente flexible para adaptarse a diversos escenarios de gateway en el contexto IoT y extensible para soportar nuevos protocolos, sensores y dispositivos. En ese panorama, Rust se planta como una posible solución a estos problemas, gracias a su modelo de ownership que garantiza seguridad de memoria, su eficiente manejo de recursos y su creciente

ecosistema para sistemas embebidos (Sharma, Sharma, Tanksalkar, Torres-Arias, & Machiry, 2024). No obstante, actualmente el ecosistema de Rust para el desarrollo de aplicaciones gateway IoT es muy limitado o se encuentra en una etapa temprana, obligando a los equipos a implementar soluciones desde cero con el consiguiente aumento en tiempo y complejidad.

Este trabajo propone el diseño de un framework en Rust que priorice modularidad, seguridad y eficiencia, con soporte para protocolos comunes y capacidad de procesamiento local. El objetivo es proporcionar una herramienta bien documentada que facilite la creación de aplicaciones robustas para gateways IoT, reduciendo la barrera de entrada para desarrolladores y optimizando el proceso de implementación.

Como resultado principal, se entregará un framework documentado que incluya los componentes esenciales para el desarrollo de aplicaciones gateway con un conjunto de sensores ya establecido, junto con especificaciones claras para su uso, extensión e integración. La validación del framework se realizará mediante pruebas técnicas en escenarios representativos del Smart Campus, asegurando su aplicabilidad en contextos reales. Con ello, se espera contribuir al avance de soluciones IoT más accesibles, eficientes y sostenibles, aportando tanto a la literatura académica como al fortalecimiento del ecosistema de Rust en entornos académicos y urbanos.

1 Planteamiento y justificación del problema

El crecimiento acelerado de las redes IoT ha generado una presión significativa sobre los dispositivos en la capa Edge, en particular los gateways. Estos dispositivos, que operan con recursos limitados, deben encargarse de múltiples tareas críticas como la recolección de datos, su agregación y filtrado, el almacenamiento temporal, el procesamiento local en algunos casos, la transmisión eficiente hacia la nube o servidores centrales, y funciones de administración como la configuración y actualización remota de dispositivos conectados. (Hong et al., 2024)

La complejidad creciente de estas tareas ha puesto en evidencia las limitaciones de muchos lenguajes tradicionales para operar en entornos con recursos tan reducidos. Esto explica por qué, pese a la amplia cantidad de lenguajes existentes, continúan surgiendo nuevas propuestas orientadas a responder a necesidades técnicas que aún no están bien resueltas. Como señalan diversos estudios, la creación de nuevos lenguajes no busca reemplazar indiscriminadamente a los ya establecidos, sino cubrir vacíos específicos relacionados con seguridad, eficiencia y capacidad de adaptación a plataformas emergentes (Sharma et al., 2024). Esta evolución es especialmente relevante en entornos como el IoT, donde las restricciones son tan marcadas que exigen herramientas diseñadas para escenarios altamente especializados. Los dispositivos usados en gateways IoT, enfrentan múltiples restricciones: muy poca memoria, alta eficiencia energética requerida y necesidad de soportar múltiples protocolos de red simultáneamente. Estas condiciones dificultan el desarrollo de soluciones confiables, seguras y mantenibles, especialmente en aplicaciones con procesamiento local.

Aunque existen diversos frameworks, estos no resuelven completamente las necesidades específicas de los gateways Edge. Mientras algunos heredan las vulnerabilidades del lenguaje de programación, otros sacrifican rendimiento y control sobre los recursos del sistema. Esta brecha

tecnológica limita el desarrollo de aplicaciones que requieren alto desempeño, seguridad robusta y eficiencia en entornos con restricciones de hardware, planteando un desafío crítico para el ecosistema IoT. Estas carencias técnicas evidencian la necesidad de enfoques más adecuados para el desarrollo de software embebido en la capa Edge.

Esta problemática adquiere relevancia en el contexto colombiano, ya que esta situación se ve aún más agravada por la falta de formación especializada en desarrollo seguro para sistemas embebidos, así como por las limitaciones presupuestarias para adquirir hardware más potente. Esto impone barreras adicionales al diseño de aplicaciones eficientes y confiables en la capa Edge. (Chanchí-Golondrino et al., 2022).

Esta propuesta se enfocará en demostrar la completitud funcional y usabilidad desde la perspectiva del desarrollador, del prototipo. Este será implementado teniendo en cuenta los requerimientos prácticos de aplicaciones embebidas en entornos reales.

2 Objetivos

2.1 Objetivo general

Diseñar un framework de desarrollo de software que facilite la creación de aplicaciones IoT eficientes en dispositivos gateway, adaptado a las necesidades de la plataforma Smart Campus UIS, mediante una arquitectura modular, ligera y flexible.

2.2 Objetivos específicos

- Establecer los requerimientos funcionales y no funcionales del framework a partir del análisis del entorno y necesidades de la plataforma Smart Campus UIS.
- Diseñar una arquitectura modular para el framework que permita una integración flexible de componentes y facilite el mantenimiento, escalabilidad y evolución del sistema.
- Prototipar los componentes fundamentales del framework que soporten los principales requerimientos identificados.
- Validar el prototipo mediante pruebas funcionales representativas del ecosistema del Smart Campus.
- Documentar el uso, extensión y adaptación del framework a través de guías prácticas orientadas a desarrolladores de la plataforma Smart Campus.

3 Marco de referencia

3.1 Internet de las Cosas (IoT)

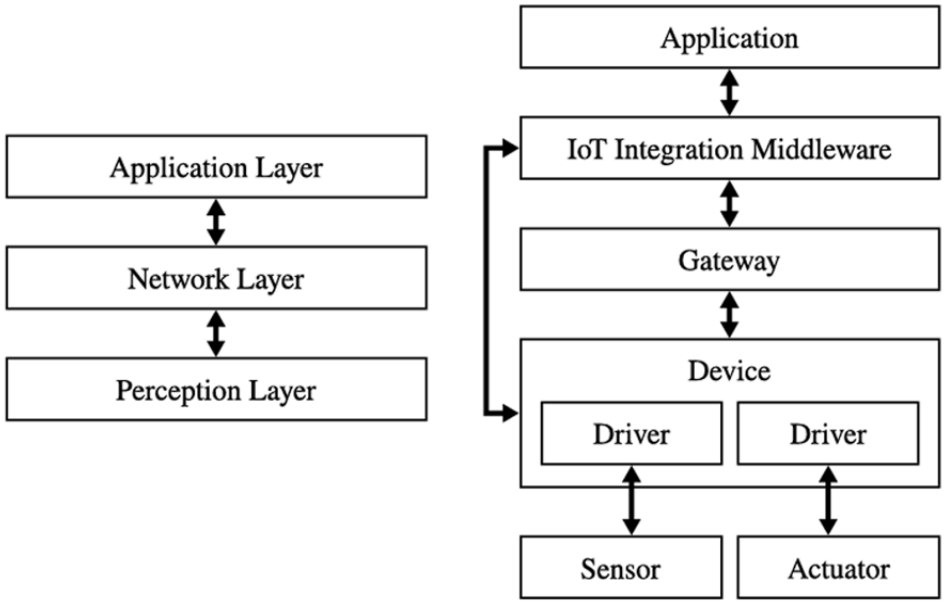
El Internet de las Cosas (IoT) se entiende como la conexión de objetos físicos que incorporan sensores, software y redes de comunicación para recolectar y compartir datos, buscando generar valor operativo y estratégico mediante el monitoreo, control y análisis de datos en tiempo real en diversos entornos (Domínguez-Bolaño et al., 2022). Tradicionalmente, los sistemas IoT se organizan en tres capas principales:

- Capa de percepción, donde los sensores y actuadores captan información del entorno.
- Capa de red, que transporta los datos hacia otros sistemas.
- Capa de aplicación, donde se procesan y presentan esos datos de forma útil.

Sin embargo, como explica el mismo estudio, han surgido arquitecturas más complejas, como modelos que estructuran los sistemas IoT en bloques funcionales diferenciados, incluyendo dispositivos y sensores, gateways, middleware de integración IoT y aplicaciones.

Figura 1.

Modelos de arquitectura IoT: modelo de tres capas y modelo de bloques funcionales



Nota. El gráfico muestra la descripción del Modelo de IoT de tres capas. Tomado de Domínguez-Bolaño et al. (2022).

Estas arquitecturas reflejan la creciente necesidad de manejar sistemas distribuidos, computación en el borde y mejorar la interoperabilidad entre dispositivos y plataformas.

En este sentido, si bien el modelo de tres capas permite comprender la base funcional del IoT, resulta insuficiente para las necesidades de hoy en día, donde las aplicaciones requieren procesamiento en tiempo real, gestión local de datos y operación continua incluso ante fallos de conectividad. Depender solo de la nube termina generando varios problemas, entre ellos la latencia, el consumo de ancho de banda y la exposición constante de datos sensibles (Hamdan et al., 2020; Salam et al., 2024).

Cuando los dispositivos IoT envían todos los datos sin procesar directamente a la nube, esos problemas se vuelven más evidentes. Por un lado, la latencia: en entornos donde se requiere respuesta inmediata como los sistemas industriales o urbanos inteligentes, el procesamiento exclusivo en la nube no logra tiempos de reacción adecuados, debido a la distancia y congestión en la transmisión de datos. También está la sobrecarga de red, porque enviar datos sin filtrar desde muchos dispositivos termina saturando las comunicaciones y aumentando los costos. Y, por último, la pérdida de conexión puede afectar gravemente la operatividad del sistema, limitando su capacidad de respuesta y procesamiento.

3.1.1 La aparición del edge y fog computing

Para superar estas limitaciones surgieron enfoques como el edge computing y fog computing, que básicamente llevan parte del procesamiento desde la nube hacia lugares más

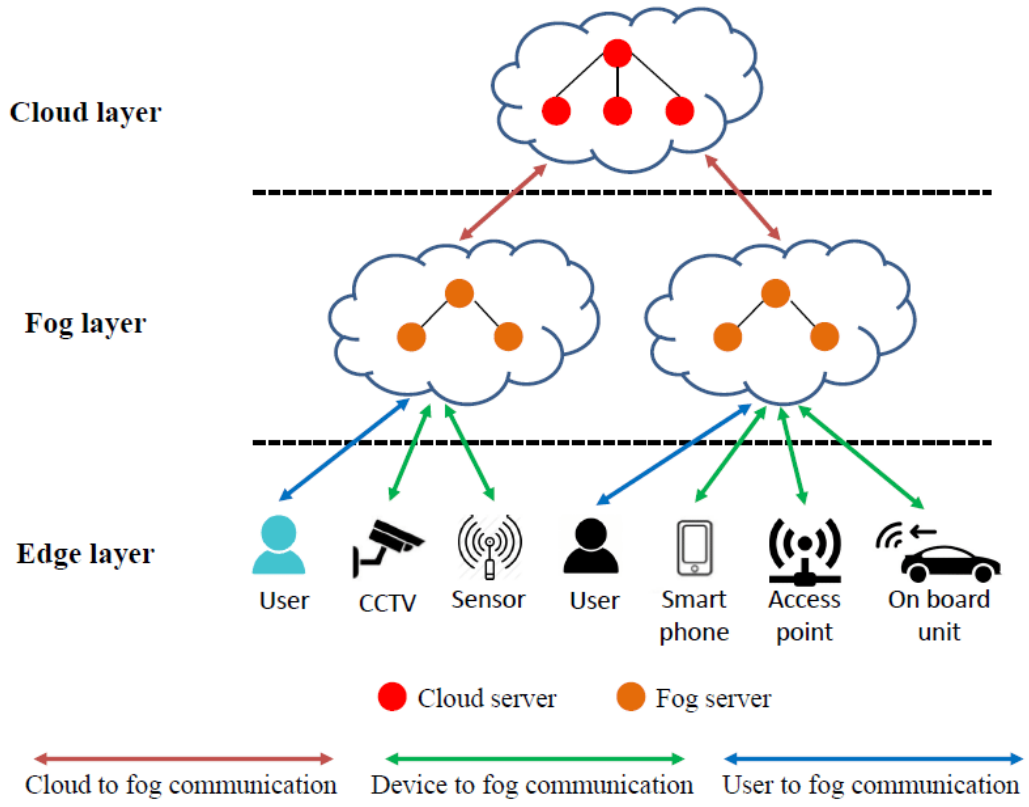
cercanos, en donde se generan los datos, esto con el fin de que las respuestas sean más rápidas y no se dependa de la conexión.

En el caso del edge computing este permite que el procesamiento ocurra directamente en los dispositivos periféricos o gateways, reduciendo significativamente la latencia y el tráfico hacia la nube (Dauda et al., 2024). El fog computing, en cambio, agrega una capa intermedia entre el edge y la cloud. Esa capa está distribuida en distintos puntos y puede almacenar o procesar parte de los datos antes de enviarlos, lo que ayuda a balancear mejor la carga.

En ambos casos, el gateway IoT deja de ser solo un puente de comunicación para convertirse en un nodo con capacidades de procesamiento, filtrado y seguridad.

Figura 2.

Arquitectura IoT basada en cloud, fog y edge computing



Nota. Distribución del procesamiento en una arquitectura IoT moderna, mostrando la ubicación de los dispositivos IoT, los gateways edge, los nodos fog y la nube. Adaptado de Three-layer architecture of fog computing (2024) por ResearchGate, https://www.researchgate.net/figure/Three-layer-architecture-of-fog-computing_fig1_355671165

Tabla 1.

Comparación entre arquitecturas IoT tradicionales y distribuidas

Criterio	Edge Computing	Fog Computing	Cloud Computing
Ubicación del procesamiento	En los dispositivos finales (sensores/dispositivos inteligentes).	En gateways o nodos intermedios cercanos a la red local.	En centros de datos remotos.
Capacidad de cómputo	Limitada por el hardware del dispositivo.	Media, superior al edge y compartida entre nodos.	Alta, con recursos prácticamente ilimitados.
Latencia	Muy baja.	Baja.	Alta.
Rol principal	Decisiones locales inmediatas.	Agregación y coordinación de múltiples edge nodes.	Almacenamiento y análisis global de datos.

Nota. La tabla se elaboró con base en la descripción de las capas edge, fog y cloud propuesta por Ortiz et al. (2024).

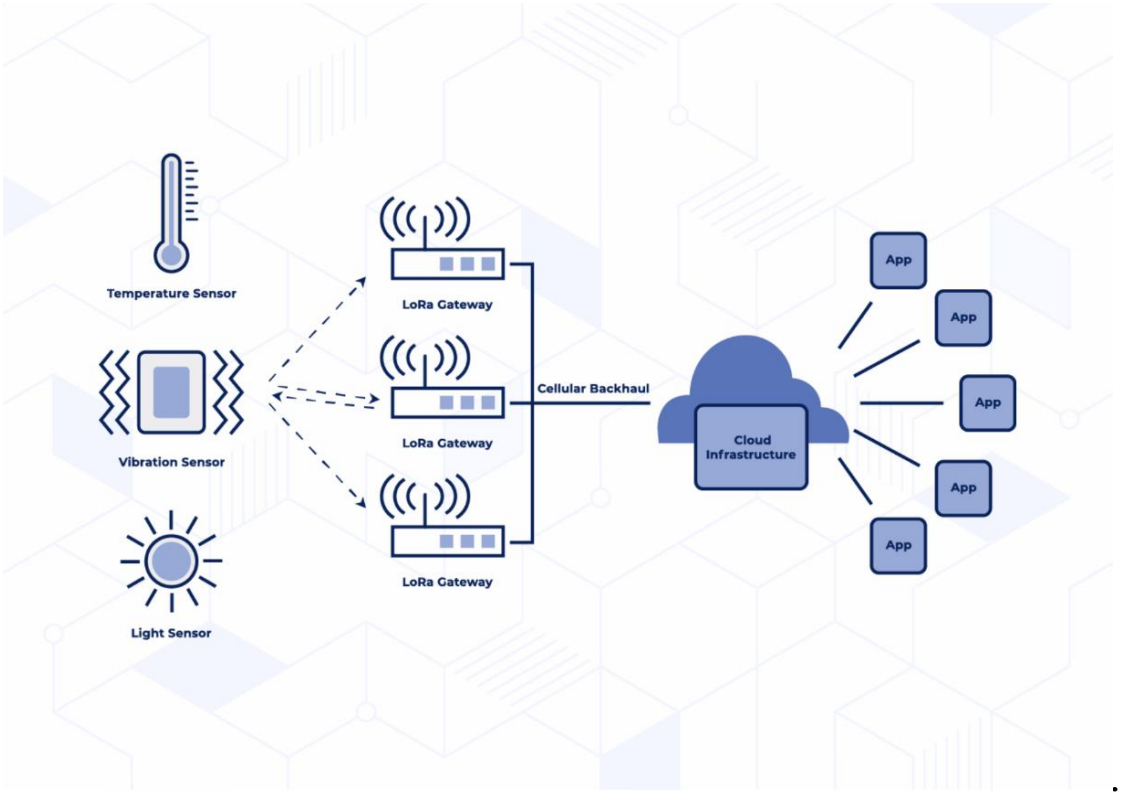
3.2 Dispositivos gateway en IoT

Un gateway IoT actúa como puente entre la capa de percepción y la nube, realizando tareas de conversión de protocolos, filtrado y preprocesamiento de datos. En las arquitecturas IoT modernas, estos dispositivos se ubican dentro de la capa edge, donde se ejecuta el procesamiento cercano a la fuente de datos para reducir la latencia, optimizar el tráfico y aplicar controles locales antes de transmitir información a las plataformas superiores. En este rol, equipos como la Raspberry Pi pueden funcionar como nodos gateway, al proporcionar la capacidad necesaria para gestionar múltiples interfaces y coordinar la comunicación entre los dispositivos del entorno y los

servicios externos. Estos dispositivos son esenciales para la gestión de la comunicación y la seguridad en sistemas IoT, ya que permiten la integración de múltiples dispositivos y tecnologías, asegurando que los datos se transmitan de manera eficiente y segura. (Ficili et al., 2025).

Figura 3.

Arquitectura de un gateway IoT



Nota. Diagrama de flujo de datos en un sistema IoT, que muestra la comunicación de sensores hacia gateways IoT, el envío de información hacia la infraestructura en la nube y su posterior consumo por aplicaciones de backend. Adaptado de What is an IoT gateway and how does it work?, por Zipit Wireless (2024).

3.3 Smart Campus UIS

El concepto de Smart Campus surge como una evolución de las iniciativas de Smart City aplicadas al entorno universitario. Su propósito consiste en integrar tecnologías digitales para optimizar procesos académicos, administrativos y operativos mediante la recopilación, procesamiento y análisis de información proveniente de múltiples fuentes distribuidas. Diversos estudios señalan que los campus inteligentes emplean tecnologías como Internet de las Cosas, computación en la nube, inteligencia artificial y analítica de datos para mejorar la eficiencia institucional y promover una gestión basada en información en tiempo real (Abuarqoub et al., 2022).

Dentro de este modelo, los sensores distribuidos en la infraestructura universitaria permiten monitorear variables como temperatura y humedad. La información generada facilita la toma de decisiones basada en datos, la automatización de procesos y la implementación de estrategias orientadas a la sostenibilidad y optimización de recursos.

La Universidad Industrial de Santander adelanta la iniciativa Smart Campus UIS con el objetivo de incorporar tecnologías IoT en la gestión de su infraestructura. Esta plataforma requiere soluciones capaces de integrar dispositivos heterogéneos, procesar datos localmente y transmitir información de forma confiable hacia sistemas centralizados. Las restricciones propias del entorno universitario como la diversidad de protocolos, la variabilidad de los sensores disponibles y las limitaciones presupuestarias imponen requisitos específicos sobre las herramientas de desarrollo utilizadas.

Esta necesidad constituye la motivación directa del framework, proporcionar una base de desarrollo reutilizable, modular y documentada que reduzca la complejidad de construir aplicaciones IoT sobre dispositivos gateway en el contexto del Smart Campus UIS.

3.4 Frameworks para sistemas embebidos

La complejidad creciente de los gateways y la necesidad de los Smart Campus ha impulsado el uso de frameworks para sistemas embebidos, que consisten en componentes reutilizables que estructuran tareas comunes, aceleran el desarrollo de aplicaciones y mejoran la confiabilidad de los sistemas. Entre los más utilizados en IoT se encuentran Zephyr, Mbed OS, RIOT OS, Tock OS y FreeRTOS, los cuales ofrecen distintas soluciones en modularidad, seguridad y eficiencia, según las necesidades del hardware.

3.5 Rust

Si bien los frameworks embebidos ofrecen una base estructurada para desarrollar software en dispositivos IoT, su implementación depende de lenguajes que permitan construir componentes seguros, modulares y eficientes. En este punto, Rust se presenta como una opción adecuada para desarrollar nuevos frameworks orientados a gateways, al proporcionar herramientas modernas para gestionar memoria, concurrencia y modularidad con mayor confiabilidad.

Este aspecto es especialmente relevante porque los gateways concentran múltiples redes, protocolos y flujos de datos, convirtiéndose en puntos críticos dentro de la infraestructura IoT. Debido a ello, el lenguaje base del framework debe contribuir a reducir vulnerabilidades comunes relacionadas con memoria, concurrencia y estabilidad del sistema, características que Rust aborda mediante un enfoque seguro por diseño.

Rust es un lenguaje de programación moderno que combina la eficiencia de lenguajes como C o C++ con mecanismos avanzados de seguridad en memoria, sin necesidad de utilizar un recolector de basura (garbage collector). Esta característica lo hace especialmente adecuado para entornos embebidos y de edge computing (Bugden & Alahmar, 2022).

Su sistema de ownership, junto con las verificaciones estrictas realizadas durante la compilación, permite evitar errores comunes como punteros colgantes (dangling pointers), condiciones de carrera (race conditions) y fugas de memoria (memory leaks), problemas frecuentes en sistemas concurrentes y de baja latencia.

Además de estas propiedades, Rust aplica estas garantías sin introducir sobrecarga significativa en tiempo de ejecución gracias a sus zero cost abstractions, permitiendo mantener un alto rendimiento incluso en dispositivos con recursos limitados. Esto resulta particularmente importante en gateways IoT, donde múltiples tareas relacionadas con lectura de sensores, procesamiento de datos y comunicación de red deben ejecutarse de forma simultánea y estable.

Estas características han llevado a la adopción de Rust en proyectos reales de sistemas embebidos e IoT, como el sistema operativo Tock OS, el framework Ferros y el entorno operativo Hubris, evidenciando su viabilidad para aplicaciones donde la confiabilidad y eficiencia son factores críticos.

3.5.1 Ecosistema y herramientas de desarrollo

El compilador rustc y el gestor de paquetes Cargo fortalecen la viabilidad de Rust para el desarrollo de software modular y mantenible. Cargo facilita procesos como la compilación, administración de dependencias, ejecución de pruebas y generación de documentación técnica, simplificando significativamente el ciclo de desarrollo.

Asimismo, el ecosistema de paquetes reutilizables conocidos como crates permite integrar funcionalidades desarrolladas por la comunidad, favoreciendo la construcción de aplicaciones extensibles y organizadas en módulos independientes. Esta capacidad resulta relevante para

frameworks IoT, donde componentes como sensores, almacenamiento y comunicación requieren desacoplamiento y facilidad de mantenimiento.

Rust también ofrece capacidades específicas para trabajar sobre hardware limitado mediante el uso del modo no_std, el cual permite desarrollar aplicaciones sin depender de la biblioteca estándar del lenguaje, reduciendo así el consumo de recursos y aumentando la compatibilidad con dispositivos embebidos.

De igual manera, el ecosistema embedded-hal proporciona interfaces de abstracción de hardware portables entre diferentes plataformas y microcontroladores, permitiendo reutilizar componentes y drivers de hardware sin depender completamente de implementaciones específicas de cada dispositivo (Sharma et al., 2024).

Gracias a estas herramientas, Rust puede implementarse en dispositivos IoT y plataformas gateway como Raspberry Pi, facilitando el desarrollo de soluciones modulares y eficientes para escenarios de edge computing y Smart Campus.

Rust proporciona capacidades particularmente relevantes para el desarrollo de aplicaciones IoT sobre gateways debido a su enfoque en seguridad, concurrencia y eficiencia. Su modelo de ownership y borrowing permite prevenir múltiples errores relacionados con memoria durante la compilación, disminuyendo riesgos de fallos en ejecución y mejorando la estabilidad del sistema.

Asimismo, Rust permite desarrollar aplicaciones seguras sin depender de un recolector de basura, manteniendo un comportamiento eficiente y predecible en tiempo de ejecución. Adicionalmente, sus mecanismos de concurrencia segura facilitan la ejecución simultánea de tareas relacionadas con sensores, comunicación y procesamiento de datos sin comprometer la integridad del sistema (Bugden & Alahmar, 2022).

Otra capacidad relevante corresponde al ecosistema de desarrollo de Rust, el cual facilita la organización de aplicaciones mediante módulos y crates reutilizables administrados a través de Cargo, favoreciendo el desarrollo de aplicaciones mantenibles y extensibles.

No obstante, Rust también presenta algunas limitaciones que deben ser consideradas durante el desarrollo de proyectos IoT. Una de las principales corresponde a la complejidad asociada al modelo de ownership, borrowing y lifetimes, debido a las restricciones y verificaciones impuestas por el compilador para garantizar seguridad en memoria.

Asimismo, estudios recientes sobre Rust en sistemas embebidos identifican desafíos relacionados con interoperabilidad, soporte de herramientas y disponibilidad de librerías especializadas para determinadas plataformas y dispositivos, especialmente en comparación con ecosistemas más consolidados como C/C++ (Petrillo, 2025).

A pesar de estas limitaciones, Rust representa una alternativa sólida para el desarrollo de frameworks IoT orientados a gateways, debido a que sus capacidades de seguridad, modularidad y eficiencia se alinean con los requerimientos de estabilidad, mantenibilidad y bajo consumo de recursos necesarios en aplicaciones del Smart Campus UIS.

3.6 Estado del arte

Para diseñar un framework que permita desarrollar aplicaciones IoT en dispositivos tipo gateway dentro del Smart Campus UIS, es fundamental analizar qué soluciones existen hoy, qué ofrecen y qué limitaciones presentan. A partir de eso, podemos entender mejor el espacio donde este proyecto se va a posicionar. Se consideran únicamente sistemas operativos ligeros, frameworks especializados en IoT y lenguajes adecuados para hardware de recursos limitados.

3.6.1 Sistemas operativos ligeros

Zephyr RTOS es un sistema operativo modular para dispositivos IoT que permite incluir solo los componentes necesarios, reduciendo el consumo de memoria. Su API con múltiples capas de abstracción facilita el desarrollo de aplicaciones complejas. Aunque su rendimiento puede ser menor que otros RTOS más simples, su consistencia y flexibilidad lo hacen una opción viable para diversas aplicaciones IoT (Belleza y Pignaton, 2018).

RIOT OS es un sistema operativo especializado para dispositivos IoT con restricciones de tamaño, memoria y consumo de energía. En el estudio de Belleza y Pignaton (2018), RIOT mostró resultados consistentes en pruebas de cambio de contexto de tareas, manejo de semáforos y comunicación entre tareas. Además, viene preempaquetado con pilas de red para múltiples protocolos.

Contiki es un sistema operativo ligero y flexible diseñado para redes de sensores inalámbricos. Ofrece soporte para la carga y reemplazo dinámico de programas y servicios individuales, lo que permite una arquitectura muy flexible y compacta. Contiki utiliza un núcleo basado en eventos, pero también proporciona multithreading preventivo opcional que se puede aplicar a procesos individuales (Dunkels, Grönvall, & Voigt, 2004).

FreeRTOS es un sistema operativo en tiempo real de código abierto, ampliamente utilizado en proyectos de microcontroladores pequeños debido a su capacidad de multitarea y servicios de comunicación y sincronización entre tareas. Aunque puede tener tiempos de respuesta más lentos en algunas operaciones, sigue siendo una opción popular por su predictibilidad, comportamiento determinista y accesibilidad de documentación. (Belleza y Pignaton, 2018; Ungurean y Gaitan, 2018).

Mongoose OS es un sistema operativo ligero diseñado para aplicaciones IoT, optimizado para dispositivos con recursos limitados. Se destaca por su capacidad de conectar dispositivos a plataformas en la nube como AWS, Google Cloud y otros servicios de IoT, utilizando APIs predefinidas que simplifican la integración.

3.6.2 Lenguajes de programación

Plauska et al. (2023) confirman que C y C++ siguen siendo los lenguajes más utilizados en sistemas embebidos gracias a su capacidad de interactuar directamente con el hardware y su eficiencia. Esta conclusión se basa en una evaluación de rendimiento que comparó la eficiencia de estos lenguajes mediante la implementación de varios algoritmos de procesamiento de datos y señales en un microcontrolador ESP32. El inconveniente es que la gestión manual de memoria los vuelve propensos a errores críticos de seguridad.

Mientras que Rust aparece como una opción muy prometedora. Barchi et al. (2023) demostraron que este lenguaje logra un excelente equilibrio entre rendimiento y seguridad, aplicándolo incluso en dispositivos basados en arquitecturas RISC-V. Además, Rust soporta entornos no_std, lo que facilita su uso en dispositivos de muy bajos recursos sin necesidad de un sistema operativo. Carnelos et al. (2025) validaron su uso en proyectos de TinyML, mostrando que Rust es perfectamente capaz de trabajar en dispositivos de bajo consumo como los gateways IoT que se quieren utilizar.

3.6.3 Frameworks especializados

RT-OCF (Lee et al., 2018) es un framework basado en el estándar de la Open Connectivity Foundation (OCF), que facilita la interoperabilidad entre dispositivos de diferentes marcas.

Aunque es útil para redes heterogéneas, su dependencia de hardware especializado limita su flexibilidad, ya que está diseñado para funcionar en plataformas específicas como TizenRT y Linux. Este framework está más orientado a la comunicación entre dispositivos dentro de una red que al desarrollo local sobre gateways, lo que lo hace menos adecuado para el enfoque centrado en gateways de este proyecto.

MicroPython es un framework que permite programar dispositivos IoT usando Python, un lenguaje conocido por su simplicidad. Según Plauska et al. (2023), MicroPython acelera el desarrollo en comparación con lenguajes más complejos, pero presenta desventajas en términos de rendimiento, ya que los programas en MicroPython son muchas veces más lentos que las implementaciones en otros lenguajes de programación como C y Rust.

TinyGo, por otro lado, es un framework que utiliza el lenguaje Go, que ha ganado popularidad en la comunidad de IoT. Este está diseñado para sistemas embebidos, lo que lo convierte en una opción atractiva para proyectos que requieren eficiencia y bajo consumo de recursos. También utiliza un planificador cooperativo y no adelanta tareas como un RTOS. Aunque aún está en las primeras etapas de desarrollo, mientras que Rust ha alcanzado un mayor grado de madurez y adopción.

Tabla 2.

Comparación entre Rust y lenguajes tradicionales en sistemas IoT

Característica	C / C++	Rust	MicroPython	TinyGo
Gestión de memoria	Manual, control total del programador	En tiempo de compilación mediante *ownership* y *borrowing*	Automática en tiempo de ejecución	Automática, con soporte limitado de *runtime*
Seguridad de memoria	Baja, susceptible a errores de manejo de memoria	Alta, validada en compilación	Alta, gracias a la abstracción del intérprete	Media, mayor que C/C++ pero inferior a Rust
Recolector de basura	No	No	Sí	Sí (ligero)
Rendimiento	Alto, el más eficiente en ejecución	Alto, comparable a C/C++	Bajo, penalizado por la interpretación	Medio–alto, inferior a C/C++ y Rust
Consumo de recursos	Bajo	Bajo–medio	Alto	Medio
Adecuación para gateways IoT	Media, eficiente pero menos seguro	Alta, combina eficiencia y seguridad	Baja, limitada por rendimiento y memoria	Media–alta, equilibrio entre simplicidad y desempeño

Nota. La tabla presenta una comparación de C/C++, Rust, MicroPython y TinyGo en términos de gestión de memoria, seguridad y rendimiento en sistemas IoT sobre ESP32, basada en los resultados experimentales reportados por Plauska et al. (2023).

3.6.4 Brechas identificadas

Luego de revisar el estado actual, se pueden identificar varias brechas claras:

- Hay soluciones que son muy especializadas, pero no flexibles, como Zephyr y RT-OCF.
- Hay plataformas que priorizan facilidad de uso, pero sacrifican eficiencia de recursos, como Mongoose OS y MicroPython.

- Si bien existen esfuerzos en Rust para IoT, aún no se consolidan como soluciones generalizadas, modulares y con documentación accesible.
- Además, muchas soluciones carecen de herramientas y guías adecuadas para su adopción en entornos académicos y de formación.

3.6.5 Necesidades para el framework

A partir del análisis y las limitaciones de las soluciones existentes, se identifican los elementos esenciales que un framework orientado a dispositivos tipo gateway debe contemplar. En primer lugar, se requiere una estructura modular que permita separar las funciones principales y facilitar la integración de nuevos componentes sin afectar al sistema completo. Esto es importante porque los gateways, como la Raspberry Pi, deben manejar tareas diversas y cambiar según las necesidades del entorno.

También es necesario que el framework sea liviano y eficiente, ya que estos dispositivos cuentan con recursos limitados y deben procesar datos, filtrar información y comunicarse con múltiples dispositivos al mismo tiempo. Asimismo, la seguridad debe estar integrada desde el diseño y en esto Rust ayuda porque previene fallos de memoria y errores que podrían afectar el funcionamiento del gateway.

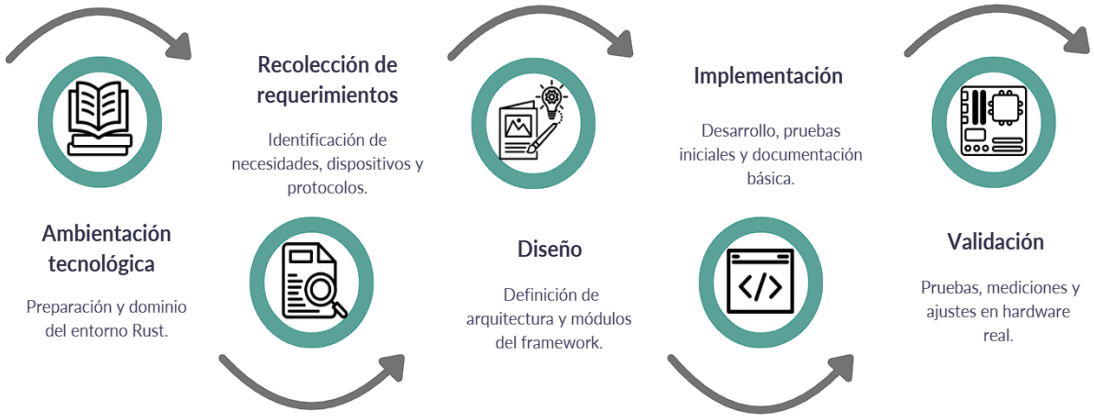
Finalmente, el framework debe ser claro y fácil de adoptar para entornos académicos. La documentación, la estructura interna y la posibilidad de modificar o extender el sistema deben apoyar tanto la experimentación como el desarrollo de nuevas aplicaciones sobre gateways como la Raspberry Pi.

4 Metodología

Este proyecto emplea una metodología de investigación aplicada para el diseño y desarrollo de un framework hecho en Rust para aplicaciones IoT destinado al Smart Campus UIS, donde se buscan cubrir necesidades como la comunicación con sensores, la gestión segura de memoria y la abstracción del hardware. Aplicando un enfoque de prototipado iterativo, el proceso se organiza en 5 fases que incluyen: ambientación tecnológica, análisis de requerimientos, diseño de la solución, implementación del framework y validación en hardware real.

Figura 4.

Diagrama de la metodología a seguir para el desarrollo del proyecto.



Nota. La imagen resume las cinco fases metodológicas del proyecto: ambientación tecnológica, recolección de requerimientos técnicos, diseño de la solución, implementación y validación, describiendo los objetivos y actividades desarrollados en cada etapa para la construcción y evaluación del framework IoT.

Ambientación tecnológica

Esta fase se enfocó en preparar el entorno de desarrollo revisando Rust y los crates usados en sistemas embebidos. El objetivo fue adquirir los conocimientos técnicos necesarios sobre el lenguaje, sus librerías y herramientas principales. También se evaluaron sus capacidades para manejar memoria con restricciones, trabajar con concurrencia y comunicarse con hardware. Con esto se estableció una base técnica sólida para avanzar al diseño e implementación del framework.

Objetivos

- Adquirir dominio del lenguaje Rust y su aplicación en entornos embebidos
- Evaluar las capacidades del ecosistema de Rust para sistemas embebidos.

Actividades

- Revisión y aprendizaje de Rust y crates embebidos
- Evaluación de condiciones del entorno de implementación

Recolección de requerimientos técnicos

Durante esta fase se realizó un levantamiento detallado de requerimientos mediante la identificación de protocolos, sensores y restricciones operativas del Smart Campus UIS. Se revisaron los dispositivos existentes y sus protocolos de comunicación, lo que permitió definir especificaciones técnicas precisas.

Objetivos

- Definir requerimientos funcionales y no funcionales del framework

Actividades

- Levantamiento de requerimientos funcionales y no funcionales
- Revisión de dispositivos existentes

Diseño de la solución

Esta fase se dedicó a definir la arquitectura y componentes del framework, adaptados específicamente al entorno embebido. Se realizó el diseño de la arquitectura general, se definieron los módulos, y se seleccionaron los crates y patrones de diseño más adecuados.

Objetivos

- Definir una arquitectura modular del framework adaptada al entorno embebido.
- Establecer los módulos, flujos y patrones de diseño que seguiría la solución.

Actividades

- Diseño de la arquitectura del framework.
- Definición de módulos y flujos de datos.
- Selección de crates y patrones de diseño.

Implementación de la solución

En esta fase se llevó a cabo el desarrollo progresivo de los componentes del framework. Se comenzó con un prototipado inicial y la implementación del esqueleto de los módulos, realizando pruebas mínimas en hardware real. Posteriormente, se avanzó con el desarrollo incremental, ampliando los módulos clave y aplicando integración continua con corrección temprana de errores.

Objetivos

- Desarrollar una versión funcional del framework con las capacidades básicas operativas.
- Establecer una comunicación estable con el hardware de prueba.

Actividades

- Prototipado inicial e implementación del esqueleto de módulos.
- Pruebas mínimas en hardware real.
- Desarrollo incremental y ampliación de módulos clave.

Validación de la solución

En esta fase final se verificó el funcionamiento del framework en condiciones reales y se consolidó la documentación del proyecto. Se realizó una validación funcional en Raspberry Pi 4, recopilando métricas de rendimiento y consumo de recursos. Como resultado, se obtuvo una validación completa del framework en escenarios reales, con los ajustes necesarios aplicados, y la documentación totalmente consolidada.

Objetivos

- Validar el framework en escenarios reales dentro del Smart Campus UIS.
- Consolidar la documentación final del framework.

Actividades

- Validación funcional en Raspberry Pi.
- Recopilación de métricas de rendimiento y consumo.
- Redacción y revisión de la documentación final del framework.

5 Ambientación tecnológica

La fase de ambientación tecnológica corresponde a la actividad de revisión y aprendizaje de Rust, crates embebidos y herramientas asociadas, esta tuvo como propósito adquirir el dominio práctico del lenguaje Rust y evaluar las herramientas del ecosistema disponibles para el desarrollo del framework, identificando sus capacidades y limitaciones en el contexto de aplicaciones IoT sobre dispositivos gateway.

5.1 Justificación del lenguaje: Rust en el contexto IoT embebido

La elección de Rust como lenguaje de implementación del framework responde a un conjunto de características técnicas que lo distinguen de las alternativas disponibles en el ecosistema embebido e IoT. A diferencia de lenguajes como C o C++, que ofrecen control directo del hardware pero delegan en el programador la gestión de la memoria y la prevención de errores de acceso, Rust garantiza la ausencia de condiciones de carrera y errores de memoria en tiempo de compilación mediante su sistema de ownership, sin recurrir a un recolector de basura que introduciría latencias impredecibles en sistemas de tiempo real (Bugden & Alahmar, 2022). Esta característica resulta especialmente relevante en dispositivos gateway que deben operar de forma continua y autónoma sin supervisión humana directa.

El ecosistema de Rust para sistemas embebidos ha madurado significativamente en los últimos años. Sharma et al. (2024) documentan cómo el modelo de ownership de Rust y su ecosistema de bibliotecas como embedded-hal establecen un estándar de abstracción de hardware que facilita el desarrollo de aplicaciones portables entre distintas plataformas ARM, precisamente el perfil de la Raspberry Pi. En un análisis comparativo entre MicroPython, JavaScript (Node.js) y Rust para el desarrollo de aplicaciones IoT en dispositivos con recursos limitados, Plauska et al.

(2023) demuestran que Rust alcanza tiempos de ejecución hasta cuatro veces menores que MicroPython con un consumo de memoria marcadamente inferior, lo que lo posiciona como una opción técnicamente superior para aplicaciones de adquisición de datos con alta frecuencia de muestreo.

Más allá del rendimiento puro, Rust ofrece un sistema de tipos expresivo que permite modelar el comportamiento de sensores, almacenamientos y comunicadores mediante traits, una abstracción directamente equivalente a las interfaces de lenguajes orientados a objetos pero evaluada en tiempo de compilación sin costo de indirección en tiempo de ejecución. Esta propiedad es la que hace viable construir un framework modular y extensible sobre Rust: los traits establecen contratos verificables entre componentes sin sacrificar eficiencia. Barchi et al. (2023) confirman que esta combinación de seguridad, rendimiento y expresividad hace de Rust una alternativa viable a C incluso en plataformas RISC-V de recursos muy limitados, fortaleciendo el argumento de que la elección del lenguaje no compromete la operabilidad del framework en plataformas de baja potencia como los dispositivos gateway utilizados en este proyecto.

5.2 Smart Campus UIS como contexto de integración

El Smart Campus UIS es la infraestructura de ciudad universitaria inteligente de la Universidad Industrial de Santander, diseñada para la recolección, procesamiento y visualización de datos provenientes de dispositivos IoT distribuidos por el campus. Su arquitectura se organiza en tres capas: una capa de sensores (Sensor Layer) responsable de la adquisición de datos físicos, una capa de borde o gateway (Edge Layer) donde reside el procesamiento local antes de transmitir hacia la nube, y una capa Cloud (Cloud Layer) que centraliza el almacenamiento y la visualización de los datos mediante tableros en Grafana. El componente Cloud del Smart Campus UIS expone

un broker MQTT para la recepción de mensajes, y espera que cada mensaje cumpla un esquema JSON estructurado con un encabezado que identifica el dispositivo emisor y un array de métricas que describen las variables medidas.

Lince se posiciona en la capa de borde de esta arquitectura. El framework corre en la Raspberry Pi que actúa como gateway, leyendo sensores físicos conectados mediante GPIO o protocolos como OneWire, procesando las lecturas localmente y publicando los datos al broker MQTT del Smart Campus UIS en el formato que el componente Cloud espera. Esta ubicación en la capa Edge no es accidental sino central al diseño: al liberar al desarrollador de la lógica repetitiva de adquisición, serialización y envío, Lince le permite enfocar el código de su aplicación en el procesamiento que únicamente tiene sentido hacer cerca del sensor filtros, umbrales de alarma, promedios deslizantes, decisiones de actuación , sin necesidad de enviar todos los datos en bruto hacia la nube para hacer ese trabajo allá, lo que reduciría la capacidad de respuesta y consumiría ancho de banda innecesariamente.

La fase de ambientación tecnológica verificó que la Raspberry Pi 4, con Raspberry Pi OS de 64 bits, es capaz de compilar y ejecutar aplicaciones Rust, acceder a GPIO mediante rppal, leer sensores DHT11, DHT22 y DS18B20 con los drivers implementados, y publicar datos al broker del Smart Campus UIS. Estos resultados confirman la viabilidad técnica de desplegar Lince en la capa Edge del Smart Campus y establecen las condiciones de referencia bajo las cuales se realizó el desarrollo y la validación del framework.

5.3 Evaluación de bibliotecas del ecosistema Rust

La selección de bibliotecas se realizó comparando alternativas disponibles en crates.io según criterios de integración con Raspberry Pi, ausencia de dependencias en otros lenguajes, compatibilidad con el modelo de seguridad de Rust y disponibilidad de documentación orientada a IoT.

Para el acceso a periféricos GPIO se evaluaron RPPAL, gpio-cdev y linux-embedded-hal. Se seleccionó RPPAL (Raspberry Pi Peripheral Access Library) por su integración específica con Raspberry Pi OS y su soporte para GPIO, I2C, SPI, PWM y UART mediante una interfaz de alto nivel (RPPAL Project, 2024). Su limitación principal es la dependencia de plataformas Raspberry Pi, lo que reduce portabilidad hacia otros dispositivos embebidos.

Como estándar de abstracción de hardware se adoptó embedded-hal, que define traits comunes como InputPin y OutputPin permitiendo desarrollar drivers independientes de una implementación específica de hardware (Embedded Rust Working Group, 2024). Esta biblioteca resultó fundamental para el diseño modular del framework, ya que desacopla la lógica de los sensores del acceso directo al hardware.

Para la comunicación MQTT se evaluaron rumqttc, Eclipse Paho MQTT y mqtt-async-client. Se seleccionó rumqttc por ser una implementación nativa de Rust sin dependencias externas en otros lenguajes, con soporte para MQTT 3.1.1, mecanismos de gestión de reconexión y configuración adaptada a entornos distribuidos (Bytebeam, 2024).

Para la configuración dinámica y serialización se seleccionó Serde junto con serde_json y la biblioteca toml. Estas herramientas permiten externalizar los parámetros del framework a archivos TOML y serializar estructuras Rust al formato JSON requerido por el protocolo del Smart Campus UIS, sin necesidad de modificar el código fuente (Serde Project, 2024).

Para el almacenamiento local se adoptó rusqlite, que proporciona acceso seguro a bases de datos SQLite. SQLite fue considerado adecuado para el contexto gateway por su bajo consumo de recursos y su funcionamiento sin servicios adicionales (Rusqlite Project, 2024). Se identificó como limitación su comportamiento en escenarios de escritura simultánea de alta frecuencia.

Para la gestión de errores se adoptó thiserror, que permite definir tipos de error diferenciados para cada módulo del framework sensores, almacenamiento y comunicación facilitando el diagnóstico de fallos en tiempo de ejecución (ThiserError Project, 2024). Complementariamente, la biblioteca spin_sleep fue incorporada para garantizar la precisión de temporización requerida por los protocolos de los sensores DHT11 y DHT22, que demandan tiempos de espera a nivel de microsegundos.

5.4 Capacidades y limitaciones identificadas del entorno

La evaluación del ecosistema confirmó que Rust ofrece las capacidades necesarias para construir el framework: seguridad de memoria garantizada en compilación, concurrencia sin recolector de basura, disponibilidad de bibliotecas especializadas para Raspberry Pi y una herramienta de gestión de dependencias (Cargo) que simplifica la integración modular.

Como limitaciones relevantes se identificaron dos. La primera corresponde a la curva de aprendizaje asociada al modelo de ownership, borrowing y lifetimes, que impone restricciones estrictas durante el desarrollo que no tienen equivalente en lenguajes como Python o C. La segunda, de particular importancia operativa, es que Linux no garantiza tiempos de respuesta de tiempo real. Dado que los protocolos de los sensores DHT11 y DHT22 requieren temporización precisa a nivel de microsegundos, las interrupciones del planificador del núcleo Linux pueden afectar la confiabilidad de las lecturas

Como resultado de esta fase se obtuvo el dominio técnico necesario sobre el lenguaje y las bibliotecas seleccionadas, y se estableció la base de decisiones que fundamenta la arquitectura del framework.

6 Definición de requerimientos

Esta fase corresponde a las actividades de levantamiento de requisitos, revisión de dispositivos y protocolos en Smart Campus, cuyo propósito fue identificar los requerimientos funcionales y no funcionales que sirvieron de base para el diseño del framework. La definición de los requerimientos del framework surge de la necesidad de contar con una plataforma escalable y robusta para el monitoreo ambiental en entornos distribuidos. El análisis de las características propias de los sistemas de monitoreo continuo revela desafíos técnicos que deben abordarse mediante especificaciones precisas.

La característica fundamental de cualquier sistema de monitoreo ambiental se basa en la capacidad de lectura de sensores físicos (RF01), requerimiento básico que permite la captura de información del entorno. Esta funcionalidad está diseñada para soportar distintos tipos de sensores, representativos de casos de uso típicos en el Smart Campus UIS, lo que permite verificar que el framework puede integrarlos y gestionarlos correctamente.

Ahora bien, la variabilidad en los parámetros a monitorear y la evolución tecnológica de los sensores disponibles en el mercado hacen necesaria la integración de nuevos sensores mediante módulos reutilizables (RF02). Esta capacidad se materializa en Rust mediante el uso de traits que definen un contrato claro para la integración de estos mismos. Cualquier nuevo sensor podrá incorporarse al sistema simplemente implementando el trait “Sensor” definido, sin necesidad de modificar el núcleo del framework.

Por tanto, debido a esta gran variedad y cantidad de sensores que usualmente se usan en un sistema de monitoreo ambiental, el framework requiere de un protocolo de comunicación eficiente para el envío de datos (RF03). Por esta razón se selecciona MQTT, protocolo que, por su bajo consumo de recursos y eficiencia en redes con limitaciones de ancho de banda, resulta ideal para entornos con múltiples dispositivos distribuidos.

La operación simultánea de múltiples sensores, sumada a la necesidad de mantener comunicaciones asíncronas con los sistemas centralizados, hace esencial que el framework implemente ejecución concurrente de tareas (RF04). Esta capacidad permite que las lecturas de sensores, el procesamiento de datos y el envío de información mediante MQTT ocurran de manera paralela, garantizando así el rendimiento del sistema cuando escala en número de dispositivos conectados. De igual forma que con los sensores, cualquier nuevo protocolo podrá integrarse al sistema implementando el trait “Communicator”, sin modificar el núcleo del framework.

Estas funciones deben implementarse con robustez, dado a la urgencia de las aplicaciones de monitoreo continuo, donde la pérdida de datos puede comprometer análisis futuros, establece la necesidad de que el sistema mantenga su operación incluso ante fallos parciales en sus componentes, como la desconexión temporal de un sensor o la indisponibilidad momentánea del broker MQTT, necesitando un manejo robusto de errores y tolerancia a fallos (RF05).

Todas estas capacidades deben garantizarse en plataformas hardware accesibles y ampliamente disponibles, por lo cual se necesita compatibilidad con Raspberry Pi (RF06), que balancea capacidades de procesamiento con consideraciones de costo y disponibilidad.

Adicionalmente, el ciclo de vida extendido de las plataformas de monitoreo y la probable evolución por parte de diferentes equipos de desarrollo del Smart Campus UIS hacen indispensable

la documentación y ejemplos de uso (RF07), en donde facilite la transferencia de conocimiento y acelere el proceso de incorporación de nuevos desarrolladores.

El framework debe permitir que cada aplicación pueda definirse mediante archivos de configuración externos, sin necesidad de modificar el núcleo del sistema (RF08). Para facilitar su uso, se proveerá una plantilla básica de configuración como ejemplo, que sirva de referencia para la personalización de parámetros según las necesidades de cada implementación.

Dado que Lince está pensado para integrarse en el ecosistema del Smart Campus UIS, resulta conveniente que el framework facilite, sin imponer, la comunicación con su componente Cloud, lo cual exige un formato de mensaje de envío de mensajes Smart Campus (RF09). Esta funcionalidad se ofrece como un módulo opcional independiente del comunicador MQTT, de modo que cualquier aplicación construida sobre Lince pueda optar por estructurar sus datos según este formato o definir su propio esquema de mensajes sin que el núcleo del framework imponga una u otra decisión.

Por otra parte, dado que la conectividad de red en despliegues IoT reales no siempre es estable, y que la pérdida de datos durante interrupciones de comunicación compromete los análisis posteriores, se requiere de almacenamiento persistente con reintento de envío (RF10). El framework debe incorporar, como alternativa al almacenamiento en memoria, un mecanismo basado en SQLite que conserve las lecturas pendientes de envío y las retransmita automáticamente al recuperarse la conexión, manteniendo el comportamiento por defecto del framework sin reintentos para no introducir complejidad innecesaria en aplicaciones que no la requieran.

Finalmente, dado que configurar manualmente un proyecto que combine sensores, comunicación y las funcionalidades opcionales descritas representa una barrera de entrada para usuarios sin experiencia previa en Rust, se requiere de un asistente de generación de aplicaciones

(RF11). El framework debe proveerse junto con una herramienta que permita seleccionar sensores, parámetros de hardware y funcionalidades opcionales mediante una interfaz simple, y que produzca como resultado un proyecto Rust completo, organizado en módulos, listo para compilar y desplegar en el dispositivo gateway.

La implementación de estos requerimientos funcionales en dispositivos edge con recursos limitados genera necesidades específicas a nivel no funcional. Las limitaciones computacionales de estos dispositivos exigen eficiencia de recursos (RNF01) y seguridad de memoria (RNF02), asegurando que el sistema opere dentro de los límites físicos del hardware sin comprometer su estabilidad.

Para organizar eficientemente estas capacidades, la modularidad (RNF03) permite desarrollar componentes de manera independiente, mientras que la escalabilidad (RNF04) anticipa el crecimiento progresivo en la cantidad de dispositivos y sensores gestionados, facilitando la expansión del sistema.

Finalmente, considerando el ciclo de vida extendido del framework, la mantenibilidad (RNF05) y observabilidad (RNF06) se orientan a garantizar la sostenibilidad a largo plazo, proporcionando mecanismos para su evolución controlada y diagnóstico efectivo de problemas durante la operación.

Esta definición de requerimientos establece las bases técnicas para una plataforma adaptable a diversos escenarios de monitoreo ambiental, dando como resultado las siguientes listas de requerimientos:

Tabla 3.

Requerimientos funcionales

Código	Requerimiento funcional	Descripción	Criterios de aceptación
RF01	Lectura de sensores físicos	El framework debe permitir la lectura de sensores físicos conectados al gateway mediante protocolos como GPIO.	• La lectura se realiza sin errores y devuelve un valor válido. • El tipo de sensor puede definirse en la configuración.
RF02	Integración de sensores con módulos reutilizables	Cada sensor debe ser implementado como un módulo que extienda un trait estándar.	• Se pueden cargar nuevos sensores sin modificar el código base del framework.
RF03	Envío de datos por MQTT	El sistema debe enviar datos recolectados a un broker MQTT, usando parámetros definidos por el usuario.	• La conexión MQTT debe ser estable y configurable (broker, tópico, QoS). • Se debe poder activar/desactivar la publicación.
RF04	Ejecución concurrente de tareas	El sistema debe poder realizar múltiples tareas en paralelo de forma asíncrona: lectura, envío, recepción y monitoreo.	• Se deben aceptar múltiples sensores simultáneamente.
RF05	Manejo de errores y tolerancia a fallos	El sistema debe registrar y manejar errores sin detener la ejecución completa.	• El sistema sigue operando si falla un sensor, un envío o una lectura.
RF06	Compatibilidad con Raspberry Pi	El framework debe ser ejecutable sobre Raspberry Pi OS Lite en dispositivos Raspberry Pi 4, utilizando crates como rppal o linux-embedded-hal.	• Los sensores deben funcionar correctamente en pruebas reales.
RF07	Documentación y ejemplos de uso	El proyecto debe incluir documentación técnica para desarrolladores y al menos un ejemplo funcional de uso del framework.	• Documentación generada con cargo doc. • Proyecto de ejemplo con sensor simulado y MQTT.

RF08	Configuración de la aplicación	El framework debe permitir que cada aplicación defina sus parámetros mediante archivos de configuración externos, sin necesidad de modificar el núcleo. Se proveerá una plantilla básica como ejemplo.	• La aplicación puede cargar parámetros desde el archivo de configuración. • La plantilla de ejemplo permite inicializar correctamente el sistema.
RF09	Formato de mensajes Smart Campus	El framework debe ofrecer, de forma opcional, un formateador que estructure los datos de sensor según el formato JSON requerido por el componente Cloud del Smart Campus UIS. El usuario decide si la aplicación usa este formato o define uno propio.	• El formateador produce un JSON válido con encabezado (deviceId, topic) y métricas (measurement, value). • El uso del formateador es opcional y no obligatorio para enviar datos por MQTT.
RF10	Almacenamiento persistente con reintento de envío	El framework debe incluir, además del almacenamiento en memoria, un mecanismo de almacenamiento persistente basado en SQLite que permita reintentar el envío de mensajes pendientes tras una interrupción de la comunicación. El uso de este mecanismo es opcional y, por defecto, el framework no reintenta el envío.	• Las lecturas se guardan en SQLite cuando el almacenamiento persistente está activado. • Tras recuperar la conexión, los mensajes pendientes se reenvían automáticamente.
RF11	Asistente de generación de aplicaciones	El framework debe proveerse junto con una herramienta que permita seleccionar sensores, parámetros de hardware y funcionalidades opcionales mediante una interfaz simple, generando como resultado un proyecto Rust completo, organizado en módulos, listo para compilar y desplegar.	• El proyecto generado compila sin errores con la configuración seleccionada. • Las funcionalidades opcionales se incluyen únicamente si fueron seleccionadas.

Nota. La tabla presenta la lista de requerimientos funcionales establecidos.

Tabla 4.

Requerimientos no funcionales

Código	Requerimiento no funcional	Descripción	Criterios de aceptación
RNF01	Eficiencia de recursos	El framework debe estar optimizado para dispositivos con recursos limitados (poca RAM, almacenamiento reducido y CPU modesta).	• El uso promedio de RAM en ejecución no debe superar los 10 MB.
RNF02	Seguridad de memoria	El framework debe evitar errores comunes de gestión de memoria como punteros nulos, fugas o condiciones de carrera.	• El código compila sin warnings ni errores.
RNF03	Modularidad	La arquitectura del framework debe estar dividida en módulos independientes, con interfaces claras entre ellos.	• Cada módulo puede desarrollarse, probarse y ampliarse por separado.
RNF04	Escalabilidad	El framework debe permitir agregar nuevos sensores, protocolos o funciones sin modificar el núcleo existente.	• Nuevos sensores deben poder integrarse implementando un trait estándar. • La arquitectura permite extender la lógica sin modificar el core.
RNF05	Mantenibilidad	El código del framework debe seguir buenas prácticas de ingeniería de software, permitiendo su mantenimiento y evolución a futuro.	• Comentarios claros y documentación en todas las interfaces públicas.
RNF06	Observabilidad	El sistema debe permitir rastrear fácilmente el comportamiento, errores y métricas internas durante su ejecución.	• Logs almacenables en archivo o mostrados por consola.

Nota. La tabla presenta la lista de requerimientos no funcionales establecidos.

7 Diseño arquitectónico del framework

Esta fase corresponde a las actividades de diseño de la arquitectura, definición de módulos y flujos, y la selección de crates y patrones de diseño, cuyo propósito fue definir una arquitectura modular que facilitara la integración, el mantenimiento y la escalabilidad del framework. El diseño del framework representa la materialización de los requerimientos funcionales y no funcionales identificados en la fase previa del proyecto. Esta sección documenta las decisiones arquitectónicas fundamentales que guían la construcción del sistema, estableciendo las bases sobre las cuales se implementarán las funcionalidades de lectura de sensores, almacenamiento de datos y comunicación con sistemas externos.

Por esto la fase de diseño aborda la organización estructural del framework mediante una arquitectura modular, que prioriza la separación de responsabilidades y la extensibilidad.

Posteriormente se documenta la función específica de cada módulo y sus interrelaciones, finalizando con una serie de diagramas que ilustran diferentes perspectivas de la arquitectura propuesta.

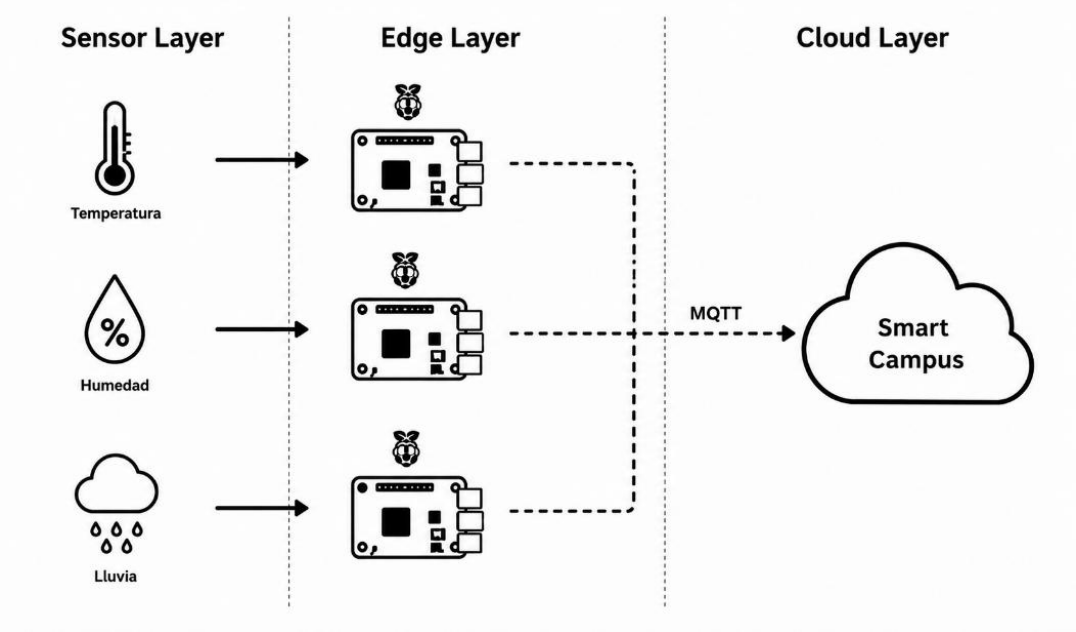
La arquitectura resultante permite que desarrolladores con conocimientos básicos de Rust puedan integrar nuevos sensores o sistemas de comunicación sin necesidad de modificar los componentes centrales del framework.

Adicionalmente, resulta importante identificar la ubicación del framework dentro de la arquitectura general del ecosistema IoT del Smart Campus UIS, ya que esto permite comprender el rol que cumple dentro del flujo completo de adquisición, procesamiento y transmisión de datos. En este contexto, el framework se ejecuta en la capa edge sobre dispositivos tipo gateway, actuando como intermediario entre los sensores físicos y la infraestructura cloud del Smart Campus. Esta ubicación es relevante porque evidencia que el framework no se limita a funciones

de visualización o monitoreo en la nube, sino que participa directamente en tareas de procesamiento local, integración de sensores, gestión de comunicación y transmisión de información.

Figura 5.

Arquitectura de integración del framework Lince en la capa Edge.



Nota. La Figura 5 presenta la arquitectura general de integración del framework Lince dentro del ecosistema Smart Campus UIS. La solución se organiza en tres capas principales: Sensor Layer, Edge Layer y Cloud Layer.

En la capa de sensores se encuentran los dispositivos encargados de capturar variables físicas del entorno, como temperatura y humedad. Estos sensores generan los datos que posteriormente son procesados por el gateway IoT.

El framework llamado Lince se ejecuta en la capa Edge, específicamente sobre dispositivos tipo gateway como Raspberry Pi. En esta capa, las aplicaciones desarrolladas con Lince realizan

tareas como adquisición de datos, procesamiento local, almacenamiento temporal y transmisión de información. De esta manera, Lince actúa como intermediario entre los sensores y la infraestructura en la nube.

Finalmente, en la capa Cloud, los datos son enviados mediante el protocolo MQTT hacia la plataforma Smart Campus UIS, donde pueden ser almacenados, visualizados y utilizados por otros servicios institucionales.

7.1 Computación en el borde

Cualquier aplicación IoT que opere sobre un gateway debe resolver un conjunto de tareas invariablemente repetitivas: inicializar el hardware, leer el sensor en un ciclo, convertir los datos a un formato transmisible, manejar fallos de red y almacenar lecturas cuando la conexión no está disponible. Estas tareas son necesarias, pero no diferenciadas, no es ahí donde reside el valor de una aplicación de monitoreo de temperatura en un campus universitario, ni de un sistema de detección de lluvia en un invernadero, ni de cualquier otro caso de uso IoT. El valor está en lo que la aplicación decide hacer con esos datos: qué umbrales activan una alarma, qué promedios sirven para detectar tendencias, qué combinaciones de sensores permiten tomar una decisión de actuación en el gateway mismo antes de que el dato llegue a la nube. Un framework resuelve el primer problema la capa repetitiva una sola vez, de forma probada y extensible, para que el desarrollador dedique su esfuerzo exclusivamente al segundo problema, que es el que varía entre una aplicación y otra (Abuarqoub et al., 2022).

La estandarización que un framework introduce va más allá de ahorrar líneas de código. Al definir contratos explícitos mediante los traits Sensor, Storage y Communicator, Lince establece que cualquier sensor nuevo se integra al sistema siguiendo el mismo patrón que los

sensores existentes, que cualquier mecanismo de almacenamiento puede sustituirse sin modificar la lógica de lectura, y que cambiar el destino de los datos no requiere reescribir nada por encima de la capa de comunicación. Esta regularidad beneficia no solo al desarrollador individual sino a equipos: cuando todos los componentes siguen las mismas interfaces, revisar, probar o extender el trabajo de otro desarrollador sobre el mismo framework es considerablemente más directo que leer una aplicación construida de cero (Shi et al., 2016).

La decisión de dejar el framework corriendo en la Raspberry Pi y no limitarse a usarla como un puente que reenvía bytes desde el sensor hasta la nube responde directamente al paradigma de la computación en el borde (edge computing). Shi et al. (2016) definen el edge computing como el conjunto de tecnologías que permiten ejecutar computación, almacenamiento y red en el borde de la red, lo más cerca posible de donde se generan los datos. En el contexto del Smart Campus UIS, la Raspberry Pi es exactamente ese borde: está físicamente junto a los sensores, conoce el dato antes que cualquier componente en la nube, y puede actuar sobre él sin incurrir en la latencia que implicaría enviarlo primero al servidor central y esperar una respuesta. Considérese un sistema de control de riego en el campus: una decisión de activar o desactivar la válvula en función de la humedad del suelo no debería requerir un viaje de red de ida y vuelta para que la nube decida; debería tomarse en el gateway, en milisegundos, con los datos que el sensor acaba de entregar. Lince posibilita exactamente eso: el desarrollador tiene acceso al valor de humedad como un número f32 antes de que ningún byte salga hacia el broker, y puede escribir cualquier lógica de decisión directamente en el loop de la aplicación.

Ejemplos concretos de lo que se puede implementar en la capa Edge con Lince incluyen calcular el promedio móvil de temperatura de los últimos diez ciclos antes de decidir si el dato es significativo para enviarlo (reduciendo el tráfico de red), detectar lecturas anómalas (valores fuera

de rango físico del sensor) y descartarlas sin transmitirlas, combinar datos de dos sensores temperatura interior y exterior para calcular un índice de confort antes de publicarlo, o activar una señal de alerta en un pin GPIO si la humedad supera un umbral definido sin depender de la nube para esa decisión. Todas estas operaciones ocurren en el mismo loop del gateway, entre la línea que lee el sensor y la línea que envía el dato, dentro del espacio que Lince le entrega libre al desarrollador tras haber resuelto la infraestructura repetitiva.

Si bien la computación en el borde define el contexto operativo en el que se ejecuta Lince y las responsabilidades que asume dentro del gateway, también plantea la necesidad de contar con una estructura de software que facilite la incorporación de nuevas funcionalidades sin afectar el resto del sistema. Para responder a este requisito, el framework adopta una arquitectura modular, en la que cada componente encapsula una responsabilidad específica y se comunica con los demás mediante interfaces bien definidas.

7.2 Arquitectura modular

El framework utilizará una arquitectura modular como base de su diseño, esta elección permite cumplir con los requisitos de modularidad y escalabilidad establecidos anteriormente. Esta forma de organizar el sistema en partes independientes pero conectadas ha sido utilizada con éxito en otros proyectos IoT, como lo muestran Szmeja et al. (2023) en ASSIST-IoT y Vecchio et al. (2021) en el proyecto AGILE, donde la modularidad ha demostrado ser especialmente útil para manejar la diversidad de dispositivos y protocolos típicos de estos entornos.

La principal ventaja de este enfoque es que permite extender el framework agregando nuevos componentes sin alterar su estructura central. Cada módulo se encarga de una función específica y se comunica con los demás mediante interfaces claras, lo que facilita situaciones en

las cuales se requiere extender, modificar o corregir alguna funcionalidad, los cambios quedan contenidos dentro de un módulo específico sin afectar el resto del framework.

Esta organización modular se concreta dividiendo el sistema en módulos especializados. Cada módulo tiene una función claramente delimitada, el núcleo del framework (core) define las abstracciones fundamentales mediante traits, el módulo para los dispositivos (devices) implementa la interacción con hardware específico, el módulo de almacenamiento (storage) gestiona la persistencia de los datos, el módulo de comunicación (network) maneja las comunicaciones externas y el módulo de controladores (drivers) proporciona acceso de bajo nivel al hardware. Esta estructura no solo hace que el framework sea más comprensible y organizado, sino que también permite que en el futuro se puedan agregar nuevas capacidades simplemente añadiendo módulos que sigan las mismas reglas de interfaz, asegurando así que el sistema pueda crecer de manera ordenada y sostenible.

7.2.1 Arquitectura general por capas

El framework implementa la arquitectura modular explicada anteriormente y adopta una organización en capas que estructura los diferentes módulos según su nivel de abstracción, implementación y control de hardware. Esta disposición separa las responsabilidades entre las diferentes capas.

En la capa superior está el módulo core, que establece los fundamentos del framework mediante los traits Sensor, Storage y Communicator. Estos traits definen el comportamiento que deben seguir todos los componentes del sistema, junto con tipos de datos compartidos como SensorOutput que garantizan consistencia en el procesamiento de información. Esta capa de abstracción funciona como un contrato que todas las implementaciones deben cumplir.

La capa de implementación contiene los módulos devices, storage y network, que materializan las abstracciones definidas en core. El módulo devices alberga las implementaciones específicas de sensores como DHT11, DHT22, DS18B20 y MH-RD que son ejemplos de sensores que se usan en la validación, cada uno adaptado a las particularidades de su protocolo de comunicación. El módulo storage proporciona MemoryStorage como implementación de referencia para persistencia temporal de datos. El módulo network implementa los comunicadores ConsoleCommunicator y MqttCommunicator para diferentes estrategias de transmisión de información.

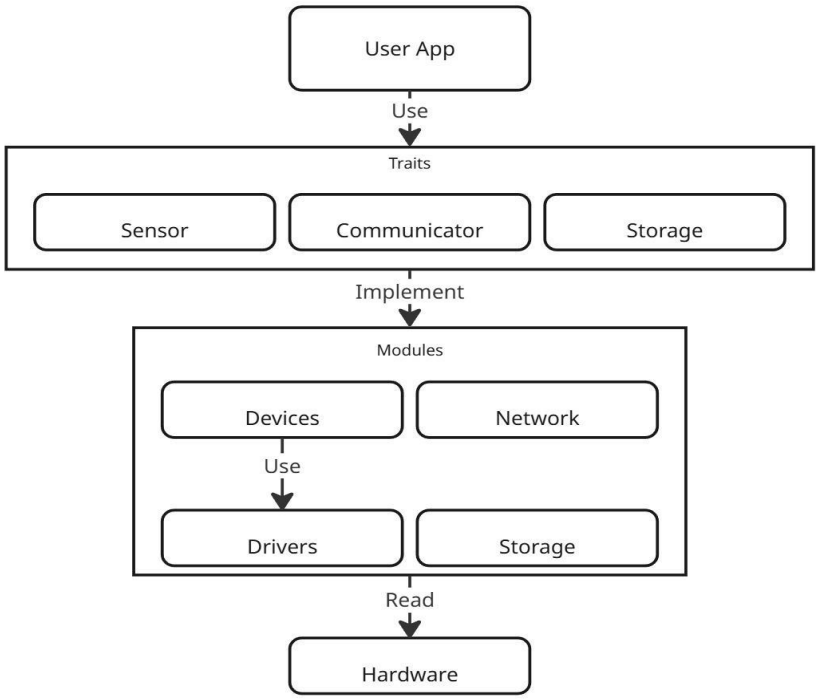
La capa inferior contiene el módulo de drivers, que constituye la interfaz de más bajo nivel con el hardware físico. El módulo drivers contiene GpioDriver, que abstrae el acceso a pines GPIO mediante una interfaz compatible con embedded-hal. Esta compatibilidad permite que el framework interopere con el ecosistema de Rust embebido, facilitando la portabilidad entre diferentes plataformas que implementen los traits estándar de embedded-hal.

El flujo de datos en la arquitectura sigue un patrón bien definido. Las lecturas se inician en la capa de drivers, donde GpioDriver gestiona la comunicación de bajo nivel con los sensores físicos. Los módulos de devices interpretan estas señales según los protocolos específicos de cada sensor y producen instancias de SensorOutput. Estos datos pueden almacenarse mediante implementaciones de Storage o transmitirse mediante Communicator hacia sistemas externos. Esta separación de responsabilidades permite que cada etapa del procesamiento se optimice independientemente.

El siguiente diagrama de arquitectura ilustra la organización en capas del framework y las dependencias entre módulos.

Figura 6.

Diagrama de arquitectura del framework.



Nota. Este diagrama proporciona una perspectiva sobre la arquitectura del framework, ilustrando de manera integral la estructura del sistema.

7.2.2 Reglas arquitectónicas, restricciones y criterios de extensión del framework

A partir de la arquitectura modular definida anteriormente, se establecieron una serie de reglas arquitectónicas que determinan cómo debe comportarse cada módulo del framework, qué responsabilidades puede asumir y cuáles son los límites que no debe sobrepasar. Estas reglas funcionan como una guía de diseño para mantener la modularidad, evitar dependencias innecesarias y asegurar que nuevas extensiones puedan integrarse sin modificar el núcleo del sistema.

El propósito de estas reglas es permitir que un desarrollador pueda extender el framework agregando nuevos sensores, mecanismos de almacenamiento o protocolos de comunicación siguiendo una estructura previamente definida. De esta manera, cada nuevo componente mantiene compatibilidad con el resto del sistema y conserva las propiedades arquitectónicas del framework.

Las restricciones definidas no representan errores del diseño, sino decisiones intencionales orientadas a mantener bajo acoplamiento, claridad estructural y facilidad de mantenimiento.

A continuación, se presentan las principales reglas, capacidades y restricciones establecidas para cada módulo del framework.

7.2.2.1 Módulo core. El módulo core funciona como el núcleo contractual del framework: define los traits Sensor, Storage y Communicator, junto con los tipos compartidos SensorOutput, SensorError, StorageError y CommunicatorError. Ningún otro módulo redefine estos contratos; todos los demás módulos del framework existen para implementarlos.

Tabla 5.

Reglas y ventajas del módulo core.

Aspecto	Detalle
Reglas establecidas	• Todo sensor debe implementar el trait Sensor. • Todo mecanismo de almacenamiento debe implementar el trait Storage. • Todo comunicador debe implementar el trait Communicator. • Los módulos externos no deben definir contratos centrales fuera de core. • Toda operación debe manejar errores devolviendo Result, nunca mediante panic en condiciones esperables.
Ventajas arquitectónicas	• Permite intercambiar implementaciones (p. ej. MemoryStorage por SqliteStorage) sin modificar el código que las consume. • Facilita pruebas unitarias mediante implementaciones simuladas de los traits. • Permite extender el framework con nuevos sensores o comunicadores compatibles con los traits existentes.

	• Mantiene una estructura única (SensorOutput) para representar datos de cualquier sensor.
Qué ahorra al desarrollador	Sin este contrato, cada combinación de sensor y destino de datos exigiría su propio código de integración. Con el trait Sensor unificado, una función que procesa lecturas puede escribirse una sola vez y operar sobre cualquier sensor presente o futuro, sin condicionales por tipo de dispositivo.

7.2.2.2 Módulo drivers. El módulo drivers proporciona acceso de bajo nivel al hardware físico mediante GPIO, encapsulado en GpioDriver, que implementa los traits InputPin y OutputPin de embedded-hal para garantizar compatibilidad con el ecosistema embebido de Rust.

Tabla 6.

Reglas y ventajas del módulo drivers.

Aspecto	Detalle
Reglas establecidas	• El acceso físico al hardware debe realizarse únicamente desde este módulo. • Los sensores no deben interactuar directamente con GPIO ni con rppal. • Los drivers deben exponer interfaces reutilizables y desacopladas del sensor que las usa. • El acceso a un pin debe mantenerse exclusivo mediante el sistema de ownership de Rust.
Ventajas arquitectónicas	• Permite reutilizar el mismo driver entre múltiples sensores que comparten protocolo. • Reduce la complejidad de acceso al hardware en los módulos superiores. • Facilita implementar nuevos protocolos de bajo nivel sin tocar la lógica de sensores existente.
Qué ahorra al desarrollador	Evita que cada implementación de sensor tenga que lidiar con la inicialización de rppal, el manejo de modos de pin o los errores de acceso a GPIO. El desarrollador de un nuevo sensor parte de GpioDriver ya probado, en vez de reescribir esa capa desde cero.

7.2.2.3 Módulo devices. El módulo devices contiene las implementaciones concretas de sensores físicos (Dht11Sensor, Dht22Sensor, Ds18b20Sensor, MhRdSensor), cada una encapsulando su propio protocolo de comunicación detrás del trait Sensor.

Tabla 7.

Reglas y ventajas del módulo devices.

Aspecto	Detalle
Reglas establecidas	• Cada sensor debe encapsular completamente su protocolo de comunicación. • Todo sensor debe retornar datos mediante la variante de SensorOutput correspondiente a su tipo de dato. • Los sensores no deben enviar datos directamente a la red. • Los sensores no deben almacenar información directamente.
Ventajas arquitectónicas	• Permite agregar nuevos sensores sin modificar el núcleo del framework. • Facilita reutilizar lógica compartida entre sensores similares, como DhtBase entre Dht11Sensor y Dht22Sensor. • Facilita probar cada sensor de forma individual. • Permite integrar sensores con protocolos distintos (1-Wire, GPIO digital) dentro del mismo sistema.
Restricciones	• Los sensores analógicos requieren un conversor analógico-digital externo para que la Raspberry Pi pueda interpretar la señal; actualmente no están soportados de forma nativa. • Sensores basados en I2C, SPI o UART requieren drivers especializados aún no implementados en el módulo drivers.
Qué ahorra al desarrollador	El protocolo de timing crítico de un sensor DHT, o el parsing del archivo w1_slave de un DS18B20, ya están resueltos. Agregar soporte para un nuevo sensor del mismo tipo no exige reimplementar el protocolo, sino seguir el mismo patrón ya validado en los sensores existentes.

7.2.2.4 Módulo storage. El módulo storage define las reglas relacionadas con almacenamiento y persistencia de lecturas, con dos implementaciones disponibles: MemoryStorage para buffer temporal en RAM, y SqliteStorage para persistencia con reintento de envío.

Tabla 8.

Reglas y ventajas del módulo storage.

Aspecto	Detalle
Reglas establecidas	• Todo almacenamiento debe implementarse mediante el trait Storage (save, list, clear). • El almacenamiento debe ser independiente de sensores y comunicación; no conoce el origen del dato ni su destino. • Una implementación de Storage puede exponer métodos adicionales propios (como flush_pending y pending_count en SqliteStorage) sin que esto forme parte del trait.
Ventajas arquitectónicas	• Permite cambiar la forma en que se almacenan los datos sin afectar sensores ni comunicadores. • Facilita implementar persistencia en memoria o en base de datos bajo la misma interfaz. • Permite usar almacenamiento como buffer ante fallos de red, con reintento automático cuando se usa SqliteStorage. • Facilita pruebas sin depender de una base de datos real, usando MemoryStorage.
Restricciones	• MemoryStorage no conserva información después de reiniciar el sistema. • El almacenamiento en memoria depende de la RAM disponible y no tiene límite de crecimiento incorporado. • No se incluyen filtros avanzados de consulta sobre los datos almacenados. • El reintento de SqliteStorage es opcional: si la aplicación no usa flush_pending, los mensajes pendientes simplemente se acumulan sin reenviarse.
Qué ahorra al desarrollador	Decidir entre persistencia y reintento no exige escribir lógica de SQL, manejo de conexiones o control de mensajes pendientes/enviados: storage.save() y storage.flush_pending() son las únicas líneas que cambian respecto a usar MemoryStorage, ya que ambas implementaciones cumplen el mismo trait.

7.2.2.5 Módulo network. El módulo network gestiona la comunicación hacia sistemas externos mediante MqttCommunicator y ConsoleCommunicator, e incluye además SmartCampusFormatter, un formateador opcional independiente de los comunicadores.

Tabla 9.

Reglas y ventajas del módulo network.

Aspecto	Detalle
Reglas establecidas	• Todo comunicador debe implementar el trait Communicator (un único método send). • El módulo únicamente transmite datos; no los produce ni los almacena. • Los comunicadores reciben datos ya serializados en bytes (&[u8]); la serialización ocurre fuera del comunicador. • SmartCampusFormatter no depende de ningún Communicator concreto: produce una cadena JSON que la aplicación decide cómo enviar.
Ventajas arquitectónicas	• Permite cambiar de protocolo de transmisión sin modificar sensores ni almacenamiento. • Facilita integrar MQTT, consola u otros protocolos futuros bajo la misma interfaz. • Facilita pruebas utilizando ConsoleCommunicator como sustituto de un broker real. • Permite que el formato Smart Campus sea opcional: una aplicación puede enviar datos sin pasar por SmartCampusFormatter en absoluto.
Restricciones	• El módulo no recibe datos externos (no implementa suscripción ni lectura desde el broker) en la versión actual. • No puede acceder directamente a sensores o almacenamiento; depende de que la aplicación le entregue los bytes a enviar.
Qué ahorra al desarrollador	Sin SmartCampusFormatter, cada proyecto tendría que reimplementar manualmente el esquema JSON exacto (header con deviceId/topic, métricas con measurement/value) que espera el componente Cloud del Smart Campus UIS. El formateador resuelve esa traducción una sola vez para todo el ecosistema de aplicaciones construidas sobre Lince.

Estas reglas arquitectónicas establecen una estructura organizada para el desarrollo y extensión del framework. Gracias a este enfoque, cualquier nuevo componente puede añadirse siguiendo contratos bien definidos, manteniendo la consistencia del sistema y evitando que futuras extensiones comprometan la modularidad de la arquitectura.

7.3 Decisiones arquitectónicas

Las decisiones arquitectónicas adoptadas durante el diseño del framework fueron orientadas principalmente a garantizar modularidad, extensibilidad, mantenibilidad y desacoplamiento entre componentes. Estas decisiones permiten que el framework pueda evolucionar progresivamente sin requerir modificaciones sobre el núcleo del sistema.

7.3.1 Uso de arquitectura modular

La arquitectura modular fue seleccionada para dividir el framework en componentes especializados con responsabilidades claramente definidas.

Este enfoque permite:

- Aislar funcionalidades específicas
- Reducir acoplamiento entre módulos
- Facilitar mantenimiento
- Simplificar pruebas
- Permitir extensiones futuras sin alterar el núcleo

Gracias a esta organización, nuevos sensores, comunicadores o mecanismos de almacenamiento pueden integrarse mediante módulos independientes.

7.3.2 Uso de traits como contratos

El framework utiliza traits como mecanismo principal para definir contratos de comportamiento entre componentes.

Esta decisión permite trabajar sobre abstracciones en lugar de implementaciones concretas, facilitando:

- Intercambio de componentes
- Reutilización de código
- Polimorfismo
- Extensibilidad
- Pruebas mediante implementaciones simuladas

De esta manera, cualquier módulo que implemente correctamente un trait puede integrarse al sistema sin modificar otros componentes.

7.3.3 Separación entre hardware y lógica de sensores

El acceso al hardware fue separado de la lógica de sensores mediante el módulo drivers. Esta decisión evita que cada sensor implemente directamente operaciones GPIO y permite reutilizar mecanismos de acceso físico entre múltiples dispositivos.

Además, esta separación facilita:

- Portabilidad hacia otras plataformas
- Reemplazo de bibliotecas de hardware
- Mantenimiento del código
- Reutilización de drivers

7.3.4 Uso de SensorOutput como estructura unificada

La utilización de SensorOutput como estructura unificada de intercambio permite representar distintos tipos de datos utilizando una misma interfaz interna.

Esta decisión facilita:

- Integración de sensores heterogéneos
- Procesamiento uniforme
- Desacoplamiento entre sensores y capas superiores
- Simplificación de almacenamiento y comunicación

Gracias a ello, módulos como storage y network pueden procesar datos sin depender del tipo específico de sensor utilizado.

7.3.5 Manejo explícito de errores

El framework implementa manejo explícito de errores en todas las operaciones críticas mediante retornos seguros que indican éxito o fallo de ejecución.

Esta decisión mejora:

- Estabilidad del sistema
- Robustez ante fallos
- Trazabilidad de errores
- Control del flujo de ejecución

Además, evita fallos silenciosos y reduce riesgos asociados a terminaciones inesperadas durante la adquisición o transmisión de datos.

7.3.6 Desacoplamiento de serialización y comunicación

La serialización de datos fue separada deliberadamente de los comunicadores.

Por esta razón, el módulo network únicamente transmite datos binarios y no conoce formatos específicos como JSON o CSV.

Esta decisión permite:

- Utilizar múltiples formatos de serialización
- Reemplazar protocolos sin modificar sensores
- Mantener comunicadores independientes del contenido
- Reutilizar la misma infraestructura de transmisión

Gracias a este desacoplamiento, el framework puede adaptarse a distintos escenarios de integración manteniendo la misma arquitectura interna.

7.4 Restricciones y alcance técnico

El framework fue diseñado con un alcance técnico específico orientado a gateways IoT ligeros basados en Raspberry Pi. Por esta razón, existen restricciones relacionadas con hardware, protocolos y capacidades operacionales que delimitan claramente el comportamiento esperado del sistema.

7.4.1 Restricciones de hardware

La implementación actual del framework se orienta principalmente a Raspberry Pi utilizando GPIO digitales.

Debido a ello:

- Sensores analógicos requieren un convertidor ADC externo

- La Raspberry Pi no puede leer señales analógicas directamente
- Protocolos como I2C, SPI y UART requieren drivers especializados
- Algunas operaciones dependen de temporización sensible al sistema operativo Linux

7.4.2 Restricciones de protocolos

Actualmente el framework implementa comunicación MQTT como mecanismo principal de transmisión externa.

La implementación actual:

- Realiza únicamente publicación de mensajes
- No implementa recepción bidireccional
- No incluye descubrimiento automático de dispositivos
- No administra autenticación avanzada ni balanceo de carga

La arquitectura permite incorporar nuevos protocolos, aunque estos aún no forman parte de la implementación validada.

7.4.3 Alcance actual del framework

El alcance actual del framework incluye:

- Lectura de sensores físicos
- Abstracción de hardware GPIO
- Procesamiento de datos de sensores
- Almacenamiento temporal
- Transmisión MQTT
- Arquitectura modular extensible

- Integración de nuevos sensores mediante traits

Sin embargo, el framework no pretende reemplazar plataformas IoT completas ni sistemas cloud integrales, sino enfocarse en la capa de recolección, procesamiento y transmisión de datos sobre gateways ligeros. Capacidades como los dashboards, la visualización gráfica, la analítica de datos o los servicios cloud integrados no son funcionalidades propias del framework; estas son provistas por herramientas y plataformas externas especializadas con las que Lince se integra. En el marco de este trabajo, esta integración fue validada mediante la conexión de Lince con la plataforma Smart Campus UIS, donde los datos transmitidos vía MQTT son consumidos, almacenados en InfluxDB y visualizados en Grafana, y analizados a través de los servicios propios de dicha plataforma. Esto evidencia la interoperabilidad del framework con el ecosistema de herramientas IoT existentes, sin necesidad de reimplementar dichas capacidades internamente, en coherencia con su enfoque como framework ligero y extensible.

El objetivo principal del framework es facilitar el desarrollo de aplicaciones IoT sobre gateways ligeros mediante una arquitectura modular reutilizable y extensible.

8 Prototipo del framework

Esta fase corresponde a las actividades de prototipado inicial, implementación de módulos, pruebas mínimas en hardware real, desarrollo incremental e integración continua y corrección de errores tempranos, cuyo propósito fue desarrollar los componentes funcionales del framework de acuerdo con la arquitectura definida en la fase anterior. Este prototipo representa la materialización de los conceptos arquitectónicos presentados en el capítulo anterior, destacando que su objetivo no es únicamente demostrar funcionalidad, sino validar las decisiones de diseño relacionadas con modularidad, extensibilidad y flexibilidad en sistemas embebidos.

Para ello se destaca la selección de Raspberry Pi 4 como plataforma de desarrollo, no solo por sus características técnicas, sino por su posición intermedia entre sistemas embebidos de recursos muy limitados y computadoras de propósito general. Esta elección permite validar que el framework puede operar eficientemente en hardware con restricciones reales mientras mantiene suficiente capacidad para desarrollo y pruebas.

El framework IoT desarrollado, denominado Lince, fue diseñado e implementado siguiendo los principios definidos previamente. Su código fuente y estructura modular se encuentran disponibles para consulta y análisis en el siguiente repositorio: Saiduts/lince: Rust IoT Framework

A partir de este punto, se describe la arquitectura interna del framework, su organización en módulos y los principales componentes implementados, haciendo referencia directa a dicha estructura.

8.1 Ecosistema de desarrollo

El desarrollo del framework se fundamenta en Rust, un lenguaje que ofrece características únicas para sistemas embebidos de seguridad crítica. La elección de Rust no responde únicamente a consideraciones de rendimiento, sino a su modelo de ownership que garantiza seguridad de memoria sin necesidad de un garbage collector, un requisito esencial cuando se trabaja con acceso directo a hardware GPIO. El compilador de Rust actúa como una primera línea de defensa contra errores comunes en programación de sistemas, detectando en tiempo de compilación problemas que en otros lenguajes solo se manifestarían durante la ejecución, causando comportamientos inadecuados en el hardware.

El entorno de despliegue se orienta específicamente hacia Raspberry Pi 4, aprovechando el acceso directo a pines GPIO mediante la biblioteca rppal. La decisión de usar rppal en lugar de

interfaces más abstractas responde a la necesidad de control preciso sobre el timing de las señales, especialmente crítico en protocolos como el utilizado por los sensores DHT, donde variaciones de microsegundos pueden causar fallos de lectura.

8.2 Organización modular del código

La estructura del proyecto refleja directamente los principios arquitectónicos del framework, organizándose en módulos que corresponden a las capas conceptuales del sistema.

El módulo core encapsula las abstracciones fundamentales del sistema mediante la definición de traits que establecen contratos de comportamiento. El trait Sensor define la interfaz que cualquier sensor debe implementar, independientemente de su tecnología subyacente o protocolo de comunicación. Esta abstracción permite que el resto del framework trabaje con sensores como concepto genérico, sin implementaciones específicas. De manera similar, los traits Storage y Communicator establecen contratos para almacenamiento y comunicación, facilitando que nuevas implementaciones se integren sin modificar el código existente. Adicionalmente, el submódulo types define las estructuras de datos compartidas como SensorOutput y los tipos de error SensorError, StorageError y CommunicatorError, garantizando consistencia en el manejo de datos y errores a través de todo el sistema.

El módulo de sensores agrupa implementaciones relacionadas mientras las mantiene independientes. Cada archivo corresponde a un modelo específico de sensor, conteniendo únicamente la lógica necesaria para su operación. Esta organización facilita la navegación del código y permite que múltiples desarrolladores trabajen en diferentes sensores simultáneamente sin conflictos.

El módulo drivers aloja las implementaciones de bajo nivel que comunican el software y el hardware físico implementando GpioDriver que define la forma estándar en la que el framework accede a los pines digitales.

El módulo storage contiene implementaciones del trait Storage, comenzando con MemoryStorage como caso base en donde se guardan las lecturas de los sensores en memoria y SQLiteStorage.

El módulo network agrupa las implementaciones de comunicadores que permiten al framework interactuar con sistemas externos. ConsoleCommunicator proporciona salida a terminal para debugging y desarrollo, mientras que MqttCommunicator implementa conectividad IoT completa mediante el protocolo MQTT.

La arquitectura modular resultante establece dependencias unidireccionales claras donde las capas superiores dependen de abstracciones definidas en core, mientras que las implementaciones concretas en devices, storage y network dependen únicamente de estas abstracciones, nunca entre sí. Esta estructura previene dependencias circulares y facilita testing unitario al permitir que cada módulo se pruebe de forma aislada con los traits necesarios.

8.2.1 Implementación de drivers de hardware

La capa de drivers representa el punto de contacto más bajo entre el software del framework y el hardware físico de la Raspberry Pi. Esta capa existe para resolver un problema fundamental en el desarrollo de sistemas embebidos: proporcionar una abstracción segura y consistente sobre el hardware que permita a los componentes de nivel superior interactuar con sensores sin acoplarse a los detalles específicos de la plataforma.

El módulo GpioDriver surge como respuesta a la necesidad de estandarizar el acceso a pines digitales, ofreciendo una interfaz uniforme que oculta la complejidad del manejo directo de registros de hardware. Esta abstracción resulta crítica porque los pines GPIO constituyen el medio primario de comunicación con sensores digitales en sistemas embebidos, y su manejo incorrecto puede causar desde lecturas erróneas hasta daños físicos en el hardware. El driver aprovecha el sistema de ownership de Rust para garantizar que un pin solo pueda ser controlado por una única parte del código en cualquier momento, eliminando errores relacionados con acceso concurrente a recursos hardware.

8.2.2 Sensores utilizados

Con esta infraestructura de acceso establecida, el siguiente paso en la construcción del prototipo consistió en implementar los drivers para los sensores seleccionados en el diseño. Estos sensores representan casos de uso diversos que prueban diferentes aspectos del framework como una comunicación con timing preciso, lectura mediante sistema de archivos del kernel, y detección de estados digitales simples. La variedad en complejidad y metodología de estos sensores permite validar que la arquitectura del framework puede adaptarse a diferentes paradigmas de comunicación sin forzar soluciones inadecuadas.

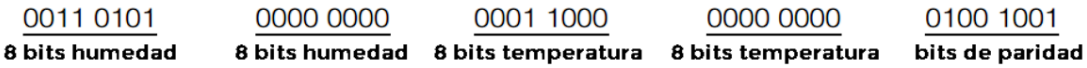
Los sensores DHT11 y DHT22, a pesar de sus diferencias en precisión y rango, comparten un protocolo de comunicación idéntico a nivel de bits. Este protocolo propietario requiere un timing preciso y un manejo cuidadoso de estados. El protocolo funciona mediante modulación de ancho de pulso, donde la duración de un pulso alto determina si el bit transmitido es 0 o 1.

La secuencia de comunicación comienza con el microcontrolador llevando la línea de datos a nivel bajo durante aproximadamente 20 milisegundos, señalizando al sensor que debe preparar

una transmisión. El sensor responde con una secuencia de sincronización, alternando la línea entre bajo y alto para establecer timing de referencia. Esta fase permite que el sensor y el microcontrolador sincronicen sus relojes internos, compensando diferencias en frecuencias de oscilador. Una vez completada la sincronización, el sensor transmite 40 bits organizados en cinco bytes.

Figura 7.

Bits correspondientes al sensor DHT11.



Nota. Tomada de Cómo utilizar el DHT11 para medir la temperatura y humedad con Arduino

Los primeros dos bytes representan la humedad relativa, los siguientes dos la temperatura, y el quinto byte funciona como checksum de suma simple, para ello se reúne toda la lógica necesaria para manejar el protocolo del DHT, como iniciar la comunicación, leer los bits y verificar que los datos sean correctos. Para detectar los cambios rápidos en la señal, se utiliza una espera activa con límite de tiempo, ya que este tipo de sensores requiere medir pulsos muy cortos con precisión. Sin embargo, la latencia introducida por interrupciones en Linux resulta demasiado variable para las exigencias temporales del protocolo DHT. El uso de un timeout evita que el sistema quede bloqueado en caso de fallas o desconexiones del sensor, permitiendo que los drivers específicos solo se enfoquen en interpretar los valores finales.

8.2.2.1 Sensor de temperatura y humedad DHT11. El DHT11 es un sensor digital básico de temperatura y humedad, ideal para proyectos de aprendizaje y aplicaciones de bajo costo, este solo utiliza la parte entera de los valores de temperatura y humedad, desperdiciando los bytes decimales que siempre contienen cero. Esta simplificación limita la resolución del sensor a un grado Celsius y un punto porcentual de humedad relativa, suficiente para aplicaciones no críticas pero inadecuada para mediciones precisas.

Figura 8.

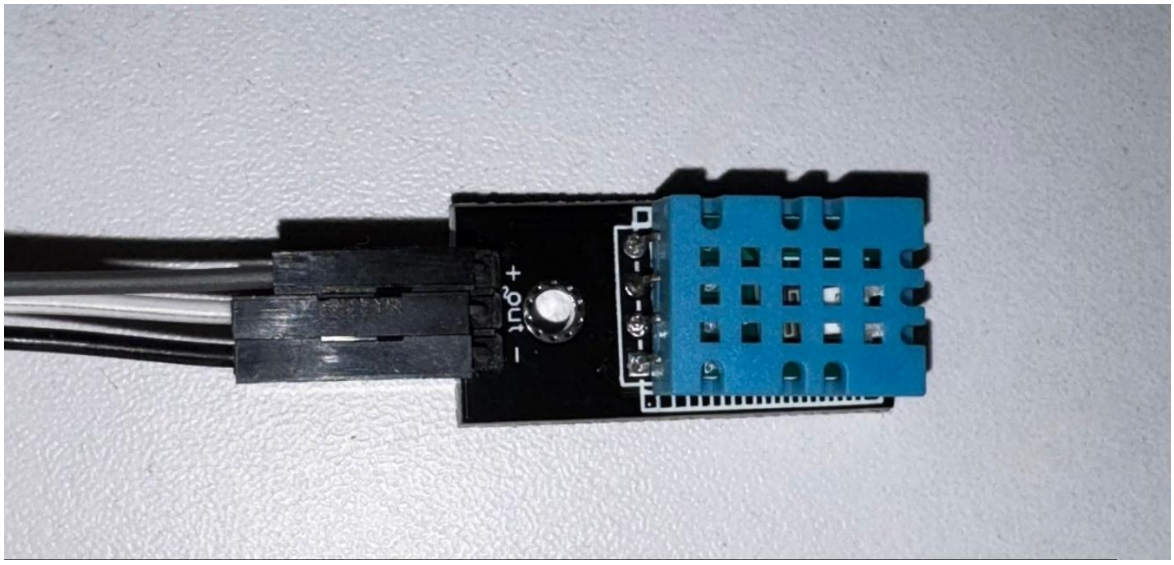
Datasheet del sensor DHT11

Parameters	Conditions	Minimum	Typical	Maximum
Humidity				
Resolution		1%RH	1%RH	1%RH
			8 Bit	
Repeatability			±1%RH	
Accuracy	25℃		±4%RH	
	0-50℃			±5%RH
Interchangeability	Fully Interchangeable			
Measurement Range	0℃	30%RH		90%RH
	25℃	20%RH		90%RH
	50℃	20%RH		80%RH
Response Time (Seconds)	1/e(63%)25℃, 1m/s Air	6 S	10 S	15 S
Hysteresis			±1%RH	
Long-Term Stability	Typical		±1%RH/year	
Temperature				
Resolution		1℃	1℃	1℃
		8 Bit	8 Bit	8 Bit
Repeatability			±1℃	
Accuracy		±1℃		±2℃
Measurement Range		0℃		50℃
Response Time (Seconds)	1/e(63%)	6 S		30 S

Nota. Especificaciones detalladas del sensor DHT11. Tomada de DHT11 Humidity & Temperature Sensor

Figura 9.

Sensor DHT11



8.2.2.2 Sensor de temperatura y humedad DHT22. El DHT22 es un sensor digital de temperatura y humedad de alta precisión, ideal para aplicaciones meteorológicas y de monitoreo ambiental, y por contraste, aprovecha completamente los 16 bits asignados a cada medición, logrando resolución de décimas de grado y punto porcentual. También el bit más significativo del byte de temperatura alto codifica el signo, permitiendo que el DHT22 reporte temperaturas bajo cero, una capacidad ausente en el DHT11.

Figura 10.

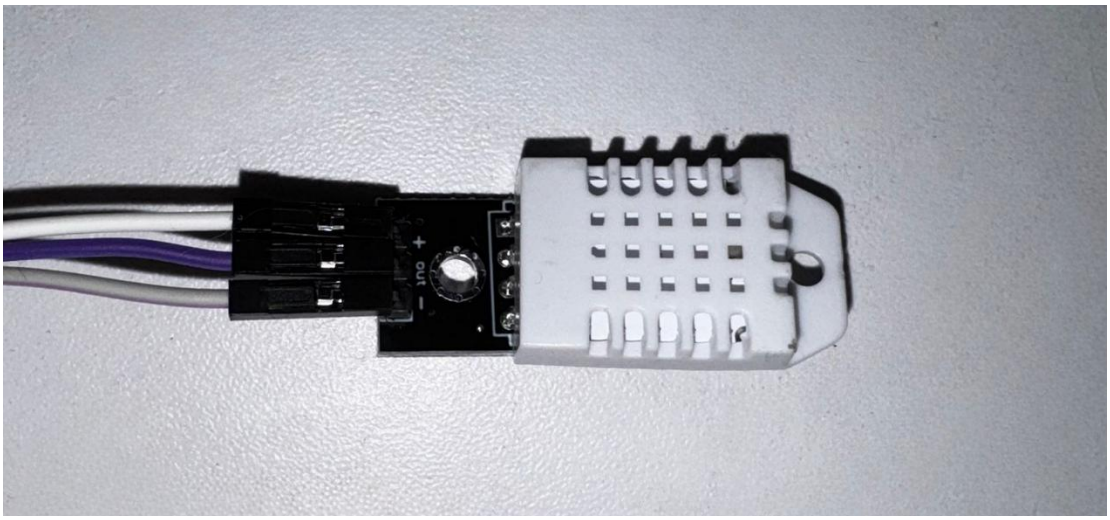
Datasheet del sensor DHT22

Model	DHT22
Power supply	3.3-6V DC
Output signal	digital signal via single-bus
Sensing element	Polymer capacitor
Operating range	humidity 0-100%RH; temperature -40~80Celsius
Accuracy	humidity +-2%RH(Max +-5%RH); temperature <+-0.5Celsius
Resolution or sensitivity	humidity 0.1%RH; temperature 0.1Celsius
Repeatability	humidity +-1%RH; temperature +-0.2Celsius
Humidity hysteresis	+-0.3%RH
Long-term Stability	+-0.5%RH/year
Sensing period	Average: 2s
Interchangeability	fully interchangeable
Dimensions	small size 14*18*5.5mm; big size 22*28*5mm

Nota. Especificaciones detalladas del sensor DHT22. Tomada de DHT22.pdf

Figura 11.

Sensor DHT22



8.2.2.3 Sensor de temperatura digital DS18B20. El DS18B20 es un sensor digital de temperatura de alta precisión que utiliza el protocolo OneWire, permitiendo conectar múltiples sensores en un solo pin, aprovechando el soporte del kernel Linux para dispositivos OneWire en lugar de implementar comunicación en espacio de usuario. El kernel expone cada sensor DS18B20 como un archivo en el sistema de archivos virtual /sys/bus/w1/devices/, permitiendo lectura mediante operaciones estándar de entrada y salida de archivos. Esta arquitectura transfiere la complejidad del protocolo OneWire al kernel, simplificando la implementación del driver.

El archivo w1_slave expuesto por el kernel contiene los datos del sensor en formato de texto, incluyendo el valor crudo de temperatura precedido por la cadena "t=". La implementación de Ds18b20Sensor analiza este formato, extrayendo el valor numérico y dividiéndolo entre 1000 para obtener grados Celsius. Esto, aunque menos eficiente que acceso directo al hardware, ofrece ventajas significativas en mantenibilidad y portabilidad.

Cada DS18B20 trae un ID único de 64 bits desde fábrica, lo que permite usar varios sensores en el mismo bus OneWire sin confusiones. Cada sensor se identifica por su dirección única, típicamente formada como "28-xxxxxxxxxxxx" donde 28 representa el código de familia DS18B20.

Figura 12.

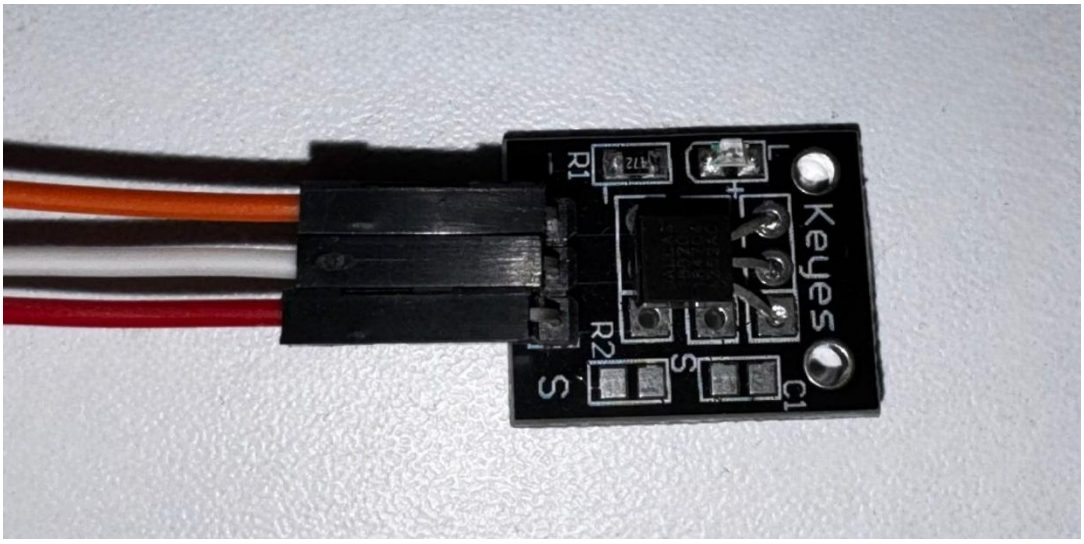
Datasheet del sensor DS18B20

PARAMETER	SYMBOL	CONDITIONS		MIN	TYP	MAX	UNITS
Supply Voltage	V_{DD}	Local power (Note 1)		+3.0		+5.5	V
Pullup Supply Voltage	V_{PU}	Parasite power	(Notes 1, 2)	+3.0		+5.5	V
		Local power		+3.0		V_{DD}	
Thermometer Error	t_{ERR}	-10°C to +85°C	(Note 3)			±0.5	°C
		-30°C to +100°C				±1	
		-55°C to +125°C				±2	
Input Logic-Low	V_{IL}	(Notes 1, 4, 5)		-0.3		+0.8	V
Input Logic-High	V_{IH}	Local power	(Notes 1,6)	+2.2		The lower of 5.5 or V_{DD} + 0.3	V
		Parasite power		+3.0			
Sink Current	I_L	$V_{I/O}$ = 0.4V		4.0			mA
Standby Current	I_{DDS}	(Notes 7, 8)			750	1000	nA
Active Current	I_{DD}	V_{DD} = 5V (Note 9)			1	1.5	mA
DQ Input Current	I_{DQ}	(Note 10)			5		µA
Drift		(Note 11)			±0.2		°C

Nota. *Especificaciones detalladas del sensor DS18B20. Tomada de DS18B20 - Programmable Resolution 1-Wire Digital Thermometer*

Figura 13.

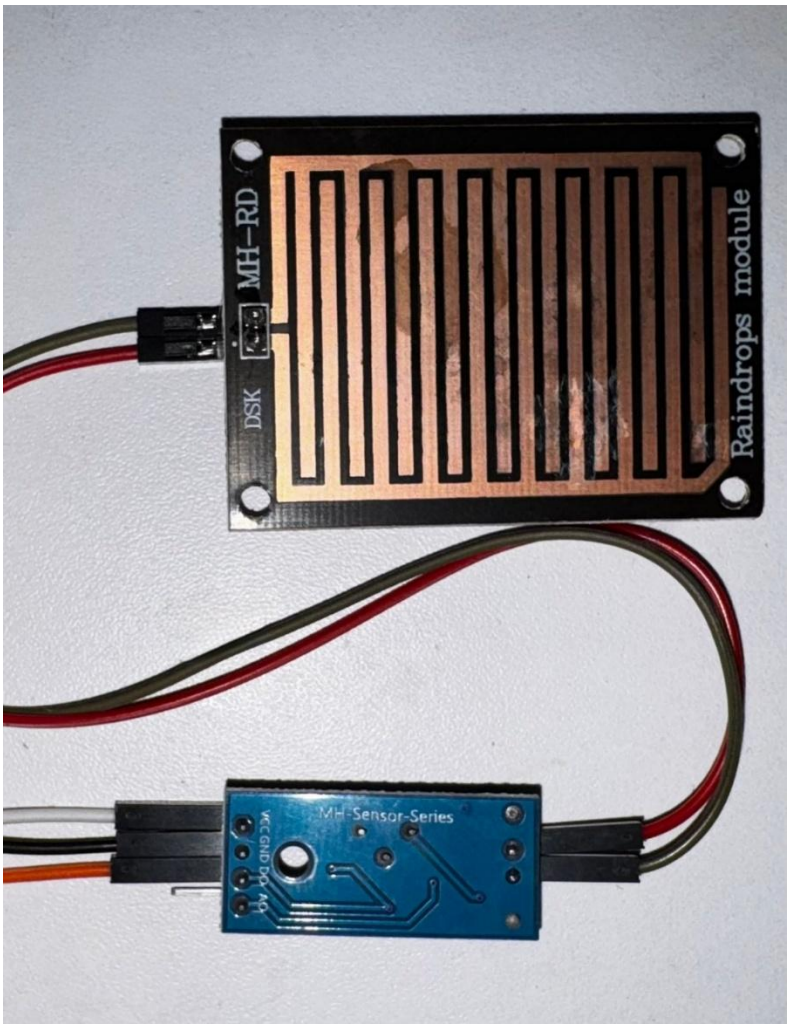
Sensor DS18B20



8.2.2.4 Sensor de lluvia digital MH-RD. El MH-RD es un sensor digital de detección de lluvia que utiliza una salida digital (DO) para indicar la presencia de agua implementa la abstracción más simple del framework con un sensor digital de dos estados. Los módulos de sensor de lluvia típicamente incorporan un comparador que convierte la resistencia variable de una placa sensor en una señal digital clara. La salida digital cambia estado cuando la resistencia cruza un umbral ajustable mediante potenciómetro, permitiendo calibración según la sensibilidad deseada.

Figura 14.

Sensor MH-RD



8.2.3 Protocolos de comunicación

Una vez definidos los sensores y su lógica de captura, el siguiente componente clave es la forma en que estos datos se transportan dentro del sistema y hacia servicios externos. Para ello, el framework debe integrar protocolos de comunicación que garanticen transmisión confiable, controlada y segura. Estos protocolos actúan como la capa que conecta los módulos de lectura de sensores con el resto de la arquitectura, permitiendo que la información fluya desde el hardware hacia los módulos de procesamiento, almacenamiento o publicación en la red.

El módulo de comunicación materializa el trait de Communicator mediante dos implementaciones. ConsoleCommunicator o comunicación por consola, la implementación más simple. Esta simplicidad no disminuye su utilidad, durante desarrollo y debugging, poder ver los datos fluyendo por el sistema en tiempo real resulta imprescindible.

MqttCommunicator opera en la capa de red y gestiona la comunicación mediante el protocolo MQTT. La librería rumqttc divide el manejo del protocolo en dos partes: el cliente, que ofrece las funciones de publicación, y la conexión, encargada del flujo de entrada y salida por la red. Esta separación exige que la conexión debe ejecutarse en un hilo independiente para mantener activo el intercambio de mensajes requerido por el protocolo y asegurar el funcionamiento correcto del cliente.

8.2.4 Formato de mensajes Smart Campus

El módulo SmartCampusFormatter, alojado dentro del módulo network, surge de la necesidad de que las aplicaciones construidas sobre Lince puedan integrarse directamente con el componente Cloud del Smart Campus UIS sin que cada desarrollador deba reimplementar la estructura del mensaje requerida por ese servicio. Esta funcionalidad se diseñó como un

componente independiente del comunicador MQTT, separando deliberadamente el problema de cómo se estructuran los datos de cómo se transmiten. El formateador no conoce ni depende de MqttCommunicator; únicamente produce una cadena de texto en formato JSON que la aplicación decide enviar por el medio que considere apropiado.

SmartCampusFormatter se construye a partir de un encabezado (SmartCampusHeader) que identifica el dispositivo emisor (deviceId) y el tópico de publicación (topic), y expone varios métodos de conversión según el origen del dato. El método desde_mapa recibe directamente el HashMap producido por SensorParser para sensores que entregan múltiples valores, como temperatura y humedad en los sensores DHT. El método desde_valor formatea un único valor escalar, caso típico del sensor DS18B20, mientras que desde_bool convierte un valor booleano, como el estado del sensor de lluvia MH-RD, a una representación numérica de 1.0 o 0.0 compatible con el esquema de métricas del Smart Campus.

8.2.5 Almacenamiento local

La comunicación suministra las lecturas en bruto desde los sensores, pero antes de enviarlas o utilizarlas es necesario trabajar con esos datos: calcular promedios, detectar cambios, filtrar valores atípicos o preparar métricas intermedias. Para esto, el módulo de almacenamiento opera como un espacio temporal donde cada lectura se guarda solo mientras se procesa.

MemoryStorage implementa el trait Storage mediante un vector en memoria que mantiene pares de timestamp y lectura de sensor. Esta implementación sacrifica deliberadamente persistencia por simplicidad, reconociendo que las necesidades de almacenamiento varían enormemente entre aplicaciones. Algunos sistemas requieren almacenamiento persistente en base de datos, otros necesitan simplemente buffer temporal para suavizar variaciones en conectividad

de red, y algunos pueden desear almacenamiento en archivos CSV para análisis posterior. Proporcionar MemoryStorage como implementación por defecto cubre el caso de uso más común mientras facilita que usuarios del framework implementen alternativas especializadas.

8.2.6 Almacenamiento persistente y reintento por desconexión

MemoryStorage, descrito en la sección de almacenamiento local, resuelve adecuadamente las necesidades de buffer temporal durante la ejecución de la aplicación, pero no sobrevive a un reinicio ni conserva datos adquiridos durante una interrupción prolongada de la red. Para escenarios donde la pérdida de lecturas resulta inaceptable, se incorporó al framework SqliteStorage, una segunda implementación del trait Storage que persiste cada lectura en una base de datos SQLite local mediante la biblioteca rusqlite, sin alterar en absoluto la interfaz que el resto del framework utiliza para guardar datos.

Cada registro almacenado por SqliteStorage incluye una columna sent que distingue mensajes pendientes de mensajes ya confirmados como enviados. Cuando MqttCommunicator reporta un error de desconexión, la aplicación puede invocar el método flush_pending, que recorre los registros pendientes en orden de inserción e intenta reenviarlos utilizando el comunicador proporcionado. Cada mensaje reenviado exitosamente se marca como enviado para no procesarse nuevamente, mientras que un fallo por desconexión detiene el ciclo de reintento, dejando los mensajes restantes para un intento posterior.

La decisión de implementar la lógica de reintento dentro del propio SqliteStorage, en lugar de extraerla a un componente independiente, responde a una evaluación deliberada durante el diseño. Una alternativa considerada consistía en crear un componente separado dedicado exclusivamente a gestionar reintentos, desacoplado del almacenamiento. Sin embargo, esta

alternativa habría introducido una capa adicional de coordinación entre dos componentes que, en la práctica, siempre operan juntos solo tiene sentido reintentar un mensaje que efectivamente se conservó en algún lugar mientras la red estuvo caída. Concentrar ambas responsabilidades en SqliteStorage mantiene el principio de responsabilidad única a nivel de la funcionalidad completa que el desarrollador necesita, sin fragmentar artificialmente un comportamiento cohesivo en múltiples piezas que de igual forma tendrían que usarse en conjunto.

8.2.7 Flujo de ejecución

El flujo de ejecución del prototipo demuestra cómo los componentes individuales del framework se integran en un sistema operativo completo que adquiere, procesa, almacena y transmite datos de sensores. Este flujo materializa el pipeline de datos definido en la arquitectura, mostrando que las abstracciones diseñadas permiten componer un sistema funcional. La secuencia de operaciones refleja patrones comunes en aplicaciones IoT reales, donde múltiples sensores deben monitorearse continuamente y sus datos enviarse a sistemas externos para análisis o visualización.

El ciclo comienza con la fase de inicialización, donde el sistema configura cada sensor conectado al hardware. Esta inicialización involucra la creación de instancias de los sensores especificando los pines GPIO correspondientes, un proceso que puede fallar si el hardware no está presente, los pines están incorrectamente configurados, o ya están en uso por otro proceso. El sistema maneja estos fallos de inicialización reportándolos apropiadamente sin comprometer la estabilidad general, permitiendo que la aplicación continúe operando con los sensores que sí se inicializaron exitosamente. Esta capacidad resulta esencial en despliegues reales donde sensores individuales pueden fallar sin que el sistema completo deba detenerse.

Mientras se inicializan los sensores, el sistema también prepara los componentes que manejarán los datos obtenidos. El módulo de almacenamiento crea un buffer temporal donde cada lectura se guarda solo mientras se procesa, evitando que la adquisición dependa del estado de la red. Al mismo tiempo, el comunicador MQTT establece la conexión con el broker y activa el thread encargado de enviar los datos. Esta separación permite que, si la red falla, el sistema continúe leyendo sensores sin interrupciones, manteniendo los valores en el buffer hasta que sea posible transmitirlos de nuevo.

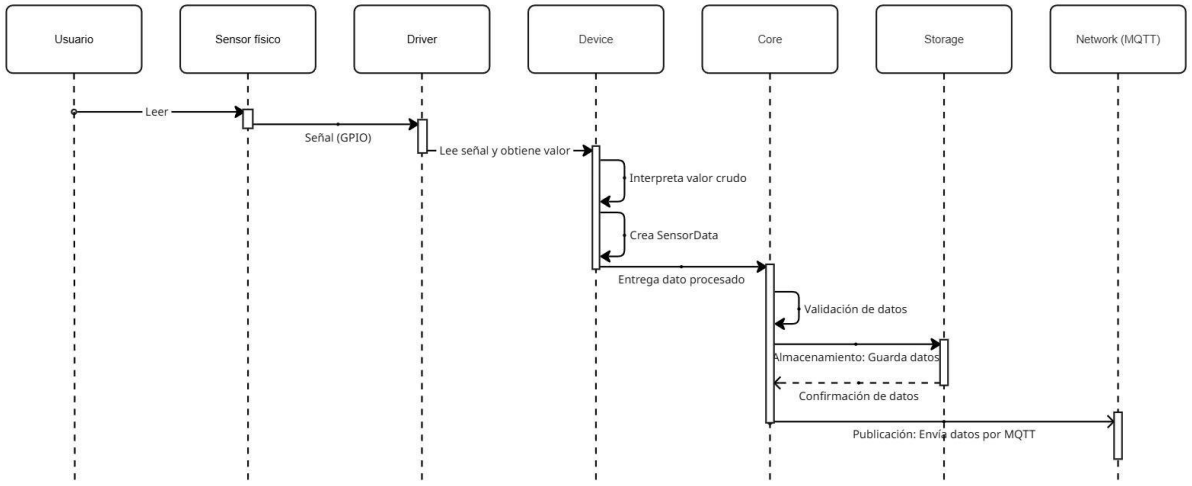
Tras completar la inicialización, el sistema entra en su ciclo principal de adquisición de datos. Cada iteración del ciclo procesa secuencialmente los sensores configurados, respetando los requisitos específicos de cada dispositivo. La lectura de sensores DHT precede un período de estabilización que permite al elemento sensor recuperarse antes de la siguiente medición, implementado mediante delays explícitos entre iteraciones.

Las lecturas exitosas fluyen inmediatamente hacia el subsistema de almacenamiento, que marca cada dato con un timestamp antes de agregarlo al buffer temporal. Este timestamping automático permite reconstruir series temporales posteriores sin requerir que los sensores individuales gestionen aspectos de temporalidad. El almacenamiento actúa como primer nivel de persistencia, garantizando que los datos no se pierdan si subsecuentes operaciones fallan.

Tras almacenar cada lectura, el sistema intenta enviarla usando el comunicador MQTT. Para esto convierte el dato del sensor en un formato que pueda transmitirse por red y lo publica en el tópico configurado. El cliente MQTT se encarga internamente de los detalles del protocolo, como confirmar que el mensaje llegó o reconectarse si la conexión se cae. Desde el punto de vista del framework, enviar un dato es simplemente una operación que puede salir bien o fallar, y en caso de error este se registra sin detener el resto del ciclo de lectura.

Figura 15.

Diagrama de secuencia



Nota. Diagrama de secuencia de una lectura típica de sensor implementando el framework.

El flujo completo demuestra cómo el sistema mantiene operación continua a través de múltiples ciclos, con cada componente ejecutando su responsabilidad específica sin conocimiento innecesario de los demás. Los sensores se concentran en adquirir datos correctamente, el almacenamiento gestiona la persistencia temporal, y los comunicadores manejan la transmisión de red. Esta separación se basa en la arquitectura basada en traits, donde cada componente interactúa con otros únicamente a través de interfaces bien definidas. El resultado es un sistema donde componentes individuales pueden fallar, recuperarse o reemplazarse sin afectar la operación de otros subsistemas.

La implementación del prototipo valida que los principios arquitectónicos del framework pueden materializarse en código funcional sobre hardware real. Los componentes desarrollados demuestran que abstracciones bien diseñadas no impiden acceso eficiente a recursos de bajo nivel, y que la modularidad no tiene que sacrificarse por rendimiento. El prototipo sirve como fundamento sólido para expansiones futuras, habiendo establecido patrones y convenciones que

guiarán desarrollos subsecuentes mientras mantiene flexibilidad para evolucionar según necesidades emergentes.

8.2.8 Compilación cruzada

El flujo de desarrollo descrito hasta este punto asume que el código se compila directamente sobre la Raspberry Pi, lo cual resulta adecuado durante la fase de prototipado pero impone limitaciones evidentes en términos de tiempo de compilación y comodidad de desarrollo. La arquitectura ARM de la Raspberry Pi, sumada a sus recursos de cómputo limitados frente a una estación de trabajo convencional, hace que compilar proyectos Rust de cierto tamaño directamente sobre el dispositivo gateway resulte considerablemente más lento que hacerlo en una máquina host x86_64. Para resolver esta limitación se incorporó un mecanismo de compilación cruzada (cross-compilation), que permite generar el binario ejecutable en la máquina de desarrollo y transferirlo posteriormente al dispositivo gateway, reservando los recursos de la Raspberry Pi exclusivamente para la ejecución de la aplicación.

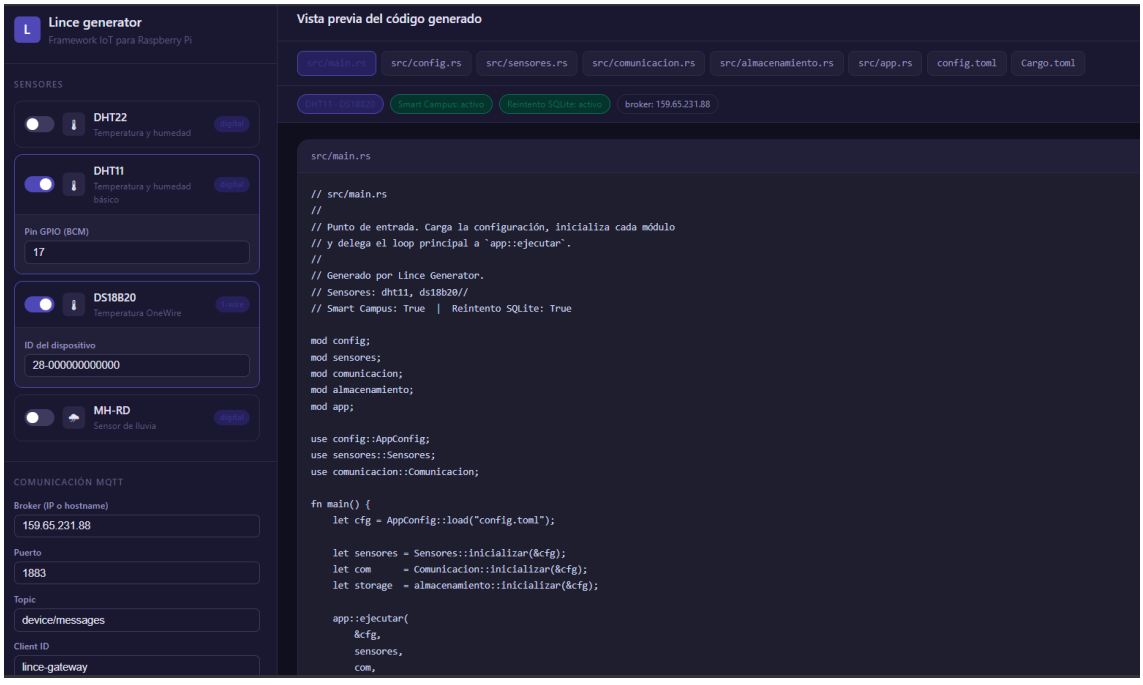
La compilación cruzada en Rust se apoya en el ecosistema provisto por rustup, que permite instalar el objetivo de compilación aarch64-unknown-linux-gnu, correspondiente a la arquitectura de 64 bits utilizada por la Raspberry Pi 4 bajo Raspberry Pi OS. Una vez instalado este objetivo y el enlazador cruzado correspondiente Cargo puede generar un binario nativo para la arquitectura ARM desde la máquina de desarrollo. El resultado de este proceso es un ejecutable listo para transferirse al dispositivo gateway sin requerir herramientas de compilación adicionales en este último.

8.2.9 Asistente de generación simple

Aunque la compilación cruzada resuelve el problema del tiempo de compilación, persiste una barrera adicional para desarrolladores que se aproximan por primera vez al framework: escribir desde cero un proyecto Cargo que integre los sensores deseados, configure el comunicador MQTT, defina opcionalmente el formato Smart Campus y, si la aplicación lo requiere, incorpore almacenamiento con reintento por desconexión. Esta tarea, aunque mecánica, exige conocer la combinación correcta de módulos, traits e importaciones del framework, lo que puede ralentizar la adopción de Lince por parte de equipos sin experiencia previa en el ecosistema Rust. Para reducir esta barrera de entrada se desarrolló un asistente de generación simple que se puede encontrar en LinceGenerator, una herramienta web que produce el código fuente de una aplicación Lince a partir de una configuración seleccionada por el usuario.

Figura 16.

Asistente de generación



Nota. Asistente de generación simple del framework.

El asistente se implementó como una aplicación Flask que expone una interfaz donde el desarrollador selecciona, mediante controles interactivos, los sensores a utilizar junto con sus parámetros de hardware (pin GPIO o identificador OneWire según corresponda), los datos de conexión al broker MQTT y, de manera independiente y seleccionable, dos funcionalidades opcionales; el formato Smart Campus y el reintento por desconexión mediante SQLite. Esta selección independiente refleja una decisión de diseño deliberada, ya que no todas las aplicaciones construidas sobre Lince requieren publicar en formato Smart Campus ni todas necesitan persistencia local, por lo que forzar ambas funcionalidades habría introducido dependencias y complejidad innecesarias en proyectos simples.

Internamente, la generación de código se apoya en plantillas Jinja2, motor de plantillas estándar del ecosistema Python que permite definir el esqueleto de cada archivo Rust con bloques condicionales y de repetición que se resuelven según la configuración recibida. En lugar de producir un único archivo main.rs con toda la lógica concentrada, el asistente genera un proyecto organizado en módulos independientes (config.rs, sensores.rs, comunicacion.rs, almacenamiento.rs cuando aplica, y app.rs), reflejando en el código generado la misma separación de responsabilidades que caracteriza al framework. Junto con el código fuente, el asistente produce un archivo config.toml con los parámetros de la aplicación, el Cargo.toml con las dependencias correspondientes a las funcionalidades seleccionadas, y el script de despliegue descrito en la sección anterior, empaquetando todo en un archivo comprimido listo para compilar.

El asistente de generación no reemplaza ni oculta el framework: el código producido es Rust ordinario que utiliza directamente los traits y módulos de Lince, por lo que el desarrollador puede inspeccionarlo, modificarlo o extenderlo libremente una vez generado. Su propósito es exclusivamente acelerar el punto de partida, entregando una base funcional y correctamente

estructurada que de otra forma requeriría escribirse manualmente. Esta herramienta complementa así los mecanismos de extensibilidad ya descritos en este capítulo, ofreciendo una vía adicional, más accesible, para que nuevos desarrolladores comiencen a construir aplicaciones sobre el framework.

8.2.10 Repositorio de aplicaciones de ejemplo

Si bien la documentación de referencia de cada módulo describe el comportamiento individual de sensores, almacenamiento y comunicadores, comprender cómo estos componentes se combinan en una aplicación completa requiere ejemplos que un desarrollador pueda ejecutar, inspeccionar y modificar directamente. Con este propósito se construyó un repositorio centralizado de aplicaciones de ejemplo, que documenta distintas combinaciones de funcionalidades del framework mediante proyectos Rust completos, listos para compilar o, alternativamente, listos para desplegar directamente a partir de binarios precompilados mediante el mecanismo de compilación cruzada descrito anteriormente.

El repositorio organiza las aplicaciones en directorios independientes, cada uno orientado a validar una característica específica del framework. Entre ellos se incluyen una aplicación de monitoreo de temperatura mediante un sensor DS18B20, que procesa las lecturas por lotes, calcula estadísticas (promedio, mínimo y máximo) y publica los resultados utilizando SmartCampusFormatter y MqttCommunicator; una aplicación de alertas de temperatura con un sensor DHT11, que genera publicaciones únicamente cuando la medición cruza un umbral definido y un monitor de lluvia basado en un sensor digital MH-RD, que publica únicamente cuando el estado del sensor cambia. Cada directorio incluye su propio Cargo.toml, un archivo config.toml

de ejemplo y un README con las instrucciones de compilación, configuración y ejecución correspondientes.

Junto al código fuente de cada ejemplo, el repositorio incluye los binarios ya compilados para la arquitectura aarch64-unknown-linux-gnu, generados mediante el mismo flujo de compilación cruzada documentado en este capítulo. Esto permite que un usuario interesado únicamente en probar el comportamiento del framework pueda descargar un binario y ejecutarlo directamente en su Raspberry Pi, sin necesidad de instalar el *toolchain* de Rust ni configurar el objetivo de compilación cruzada, reduciendo así la barrera de entrada para quienes evalúan Lince por primera vez.

La implementación completa de las aplicaciones de ejemplo se encuentra disponible en el siguiente repositorio LinceEjemplos. Este repositorio permite a cualquier desarrollador descargar, compilar y ejecutar las aplicaciones de ejemplo, o desplegar directamente los binarios precompilados en un dispositivo gateway.

9 Validación del framework

Esta fase corresponde a las actividades de validación funcional, pruebas en Raspberry Pi 4 y recopilación de métricas de rendimiento, cuyo propósito fue validar el funcionamiento de los componentes funcionales del framework en un entorno real. La validación del framework constituye el proceso mediante el cual se verifica que los componentes desarrollados cumplen con los requerimientos establecidos anteriormente mencionados y operan correctamente en condiciones reales. Esta fase no busca únicamente demostrar funcionalidad básica, sino validar que las decisiones arquitectónicas tomadas permiten construir aplicaciones IoT robustas, modulares y eficientes sobre dispositivos gateway con recursos limitados.

La validación se centra en demostrar cuatro aspectos fundamentales del framework. Primero, que la arquitectura modular permite integrar sensores físicos reales sin necesidad de modificar el núcleo del sistema, validando así el diseño basado en traits y la separación de responsabilidades. En este sentido, la implementación de los sensores mostrada anteriormente no es únicamente una tarea de prototipado, sino el inicio de este proceso, cada sensor incorporado al framework es una prueba de que la arquitectura modular y el sistema de traits funcionan correctamente ante distintos paradigmas de hardware. Segundo, que los diferentes módulos del framework (drivers de hardware, almacenamiento temporal y comunicación de red) operan de forma coherente e integrada, intercambiando datos correctamente a través de las interfaces definidas. Tercero, que el flujo completo de adquisición, procesamiento y transmisión de datos puede ejecutarse de manera continua en un dispositivo gateway real, específicamente una Raspberry Pi 4, sin comprometer la estabilidad del sistema. Cuarto, que el framework mantiene eficiencia de recursos validando la elección de Rust como lenguaje de implementación.

9.1 Escenario de validación

El escenario de validación reproduce un caso de uso real de monitoreo ambiental distribuido, contexto común en sistemas IoT desplegados en entornos como el Smart Campus UIS. El hardware utilizado consiste en una Raspberry Pi 4 Model B con 4GB de RAM ejecutando Raspberry Pi OS Lite (64-bit, basado en Debian Bookworm), sistema operativo seleccionado por su balance entre funcionalidad y consumo de recursos. Al dispositivo se conectaron cuatro sensores representativos de diferentes paradigmas de comunicación: un DHT22 para medición de temperatura y humedad mediante protocolo propietario con timing crítico, un DS18B20 para medición de temperatura mediante protocolo OneWire gestionado por el kernel, un DHT11 como

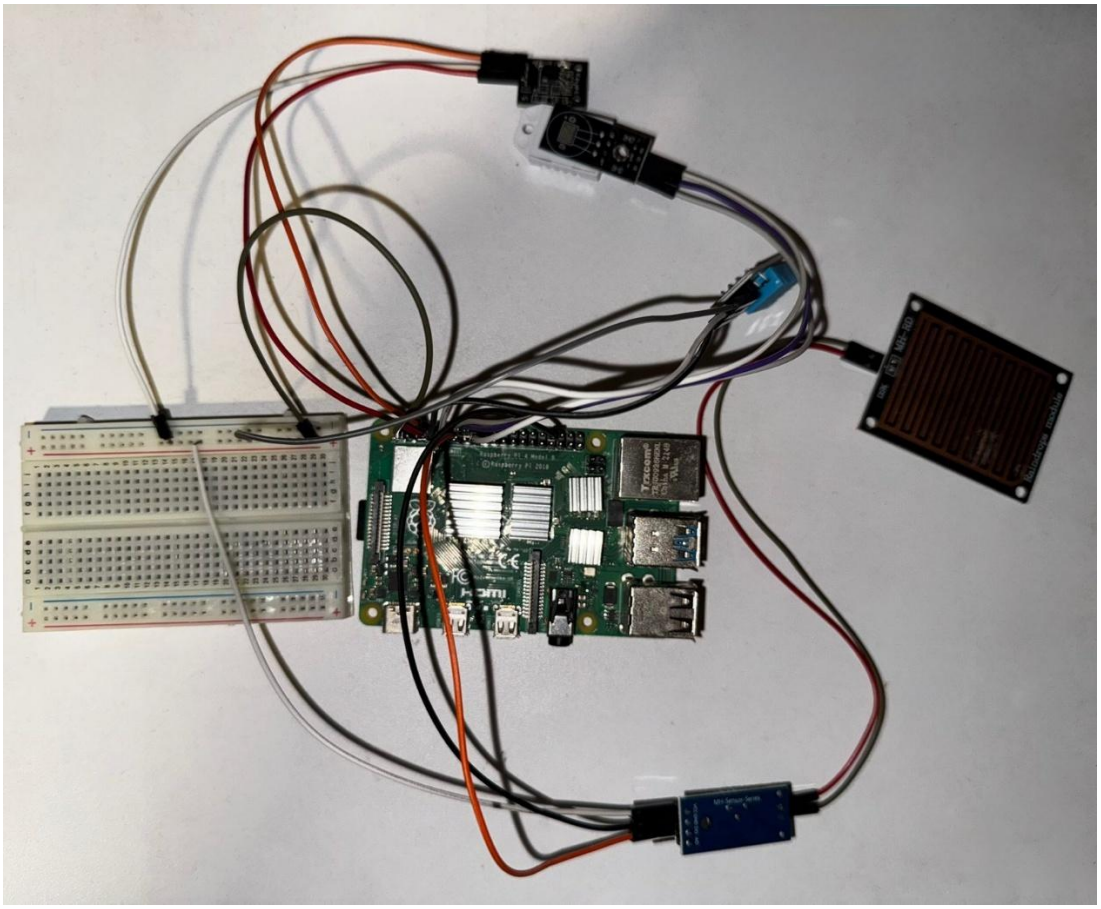
sensor de menor precisión pero mayor disponibilidad, y un MH-RD para detección digital simple de presencia de lluvia.

La implementación completa del escenario de validación, se encuentra disponible en el siguiente repositorio: Saiduts/validacion_lince: Validacion de funcionalidades del framework lince

Este repositorio permite reproducir las pruebas realizadas y verificar el comportamiento del framework en un entorno real.

Figura 17.

Montaje con el hardware utilizado en la validación

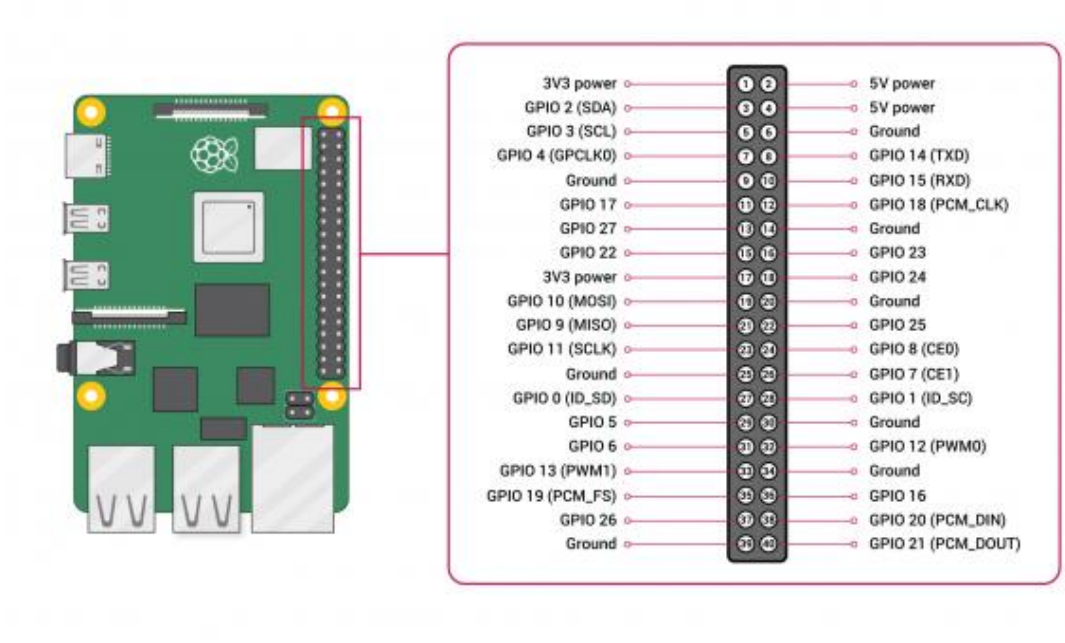


La configuración de red involucra un broker MQTT Mosquitto versión 2.0.11 ejecutándose localmente en la misma Raspberry Pi, eliminando variables de latencia de red externa y

permitiendo validación controlada de la comunicación. Los sensores se conectaron a pines GPIO específicos seleccionados para evitar conflictos con interfaces reservadas: GPIO 23 para el DHT22, GPIO 4 para el DS18B20, GPIO 17 para el DHT11 y GPIO 27 para el MH-RD. Esta distribución respeta las restricciones eléctricas de la Raspberry Pi.

Figura 18.

Pinout de la raspberry pi 4



Nota. Diagrama de pines GPIO de la Raspberry Pi tomado de Introduction to the Raspberry Pi GPIO and Physical Computing - SparkFun Learn

El entorno de software incluye Rust versión 1.75.0 instalado mediante rustup, el gestor de paquetes Cargo para compilación y gestión de dependencias, y las bibliotecas rppal versión 0.14 para acceso GPIO directo y rumqttc versión 0.23 para comunicación MQTT. El kernel Linux se

configuró para habilitar el soporte OneWire necesario para el DS18B20, activando los módulos w1-gpio y w1-therm mediante el archivo /boot/config.txt.

9.1.1 Metodología de validación

La metodología de validación se estructuró en tres fases complementarias que evalúan progresivamente la funcionalidad del framework, desde componentes individuales hasta el sistema integrado completo. Esta aproximación incremental permite identificar problemas específicos en cada capa antes de evaluar interacciones complejas.

9.1.1.1 Validación de drivers y sensores. La primera fase verificó que cada sensor puede inicializarse correctamente y producir lecturas válidas de forma aislada. Para cada dispositivo se implementó un programa de prueba mínimo que utiliza únicamente el driver correspondiente y el módulo core del framework, sin involucrar almacenamiento ni comunicación. Este aislamiento garantiza que cualquier problema detectado corresponde específicamente al driver evaluado y no a interacciones con otros componentes.

La validación de los sensores DHT se enfocó en verificar el timing crítico del protocolo propietario, aspecto que representa el mayor desafío técnico en su implementación. Se ejecutaron ciclos de lectura continua con intervalos de 10 segundos. Las lecturas se validaron contra rangos físicamente posibles: temperatura entre -40°C y 80°C para DHT22, 0°C a 50°C para DHT11, y humedad relativa entre 0% y 100% para ambos.

El DS18B20 requirió validación diferente por utilizar la interfaz del kernel en lugar de acceso GPIO directo. Se verificó la correcta detección del sensor mediante la presencia de su directorio único en /sys/bus/w1/devices/, la lectura correcta del identificador de 64 bits, y la

capacidad de obtener mediciones de temperatura parseando el archivo w1_slave. La validación incluyó verificar que el sistema maneja apropiadamente errores como sensor desconectado o archivo no disponible.

El MH-RD, siendo un sensor digital binario, se validó verificando transiciones de estado al simular presencia y ausencia de agua en su placa sensora. Se comprobó que el driver detecta correctamente cambios de estado y que la configuración del pin como entrada con pull-up interno funciona según lo esperado.

9.1.1.2 Validación de comunicación. La tercera fase se enfocó específicamente en la comunicación MQTT, componente crítico que permite la integración con sistemas externos. Se validó que el comunicador establece conexión correctamente con el broker, publica mensajes en los topics configurados respetando el formato JSON definido y maneja situaciones de error como broker no disponible o red desconectada.

Se verificó que el sistema recupera automáticamente la conexión MQTT después de interrupciones temporales de red, característica implementada mediante la funcionalidad de reconexión automática de la biblioteca rumqttc. Durante las pruebas se simularon desconexiones deteniendo temporalmente el broker Mosquitto, verificando que el framework detecta la desconexión, continúa operando localmente y reestablece la comunicación automáticamente cuando el broker vuelve a estar disponible.

9.1.1.3 Validación del flujo integrado. La segunda fase evaluó la operación del framework como sistema completo, integrando sensores, almacenamiento temporal y comunicación MQTT en una aplicación funcional. Se implementó un programa que adquirió datos de todos los sensores disponibles simultáneamente, almacenándolos temporalmente en MemoryStorage y publicándolos en un topic MQTT.

El ciclo de ejecución se configuró con un intervalo de muestreo de 10 segundos, permitiendo que los sensores DHT se estabilicen entre lecturas mientras mantiene frecuencia suficiente para detectar cambios ambientales. Esta frecuencia también facilita observación del comportamiento del sistema durante períodos de prueba razonables sin generar volúmenes excesivos de datos.

Se verificó que el sistema maneja correctamente situaciones de error parcial, como la falla temporal de un sensor individual. El framework debe continuar operando con los sensores funcionales disponibles, registrando apropiadamente los errores sin detener la ejecución completa. Esta capacidad resulta crítica en despliegues reales donde sensores individuales pueden fallar sin comprometer toda la infraestructura de monitoreo.

La validación de almacenamiento temporal verificó que MemoryStorage mantiene el historial de lecturas durante la ejecución y permite consultar los datos de este.

9.1.1.4 Criterios de éxito. La validación se consideró exitosa basándose en criterios objetivos y medibles que reflejan los requerimientos funcionales y no funcionales establecidos. Estos criterios proporcionan una evaluación cuantitativa del desempeño del framework y permiten comparaciones futuras con versiones mejoradas o alternativas.

- **Estabilidad de lectura de sensores:** Los drivers deben producir lecturas válidas bajo condiciones normales de operación. Las lecturas deben encontrarse dentro de rangos físicamente posibles para cada sensor y mostrar coherencia temporal, es decir, variaciones graduales en lugar de cambios abruptos no justificados. Este criterio valida que la implementación de los protocolos de comunicación con sensores respeta los requisitos de timing y manejo de señales.
- **Integridad del flujo de datos:** El ciclo completo de adquisición, almacenamiento y transmisión debe ejecutarse sin errores críticos durante períodos prolongados. Una sesión de monitoreo debe completarse sin fallos que detengan la aplicación. Los errores recuperables, como lecturas fallidas ocasionales de sensores, deben manejarse apropiadamente sin comprometer la operación general.
- **Modularidad verificable:** Debe ser posible agregar, remover o reemplazar sensores modificando únicamente la configuración de la aplicación, sin alteraciones en el código del framework. Esta capacidad demuestra que la arquitectura basada en traits cumple su objetivo de extensibilidad. La validación incluyó compilar y ejecutar variantes de la aplicación de prueba con diferentes combinaciones de sensores, verificando que cada configuración funciona correctamente.
- **Tolerancia a fallos:** El sistema debe continuar operando cuando componentes individuales fallan. La desconexión física de un sensor no debe afectar la lectura de otros sensores. La indisponibilidad temporal del broker MQTT no debe detener la adquisición de datos. Esta característica resulta esencial en despliegues reales donde fallos parciales son inevitables pero no deben comprometer toda la infraestructura.

- **Eficiencia de recursos:** El framework debe operar dentro de los recursos disponibles en la Raspberry Pi 4 sin saturar CPU, memoria o ancho de banda de red. El consumo de memoria debe mantenerse estable o crecer linealmente con el historial de datos almacenados. El uso de CPU no debe impedir que el sistema realice otras tareas concurrentemente.

9.1.2 Resultados obtenidos

La ejecución de las pruebas de validación produjo resultados que confirman el cumplimiento de los objetivos establecidos, demostrando que el framework puede soportar aplicaciones IoT reales en dispositivos gateway con recursos limitados.

9.1.2.1 Configuración de las pruebas. La validación se realizó mediante una sesión de monitoreo continuo de 15 minutos con intervalos de muestreo de 10 segundos, generando 90 ciclos de lectura completos. Esta configuración permite observar el comportamiento del sistema durante un período representativo mientras genera datos suficientes para análisis estadístico significativo. Las métricas de rendimiento se recopilaron mediante dos métodos complementarios: el crate sysinfo de Rust para monitoreo interno del proceso durante toda la ejecución, y la herramienta time de Linux para validación externa de los recursos consumidos.

9.1.2.2 Desempeño de sensores. Los sensores DHT22 y DHT11 alcanzaron tasas de éxito de 95.56% y 97.78% respectivamente. Las lecturas fallidas se atribuyen principalmente a las limitaciones de timing en la Raspberry Pi 4, donde el sistema operativo Linux introduce interrupciones impredecibles que afectan la lectura del protocolo DHT, el cual requiere precisión de microsegundos.

El protocolo DHT requiere precisión temporal en el orden de microsegundos para detectar correctamente la modulación de ancho de pulso que codifica los bits de datos. En un sistema operativo de tiempo real, el software puede garantizar respuestas dentro de ventanas temporales estrictas. Sin embargo, Linux en Raspberry Pi 4 es un sistema operativo multitarea preemptivo donde el kernel puede interrumpir procesos de usuario en cualquier momento para atender interrupciones de hardware, cambios de contexto o tareas del sistema.

Cuando el framework intenta leer el sensor DHT, debe medir con precisión la duración de pulsos que oscilan entre 26 y 70 microsegundos. Si durante esta ventana crítica el kernel interrumpe el proceso para atender otra tarea, aunque sea brevemente, el timing se pierde irremediablemente. Los errores reportados se clasificaron principalmente como "Timeout" e "InvalidData", indicando que el sistema no pudo detectar las transiciones de señal dentro de los límites temporales esperados o que los bits recibidos no pasaron la verificación de checksum.

Para mitigar este inconveniente, se implementó un mecanismo de reintentos que permite al framework intentar nuevamente la lectura cuando falla, mejorando significativamente la confiabilidad del sistema sin necesidad de hardware adicional.

El DS18B20 demostró una tasa de éxito del 95.56%. Los cuatro fallos registrados (iteraciones 11-14) ocurrieron consecutivamente, comportamiento que no corresponde a errores aleatorios de timing sino a una condición específica. Durante la validación, se desconectó

deliberadamente el sensor DS18B20 de sus pines GPIO durante estas iteraciones para verificar la capacidad de tolerancia a fallos del framework. El sistema detectó correctamente la ausencia del sensor, reportó los errores apropiadamente mediante el tipo SensorError::InvalidData, y continuó operando normalmente con los demás sensores sin detener la ejecución completa.

Esta prueba de desconexión física valida uno de los requerimientos funcionales críticos del framework, el manejo robusto de errores y tolerancia a fallos. En despliegues reales, sensores pueden desconectarse accidentalmente, sufrir daños físicos o fallar temporalmente. El framework debe permitir que la aplicación continúe operando con los recursos disponibles en lugar de fallar completamente. La recuperación automática después de reconectar el sensor demuestra que el framework maneja apropiadamente el ciclo completo de fallo y recuperación sin requerir reinicio de la aplicación.

El sensor de lluvia MH-RD funcionó con confiabilidad del 100%, completando exitosamente las 90 lecturas sin ningún error. Este resultado era esperado dado que la lectura de un pin digital simple mediante GPIO no presenta los desafíos de timing de protocolos más complejos. La señal digital del MH-RD cambia entre estados alto y bajo según la presencia de humedad en su placa sensora, operación que el sistema operativo puede manejar confiablemente sin requisitos temporales estrictos.

Tabla 10.

Resultados de confiabilidad de sensores durante validación de 15 minutos

Sensor	Lecturas totales	Lecturas exitosas	Tasa de éxito	Causa principal de fallos
DHT22	90	86	95.56%	Interferencias del SO en timing crítico
DHT11	90	88	97.78%	Interferencias del SO en timing crítico
DS18B20	90	86	95.56%	Desconexión física intencional

MH-RD	90	90	100%	N/A

Nota. La tabla presenta los resultados de pruebas de lectura realizadas sobre distintos sensores, mostrando el número total de lecturas, lecturas exitosas, la tasa de éxito y la causa principal de los fallos identificados durante la validación del sistema.

9.1.2.3 Comportamiento del sistema integrado. El framework completó exitosamente los 90 ciclos de muestreo durante los 15 minutos de prueba, manejando correctamente 350 lecturas exitosas de un total de 360 intentos. El sistema demostró capacidad de recuperación automática ante fallos: cuando se desconectó el DS18B20 durante la ejecución, el framework continuó operando normalmente con los demás sensores. De manera similar, cuando se detuvo el broker MQTT Mosquitto y posteriormente se reinició, el sistema detectó la desconexión, intentó reconectar automáticamente y restauró la comunicación sin intervención manual.

El almacenamiento temporal en memoria acumuló las 350 lecturas exitosas sin problemas, permitiendo consultar el historial completo durante toda la sesión de prueba.

9.1.2.4 Consumo de recursos. Para medir el consumo de recursos se utilizaron dos herramientas complementarias: el comando time de Linux y la biblioteca sysinfo de Rust. Mientras que time proporciona métricas generales del proceso desde el inicio hasta el final de la ejecución, sysinfo permite monitorear el comportamiento del sistema en tiempo real durante toda la prueba.

Uso de CPU: El framework mostró un uso promedio de CPU del 3.01%, con picos ocasionales de hasta 112.89% durante las lecturas de sensores DHT que requieren espera activa. Pero en general, el uso sostenido de CPU se mantuvo bajo, lo que confirma que la ejecución del framework es eficiente y permite operar simultáneamente otros procesos en el dispositivo sin riesgo de saturación.

Memoria RAM: El consumo promedio de memoria fue de 6.71 MB (6,874 KB), con un máximo de 6.77 MB. Esta medición proviene de la Resident Set Size (RSS), que indica la memoria física real que el proceso está usando. El comando time reportó un máximo de 213,860 KB (aproximadamente 209 MB) de memoria virtual, que incluye todo el espacio de direcciones del proceso. La diferencia entre ambos valores es normal: la memoria virtual incluye código, bibliotecas compartidas y espacio reservado pero no utilizado, mientras que la RSS muestra únicamente la memoria física realmente ocupada, que es la métrica más relevante para evaluar la eficiencia del framework.

Almacenamiento: El sistema guardó 360 lecturas en total, generando archivos de log y métricas que ocuparon aproximadamente 120 KB durante la sesión de 15 minutos. Este tamaño incluye los registros detallados de eventos y las métricas del sistema.

Tabla 11.

Consumo de recursos durante operación normal

Recurso	Valor promedio	Valor máximo	Observaciones
CPU	3.01%	112.89%	Picos durante lectura de DHT
RAM	6.71 MB	6.77 MB	Memoria física utilizada
Memoria Virtual	206.09 MB	209 MB	Espacio total reservado
Almacenamiento	-	150 KB	Logs y métricas de 15 min

Nota. La tabla muestra el uso de recursos del sistema durante la ejecución del framework, indicando valores promedio y máximos de CPU, memoria y almacenamiento.

9.1.3 Limitaciones observadas

La validación también reveló limitaciones inherentes al diseño actual del framework que deben considerarse al evaluar su aplicabilidad a diferentes escenarios y que representan oportunidades de mejora en futuras iteraciones.

9.1.3.1 Tiempo de lectura de sensores. Algunos sensores, particularmente los DHT, requieren períodos de estabilización relativamente largos entre lecturas. El protocolo DHT especifica un mínimo de dos segundos entre mediciones consecutivas para permitir que el elemento sensor se recupere. Esta restricción limita la frecuencia máxima de muestreo y puede resultar inadecuada para aplicaciones que requieren respuesta rápida a cambios ambientales.

El framework no oculta ni abstrae estas limitaciones de hardware, decisión deliberada que prioriza transparencia sobre conveniencia. Desarrolladores deben conocer y respetar las restricciones de los sensores que utilizan, configurando intervalos de muestreo apropiados. La documentación del framework proporciona guías claras sobre los requisitos de cada sensor soportado.

9.1.3.2 Gestión de conectividad de red. El comunicador MQTT implementa reconexión automática después de desconexiones, pero el escenario de validación documentado en el capítulo anterior operó únicamente con MemoryStorage, sin manejar la retransmisión de datos acumulados durante períodos de indisponibilidad de red. Bajo esa configuración, los datos adquiridos mientras el broker estaba inalcanzable se perdían una vez que el buffer temporal en memoria se llenaba o cuando la aplicación terminaba.

Esta limitación fue abordada incorporando al framework SqliteStorage, una implementación del trait Storage que persiste cada lectura en una base de datos SQLite local marcándola como pendiente de envío. Cuando el comunicador MQTT reporta una desconexión, la aplicación puede invocar flush_pending para reintentar el envío de los mensajes acumulados, los cuales se marcan como enviados únicamente tras una confirmación exitosa. El uso de este mecanismo es opcional, ya que el desarrollador decide si su aplicación lo requiere, y el comportamiento por defecto del framework permanece sin reintentos para no introducir persistencia ni dependencias adicionales en proyectos que no las necesitan. Esta extensión, descrita en detalle en la sección de funcionalidades opcionales del framework, demuestra que el diseño modular permite incorporar almacenamiento persistente sin modificar los componentes de adquisición o comunicación existentes.

9.1.3.3 Ausencia de configuración dinámica. Si bien el framework no incluye configuración dinámica porque su diseño asume que cada proyecto definirá sus propios sensores, intervalos y parámetros según sus necesidades particulares, el requerimiento RF08 fue satisfecho mediante la provisión de una plantilla del framework que lee parámetros externos desde un archivo config.toml en tiempo de ejecución. Esta plantilla, disponible en Lince template, permite al desarrollador ajustar el pin del sensor, la dirección del broker MQTT, el topic y el identificador del cliente sin necesidad de recompilar el núcleo del framework, demostrando que la configuración externa es viable y es un punto de partida para nuevas implementaciones.

9.1.3.4 Cobertura limitada de sensores. El framework soporta actualmente cuatro tipos de sensores específicos seleccionados por su representatividad y disponibilidad. Aunque la arquitectura modular facilita la incorporación de nuevos dispositivos, esta cobertura inicial limita la aplicabilidad inmediata a escenarios que requieren otros tipos de sensores como medidores de calidad de aire, sensores de luminosidad o actuadores como relés o servomotores.

Esta limitación representa una característica transitoria más que un defecto fundamental. El valor del framework radica precisamente en proporcionar la infraestructura y patrones que permiten a desarrolladores extenderlo según sus necesidades específicas, como se documenta extensamente en la sección de implementación de sensores personalizados. La inclusión de cuatro sensores diversos demuestra la aplicabilidad del diseño a diferentes paradigmas de comunicación, validando que la arquitectura no impone restricciones artificiales.

9.1.4 Retroalimentación de la validación

La validación experimental demuestra que el framework cumple exitosamente sus objetivos de proporcionar una base modular, eficiente y confiable para el desarrollo de aplicaciones IoT en dispositivos gateway con recursos limitados. Los resultados confirman que las decisiones arquitectónicas fundamentales, la separación en capas, el uso de traits para abstracciones y la selección de Rust como lenguaje de implementación se traducen en un sistema funcional que opera correctamente en hardware real bajo condiciones representativas de despliegues reales.

La capacidad de integrar múltiples sensores con diferentes protocolos de comunicación sin modificar el núcleo del framework valida el diseño modular. Desarrolladores pueden incorporar nuevos dispositivos implementando el trait Sensor, reutilizando toda la infraestructura de

almacenamiento, comunicación y manejo de errores sin duplicar código o comprometer la estabilidad de componentes existentes. Esta extensibilidad resulta esencial en el ecosistema IoT caracterizado por su heterogeneidad de dispositivos y protocolos.

Los resultados de rendimiento obtenidos muestran un uso promedio de CPU cercano al 3%, con incrementos puntuales superiores al 100% durante lecturas de sensores con temporización estricta, como los DHT11 y DHT22. Dichos picos corresponden a periodos de espera activa y no representan una carga sostenida. En cuanto a memoria, el consumo real (RSS) se mantuvo estable entre 6 MB y 7 MB durante toda la ejecución, sin crecimiento progresivo, lo que evidencia ausencia de fugas y una gestión eficiente de recursos. Estas mediciones muestran que es posible desarrollar soluciones IoT en Rust con un nivel de eficiencia comparable al observado en lenguajes de bajo nivel como C o C++, manteniendo beneficios adicionales como seguridad en memoria y manejo explícito de errores.

La tolerancia a fallos demostrada, donde el sistema continúa operando ante desconexiones de sensores individuales o pérdida temporal de conectividad de red, valida que el framework puede soportar las condiciones adversas típicas de despliegues IoT reales. El manejo explícito de errores mediante el sistema de tipos Result de Rust contribuye a esta robustez, forzando a desarrolladores a considerar y manejar apropiadamente todos los casos de fallo posibles.

Las limitaciones identificadas, dependencia del sistema operativo para timing crítico, restricciones en frecuencia de muestreo de ciertos sensores y ausencia de características avanzadas como configuración dinámica, representan compromisos conscientes que priorizan simplicidad arquitectónica y trazabilidad del comportamiento. Estas funcionalidades pueden incorporarse en versiones futuras sin modificar los principios fundamentales del diseño.

El framework desarrollado constituye una contribución significativa al ecosistema de desarrollo IoT en Rust, proporcionando una alternativa viable a soluciones existentes que sacrifican eficiencia por conveniencia o que requieren expertise profundo en programación de sistemas. La documentación exhaustiva y los ejemplos funcionales reducen la barrera de entrada, permitiendo que desarrolladores con conocimientos básicos de Rust puedan construir aplicaciones IoT robustas sobre dispositivos gateway accesibles como la Raspberry Pi 4.

9.2 Validación de integración con Smart Campus UIS Cloud

El escenario de validación descrito en la sección 9.1 se enfocó en verificar el comportamiento del framework desde la adquisición del dato en el sensor hasta su publicación en el broker MQTT, sin extenderse hacia el componente Cloud del Smart Campus UIS ni hacia la visualización final de los datos. Dado que uno de los objetivos centrales del framework es facilitar la integración de dispositivos gateway con dicha plataforma, se diseñó un segundo escenario de validación que cubre la cadena completa: desde la lectura del sensor en el gateway hasta el almacenamiento y visualización del dato en un dashboard del Smart Campus UIS.

9.2.1 Arquitectura del escenario

Para este escenario se desplegó en la Raspberry Pi una aplicación construida con Lince disponible en el siguiente repositorio LinceEjemplos, la cual lee temperatura mediante un sensor DS18B20 conectado al bus OneWire (device_id 28-00000b0e60f1). La aplicación acumula lecturas en lotes de 10, con un intervalo de 1 s entre lecturas individuales y, al completar cada lote, calcula el promedio, el valor mínimo y el valor máximo de temperatura del lote. La aplicación formatea cada resultado mediante SmartCampusFormatter, estructurando el mensaje según el

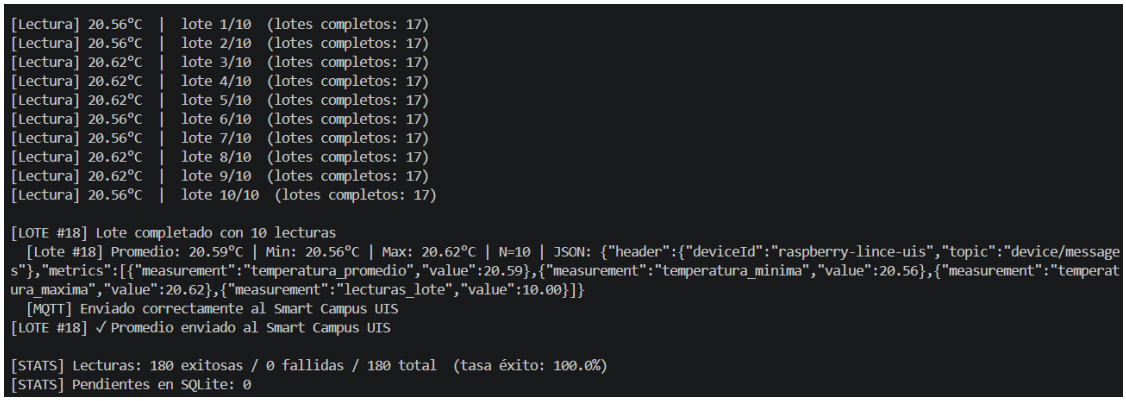
esquema de header (deviceId raspberry-lince-uis, topic device/messages) y métricas (temperatura_promedio, temperatura_minima, temperatura_maxima, lecturas_lote) esperado por el componente Cloud, y lo publica mediante MqttCommunicator hacia el broker del Smart Campus UIS en la dirección 159.65.231.88:1883. El componente Cloud, ya desplegado y administrado por el Smart Campus UIS, recibe el mensaje, lo almacena y lo expone a través de un panel de Grafana configurado para esta validación.

9.2.2 Procedimiento de validación

La validación se ejecutó siguiendo el procedimiento descrito a continuación. Primero, se verificó la conectividad entre el gateway y el broker MQTT del Smart Campus UIS mediante mosquitto_sub -h 159.65.231.88 -p 1883 -t device/messages -v, confirmando la recepción de los mensajes publicados por la aplicación. Segundo, se ejecutó la aplicación de ejemplo en la Raspberry Pi durante un período aproximado de 30 minutos, registrando en consola cada lote de 10 lecturas junto con su promedio, mínimo y máximo. Tercero, se accedió al dashboard de Grafana y se verificó que las métricas recibidas (temperatura_promedio, temperatura_minima, temperatura_maxima) coincidieran con los valores calculados localmente en el gateway. Finalmente, se documentó mediante capturas de pantalla cada etapa relevante del flujo, desde la consola de la aplicación hasta el panel de visualización.

Figura 19.

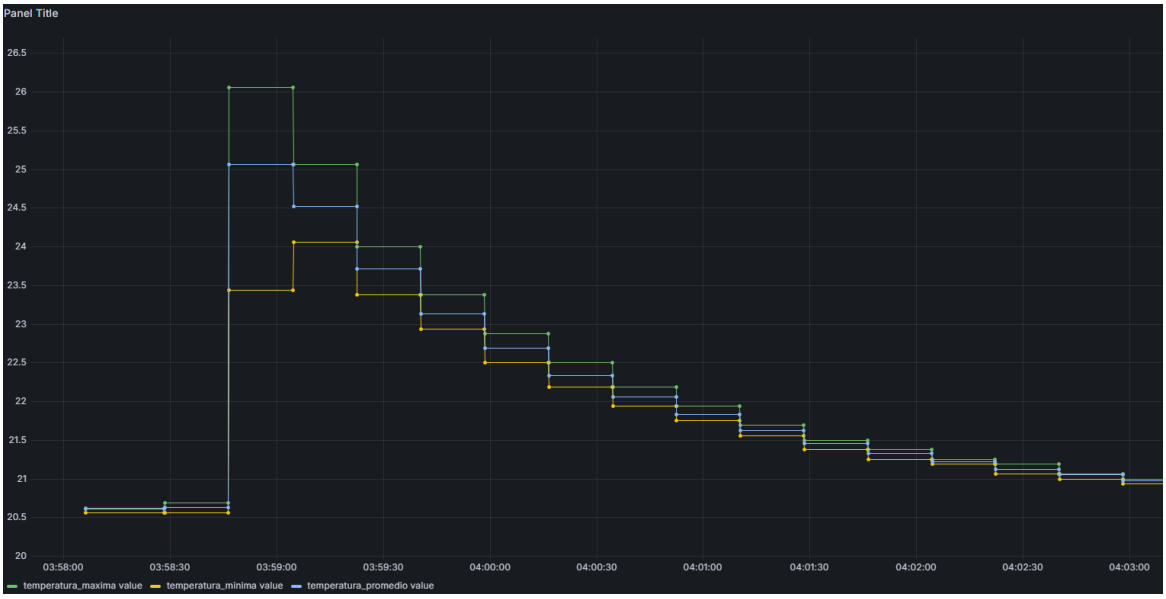
Consola de la aplicación en la Raspberry Pi durante el flujo hacia el Smart Campus UIS



La consola muestra el lote #18 completado con 10 lecturas del sensor DS18B20, con un promedio de 20.59°C (mínimo 20.56°C, máximo 20.62°C), formateado mediante SmartCampusFormatter y enviado exitosamente al broker del Smart Campus UIS. Las estadísticas acumuladas confirman 180 lecturas exitosas de 180 totales (100% de éxito) y 0 mensajes pendientes en SQLite, evidenciando el funcionamiento nominal del flujo completo bajo condiciones normales de conectividad.

Figura 20.

Panel de Grafana del Smart Campus UIS mostrando las métricas recibidas



El panel de Grafana confirma la recepción de las tres métricas configuradas (temperatura_promedio, temperatura_minima, temperatura_maxima) durante la ventana de prueba, con valores consistentes con los reportados en consola por la aplicación, verificando así que el dato recorre exitosamente toda la cadena gateway, bróker, componente Cloud, visualización sin intervención manual.

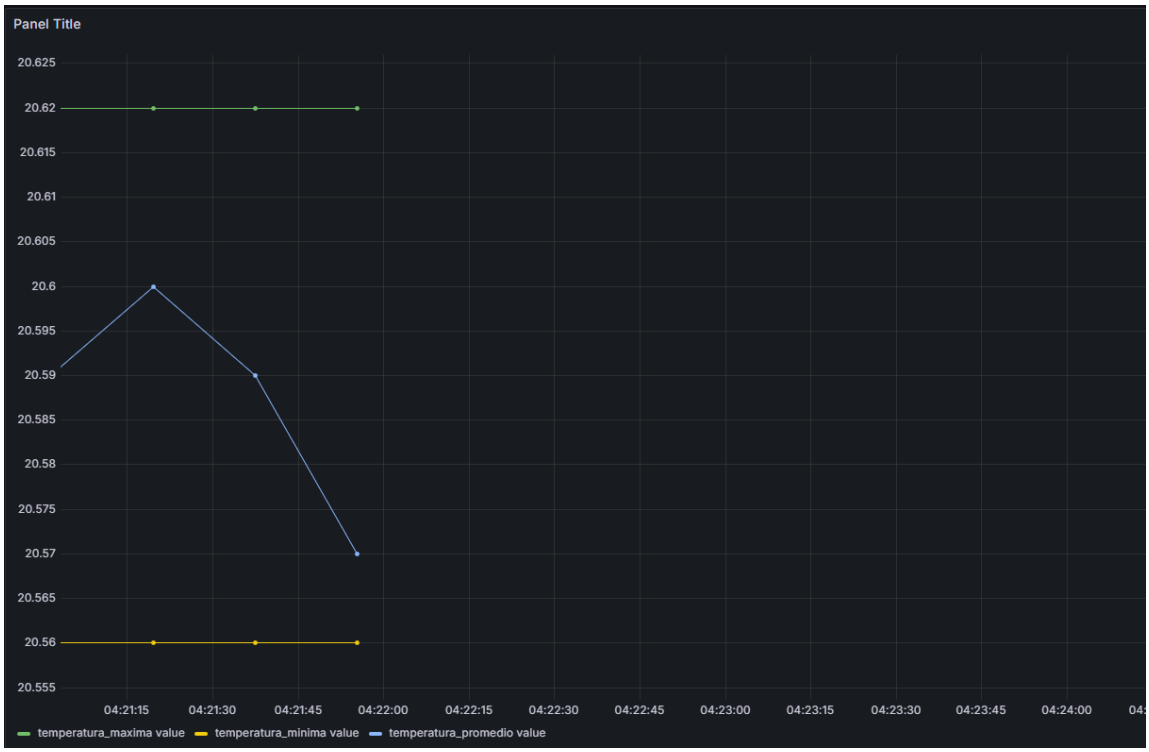
9.2.3 Validación del reintento por desconexión con SqliteStorage

Además de verificar el flujo nominal de datos hacia el Smart Campus UIS, se diseñó una prueba específica para demostrar el funcionamiento del mecanismo de almacenamiento persistente con reintento de envío. Esta prueba valida que SqliteStorage conserva las lecturas durante una interrupción de la red y las reenvía automáticamente al recuperarse la conectividad, garantizando que ningún dato adquirido durante el período de desconexión se pierda.

El procedimiento de prueba consistió en los siguientes pasos. Primero, se inició la aplicación con SqliteStorage habilitado, verificando que el contador de mensajes pendientes reportado al arranque era cero. Segundo, con la aplicación corriendo y publicando lecturas con éxito, se interrumpió deliberadamente la comunicación de red bloqueando el puerto 1883 del broker, manteniendo activa la conexión SSH para supervisar la ejecución en la Raspberry Pi. Tercero, durante aproximadamente 7 minutos de desconexión, la aplicación continuó leyendo el sensor y almacenando cada lectura en SQLite con estado sent = 0, reportando en consola el error CommunicatorError::Disconnected e incrementando el contador de pendientes en cada ciclo. Durante este período, el framework intentó periódicamente reenviar la cola acumulada; al no haber conectividad, estos intentos reportaron en consola el resultado "Enviados en este ciclo: 0/N" y el mensaje generado en ese ciclo se sumaba a la cuenta de pendientes. Al finalizar el período de desconexión, pending_count reportaba 23 mensajes pendientes. Cuarto, se restauró la conectividad eliminando la regla de iptables, tras lo cual el primer ciclo exitoso de envío activó flush_pending, que procesó la cola completa de 23 mensajes pendientes y los marcó como sent = 1 en la base de datos.

Figura 21.

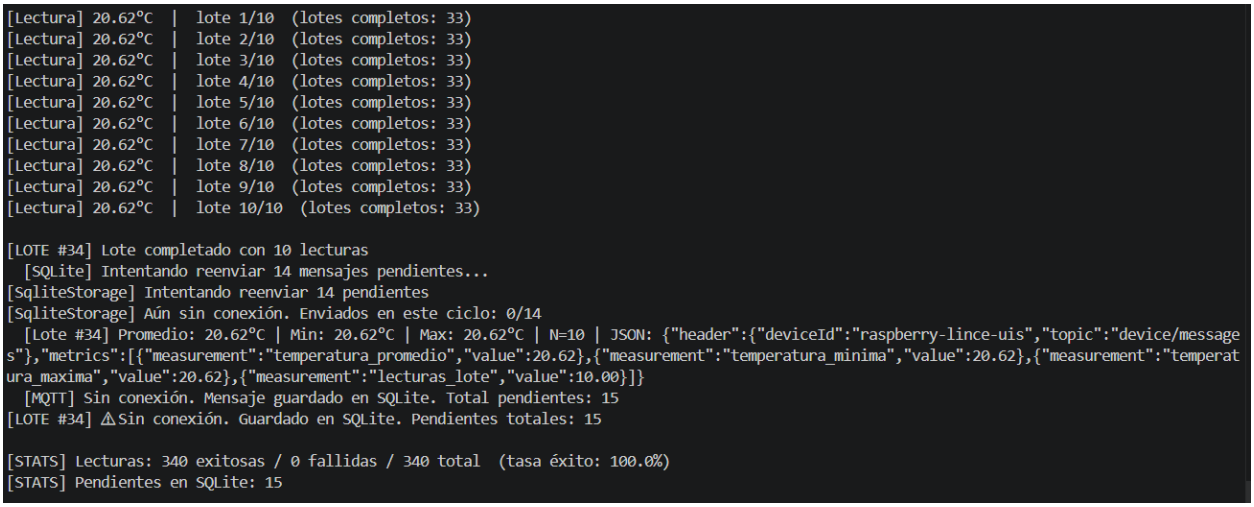
Panel de Grafana al inicio de la interrupción de red durante la prueba de reintento con SqliteStorage



El panel muestra la lectura de temperatura del sensor DS18B20 estabilizada entre 20.55°C y 20.62°C justo en el tramo donde inicia el bloqueo del puerto 1883, momento a partir del cual el componente Cloud deja de recibir nuevas actualizaciones aunque el gateway continúa midiendo localmente.

Figura 22.

Consola del gateway durante la desconexión



La captura corresponde al lote #34, ya con la red bloqueada: el sistema reporta 14 mensajes pendientes acumulados en SQLite e intenta reenviarlos automáticamente, pero al no existir conectividad el resultado del ciclo es “Enviados en este ciclo: 0/14”. El lote recién completado también se almacena localmente con sent = 0, elevando el contador a 15 mensajes pendientes. Esto evidencia que el mecanismo de reintento se ejecuta de forma periódica durante toda la desconexión, no solo al momento de restablecer la red.

Figura 23.

Tabla messages en SQLite durante la desconexión

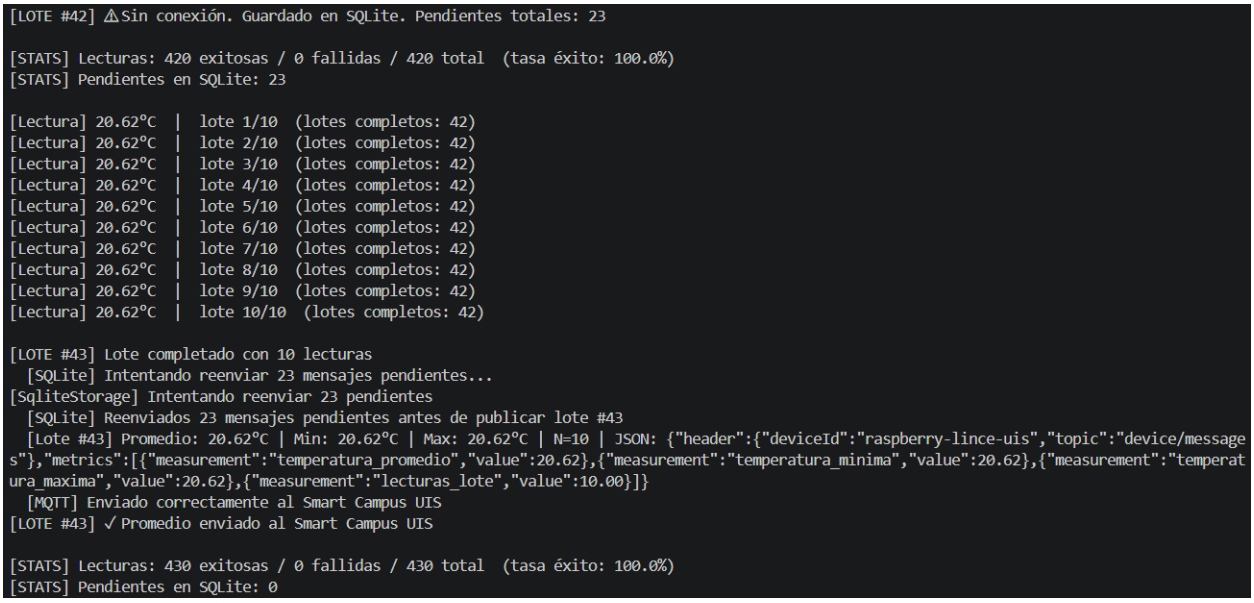
☐	40	1783242591	Text	{"header":{"deviceId":"raspber ...	1
☐	41	1783242609	Text	{"header":{"deviceId":"raspber ...	1
☐	42	1783242627	Text	{"header":{"deviceId":"raspber ...	1
☐	43	1783242645	Text	{"header":{"deviceId":"raspber ...	1
☐	44	1783242663	Text	{"header":{"deviceId":"raspber ...	1
☐	45	1783242681	Text	{"header":{"deviceId":"raspber ...	1
☐	46	1783242698	Text	{"header":{"deviceId":"raspber ...	1
☐	47	1783242716	Text	{"header":{"deviceId":"raspber ...	1
☐	48	1783243333	Text	{"header":{"deviceId":"raspber ...	0
☐	49	1783243351	Text	{"header":{"deviceId":"raspber ...	0
☐	50	1783243369	Text	{"header":{"deviceId":"raspber ...	0

☐	id	timestamp	kind	value	sent
☐	51	1783243387	Text	{"header":{"deviceId":"raspber ...	0
☐	52	1783243405	Text	{"header":{"deviceId":"raspber ...	0
☐	53	1783243423	Text	{"header":{"deviceId":"raspber ...	0
☐	54	1783243440	Text	{"header":{"deviceId":"raspber ...	0
☐	55	1783243458	Text	{"header":{"deviceId":"raspber ...	0
☐	56	1783243476	Text	{"header":{"deviceId":"raspber ...	0
☐	57	1783243494	Text	{"header":{"deviceId":"raspber ...	0
☐	58	1783243512	Text	{"header":{"deviceId":"raspber ...	0
☐	59	1783243530	Text	{"header":{"deviceId":"raspber ...	0
☐	60	1783243548	Text	{"header":{"deviceId":"raspber ...	0
☐	61	1783243566	Text	{"header":{"deviceId":"raspber ...	0
☐	62	1783243584	Text	{"header":{"deviceId":"raspber ...	0
☐	63	1783243602	Text	{"header":{"deviceId":"raspber ...	0
☐	64	1783243620	Text	{"header":{"deviceId":"raspber ...	0
☐	65	1783243638	Text	{"header":{"deviceId":"raspber ...	0
☐	66	1783243656	Text	{"header":{"deviceId":"raspber ...	0
☐	67	1783243674	Text	{"header":{"deviceId":"raspber ...	0

Las filas de la tabla messages muestran los identificadores 40 a 67 de la base SQLite. Las primeras (id 40 a 47) conservan sent = 1 por haberse enviado antes de bloquear la red; a partir del id 48 la columna sent cambia a 0, marcando el inicio de la cola de mensajes almacenados localmente mientras la conectividad permanece interrumpida.

Figura 24.

Consola del gateway al restablecerse la conectividad



La consola confirma que, justo antes de reconectar, el contador de pendientes había llegado a 23 mensajes. Al completarse el siguiente lote (#43) con conectividad ya restablecida, el sistema reportó “Reenviados 23 mensajes pendientes antes de publicar lote #43”, y el contador de pendientes en SQLite volvió a 0, confirmando que flush_pending procesó la totalidad de la cola sin pérdida de datos.

Figura 25.

Tabla messages en SQLite tras el reenvio.

id	timestamp	kind	value	sent
51	1783243387	Text	{"header":{"deviceId":"raspber ...	1
52	1783243405	Text	{"header":{"deviceId":"raspber ...	1
53	1783243423	Text	{"header":{"deviceId":"raspber ...	1
54	1783243440	Text	{"header":{"deviceId":"raspber ...	1
55	1783243458	Text	{"header":{"deviceId":"raspber ...	1
56	1783243476	Text	{"header":{"deviceId":"raspber ...	1
57	1783243494	Text	{"header":{"deviceId":"raspber ...	1
58	1783243512	Text	{"header":{"deviceId":"raspber ...	1
59	1783243530	Text	{"header":{"deviceId":"raspber ...	1
60	1783243548	Text	{"header":{"deviceId":"raspber ...	1
61	1783243566	Text	{"header":{"deviceId":"raspber ...	1
62	1783243584	Text	{"header":{"deviceId":"raspber ...	1
63	1783243602	Text	{"header":{"deviceId":"raspber ...	1
64	1783243620	Text	{"header":{"deviceId":"raspber ...	1
65	1783243638	Text	{"header":{"deviceId":"raspber ...	1
66	1783243656	Text	{"header":{"deviceId":"raspber ...	1
67	1783243674	Text	{"header":{"deviceId":"raspber ...	1
68	1783243692	Text	{"header":{"deviceId":"raspber ...	1
69	1783243710	Text	{"header":{"deviceId":"raspber ...	1
70	1783243728	Text	{"header":{"deviceId":"raspber ...	1

Figura 26.

Vista tabular en Grafana de los valores reenviados.

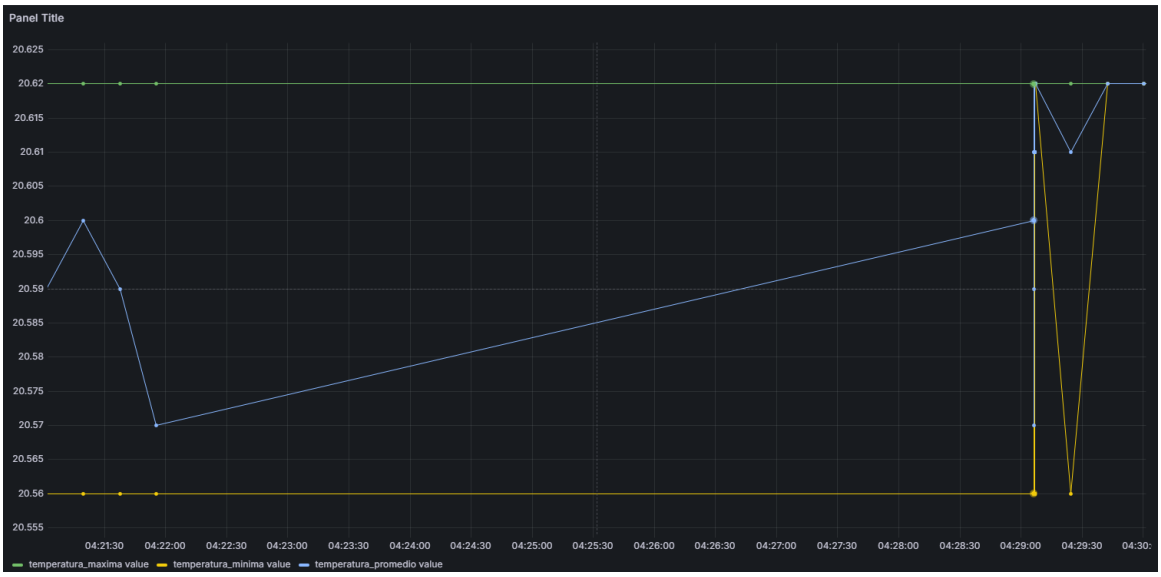
Time	value
2026-07-05 04:20:43	20.6
2026-07-05 04:21:01	20.6
2026-07-05 04:21:19	20.6
2026-07-05 04:21:37	20.6
2026-07-05 04:21:55	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6
2026-07-05 04:29:06	20.6

La tabla del panel de Grafana confirma, desde el lado del componente Cloud, el efecto del reenvío: mientras los primeros puntos llegan espaciados cada 18 segundos aproximadamente (ritmo normal de publicación por lote), un conjunto de registros comparte exactamente la misma

marca de tiempo de recepción (2026-07-05 04:29:06), evidencia directa de que esos datos, adquiridos minutos antes durante la desconexión, llegaron todos juntos al restablecerse la comunicación gracias a flush_pending.

Figura 27.

Gráfica de Grafana del período completo de desconexión y recuperación



En conjunto, la evidencia recopilada en consola, en la base de datos SQLite y en el propio panel de Grafana confirma de manera consistente el comportamiento esperado de SqliteStorage: ningún dato adquirido durante la interrupción de red se perdió, todos los mensajes acumulados se marcaron correctamente como pendientes, y la totalidad de la cola se reenvió y confirmó exitosamente al restablecerse la conectividad, sin intervención manual ni reinicio de la aplicación.

9.2.4 Conclusiones del escenario de integración

Este escenario complementa la validación funcional presentada en la sección 9.1, que se limitó a verificar el comportamiento del framework hasta la publicación del mensaje en el broker MQTT. Al extender la prueba hasta el componente Cloud y el dashboard del Smart Campus UIS,

se demuestra que la integración fue exitosa sin necesidad de que el desarrollador implementara lógica adicional de formateo o protocolo, así como también se verifico que el sistema de reintento de mensajes funcionaba correctamente.

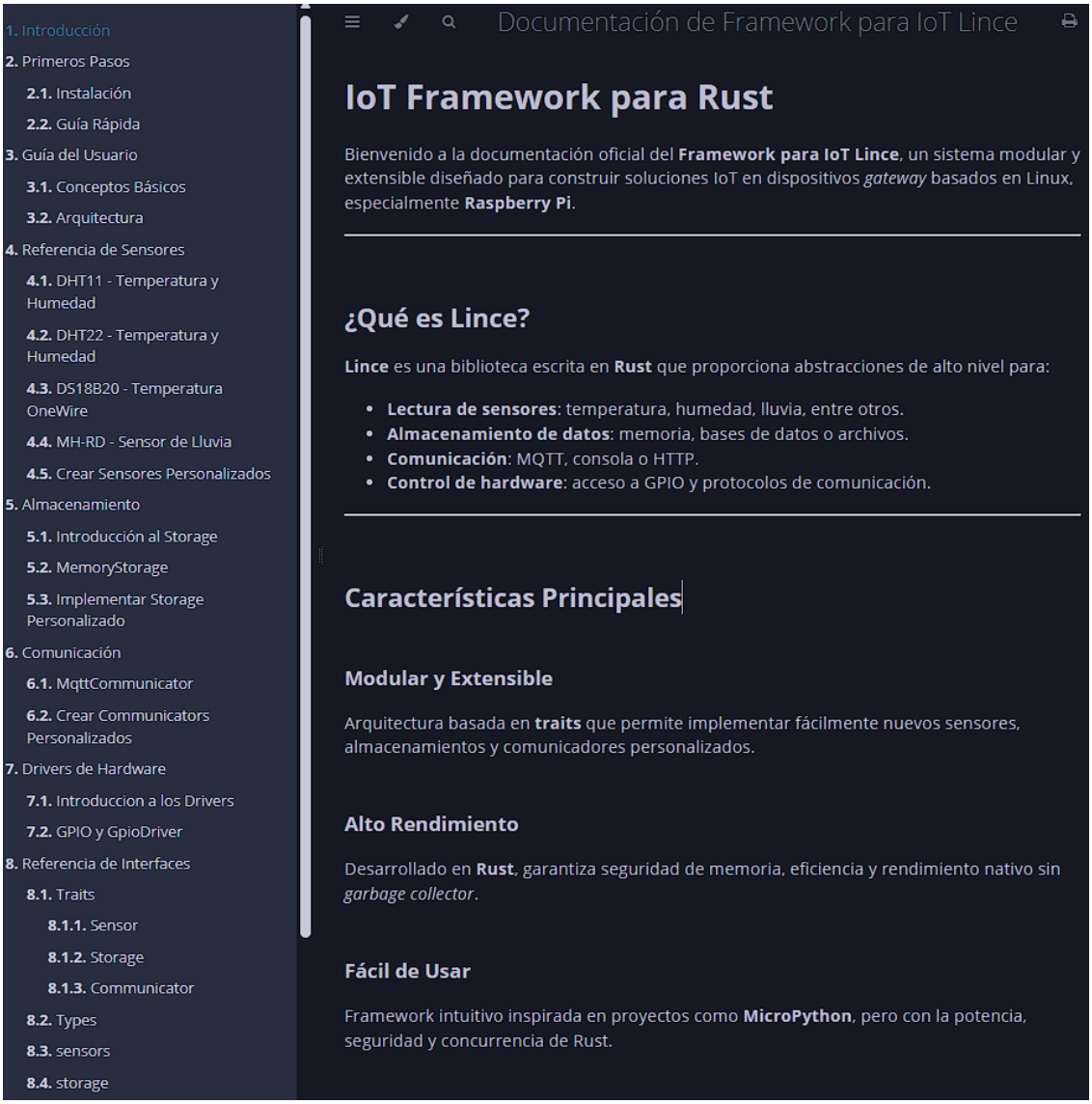
10 Documentación del framework

Esta fase corresponde a las actividades de documentación generada durante el desarrollo del proyecto, cuyo propósito fue consolidar los resultados obtenidos, documentar el uso y funcionamiento del framework, y presentar de manera estructurada el trabajo desarrollado. La documentación técnica constituye un componente fundamental para garantizar la adopción y mantenibilidad del framework. Conscientes de que la curva de aprendizaje de Rust puede representar una barrera para desarrolladores sin experiencia previa en el lenguaje, se desarrolló un sistema de documentación integral que acompaña al usuario desde la instalación inicial hasta la implementación de componentes personalizados avanzados.

La estructura de la documentación fue concebida para facilitar diferentes niveles de interacción con el framework. Se utilizó "mdBook” como herramienta principal de generación, una elección natural dentro del ecosistema Rust que permite crear documentación estática navegable de forma intuitiva. El punto de entrada natural es la introducción, donde se presenta la visión general del framework. Esta sección inicial establece las expectativas sobre lo que el framework puede hacer y da paso a las siguientes secciones de la documentación.

Figura 28.

Pagina de inicio de la documentación del framework Lince



Nota. Captura tomada de la página web oficial de la documentación del framework Lince (https://documentacionlincefrm.web.app/).

Inmediatamente después de comprender qué es Lince, los desarrolladores necesitan poder probarlo rápidamente. Por ello, la documentación incluye una sección de Primeros Pasos que guía al usuario a través de la instalación del entorno de desarrollo, la configuración inicial de la Raspberry Pi 4 y la ejecución de un ejemplo mínimo funcional.

Seguido de esto se profundiza en los conceptos básicos necesarios para una mejor comprensión del framework a través de la Guía del Usuario. También en esta sección se explica la arquitectura por capas del framework, comenzando por el núcleo que define los traits Sensor, Storage y Communicator. La explicación de estos traits no se presenta de forma aislada, sino mostrando cómo cada abstracción resuelve un problema concreto del desarrollo IoT.

La guía del usuario profundiza en los fundamentos conceptuales y arquitectónicos del framework. Esta sección se divide en dos componentes principales que se complementan mutuamente. Los conceptos básicos introducen las abstracciones fundamentales sobre las cuales se construye todo el sistema. Se explica el papel de los traits como contratos de comportamiento, los tres traits principales: Sensor para la lectura de datos, Storage para el almacenamiento y Communicator para la comunicación. Esta sección también aborda conceptos cruciales del lenguaje Rust como ownership, borrowing y manejo de errores, contextualizándolos siempre dentro del dominio específico de aplicaciones IoT.

La descripción de la arquitectura complementa estos conceptos básicos con una visión estructural del framework. Se presenta una organización modular que distingue claramente entre el núcleo del sistema, las implementaciones de dispositivos, los drivers de hardware, los sistemas de almacenamiento y los mecanismos de comunicación. El núcleo define las abstracciones mediante traits y los tipos de datos compartidos, mientras que el módulo de dispositivos contiene las implementaciones concretas de sensores físicos como DHT11, DHT22, Ds18b20 y MH-RD. Los drivers proporcionan una capa de abstracción sobre el hardware de bajo nivel, particularmente a través del GpioDriver que implementa interfaces estándar de embedded-hal para garantizar portabilidad. Los sistemas de almacenamiento y comunicación permiten la persistencia de datos y la integración con sistemas externos, respectivamente.

La referencia de sensores constituye una de las secciones más extensas de la documentación, reflejando la importancia central que estos componentes tienen en cualquier aplicación IoT. Cada sensor soportado recibe un tratamiento detallado que incluye sus especificaciones técnicas, esquemas de conexión de hardware, instrucciones de uso y ejemplos prácticos. Explicando las limitaciones de cada sensor en términos de precisión y rango de operación. Para cada sensor, se proporciona información sobre el protocolo de comunicación específico, el formato de los datos, las limitaciones de hardware y las consideraciones de timing necesarias para lecturas correctas.

La documentación también incluye una sección dedicada a la creación de sensores personalizados, reconociendo que las aplicaciones reales frecuentemente requieren integrar hardware no incluido en las implementaciones predeterminadas. Esta guía explica la anatomía básica de un sensor personalizado, describiendo cómo debe estructurarse la implementación para cumplir con el trait Sensor. Se proporcionan ejemplos como sensores simulados simples, siempre enfatizando aspectos críticos como el manejo apropiado de errores, la validación de datos y la gestión eficiente de recursos de hardware.

El módulo de almacenamiento demuestra la importancia de la persistencia de datos en aplicaciones IoT. La implementación MemoryStorage se presenta como el sistema de almacenamiento fundamental incluido en el framework. Esta implementación en memoria RAM ofrece acceso extremadamente rápido pero sin persistencia entre reinicios, lo cual la hace ideal para escenarios de prueba, buffer temporal o caché de datos recientes. Se proporcionan ejemplos prácticos que demuestran casos de uso comunes, como el cálculo de estadísticas sobre conjuntos de lecturas. La sección se complementa con una guía para implementar sistemas de

almacenamiento personalizados, permitiendo a los desarrolladores crear soluciones específicas según las necesidades de sus aplicaciones.

La comunicación, implementada principalmente a través del módulo network, representa el mecanismo mediante el cual las aplicaciones IoT interactúan con sistemas externos. El componente MqttCommunicator se presenta como la implementación principal, proporcionando integración con el protocolo MQTT. La documentación explica los conceptos fundamentales de este protocolo, incluyendo el modelo publicación-suscripción, los brokers, topics y niveles de calidad de servicio. Se detallan los pasos necesarios para instalar y configurar un broker MQTT local usando Mosquitto, así como las consideraciones para configurar el comunicador dentro de una aplicación. La sección también aborda la creación de comunicadores personalizados, reconociendo que diferentes aplicaciones pueden requerir protocolos alternativos como HTTP, WebSocket o comunicación serial.

La sección de drivers que constituyen la capa más baja de abstracción en el framework, proporcionando interfaces para interactuar con los periféricos del dispositivo. El GpioDriver es el componente fundamental para el control de pines de entrada/salida de propósito general. La documentación explica cómo este driver encapsula la funcionalidad de la biblioteca rppal específica de Raspberry Pi, presentando una interfaz simple y compatible con los traits estándar de embedded-hal. Se detallan las operaciones básicas de lectura y escritura de pines, la configuración de modos y las especificaciones eléctricas críticas que los desarrolladores deben considerar. Esta sección también proporciona recomendaciones sobre qué pines son seguros para uso general y cuáles deben evitarse por estar reservados para protocolos específicos como I2C, SPI o UART.

La referencia completa de interfaces proporciona documentación técnica detallada de todos los traits, tipos y módulos del framework. Esta sección está organizada de manera sistemática, comenzando por los traits del núcleo que definen los contratos fundamentales del sistema. Para cada trait se especifica su propósito, los métodos requeridos, los tipos asociados y los comportamientos esperados de las implementaciones. Los tipos como SensorOutput y los diversos enums de error se documentan explicando cada variante y proporcionando ejemplos de uso apropiado. Las implementaciones concretas de sensores, sistemas de almacenamiento y comunicadores reciben referencias detalladas que incluyen la firma de cada método público, sus parámetros, valores de retorno y posibles condiciones de error.

Los apéndices complementan la documentación principal con material de referencia que resulta invaluable durante el desarrollo. El glosario define términos técnicos, acrónimos y conceptos específicos tanto del dominio IoT como del framework particular. Términos como GPIO, MQTT, OneWire, pull-up y trait se explican de manera accesible, proporcionando el contexto necesario para comprender la documentación más técnica.

La referencia de códigos de error constituye una guía práctica para la resolución de problemas, categorizando los errores posibles y proporcionando para cada uno su descripción, causas comunes y soluciones recomendadas. Esta sección resulta particularmente útil durante la depuración, ya que permite a los desarrolladores identificar rápidamente la causa raíz de problemas frecuentes como timeouts de sensores, errores de permisos de GPIO o fallos de comunicación.

La documentación de hardware compatible proporciona información técnica detallada sobre la Raspberry Pi 4, que es la plataforma principal para la cual está diseñado el framework. Se presentan las especificaciones completas del dispositivo, incluyendo el procesador, memoria, configuración de GPIO y características eléctricas. El diagrama de pinout se incluye como

referencia visual esencial, claramente etiquetado con la numeración BCM que utiliza el framework. Esta sección también detalla los requisitos del sistema operativo, las dependencias de software necesarias y los procedimientos de configuración específicos para habilitar diferentes interfaces de hardware. Las consideraciones eléctricas reciben especial atención, advirtiendo sobre los límites de corriente, los niveles de voltaje y la necesidad de protección adecuada para evitar daños al hardware. Se proporcionan tablas de referencia que indican qué pines son seguros para uso general y cuáles están reservados para funciones específicas, así como recomendaciones sobre la alimentación apropiada del dispositivo y los sensores conectados.

Esta documentación constituye un recurso fundamental que no solo explica cómo usar el framework, sino que también transmite los principios de diseño y las decisiones arquitectónicas que lo fundamentan. Permite a los usuarios no solo implementar aplicaciones funcionales, sino comprender cómo funciona el sistema, capacitándolos para extenderlo, adaptarlo y depurarlo efectivamente según las necesidades específicas de sus proyectos.

11 Trabajo futuro

El desarrollo del framework Lince permitió establecer las bases de una arquitectura modular, extensible y orientada al uso práctico en dispositivos gateway como la Raspberry Pi 4. Sin embargo, como todo proyecto de software embebido y orientado a IoT, su evolución depende de incorporar nuevas capacidades técnicas y mejorar aquellas que hoy funcionan, pero pueden optimizarse.

11.1 Ampliación del soporte para sensores y protocolos de hardware

Actualmente, Lince integra sensores DHT11, DHT22, DS18B20 y MH-RD a través de un driver de comunicación digital simple. Una fase futura consiste en extender la librería hacia sensores basados en I2C, SPI y UART, permitiendo cubrir una variedad más amplia de aplicaciones IoT como de medición de calidad del aire, energía, presión barométrica o presencia.

11.2 Implementación de un runtime más robusto

El framework ya permite la ejecución paralela de sensores, aunque actualmente requiere que cada uno sea inicializado manualmente y que su ciclo de lectura se configure explícitamente. Una siguiente etapa podría simplificar este proceso mediante:

- Inicialización automática basada en configuración.
- Scheduling unificado para controlar intervalos de ejecución.
- Límites de tiempo por lectura para evitar bloqueos.
- Separación más estricta entre sensores y módulos de comunicación.

Estas mejoras reducirían el esfuerzo de configuración y harían el framework más robusto y escalable, alineándolo con prácticas de frameworks IoT orientados a producción.

11.3 Extension de la integración de almacenamiento persistente

El módulo MemoryStorage cumple su propósito para validación rápida, pero no permite persistencia real ante reinicios. Esta limitación ya fue abordada mediante la incorporación de SqliteStorage, descrita en la sección de funcionalidades del framework, que provee almacenamiento persistente con reintento automático de envío. Los siguientes pasos en esta línea de trabajo implicarían extender las opciones de persistencia hacia:

- Archivos locales (JSON, CSV o binarios)
- Sistemas distribuidos para gateways con mayor capacidad

  La persistencia es clave para escenarios de auditoría y análisis histórico.

11.4 Portal de administración o CLI

Un primer paso en esta dirección, listar los sensores disponibles, configurar sus parámetros de hardware y habilitar o deshabilitar módulos opcionales sin escribir código manualmente, ya fue cubierto mediante el asistente de generación simple descrito en la sección de funcionalidades del framework. Dicha herramienta permite seleccionar sensores, definir sus pines o identificadores, y activar o desactivar el formato Smart Campus y el almacenamiento persistente con reintento, generando como resultado un proyecto Rust completo y compilable. Queda pendiente, sin embargo, una capacidad que el asistente no cubre por tratarse de una herramienta de generación previa a la ejecución: el monitoreo en tiempo real de métricas del gateway una vez la aplicación está corriendo.

- Monitoreo en tiempo real de métricas del gateway (uso de CPU, memoria, estado de conexión MQTT y tasa de éxito por sensor) mientras la aplicación generada se ejecuta en el dispositivo.

La incorporación del asistente de generación acerca a Lince a la accesibilidad buscada para usuarios no expertos en programación, mientras que el monitoreo en vivo representa la extensión natural una vez la aplicación generada se encuentra desplegada. En conjunto, estas líneas de trabajo permitirían consolidar a Lince como un framework IoT estable y confiable, capaz de integrarse en escenarios reales de un Smart Campus o de un entorno industrial ligero.

12 Conclusiones

El proyecto Lince demostró que es posible construir un framework enfocado a la capa Edge para aplicaciones IoT, de tal forma Lince es un framework modular, flexible, extensible, y eficiente utilizando Rust como tecnología principal. La arquitectura diseñada permitió integrar sensores heterogéneos, gestionar almacenamiento temporal y realizar comunicación confiable mediante MQTT, todo bajo un esquema de contratos conocidos como traits que favorece la extensibilidad.

Una de las conclusiones más relevantes es que Rust ofrece ventajas reales para entornos embebidos, especialmente en gateways IoT, donde la seguridad, la predictibilidad y el rendimiento son requisitos esenciales. La ausencia de un garbage collector y el sistema de ownership permiten desarrollar componentes de bajo nivel sin sacrificar robustez ni claridad estructural ofrecidos por lenguajes de alto nivel.

Asimismo, la estructura modular del framework probó ser adecuada para escenarios donde se requiere agregar o reemplazar componentes sin afectar el comportamiento global. El diseño basado en traits permitió que nuevos sensores, modelos de comunicación, modelos de almacenamiento o formatos de mensajes puedan incorporarse sin reescribir la lógica central.

Igualmente, Lince se integra dentro de la iniciativa Smart Campus UIS ofreciendo la posibilidad de facilitar el desarrollo de componentes de sistemas IoT en la capa Edge e integrando esos componentes con la capa Cloud para lograr funcionalidades IoT de valor agregado en los diferentes niveles necesarios (sensores, Edge, cloud).

El prototipo funcional validó la idea principal, es posible abstraer sensores físicos, almacenamientos y canales de comunicación en un mismo ecosistema, manteniendo simplicidad para el desarrollador final.

Durante el desarrollo y validación se identificaron limitaciones inherentes al enfoque actual. El timing crítico de protocolos como DHT requiere precisión de microsegundos que sistemas operativos de propósito general no garantizan, resultando en tasas de error altas sin implementar un reintento. El escenario de validación documentado se ejecutó con MemoryStorage, opción que por sí sola limita aplicaciones que requieren persistencia, aunque esta limitación fue posteriormente abordada incorporando SqliteStorage como alternativa opcional con reintento de envío. De manera similar, la configuración dinámica mediante archivos externos, inicialmente ausente, fue habilitada a través de la plantilla de configuración basada en config.toml. Estas limitaciones puntuales no invalidan el diseño, sino que marcan el proceso natural de evolución de un framework académico, cuyo alcance se amplía progresivamente más allá del prototipo inicial validado.

Finalmente, el proyecto deja claro que Lince no es un producto terminado, sino una base coherente y sólida sobre la cual continuar construyendo. Las mejoras futuras como almacenamiento persistente, drivers adicionales, nuevas capas de comunicación y mayor seguridad representan oportunidades de crecimiento natural del framework.

Lince demuestra que es posible combinar investigación académica con soluciones prácticas, aportando no solo conocimiento técnico, sino una plataforma reutilizable que podría integrarse en iniciativas reales del campus o de entornos IoT similares.

Referencias Bibliográficas

Abuarqoub, A., Alsmadi, M., Al-Ayyoub, M., Gupta, B., & Jararweh, Y. (2022). Smart campus towards connected learning environments: A review of enabling technologies and applications. *Sensors*, *22*(18), 6912. https://doi.org/10.3390/s22186912

Al-Sarawi, S., Anbar, M., Abdullah, R., & Al Hawari, A. B. (2020). Internet of Things market analysis forecasts, 2020–2030. *2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*, 449–453. https://doi.org/10.1109/WorldS450073.2020.9210375

Barchi, F., Pasini, G., Parisi, E., Tagliavini, G., Bartolini, A., & Acquaviva, A. (2023). RUST-encoded stream ciphers on a RISC-V parallel ultra-low-power processor (invited paper). *Schloss Dagstuhl – Leibniz-Zentrum für Informatik*. https://doi.org/10.4230/OASICS.PARMA-DITAM.2023.3

Belleza, R. R., & Pignaton, E. (2018). Performance study of real-time operating systems for internet of things devices. *IET Software*, *12*(3), 176–182. https://doi.org/10.1049/iet-sen.2017.0048

Bugden, W., & Alahmar, A. (2022). Rust: The programming language for safety and performance. *arXiv*. http://arxiv.org/abs/2206.05503

Bytebeam. (2024). *rumqttc: An MQTT client for Rust* [Software]. crates.io. https://crates.io/crates/rumqttc

Carnelos, M., Pasti, F., & Bellotto, N. (2025). MicroFlow: An efficient Rust-based inference engine for TinyML. *Internet of Things*, *30*, artículo 101498. https://doi.org/10.1016/j.iot.2025.101498

Chakraborty, P., Shahriyar, R., & Iqbal, A. (2019). Empirical analysis of the growth and challenges of new programming languages. *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, 191–196. https://doi.org/10.1109/COMPSAC.2019.00034

Chanchí-Golondrino, G.-E., Ospina-Alarcón, M.-A., & Saba, M. (2022). Sistema IoT para el monitoreo de variables climatológicas en cultivos de agricultura urbana. *Revista Científica*, *44*(2), 257–271. https://doi.org/10.14483/23448350.18470

Dauda, A., Flauzac, O., & Nolot, F. (2024). A survey on IoT application architectures. *Sensors*, *24*(16), artículo 5320. https://doi.org/10.3390/s24165320

Domínguez-Bolaño, T., Campos, O., Barral, V., Escudero, C. J., & García-Naya, J. A. (2022). An overview of IoT architectures, technologies, and existing open-source projects. *Internet of Things*, *20*, artículo 100626. https://doi.org/10.1016/j.iot.2022.100626

Dunkels, A., Grönvall, B., & Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. *29th Annual IEEE International Conference on Local Computer Networks*, 455–462. https://doi.org/10.1109/LCN.2004.38

Embedded Rust Working Group. (2024). *embedded-hal: A hardware abstraction layer (HAL) for embedded systems* [Software]. crates.io. https://crates.io/crates/embedded-hal

Ficili, I., Giacobbe, M., Tricomi, G., & Puliafito, A. (2025). From sensors to data intelligence: Leveraging IoT, cloud, and edge computing with AI. *Sensors*, *25*(6), 1763. https://doi.org/10.3390/s25061763

Hong, J., Hong, Y.-G., de Foy, X., Kovatsch, M., Schooler, E., & Kutscher, D. (2024). *RFC 9556: Internet of Things (IoT) edge challenges and functions*. RFC Editor. https://www.rfc-editor.org/rfc/rfc9556.html

Lagsaiar, L., Shahrour, I., Aljer, A., & Soulhi, A. (2021). Modular software architecture for local smart building servers. *Sensors*, *21*(17), 5810. https://doi.org/10.3390/s21175810

Lee, C., Jo, J., Lee, J., An, D., Cho, J., & Jung, R. (2018). RT-OCF: A lightweight device-to-device framework for the Internet of Things. In D. Georgakopoulos & L.-J. Zhang (Eds.), *Internet of Things – ICIOT 2018* (Vol. 10972). Springer. https://doi.org/10.1007/978-3-319-94370-1_13

Makarov (2022). Open-source model-based design driven software framework for real-time safety-critical embedded systems. *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*, 1–8. https://doi.org/10.1109/DASC55683.2022.9925881

Min, D., Xiao, Z., Sheng, B., Quanyong, H., & Xuwei, P. (2014). Design and implementation of heterogeneous IoT gateway based on dynamic priority scheduling algorithm. *Transactions of the Institute of Measurement and Control*, *36*(7), 924–931. https://doi.org/10.1177/0142331214527600

Ortiz, G., Zouai, M., Kazar, O., García-de-Prado, A., & Boubeta-Puig, J. (2024). Atmosphere: Context and situational-aware collaborative IoT architecture for edge–fog–cloud computing. *arXiv*. https://arxiv.org/abs/2401.14968

Petrillo, F. (2025). Should we use Rust platform in our IoT applications? A multivocal review. *Proceedings of the 2025 IEEE/ACM 7th International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT)*. https://doi.org/10.1109/SERP4IoT66600.2025.00009

Plauska, I., Liutkevičius, A., & Janavičiūtė, A. (2023). Performance evaluation of C/C++, MicroPython, Rust and TinyGo programming languages on ESP32 microcontroller. *Electronics*, *12*(1), 143. https://doi.org/10.3390/electronics12010143

RPPAL Contributors. (2024). *RPPAL: Raspberry Pi Peripheral Access Library* [Software]. crates.io. https://crates.io/crates/rppal

Rusqlite Contributors. (2024). *Rusqlite: Ergonomic bindings to SQLite for Rust* [Software]. crates.io. https://crates.io/crates/rusqlite

Serde Contributors. (2024). *Serde: A serialization/deserialization framework for Rust* [Software]. crates.io. https://crates.io/crates/serde

Shanmuganathan, H., & Mahendran, A. (2021). Current trend of IoT market and its security threats. *2021 International Conference on Innovative Computing, Intelligent Communication and Smart Electrical Systems (ICSES)*, 1–9. https://doi.org/10.1109/ICSES52305.2021.9633850

Sharma, A., Sharma, S., Tanksalkar, S. R., Torres-Arias, S., & Machiry, A. (2024). Rust for embedded systems: Current state and open problems. *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2296–2310. https://doi.org/10.1145/3658644.3690275

Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, *3*(5), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

thiserror Contributors. (2024). *thiserror: derive(Error) for Rust* [Software]. crates.io. https://crates.io/crates/thiserror

Ungurean, I., & Gaitan, N. C. (2018). Performance analysis of tasks synchronization for real time operating systems. *2018 International Conference on Development and Application Systems (DAS)*, 63–66. https://doi.org/10.1109/DAAS.2018.8396072

Uzlu, T., & Şaykol, E. (2017). On utilizing Rust programming language for Internet of Things. *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, 93–96. https://doi.org/10.1109/CICN.2017.8319363

Vecchio, M., Azzoni, P., Menychtas, A., Maglogiannis, I., & Felfernig, A. (2021). A fully open-source approach to intelligent edge computing: AGILE's lesson. *Sensors*, *21*(4), 1309. https://doi.org/10.3390/s21041309

Zipit Wireless. (2024). What is an IoT gateway and how does it work? https://www.zipitwireless.com/blog/what-is-an-iot-gateway-and-how-does-it-work