REESTRUCTURACIÓN ARQUITECTURAL Y DOCUMENTACIÓN INTEGRAL DE LA PLATAFORMA SMART CAMPUS UIS

SEBASTIAN CAMILO SUAREZ GALVIS
ALEJANDRO NUÑEZ HERRERA

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECANICAS
ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
PROGRAMA DE PREGRADO EN INGENIERÍA DE SISTEMAS E INFORMÁTICA
BUCARAMANGA
2025

REESTRUCTURACIÓN ARQUITECTURAL Y DOCUMENTACIÓN INTEGRAL DE LA PLATAFORMA SMART CAMPUS UIS

SEBASTIAN CAMILO SUAREZ GALVIS

ALEJANDRO NUÑEZ HERRERA

Trabajo de Grado para optar por el título de:

Ingeniero de Sistemas

Director:

Gabriel Rodrigo Pedraza Ferreira

Ph.D

Codirector:

Henry Andres Jimenez Herrera

MS.c

UNIVERSIDAD INDUSTRIAL DE SANTANDER

FACULTAD DE INGENIERÍAS FISICOMECANICAS

ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

PROGRAMA DE PREGRADO EN INGENIERÍA DE SISTEMAS E INFORMÁTICA

BUCARAMANGA

2025

DEDICATORIA

Sin duda alguna este logro va dirigido especialmente para:

Wilma Maria Galvis Pereira

Ricardo Suarez Buitrago

Workman Ricardo Suarez Galvis

Vilma Yiseth Suarez Galvis

Hugo Andres Suarez Galvis

Ciro Alberto Suarez Galvis

Porque si llegué hasta aquí es gracias a ustedes.

-SEBASTIAN CAMILO SUAREZ GALVIS

DEDICATORIA

A Dios, por darme la fuerza y la sabiduría para continuar, incluso en los momentos más difíciles.
A mis padres, por ser ejemplo de esfuerzo, amor y constancia. Todo lo que soy es gracias a ustedes.
A mi esposa, por acompañarme en cada paso, por su infinita paciencia, y por sostenerme con amor cuando flaqueé.
A mis familiares, quienes siempre han estado ahí, confiando en mí sin condiciones.
Dedico este trabajo a todos ustedes, porque este logro también es suyo.

-ALEJANDRO NUÑEZ HERRERA

AGRADECIMIENTOS

Agradezco a la Universidad Industrial de Santander por ser cuna de grandes profesionales y permitirme ser parte de ella. Agradezco a toda persona que directa o indirectamente colaboró en poner mi plato de desayuno, almuerzo y cena en la sección de comedores, sin duda, una de las columnas más importantes de este logro. Agradezco al profesor y director de grado Gabriel Pedraza, al profesor y también codirector Henry Jiménez, ambos por su gran labor y pasión para enseñar e investigar. Agradezco a todos los profesores que con esmero transmiten su valioso conocimento. Agradezco los compañeros de risas y estudio, en especial la banda los Kunis. Por último y más importante, agradezco a mi madre, por enseñarme la fortaleza, a mi padre, por sostenerme aún así con miles de dificultades, a mis hermanos, los cuales siempre me apoyaron sin limite alguno, a mi pareja, la cual me acompaña con amor y paciencia. Y un agradecimiento especial a mi hermano Ciro Suarez, el cual fue el piñón que desencadenó este logro, por sus sabias palabras y consejos, y a mi mejor amigo y gran compañero de viaje, mi mascota Black.

-SEBASTIAN CAMILO SUAREZ GALVIS

AGRADECIMIENTOS

Agradezco profundamente a todas las personas que me acompañaron y apoyaron durante este proceso académico y personal.

En primer lugar, a mis padres, por ser el pilar fundamental de mi formación, por sus valores, consejos y por brindarme siempre su amor incondicional. Su esfuerzo y dedicación me han inspirado a seguir adelante en cada etapa de mi vida.

A mi esposa, por su paciencia, comprensión y por estar a mi lado en los momentos más retadores. Gracias por tu apoyo inquebrantable y por creer en mí incluso cuando yo dudaba.

A mis tíos, por su respaldo constante a lo largo de mis estudios. Sus palabras de aliento, ayuda y acompañamiento han sido fundamentales para alcanzar esta meta.

Este trabajo no habría sido posible sin el apoyo de cada uno de ustedes. Gracias por ser parte de este logro.

-ALEJANDRO NUÑEZ HERRERA

RESUMEN

TÍTULO: REESTRUCTURACIÓN ARQUITECTURAL Y DOCUMENTACIÓN INTEGRAL DE LA PLATAFORMA SMART CAMPUS UIS. *

AUTOR: SEBASTIAN CAMILO SUAREZ GALVIS, ALEJANDRO NUÑEZ HERRERA **

PALABRAS CLAVE: IoT, Arquitectura, Documentación, Smart Campus, Erosión, Reestructuración.

DESCRIPCIÓN: Este proyecto se basa en el estudio general y de la arquitectura de una plataforma IoT distribuida en la Universidad Industrial de Santander, cuyo objetivo actual es brindar un ecosistema que pueda ser utilizado en la investigación, aplicaciones y desarrollo de software asociado al internet de las cosas. A lo largo del proyecto se estudia la arquitectura actual de la plataforma, se propone una reestructuración e implementación, en base a consideraciones de estudio e investigación. Además, se realiza una documentación técnica, arquitectural y de negocio, con el fin de facilitar la comprensión, mantenimiento y evolución del sistema. Esta documentación y la nueva arquitectura buscan mitigar la erosión y degradación arquitectónica, garantizando la sostenibilidad y una mejor escalabilidad de la plataforma para futuras investigaciones y desarrollos.

---

* Trabajo de investigación

** Facultad de Ingenierías Físicomecánicas. Escuela de Ingeniería de Sistemas e Informática. Director: Gabriel Rodrigo Pedraza Ferreira, Ph.D. Codirector: Henry Andres Jimenez Herrera, M.Sc.

ABSTRACT

TITLE: ARCHITECTURAL RESTRUCTURING AND INTEGRAL DOCUMENTATION OF THE SMART CAMPUS UIS PLATFORM. *

AUTHOR: SEBASTIAN CAMILO SUAREZ GALVIS, ALEJANDRO NUÑEZ HERRERA **

KEYWORDS: IoT, Architecture, Documentation, Smart Campus, Erosion, Restructuring

DESCRIPTION: This project is based on the general study and architecture of a distributed IoT platform at the Industrial University of Santander, whose current objective is to provide an ecosystem that can be used in research, applications, and development of software associated with the Internet of Things. Throughout the project, the current architecture of the platform is studied, and a restructuring and implementation is proposed based on study and research considerations. Additionally, technical, architectural, and business documentation is created to facilitate the understanding, maintenance, and evolution of the system. This documentation and the new architecture aim to mitigate architectural erosion and degradation, ensuring sustainability and better scalability of the platform for future research and developments.

---

* Research work

** Faculty of Physics-Mechanics Engineering. School of Systems Engineering and Informatics. Advisor: Gabriel Rodrigo Pedraza Ferreira, Ph.D. Co-advisor: Henry Andres Jimenez Herrera, M.Sc.

CONTENIDO

page

INTRODUCCIÓN **20**

1. PLANTEAMIENTO Y JUSTIFICACIÓN DEL PROBLEMA **22**

1.1. Caso de estudio 23

2. OBJETIVOS **24**

3. ESTADO DEL ARTE **25**

3.1. M.I.N.G Stack 25

3.2. T.I.C.K Stack 26

3.3. Arquitectura EDA (Event-Driven Architecture) 26

3.4. NETIG Stack 27

4. MARCO DE REFERENCIA **29**

4.1. ARQUITECTURA DE SOFTWARE 29

4.2. DOCUMENTACIÓN INTEGRAL. 29

4.3. DOCKER. 30

4.3.1 Docker compose. 30

4.3.2 Estandarización docker-compose. 30

4.4. JAVA SPRINGBOOT. 31

4.5. CONTROL DE VERSIONES Y PERSISTENCIA DEL CÓDIGO. 32

4.5.1 Git. 32

4.5.2 Git-Flow. 32

4.6. TECNOLOGÍAS ÓPTIMAS. 32

4.6.1 Influx DB. 32

4.6.2 MQTT (*Message Queuing Telemetry Transport*). 33
4.6.3 Brokers de mensajeria MQTT. 33
4.6.4 EMQX. 33
4.6.5 HiveMQ. 34
4.6.6 Mosquitto 35
4.6.7 Comparación EMQX vs HiveMQ vs Mosquitto 36

5. METODOLOGÍA 38
5.1. Enfoque por Sprints 39
5.2. Metodología aplicada 40

6. DESARROLLO 41
6.1. SPRINT 1. DEFINICION ARQUITECTURA SMART CAMPUS UIS 41
6.1.1 Dividir: 41
6.1.2 Analizar: 43
6.1.3 Procesar: 44
6.1.4 Review: 45
6.2. SPRINT 2. DOCUMENTACIÓN SWAGGER CAPA DE ADMINISTRACIÓN . . . 45
6.2.1 Dividir: 45
6.2.2 Analizar: 46
6.2.3 Procesar: 48
6.2.4 Review: 49
6.3. SPRINT 3. REFACTORIZACIÓN CAPA DE DATOS 50
6.3.1 Dividir: 50
6.3.2 Analizar: 50
6.3.3 Procesar: 53
6.3.4 Review: 56
6.4. SPRINT 4. CAPA EXTERNA 56

6.4.1 Dividir: . . . 56
6.4.2 Analizar: . . . 56
6.4.3 Procesar: . . . 57
6.4.4 Review: . . . 58
6.5. SPRINT 5. DOCUMENTACIÓN INTEGRAL CON MKDOCS . . . 59
6.5.1 Dividir: . . . 59
6.5.2 Analizar: . . . 59
6.5.3 Procesar: . . . 61
6.5.4 Review: . . . 61
6.6. SPRINT 6. DESPLIEGUE EN GCP . . . 61
6.6.1 Dividir: . . . 61
6.6.2 Analizar: . . . 61
6.6.3 Procesar: . . . 66
6.6.4 Review: . . . 69
6.7. SPRINT 7. REGLAS DE INTEGRACIÓN Y EXTENSIÓN . . . 69
6.7.1 Dividir: . . . 69
6.7.2 Analizar: . . . 69
6.7.3 Procesar: . . . 70
6.7.4 Review: . . . 70

7. RESULTADOS LOGRADOS . . . 72

8. CONCLUSIONES . . . 75

9. TRABAJO FUTURO . . . 77

BIBLIOGRAFÍA . . . 78

LISTA DE TABLAS

page

Tabla 1. Brokers de mensajería especializados para protocolo MQTT 36

Tabla 2. Credenciales por defecto de los servicios desplegados 69

LISTA DE FIGURAS

page

Figura 1. Divide y vencerás 38
Figura 2. Ciclos sprint 39
Figura 3. Arquitectura Smart Campus UIS 42
Figura 4. Capa de administración Smart Campus UIS 43
Figura 5. Capa de datos Smart Campus UIS 44
Figura 6. Capa Gateway o externa de Smart Campus UIS 44
Figura 7. Tecnologías de la capa de administración Smart Campus UIS 46
Figura 8. Front Admin Code 46
Figura 9. Front Admin Code 47
Figura 10.Evidencia documentación Swagger 49
Figura 11.Tecnologías de la capa de datos Smart Campus UIS 50
Figura 12.NETIG STACK 53
Figura 13.Documentación integral del proyecto 59
Figura 14.Diagrama de la documentación integral del proyecto 60
Figura 15.Cuenta en Google Cloud Platform 62
Figura 16.Seleccion de proyecto en Google Cloud Platform 62
Figura 17.Instancias de VM 63
Figura 18.Crear instancia de VM 63
Figura 19.Seleccion de region y zona 64
Figura 20.Seleccion de tipo de maquina 64
Figura 21.Configurar caracteristicas de la maquina virtual 65
Figura 22.Configurar imagen de la maquina virtual 65
Figura 23.Configurar redes y etiquetas de acceso de la maquina virtual 66

Figura 24.Configurar proyecto en Google Cloud Platform 67
Figura 25.Transferir archivo desde a la maquina virtual 67
Figura 26.Acceder a la maquina virtual por medio de SSH 68
Figura 27.Abrir puertos de la maquina virtual 68

GLOSARIO

IoT: Red de dispositivos físicos interconectados que recopilan y comparten datos.

MQTT: Protocolo de comunicación ligero basado en publicación/suscripción comúnmente utilizado para IoT.

InfluxDB: Base de datos optimizada para el almacenamiento y consulta de series temporales, utilizada en aplicaciones IoT y monitoreo.

Node-RED: Herramienta basada en flujo para la integración y automatización de sistemas en entornos IoT.

EMQX: Broker de mensajería MQTT, de alto rendimiento, que facilita una comunicación publish-suscribe.

Grafana: Plataforma para la visualización y monitoreo de datos en tiempo real compatible con múltiples fuentes, incluida InfluxDB.

Documentación integral: Conjunto de subconjuntos de documentaciones, en este proyecto, documentación técnica, documentación arquitectural y documentación de negocio.

Docker: Plataforma de contenedores, principalmente para ejecutar aplicaciones contenerizadas.

Docker compose: Herramienta que permite ejecutar aplicaciones multi-contenedor, mediante archivos YAML.

YAML: Formato de serialización de datos legible por humanos, utilizado para la configuración de aplicaciones, definición de infraestructura como código y almacenamiento de datos estructurados.

JSON: Formato de intercambio de datos que utiliza texto plano y estructurado en clave-valor.

AMQP: Protocolo de mensajería avanzado basado en colas, utilizado como alternativa a MQTT para sistemas que requieren mayor confiabilidad y control en la entrega de mensajes.

API REST: Interfaz de programación que permite la comunicación entre aplicaciones mediante el uso de operaciones HTTP estándar como GET, POST, PUT y DELETE.

CI/CD: Conjunto de prácticas de desarrollo de software que permiten la integración continua del código y su despliegue automático de forma segura y controlada.

Cloud Deployment: Proceso de puesta en marcha de una solución de software en servidores remotos, generalmente ofrecidos por plataformas como Google Cloud, AWS o Azure.

Docker Hub: Registro centralizado de imágenes Docker, que permite almacenar, compartir y reutilizar contenedores listos para usar en distintos entornos.

Google Cloud Platform (GCP): Conjunto de servicios en la nube de Google, utilizado para crear, desplegar y escalar aplicaciones mediante infraestructura virtualizada.

JSON Template: Plantilla en formato JSON que define la estructura esperada de los mensajes simulados o enviados, útil en procesos de integración y simulación.

mkdocs: Herramienta estática de documentación que permite generar sitios web navegables a partir de archivos Markdown.

Mocker: Plataforma de simulación de sensores que permite generar y enviar datos sintéticos a través de protocolos como MQTT o AMQP, útil para pruebas sin hardware físico.

PostgreSQL: Sistema de gestión de bases de datos relacional de código abierto, conocido por su robustez, soporte de tipos avanzados y extensibilidad.

React: Biblioteca de JavaScript para construir interfaces de usuario interactivas y reactivas, desarrollada por Facebook.

Spring Boot: Framework de desarrollo basado en Java que simplifica la creación de aplicaciones backend al ofrecer configuración automática y estructuras listas para producción.

Telegraf: Agente de recopilación de métricas utilizado para recolectar, procesar y enviar datos a bases como InfluxDB.

VM (Virtual Machine): Entorno computacional virtualizado que simula el hardware de una computadora física para ejecutar sistemas operativos y aplicaciones de forma aislada.

YAML Schema: Definición estructurada en formato YAML que especifica la forma, tipos de datos y restricciones de archivos de configuración o simulación.

Broker: Entidad intermediaria en una arquitectura publish-subscribe (como MQTT o AMQP), que recibe mensajes de los publicadores y los distribuye a los suscriptores.

Endpoint: Punto de acceso a una API donde se puede enviar o recibir información, generalmente asociado a una ruta URL y un método HTTP.

Microservicio: Estilo arquitectónico que divide una aplicación en múltiples servicios pequeños, autónomos y especializados, que se comunican entre sí mediante APIs.

Frontend: Parte del sistema que interactúa con el usuario final, generalmente construida con tecnologías web como HTML, CSS y JavaScript.

Backend: Parte del sistema que gestiona la lógica de negocio, el acceso a datos y la comunicación entre servicios, ejecutándose en el servidor.

Despliegue automatizado: Proceso mediante el cual una aplicación es instalada y configurada automáticamente en un entorno, normalmente usando scripts o herramientas de orquestación como Docker Compose o CI/CD.

Docker Compose File: Archivo YAML que define y coordina múltiples contenedores de Docker como un solo servicio.

SSH (Secure Shell): Protocolo de red que permite acceder de forma segura a máquinas remotas para administración o ejecución de comandos.

Snapshot: Copia puntual del estado de un disco o sistema, utilizada para respaldos o restauraciones rápidas, especialmente en entornos cloud.

Service Tag: Etiqueta asociada a una instancia de VM o recurso en la nube, usada para aplicar reglas de firewall, monitoreo o agrupación lógica.

Firewall Rule: Regla que define qué tráfico de red está permitido o bloqueado hacia/desde un recurso específico (como una VM en GCP).

INTRODUCCIÓN

El Internet de las cosas (IoT) se refiere a una red de dispositivos físicos, vehículos y otros objetos físicos que están integrados con sensores, software y conectividad de red integrados, lo que les permite recopilar y compartir datos[1]. Los tipos de aplicaciones que se pueden realizar con IoT son extensos, principalmente utilizados en sectores como la industria, la agricultura, la medicina, ciudades inteligentes, vehículos inteligentes y la domótica.

En un contexto general un sistema IoT se utiliza para analizar, administrar o supervisar el estado de un objetivo físico, entre los cuales se puede tratar una variada lista de parámetros y datos tales como: calidad del aire, consumo de energía, temperatura, humedad, posición, presión, tracción, entre muchos otros.

Smart Campus es una plataforma que permite la comunicación y construcción de aplicaciones basadas en Internet de las Cosas (IoT), su principal función es modernizar los diferentes espacios de la universidad, mediante la obtención de datos del ambiente con uso de sensores, procesamiento de los mismos y posteriomente generararación de información relevante. Esta plataforma es usada como base de investigación, la cual se ha extendido técnica y funcionalmente, por proyectos de grado e investigaciones de maestría. Su arquitectura se ha visto expuesta a la erosión y degradación arquitectural, "el deterioro general de la calidad de ingeniería de un sistema de software durante su evolución"[2]. Por esta razón se plantea el objetivo de reestructurar la arquitectura de la plataforma.

---

[1] IBM. *Internet of Things (IoT)*. 2025. URL: `https://www.ibm.com/mx-es/topics/internet-of-things`.

[2] Lakshitha DE SILVA and Dharini BALASUBRAMANIAM. “Controlling software architecture erosion: A survey”. In: *Journal of Systems and Software* 85.1 (2012). Dynamic Analysis and Testing of Embedded Software, pp. 132–151. DOI: `https://doi.org/10.1016/j.jss.2011.07.036`.

Hoy en día, todo modelo software se deteriora funcionalmente en el tiempo, una forma de contrarrestarlo es mantener una sólida documentación, facilitando la comprensión, el mantenimiento y evolución del sistema. "La palabra documentar se asocia al proceso de elaborar un documento, o conjunto de documentos, que tienen como propósito comunicar información relevante de un proceso, producto o entidad"[3]. En este proyecto, el término documentación integral se utilizará para referirse a la combinación de documentación técnica, arquitectural y de negocio, elementos esenciales para garantizar la sostenibilidad del sistema.

Durante el proyecto se realizó un estudio general del sistema y de su arquitectura actual, se hará una reestructuración de la arquitectura, mejor orientada a los sistemas IoT y se presentará una implementación de la nueva arquitectura. Este estudio no solo permitirá mejorar la arquitectura de Smart Campus UIS, sino que también establecerá un marco de documentación integral que facilitará su evolución y mantenimiento.

[3] Humberto CERVANTES MACEDA; PERLA VELASCO, and LUIS CASTRO. *Arquitectura de Software.* 2016.

1. PLANTEAMIENTO Y JUSTIFICACIÓN DEL PROBLEMA

En esta sección se plantea los principales problemas que se planean resolver, y una justificación del porqué deberían mejorarsen.

La plataforma Smart Campus UIS posee dos problemas principales. En primer lugar, su arquitectura actual no presenta las tecnologías adecuadas, principalmente porque no es escalable para manejar grandes cantidades de datos, es decir, funciona adecuadamente para una cantidad limitada de sensores, pero cuando se piensa en escalar a cientos o miles de sensores esto se traduce a millones de datos en el día, la arquitectura debe ser especializada, optimizada y escalable para manejar desde uno a miles de datos por minuto y persistir altos volúmenes de información temporales, por ende, es necesaria una reestructuración arquitectural de la plataforma. Como menciona Roldán (2015), los cambios en las arquitecturas de software pueden originarse por diversos factores, como la redefinición de requerimientos, cambios tecnológicos, errores previos o incluso nuevas legislaciones. [4]

En segundo lugar, no existe documentación de la plataforma, esto genera distintos problemas, principalmente en la capa técnica, ya que la evolución del software se ve agravada por la gestión de requisitos y el análisis del sistema, la falta de documentación dificulta la comprensión del estado actual de la plataforma y puede llevar a interpretaciones erróneas. Por esto, es necesario realizar una documentación técnica, arquitectural y de negocio, lo que se constituye en este proyecto como documentación integral. "Pese a haber diseñado la mejor arquitectura, este trabajo puede ser poco productivo si personas como los diseñadores y desarrolladores

[4] María Luciana ROLDÁN; Silvio GONNET, and Horacio LEONE. “Representación de la Evolución y Refactoring de Arquitecturas de Software mediante la Aplicación y Captura de Operaciones Arquitectónicas”. In: *Revista Tecnología y Ciencia* 27 (2015), pp. 197–213.

del sistema tienen problemas al realizar sus tareas porque no existe documentación que la describa".[5]

1.1. Caso de estudio

El caso de estudio se centra principalmente en una red de dispositivos distribuidos en el campus de la Universidad Industrial de Santander con el fin de modernizar los diferentes espacios del campus, ofreciendo soluciones como: apagar aires acondicionados a partir de cierta hora, o temperatura, conocer la húmedad de diferentes espacios, controlar la luz con sensores en espacios inhabitados, entre otros. Generalmente estos dispositivos se consideran sensores y accionadores, aunque pueden ser una gran variedad de otros dispositivos. El sistema cuenta con la capacidad de comunicar y compartir datos en tiempo real a una capa especializada, capaz de recibir estos datos, procesarlos y almacenarlos de forma óptima.

En conclusión de esta sección se considera que Smart Campus contiene problemas de erosión de la arquitectura, y problemas con la extensibilidad de la plataforma, a su vez, se estima que los objetivos posteriormente propuestos se alinean con la solución de los problemas propuestos.

---

[5] Jenny Ruiz de la PEÑA and Oscar Aguilera CRUZ. "Importancia de la Ingeniería de Software en la producción de software". In: *Ciencias Holguín* 13.2 (2007), pp. 1–8.

2. OBJETIVOS

Objetivo General

- Reestructurar arquitecturalmente la plataforma Smart Campus UIS, bajo un análisis previo, que organice y facilite su respectiva evolución.

Objetivos Específicos

- Definir la estructura de la arquitectura base de la plataforma Smart Campus UIS.
- Documentar integralmente la plataforma incluyendo su arquitectura y diferentes componentes.
- Refactorizar si es necesario uno o diferentes componentes de la plataforma para mantener la coherencia de la misma.
- Definir un conjunto de reglas de extensión que permitan la evolución de la plataforma a futuro.

3. ESTADO DEL ARTE

En este apartado se presenta un estudio actualizado de las arquitecturas más utilizadas en sistemas IoT, basándose en aplicaciones similares al caso de estudio propuesto, con el objetivo de identificar enfoques y tecnologías relevantes para el desarrollo de Smart Campus UIS. Se analizan diversas soluciones, se comparan sus características y se evalúa su viabilidad en función de los objetivos del proyecto, permitiendo fundamentar la elección de una arquitectura adecuada.

3.1. M.I.N.G Stack

Actualmente en la industria IoT se ha destacado una arquitectura que ganó tracción a partir del año 2019, creada por el equipo de Balena. MING Stack hace referencia a la arquitectura compuesta por cuatro tecnologías: **MQTT, InfluxDB, Node-RED y Grafana**, de código abierto. Esta pila busca brindar una solución escalable para abordar el gran volumen de datos en tiempo real generados por dispositivos IoT, permitiendo su recolección, almacenamiento, análisis y visualización de manera eficiente[6].

- Orientación a datos en tiempo real.
- Componentes 100% de código abierto.
- Fácil integración y despliegue en entornos distribuidos.
- Visualización amigable de datos históricos y actuales.

[6] Vincent CHOSEN. *The MING Stack: What It Is and How It Works.* 2024. URL: `https://www.influxdata.com/blog/-ming-stack-introduction-influxdb/`.

3.2. T.I.C.K Stack

TICK Stack es una arquitectura desarrollada por InfluxData, conformada por **Telegraf, InfluxDB, Chronograf y Kapacitor**. Está pensada para sistemas de monitoreo y análisis de datos en tiempo real, ofreciendo una solución robusta para aplicaciones IoT donde el procesamiento y la gestión eficiente de grandes volúmenes de datos es crucial[7].

- Recolección flexible de datos con Telegraf.
- Almacenamiento optimizado para series temporales (InfluxDB).
- Visualización con Chronograf y procesamiento de eventos con Kapacitor.
- Alto rendimiento y adaptabilidad en entornos industriales.

3.3. Arquitectura EDA (Event-Driven Architecture)

La arquitectura orientada a eventos (EDA, por sus siglas en inglés) es un paradigma de diseño en el cual los componentes del sistema se comunican mediante eventos asincrónicos. Este enfoque es altamente adecuado para entornos IoT, donde dispositivos remotos generan eventos de forma constante. En una EDA típica, los gateways publican datos hacia un *broker de mensajería*, y diferentes servicios suscriptores los procesan de manera desacoplada.

- Fuerte desacoplamiento entre productores y consumidores.
- Alta escalabilidad y tolerancia a fallos.
- Integración natural con brokers MQTT o AMQP.
- Arquitectura reactiva y flexible, ideal para entornos dinámicos.

---

[7] Gunnar AASEN. *Introduction to InfluxData's InfluxDB and TICK Stack*. 2017. URL: `https://www.influxdata.com/blog/introduction-to-influxdatas-influxdb-and-tick-stack/`.

3.4. NETIG Stack

Como resultado del análisis de las arquitecturas MING, TICK y EDA, se propone una arquitectura híbrida denominada **NETIG**, la cual combina lo mejor de cada enfoque, incorporando un **broker MQTT** especializado para entornos IoT. Esta pila está conformada por:

- **Node-RED**: Motor visual de flujos que permite la creación rápida de lógica de comunicación entre gateways y la capa de datos.
- **EMQX**: Broker MQTT altamente escalable, encargado de recibir los datos de gateways físicos o simulados.
- **Telegraf**: Agente encargado de recolectar y transformar los datos, enviándolos a la base de datos.
- **InfluxDB**: Base de datos de series temporales, optimizada para almacenar datos generados por sensores IoT.
- **Grafana**: Herramienta de visualización que permite explorar y analizar los datos en tiempo real o históricos.

Este stack fue seleccionado por su rendimiento, escalabilidad, facilidad de integración con diferentes fuentes de datos y su naturaleza completamente OpenSource. NETIG representa una arquitectura optimizada y moderna que responde directamente a las necesidades de la plataforma Smart Campus UIS.

En conclusión, el análisis de las arquitecturas más utilizadas en el ámbito IoT permitió identificar las fortalezas y limitaciones de cada enfoque. La pila MING destaca por su simplicidad y visualización, TICK por su procesamiento de eventos, y EDA por su enfoque escalable y desacoplado.

Con base en este estudio, se concluye que una solución integral, que combine una arquitectura híbrida optimizada para IoT con una documentación estructurada y navegable, es el camino más adecuado para el caso de estudio. Por esta razón, se diseñó e implementó una arquitectura basada en los principios del stack NETIG (Node-RED, EMQX, Telegraf, InfluxDB, Grafana), complementada con una documentación integral construida mediante la herramienta **MkDocs**.

Esta combinación no solo garantiza el rendimiento, escalabilidad y flexibilidad tecnológica del sistema, sino que también proporciona una guía clara y accesible para su comprensión, mantenimiento y evolución futura, mitigando así los efectos de la erosión arquitectónica en el tiempo.

4. MARCO DE REFERENCIA

El presente capítulo expone los fundamentos teóricos y tecnológicos que sustentan el desarrollo del proyecto. Se abordan los conceptos clave de arquitectura de software, control de versiones, tecnologías de virtualización y estandarización, así como las herramientas seleccionadas que permiten garantizar eficiencia, escalabilidad y mantenibilidad del sistema propuesto. Además, se justifica la elección de determinadas tecnologías como bases de datos, protocolos de comunicación y brokers especializados, que resultan especialmente adecuadas para entornos de tipo IoT y arquitecturas distribuidas.

4.1. ARQUITECTURA DE SOFTWARE

La arquitectura de software de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema. Comprende elementos de software, relaciones entre ellos, y propiedades de ambos[8]. Una arquitectura abarca diferentes capas o niveles que definen cada componente, entre ellos principalmente:

4.2. DOCUMENTACIÓN INTEGRAL.

La documentación integral en el software es fundamental porque proporciona un marco claro y accesible para comprender, mantener y evolucionar el sistema de forma eficiente. Una buena documentación asegura que los desarrolladores comprendan la arquitectura, el código y los requisitos, lo que facilita la resolución de problemas y la incorporación de mejoras.

---

[8] Paul C CLEMENTS. “Software architecture in practice”. In: *Diss. Software Engineering Institute* (2002).

4.3. DOCKER.

Docker es una plataforma de software que permite la creación, implementación y ejecución de aplicaciones en contenedores. Los contenedores son entornos aislados que incluyen lo necesario para que una aplicación se ejecute de manera independiente, como bibliotecas, dependencias y el código de la aplicación. Docker simplifica el desarrollo, prueba y despliegue de software, proporcionando un enfoque eficiente y escalable para administrar aplicaciones en entornos heterogéneos. A diferencia de las máquinas virtuales tradicionales, que requieren un sistema operativo completo, los contenedores de Docker comparten el núcleo del sistema operativo del anfitrión, lo que reduce significativamente el consumo de recursos y mejora el rendimiento. Docker permite a los desarrolladores empaquetar aplicaciones de manera consistente, asegurando que se ejecutarán de la misma forma, independientemente del entorno en el que se implementen.

4.3.1. Docker compose. Es una herramienta que permite definir y gestionar aplicaciones con múltiples contenedores de manera sencilla. A través de un archivo de configuración en formato YAML, Docker Compose permite especificar cómo deben construirse, conectarse y ejecutarse varios contenedores que forman parte de una misma aplicación. Esto es particularmente útil en entornos de microservicios o arquitecturas modulares, donde distintos servicios necesitan comunicarse entre sí de manera eficiente.

4.3.2. Estandarización docker-compose. Es la creación de un conjunto de prácticas, convenciones y estructuras comunes para definir y gestionar contenedores dentro de un entorno de desarrollo o producción utilizando Docker Compose. Esta estandarización asegura que los miembros de un equipo sigan las mismas reglas para la configuración y despliegue de contenedores, lo que facilita la colaboración, el mantenimiento y la escalabilidad del sistema. Mediante la estandarización de los archivos docker-compose.yml, se logra uniformidad en la forma en que los servicios se definen, interconectan y despliegan en diferentes entornos (desarrollo, pruebas, producción). Esto ayuda a prevenir inconsistencias y errores, y permite

que el proceso de despliegue sea reproducible en cualquier entorno.

Ventajas de la estandarización de Docker Compose:

- **Configuración consistente:** Los desarrolladores y sistemas utilizan la misma configuración para los servicios, lo que reduce errores y facilita la gestión.
- **Facilidad de despliegue:** La estandarización permite que el despliegue de los servicios sea automatizado y reproducible, sin depender de configuraciones manuales o específicas de un entorno.
- **Mantenibilidad:** Con una configuración estandarizada, es más fácil actualizar los servicios o añadir nuevos sin romper la configuración existente..

4.4. JAVA SPRINGBOOT.

Es un framework basado en Java que facilita la creación de aplicaciones empresariales robustas y escalables. Se construye sobre el popular framework Spring, pero con la ventaja de proporcionar una configuración automática y simplificada, lo que permite a los desarrolladores enfocarse más en la lógica de negocio en lugar de en la configuración compleja. Spring Boot se destaca por sus capacidades de creación rápida de aplicaciones, ya que viene con configuraciones predeterminadas y listas para usar.

Uno de los principales beneficios de Spring Boot es su capacidad para permitir la creación de aplicaciones "stand-alone" con un servidor embebido, como Tomcat o Jetty, lo que elimina la necesidad de configurar un servidor web externo. Además, proporciona una amplia integración con otras tecnologías, lo que facilita la conexión a bases de datos, el uso de APIs REST y la gestión de la seguridad.

4.5. CONTROL DE VERSIONES Y PERSISTENCIA DEL CÓDIGO.

4.5.1. Git.

Es un sistema de control de versiones distribuido que permite a los desarrolladores gestionar el historial de cambios en su código de manera eficiente. A diferencia de los sistemas de control de versiones centralizados, Git almacena una copia completa del repositorio en cada máquina de los desarrolladores, lo que proporciona mayor seguridad y flexibilidad al trabajar en equipos distribuidos o sin conexión. Git permite rastrear modificaciones, revertir cambios, crear ramas de desarrollo y fusionar diferentes versiones del código, lo que lo convierte en una herramienta esencial para la colaboración en proyectos de software.

4.5.2. Git-Flow.

Es un modelo de branching (ramificación) que organiza el desarrollo de software en Git a través de un conjunto de prácticas recomendadas. Este flujo de trabajo introduce una estructura clara con ramas dedicadas para el desarrollo de nuevas funcionalidades, la corrección de errores y la liberación de versiones, lo que facilita el trabajo en equipo y la gestión de versiones en proyectos grandes y complejos.
También es importante seguir una estructura de conventional commits, para llevar una documentación y regsitro más unificado de las implementaciones a lo largo del tiempo.

4.6. TECNOLOGÍAS ÓPTIMAS.

Durante la investigación se hizo necesario mostrar alternativas más viables para la arquitectura del proyecto por lo mismo, se proponen las siguientes tecnologías con su respectivas justificación:

4.6.1. Influx DB.

Es una base de datos de series temporales diseñada específicamente para manejar grandes volúmenes de datos que cambian con el tiempo. Esto la hace particularmente adecuada para aplicaciones de Internet de las Cosas (IoT), donde los dispositivos generan datos de sensores de forma continua y en intervalos regulares.

Características principales:

- **Optimización para Datos de Series Temporales:** InfluxDB está diseñada para almacenar datos basados en el tiempo, como las lecturas de sensores IoT, de manera eficiente.
- **Escalabilidad:** Soporta grandes volúmenes de datos, lo que es esencial cuando se manejan múltiples dispositivos IoT en tiempo real.
- **Consultas en Tiempo Real:** InfluxQL, el lenguaje de consulta de InfluxDB, permite ejecutar consultas complejas y análisis de datos en tiempo real. Esto es útil en aplicaciones IoT que requieren monitoreo constante o alertas basadas en eventos.

4.6.2. MQTT (*Message Queuing Telemetry Transport*).

Es un protocolo de mensajería ligero diseñado para la comunicación entre dispositivos en redes de baja capacidad o dispositivos con recursos limitados, como los que se encuentran en entornos de Internet de las Cosas (IoT).

4.6.3. Brokers de mensajeria MQTT.

Cuando se habla de una sola interacción se habla de una comunicación directa. Sin embargo, analizando un esquema a futuro, donde hayan cientos de request de publicadores de gateways de altas cantidades de sensores se convierte en un problema,por eso, se hace necesario implementar un broker de mensajeria para gestionar el tráfico y optimizar el ambiente y la comunicación, mejorando significativamente la plataforma.

4.6.4. EMQX.

Es un broker MQTT distribuido y de código abierto que está diseñado para manejar grandes volúmenes de mensajes y dispositivos conectados de manera eficiente en aplicaciones de Internet de las Cosas (IoT). Es uno de los brokers MQTT más escalables, con soporte para millones de conexiones simultáneas, lo que lo convierte en una solución popular en implementaciones industriales y empresariales.

Características clave:

- **Alto rendimiento:** EMQX puede manejar millones de conexiones concurrentes y permite el procesamiento de mensajes a gran escala, ideal para entornos IoT masivos.
- **Distribuido:** Es compatible con arquitecturas de alta disponibilidad y permite configuraciones en clústeres, lo que asegura que la plataforma pueda escalar horizontalmente.
- **Compatibilidad con varios protocolos:** Además de MQTT (v3.1, v3.1.1 y v5.0), EMQX soporta otros protocolos como CoAP, HTTP, WebSocket y LwM2M, lo que lo hace flexible para integrar diferentes tipos de dispositivos y aplicaciones.
- **Integración:** Tiene compatibilidad con bases de datos (InfluxDB, MongoDB, PostgreSQL), sistemas de almacenamiento en la nube (AWS, Google Cloud), y plataformas de procesamiento de datos en tiempo real como Apache Kafka.
- **Seguridad avanzada:** EMQX ofrece soporte para autenticación y autorización basadas en certificados, LDAP, JWT, y OAuth 2.0, así como cifrado TLS/SSL.

4.6.5. HiveMQ. Es un broker MQTT comercial diseñado para ofrecer una solución robusta, escalable y segura para aplicaciones empresariales IoT. Está optimizado para manejar millones de conexiones simultáneas y es altamente confiable, lo que lo hace ideal para aplicaciones de misión crítica.

Características clave:

- **Escalabilidad empresarial:** HiveMQ puede manejar millones de dispositivos y mensajes por segundo, con características específicas para entornos empresariales y escalabilidad horizontal.

- **Alta disponibilidad y clustering:** HiveMQ está diseñado para ser altamente disponible, con soporte para clustering nativo, lo que permite configuraciones distribuidas y resilientes.

- **Soporte para últimas versiones de MQTT:** HiveMQ soporta la última versión del protocolo MQTT, lo que permite el uso de nuevas funciones avanzadas, como Reason Codes, Session Expiry y flujos de control más sofisticados.

- **Integraciones avanzadas:** Proporciona integraciones listas para usar con plataformas de almacenamiento en la nube, bases de datos, Apache Kafka, y otros sistemas de procesamiento de datos.

- **Herramientas de monitoreo y administración:** HiveMQ ofrece una interfaz gráfica de usuario (GUI) para la administración y monitoreo de las conexiones, mensajes y rendimiento del sistema en tiempo real.

- **Seguridad empresarial:** Incluye funciones avanzadas de seguridad, como autenticación basada en certificados, control de acceso granular, y cifrado TLS.

- **Plugin System:** HiveMQ tiene un sistema de plugins flexible que permite personalizar y ampliar las funcionalidades del broker según las necesidades del usuario.

4.6.6. Mosquitto Es un broker MQTT ligero y de código abierto que está diseñado para implementaciones pequeñas y dispositivos embebidos, pero también puede ser utilizado en entornos empresariales y de producción. Es uno de los brokers MQTT más populares debido a su simplicidad y facilidad de implementación.

Características clave:

- **Ligero y eficiente:** Mosquitto está diseñado para ser ligero, lo que lo hace adecuado para dispositivos con recursos limitados como microcontroladores y sistemas embebidos.

- **Cumple con los estándares MQTT:** Es totalmente compatible con las versiones del protocolo MQTT, lo que garantiza que puede ser utilizado en una amplia gama de aplicaciones.

- **Fácil de instalar:** Es muy fácil de configurar y usar, lo que lo convierte en la opción favorita para desarrolladores que buscan un broker simple para pruebas y pequeñas implementaciones IoT.

- **Soporte para TLS y autenticación:** Mosquitto permite asegurar las conexiones mediante TLS/SSL y cuenta con mecanismos de autenticación simples como usuarios y contraseñas.

- **Clustering básico:** Aunque Mosquitto no está diseñado nativamente para clustering como EMQX o HiveMQ, existen configuraciones personalizadas para crear clusters o agregar alta disponibilidad con otros servicios.

- **Comunidad Open-Source:** Mosquitto es totalmente de código abierto y tiene una comunidad activa, lo que facilita encontrar soporte y ejemplos de configuración.

4.6.7. Comparación EMQX vs HiveMQ vs Mosquitto A continuación en la **tabla 1** se presentan la comparativa de las caracteristicas más relevantes de los principales brokers de mensajería MQTT.

	EMQX	***HiveMQ***	***Mosquitto***
Escalabilidad	Muy alta (clustering avanzado)	Muy alta (empresarial)	Baja (aunque configurable)
Facilidad de uso	Moderada	Alta (con GUI)	Sencilla
Protocolos soportados	MQTT, CoAP, HTPP...	Principal MQTT	Solo MQTT
Licencia	OpenSource + Enterprise	Comercial	Open Source

Tabla 1
Brokers de mensajería especializados para protocolo MQTT. *Principales características contrastadas de los brokers de mensajería EMQX, HiveMQ y Mosquitto.*

En resumen, este marco de referencia tiene como objetivo brindar una base para comprender el contexto tecnológico y las herramientas clave que de las cuales se hará uso más adelante o se implementarán.

5. METODOLOGÍA

En este apartado aborda y explica más a fondo la metodología utilizada para llevar a cabo los objetivos propuestos, presenta una explicación resumida y al final se concluye si fue apropiada o no.

La metodología empleada para el desarrollo del proyecto se basó en un enfoque ágil **metodología iterativa**, agregando los principios de **divide y vencerás** para estructurar y organizar el trabajo en ciclos iterativos e incrementales denominados **sprints**. Esta elección metodológica permitió una adaptación flexible a cambios, una evolución constante del sistema, y una validación continua de los objetivos alcanzados.

La principal idea de esta metolodogía es dividir la plataforma Smart Campus UIS como se aprecia en la **figura 1**, recorrer cada parte en un sprint, identificar mejoras y aplicar una reestructuración, anteriormente mencionado en los objetivos la reestructuración es el objetivo general, y la componen los objetivos específicos.

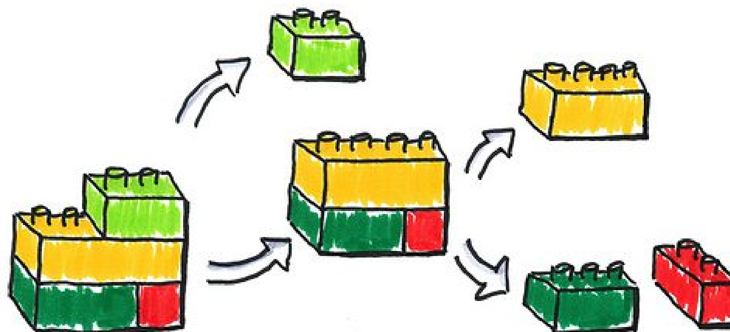

Figura 1
Divide y vencerás. *Imagen tomada de: https://ortizvivas.com/blog/divide-y-venceras*[9]

Cada sprint supone un ciclo para reestructurar una capa o lego, como se muestra en la siguiente figura.

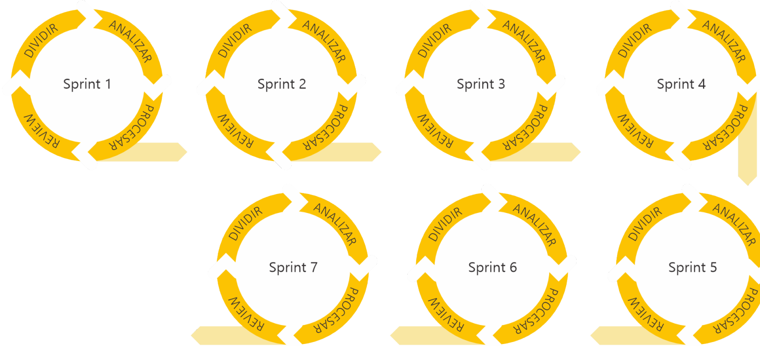

Figura 2
Ciclos sprint. *Imagen propia creada por los autores.*

5.1. Enfoque por Sprints

El desarrollo se dividió en **siete sprints**, cada uno con objetivos específicos orientados a la evolución progresiva del sistema:

- **Sprint 1:** Se realizó la división general del sistema en tres capas principales (administración, datos y gateways), y se diseñaron las arquitecturas iniciales para cada una de ellas.
- **Sprint 2:** Se profundizó en la capa de administración (AdminService), se analizaron sus tecnologías (React, Springboot, PostgreSQL) y se documentaron sus servicios REST mediante Swagger.
- **Sprint 3:** Se evaluó y reestructuró completamente la capa de datos (DataService), proponiendo una nueva arquitectura basada en el stack NETIG, más eficiente para entornos IoT.
- **Sprint 4:** Se abordó la capa externa (gateways), definiendo una estructura de mensaje JSON para promover la extensibilidad y mantener estándares.

- **Sprint 5:** Se consolidó la documentación técnica del sistema usando MkDocs, incluyendo arquitectura, APIs, despliegue y buenas prácticas.
- **Sprint 6:** Se desplegó la solución en una instancia de máquina virtual en Google Cloud Platform (GCP), validando la portabilidad y accesibilidad remota del sistema.
- **Sprint 7:** Se definió una guía con reglas de extensibilidad para nuevas implementaciones y extensiones del sistema de Smart Campus UIS.

5.2. Metodología aplicada

Cada sprint siguió una estructura de trabajo compuesta por cuatro fases principales:

1. **Dividir:** Dividir el problema, en problemas más pequeños, para analizarlos posteriormente y luego procesar acciones independientes.
2. **Analizar:** Estudio detallado de los elementos divididos, evaluación de tecnologías, identificación de problemas o mejoras, con el fin de cumplir los objetivos propuestos.
3. **Procesar:** Realización de acciones en pro de lograr los objetivos dentro de cada problema encontrado.
4. **Review:** Evaluación de resultados, verificación de objetivos alcanzados, validación del avance logrado y preparación para el siguiente sprint.

En conclusión, esta metodología permitió lograr los objetivos propuestos, de una forma eficiente, organizada, modular, granular y especifica, de modo que cada capa o pieza de lego fue trabajada de forma aislada, proporcionando una solución independiente, a su vez, la metodología aceleró los procesos, ya que al delimitar tiempos específicos para cada sprint generaba una efecto de aceleración en el equipo.

6. DESARROLLO

Esta sección compacta el desarrollo que se llevó a cabo dentro de cada sprint, los pasos que se tuvieron en cuenta, los analisis, los procesos realizado, y unas conclusiones finales de cada circuito.

6.1. SPRINT 1. DEFINICION ARQUITECTURA SMART CAMPUS UIS

6.1.1. Dividir: Inicialmente se definió graficamente la estructura que compone a la plataforma Smart Campus UIS como se aprecia en la **figura 3**, para así tomar un punto de partida y poder dividir los caminos de los siguientes sprints. **Diagrama de Arquitectura de Software.** Un diagrama de arquitectura de software es una representación gráfica o visual de la estructura de un sistema de software. Este diagrama describe los componentes principales del sistema (anteriormente mencionados), las interacciones entre ellos, y cómo se organizan en capas o módulos. Su propósito es proporcionar una visión más clara del diseño del sistema, facilitando la comprensión y lectura del sistema. Estos diagramas tienen un papel muy importante a la hora de comunicar un sentido técnico, ya sean desarrolladores, clientes, arquitectos, etc, ya que muestra información de tecnologías, módulos, componentes, protocolos, además, son sumamente útiles para identificar posibles áreas de mejora en términos de escalabilidad.

Este diagrama describe los componentes principales del sistema, las interacciones entre ellos, y cómo se organizan en capas o módulos. Su propósito es proporcionar una visión más clara del diseño del sistema, facilitando la comprensión y lectura del sistema. Estos diagramas tienen un papel muy importante a la hora de comunicar un sentido técnico, ya sean desarrolladores, clientes, arquitectos, etc, ya que muestra información de tecnologías, módulos, componentes, protocolos, además, son sumamente útiles para identificar posibles áreas de

mejora en términos de escalabilidad.

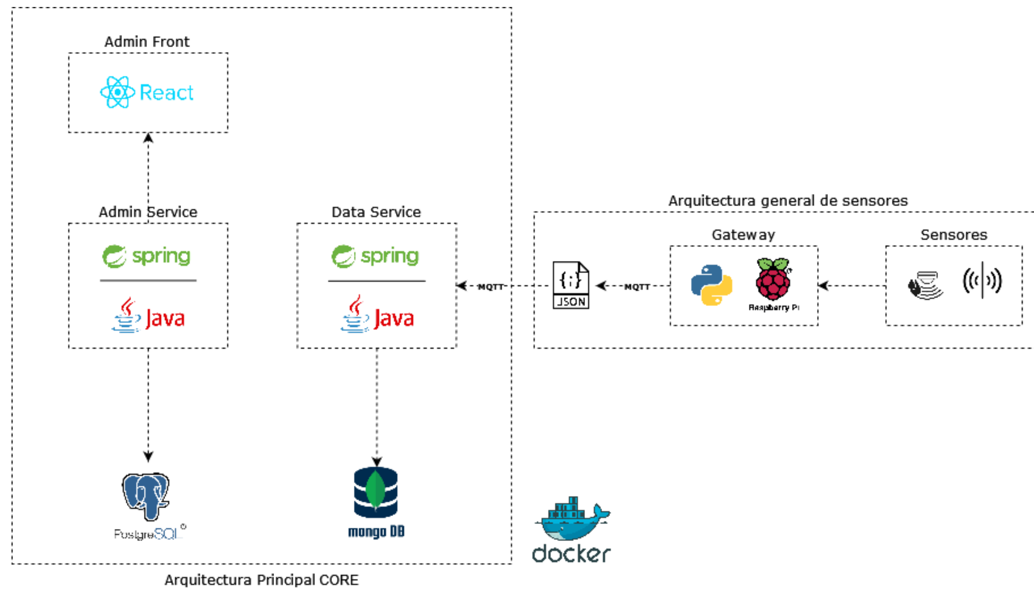

Figura 3
Arquitectura Smart Campus UIS. *Diagrama técnico sobre la arquitectura general de Smart Campus UIS*

- **Capa de presentación:** La interfaz de usuario o front-end, que interactúa con el usuario final.
- **Capa de servicios (*Back-End*):** Los servicios que contienen la lógica de negocio.
- **Capa de persistencia:** El acceso a bases de datos y otros servicios de almacenamiento.
- **Servicios externos (*APIs*):** Servicios de terceros o APIs con los que interactúa el sistema.
- **Comunicación entre componentes:** Especifica comunmente los protocolos de comunicación entre las capas y los servicios APIs.

Se consideraron los objetivos propuestos, y se concretó dividir el sistema SmartCampusUIS por 3 diferentes capas, **la capa de administración (AdminService),**

la capa de datos (DataService), y la capa de Gateway (ExternalLayer), esto con el fin de aislar los procedimientos y tener soluciones personalizadas para cada capa.

6.1.2. Analizar: En este paso se analizaron las diferentes capas ya divididas, y se entra a hacer una investigación superficial de cada capa.

- **Capa administración (*AdminService*):** Esta capa está orientada a brindar servicios de administración, principalmente para llevar un inventario de los gateways, modelos, dispositivos, de un sistema IoT definido. La arquitectura para está capa está definida de la siguiente manera, para el front se utiliza **React**, para servicios backend REST se utiliza **SpringBoot** y para la persistencia de los datos se utiliza **PostgreSQL**. **Tal como se observa en la figura 4.**

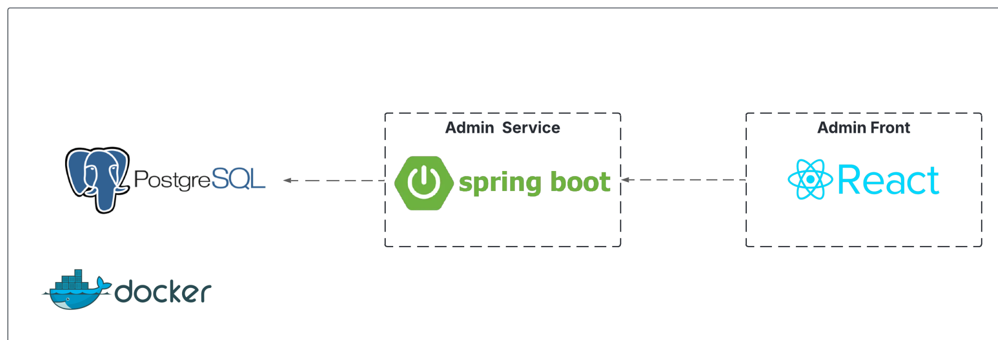

Figura 4
Capa de administración Smart Campus UIS. *Se elabora el diagrama arquitectural inicial de la capa de administración de la plataforma Smart Campus UIS.*

- **Capa de datos (*DataService*):** Esta capa se encarga recibir información que proviene de diferentes gateways, los procesa y almacena en un base de datos MongoDB. La arquitectura para esta capa está definida de la siguiente manera, para el back o servicios REST se utiliza **SpringBoot** y para la persisencia de datos se utiliza **MongoDB**. Tal como se observa en la figura 5.

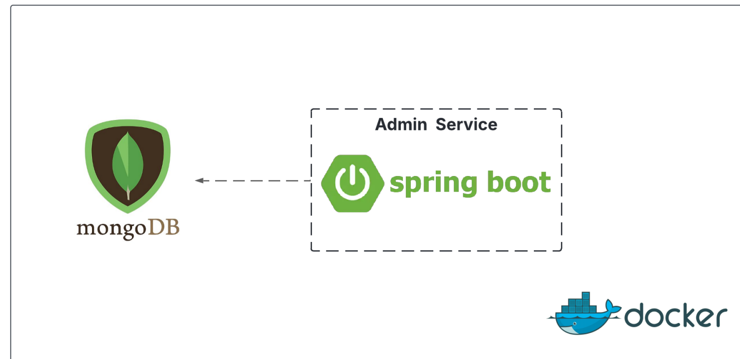

Figura 5
Capa de datos Smart Campus UIS. *Se elabora el diagrama arquitectural inicial de la capa de datos de la plataforma Smart Campus UIS.*

- **Capa Gateway (*External-Layer*):** Esta capa es dinámica, puede ser uno o un conjunto de gateways que envian información a la capa de datos, no se tiene una arquitectura definida, ya que es externo al sistema e independiente, la única dependencia existente es una estructura con campos definidos, tipo JSON para comunicarse de forma efectiva con el DataService. Tal como se observa en la figura 6.

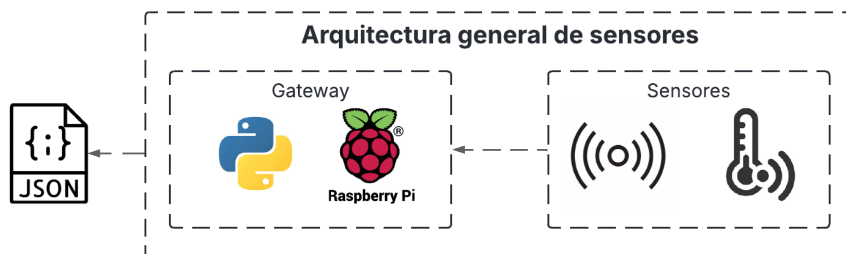

Figura 6
Capa Gateway o externa de Smart Campus UIS. *Se elabora el diagrama arquitectural inicial de la capa de gateways, plasmando una arquitectura general de los gateways.*

6.1.3. Procesar: Luego de dividir y analizar, se planean los siguientes sprints de la siguiente forma:

- **Sprint 2: Capa de AdminService**

- **Sprint 3: Capa de DataService**
- **Sprint 4: Capa externa**
- **Sprint 5: Documentación Mkdocs**
- **Sprint 6: Desplieguie Cloud**
- **Sprint 7: Reglas de extensión**

6.1.4. Review: Durante este primer sprint se logró dividir el sistema Smart Campus UIS en tres capas principales: administración, datos y gateways. Esta división permite establecer una arquitectura más clara y entendible, que facilita tanto el desarrollo individual de cada componente como su mantenimiento futuro. También se diseñaron los diagramas arquitecturales iniciales que servirán como guía para su implementación. A su vez se definió un plan de acción para los sigiuentes sprints. El avance logrado en este sprint proporciona una base sólida para la evolución del sistema en los siguientes ciclos.

6.2. SPRINT 2. DOCUMENTACIÓN SWAGGER CAPA DE ADMINISTRACIÓN

6.2.1. Dividir: En este segundo sprint se contaba con 3 capas divididas, **la capa de administración (AdminService), la capa de datos (DataService), y la capa de Gateway (ExternalLayer)**. Para este sprint se decidió abordar **la capa de administración (AdminService)**. En este paso se separó la capa por tecnologías, es decir, se analizaron individualmente las implementaciones de cada tecnología. Tal como se observa en la figura 7.

Figura 7
Tecnologías de la capa de administración Smart Campus UIS. *Diagrama de cada tecnología involucrada en la capa de administración.*

6.2.2. Analizar: Se profundizó cada tecnología por aparte, logrando un análisis independiente con soluciones independientes.

- **React:** La interfaz de administración está implementada con React, se realizó un análisis del código, no se encontraron oportunidades de mejora, la estructura y composición eran claras. Ver en la figura 8.

Figura 8
Código del front de la capa de Administración. *Organización clara y limpia del front.*

- **Springboot:** El back de la capa de administración está implementado con Springboot, se realizó un análisis del código, en búsqueda de cumplir los objetivos, y para este caso se encontraba nula documentación, a partir de lo concebido, se decidió realizar una documentación a nivel del código REST, mediante swagger. Ver en la figura 9.

Figura 9
Código del back de la capa de Administración. *Nula documentación en los servicios REST de los controladores.*

- **PostgreSQL:** Para la persistencia de esta capa se integra con PostgreSQL, en este caso se comparó las tablas y columnas con estándares de PostgreSQL y no se encontraron diferencias, haciendo caso omiso para acciones más adelante dentro del sprint para la base de datos.
  - Uso de claves primarias bien definidas para cada entidad.
  - Nombres de tablas y columnas en minúsculas y con snake_case, siguiendo la convención PostgreSQL.
  - Tipos de datos apropiados según el tipo de información almacenada (por ejemplo, ‘INTEGER‘, ‘VARCHAR‘, ‘TIMESTAMP‘, ‘BOOLEAN‘).
  - Definición de restricciones ‘NOT NULL‘ para campos obligatorios.
  - Relaciones entre tablas bien modeladas mediante claves foráneas (‘FOREIGN KEY‘).

– Índices definidos sobre campos clave para mejorar el rendimiento de las consultas.

6.2.3. Procesar:

En este paso se llevó a cabo las notas del anterior paso, **documentar con swagger los servicios REST del back de la capa de administración.** Para ello fue necesario;

- Agregar dependencia al pom.xml

```
<dependency>
<groupId>org.springdoc</groupId>
<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
<version>2.2.0</version>
</dependency>
```

- Acceder a la UI de Swagger

```
http://localhost:8080/swagger-ui.html
```

- Por último, se documentó los diferentes servicios REST de cada controller. Ver ejemplo en la figura 10.

```java
package com.iot.admin.admin.controller;

import java.util.List;
import java.util.Map;

import javax.validation.Valid;

import com.iot.admin.admin.dto.DeviceDetails;
import com.iot.admin.admin.dto.DeviceForm;
import com.iot.admin.admin.dto.DeviceResourcePropertyForm;
import com.iot.admin.admin.dto.PropertyDetails;
import com.iot.admin.admin.dto.PropertyForm;
import com.iot.admin.admin.service.DeviceService;
import com.iot.admin.admin.utils.Pagination;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("device")
public class DeviceController {

    @Autowired
    private DeviceService service;

    @GetMapping
    public List<DeviceDetails> list(){
        return service.findAll();
    }

    @Operation(summary = "Listar dispositivos", description = "Obtiene una lista de todos los dispositivos registrados")
    @GetMapping("pageable")
    public Map<String,Object> page_list(@RequestParam Map<String,String> params){
        Page<DeviceDetails> data = service.paginate(params);
        Map<String,Object> result= Pagination.mapPage(data);
        return result;
    }

    @Operation(summary = "Registrar dispositivo", description = "Crea un nuevo dispositivo en el sistema")
    @PostMapping
    @ResponseStatus(code = HttpStatus.CREATED)
    public DeviceDetails create(@RequestBody @Valid DeviceForm data){
        return service.create(data);
    }
```

Figura 10

Documentación Swagger de servicios REST. *Se realiza la respectiva documentación de cada servicio.*

6.2.4. Review: Durante este sprint se abordó en profundidad la capa de administración del sistema Smart Campus UIS. Se analizaron las tecnologías que la componen (React, Springboot y PostgreSQL), encontrando una correcta implementación general, pero evidenciando la ausencia de documentación en los servicios REST del backend. Como resultado, se integró Swagger en el proyecto y se documentaron los distintos endpoints, lo cual mejora la mantenibilidad del sistema y facilita su uso de otros desarrolladores. Esta documentación mejora las futuras integraciones, y representa un paso importante hacia la consolidación de una arquitectura limpia, entendible

y estandarizada.

6.3. SPRINT 3. REFACTORIZACIÓN CAPA DE DATOS

6.3.1. Dividir: En este tercer sprint, se profundizó en la capa de datos (DataService), éste módulo es bastante crítico, ya que es el corazón y mente del sistema. En este paso se separó la capa por tecnologías, es decir, se analizó individualmente las implementaciones de cada tecnología. Ver en la figura 11.

Figura 11
Tecnologías de la capa de datos Smart Campus UIS. *Diagrama de cada tecnología involucrada en la capa de datos.*

6.3.2. Analizar: Esta capa o componente es muy importante, ya que es la principal encargada de procesar grandes cantidades de datos, en tiempo real, y detallar series temporales.

Se investigó sobre varias arquitecturas IoT en la industria, entre las que destacan MING y TICK, contando MING con un entorno más amigable y robusto que TICK. En base a estos stacks tecnologicos, ambos carecían de un broker de mensajería MQTT, y con unas necesidades más específicas para el caso de estudio, se implementó lo mejor de cada pila en función de optimizar la arquitectura hacia el caso de estudio. Al final se obtuvo el siguiente stack llamado: NETIG.

- **NodeRED(*N*):** no está diseñado específicamente para manejar series de tiempo de manera eficiente, lo que es fundamental en IoT donde los sensores generan datos

constantemente.

- **EMQX($\boldsymbol{E}$):** Es el broker de mensajería, diseñado para usar MQTT como protocolo principal, una tecnología bastante robusta, escalable y diseñada para ambientes IoT. Permite, facilita y controla la comunicación de la capa externa o gateways, con la capa de datos.

- **Telegraf($\boldsymbol{T}$):** Es un agente de datos, fabricado por InfluxData(creadores de InfluxDB), se encarga de recopilar, procesar y enviar datos de una fuente hacia la base de datos, facilitando y optimizando la comunicación hacía la base de datos.

- **InfluxDB($\boldsymbol{I}$):** Es una base de datos, diseñada especialmente para almacenar datos de series de tiempo, garantizando un alto rendimiento en la persistencia y consulta de datos generados por los dispositivos IoT.

- **Grafana($\boldsymbol{G}$):** Plataforma de visualización que permite analizar los datos históricos y en tiempo real ofreciendo una interfaz amigable e intuitiva, facilitando la toma de decisiones basada en los datos recopilados.

¿Por qué InfluxDB sobre MongoDB? Es una base de datos no relacional, poderosa y flexible, pero su aquitectura y forma de leer los datos la convierten en una mala elección para sistemas IoT. A continuación las principales razones por las que MongoDB no es completamente óptimo para procesamientos IoT:

- MongoDB no está diseñado específicamente para manejar series de tiempo de manera eficiente, lo que es fundamental en IoT donde los sensores generan datos constantemente.

- MongoDB consume mucha RAM para indexar y almacenar datos, lo que puede ser un problema en entornos IoT con recursos limitados.

- La replicación y el manejo de grandes volúmenes de datos pueden volverse costosos en términos de almacenamiento y rendimiento a medida que el número de dispositivos IoT

crece.

- Para consultas de datos históricos en IoT, bases de datos optimizadas para series de tiempo como InfluxDB o TimescaleDB son mucho más eficientes.

Según el siguiente articulo "InfluxDB vs MongoDB"[10] se compara a detalle el uso de ambas bases de datos en ambientes IoT, en las que concluye la importancia de una base de datos especializada para IoT como lo es InfluxDB.

Según este otro documento de una conferencia publicada en IEEE-Xplore, llamado **Comparative Analysis of MongoDB and InfluxDB for Time Series Data Management in IoT Environments: A Study on Performance, Scalability, and Concurrency**[11], compara detalladamente enfocando principalmente aspectos sobre recursos, tales como: tiempos de respuesta, consumo de cpu, ram, memoria, entre otros.

En conclusión, siguiendo estas dos referencias se concluye que influxDB es mucho más adecuado que MongoDB para el caso de estudio propuesto.

¿Por qué eliminar Springboot? Es una tecnología demasiado compleja y robusta, perfecta para aplicaciones empresariales personalizadas, pero cuando se trata de entornos IoT, es una pésima opción en cuanto a rendimiento y escalabilidad, generando grandes pérdidas de recursos. A continuación las principales razones por las que Springboot no es completamente óptimo para procesamientos IoT:

---

[10] INFLUXDATA. *InfluxDB vs MongoDB*. 2024. URL: `https://www.influxdata.com/comparison/influxdb-vs-mongodb/`.

[11] Piyush TRIPATHI; Mahdi H. MIRAZ, and Snigdha JOSHI. “Comparative Analysis of MongoDB and InfluxDB for Time Series Data Management in IoT Environments: A Study on Performance, Scalability, and Concurrency”. In: *2023 International Conference on Computing, Networking, Telecommunications Engineering Sciences Applications (CoNTESA)*. 2023, pp. 39–42. DOI: `10.1109/CoNTESA61248.2023.10384962`.

- Springboot no está orientado para manejar grandes volumenes de datos en poco tiempo, ya que las peticiones consumen significativamente recursos hardware, como cpu y ram.
- Springboot no soporta el protocolo MQTT de forma nativa, protocolo con el cual se comunica con la capa de gateways.
- Hacer escritura de grandes volumenes de datos puede saturar la ejecución de hilos, haciendo interminable la persistencia en bases de datos.

Por último se realiza un diagrama de la arquitectura final para la capa de datos, como se muestra en la **figura 12**.

Figura 12
NETIG STACK *Arquitectura NETIG para la capa de datos de Smart Campus UIS.*

6.3.3. Procesar:

Este paso fue netamente práctico e investigativo, ya que se definió la arquitectura para la capa de datos, pero la implementación suponía un reto, ya que hay poca información a nivel técnico de estas tecnologías trabajando en conjunto. Cabe resaltar que las tecnologías utilizadas en este stack son OpenSource, y la arquitectura se despliega en un contenedor Docker, su archivo de configuración YAML es el siguiente:

```
services:
  nodered:
    image: "nodered/node-red:4.0.8"
```

```
    container_name: nodered
    restart: always
    ports:
      - "1880:1880"
    volumes:
      - ./data/nodered:/data
    networks:
      - iot_network
    depends_on:
      - emqx
  emqx:
    user: root
    image: "emqx/emqx:5.8"
    container_name: emqx
    ports:
      - "18083:18083"
    volumes:
      - ./data/emqx/data:/opt/emqx/data
      - ./data/emqx/log:/opt/emqx/log
    networks:
      - iot_network
  influxdb:
    image: "influxdb:2.7.11"
    container_name: influxdb
    ports:
      - "8086:8086"
    volumes:
      - ./data/influxdb/data:/var/lib/influxdb2
      - ./data/influxdb/config:/etc/influxdb2
    environment:
      - DOCKER_INFLUXDB_INIT_MODE=setup
      - DOCKER_INFLUXDB_INIT_USERNAME=admindb
      - DOCKER_INFLUXDB_INIT_PASSWORD=smartcampusuis
```

```
      - DOCKER_INFLUXDB_INIT_ORG=uis
      - DOCKER_INFLUXDB_INIT_BUCKET=iotuis
      - DOCKER_INFLUXDB_INIT_RETENTION=1w
      - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=smartcampusuis-iot-auth-token
    networks:
      - iot_network
  telegraf:
    image: "telegraf:1.33"
    container_name: telegraf
    volumes:
      - ./data/telegraf/telegraf.conf:/etc/telegraf/telegraf.conf
    depends_on:
      - emqx
      - influxdb
    restart: unless-stopped
    networks:
      - iot_network
  grafana:
    image: "grafana/grafana:11.5.0"
    container_name: grafana
    ports:
      - "3000:3000"
    volumes:
      - ./data/grafana:/var/lib/grafana
    networks:
      - iot_network
    depends_on:
      - influxdb

networks:
  iot_network:
    driver: bridge
```

6.3.4. Review:

Durante este sprint se identificaron limitaciones en la arquitectura inicial de la capa de datos del sistema Smart Campus UIS, evidenciando su poca eficiencia y escalabilidad frente a los requerimientos IoT del caso de estudio. Tras un análisis exhaustivo, se propuso y documentó una nueva arquitectura basada en el stack NETIG (Node-RED, EMQX, Telegraf, InfluxDB y Grafana), la cual nace como propuesta para el caso de estudio propuesto, a partir de investigación de otras arquitecturas similares utilizadas en el campo e industria IoT. Esta arquitectura se adapta mejor a las necesidades del procesamiento de datos en tiempo real, manejo de series temporales y visualización. Esta nueva solución se integró exitosamente utilizando tecnologías Open Source y contenedores Docker, sentando las bases de una arquitectura modular, eficiente y especializada para entornos IoT.

6.4. SPRINT 4. CAPA EXTERNA

6.4.1. Dividir:

En este sprint se abordó la capa externa o capa de **gateways** del sistema Smart Campus UIS. Esta capa representa los dispositivos que recopilan datos del entorno físico, y los transmiten al sistema a través del protocolo **MQTT**.

Dado que esta capa es completamente externa e independiente de la arquitectura interna del sistema, se decidió dividirla en función del formato de comunicación, es decir, en la capacidad de los dispositivos (reales o simulados) de enviar datos válidos en estructura JSON, en lugar de dividir por tipo de hardware o tecnología utilizada.

6.4.2. Analizar:

Se analizó el papel fundamental que tienen los gateways en un sistema IoT. Aunque estos pueden variar en forma, capacidades o tecnología (ESP32, Raspberry Pi, scripts simuladores, etc.), lo único relevante para el sistema central es la estructura del mensaje JSON que transmiten estos gateways, el cual debe seguir el estándar definido por el `DataService` para asegurar compatibilidad y correcta persistencia.

Esto permite abstraer la capa de gateways, haciendo que el sistema funcione sin importar si los datos provienen de un dispositivo físico, un simulador, una API, o incluso una prueba

manual por consola, siempre que el mensaje JSON cumpla con la estructura requerida.

6.4.3. Procesar: Se estableció un estándar en la estructura JSON de estos datos:


```
{
  "name": "Test",
  "protocols": [
    {
      "type": "mqtt",
      "host": "localhost",
      "port": 1883,
      "topic": "device-messages",
      "clientId": "mocker-client",
      "username": "user",
      "password": "password"
    }
  ],
  "sampler": {
    "type": "sequential",
    "steps": [
      {
        "type": "loop",
        "delay": 2500
      }
    ]
  },
  "generators": [
    {
```

```json
      "type": "timestamp",
      "name": "TimestampGenerator"
    },
    {
      "type": "boolean",
      "name": "BooleanGenerator",
      "probability": 0.5
    },
    {
      "type": "random_integer",
      "name": "RandomInteger",
      "min": 25,
      "max": 30
    },
    {
      "type": "random_double",
      "name": "RandomDouble",
      "min": 50,
      "max": 70,
      "decimals": 2
    }
  ]
}
```

6.4.4. Review: Se verificó que la arquitectura del sistema, especialmente el `DataService` con EMQX y Telegraf, responde correctamente a cualquier fuente externa válida. Este sprint confirma la independencia entre la capa de gateways y el resto del sistema, lo cual:

- Aumenta la portabilidad del sistema.
- Permite integración progresiva de dispositivos reales.
- Asegura sostenibilidad al admitir nuevas tecnologías sin modificar el core.

6.5. SPRINT 5. DOCUMENTACIÓN INTEGRAL CON MKDOCS

6.5.1. Dividir: Para este sprint se decidió integrar y documentar los avances técnicos en una estructura unificada. Se trabajó sobre los componentes del sistema previamente definidos (*AdminService*, *DataService* y *Gateway*), enfocándose en la documentación, despliegue y buenas prácticas para la sostenibilidad del proyecto Smart Campus UIS. Ver en la figura 13.

Figura 13
Documentación integral del proyecto. *Interfaz MKdocs.*

6.5.2. Analizar: Se consolidó la información relevante en archivos Markdown y se estructuró con la herramienta **MkDocs**. Esto permitió organizar temas como:

- Arquitectura general del sistema y de cada capa.
- Uso del repositorio `smart_campus_core` y despliegue con Docker Compose.

- Documentación detallada del servicio de administración y sus APIs REST.
- Proceso de despliegue automatizado en una máquina virtual de Google Cloud.
- Simulaciones de sensores usando protocolos MQTT y AMQP con configuración YAML y Mocker CLI/Web.
- Reglas y buenas prácticas para la extensión futura del sistema. Ver figura 14.

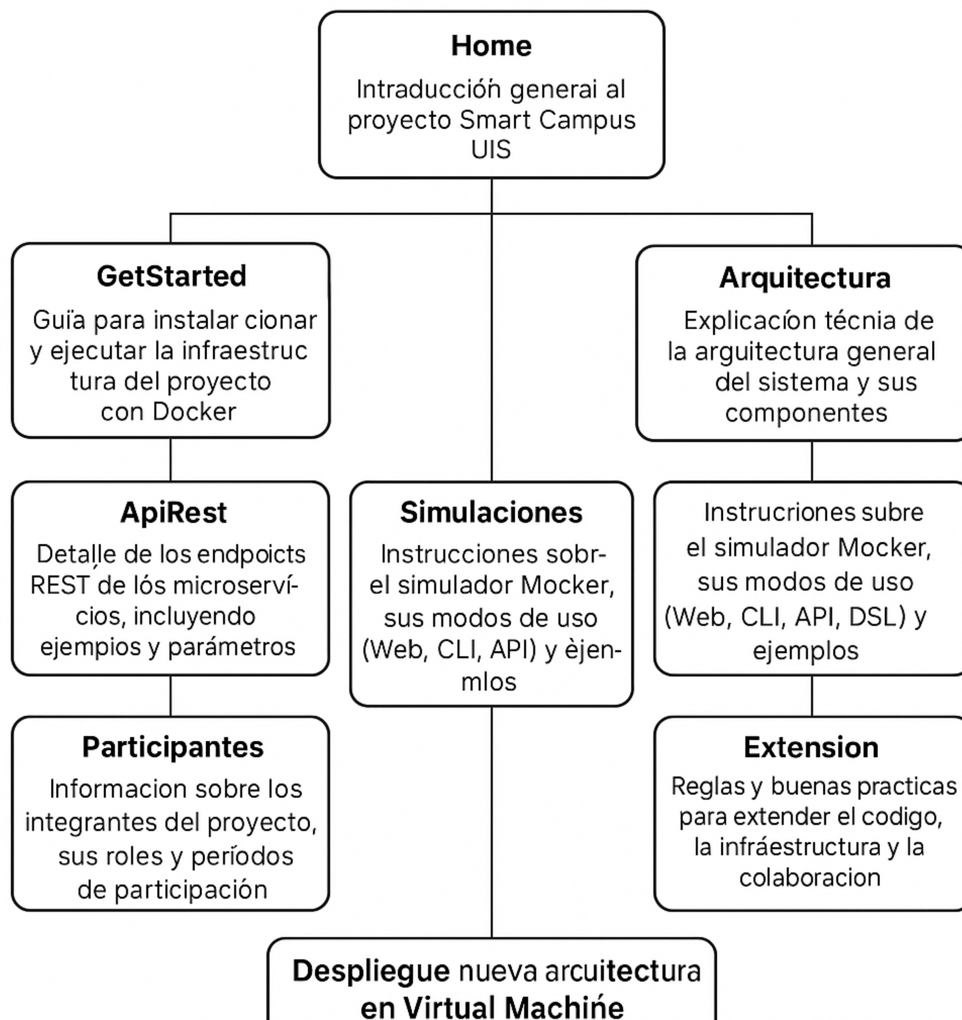

Figura 14

Diagrama de la documentación integral del proyecto. *Diagrama de la documentación con sus componentes.*

6.5.3. Procesar:

Se organizaron los documentos en la herramienta **MkDocs**, permitiendo generar una documentación navegable. Se implementó un archivo `mkdocs.yml` que define el menú de navegación, agrupando los contenidos según su rol en el sistema (arquitectura, APIs, despliegue, extensión, etc.). Además, se incluyeron ejemplos prácticos, configuraciones YAML, comandos CLI y visualizaciones para facilitar su uso.

6.5.4. Review:

Se validó que la documentación es funcional y refleja el estado actual del sistema. Se probó el despliegue completo en una máquina virtual de Google Cloud desde cero y se verificó el acceso a los servicios (EMQX, Node-RED, Grafana, InfluxDB). También se revisó la generación de simulaciones y la correcta respuesta de los servicios REST.

6.6. SPRINT 6. DESPLIEGUE EN GCP

6.6.1. Dividir:

En este sprint se abordó el despliegue de la solución Smart Campus UIS en la nube, específicamente utilizando los servicios de **Google Cloud Platform (GCP)**. Se dividió el trabajo en dos grandes bloques: la creación de la máquina virtual (VM) y el despliegue de la plataforma sobre dicha VM.

6.6.2. Analizar:

El objetivo fue replicar el sistema completo en un entorno cloud, permitiendo su disponibilidad remota y evaluación desde cualquier parte del mundo. Esto implicó configurar una VM, transferir el proyecto y asegurar la correcta ejecución de los servicios en contenedores Docker.

Creación de la VM en GCP

1. Tener una cuenta activa en Google Cloud.

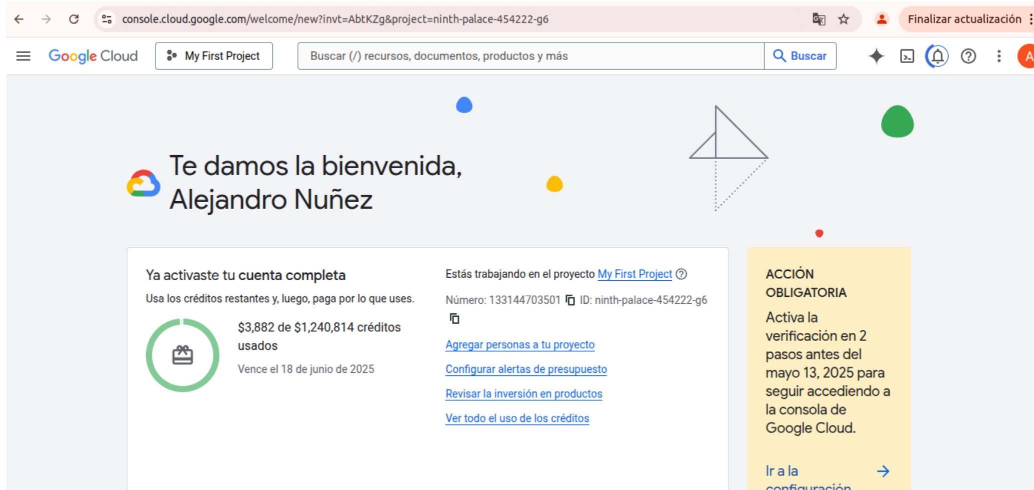

Figura 15
Cuenta en Google Cloud Platform. *Cuenta activa en Google Cloud Platform.*

2. Crear o seleccionar un proyecto en la consola.

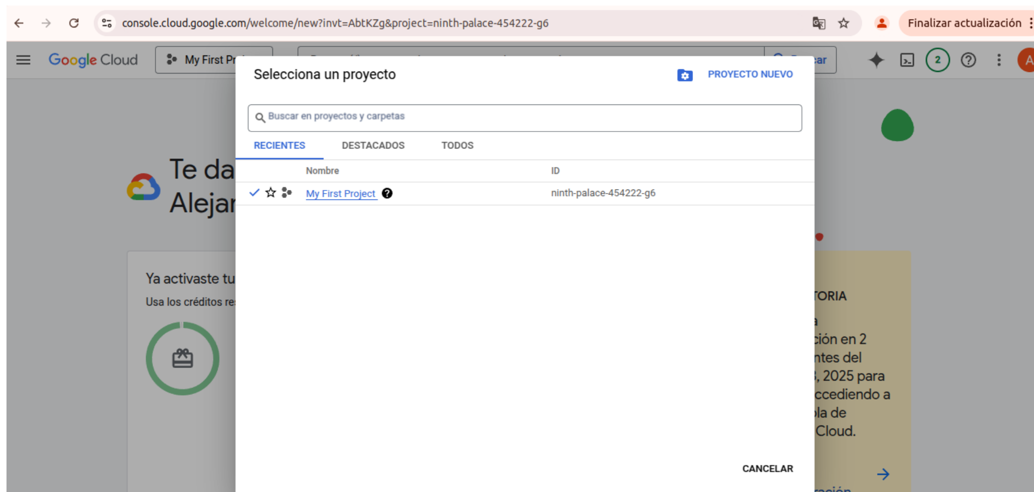

Figura 16
Selección de proyecto Google Cloud Platform. *Proyecto activo en Google Cloud Platform.*

3. Ir a la sección *Instancias de VM.*

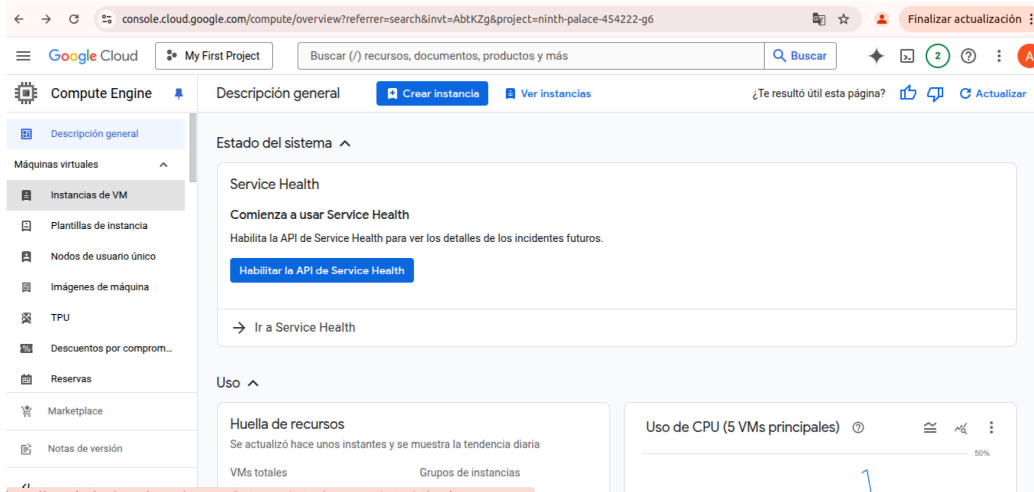

Figura 17

Instancias de maquinas virtuales. *Instancias de maquinas virtuales en Google Cloud Platform.*

4. Crear una nueva instancia personalizada.

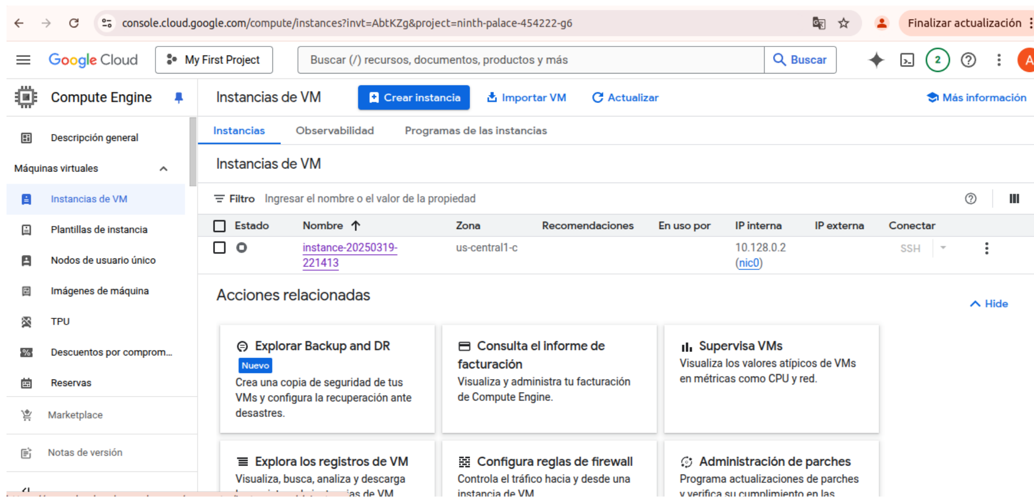

Figura 18

Crear instancia de maquina virtual. *Creación de instancia de maquinas virtuales en Google Cloud Platform.*

5. Seleccionar la región y zona del servidor.

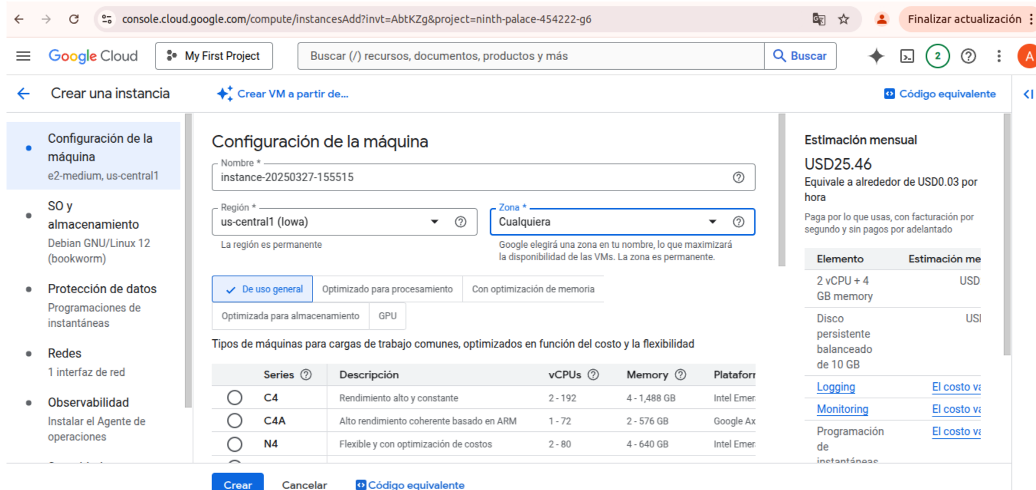

Figura 19

Selección de región y zona de la maquina virtual. *Selección de región y zona de la instancia de la maquina virtual en Google Cloud Platform.*

6. Elegir el tipo de máquina `e2-medium`.

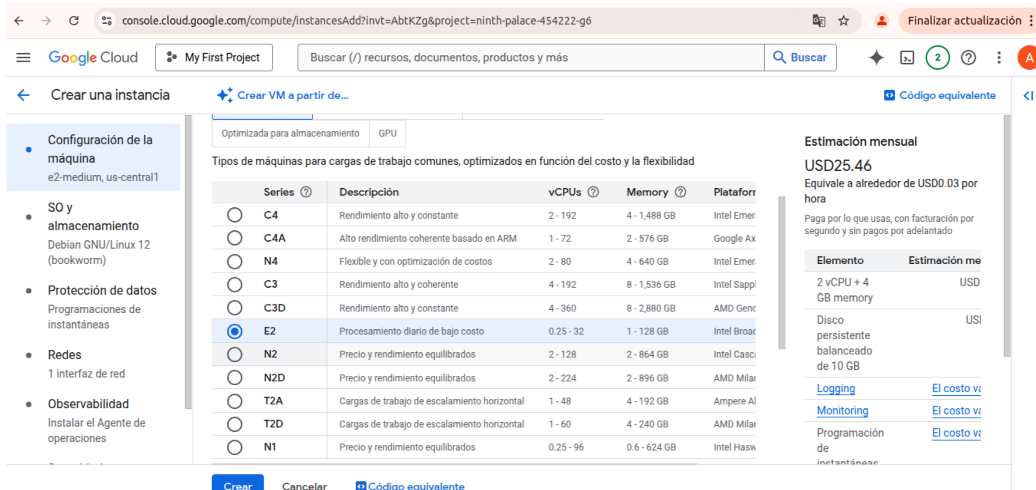

Figura 20

Selección de tipo de maquina. *Selección del tipo de la instancia de la maquina virtual en Google Cloud Platform.*

7. Configurar características: CPU, RAM, disco, etc.

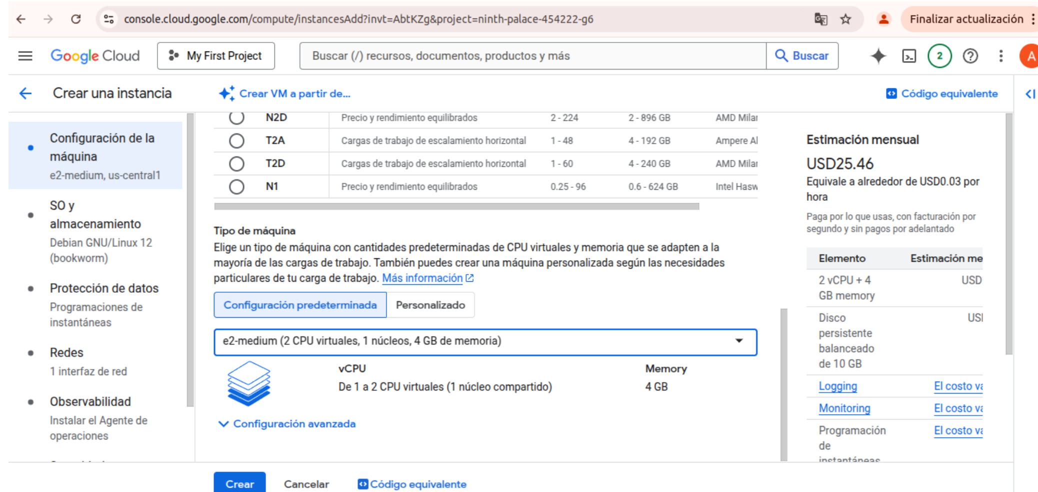

Figura 21

Selección de caracteristicas de la maquina virtual. *Selección de caracteristicas de la instancia de la maquina virtual en Google Cloud Platform.*

8. Seleccionar imagen de sistema: Ubuntu 22.04.

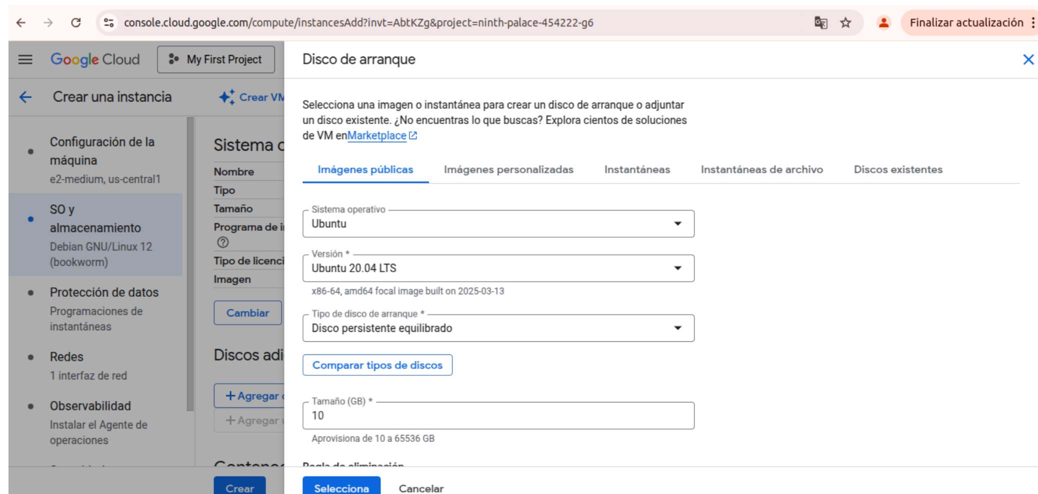

Figura 22

Selección de la imagen de la maquina virtual. *Selección de imagen de la instancia de la maquina virtual en Google Cloud Platform.*

9. Configurar redes y etiquetas de acceso.

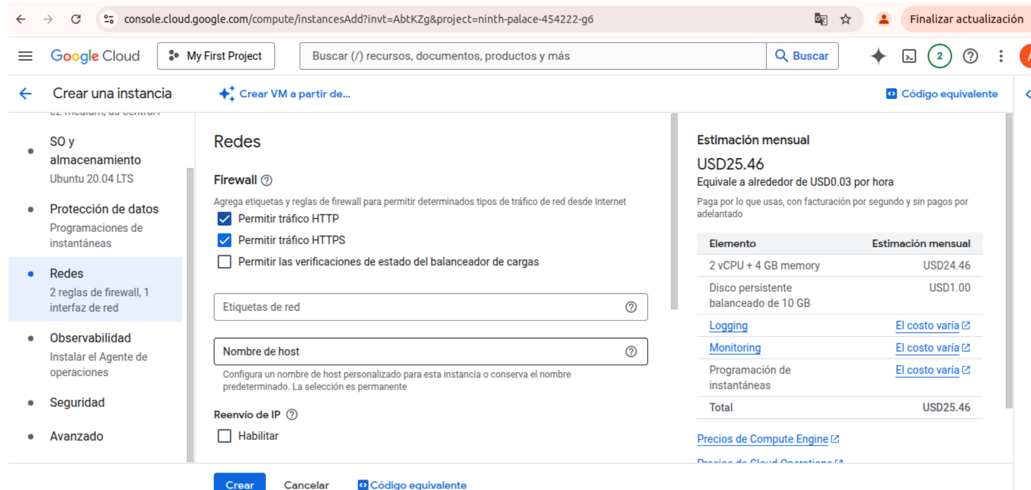

Figura 23

Configurar redes y etiquetas de acceso de la maquina virtual. *Configurar redes y etiquetas de acceso de la maquina virtual en Google Cloud Platform.*

6.6.3. Procesar:

El procedimiento de despliegue de la solución fue:

1. Subir archivo `SmartCampus.zip` desde máquina local.

2. Configurar el proyecto en GCloud:

```
alejandro@Alejopc:~/Documentos/Universidad/Proyecto de grado/DocSmartCampus$ gcl
oud config set project ninth-palace-454222-g6
```

Figura 24
Configurar proyecto en Google Cloud Platform. *Configurar proyecto en Google Cloud Platform*

3. Transferir el archivo a la VM:

```
alejandro@Alejopc:~/Documentos/Universidad/Proyecto de grado/DocSmartCampus$ gcl
oud compute scp SmartCampus.zip instance-20250327-155515:~ --zone=us-central1-c
```

Figura 25
Transferir archivo desde a la maquina virtual. *Transferir archivo desde local a la maquina virtual en Google Cloud Platform.*

4. Acceder a la VM vía SSH.

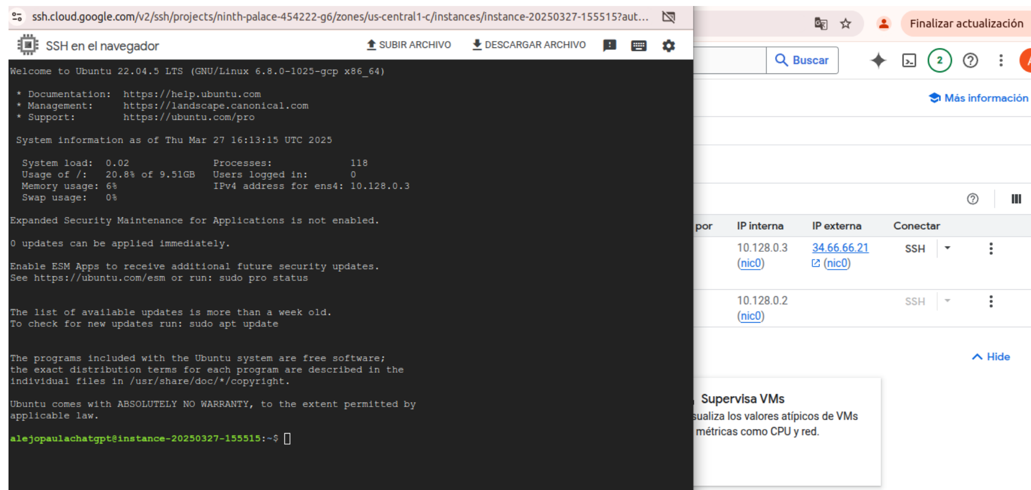

Figura 26

Acceder a la maquina virtual por medio de SSH. *Acceder a la maquina virtual por medio de SSH en Google Cloud Platform*

5. Instalar herramientas: `unzip`, `docker`, `docker-compose`.

6. Extraer el proyecto y levantar los servicios con:

```
docker-compose up -d
```

7. Abrir puertos necesarios:

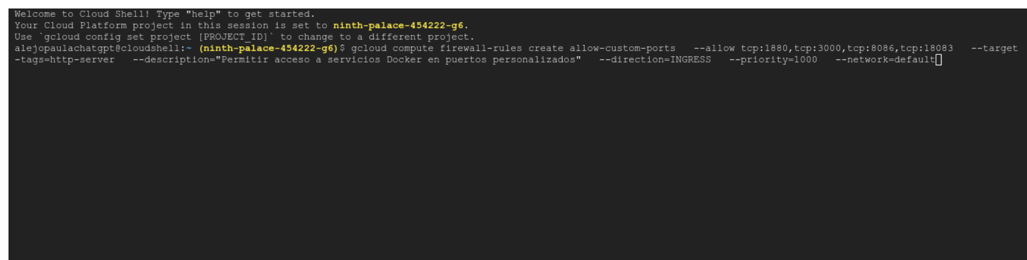

Figura 27

Abrir puertos de la maquina virtual. *Abrir puertos de la maquina virtual en Google Cloud Platform.*

6.6.4. Review: Con el sistema en ejecución, se verificó el acceso desde un navegador web usando la IP pública de la VM. Los servicios desplegados fueron:

- **Node-RED:** `http://34.66.66.21:1880`
- **Grafana:** `http://34.66.66.21:3000`
- **InfluxDB:** `http://34.66.66.21:8086`
- **EMQX Dashboard:** `http://34.66.66.21:18083`

Servicio	Usuario	Contraseña
Grafana	admin	admin
Node-RED	(sin auth por defecto)	—
InfluxDB 2	(configurable)	—
EMQX	admin	public

Tabla 2
Credenciales por defecto de los servicios desplegados

Este despliegue en la nube valida la portabilidad del sistema y deja lista la plataforma para futuras pruebas con usuarios reales o integraciones con dispositivos físicos.

6.7. SPRINT 7. REGLAS DE INTEGRACIÓN Y EXTENSIÓN

6.7.1. Dividir: Para este sprint se planteó como objetivo central la definición de un conjunto de reglas y lineamientos que permitan extender y evolucionar la plataforma Smart Campus UIS de manera controlada, sostenible y colaborativa. El trabajo se dividió en cinco ejes temáticos: principios generales, extensión del código, infraestructura y despliegue, gestión de datos, y colaboración.

6.7.2. Analizar: Se analizaron experiencias previas de mantenimiento de software, buenas prácticas de arquitectura sostenible. También se tuvo en cuenta técnicas de documentación

como (Swagger, MkDocs, estructura modular) para garantizar coherencia entre los lineamientos y la solución implementada.

6.7.3. Procesar: Se diseñó un documento de reglas de extensión para el proyecto Smart Campus, estructurado en cinco apartados:

- **Principios Generales:** Modularidad, escalabilidad, seguridad, mantenibilidad y comunicación como pilares del sistema.
- **Reglas para la Extensión del Código:** Uso de buenas prácticas (SOLID, patrones de diseño), obligatoriedad de pruebas automatizadas, versionado adecuado, documentación y revisión de código.
- **Reglas para la Infraestructura y Despliegue:** Estándar de contenerización con Docker, automatización CI/CD, monitoreo, logging y configuración desacoplada.
- **Reglas para la Gestión de Datos:** Garantizar integridad, estructura de datos documentada, manejo seguro de la retención y cumplimiento de normativas de privacidad.
- **Reglas de Colaboración:** Uso de herramientas de gestión de tareas, capacitación, trabajo en equipo y mejora continua.

Este documento quedó referenciado en la documentación técnica general del proyecto generada con MkDocs, garantizando su acceso y consulta por futuros colaboradores.

6.7.4. Review: El conjunto de reglas fue validado con base en la arquitectura existente y los flujos de trabajo ya implementados. Se comprobó su alineación con las tecnologías usadas (Docker, Git, MkDocs, Swagger) y se verificó que cubre los aspectos críticos para garantizar la sostenibilidad del sistema. Con ello, se cumple el objetivo de establecer una guía clara para futuras extensiones, contribuyendo a la evolución

A lo largo del proceso de desarrollo, estructurado mediante la metodología ágil basada en sprints o circuitos, se logró avanzar significativamente en la reestructuración arquitectónica de la plataforma Smart Campus UIS. Cada sprint permitió abordar de manera modular y focalizada las diferentes capas del sistema: administración, datos y gateways, asegurando un tratamiento especializado para cada una.

Se implementaron mejoras clave como la adopción del stack tecnológico NETIG, optimizado para entornos IoT, la documentación integral con Swagger y MkDocs, y el despliegue funcional de la plataforma en la nube mediante Google Cloud Platform. Además, se definieron reglas de extensibilidad que aseguran la escalabilidad y mantenimiento del sistema a futuro.

Este enfoque dividido y progresivo permitió no solo reorganizar técnicamente la plataforma, sino también consolidar prácticas sólidas de documentación, despliegue, integración y mantenimiento. En conjunto, los resultados del desarrollo permiten afirmar que se cumplieron los objetivos técnicos propuestos, estableciendo una base robusta y sostenible para futuras evoluciones del sistema Smart Campus UIS.

7. RESULTADOS LOGRADOS

En esta sección se presentan los resultados obtenidos tras el desarrollo del proyecto, los cuales evidencian el cumplimiento del objetivo general y los objetivos específicos planteados al inicio. Cada resultado se deriva del análisis, diseño, implementación y validación de la nueva arquitectura para la plataforma Smart Campus UIS, así como de la documentación integral realizada.

OBJETIVO 1: Definir la estructura de la arquitectura base de la plataforma Smart Campus UIS.

Este objetivo se trabajó unicamente en el primer **sprint 1** siendo de suma importancia ya que a partir de él, se definiría una base primordial en el desarrollo de los siguientes objetivos. Se definió una arquitectura organizada en tres capas: *Administración*, *Datos* y *Gateways*. Para cada una se diseñaron diagramas estructurales y se seleccionaron tecnologías adecuadas según el rol de cada capa. Esto permitió establecer una base clara para el desarrollo y evolución futura de la plataforma, logrando el **primer objetivo específico**.

OBJETIVO 2: Documentar integralmente la plataforma incluyendo su arquitectura y diferentes componentes.

Para este objetivo se llevó a cabo una documentación general en el **sprint 5**, es decir, documentación técnica, arquitectónica y de negocio, consolidada mediante la herramienta **MkDocs**, que permite navegar fácilmente entre secciones como arquitectura, APIs REST, despliegue, y simulación de sensores. Adicionalmente, se integró una documentación con **Swagger** en el **sprint 2**. Este conjunto de entregables constituye una **documentación integral**, cumpliendo con el **segundo objetivo específico**.

OBJETIVO 3: Refactorizar si es necesario uno o diferentes componentes de la plataforma para mantener la coherencia de la misma.

Durante el desarrollo, específicamente en el **sprint 3**, se realizó la refactorización de la capa de datos, como resultado del análisis de la arquitectura inicial de la plataforma, se identificaron limitaciones relacionadas con la escalabilidad y especialización en entornos IoT. En respuesta, se diseñó y documentó una nueva arquitectura modular basada en el stack **NETIG** (Node-RED, EMQX, Telegraf, InfluxDB y Grafana), que reemplaza componentes no especializados como Spring Boot y MongoDB. Esta reestructuración mejora la capacidad del sistema para manejar grandes volúmenes de datos en tiempo real y optimiza el rendimiento en escenarios distribuidos y de alta demanda. Esta refactorización se implementó de forma modular usando **Docker Compose**, permitiendo su despliegue y mantenimiento eficiente. De esta forma se cumple con el **tercer objetivo específico**.

OBJETIVO 4: Definir un conjunto de reglas de extensión que permitan la evolución de la plataforma a futuro.

Este objetivo se desarrolló principalmente en el **sprint 7**, fue de suma importancia realizarlo de último para abordar el mayor margen de soluciones posible. Se definieron buenas prácticas y lineamientos que aseguran la evolución sostenible del sistema, incluyendo la estandarización del mensaje JSON entre gateways y servicios, recomendaciones de tecnologías escalables, y estructuras para futuras integraciones (ej. CI/CD, soporte multi-campus, monitoreo remoto). Esto responde al **cuarto objetivo específico**.

OBJETIVO 5: Despliegue versión cloud

Finalmente, este objetivo se trabajó en el **sprint 6** no se tuvo en cuenta desde un inicio, debido a su complejidad y altos costos de accesibilidad, sin embargo, luego de investigaciones se logró obtener una prueba gratuita en la que se desplegó la solución completa en una máquina virtual

de **Google Cloud Platform**, validando su funcionamiento remoto, la portabilidad del sistema y la integración de servicios distribuidos. Esta validación garantiza que la solución propuesta no solo es teóricamente viable, sino también funcional en entornos reales y escalables.

Los resultados alcanzados permiten concluir que se logró una solución arquitectónica especializada para IoT, acompañada de una documentación completa y una estrategia de despliegue modular, cumpliendo así con los objetivos del proyecto.

8. CONCLUSIONES

Se determinó los aspectos fuertes y debilidades de la arquitectura de la plataforma Smart Campus UIS, lo que permitió una intervención integral que incluyen aspectos de documentación, adaptación, despliegue y reglas de extensión. Está intervención permitirá evitar problemas de degradación y erosión de la arquitectura y agilizar futuras intervenciones en la plataforma.

El presente trabajo permitió realizar una reestructuración profunda de la arquitectura del sistema Smart Campus UIS, orientándola a un entorno especializado para aplicaciones IoT. A partir del análisis de la arquitectura original y la evaluación de múltiples tecnologías, se concluyó que el sistema requería una solución más modular, escalable y eficiente.

La implementación de una arquitectura basada en el stack NETIG (Node-RED, EMQX, Telegraf, InfluxDB y Grafana) ofreció mejoras significativas en el procesamiento, almacenamiento y visualización de datos en tiempo real. Esta nueva arquitectura se adaptó completamente a las necesidades del caso de estudio, integrando principios de arquitectura software y tecnologías Open Source optimizadas para entornos IoT.

La metodología utilizada facilitó en gran medida el desarrollo de los objetivos planteados, ya que al tener alcances y soluciones muy aisladas y diferentes era necesario dividir, analizar, procesar y hacer un review independiente para cada problema.

La construcción de una documentación integral, soportada en MkDocs y estructurada por capas, flujos y componentes, garantiza que la plataforma pueda ser comprendida, mantenida y extendida fácilmente por futuros desarrolladores.

Finalmente, el despliegue exitoso de la plataforma en la nube (Google Cloud Platform) valida la portabilidad del sistema, su capacidad de operación remota y su preparación para pruebas con usuarios reales o integraciones físicas.

9. TRABAJO FUTURO

A partir de los resultados obtenidos, se proponen las siguientes líneas de trabajo futuro para continuar con la evolución de la plataforma Smart Campus UIS:

- **Integración con dispositivos físicos:** Conectar sensores reales (IoT) a través de gateways para recopilar datos del entorno universitario en tiempo real, validando el sistema en escenarios reales.
- **Análisis avanzado de datos:** Implementar mecanismos de análisis estadístico, aprendizaje automático o detección de anomalías sobre las series temporales almacenadas.
- **Gestión remota y notificaciones:** Desarrollar funciones que permitan emitir alertas automáticas (correo, dashboards, notificaciones push) ante condiciones críticas detectadas por los sensores.
- **Monitoreo multi-campus:** Escalar la arquitectura para soportar múltiples campus o edificios, permitiendo su uso como sistema centralizado de monitoreo universitario.
- **Automatización CI/CD:** Incluir pipelines de integración y despliegue continuo (CI/CD) para facilitar actualizaciones del sistema y despliegues en producción con mínima intervención manual.
- **Soporte multilenguaje:** Permitir que los mensajes enviados desde los gateways puedan ser consumidos en diferentes lenguajes, agregando mayor flexibilidad a los sistemas externos conectados.

Estas proyecciones permitirán fortalecer el sistema Smart Campus UIS como una plataforma de investigación robusta, flexible y abierta a la innovación.

BIBLIOGRAFÍA

AASEN, Gunnar. *Introduction to InfluxData's InfluxDB and TICK Stack*. 2017. URL: `https://www.influxdata.com/blog/introduction-to-influxdatas-influxdb-and-tick-stack/` (cit. on p. 26).

CERVANTES MACEDA, Humberto; VELASCO, PERLA, and CASTRO, LUIS. *Arquitectura de Software*. 2016 (cit. on p. 21).

CHOSEN, Vincent. *The MING Stack: What It Is and How It Works*. 2024. URL: `https://www.influxdata.com/blog/-ming-stack-introduction-influxdb/` (cit. on p. 25).

CLEMENTS, Paul C. "Software architecture in practice". In: *Diss. Software Engineering Institute* (2002) (cit. on p. 29).

DE SILVA, Lakshitha and BALASUBRAMANIAM, Dharini. "Controlling software architecture erosion: A survey". In: *Journal of Systems and Software* 85.1 (2012). Dynamic Analysis and Testing of Embedded Software, pp. 132–151. DOI: `https://doi.org/10.1016/j.jss.2011.07.036` (cit. on p. 20).

IBM. *Internet of Things (IoT)*. 2025. URL: `https://www.ibm.com/mx-es/topics/internet-of-things` (cit. on p. 20).

INFLUXDATA. *InfluxDB vs MongoDB*. 2024. URL: `https://www.influxdata.com/comparison/influxdb-vs-mongodb/` (cit. on p. 52).

PEÑA, Jenny Ruiz de la and CRUZ, Oscar Aguilera. "Importancia de la Ingeniería de Software en la producción de software". In: *Ciencias Holguín* 13.2 (2007), pp. 1–8 (cit. on p. 23).

ROLDÁN, María Luciana; GONNET, Silvio, and LEONE, Horacio. "Representación de la Evolución y Refactoring de Arquitecturas de Software mediante la Aplicación y Captura de Operaciones Arquitectónicas". In: *Revista Tecnología y Ciencia* 27 (2015), pp. 197–213 (cit. on p. 22).

TRIPATHI, Piyush; MIRAZ, Mahdi H., and JOSHI, Snigdha. "Comparative Analysis of MongoDB and InfluxDB for Time Series Data Management in IoT Environments: A Study on Performance, Scalability, and Concurrency". In: *2023 International Conference on Computing, Networking, Telecommunications Engineering Sciences Applications (CoNTESA)*. 2023, pp. 39–42. DOI: 10.1109/CoNTESA61248.2023.10384962 (cit. on p. 52).

VIVAS, Ortiz. *Divide y Vencerás*. 2024 (cit. on p. 38).