

PONTIFICIA UNIVERSIDAD CATOLICA DEL ECUADOR

FACULTAD DE HÁBITAT INFRAESTRUCTURA Y

CREATIVIDAD

CARRERA: INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Trabajo de Titulación como requisito previo para la obtención del título de Ingeniero en Sistemas de Información

TEMA:

PROTOTIPO DE SISTEMA DE MONITOREO DE RENDIMIENTO Y USO DE RECURSOS EN TIEMPO REAL PARA APLICACIONES BACKEND Y BASES DE DATOS

AUTOR: Juan David Erazo Unda

DIRECTOR: Phd. Rafael Melgarejo Heredia

QUITO DM, JULIO 2026

DEDICATORIA

Dedico este trabajo de titulación a mis padres, quienes han sido el motor de mi vida y el ejemplo que ha guiado cada uno de mis pasos. Gracias por su amor incondicional, por los sacrificios realizados para brindarme las mejores oportunidades y por enseñarme, con su esfuerzo diario, el valor de la perseverancia, la responsabilidad y la humildad. Cada logro alcanzado durante mi formación académica es también suyo, porque detrás de cada meta cumplida siempre estuvieron con su apoyo, sus consejos y su confianza inquebrantable en mí. Sin ustedes, este sueño no habría sido posible.

A mi hermano, por estar siempre presente a lo largo de este camino. Gracias por tu apoyo constante, por confiar en mí desde el inicio y por creer que podía alcanzar esta meta incluso cuando aún parecía lejana. Gracias por estar pendiente de mí, por acompañarme en cada etapa de este proceso y por compartir conmigo tanto los momentos de alegría como los desafíos que surgieron en el camino. Tu confianza, compañía y respaldo significaron mucho para mí y fueron una motivación adicional para seguir adelante hasta alcanzar este logro.

A ustedes, mi familia, les dedico este logro con todo mi cariño y gratitud. Este trabajo representa años de esfuerzo, aprendizaje y dedicación, pero sobre todo es el reflejo del amor, los valores y el respaldo que siempre he recibido de ustedes. Todo lo que he conseguido hasta hoy tiene sus raíces en el hogar que construyeron y en las enseñanzas que me han acompañado durante toda mi vida.

AGRADECIMIENTO

Expreso mi más sincero agradecimiento a todos los profesores de la Escuela de Ingeniería en Sistemas de Información de la Pontificia Universidad Católica del Ecuador, quienes a lo largo de mi formación universitaria contribuyeron de manera significativa a mi crecimiento académico, profesional y personal.

Gracias por compartir sus conocimientos, experiencias y enseñanzas con dedicación y compromiso. Cada asignatura cursada, cada proyecto desarrollado y cada reto planteado dentro del aula representaron una oportunidad para aprender, crecer y fortalecer las competencias que hoy me permiten culminar esta importante etapa de mi vida. Más allá de los contenidos técnicos, ustedes transmitieron valores fundamentales como la ética profesional, la disciplina, la responsabilidad, el pensamiento crítico y la búsqueda constante de la excelencia.

Agradezco especialmente la exigencia académica con la que guiaron nuestro proceso de formación, ya que fue precisamente esa exigencia la que nos impulsó a superar nuestros propios límites, a desarrollar resiliencia frente a las dificultades y a comprender la verdadera responsabilidad que implica ejercer la profesión de ingeniero en sistemas. Gracias por motivarnos a cuestionar, investigar, innovar y buscar soluciones que generen un impacto positivo en nuestro entorno.

Hoy culmino esta etapa con la convicción de que las enseñanzas recibidas en esta institución han sentado bases sólidas para mi futuro profesional. Llevaré conmigo cada aprendizaje, consejo y experiencia compartida durante estos años, con el compromiso de representar con orgullo los valores de mi formación y de ejercer mi profesión con responsabilidad, integridad y vocación de servicio. A todos ustedes, mi más profundo respeto, admiración y gratitud.

RESUMEN

Las empresas de desarrollo de software administran un número creciente de servicios distribuidos, pero con frecuencia carecen de una visión centralizada de su estado y de su rendimiento. La supervisión suele depender de la revisión manual de los registros (logs) dispersos en cada servidor, un procedimiento lento y propenso a errores que retrasa la detección y el diagnóstico de los fallos. Ante esta necesidad, el presente trabajo desarrolla un prototipo de sistema de monitoreo de rendimiento que reúne en una sola plataforma las métricas de uso de recursos de aplicaciones backend y bases de datos.

El prototipo se construyó con Java y Spring Boot en el backend, Vue.js en el frontend y el ecosistema de observabilidad formado por Micrometer, Prometheus y Grafana, empleando PostgreSQL como base de datos. Su aporte principal es un orquestador que genera y recarga de forma dinámica la configuración de Prometheus a partir de los sistemas registrados desde la interfaz web, lo que elimina la edición manual de archivos. Asimismo, se incorporó un módulo de registro histórico del tráfico HTTP que conserva la actividad de los servicios con independencia de la política de retención de Prometheus. El desarrollo se llevó a cabo bajo la metodología ágil Scrum.

La validación experimental evidenció una reducción cercana al 98 % en el tiempo de detección e identificación de fallos frente al método manual de revisión de logs, y de alrededor del 90 % en el tiempo de incorporación de nuevos sistemas al monitoreo, manteniendo los tiempos de respuesta dentro de los límites establecidos al aumentar la carga. Los resultados confirman que la centralización y la automatización del monitoreo mejoran de manera significativa la visibilidad operativa y reducen el esfuerzo manual requerido para diagnosticar incidentes.

ÍNDICE

Contenido

ÍNDICE DE FIGURAS 8
ÍNDICE DE TABLAS 10
CAPÍTULO I: INTRODUCCIÓN 11
1. Marco de Referencia: 11
1.1 Justificación 11
1.2 Planteamiento del problema 11
1.3 Objetivo General 13
1.3.1 Objetivos Específico 13
1.4 Alcance 13
1.5 Descripción General 14
1.5.1 Perspectiva del sistema 14
1.5.2 Características de los usuarios 15
1.5.3 Restricciones 16
1.5.4 Dependencias del sistema 16
CAPÍTULO II: FUNDAMENTACIÓN TEÓRICA 18
2.1 Marco Teórico 18
2.2 Definiciones, Acrónimos y Abreviaturas 18
2.2.1 Observabilidad 18
2.2.2 Site Reliability Engineering (SRE) 19
2.2.3 Four Golden Signals 19
2.2.4 Métodos RED y USE 20
2.2.5 Telemetría y OpenTelemetry 21
2.2.6 Monitoreo Cloud-Native y Ecosistema CNCF 22
2.2.7 DevOps Observability 22
2.2.8 Métricas y Series Temporales 23
2.2.9 Spring Boot 24
2.2.10 Micrometer 25
2.2.11 Prometheus 25
2.2.12 PromQL 25
2.2.13 Exporters de Prometheus 26
2.2.14 Grafana 26
2.2.15 Service Discovery y Registro Semi-automático 26
2.2.16 PostgreSQL 27

2.2.17 API REST 27
2.2.18 Vue.js 27
2.2.19 Análisis comparativo de tecnologías 28
2.2.20 Docker y Docker Compose 29
2.2.21 Herramientas de Soporte: Git, GitHub e IDEs 29
2.2.22 Diagramas UML y Pruebas de Software 30
2.3 Metodología de Desarrollo 30
2.3.1 Marco Metodológico 30
2.3.2 Product Backlog 30
2.3.3 Sprints del Proyecto 33
2.3.4 Definition of Done 35
2.3.5 Métricas Ágiles del Proyecto 35
CAPÍTULO III: ANÁLISIS Y DISEÑO DEL SISTEMA 37
3.1 Análisis de Requerimientos 37
3.2 Requerimientos Funcionales 37
3.3 Requerimientos No Funcionales 40
3.4 Casos de Uso 41
3.4.1 Actores del Sistema 41
3.4.2 Diagrama de Casos de Uso General 41
3.5 Diagramas de Casos de Uso Siguiente Nivel 42
3.5.1 Casos de Uso Siguiente Nivel R1: Gestión de Sistemas Monitoreados 42
Ingreso de Sistema Monitoreado 43
Modificación de Sistema Monitoreado 44
Eliminación de Sistema Monitoreado 45
Consulta y Cambio de Estado de Sistemas 45
3.5.2 Casos de Uso Siguiente Nivel R2: Generación Dinámica de Configuración 46
Generación Automática del prometheus.yml 46
3.5.3 Casos de Uso Siguiente Nivel R3: Visualización de Métricas en Tiempo Real 47
Visualización de Métricas de Aplicación Spring Boot 48
Visualización de Métricas de Base de Datos PostgreSQL 48
3.5.4 Casos de Uso Siguiente Nivel R4: Consulta de Historial de Tráfico HTTP .. 49
Recolección Automática de Tráfico HTTP 49
Consulta Filtrada de Historial de Tráfico HTTP 50
3.6 Modelado del Sistema 51

3.6.1 Diagrama Entidad-Relación (ERD) 51
3.7 Diseño de la Arquitectura del Sistema 58
3.7.1 Diagrama de Arquitectura General 58
3.7.2 Arquitectura en Capas del Backend Spring Boot 60
3.8 Diseño de la Base de Datos 60
3.9 Diseño de la Interfaz de Usuario 61
3.9.1 Estructura de Navegación 61
3.9.2 Vista: Sistemas (ruta /sistemas) 62
3.9.3 Vista: Dashboard (ruta /) 63
3.9.4 Vista: Tráfico HTTP (ruta /trafico) 64
3.10 Desarrollo del Sistema 65
3.10.1 Implementación de Scrum 65
3.10.3 Sprint Planning 66
3.10.4 Sprint 1: Configuración del Stack e Instrumentación 69
3.10.5 Sprint 2: CRUD de Sistemas Monitoreados 73
3.10.6 Sprint 3: Dashboard de Visualización en Tiempo Real 80
3.10.7 Sprint 4: Módulo de Tracking de Tráfico HTTP 82
3.10.8 Sprint 5: Auditoría 84
3.10.9 Sprint 6: Pruebas y Validación Experimental 85
3.12.4 Análisis de Resultados 102
3.13 Resumen del Capítulo 103
CAPÍTULO IV: CONCLUSIONES Y RECOMENDACIONES 105
4.1 Conclusiones 105
4.2 Trabajos Futuros 106
BIBLIOGRAFÍA 108

ÍNDICE DE FIGURAS

Figura 1. Diagrama de casos de uso general. 42
Figura 2. Casos de uso siguiente nivel R1: Gestión de Sistemas Monitoreados. 43
Figura 3. Casos de uso siguiente nivel R2: Generación Dinámica de Configuración. ... 46
Figura 4. Casos de uso siguiente nivel R3: Visualización de Métricas en Tiempo Real. 47
Figura 5. Casos de uso siguiente nivel R4: Consulta de Historial de Tráfico HTTP. 49
Figura 6. Diagrama Entidad-Relación (ERD). 52
Figura 7. Diagrama de Arquitectura General. 59
Figura 8. Diagrama de navegación del frontend. 62
Figura 9. Prototipo de la vista Sistemas (/sistemas). 62
Figura 10. Prototipo de la vista Dashboard (/). 64
Figura 11. Prototipo de la vista Tráfico HTTP (/trafico). 64
Figura 12. Diagrama del proceso Scrum. 66
Figura 13. Configuración del stack de monitoreo en Docker Compose. 70
Figura 14. Métricas expuestas por la aplicación instrumentada. 72
Figura 15. Vista de gestión de sistemas monitoreados. 78
Figura 16. Archivo prometheus.yml generado dinámicamente. 79
Figura 17. Dashboard de métricas en tiempo real de una aplicación Spring Boot. 81
Figura 18. Dashboard de métricas de base de datos PostgreSQL. 81
Figura 19. Vista de historial de tráfico HTTP. 83
Figura 20. Infraestructura de auditoría. 84
Figura 21. Resultado de la ejecución de la prueba PU-01. 87
Figura 22. Resultado de la ejecución de la prueba PU-02. 88
Figura 23. Resultado de la ejecución de la prueba PU-02. 89
Figura 24. Resultado de la ejecución de la prueba PU-04. 89
Figura 25. Resultado de la ejecución de la prueba PU-05. 90
Figura 26. Resultado de la ejecución de la prueba PU-06. 91
Figura 27. Resultado de la ejecución de la prueba PU-07. 91
Figura 28. Resultado de la ejecución de la prueba PU-08. 92
Figura 29. Resultado de la ejecución de la prueba PI-01. 94
Figura 30. Resultado de la ejecución de la prueba PI-02. 94
Figura 31. Resultado de la ejecución de la prueba PI-03. 95

Figura 32. Resultado de la ejecución de la prueba PI-04. 96
Figura 33. Resultado de la ejecución de la prueba PI-05 / PI-06. 97
Figura 34. Resultado de la ejecución de la prueba PI-07. 97
Figura 35. Reporte consolidado de la ejecución de las pruebas automatizadas. 98

ÍNDICE DE TABLAS

Tabla 1. Comparativa de los principales frameworks de backend Java.....24
Tabla 2. Comparativa de los principales frameworks de frontend JavaScript.....28
Tabla 3. Comparativa de herramientas de monitoreo para aplicaciones backend. 28
Tabla 4. Product Backlog del prototipo de sistema de monitoreo. 31
Tabla 5. Métricas ágiles por sprint.....35
Tabla 6. Diccionario de datos, tabla sistemas. 52
Tabla 7. Diccionario de datos, tabla eventos_http. 54
Tabla 8. Diccionario de datos, tabla usuarios. 55
Tabla 9. Diccionario de datos, tabla roles.....56
Tabla 10. Diccionario de datos, tabla usuarios_roles.....56
Tabla 11. Diccionario de datos, tabla tipos_agente. 57
Tabla 12. Diccionario de datos, tabla logs. 57
Tabla 13. Sprint Planning. 66
Tabla 14. Sprint Backlog 1. 73
Tabla 15. Sprint Backlog 2. 79
Tabla 16. Sprint Backlog 3. 82
Tabla 17. Sprint Backlog 4. 83
Tabla 18. Sprint Backlog 5. 84
Tabla 19. Sprint Backlog 6. 85
Tabla 20. Casos de prueba unitaria. 86
Tabla 21. Casos de prueba de integración.....92
Tabla 22. Comparación del tiempo de detección e identificación de fallos. 100
Tabla 23. Comparación de onboarding manual vs. automatizado. 101
Tabla 24. Tiempo de respuesta y escalabilidad del prototipo. 101

CAPÍTULO I: INTRODUCCIÓN

1. Marco de Referencia:

1.1 Justificación

Actualmente, las empresas de desarrollo de software operan con entornos tecnológicos cada vez más complejos, conformados por múltiples servicios que deben funcionar de manera continua y eficiente para garantizar la disponibilidad de las aplicaciones que ofrecen a sus clientes. Estas infraestructuras suelen incluir herramientas como Kafka, Zookeeper, Jetty, Mirror Maker y PostgreSQL, entre otras, que trabajan de forma conjunta para sostener operaciones críticas dentro de los sistemas empresariales.

Sin embargo, uno de los principales desafíos que enfrentan los equipos técnicos es la falta de visibilidad y control sobre el rendimiento individual y colectivo de estos servicios. En muchos casos, los mecanismos de supervisión existentes son limitados o dispersos, pues se basan únicamente en métricas aisladas o en la revisión manual de registros de logs, lo que dificulta la detección temprana de fallos y la identificación de cuellos de botella en la infraestructura.

Ante este contexto, se justifica el desarrollo de una herramienta que centralice la información sobre el estado de los servicios, las métricas de uso de recursos y la disponibilidad de los módulos que conforman la arquitectura. La solución propuesta busca servir como un apoyo para el área técnica, al facilitar la observación en tiempo real del comportamiento de las aplicaciones y contribuir a una gestión más proactiva frente a incidentes o degradaciones del servicio.

Desde una perspectiva académica, este trabajo aporta al estudio de la observabilidad en entornos empresariales, al proponer un enfoque modular y adaptable que puede evolucionar hacia implementaciones más robustas. En el ámbito profesional, representa un paso inicial hacia la automatización de la supervisión, lo que fortalece la eficiencia operativa y la toma de decisiones basada en datos dentro de una empresa de desarrollo de software.

1.2 Planteamiento del problema

Las empresas de desarrollo de software dependen de infraestructuras tecnológicas compuestas por múltiples servicios que deben mantenerse operativos de manera

constante. No obstante, la supervisión de estos servicios suele realizarse mediante herramientas aisladas o procesos manuales, lo que genera dificultades para identificar a tiempo fallos de rendimiento, saturaciones o interrupciones en alguno de los componentes del sistema.

En el entorno analizado, los servicios como *Kafka, Zookeeper, Jetty, Mirror Maker y PostgreSQL* pueden fallar de manera independiente, afectando el desempeño general del sistema. Si bien existen soluciones de monitoreo disponibles en el mercado, su implementación completa suele requerir licencias costosas o configuraciones complejas que no siempre se ajustan a las necesidades o capacidades de una empresa de desarrollo en crecimiento.

Ante esta situación, surge la necesidad de desarrollar un prototipo de sistema de monitoreo de rendimiento que permita recopilar y visualizar métricas clave de los servicios que conforman la infraestructura, integrando datos sobre consumo de recursos, disponibilidad y registros de actividad en una interfaz centralizada y de fácil interpretación.

A partir de esta problemática, se plantea la siguiente pregunta principal:

¿Cómo desarrollar un prototipo de sistema de monitoreo de rendimiento que permita centralizar las métricas e información de los servicios de una empresa de desarrollo de software, contribuyendo a mejorar la observabilidad y detección temprana de fallos?

De esta pregunta se derivan las siguientes preguntas secundarias:

1. ¿Qué herramientas o agentes son adecuados para la recolección de métricas y logs en los servicios empresariales?
2. ¿Qué métricas de rendimiento son relevantes para evaluar el estado de los sistemas en tiempo real?
3. ¿Cómo integrar la información recolectada en una interfaz unificada que facilite la interpretación de datos?
4. ¿Qué estrategias pueden aplicarse para emitir alertas o notificaciones ante fallos o comportamientos anómalos?
5. ¿De qué manera el prototipo puede contribuir a reducir el tiempo de respuesta frente a incidentes en los sistemas monitoreados?

1.3 Objetivo General

Desarrollar un prototipo de sistema de monitoreo de rendimiento que permita recolectar, centralizar y visualizar métricas operativas de aplicaciones y bases de datos en una empresa de desarrollo de software, incorporando un proceso de configuración semi-automático que facilite el monitoreo de nuevas aplicaciones mediante el uso de Spring Boot, Micrometer, Prometheus y Grafana para mejorar la observabilidad y la detección temprana de fallos.

1.3.1 Objetivos Específico

- Implementar y documentar la instrumentación de aplicaciones Spring Boot utilizando Micrometer, para exponer métricas clave como tiempos de respuesta y uso de recursos mediante el endpoint /actuator/prometheus.
- Desarrollar un mecanismo simple de registro automático que permita agregar nuevas aplicaciones al monitoreo desde una interfaz web, actualizando la configuración de Prometheus sin necesidad de editar archivos manualmente.
- Diseñar y construir dashboards en Grafana que muestren en tiempo real e históricamente las métricas recolectadas, facilitando la identificación de problemas de rendimiento y permitiendo un análisis claro del comportamiento de las aplicaciones.

1.4 Alcance

El presente proyecto contempla el desarrollo de una herramienta de monitoreo enfocada exclusivamente en aplicaciones construidas con Spring Boot, instrumentadas mediante Micrometer para exponer sus métricas operativas a través del endpoint /actuator/prometheus. Adicionalmente, se incluye el monitoreo de bases de datos PostgreSQL mediante un endpoint exportador propio, construido en el backend con Micrometer, que expone métricas esenciales como el tamaño de la base de datos, las conexiones activas, las transacciones (commits y rollbacks) y las estadísticas de operaciones sobre las filas, sin requerir la instalación de un exporter externo. La solución integra un frontend web que permite registrar nuevas aplicaciones de forma semi-automática, regenerando la configuración de Prometheus sin intervención manual y centralizando las fuentes de métricas que se desean observar. Sobre esa base, el propio frontend ofrece dashboards funcionales que visualizan, en tiempo real e históricamente,

el comportamiento de las aplicaciones y de las bases de datos monitoreadas, diferenciando la visualización según el tipo de agente. De forma complementaria, se integran las interfaces de Grafana y Prometheus mediante enlaces externos para análisis adicionales. Finalmente, se incorpora un módulo de registro histórico del tráfico HTTP y de auditoría de operaciones que preserva la actividad de los sistemas con independencia de la política de retención de Prometheus.

Quedan fuera del alcance el monitoreo completo de la infraestructura del servidor (red y disco), la integración con sistemas de trazabilidad distribuida, la implementación de agentes externos, los sistemas de alertas automáticas y los orquestadores de contenedores como Kubernetes. La herramienta tampoco considera aplicaciones desarrolladas en otros lenguajes o frameworks, ni motores de base de datos distintos de PostgreSQL. Su alcance se limita a demostrar la viabilidad de una solución de observabilidad centrada en las métricas expuestas por Spring Boot y PostgreSQL, complementada con una interfaz web que facilita la incorporación de nuevas aplicaciones al monitoreo y el análisis histórico de su tráfico.

1.5 Descripción General

1.5.1 Perspectiva del sistema

El prototipo de sistema de monitoreo de rendimiento representa una solución orientada a mejorar la observabilidad de las infraestructuras tecnológicas en empresas de desarrollo de software. Al centralizar la recolección y visualización de métricas operativas, el sistema permitirá a los equipos técnicos contar con una visión unificada del estado de sus aplicaciones Spring Boot y bases de datos PostgreSQL en tiempo real.

La solución se concibe como un sistema independiente que se integra de forma no intrusiva a las aplicaciones existentes, aprovechando el endpoint /actuator/prometheus expuesto por Micrometer para recolectar métricas sin modificar el código de las aplicaciones monitoreadas. A través de un frontend web, el equipo técnico podrá registrar nuevas aplicaciones a la herramienta de manera semi-automática, eliminando la necesidad de editar manualmente los archivos de configuración de Prometheus. Los datos recolectados serán almacenados como series temporales en Prometheus y visualizados mediante dashboards interactivos en Grafana, facilitando tanto el análisis en tiempo real como el histórico del comportamiento de los servicios.

Este sistema representa un paso inicial hacia la automatización del monitoreo en entornos empresariales, con una arquitectura modular que puede evolucionar hacia implementaciones más robustas a medida que crezcan las necesidades del equipo técnico.

1.5.2 Características de los usuarios

Características del Administrador del Sistema

Tipo de usuario: Administrador

- Formación: Conocimientos en administración de infraestructura tecnológica, gestión de servicios y herramientas de monitoreo.
- Habilidades: Capacidad para configurar y mantener los componentes del sistema (Prometheus, Grafana, exporters), gestionar las aplicaciones registradas en el monitoreo y resolver incidentes técnicos.
- Actividades: Gestiona la configuración general del sistema, registra y elimina aplicaciones en la interfaz de monitoreo, supervisa el estado de los componentes del prototipo y garantiza la disponibilidad de los servicios de observabilidad.

Características del Desarrollador

Tipo de usuario: Usuario (Técnico o Desarrollador)

- Formación: Conocimientos en desarrollo de software con Spring Boot y administración básica de bases de datos PostgreSQL.
- Habilidades: Capacidad para interpretar métricas de rendimiento, identificar comportamientos anómalos en los dashboards y aplicar los ajustes necesarios en las aplicaciones instrumentadas.
- Actividades: Consulta los dashboards de Grafana para monitorear el comportamiento de las aplicaciones a su cargo, registra nuevas aplicaciones al sistema desde la interfaz web y analiza métricas históricas para detectar problemas de rendimiento.

Los técnicos y desarrolladores serán los principales usuarios del sistema, ya que son quienes interactúan directamente con las aplicaciones monitoreadas y requieren visibilidad inmediata sobre su estado operativo para tomar decisiones informadas ante incidentes o degradaciones del servicio.

1.5.3 Restricciones

Una restricción relevante es el alcance tecnológico del prototipo, el cual está limitado exclusivamente al monitoreo de aplicaciones desarrolladas con Spring Boot instrumentadas mediante Micrometer, y a bases de datos PostgreSQL. Aplicaciones construidas con otros frameworks o lenguajes de programación, así como otros motores de base de datos, quedan fuera del alcance del sistema, lo que limita su aplicabilidad a entornos que no utilicen este stack tecnológico.

Otra restricción importante es la necesidad de que las aplicaciones monitoreadas expongan correctamente el endpoint /actuator/prometheus. En caso de que una aplicación no esté instrumentada con Micrometer o no tenga habilitado Spring Boot Actuator, el sistema no podrá recolectar sus métricas, quedando excluida del monitoreo sin posibilidad de integración mediante el flujo semi-automático propuesto.

Adicionalmente, el prototipo no contempla monitoreo completo de infraestructura de red ni de almacenamiento en disco, ni la integración con sistemas avanzados de logs o trazabilidad distribuida. Esta limitación implica que el sistema ofrece una visibilidad parcial del entorno tecnológico, enfocada únicamente en las métricas de rendimiento de las aplicaciones y la base de datos, sin cubrir la totalidad de los indicadores que conforman una solución de observabilidad empresarial completa.

1.5.4 Dependencias del sistema

Para el correcto funcionamiento del prototipo de monitoreo, se requiere que el entorno donde se despliegue cuente con los siguientes componentes en ejecución: Prometheus como motor de recolección y almacenamiento de series temporales, Grafana como plataforma de visualización de dashboards, y el exporter de PostgreSQL para exponer las métricas de la base de datos en el formato esperado por Prometheus. Estos tres componentes son interdependientes y deben estar operativos simultáneamente para garantizar el flujo completo de recolección y visualización de métricas.

Por parte de las aplicaciones a monitorear, es indispensable que estén desarrolladas con Spring Boot y que tengan configuradas las dependencias de Micrometer y Spring Boot Actuator, habilitando el endpoint /actuator/prometheus con acceso desde la red donde

opera Prometheus. Sin esta configuración previa en las aplicaciones, el sistema no podrá incorporarlas al monitoreo.

Finalmente, el frontend web del prototipo requiere conectividad con las APIs expuestas por el backend y las configuraciones de Prometheus para poder actualizar dinámicamente los targets de scraping al registrar nuevas aplicaciones, por lo que se asume disponibilidad de red entre los componentes del sistema durante su operación.

CAPÍTULO II: FUNDAMENTACIÓN TEÓRICA

2.1 Marco Teórico

Este capítulo presenta las bases conceptuales y teóricas que sustentan el desarrollo del prototipo. La fundamentación abarca desde los principios modernos de la ingeniería de confiabilidad de sitios (SRE) y la observabilidad distribuida, hasta las herramientas concretas del stack implementado. Se prioriza la profundidad teórica en los conceptos que definen la observabilidad como disciplina de ingeniería, telemetría, monitoreo cloud-native, métodos de instrumentación y estándares CNCF, y se sintetizan los aspectos tecnológicos de herramientas de soporte cuya justificación de selección es suficientemente clara.

2.2 Definiciones, Acrónimos y Abreviaturas

2.2.1 Observabilidad

La observabilidad, término originado en la teoría de control de Rudolf Kalman (1960) y adoptado por la ingeniería de software a mediados de la década de 2010, describe la capacidad de inferir el estado interno de un sistema a partir de sus salidas externas, sin necesidad de conocer de antemano qué preguntas se formularán. Esta distinción es fundamental: mientras el monitoreo tradicional responde preguntas conocidas mediante umbrales predefinidos, la observabilidad permite explorar preguntas desconocidas sobre el comportamiento emergente del sistema (Majors et al., 2022).

Charity Majors, figura central en la formalización del concepto, define la observabilidad como la capacidad de depurar cualquier estado que el sistema pueda alcanzar, incluso si nunca antes se ha producido, utilizando únicamente la telemetría que este emite (Majors et al., 2022). Esta propiedad es especialmente crítica en arquitecturas de microservicios y sistemas distribuidos, donde el espacio de estados posibles crece exponencialmente con el número de servicios interactuantes.

IBM (s.f.) formaliza los tres pilares técnicos de la observabilidad: las métricas (valores numéricos a lo largo del tiempo), los logs (registros textuales de eventos discretos) y las trazas distribuidas (representación del flujo de una solicitud a través de múltiples servicios). He et al. (2021) amplían este modelo en el contexto cloud-native, señalando que la correlación entre los tres pilares, vinculando una traza a sus métricas de latencia y a los logs de error correspondientes, es lo que habilita el análisis de causa raíz en sistemas distribuidos complejos. El presente prototipo se centra en el pilar de métricas,

reconociendo explícitamente que los pilares de logs y trazas distribuidas constituyen trabajo futuro de extensión del sistema.

2.2.2 Site Reliability Engineering (SRE)

Site Reliability Engineering (SRE) es una disciplina de ingeniería surgida en Google alrededor del año 2003, cuya premisa fundamental es aplicar principios y prácticas de ingeniería de software a la operación de sistemas a escala. Beyer et al. (2016) la definen como «lo que ocurre cuando un ingeniero de software es encargado de lo que solía llamarse operaciones», consolidando la convergencia entre desarrollo y operaciones que la cultura DevOps propugna desde una perspectiva organizacional.

El framework SRE introduce tres conceptos de medición que definen la relación entre un servicio y sus usuarios desde la perspectiva de la confiabilidad. Los Service Level Indicators (SLI) son métricas cuantitativas del nivel de servicio percibido; los Service Level Objectives (SLO) establecen los objetivos de cumplimiento de esos indicadores; y los Service Level Agreements (SLA) son los compromisos contractuales derivados. El error budget, o presupuesto de error, se deriva del SLO y representa la cantidad tolerable de indisponibilidad o degradación dentro de un período, siendo el mecanismo que balancea la velocidad de innovación con la estabilidad operacional (Beyer et al., 2016).

El concepto de toil (trabajo pesado), central en SRE, describe las tareas operativas manuales, repetitivas y no escalables que no aportan valor duradero al sistema. La edición manual de archivos de configuración de Prometheus cada vez que se incorpora un nuevo servicio al monitoreo es un ejemplo directo de toil. El prototipo desarrollado aborda precisamente este problema mediante la generación dinámica del prometheus.yml, reduciendo el toil operativo del equipo técnico y mejorando la escalabilidad del proceso de incorporación de nuevos sistemas al monitoreo.

2.2.3 Four Golden Signals

El Capítulo 6 del libro Site Reliability Engineering (Beyer et al., 2016) propone los Four Golden Signals como el conjunto mínimo de métricas suficientes para monitorear cualquier servicio orientado al usuario. A diferencia de metodologías que priorizan el monitoreo de infraestructura, los Golden Signals se centran en lo que el usuario experimenta directamente:

Latencia: tiempo que tarda el sistema en procesar una solicitud. Es importante distinguir entre la latencia de solicitudes exitosas y la de solicitudes fallidas; una tasa de errores baja

con alta latencia puede ocultar un problema grave. En el prototipo, esta señal es medida mediante las métricas HTTP de Micrometer (http_server_requests_seconds).

Tráfico: medida de la demanda que se está aplicando al sistema, expresada en solicitudes por segundo para servicios web o en transacciones por segundo para bases de datos. En la solución, esta señal es observada mediante la función rate() de PromQL sobre las métricas de Micrometer.

Errores: tasa de solicitudes que fallan, ya sea por errores explícitos (HTTP 5xx), implícitos (HTTP 200 con contenido incorrecto) o de política (respuestas que superan el SLO de latencia). En el sistema desarrollado, se calcula como la proporción de respuestas con status 5xx sobre el total de solicitudes.

Saturación: medida del grado en que el recurso más limitado del sistema está siendo utilizado. Expresa cuánto espacio de capacidad resta antes de que el servicio se degrade. Para la herramienta de monitoreo, la saturación de la JVM se observa a través del uso de memoria heap y la saturación de la base de datos a través del pool de conexiones HikariCP.

La correspondencia entre los Four Golden Signals y las métricas expuestas por Micrometer a través del endpoint de Spring Boot Actuator justifica la selección del stack tecnológico del prototipo desde una perspectiva de ingeniería de confiabilidad, más allá de la conveniencia técnica.

2.2.4 Métodos RED y USE

Complementarios a los Four Golden Signals, existen dos metodologías de instrumentación que ofrecen perspectivas específicas según el tipo de componente a monitorear.

El método RED (Rate, Errors, Duration), propuesto por Tom Wilkie (2018), es una especialización de los Golden Signals orientada a microservicios y APIs. Para cada servicio se mide: la tasa de solicitudes por segundo (Rate), el número o porcentaje de solicitudes fallidas (Errors) y la distribución del tiempo de procesamiento de cada solicitud (Duration). Wilkie desarrolló este método a partir de la observación de que los Four Golden Signals resultan difíciles de aplicar operativamente en entornos con decenas o cientos de microservicios, y que una simplificación de tres señales uniformes permite estandarizar el monitoreo a escala. En el prototipo, las tres señales RED están directamente disponibles a través de las métricas HTTP de Micrometer.

El método USE (Utilization, Saturation, Errors), propuesto por Brendan Gregg (2020), está orientado al análisis de recursos de infraestructura. Para cada recurso (CPU, memoria, disco, red, pool de conexiones) se mide: el porcentaje de tiempo que el recurso está ocupado (Utilization), el grado en que el recurso está siendo demandado más allá de su capacidad (Saturation) y el número de errores producidos (Errors). El método USE es especialmente relevante para el monitoreo de la base de datos PostgreSQL a través del endpoint exportador del backend, donde las métricas de conexiones activas (utilización del pool), conexiones en espera (saturación) y errores de transacción cubren las tres dimensiones del método.

2.2.5 Telemetría y OpenTelemetry

La telemetría es el proceso de recolección automática y transmisión de datos de medición desde sistemas remotos para su análisis en un punto centralizado. En el contexto de la ingeniería de software, la telemetría abarca los tres pilares de la observabilidad, métricas, logs y trazas, junto con los mecanismos de instrumentación, transporte y almacenamiento de esos datos (Majors et al., 2022). Zhang et al. (2021) distinguen entre instrumentación manual, donde el desarrollador agrega llamadas explícitas a una librería de telemetría, e instrumentación automática, donde el framework inyecta la recolección de datos de forma transparente. El prototipo utiliza la segunda modalidad: las aplicaciones externas monitoreadas emiten telemetría automáticamente a través de Micrometer sin intervención del equipo de monitoreo.

OpenTelemetry es el estándar unificado de telemetría para aplicaciones cloud-native, desarrollado bajo el auspicio de la Cloud Native Computing Foundation (CNCF) como resultado de la fusión de los proyectos OpenCensus y OpenTracing en 2019. Proporciona APIs, SDKs e infraestructura de recolección (el OpenTelemetry Collector) para los tres pilares de la observabilidad, con soporte para más de quince lenguajes de programación. Su modelo de datos estandarizado permite exportar telemetría a múltiples backends, Prometheus, Jaeger, Zipkin, Datadog, sin cambiar el código de instrumentación (OpenTelemetry, 2024). En el presente prototipo no se implementa OpenTelemetry directamente, dado que Micrometer cubre el pilar de métricas con menor complejidad de configuración, pero su existencia contextualiza el stack elegido dentro del ecosistema de estándares CNCF y define la dirección de evolución natural del sistema hacia trazas distribuidas.

2.2.6 Monitoreo Cloud-Native y Ecosistema CNCF

El término cloud-native describe aplicaciones construidas y operadas específicamente para entornos de nube, caracterizadas por su empaquetamiento en contenedores, su orquestación mediante plataformas como Kubernetes, su comunicación a través de microservicios y su entrega mediante pipelines de integración y despliegue continuo (CNCF, 2024). Esta arquitectura impone desafíos específicos al monitoreo: las instancias son efímeras y sus direcciones IP cambian constantemente, los servicios se escalan horizontalmente de forma dinámica, y la frontera entre aplicación e infraestructura se vuelve difusa.

El monitoreo basado en infraestructura estática, verificar si un servidor específico está vivo, es insuficiente en estos entornos. Burns et al. (2016), en su análisis de los sistemas de orquestación de Google (Borg, Omega y Kubernetes), señalan que la observabilidad en sistemas cloud-native requiere instrumentación a nivel de servicio, no de instancia, y mecanismos de descubrimiento dinámico de targets. Esta observación fundamenta directamente el mecanismo de registro semi-automático de la herramienta de monitoreo: ante la imposibilidad de gestionar manualmente un número creciente de servicios, el sistema actualiza su configuración de forma programática, adoptando el patrón de file-based service discovery de Prometheus.

La Cloud Native Computing Foundation (CNCF) es el organismo que alberga y certifica los proyectos de referencia del ecosistema cloud-native. Prometheus y Grafana son proyectos graduados de la CNCF, el nivel más alto de madurez, lo que garantiza su adopción comunitaria, estabilidad de APIs y compatibilidad con el estándar OpenMetrics que define el formato de exposición de métricas utilizado por el prototipo (CNCF, 2024). OpenTelemetry es un proyecto incubado de la CNCF en proceso de graduación, siendo el único estándar de telemetría respaldado simultáneamente por los principales proveedores de nube (AWS, Google Cloud, Microsoft Azure) y empresas de monitoreo (Datadog, New Relic, Dynatrace).

2.2.7 DevOps Observability

DevOps es una cultura y conjunto de prácticas que rompe la separación organizacional entre los equipos de desarrollo (Dev) y operaciones (Ops), promoviendo la responsabilidad compartida sobre el ciclo de vida completo del software. Kim et al. (2016) identifican la observabilidad como uno de los pilares técnicos de la cultura

DevOps, junto con el control de versiones, la integración continua y el despliegue continuo: un equipo no puede «you build it, you run it» sin visibilidad sobre el comportamiento del sistema en producción.

El concepto de shift-left observability extiende esta idea hacia etapas más tempranas del ciclo de vida, promoviendo que la instrumentación sea parte del proceso de desarrollo y no una responsabilidad delegada al equipo de operaciones al momento del despliegue. En términos prácticos, esto significa que un desarrollador que construye un servicio Spring Boot debe incluir la dependencia de Micrometer y activar el endpoint de métricas desde el inicio, no como una tarea posterior. El prototipo está diseñado bajo este principio: el proceso de incorporación de una nueva aplicación al monitoreo requiere únicamente que el servicio ya esté instrumentado con Micrometer, reduciendo la fricción de onboarding a la configuración mínima a través de la interfaz web.

Richardson (2019) señala que en arquitecturas de microservicios, la observabilidad no es una característica opcional sino una precondición para la operabilidad del sistema: sin métricas, logs y trazas, la depuración de problemas en un entorno con decenas de servicios interactuantes es prácticamente inviable. Esta premisa justifica el desarrollo de la solución desde una perspectiva de ingeniería de software empresarial, más allá de su dimensión académica.

2.2.8 Métricas y Series Temporales

Una métrica es una medición numérica que representa el valor de un atributo del sistema en un instante determinado. Cuando se registran continuamente a intervalos regulares, forman una serie temporal (time series) que permite analizar tendencias, detectar anomalías y correlacionar eventos con degradaciones de rendimiento (Prometheus, 2024). La densidad de información de una serie temporal crece con la frecuencia de recolección y la cardinalidad de las etiquetas (labels) asociadas; Prometheus gestiona esta tensión mediante su motor de almacenamiento TSDB, optimizado específicamente para operaciones de escritura y consulta sobre series temporales de alta cardinalidad.

Micrometer clasifica las métricas en cuatro tipos: los contadores (counters), que acumulan valores de forma monótonamente creciente; los medidores (gauges), que representan valores instantáneos variables; los histogramas (histograms), que distribuyen valores en rangos para calcular percentiles (p50, p95, p99); y los resúmenes (summaries), que calculan percentiles directamente en el cliente. La distinción entre histogramas del lado

del servidor, calculados en Prometheus mediante histogram_quantile(), y resúmenes del lado del cliente tiene implicaciones directas sobre la flexibilidad de consulta: los histogramas permiten recalcular percentiles post-hoc con cualquier rango de tiempo, mientras que los resúmenes los fijan en el momento de la recolección.

2.2.9 Spring Boot

Spring Boot es un framework de desarrollo Java que simplifica la creación de servicios basados en el ecosistema Spring mediante autoconfiguración y servidor embebido. Bhatia & Chhillar (2023) señalan que su módulo Spring Boot Actuator autoconfigura y expone métricas clave de rendimiento de forma nativa, convirtiéndolo en el framework Java con menor fricción de instrumentación. En el prototipo, Spring Boot cumple un doble rol: es el framework de las aplicaciones externas monitoreadas (que exponen sus métricas automáticamente a través de Micrometer) y es también el framework del backend central del sistema, que gestiona el registro de sistemas monitoreados y genera dinámicamente el archivo de configuración de Prometheus mediante construcción programática de texto.

Tabla 1. Comparativa de los principales frameworks de backend Java.

Característica	**Spring Boot**	**Quarkus**	**Micronaut**	**Jakarta EE**
Versión actual	3.x	3.x	4.x	10.x
Desarrollador	VMware / Pivotal	Red Hat	Object Computing	Eclipse Foundation
Integración Prometheus	Nativa (Micrometer)	Nativa (Micrometer)	Nativa (Micrometer)	Mediante librerías
Generación dinámica YAML	Sí (generación programática)	Sí	Sí	Manual
Curva de aprendizaje	Baja-Media	Media	Media	Alta
Comunidad	Muy grande	Grande	Media	Grande

Nota. Comparativa de frameworks Java. Spring Boot destaca por su integración nativa con Micrometer y soporte para generación dinámica de YAML, factores decisivos para el backend del sistema desarrollado. Autoría propia.

Se selecciona Spring Boot 3.x como framework del backend central del prototipo por su integración nativa con Micrometer, amplia comunidad y prevalencia en entornos empresariales Java.

2.2.10 Micrometer

Micrometer es una librería de instrumentación para aplicaciones JVM que actúa como capa de abstracción (facade) sobre múltiples sistemas de monitoreo. Su diseño permite que una misma aplicación exporte sus métricas a Prometheus, Datadog, InfluxDB u otros backends con solo cambiar la dependencia de registro (Prometheus, 2024). Integrado con Spring Boot Actuator, expone automáticamente métricas de JVM (memoria heap/non-heap, garbage collection, threads), del servidor HTTP embebido (Tomcat/Netty) y del pool de conexiones HikariCP, cubriendo directamente las tres señales del método RED y la dimensión de saturación del método USE.

2.2.11 Prometheus

Prometheus es un sistema de monitoreo y base de datos de series temporales de código abierto, desarrollado por SoundCloud en 2012 y actualmente proyecto graduado de la CNCF. Se ha consolidado como el estándar de facto para monitoreo cloud-native (Prometheus, 2024). Su arquitectura pull (scraping), donde Prometheus descarga las métricas de los targets en lugar de recibirlas, evita la saturación de los servicios monitoreados y simplifica el modelo de seguridad. La configuración de targets se gestiona mediante el archivo prometheus.yml; en el prototipo, este archivo es regenerado programáticamente por el backend Spring Boot tras cada operación CRUD sobre el registro de sistemas, implementando una forma simplificada de file-based service discovery que elimina el toil de configuración manual descrito en el marco SRE.

2.2.12 PromQL

PromQL es el lenguaje de consulta funcional de Prometheus, diseñado para operar sobre series temporales. Sus funciones de agregación, rate(), sum(), avg(), histogram_quantile(), permiten calcular directamente las señales de los métodos RED y

USE: rate() calcula la tasa de cambio de un contador para obtener solicitudes por segundo, histogram_quantile() obtiene el percentil 95 de latencia, y avg_over_time() suaviza valores de gauge para análisis histórico (Formadores IT, 2024). En el prototipo, las consultas PromQL son ejecutadas directamente desde el frontend Vue.js contra la API HTTP de Prometheus para construir los paneles del dashboard.

2.2.13 Exporters de Prometheus

Los exporters son programas intermediarios que recolectan métricas de sistemas que no exponen nativamente el formato Prometheus, convirtiéndolas al formato OpenMetrics esperado por el sistema de scraping. IBM Cloud Docs (s.f.) los define como esenciales para monitorear bases de datos, servidores web y sistemas operativos. En el prototipo, el propio backend cumple este rol para PostgreSQL: mediante la librería Micrometer se conecta a la base de datos y expone, en la ruta /api/metrics/postgres/{id}, métricas que cubren las tres dimensiones del método USE: utilización (conexiones activas / max_connections), saturación (conexiones en espera) y errores (transacciones revertidas, deadlocks).

2.2.14 Grafana

Grafana es la plataforma de visualización open source estándar para datos de series temporales, reconocida por Zhang et al. (2021) como el software de referencia para construir dashboards de monitoreo. Sus capacidades de paneles con variables dinámicas, destacadas por Shahi et al. (2022), permiten que un único dashboard parametrizado cubra múltiples sistemas al cambiar el valor de un filtro, $job en PromQL, lo que alinea directamente con el modelo multi-sistema del prototipo. En el proyecto, Grafana consume Prometheus como datasource y presenta dashboards diferenciados para métricas de JVM, HTTP, HikariCP y PostgreSQL, con acceso desde el frontend a través de enlace externo.

2.2.15 Service Discovery y Registro Semi-automático

El service discovery es el mecanismo mediante el cual los sistemas distribuidos registran y localizan dinámicamente las instancias de servicios disponibles. He et al. (2021) lo identifican como un componente esencial de la arquitectura de observabilidad cloud-native, ya que sin él la configuración de monitoreo se convierte en un cuello de botella operativo a medida que el ecosistema de servicios crece. Prometheus ofrece soporte nativo para mecanismos de descubrimiento en Kubernetes, Consul y DNS; en entornos

sin orquestador, el file-based service discovery lee periódicamente archivos JSON/YAML de targets y actualiza su configuración sin reinicio.

El prototipo implementa una versión simplificada de este patrón: cada operación CRUD sobre el registro de sistemas desencadena la regeneración del prometheus.yml por parte del PrometheusOrquestadorService del backend Spring Boot. Prometheus detecta el cambio en su ciclo de recarga y actualiza sus targets activos de forma automática. Esta aproximación reduce el toil operativo frente al registro completamente manual y es consistente con el patrón de file-based service discovery documentado por Shahi et al. (2022).

2.2.16 PostgreSQL

PostgreSQL es un sistema de gestión de bases de datos relacional open source que soporta transacciones ACID, tipos de datos avanzados (JSON, arrays) y un motor de índices extensible. En el prototipo cumple dos funciones complementarias: es la base de datos de persistencia del backend Spring Boot (almacena el registro de sistemas, usuarios, roles y logs de auditoría) y es a la vez uno de los sistemas externos monitoreados, cuyas métricas operativas son recolectadas por Prometheus a través del endpoint exportador del backend. Esta dualidad permite validar el funcionamiento del mecanismo de monitoreo de bases de datos con datos reales de la propia infraestructura del sistema.

2.2.17 API REST

Una API REST (Representational State Transfer) sigue los principios arquitectónicos definidos por Roy Fielding en el año 2000, utilizando el protocolo HTTP y operaciones estándar (GET, POST, PUT/PATCH, DELETE) sobre recursos identificados por URLs. En el prototipo, el backend Spring Boot expone una API REST para el CRUD de sistemas monitoreados. Cada operación de escritura desencadena la regeneración del prometheus.yml, cerrando el ciclo entre la gestión del registro y la actualización del sistema de scraping de forma programática.

2.2.18 Vue.js

Vue.js es un framework progresivo JavaScript para el desarrollo de SPAs, caracterizado por su arquitectura basada en componentes, su sistema de enlace reactivo y su baja curva de aprendizaje frente a Angular. Su versión 3, con Composition API y soporte nativo para TypeScript, es la base del frontend del prototipo, que gestiona los sistemas monitoreados

mediante una interfaz de dos vistas y consulta directamente la API HTTP de Prometheus con PromQL para construir los gráficos del dashboard.

2.2.19 Análisis comparativo de tecnologías

Tabla 2. Comparativa de los principales frameworks de frontend JavaScript.

Característica	Vue.js 3	React 18	Angular 17	Svelte 4
Curva de aprendizaje	Baja	Media	Alta	Baja
Integración con APIs REST	Excelente (Axios)	Excelente	Excelente	Excelente
TypeScript nativo	Sí (Composition API)	Parcial	Sí	Sí
Tamaño del bundle	Ligero	Medio	Pesado	Muy ligero
Documentación	Excelente	Excelente	Excelente	Buena

Nota. Vue.js 3 se selecciona por su baja curva de aprendizaje, soporte TypeScript nativo y excelente integración con APIs REST. Autoría propia.

Tabla 3. Comparativa de herramientas de monitoreo para aplicaciones backend.

Característica	Prometheus + Grafana	Datadog	New Relic	Zabbix
Modelo	Open Source	SaaS comercial	SaaS comercial	Open Source
Costo	Gratuito	De pago	De pago	Gratuito

Característica	Prometheus + Grafana	Datadog	New Relic	Zabbix
Integración Spring Boot	Nativa (Micrometer)	Agente externo	Agente externo	Plugins
Config. dinámica targets	Sí (YAML/SD)	Sí (API)	Sí (API)	Manual
Soporte OpenTelemetry	Sí (OTLP receiver)	Sí	Sí	No
Estándar CNCF	Graduado	Externo	Externo	No

Nota. Prometheus + Grafana es la única combinación open source con integración nativa con Micrometer, estándar CNCF graduado y soporte para configuración dinámica de targets. Autoría propia.

Se selecciona Prometheus + Grafana por su modelo open source, condición de estándar CNCF, integración nativa con Micrometer y capacidad de configuración dinámica de targets. Las soluciones SaaS no permiten el control programático sobre el archivo de configuración que requiere el mecanismo de generación dinámica del prototipo.

2.2.20 Docker y Docker Compose

Docker es una plataforma de contenedorización que empaqueta aplicaciones con sus dependencias en unidades portátiles ejecutables en cualquier entorno. Docker Compose orquesta múltiples contenedores mediante un único archivo YAML, coordinando red interna y orden de arranque. En el prototipo, Docker Compose despliega conjuntamente Prometheus y Grafana con política restart: unless-stopped, garantizando la disponibilidad del stack de monitoreo ante fallos de contenedor y la reproducibilidad del entorno en cualquier máquina con Docker instalado.

2.2.21 Herramientas de Soporte: Git, GitHub e IDEs

Git es el sistema de control de versiones distribuido utilizado para gestionar el código fuente del prototipo. GitHub actúa como repositorio remoto central, almacenando el

backend Spring Boot, el frontend Vue.js y los archivos de configuración de Prometheus y Grafana. Para el desarrollo se emplean IntelliJ IDEA Community Edition, por su soporte nativo para Spring Boot y depurador JVM, y Visual Studio Code, por su soporte nativo para Vue.js y TypeScript, . TypeScript se utiliza en el frontend como superset tipado de JavaScript, mejorando la robustez del código mediante detección de errores en tiempo de compilación. Estas herramientas son estándar en el ecosistema de desarrollo Java/Vue.js y su selección no requiere análisis comparativo extendido más allá de confirmar su disponibilidad gratuita y compatibilidad con el stack del proyecto. Repositorio del código fuente del proyecto: github.com/daviderazo04/Automated-Monitoring-Analytics-System

2.2.22 Diagramas UML y Pruebas de Software

Los diagramas de casos de uso UML modelan las interacciones entre actores y el sistema desde la perspectiva funcional, siendo la herramienta de comunicación estándar en la fase de análisis de requerimientos (IONOS, 2020). El diagrama entidad-relación (ERD), introducido por Chen (1976), representa la estructura de la base de datos a nivel conceptual. Las pruebas unitarias, implementadas con JUnit y Mockito en el ecosistema Java, verifican el comportamiento aislado de componentes individuales, mientras que las pruebas de integración validan el flujo completo entre módulos. En el prototipo, las pruebas unitarias cubren la lógica de generación del prometheus.yml y las pruebas de integración validan el ciclo CRUD → regeneración → actualización de targets.

2.3 Metodología de Desarrollo

2.3.1 Marco Metodológico

El desarrollo del prototipo adopta Scrum como marco de trabajo ágil (Schwaber & Sutherland, 2020), estructurando el trabajo en seis sprints de dos semanas. La elección de Scrum responde a la naturaleza iterativa del problema: los requerimientos del sistema de monitoreo emergen progresivamente a medida que se implementa cada componente del stack (instrumentación → backend → frontend → dashboards → pruebas), lo que requiere capacidad de adaptación entre ciclos. A diferencia de un enfoque en cascada, Scrum permite entregar incrementos funcionales verificables al final de cada sprint y ajustar el backlog según el aprendizaje técnico obtenido en el ciclo anterior.

2.3.2 Product Backlog

El Product Backlog es la lista ordenada y priorizada de todos los requerimientos del sistema. Cada ítem representa una historia de usuario (HU) o tarea técnica con su

estimación en puntos de historia (SP) bajo la escala Fibonacci (1, 2, 3, 5, 8). La Tabla 4 presenta el Product Backlog inicial del prototipo:

Tabla 4. Product Backlog del prototipo de sistema de monitoreo.

ID	Historia de usuario / Tarea técnica	Prioridad	SP	Sprint
HU-01	Como administrador, necesito registrar un sistema monitoreado con alias, host, puerto y tipo de agente para que Prometheus lo incluya en su configuración.	Alta	5	1
HU-02	Como administrador, necesito que el prometheus.yml se actualice automáticamente al registrar un sistema para no editar archivos manualmente.	Alta	8	1
HU-03	Como administrador, necesito editar los datos de un sistema registrado para corregir configuraciones incorrectas.	Alta	3	2
HU-04	Como administrador, necesito eliminar un sistema del monitoreo con confirmación previa para evitar borrados accidentales.	Alta	3	2
HU-05	Como administrador, necesito pausar y reanudar el monitoreo de un sistema sin eliminarlo para gestionar mantenimientos temporales.	Media	3	2
HU-06	Como administrador, necesito que el sistema diferencie el formulario según el tipo de agente (Spring Boot /	Alta	5	2

ID	Historia de usuario / Tarea técnica	Prioridad	SP	Sprint
	dbpostgres) para una configuración precisa.			
HU-07	Como administrador, necesito ver métricas de JVM (heap, threads) y HTTP (tasa, latencia, errores) de aplicaciones Spring Boot en tiempo real.	Alta	8	3
HU-08	Como administrador, necesito ver métricas de PostgreSQL (conexiones, tamaño, bloqueos) en el mismo dashboard para centralizar el monitoreo.	Alta	5	3
HU-09	Como administrador, necesito seleccionar el sistema y el rango temporal en el dashboard para analizar períodos específicos.	Alta	5	3
HU-10	Como administrador, necesito que el dashboard se actualice cada 15 segundos para mantener visibilidad en tiempo real.	Media	3	3
HU-11	Como administrador, necesito acceder a las interfaces de Grafana y Prometheus desde la barra lateral para análisis avanzado.	Baja	2	4
HU-12	Como sistema, necesito registrar en tabla de logs cada operación CRUD con fecha, origen, acción e IP para trazabilidad de cambios.	Media	5	4

ID	Historia de usuario / Tarea técnica	Prioridad	SP	Sprint
TT-01	Configurar el stack Docker Compose con Prometheus y Grafana con políticas de reinicio automático.	Alta	5	1
TT-02	Instrumentar aplicación Spring Boot de demostración con Micrometer para validar el flujo completo de scraping.	Alta	3	1
TT-04	Desarrollar pruebas unitarias para PrometheusOrquestadorService y pruebas de integración del ciclo CRUD → YAML → reload.	Media	5	5
TT-05	Validación experimental: pruebas de tiempo de respuesta API, tiempo de regeneración de YAML y onboarding multi-sistema.	Media	8	6

Nota. Product Backlog ordenado por prioridad y sprint asignado. SP = story points en escala Fibonacci. Velocidad objetivo por sprint: 16-24 SP. Autoría propia.

2.3.3 Sprints del Proyecto

El desarrollo se organizó en seis sprints de dos semanas cada uno, con una velocidad objetivo de entre 16 y 24 puntos de historia por ciclo. A continuación se describe el Sprint Goal, el Sprint Backlog y el incremento entregado en cada sprint:

Sprint 1 (Semanas 1-2): Entorno base e instrumentación.

Sprint Goal: tener el stack de monitoreo funcionando con al menos una aplicación instrumentada. Sprint Backlog: HU-01, HU-02, TT-01, TT-02 (21 SP). Incremento: Docker Compose con Prometheus y Grafana operativos; aplicación Spring Boot de demostración instrumentada con Micrometer; primer prometheus.yml generado dinámicamente al registrar el primer sistema desde la API REST.

Sprint 2 (Semanas 3-4): CRUD completo de sistemas.

Sprint Goal: el administrador puede gestionar sistemas monitoreados de forma completa. Sprint Backlog: HU-03, HU-04, HU-05, HU-06 (14 SP). Incremento: endpoints REST PUT, DELETE y PATCH /monitoreo operativos; formulario diferenciado por tipo de agente en el frontend; regeneración automática del prometheus.yml en todas las operaciones de escritura.

Sprint 3 (Semanas 5-6): Dashboard de métricas:

Sprint Goal: el administrador puede visualizar métricas en tiempo real de cualquier sistema registrado. Sprint Backlog: HU-07, HU-08, HU-09, HU-10 (21 SP). Incremento: vista Dashboard.vue con selector de sistema y rango temporal; KPIs y gráficos de series temporales diferenciados por tipo de agente; polling automático cada 15 segundos contra la API de Prometheus.

Sprint 4 (Semanas 7-8): Navegación y auditoría:

Sprint Goal: la interfaz de navegación está completa y todas las operaciones quedan registradas. Sprint Backlog: HU-11, HU-12 (7 SP). Incremento: barra lateral con acceso a Grafana y Prometheus; tabla logs operativa con registro automático de CRUD desde SistemaService.

Sprint 5 (Semanas 9-10): Auditoría y pruebas.

Sprint Goal: el registro de auditoría está implementado y las pruebas del núcleo del sistema están completas. Sprint Backlog: TT-04 (5 SP). Incremento: registro asíncrono de las operaciones administrativas en la tabla logs; pruebas unitarias de PrometheusOrquestadorService con cobertura del flujo de generación YAML; pruebas de integración del ciclo CRUD, regeneración y verificación de targets.

Sprint 6 (Semanas 11-12): Validación experimental y cierre:

Sprint Goal: el prototipo está validado con evidencias experimentales medibles. Sprint Backlog: TT-05 (8 SP). Incremento: resultados de pruebas de tiempo de respuesta de la API, tiempo de regeneración del prometheus.yml, comparación manual vs automatizado del proceso de onboarding de sistemas, y simulación de registro de múltiples sistemas simultáneos.

2.3.4 Definition of Done

Un ítem del Sprint Backlog se considera terminado (Done) cuando cumple los siguientes criterios: el código está integrado en la rama principal del repositorio y el build es exitoso; las pruebas unitarias correspondientes están implementadas y pasan; la funcionalidad ha sido verificada manualmente en el entorno de desarrollo local con Docker Compose; la documentación del código (Javadoc para el backend, JSDoc para el frontend) está actualizada; y el incremento no degrada ninguna funcionalidad entregada en sprints anteriores.

2.3.5 Métricas Ágiles del Proyecto

Las métricas de seguimiento del proceso Scrum empleadas en el proyecto son las siguientes: la velocidad por sprint (story points completados), que permite comparar la planificación con la ejecución real; el porcentaje de cumplimiento del Sprint Goal, que evalúa si el objetivo de negocio del ciclo fue alcanzado independientemente del número exacto de ítems completados; y el lead time de una historia de usuario, medido desde su incorporación al Sprint Backlog hasta su inclusión en el incremento entregado. La Tabla 5 resume los valores objetivo y realizados de estas métricas por sprint.

Tabla 5. Métricas ágiles por sprint.

Sprint	SP planificados	SP completados	Sprint Goal	Fecha inicio	Fecha fin
Sprint 1	21	21	Cumplido	Mar. semana 3	Mar. semana 4 / Abr. sem. 1
Sprint 2	14	14	Cumplido	Abr. semana 1	Abr. semana 2
Sprint 3	21	21	Cumplido	Abr. semana 3	Abr. semana 4 / May. sem. 1
Sprint 4	7	7	Cumplido	May. semana 1	May. semana 2

Sprint	SP planificados	SP completados	Sprint Goal	Fecha inicio	Fecha fin
Sprint 5	10	10	Cumplido	May. semana 3	May. semana 4 / Jun. sem. 1
Sprint 6	8	8	Cumplido	Jun. semana 1	Jun. semana 2
Total	81	81	100%	,	,

Nota. Métricas ágiles del proyecto. La velocidad promedio fue de 13.5 SP/semana, consistente con un equipo de desarrollo de una persona. Los sprints 1 y 3 fueron los de mayor carga técnica por la configuración del stack de monitoreo y la implementación del dashboard respectivamente. Autoría propia.

CAPÍTULO III: ANÁLISIS Y DISEÑO DEL SISTEMA

El presente capítulo describe el análisis y diseño completo de la herramienta de monitoreo. Se definen los requerimientos funcionales y no funcionales, se modelan los casos de uso, se presenta el diseño de arquitectura, la base de datos y la interfaz de usuario. Toda la información se deriva directamente de las funcionalidades implementadas en el prototipo, incluyendo el módulo de tracking histórico de tráfico HTTP incorporado en la etapa de desarrollo.

3.1 Análisis de Requerimientos

En esta etapa se definen los requerimientos que el sistema debe satisfacer para cumplir con el objetivo de centralizar el monitoreo de múltiples aplicaciones backend y bases de datos, incorporando además la capacidad de registrar históricamente la actividad HTTP de los sistemas monitoreados. El análisis se realizó a partir de las necesidades identificadas en el entorno empresarial descrito en el Capítulo I y se organizó siguiendo el Product Backlog de la metodología Scrum presentado en el Capítulo II.

3.2 Requerimientos Funcionales

Los requerimientos funcionales describen las capacidades que el sistema debe proveer a sus usuarios, organizados por módulo:

Módulo de Gestión de Sistemas Monitoreados:

- RF01: El sistema permitirá registrar un nuevo sistema monitoreado ingresando: alias (job name), host, puerto, tipo de agente e intervalo de scrape; y de forma condicional, la ruta de métricas para agentes Spring Boot, o los datos de conexión a la base de datos para agentes dbpostgres.
- RF02: El sistema permitirá editar los datos de un sistema monitoreado previamente registrado.
- RF03: El sistema permitirá eliminar un sistema monitoreado, solicitando confirmación explícita antes de ejecutar la operación.
- RF04: El sistema permitirá listar todos los sistemas monitoreados registrados, mostrando alias, host, puerto, tipo de agente y estado de monitoreo.
- RF05: El sistema permitirá pausar o reanudar el monitoreo de un sistema registrado, cambiando su atributo monitoreado entre true y false sin eliminar el registro.

- RF06: El sistema diferenciará el formulario de registro según el tipo de agente: para Spring Boot mostrará el campo Metrics Path; para dbpostgres mostrará los campos Nombre Base de Datos, Usuario DB y Contraseña DB.

Módulo de Generación Dinámica de Configuración:

- RF07: El sistema regenerará automáticamente el archivo prometheus.yml cada vez que se registre, edite, elimine o cambie el estado de monitoreo de un sistema, sin intervención manual del operador.
- RF08: El archivo prometheus.yml generado incluirá únicamente los sistemas cuyo atributo monitoreado sea true, utilizando el alias como job_name y el host:puerto como target.
- RF09: Para sistemas de tipo Spring Boot, el archivo generado incluirá la ruta de métricas (metrics_path) configurada en el campo path.
- RF10: Para sistemas de tipo dbpostgres, el backend expondrá las métricas de la base de datos mediante un endpoint exportador propio (/api/metrics/postgres/{id}), construido con la librería Micrometer a partir de los datos de conexión del sistema registrado, sin requerir un exporter externo.

Módulo de Visualización en Tiempo Real:

- RF11: El sistema presentará un dashboard con selector de sistema monitoreado y selector de rango temporal (15 minutos, 1 hora, 6 horas, 24 horas).
- RF12: El dashboard diferenciará las métricas según el tipo de agente: para Spring Boot mostrará métricas de la JVM, peticiones HTTP y pool de conexiones HikariCP; para dbpostgres mostrará conexiones activas, tamaño de la base de datos, ratio de aciertos de caché, transacciones (commits y rollbacks), filas afectadas, bloqueos, deadlocks, tuplas muertas, checkpoints y archivos temporales.
- RF13: El sistema proveerá acceso directo a las interfaces de Prometheus y Grafana mediante enlaces externos en la barra lateral de navegación.

Módulo de Tracking Histórico de Tráfico HTTP:

- RF14: El sistema detectará y persistirá, cada 60 segundos, la cantidad de peticiones HTTP respondidas por los sistemas de tipo Spring Boot monitoreados, clasificadas por código de estado HTTP, método y ruta (URI templated de Micrometer). La detección se basa en el cálculo del incremento de peticiones (la diferencia entre dos lecturas consecutivas de los contadores acumulados expuestos por Micrometer), sin depender de la retención del TSDB de Prometheus.
- RF15: El sistema excluirá del tracking las peticiones al endpoint /actuator/prometheus para eliminar el ruido generado por el propio scraping de Prometheus (~4 peticiones por ciclo por sistema).
- RF16: El sistema permitirá consultar el historial de tráfico HTTP con filtros por sistema, grupo de status (2xx/3xx/4xx/5xx), fecha y rango horario, con respuesta paginada.
- RF17: El sistema presentará un resumen del tráfico HTTP del período consultado incluyendo: total de peticiones, total de errores 4xx, total de errores 5xx, porcentaje de tasa de error y hora pico de errores.
- RF18: El sistema presentará un gráfico de barras apiladas por hora mostrando la distribución del tráfico por grupo de status (2xx/3xx/4xx/5xx) en el período consultado.
- RF19: El sistema ejecutará una limpieza nocturna automática de registros de tráfico HTTP con antigüedad superior al período de retención configurable (por defecto 7 días).

Módulo de Auditoría:

- RF20: El sistema registrará en la tabla logs cada operación de creación, edición, eliminación o cambio de estado de un sistema monitoreado, almacenando fecha, nivel, origen, acción, detalle e IP del cliente.

3.3 Requerimientos No Funcionales

Los requerimientos no funcionales definen los atributos de calidad del sistema:

- RNF01: Disponibilidad: el stack de monitoreo debe permanecer operativo de forma continua. El despliegue con Docker Compose y política restart: unless-stopped garantiza la disponibilidad de los contenedores ante fallos.
- RNF02: Rendimiento: la API REST debe responder en menos de 2 segundos ante operaciones CRUD. La regeneración del prometheus.yml debe completarse en menos de 500 milisegundos tras cualquier operación de escritura.
- RNF03: Usabilidad: la interfaz web debe permitir registrar un nuevo sistema monitoreado en un máximo de cinco pasos, con retroalimentación visual inmediata ante operaciones exitosas o erróneas.
- RNF04: Resiliencia: el módulo de tracking HTTP debe ser tolerante a fallos individuales por sistema; la falla de recolección de un sistema no debe interrumpir el ciclo de recolección de los demás. Las consultas a Prometheus tienen un timeout máximo de 10 segundos para evitar que un Prometheus caído bloquee el hilo del scheduler.
- RNF05: Seguridad: el backend validará todas las entradas en los endpoints REST para prevenir la inyección de configuraciones maliciosas en el prometheus.yml. El cifrado de las contraseñas de conexión a las bases de datos monitoreadas queda como mejora pendiente.
- RNF06: Portabilidad: el stack de monitoreo debe poder desplegarse en cualquier máquina con Docker instalado mediante un único comando (docker compose up), independientemente del sistema operativo.
- RNF07: Mantenibilidad: el backend debe seguir la arquitectura en capas de Spring Boot (controller, service, repository) para facilitar la extensión hacia nuevos tipos de agentes sin modificar componentes existentes.
- RNF08: Escalabilidad: el mecanismo de generación del prometheus.yml debe soportar múltiples sistemas registrados sin degradación de rendimiento, dado que la escritura del archivo es lineal respecto al número de sistemas activos.
- RNF09: Trazabilidad: el histórico de tráfico HTTP debe sobrevivir a la eliminación del sistema monitoreado, dado que los registros de la tabla

eventos_http almacenan el alias del sistema de forma desnormalizada, sin clave foránea.

- RNF10: Compatibilidad: el frontend debe ser funcional en Google Chrome (versión 120+) y Mozilla Firefox (versión 121+) sin extensiones adicionales.

3.4 Casos de Uso

Los casos de uso se agrupan en cuatro módulos funcionales principales:

R1: Gestión de Sistemas Monitoreados

R2: Generación Dinámica de Configuración de Prometheus

R3: Visualización de Métricas en Tiempo Real

R4: Consulta de Historial de Tráfico HTTP

3.4.1 Actores del Sistema

Administrador de Monitoreo: actor humano principal. Tiene acceso completo al CRUD de sistemas, a la visualización del dashboard en tiempo real y a la consulta del historial de tráfico HTTP. Es el único actor humano operativo del prototipo en su versión actual.

Prometheus Server: actor externo no humano. Lee periódicamente el prometheus.yml generado por el backend para hacer scraping de los targets registrados. Es también la fuente de datos consultada por el módulo de tracking HTTP mediante la API PromQL.

Sistema Externo Monitoreado: actor externo no humano. Representa cualquier aplicación Spring Boot instrumentada con Micrometer o instancia de PostgreSQL cuyas métricas son recolectadas por Prometheus.

3.4.2 Diagrama de Casos de Uso General

El diagrama de casos de uso general ofrece una visión global de las funcionalidades del sistema y de los actores que interactúan con él. La Figura 1 presenta este diagrama, en el que se identifican los cuatro módulos funcionales del prototipo y las relaciones entre el administrador de monitoreo y los actores externos que participan en el proceso.

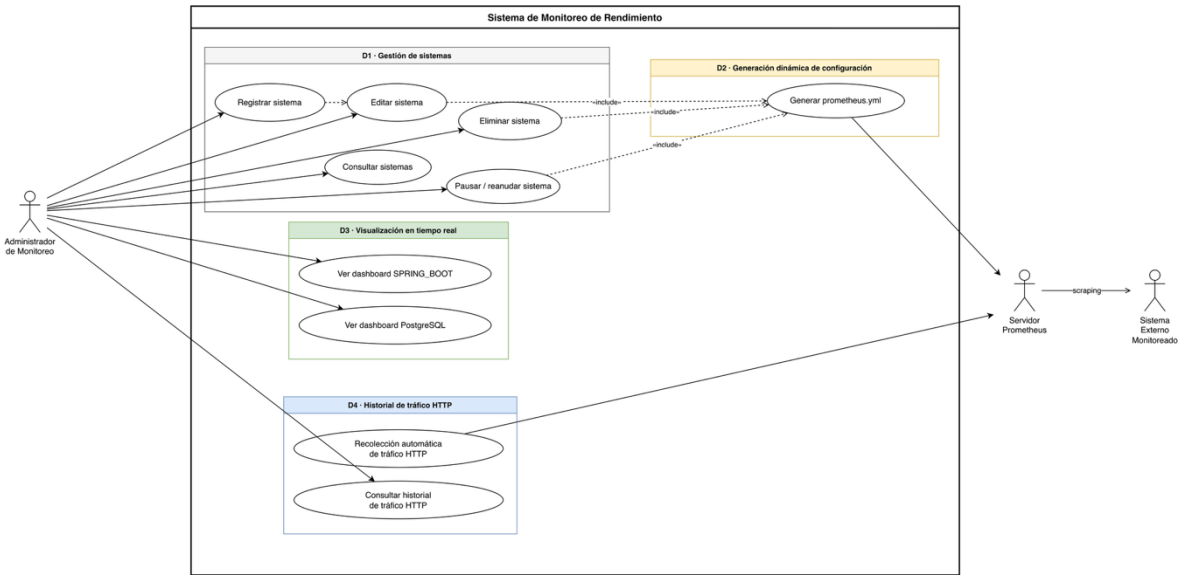


Figura 1. Diagrama de casos de uso general.

Nota. Elaboración propia

3.5 Diagramas de Casos de Uso Siguiente Nivel

En esta sección se detallan los diagramas de casos de uso de siguiente nivel, que descomponen las funcionalidades generales del sistema en las interacciones específicas asociadas a cada requerimiento. A continuación se presentan los cuatro grupos de casos de uso definidos.

3.5.1 Casos de Uso Siguiente Nivel R1: Gestión de Sistemas Monitoreados

La Figura 2 muestra los casos de uso detallados de la gestión de sistemas monitoreados.

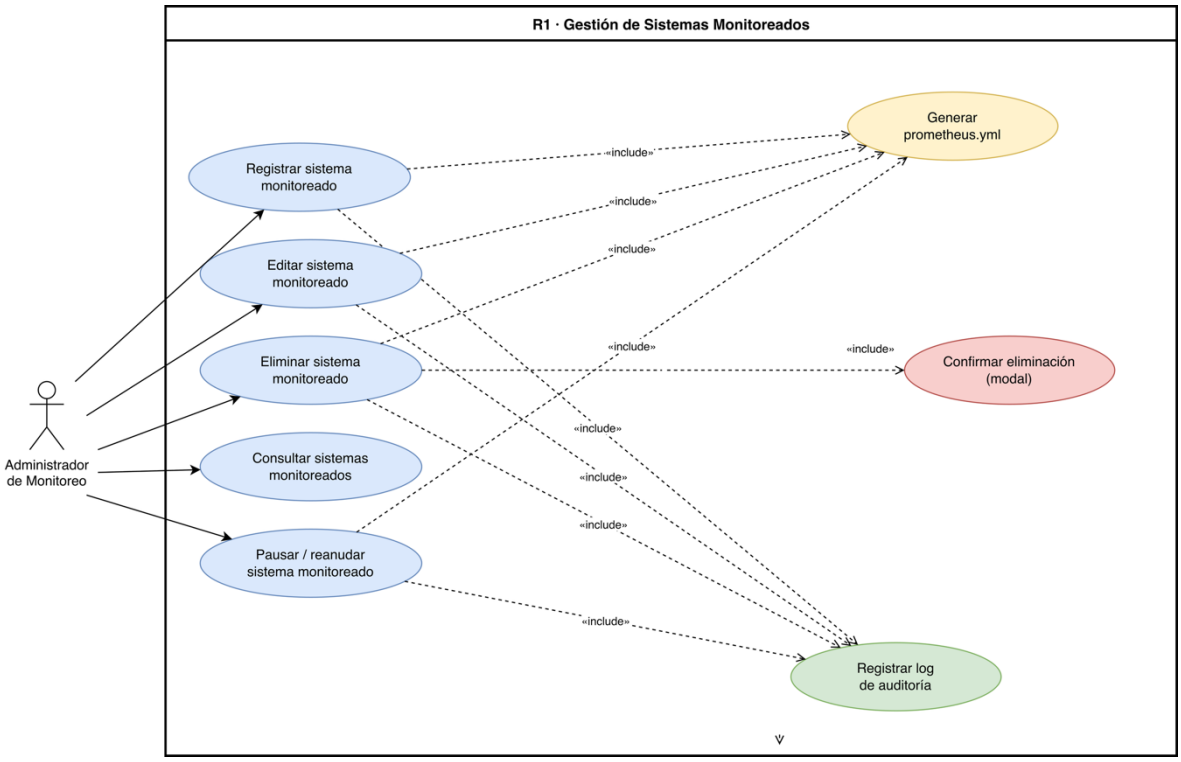


Figura 2. Casos de uso siguiente nivel R1: Gestión de Sistemas Monitoreados.

Nota. Elaboración Propia.

Ingreso de Sistema Monitoreado

Descripción: El sistema permite al Administrador registrar un nuevo sistema cuyas métricas serán recolectadas por Prometheus.

Actor: Administrador de Monitoreo

Flujo principal:

- El actor accede a la vista Sistemas desde la barra lateral de navegación.
- El sistema muestra el listado de sistemas actualmente registrados.
- El actor selecciona la opción Nuevo Sistema.
- El sistema despliega el formulario con los campos: Tipo de Sistema/Agente, Alias, Host, Puerto e Intervalo Scrape.
- El actor selecciona el tipo de agente (Spring Boot o dbpostgres).
- El sistema muestra los campos condicionales: para Spring Boot presenta el campo Metrics Path; para dbpostgres presenta Nombre BD, Usuario DB y Contraseña DB.

- El actor completa los campos requeridos y selecciona Guardar.
- El sistema valida que el alias no esté duplicado en la base de datos. E1.
- El sistema persiste el nuevo sistema en la tabla sistemas de PostgreSQL.
- El sistema regenera automáticamente el prometheus.yml incluyendo el nuevo target.
- Para sistemas de tipo Spring Boot, el HttpTrackingService incorpora el nuevo sistema en el próximo ciclo de recolección de incrementos HTTP.
- El sistema muestra una notificación de éxito al actor.

Excepciones:

- E1: Ya existe un sistema con el mismo alias. El sistema muestra un error y no persiste el registro.

Modificación de Sistema Monitoreado

Descripción: El sistema permite editar los datos de un sistema previamente registrado.

Actor: Administrador de Monitoreo

Flujo principal:

- El actor accede a la vista Sistemas y selecciona Editar sobre el sistema a modificar.
- El sistema carga el formulario con los datos actuales del sistema.
- El actor modifica los campos requeridos y selecciona Guardar.
- El sistema valida que el alias modificado no genere un duplicado. E1.
- El sistema actualiza el registro en PostgreSQL.
- El sistema regenera automáticamente el prometheus.yml con los datos actualizados.
- El sistema muestra una notificación de éxito.

Excepciones:

- E1: El alias modificado ya existe para otro sistema. El sistema muestra un error y no persiste los cambios.

Eliminación de Sistema Monitoreado

Descripción: El sistema permite eliminar de forma permanente un sistema registrado.

Actor: Administrador de Monitoreo

Flujo principal:

- El actor accede a la vista Sistemas y selecciona Eliminar sobre el sistema a borrar.
- El sistema despliega un modal de confirmación.
- El actor confirma la eliminación.
- El sistema elimina el registro de la tabla sistemas en PostgreSQL.
- El sistema regenera el prometheus.yml excluyendo el sistema eliminado.
- El histórico de tráfico HTTP del sistema eliminado se conserva en la tabla eventos_http (desnormalizado, sin FK).
- El sistema muestra una notificación de éxito.

Excepciones:

- E1: El actor cancela en el modal. El sistema cierra el modal sin modificaciones.

Consulta y Cambio de Estado de Sistemas

Descripción: El sistema permite consultar sistemas registrados y pausar o reanudar su monitoreo.

Actor: Administrador de Monitoreo

Flujo principal:

- El actor accede a la vista Sistemas.
- El sistema recupera la lista de sistemas desde la base de datos y la muestra con alias, host, tipo y estado.
- El actor selecciona Pausar sobre un sistema activo, o Reanudar sobre uno pausado.
- El sistema actualiza el atributo monitoreado (true ↔ false) en PostgreSQL.
- El sistema regenera el prometheus.yml incluyendo o excluyendo el sistema según su nuevo estado.
- Los sistemas pausados son excluidos del ciclo de recolección HTTP en el próximo tick del HttpTrackingService.

- El sistema muestra una notificación de éxito con el nuevo estado.

Excepciones:

- E1: No existen sistemas registrados. El sistema muestra un mensaje indicando que no hay sistemas.

3.5.2 Casos de Uso Siguiente Nivel R2: Generación Dinámica de Configuración

La Figura 3 ilustra el caso de uso de generación dinámica del prometheus.yml.

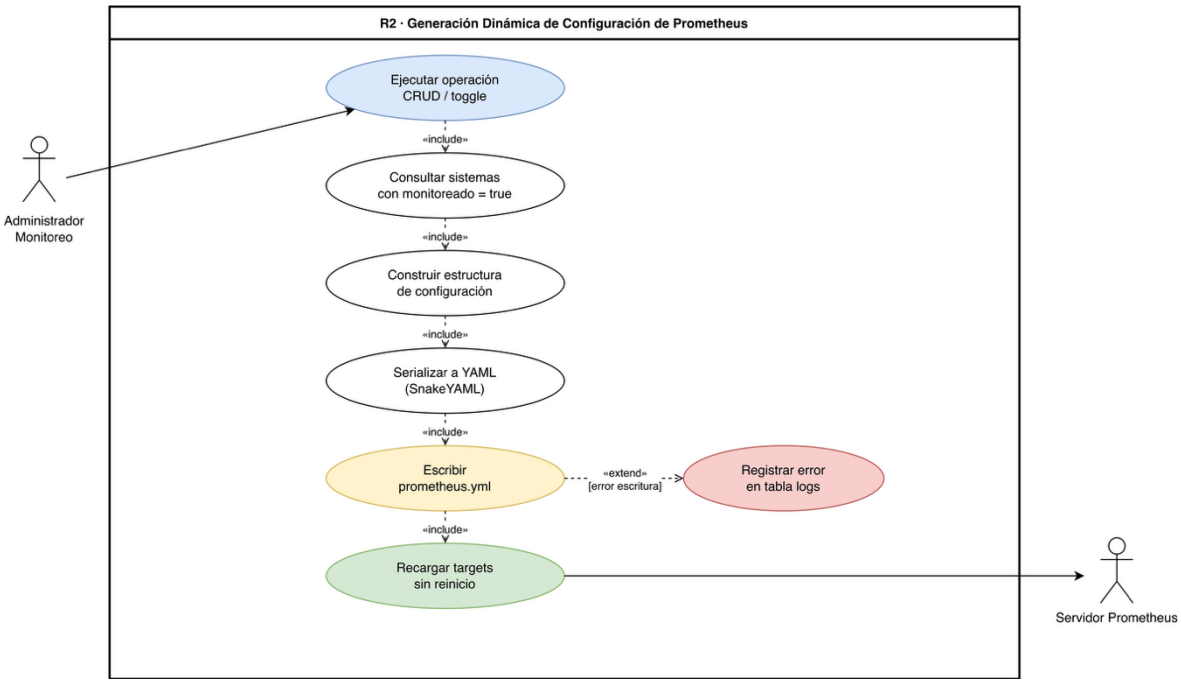


Figura 3. Casos de uso siguiente nivel R2: Generación Dinámica de Configuración.

Nota. Elaboración Propia.

Generación Automática del prometheus.yml

Descripción: El backend genera y sobreescribe el prometheus.yml tras cada cambio en el registro de sistemas.

Actor: Backend Spring Boot (desencadenado por operación CRUD del Administrador)

Flujo principal:

- El SistemaService recibe una operación de creación, edición, eliminación o cambio de estado.

- El servicio consulta la base de datos y recupera todos los sistemas con monitoreado = true.
- Por cada sistema activo construye un job: alias como job_name, host:puerto como target; para Spring Boot añade metrics_path.
- El servicio serializa la estructura al formato YAML requerido por Prometheus mediante construcción programática de texto.
- El backend sobreescribe el prometheus.yml en el volumen compartido con el contenedor de Prometheus.
- Prometheus detecta el cambio en su próximo ciclo de recarga y actualiza sus targets sin reinicio.

Excepciones:

- E1: Sin permisos de escritura sobre el directorio del prometheus.yml. Se registra el error en logs y se retorna HTTP 500 al frontend.

3.5.3 Casos de Uso Siguiente Nivel R3: Visualización de Métricas en Tiempo Real

La Figura 4 presenta los casos de uso del dashboard de métricas en tiempo real.

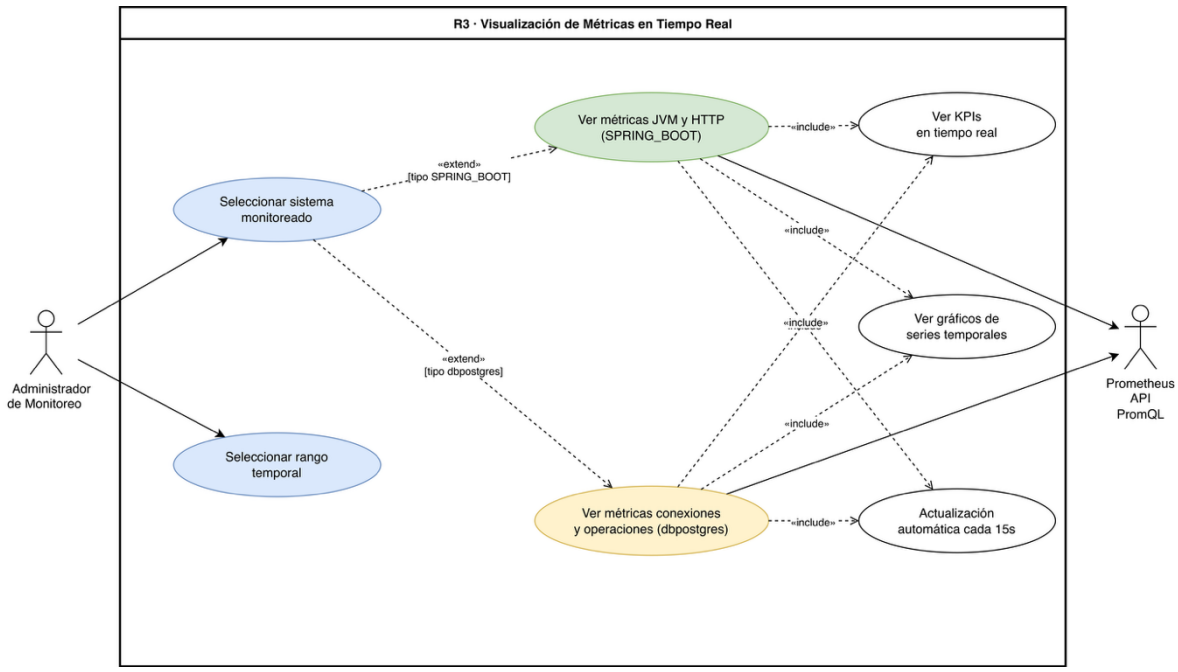


Figura 4. Casos de uso siguiente nivel R3: Visualización de Métricas en Tiempo Real.

Nota. Elaboración Propia.

Visualización de Métricas de Aplicación Spring Boot

Descripción: El sistema presenta el dashboard de métricas en tiempo real de un sistema Spring Boot seleccionado.

Actor: Administrador de Monitoreo

Flujo principal:

- El actor accede a la vista Dashboard y selecciona un sistema de tipo Spring Boot.
- El actor selecciona el rango temporal (15 min, 1h, 6h, 24h).
- El frontend consulta la API de Prometheus con PromQL para obtener las métricas del sistema en el rango indicado.
- El sistema presenta KPIs: uso de memoria heap, hilos activos, tasa de solicitudes HTTP, tasa de errores, tiempo de respuesta p95 y conexiones HikariCP activas.
- El dashboard se actualiza automáticamente cada 15 segundos.

Excepciones:

- E1: El sistema no está activo o Prometheus no completó el primer scraping. El dashboard muestra paneles vacíos con mensaje de ausencia de datos.

Visualización de Métricas de Base de Datos PostgreSQL

Descripción: El sistema presenta el dashboard de métricas de un sistema dbpostgres seleccionado.

Actor: Administrador de Monitoreo

Flujo principal:

- El actor accede al Dashboard y selecciona un sistema de tipo dbpostgres.
- El frontend consulta la API de Prometheus para obtener métricas del endpoint exportador del backend.
- El sistema presenta: conexiones activas e inactivas, tamaño de la base de datos, operaciones de lectura/escritura por tabla, estadísticas de bloqueos y deadlocks.
- El dashboard se actualiza automáticamente cada 15 segundos.

Excepciones:

- E1: El endpoint exportador del backend no tiene conectividad con PostgreSQL. El dashboard muestra paneles vacíos.

3.5.4 Casos de Uso Siguiente Nivel R4: Consulta de Historial de Tráfico HTTP

La Figura 5 presenta los casos de uso del módulo de tracking histórico de tráfico HTTP. Este módulo opera en dos dimensiones: una recolección automática en segundo plano ejecutada por el HttpTrackingService, y una consulta interactiva desde la vista TraficoHttp.vue.

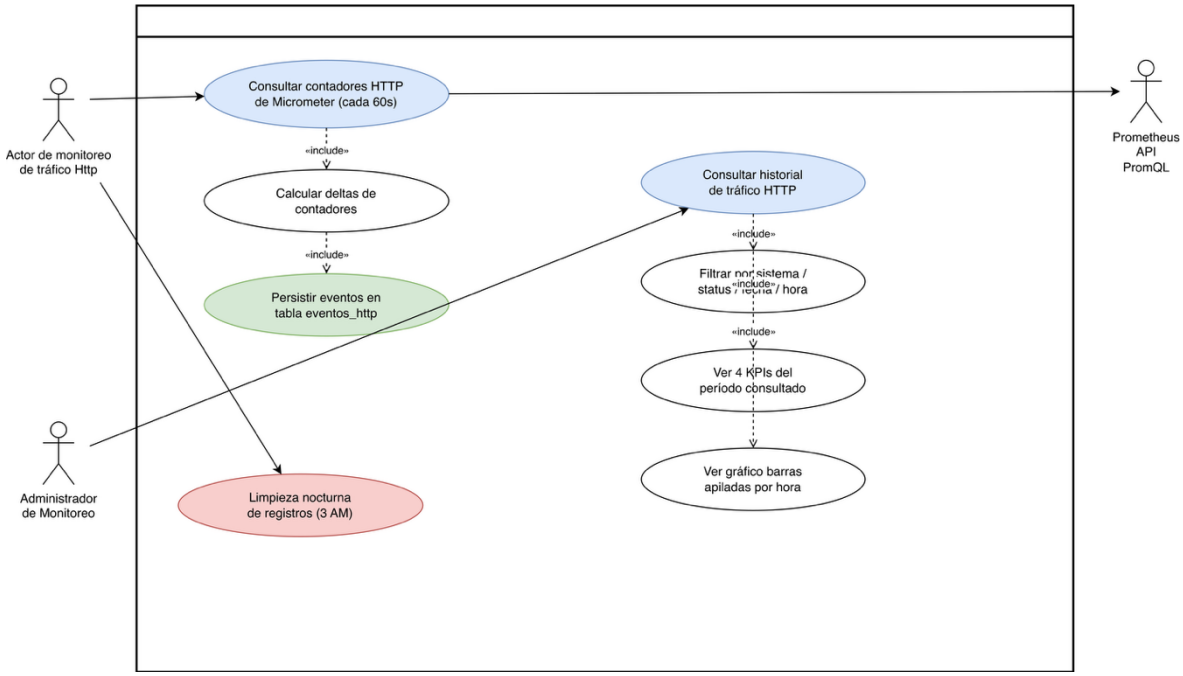


Figura 5. Casos de uso siguiente nivel R4: Consulta de Historial de Tráfico HTTP.

Nota. Elaboración Propia.

Recolección Automática de Tráfico HTTP

Descripción: El HttpTrackingService detecta y persiste cada 60 segundos las peticiones HTTP de los sistemas Spring Boot activos, calculando incrementos sobre los contadores de Micrometer.

Actor: HttpTrackingService (Spring @Scheduled, automático)

Flujo principal:

- Cada 60 segundos, el scheduler recupera de la base de datos los sistemas con monitoreado = true y tipo distinto de dbpostgres.

- Para cada sistema activo, consulta a Prometheus: sum by (status, method, uri) (http_server_requests_seconds_count{job=ALIAS, uri!~/actuator.*}).
- Compara los valores actuales contra la lectura de referencia guardada en memoria (ConcurrentHashMap). Si actual < previo (reinicio de la app), reinicia la lectura de referencia desde cero.
- Por cada combinación de (status, metodo, ruta) cuyo contador haya incrementado, persiste una fila en la tabla eventos_http con la cantidad de peticiones detectadas en el intervalo.
- Actualiza la lectura de referencia en memoria con los valores actuales.
- Descarta de la lectura de referencia las entradas de sistemas que ya no están activos, previniendo fugas de memoria.

Excepciones:

- E1: Prometheus no responde dentro del timeout (10 segundos). Se omite la recolección de ese sistema en el ciclo actual y se continúa con los demás.
- E2: Error de red o sistema caído. La excepción se captura individualmente por sistema; los demás sistemas del ciclo no se ven afectados.
- E3: Primer ciclo tras arranque del orquestador. Solo se establece la lectura de referencia; no se persisten eventos (~1 minuto de intervalo inicial sin registro). Es una concesión aceptable para un sistema de monitoreo histórico.

Consulta Filtrada de Historial de Tráfico HTTP

Descripción: El sistema permite al Administrador consultar el historial de tráfico HTTP con múltiples filtros combinables.

Actor: Administrador de Monitoreo

Flujo principal:

- El actor accede a la vista Tráfico HTTP desde la barra lateral de navegación.
- El sistema carga el selector de sistema (solo sistemas Spring Boot), el selector de fecha, los selectores de hora desde/hasta y los chips de grupo de status (Todos / 2xx / 3xx / 4xx / 5xx).

- El sistema carga por defecto la fecha del día actual en hora local (UTC-5) y el resumen del día.
- El actor ajusta los filtros deseados.
- El sistema actualiza los 4 KPIs (total peticiones, errores 4xx, errores 5xx con porcentaje de tasa de error, y hora pico de errores) y el gráfico de barras apiladas por hora.
- La tabla paginada muestra: hora, sistema, método (badge), ruta, status (badge semafórico) y cantidad.
- Los chips de grupo de status filtran únicamente la tabla; el gráfico y los KPIs muestran siempre el panorama completo del rango para mantener el contexto.
- El dashboard se actualiza automáticamente cada 60 segundos.

Excepciones:

- E1: No existen eventos para los filtros seleccionados. El sistema muestra los KPIs en cero y la tabla vacía con mensaje informativo.

3.6 Modelado del Sistema

3.6.1 Diagrama Entidad-Relación (ERD)

La Figura 6 presenta el diagrama entidad-relación del prototipo. El modelo consta de siete tablas en 3NF: las seis originales del sistema de gestión de monitoreo (sistemas, usuarios, roles, usuarios_roles, tipos_agente, logs) más la tabla eventos_http incorporada con el módulo de tracking histórico de tráfico HTTP.

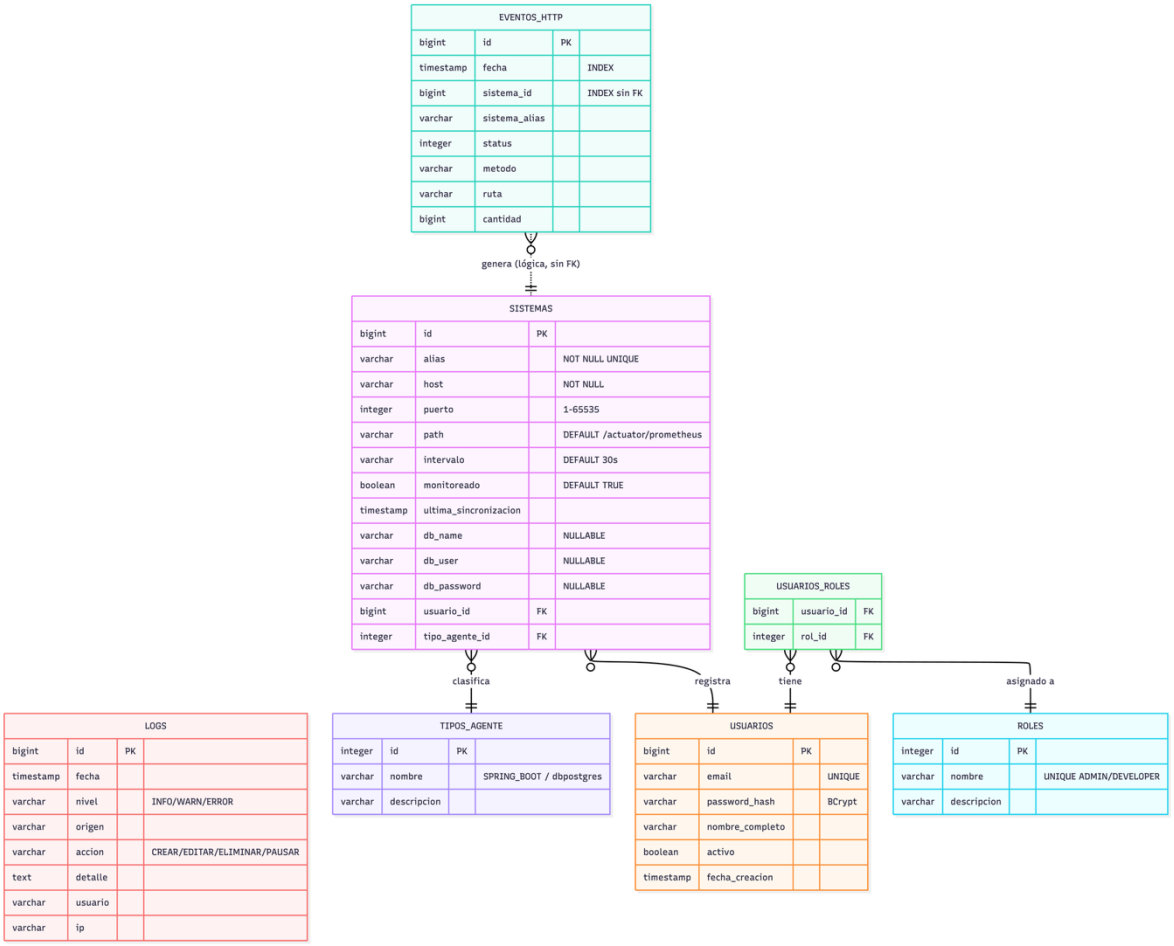


Figura 6. Diagrama Entidad-Relación (ERD).

Nota. Elaboración Propia.

Tabla 6. Diccionario de datos, tabla sistemas.

Campo	Tipo	Restricción	Descripción
id	BIGINT	PK, AUTO_INCREMENT	Identificador único del sistema
alias	VARCHAR	NOT NULL, UNIQUE	Nombre del job en Prometheus (job_name)
host	VARCHAR	NOT NULL	Dirección IP o nombre de host

Campo	Tipo	Restricción	Descripción
puerto	INTEGER	NOT NULL (1-65535)	Puerto de exposición de métricas
path	VARCHAR(100)	DEFAULT /actuator/prometheus	Ruta del endpoint de métricas (Spring Boot)
intervalo	VARCHAR(10)	DEFAULT 30s	Intervalo de scraping de Prometheus
monitoreado	BOOLEAN	DEFAULT TRUE	Indica si el sistema está activo en el monitoreo
ultima_sincronizacion	TIMESTAMP	NULLABLE	Fecha/hora del último prometheus.yml generado
db_name	VARCHAR	NULLABLE	Nombre de BD (solo tipo dbpostgres)
db_user	VARCHAR	NULLABLE	Usuario de BD (solo tipo dbpostgres)
db_password	VARCHAR	NULLABLE	Contraseña de BD, texto plano (mejora pendiente)

Campo	Tipo	Restricción	Descripción
usuario_id	BIGINT	FK → usuarios.id	Usuario que registró el sistema
tipo_agente_id	INTEGER	FK → tipos_agente.id, NOT NULL	Tipo de agente del sistema

Nota. Tabla principal del modelo. Los campos db_name, db_user y db_password son exclusivos de sistemas de tipo dbpostgres. Autoría propia.

Tabla 7. Diccionario de datos, tabla eventos_http.

Campo	Tipo	Restricción	Descripción
id	BIGINT	PK, AUTO_INCREMENT	Identificador único del evento
fecha	TIMESTAMP	NOT NULL, INDEX	Minuto exacto en que se detectó el movimiento del contador
sistema_id	BIGINT	NOT NULL, INDEX (sin FK)	ID del sistema, desnormalizado para conservar el histórico si el sistema es eliminado
sistema_alias	VARCHAR	NOT NULL	Alias del sistema, desnormalizado por la misma razón
status	INTEGER	NOT NULL	Código HTTP: 200, 400, 404, 500, etc.

Campo	Tipo	Restricción	Descripción
metodo	VARCHAR(10)	NOT NULL	Método HTTP: GET, POST, PUT, DELETE, PATCH
ruta	VARCHAR(255)	NOT NULL	URI templated de Micrometer (ej. /api/pagos/{id})
cantidad	BIGINT	NOT NULL	Número de peticiones detectadas en el intervalo de 1 minuto

Nota. Tabla del módulo de tracking HTTP. No tiene clave foránea hacia sistemas para que el histórico de tráfico sobreviva a la eliminación del sistema monitoreado (RNF09). Los índices sobre fecha y sistema_id optimizan las consultas filtradas. Autoría propia.

Tabla 8. Diccionario de datos, tabla usuarios.

Campo	Tipo	Restricción	Descripción
id	BIGINT	PK, AUTO_INCREMENT	Identificador único del usuario
email	VARCHAR	NOT NULL, UNIQUE	Correo electrónico (credencial de acceso)
password_hash	VARCHAR	NOT NULL	Contraseña del usuario (almacenada de forma cifrada)
nombre_completo	VARCHAR	NOT NULL	Nombre completo del usuario

Campo	Tipo	Restricción	Descripción
activo	BOOLEAN	DEFAULT TRUE	Indica si el usuario está habilitado
fecha_creacion	TIMESTAMP	NOT NULL	Fecha y hora de registro del usuario

Nota. Tabla de usuarios del sistema. La estructura queda preparada para una futura gestión de acceso, fuera del alcance del prototipo. Autoría propia.

Tabla 9. Diccionario de datos, tabla roles.

Campo	Tipo	Restricción	Descripción
id	INTEGER	PK, AUTO_INCREMENT	Identificador único del rol
nombre	VARCHAR	NOT NULL, UNIQUE	Nombre del rol (ej: ADMIN, DEVELOPER)
descripcion	VARCHAR	NULLABLE	Descripción del propósito del rol

Nota. Catálogo de roles del sistema. Estructura preparada para una futura diferenciación de perfiles de acceso, fuera del alcance del prototipo. Autoría propia.

Tabla 10. Diccionario de datos, tabla usuarios_roles.

Campo	Tipo	Restricción	Descripción
usuario_id	BIGINT	FK → usuarios.id, NOT NULL	Referencia al usuario
rol_id	INTEGER	FK → roles.id, NOT NULL	Referencia al rol asignado

Nota. Tabla pivote de la relación ManyToMany entre usuarios y roles. Autoría propia.

Tabla 11. Diccionario de datos, tabla tipos_agente.

Campo	Tipo	Restricción	Descripción
id	INTEGER	PK, AUTO_INCREMENT	Identificador único del tipo de agente
nombre	VARCHAR	NOT NULL, UNIQUE	Identificador interno (Spring Boot, dbpostgres)
descripcion	VARCHAR	NULLABLE	Descripción del tipo de agente

Nota. Catálogo de tipos de agente. El campo nombre determina el comportamiento del formulario y la configuración del prometheus.yml. Autoría propia.

Tabla 12. Diccionario de datos, tabla logs.

Campo	Tipo	Restricción	Descripción
id	BIGINT	PK, AUTO_INCREMENT	Identificador único del registro de auditoría
fecha	TIMESTAMP	NOT NULL	Fecha y hora del evento
nivel	VARCHAR	NOT NULL	Nivel del evento (INFO, WARN, ERROR)
origen	VARCHAR	NOT NULL	Componente que generó el evento
accion	VARCHAR	NOT NULL	Tipo de operación (CREAR, EDITAR, ELIMINAR, PAUSAR)

Campo	Tipo	Restricción	Descripción
detalle	TEXT	NULLABLE	Descripción detallada del evento
usuario	VARCHAR	NULLABLE	Email del usuario que ejecutó la operación
ip	VARCHAR	NULLABLE	Dirección IP del cliente

Nota. Tabla de auditoría de operaciones CRUD sobre sistemas monitoreados. Autoría propia.

3.7 Diseño de la Arquitectura del Sistema

El prototipo adopta una arquitectura de tres capas: el frontend Vue.js como capa de presentación, el backend Spring Boot como capa de lógica de negocio y datos, y el stack de monitoreo (Prometheus y Grafana) como capa de recolección y visualización. A estas capas se agrega el módulo de tracking HTTP, que opera como un servicio interno del backend con un ciclo de recolección programado independiente del flujo de solicitudes HTTP normales.

3.7.1 Diagrama de Arquitectura General

La Figura 7 presenta el diagrama de arquitectura general del prototipo, mostrando todos los componentes y sus relaciones de comunicación.

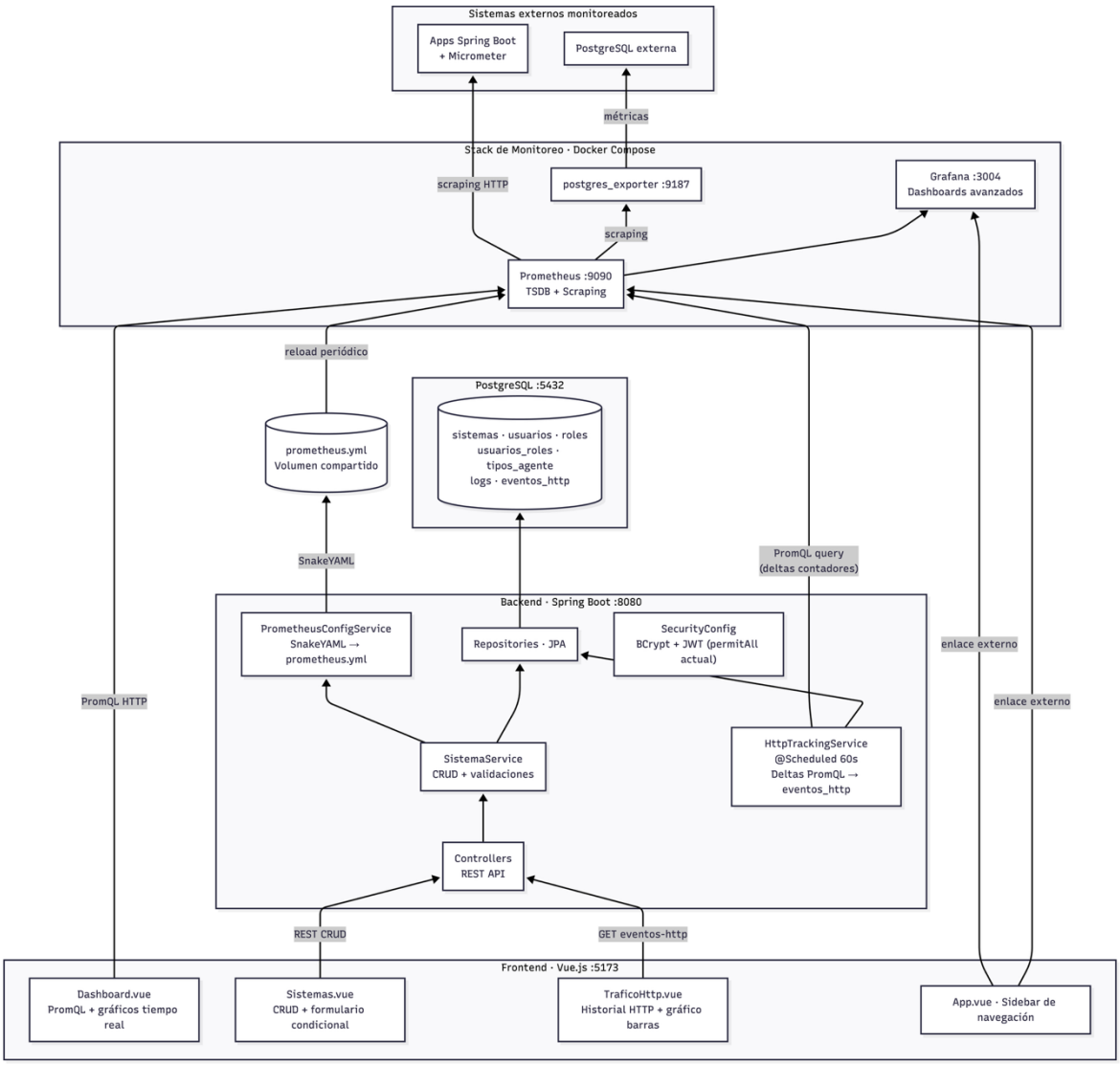


Figura 7. Diagrama de Arquitectura General.

Nota. Elaboración Propia.

Backend Spring Boot (puerto 8080): expone la API REST para el CRUD de sistemas monitoreados. El PrometheusOrquestadorService genera dinámicamente el prometheus.yml mediante construcción programática de YAML. El HttpTrackingService ejecuta un job programado cada 60 segundos que consulta la API PromQL de Prometheus para calcular incrementos de los contadores HTTP y persiste los resultados en la tabla eventos_http de PostgreSQL. La anotación @EnableScheduling activa ambos schedulers (@Scheduled del HttpTrackingService y la limpieza nocturna).

Frontend Vue.js (puerto 5173): aplicación SPA con tres vistas: Dashboard.vue (métricas en tiempo real), Sistemas.vue (CRUD) y TraficoHttp.vue (historial HTTP).

Consulta la API REST del backend y directamente la API HTTP de Prometheus con PromQL.

Prometheus (puerto 9090): sistema de recolección TSDB. Lee el prometheus.yml y hace scraping de los targets. Expone una API HTTP consultada tanto por el frontend (para el dashboard) como por el HttpTrackingService del backend (para el tracking de incrementos).

Grafana (puerto 3004): plataforma de visualización avanzada accesible desde el enlace externo del sidebar.

Endpoint exportador del backend (/api/metrics/postgres/{id}): traduce las métricas de cada base de datos PostgreSQL monitoreada al formato Prometheus mediante Micrometer, ejecutándose dentro del propio backend sin requerir un exporter externo.

PostgreSQL (puerto 5432): base de datos del backend. Almacena sistemas, usuarios, roles, logs de auditoría y ahora también los eventos_http del módulo de tracking.

3.7.2 Arquitectura en Capas del Backend Spring Boot

El backend implementa la arquitectura en capas estándar de Spring Boot:

Capa Controller: recibe las peticiones HTTP y delega a la capa de servicio. Endpoints REST: GET/POST/PUT/DELETE /api/sistemas, PATCH /api/sistemas/{id}/monitoreo, GET /api/eventos-http y GET /api/eventos-http/resumen.

Capa Service: contiene la lógica de negocio. SistemaService gestiona el CRUD e invoca al PrometheusOrquestadorService. PrometheusOrquestadorService genera el prometheus.yml mediante construcción programática de texto con escape de caracteres especiales. HttpTrackingService ejecuta el ciclo de recolección de incrementos de tráfico HTTP programado, manteniendo la lectura de referencia en memoria con un ConcurrentHashMap y persistiendo los eventos detectados mediante saveAll().

Capa Repository: abstrae el acceso a PostgreSQL mediante JpaRepository. Incluye EventoHttpRepository con consultas JPQL parametrizadas para los filtros de la vista de tráfico HTTP y un método deleteByFechaBefore() para la limpieza periódica.

3.8 Diseño de la Base de Datos

El modelo relacional se fundamenta en la Tercera Forma Normal (3NF) y consta de siete tablas. La tabla eventos_http introduce un patrón de desnormalización deliberada: los

campos sistema_id y sistema_alias se almacenan sin clave foránea hacia la tabla sistemas, siguiendo el mismo criterio de la tabla logs. Esta decisión garantiza que el historial de tráfico HTTP sobreviva a la eliminación de un sistema monitoreado, preservando la trazabilidad de los datos aunque el sistema origen ya no exista en el registro (RNF09).

Los índices sobre los campos fecha y sistema_id de la tabla eventos_http responden a los patrones de consulta del módulo: la vista TraficoHttp.vue filtra principalmente por fecha y sistema, por lo que estos índices reducen el costo de las consultas filtradas sobre un volumen de datos que crece linealmente con el número de sistemas activos y la cantidad de días de retención configurada.

La retención de datos del módulo de tracking se gestiona mediante una limpieza nocturna programada a las 3:00 AM, que elimina registros de eventos_http con antigüedad superior al período de retención configurable (por defecto 7 días), manteniendo el volumen de la tabla acotado a un horizonte operativamente relevante.

3.9 Diseño de la Interfaz de Usuario

En esta sección se describe el diseño de la interfaz de usuario del sistema, comenzando por su estructura de navegación y continuando con el detalle de cada una de las vistas principales que componen la aplicación.

El frontend del prototipo está compuesto por tres vistas principales accesibles desde una barra lateral de navegación persistente. El módulo de tracking HTTP agrega la vista TraficoHttp.vue como tercera vista del sistema, complementando el dashboard de métricas en tiempo real con un análisis histórico persistente de la actividad HTTP.

3.9.1 Estructura de Navegación

La barra lateral de navegación persistente (App.vue) contiene:

- Dashboard (ruta /): visualización de métricas en tiempo real de Prometheus.
- Sistemas (ruta /sistemas): gestión del CRUD de sistemas monitoreados.
- Tráfico HTTP (ruta /trafico): historial de actividad HTTP clasificada por status, método y ruta.
- Grafana (enlace externo, localhost:3004): dashboard avanzado de Grafana en nueva pestaña.

- Prometheus (enlace externo, localhost:9090): interfaz de Prometheus en nueva pestaña.

La Figura 8 presenta el diagrama de navegación del frontend, que muestra las rutas principales de la aplicación y la forma en que el usuario transita entre las distintas vistas del sistema.

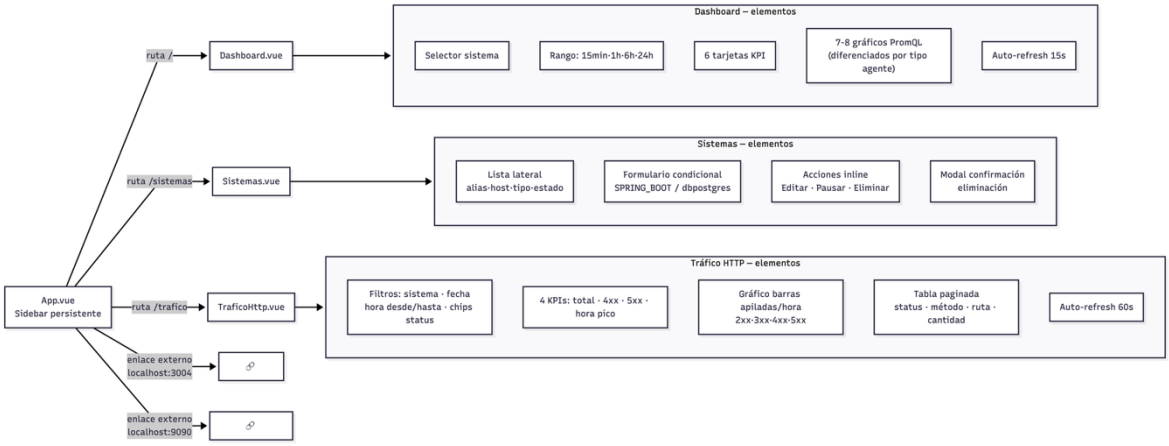


Figura 8. Diagrama de navegación del frontend.

Nota. Elaboración Propia.

3.9.2 Vista: Sistemas (ruta /sistemas)

La Figura 9 presenta el prototipo de la vista Sistemas, interfaz de gestión del registro de sistemas monitoreados.

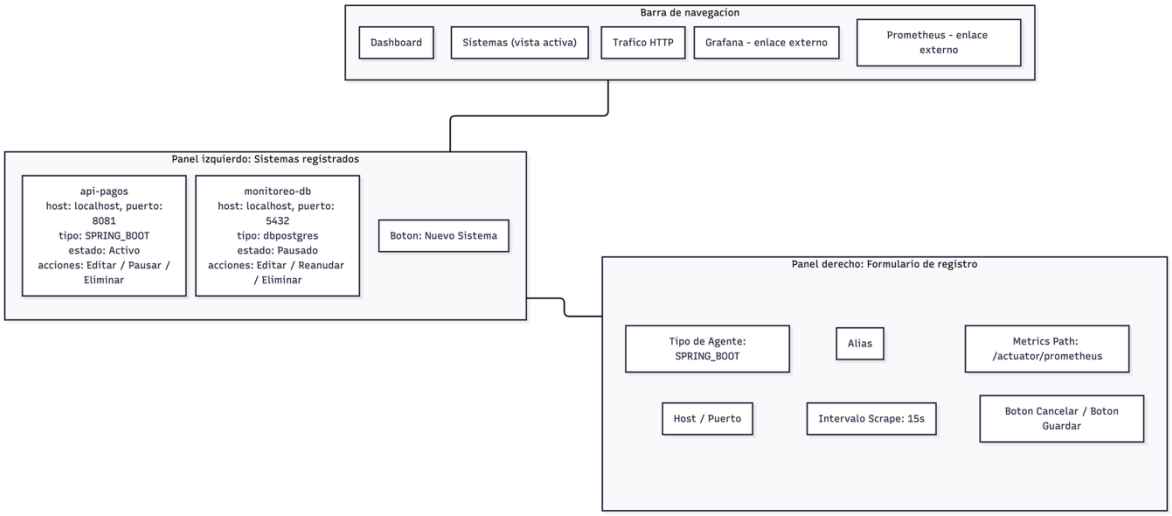


Figura 9. Prototipo de la vista Sistemas (/sistemas).

Nota. Elaboración Propia.

La vista incluye: lista de sistemas con alias, host:puerto, tipo y estado (con acciones Editar/Pausar/Eliminar), formulario de registro/edición con campos condicionales por tipo de agente, modal de confirmación de eliminación y notificaciones de éxito/error.

3.9.3 Vista: Dashboard (ruta /)

La Figura 10 presenta el prototipo de la vista Dashboard, interfaz de visualización de métricas en tiempo real. Contiene selector de sistema, selector de rango temporal, 6 KPIs y 7-8 gráficos de series temporales diferenciados por tipo de agente (Spring Boot o dbpostgres), con actualización automática cada 15 segundos.

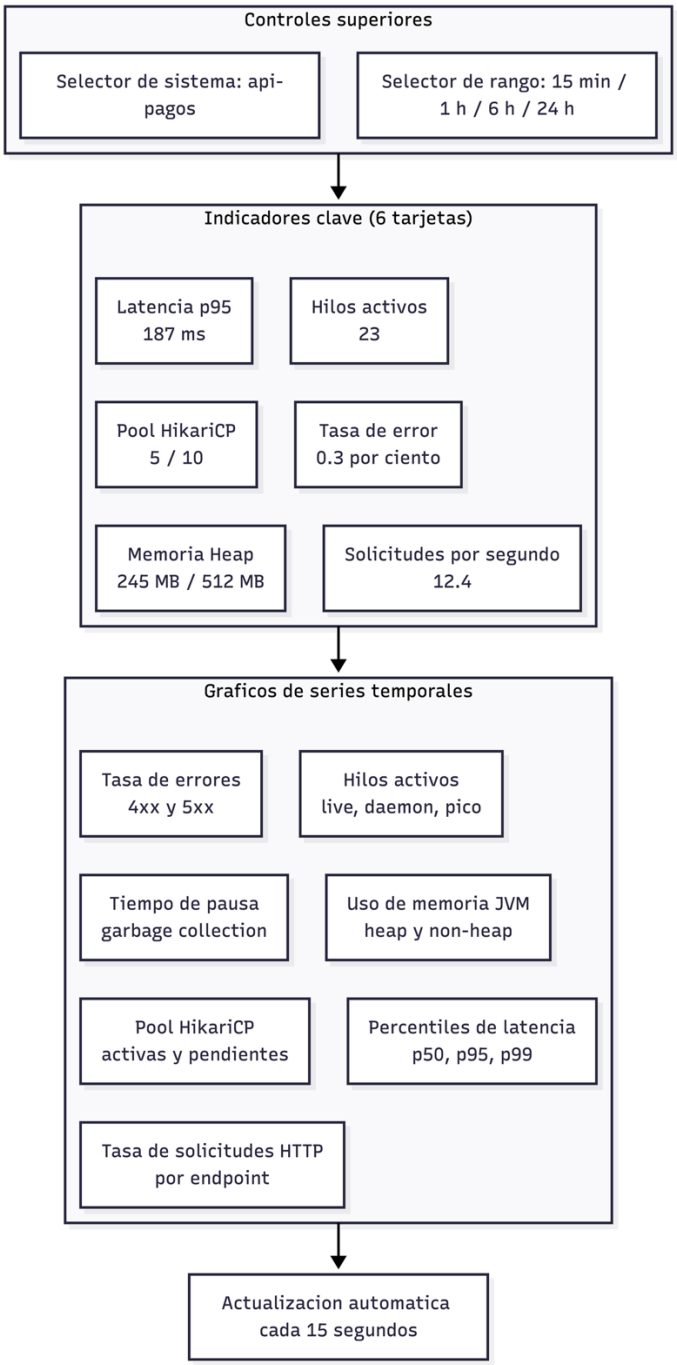


Figura 10. Prototipo de la vista Dashboard (/).

Nota. Elaboración Propia.

3.9.4 Vista: Tráfico HTTP (ruta /trafico)

La Figura 11 presenta el prototipo de la vista Tráfico HTTP, interfaz de análisis del historial de actividad HTTP de los sistemas monitoreados. Esta vista constituye la interfaz del módulo de tracking HTTP y es exclusiva para sistemas de tipo Spring Boot.

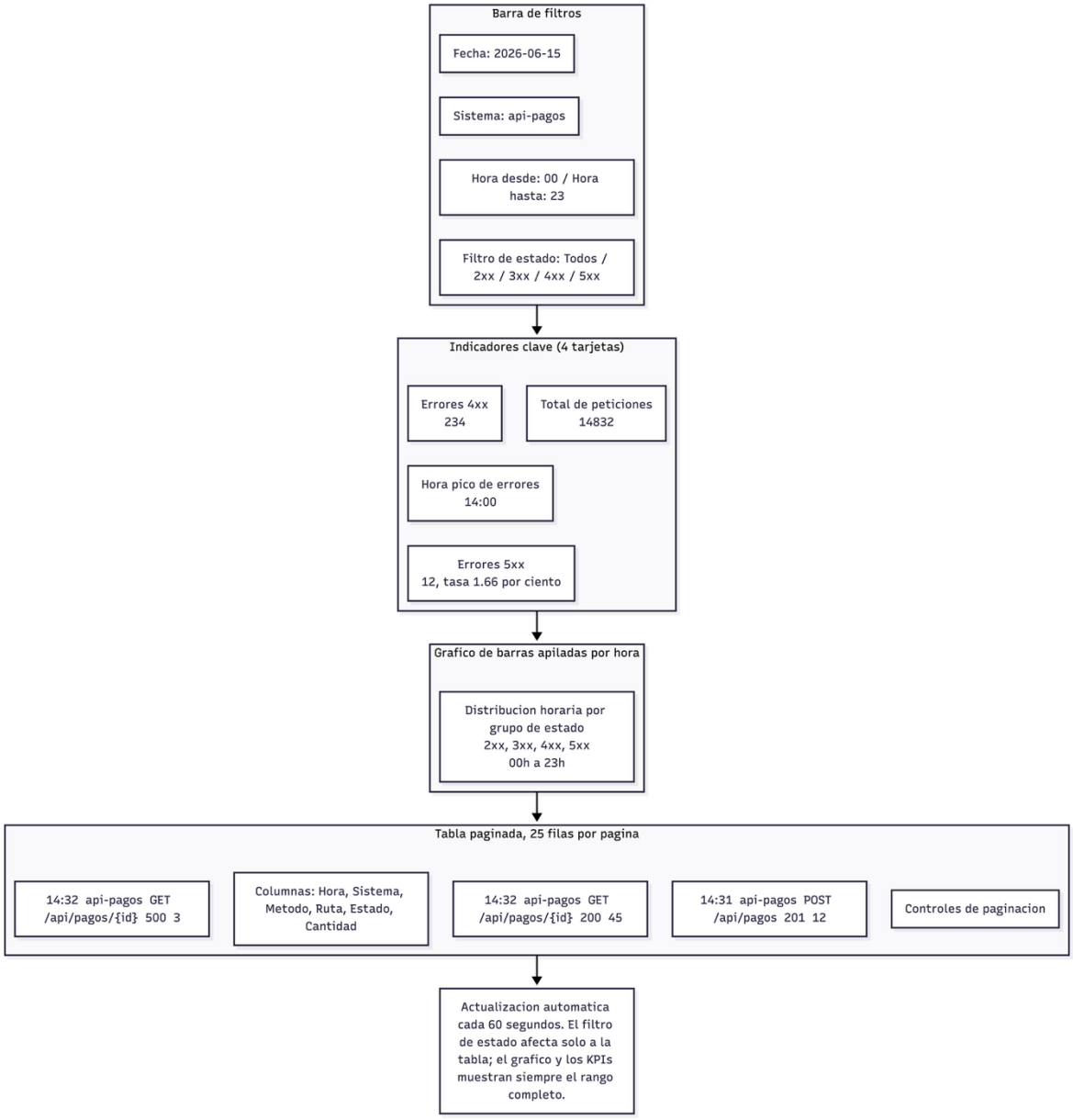


Figura 11. Prototipo de la vista Tráfico HTTP (/trafico).

Nota. Elaboración Propia.

La vista está organizada en cuatro secciones:

Barra de filtros: selector de sistema (con filtrado automático de sistemas dbpostgres, que no generan tráfico HTTP), selector de fecha, selectores de hora desde/hasta (00-23) y chips de colores para el grupo de status: Todos · 2xx (verde) · 3xx (celeste) · 4xx (ámbar) · 5xx (rojo). Los chips filtran únicamente la tabla detallada; el gráfico y los KPIs muestran siempre el panorama completo del rango para no perder el contexto global.

4 KPIs: Total peticiones del período · Errores 4xx · Errores 5xx (con porcentaje de tasa de error) · Hora pico de errores. Permiten al administrador identificar de un vistazo si el período analizado presenta anomalías de disponibilidad.

Gráfico de barras apiladas por hora: construido con ApexCharts, muestra la distribución horaria del tráfico según grupo de status (verde 2xx, celeste 3xx, ámbar 4xx, rojo 5xx). Permite identificar visualmente en qué hora del día se concentraron los errores.

Tabla paginada: muestra Hora · Sistema · Método (badge) · Ruta · Status (badge semafórico por grupo) · Cantidad, ordenada por fecha descendente, con paginación de 25 registros por página. El cambio de cualquier filtro reinicia la paginación a la página cero. La vista se actualiza automáticamente cada 60 segundos, sincronizándose con el ritmo del HttpTrackingService del backend.

3.10 Desarrollo del Sistema

En esta sección se presenta el desarrollo del sistema siguiendo la metodología Scrum. Se describe primero la implementación del marco de trabajo y, posteriormente, el detalle de cada uno de los sprints ejecutados durante el proyecto.

Una vez definidos el análisis y el diseño de la herramienta de monitoreo, se procede a documentar el desarrollo del sistema siguiendo la metodología ágil Scrum. Esta sección describe la implementación iterativa del prototipo, organizada en sprints, con sus respectivos objetivos, tareas, evidencias de implementación y backlog. El desarrollo se gestionó mediante ciclos cortos que permitieron entregar incrementos funcionales evaluables y adaptar el trabajo a los aprendizajes técnicos obtenidos en cada iteración.

3.10.1 Implementación de Scrum

La metodología Scrum proporciona un marco flexible y adaptable para el manejo de proyectos complejos en entornos dinámicos. Al enfocarse en la entrega progresiva de valor, fomentar la autoorganización y promover la inspección y adaptación, Scrum

permite enfrentar la incertidumbre inherente al desarrollo de software (Schwaber & Sutherland, 2020). Considerando las características del presente proyecto, cuyos requerimientos emergen progresivamente a medida que se integra cada componente del stack de monitoreo, se optó por Scrum como metodología de gestión, aprovechando su flexibilidad para adaptarse a cambios, la mejora continua a través de iteraciones cortas y la entrega de incrementos funcionales evaluables al final de cada sprint.

La Figura 12 ilustra el proceso Scrum aplicado en el desarrollo del proyecto, en el que el trabajo se organizó en iteraciones cortas con entregables funcionales al final de cada sprint.

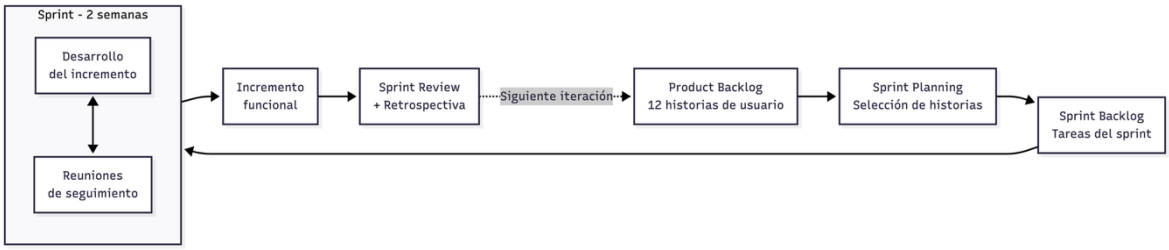


Figura 12. Diagrama del proceso Scrum.

Nota. Elaboración Propia.

3.10.2 Roles de Scrum

Para la ejecución del proyecto se definieron los siguientes roles dentro del marco Scrum:

Product Owner: responsable de definir y priorizar los requerimientos del sistema de monitoreo.

Scrum Master: docente director del trabajo de titulación, encargado de facilitar el proceso y remover impedimentos.

Development Team: Juan David Erazo Unda, responsable del diseño, desarrollo e implementación del prototipo.

3.10.3 Sprint Planning

Dentro de este apartado se define el Product Backlog general del proyecto, a partir del cual se establecen y describen las actividades a realizar en cada sprint. La Tabla 13 presenta la planificación de los sprints con las historias de usuario, su prioridad y esfuerzo estimado.

Tabla 13. Sprint Planning.

Sprint	ID	Historia	Enunciado de la historia	Prioridad	Esfuerzo
1	HU-01	Configurar el stack de monitoreo e instrumentar una aplicación.	Stack Docker Compose operativo y aplicación Spring Boot instrumentada con Micrometer.	Alta	Alto
1	HU-02	Generar dinámicamente el archivo prometheus.yml.	El backend genera el archivo de configuración al registrar un sistema.	Alta	Alto
2	HU-03	Registrar, editar y eliminar sistemas monitoreados.	CRUD de sistemas operativo desde la API REST.	Alta	Alto
2	HU-04	Pausar y reanudar el monitoreo de un sistema.	El atributo monitoreado cambia y se refleja en el prometheus.yml.	Media	Medio
2	HU-05	Diferenciar el formulario según el tipo de agente.	Campos condicionales para Spring Boot y dbpostgres.	Alta	Medio
3	HU-06	Visualizar métricas de aplicaciones	Dashboard con KPIs y gráficos de JVM, HTTP y HikariCP.	Alta	Alto

Sprint	ID	Historia	Enunciado de la historia	Prioridad	Esfuerzo
		Spring Boot en tiempo real.			
3	HU-07	Visualizar métricas de base de datos PostgreSQL.	Dashboard con métricas del endpoint exportador del backend.	Alta	Medio
4	HU-08	Registrar históricamente el tráfico HTTP.	Tabla eventos_http poblada con incrementos de los contadores cada minuto.	Alta	Alto
4	HU-09	Consultar el historial de tráfico HTTP con filtros.	Vista con filtros, KPIs, gráfico apilado y tabla paginada.	Media	Medio
5	HU-10	Implementar la auditoría de operaciones.	Tabla logs poblada con cada operación CRUD.	Media	Medio
6	HU-11	Realizar pruebas y validación experimental.	Pruebas unitarias, de integración y experimentos de desempeño.	Alta	Alto

Nota. La tabla de Sprint Planning detalla las historias de usuario, su prioridad y esfuerzo estimado para cada sprint del proyecto. Autoría propia.

3.10.4 Sprint 1: Configuración del Stack e Instrumentación

Duración: 2 semanas.

Objetivos del Sprint: configurar el stack de monitoreo mediante Docker Compose, instrumentar una aplicación Spring Boot con Micrometer y lograr la generación dinámica del primer archivo prometheus.yml.

Involucrados: Development Team.

Tareas:

- Configurar el archivo docker-compose.yml con los servicios de PostgreSQL, Prometheus y Grafana.

La Figura 13 muestra la configuración del stack de monitoreo definida en el archivo docker-compose.yml, donde se orquestan los servicios de PostgreSQL, Prometheus y Grafana.

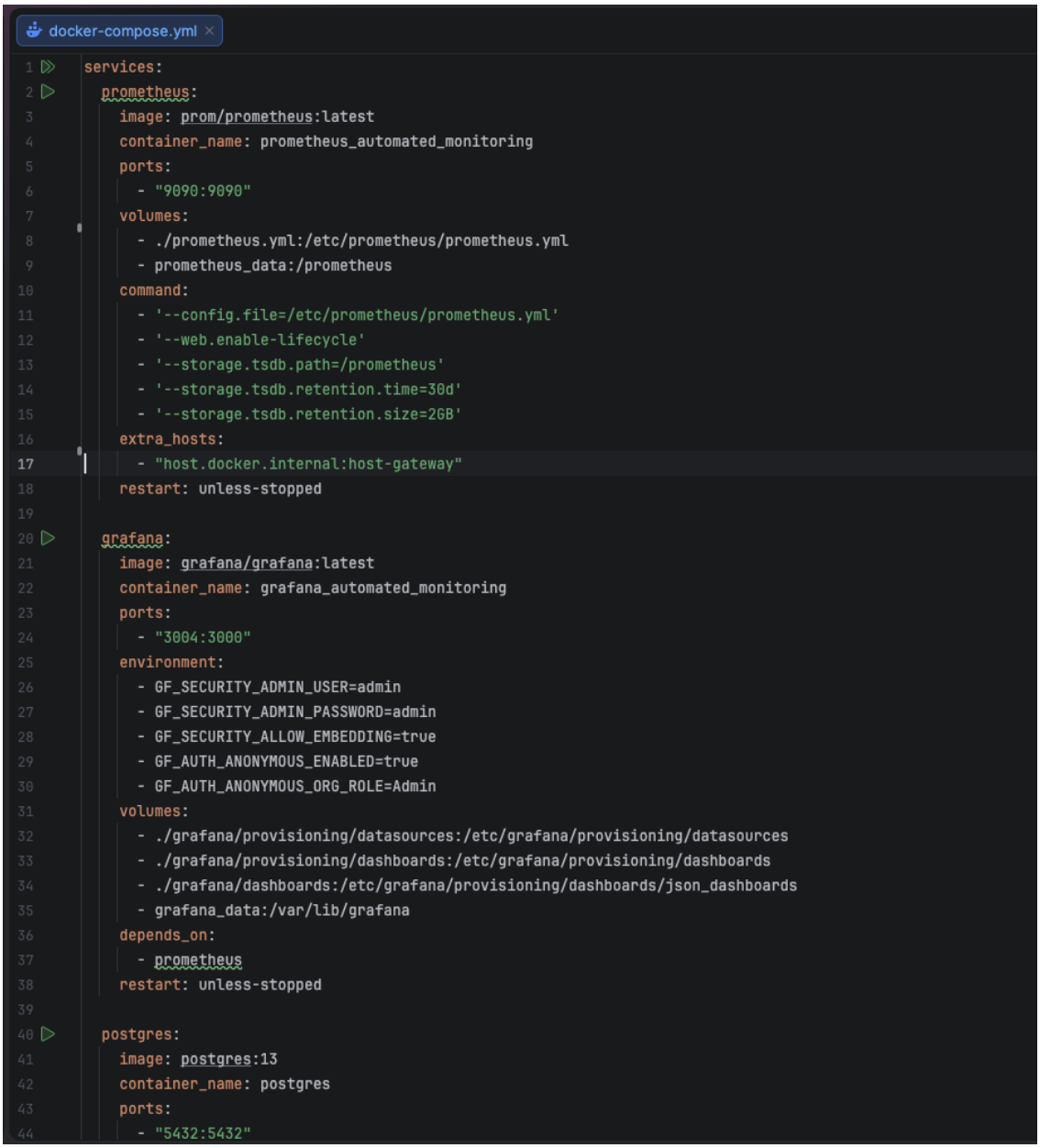


Figura 13. Configuración del stack de monitoreo en Docker Compose.

Nota. Elaboración Propia.

- Instrumentar una aplicación Spring Boot agregando las dependencias de Micrometer y Spring Boot Actuator.

Dependencias de instrumentación en build.gradle:

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '4.0.2'
    id 'io.spring.dependency-management' version '1.1.7'
```

```
}

group = 'com.daviderazo04'
version = '0.0.1-SNAPSHOT'
description = 'Performance monitoring prototype designed to centralize and automate
metrics collection from diverse enterprise systems'

java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(17)
    }
}

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-webmvc'
    implementation 'org.springframework.boot:spring-boot-starter-validation'
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
    implementation 'io.micrometer:micrometer-registry-prometheus'
    runtimeOnly 'org.postgresql:postgresql'
    compileOnly 'org.projectlombok:lombok'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    annotationProcessor 'org.projectlombok:lombok'
```

```
    testImplementation 'org.springframework.boot:spring-boot-starter-data-jpa-test'
    testImplementation 'org.springframework.boot:spring-boot-starter-security-test'
    testImplementation 'org.springframework.boot:spring-boot-starter-webmvc-test'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'

    implementation 'io.jsonwebtoken:jjwt-api:0.11.5'
    runtimeOnly 'io.jsonwebtoken:jjwt-impl:0.11.5'
    runtimeOnly 'io.jsonwebtoken:jjwt-jackson:0.11.5'
}

tasks.named('test') {
    useJUnitPlatform()
}
```

- Verificar la exposición del endpoint de métricas y la generación del primer prometheus.yml.

La Figura 14 presenta las métricas expuestas por la aplicación instrumentada a través del endpoint de Actuator, que confirman la correcta recolección de datos por parte de Micrometer.

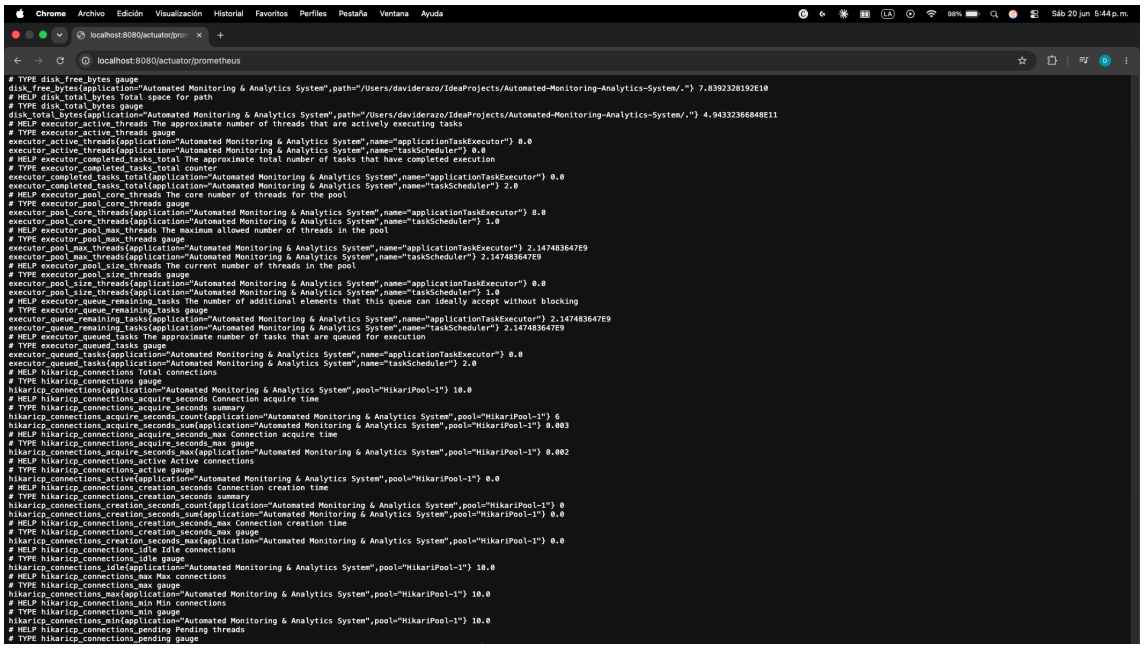


Figura 14. Métricas expuestas por la aplicación instrumentada.

Nota. Elaboración Propia.

Tabla 14. Sprint Backlog 1.

ID	Historia de Usuario	Tarea	Responsable	Esfuerzo
HU-01	Configurar el stack e instrumentar una aplicación.	Configurar docker-compose.yml.	Development Team	Alto
HU-01	Configurar el stack e instrumentar una aplicación.	Instrumentar la app con Micrometer.	Development Team	Alto
HU-02	Generar dinámicamente el prometheus.yml.	Implementar PrometheusOrquestadorService.	Development Team	Alto

Nota. En el Sprint 1 se configuró el stack de monitoreo, se instrumentó la aplicación y se desarrolló el servicio de generación dinámica de la configuración. Autoría propia.

3.10.5 Sprint 2: CRUD de Sistemas Monitoreados

Duración: 2 semanas.

Objetivos del Sprint: implementar las operaciones de registro, edición, eliminación y cambio de estado de los sistemas monitoreados, con regeneración automática del prometheus.yml y formulario diferenciado por tipo de agente.

Involucrados: Development Team.

Tareas:

- Implementar el controlador REST con los endpoints CRUD y el servicio SistemaService.

Lógica de generación del prometheus.yml en PrometheusOrquestadorService:

```
package com.daviderazo04.automated_monitoring_system.service;
```

```
import com.daviderazo04.automated_monitoring_system.model.Sistema;
import com.daviderazo04.automated_monitoring_system.repository.SistemaRepository;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;

@Service
@RequiredArgsConstructor
@Slf4j
public class PrometheusOrquestadorService {

    private final SistemaRepository sistemaRepository;

    @Value("${prometheus.config.path}")
    private String prometheusConfigPath;

    @Value("${prometheus.api.url}")
    private String prometheusApiUrl;

    @Value("${manager.host:host.docker.internal}")
    private String managerHost;

    @Value("${manager.port:8080}")
    private int managerPort;

    @Value("${prometheus.global.scrape-interval:15s}")
    private String globalScrapeInterval;
```

```
   public void generarArchivoConfiguracion() {
       try {
           List<Sistema> sistemasActivos = sistemaRepository.findByMonitoreadoTrue();
           StringBuilder yamlBuilder = new StringBuilder();

           yamlBuilder.append("global:\n");
           yamlBuilder.append("  scrape_interval:
").append(globalScrapeInterval).append("\n\n");
           yamlBuilder.append("scrape_configs:\n");

           // Self-scrape: the orchestrator exposes its own /actuator/prometheus
           yamlBuilder.append("  - job_name: 'automated_monitoring_manager'\n");
           yamlBuilder.append("    metrics_path: '/actuator/prometheus'\n");
           yamlBuilder.append("    static_configs:\n");
           yamlBuilder.append("      - targets:
['").append(managerHost).append(":").append(managerPort).append("']\n\n");

           for (Sistema sistema : sistemasActivos) {
               String jobName = sistema.getAlias();
               String scrapeInterval = sistema.getIntervalo() != null ? sistema.getIntervalo() :
globalScrapeInterval;
               String metricsPath = sistema.getPath() != null ? sistema.getPath() :
"/actuator/prometheus";
               String host = sistema.getHost();
               int puerto = sistema.getPuerto();
               String scheme = "http";

               if ("dbpostgres".equals(sistema.getTipoAgente().getNombre())) {
                   jobName = "postgres_" + sistema.getAlias();
                   metricsPath = "/api/metrics/postgres/" + sistema.getId();
                   host = managerHost;
                   puerto = managerPort;
               } else if (sistema.getPuerto() == 443) {
```

```
                scheme = "https";
            }

            yamlBuilder.append("  - job_name:
").append(yamlString(jobName)).append("\n");
            yamlBuilder.append("    scrape_interval:
").append(scrapeInterval).append("\n");
            yamlBuilder.append("    metrics_path:
").append(yamlString(metricsPath)).append("\n");

            if ("https".equals(scheme)) {
                yamlBuilder.append("    scheme: https\n");
            }

            yamlBuilder.append("    static_configs:\n");
            yamlBuilder.append("      - targets: [").append(yamlString(host + ":" +
puerto)).append("]\n\n");
        }

        Path path = Paths.get(prometheusConfigPath);
        Files.writeString(path, yamlBuilder.toString());
        log.info("Archivo prometheus.yml generado exitosamente con {} targets.",
sistemasActivos.size());

        recargarPrometheusEnCaliente();

    } catch (Exception e) {
        log.error("Error al generar el archivo prometheus.yml", e);
    }
}

private String yamlString(String raw) {
    String escaped = raw.replace("\\", "\\\\").replace("\"", "\\\"");
    return "\"" + escaped + "\"";
```

```
    }

    private void recargarPrometheusEnCaliente() {
        try {
            RestTemplate restTemplate = new RestTemplate();
            String reloadEndpoint = prometheusApiUrl + "/-/reload";
            restTemplate.postForEntity(reloadEndpoint, null, String.class);
            log.info("Prometheus recargado exitosamente en: {}", reloadEndpoint);
        } catch (Exception e) {
            log.warn("No se pudo recargar Prometheus en {}. Error: {}",
                    prometheusApiUrl, e.getMessage());
        }
    }
}
```

- Desarrollar el formulario de registro con campos condicionales según el tipo de agente.

La Figura 15 muestra la vista de gestión de sistemas monitoreados, interfaz desde la cual el usuario registra, edita y elimina los sistemas que serán supervisados.

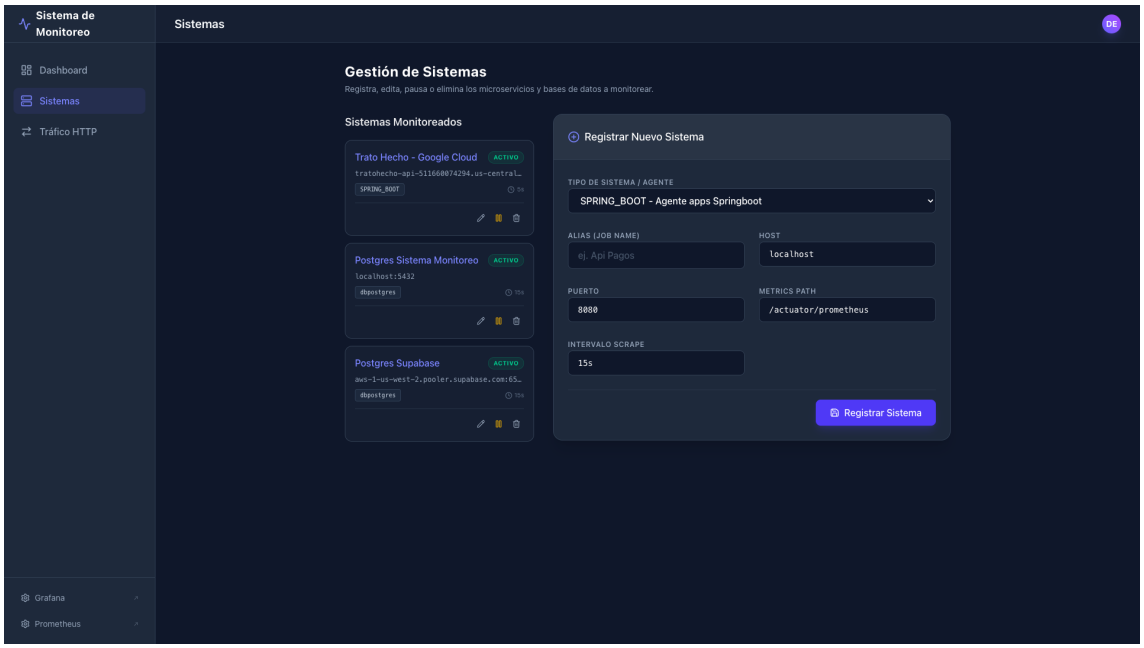


Figura 15. Vista de gestión de sistemas monitoreados.

Nota. Elaboración Propia.

- Verificar la generación del prometheus.yml tras las operaciones CRUD.

Ejemplo de prometheus.yml generado para 1 sistema backend hosteado en Google Cloud y 2 bd Postgres (1 local, 1 supabase):

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'automated_monitoring_manager'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['host.docker.internal:8080']

  - job_name: "Trato Hecho - Google Cloud"
    scrape_interval: 5s
    metrics_path: "/actuator/prometheus"
    scheme: https
    static_configs:
      - targets: ["tratohecho-api-511660074294.us-central1.run.app:443"]

  - job_name: "postgres_Postgres Sistema Monitoreo"
    scrape_interval: 15s
    metrics_path: "/api/metrics/postgres/2"
    static_configs:
      - targets: ["host.docker.internal:8080"]

  - job_name: "postgres_Postgres Supabase"
    scrape_interval: 15s
    metrics_path: "/api/metrics/postgres/3"
    static_configs:
      - targets: ["host.docker.internal:8080"]
```

La Figura 16 presenta el archivo prometheus.yml generado dinámicamente por el sistema tras una operación de registro, lo que evidencia la actualización automática de la configuración sin edición manual.

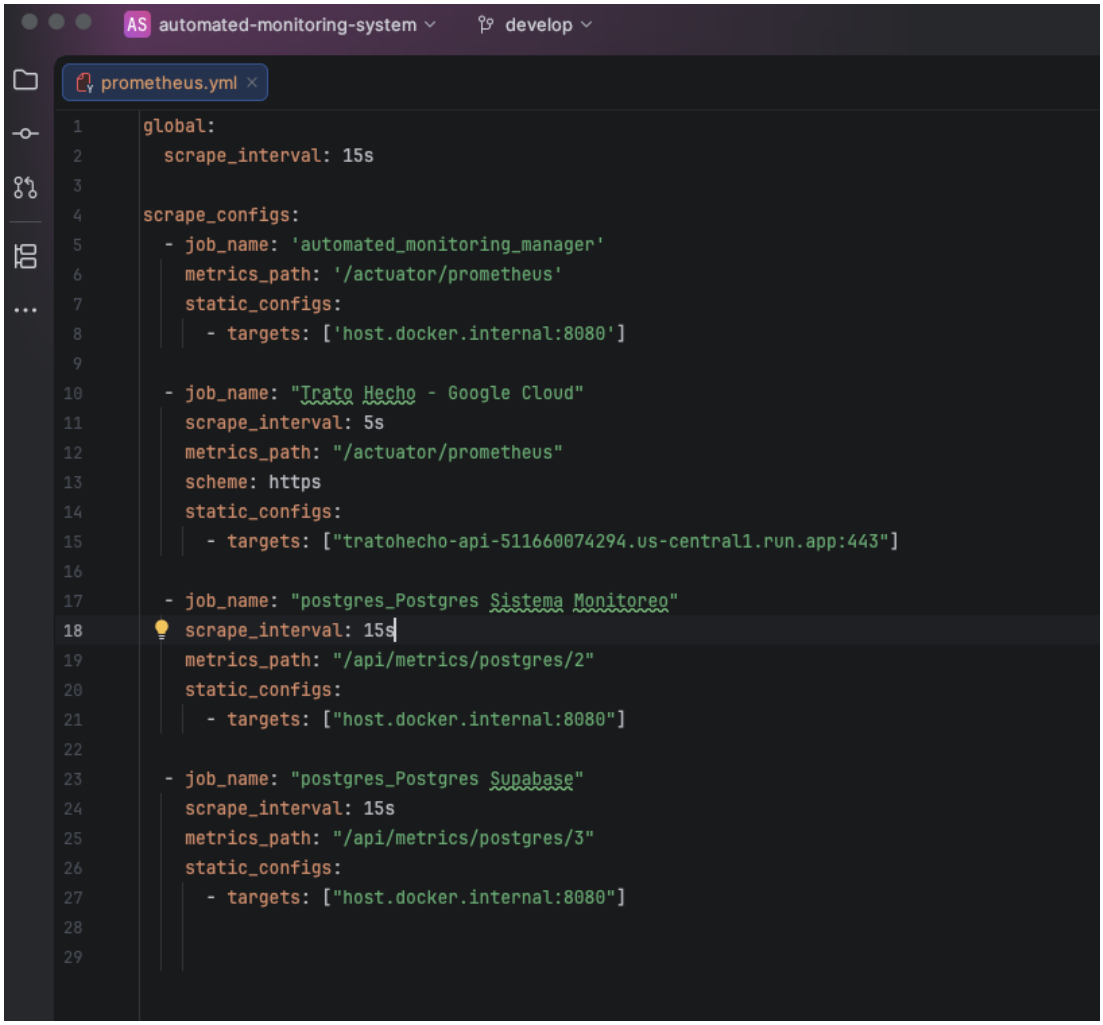


Figura 16. Archivo prometheus.yml generado dinámicamente.

Nota. Elaboración Propia.

Tabla 15. Sprint Backlog 2.

ID	Historia de Usuario	Tarea	Responsable	Esfuerzo
HU-03	Registrar, editar y eliminar sistemas.	Implementar controlador y servicio CRUD.	Development Team	Alto

ID	Historia de Usuario	Tarea	Responsable	Esfuerzo
HU-04	Pausar y reanudar el monitoreo.	Implementar endpoint de cambio de estado de monitoreo.	Development Team	Medio
HU-05	Diferenciar el formulario por tipo de agente.	Desarrollar formulario condicional.	Development Team	Medio

Nota. En el Sprint 2 se implementó el CRUD completo de sistemas, el cambio de estado y el formulario diferenciado por tipo de agente. Autoría propia.

3.10.6 Sprint 3: Dashboard de Visualización en Tiempo Real

Duración: 2 semanas.

Objetivos del Sprint: desarrollar el dashboard de visualización de métricas en tiempo real, diferenciando los paneles según el tipo de agente del sistema seleccionado.

Involucrados: Development Team.

Tareas:

- Desarrollar la vista Dashboard con selector de sistema y de rango temporal.
- Implementar las consultas PromQL contra la API de Prometheus para los KPIs y gráficos.

La Figura 17 muestra el dashboard de métricas en tiempo real de una aplicación monitoreada, donde se visualizan los principales indicadores de rendimiento recolectados.



Figura 17. Dashboard de métricas en tiempo real de una aplicación Spring Boot.

Nota. Elaboración Propia.

- Diferenciar los paneles para sistemas de tipo dbpostgres.

La Figura 18 presenta el dashboard de métricas correspondiente a la base de datos PostgreSQL, con indicadores como las conexiones activas y el estado del pool de conexiones.



Figura 18. Dashboard de métricas de base de datos PostgreSQL.

Nota. Elaboración Propia.

Tabla 16. Sprint Backlog 3.

ID	Historia de Usuario	Tarea	Responsable	Esfuerzo
HU-06	Visualizar métricas de aplicaciones Spring Boot.	Desarrollar dashboard con consultas PromQL.	Development Team	Alto
HU-07	Visualizar métricas de PostgreSQL.	Diferenciar paneles por tipo de agente.	Development Team	Medio

Nota. En el Sprint 3 se desarrolló el dashboard de visualización en tiempo real con diferenciación de métricas por tipo de agente. Autoría propia.

3.10.7 Sprint 4: Módulo de Tracking de Tráfico HTTP

Duración: 2 semanas.

Objetivos del Sprint: implementar el módulo de registro histórico de tráfico HTTP mediante la detección de incrementos de los contadores, y su vista de consulta con filtros, KPIs y gráfico.

Involucrados: Development Team.

Tareas:

- Implementar el HttpTrackingService con el ciclo de recolección de diferencia programado cada 60 segundos.

Lógica de detección de incrementos en HttpTrackingService:

```
public void recolectarEventosHttp() {
    List<Sistema> objetivos = sistemaRepository.findByMonitoreadoTrue().stream()
            .filter(s -> s.getTipoAgente() != null
                    && !TIPO_POSTGRES.equals(s.getTipoAgente().getNombre()))
            .toList();

    Set<Long> idsActivos = new HashSet<>();
    objetivos.forEach(s -> idsActivos.add(s.getId()));
    baselines.keySet().retainAll(idsActivos);
```

```
  for (Sistema sistema : objetivos) {
      try {
          recolectarDeSistema(sistema);
      } catch (Exception e) {
          log.warn("Tracking HTTP: no se pudo leer métricas de '{}': {}",
                  sistema.getAlias(), e.getMessage());
      }
  }
}
```

- Desarrollar la vista Tráfico HTTP con filtros, KPIs, gráfico de barras apiladas y tabla paginada.

La Figura 19 muestra la vista de historial de tráfico HTTP, que permite consultar la actividad registrada de los sistemas monitoreados a lo largo del tiempo.

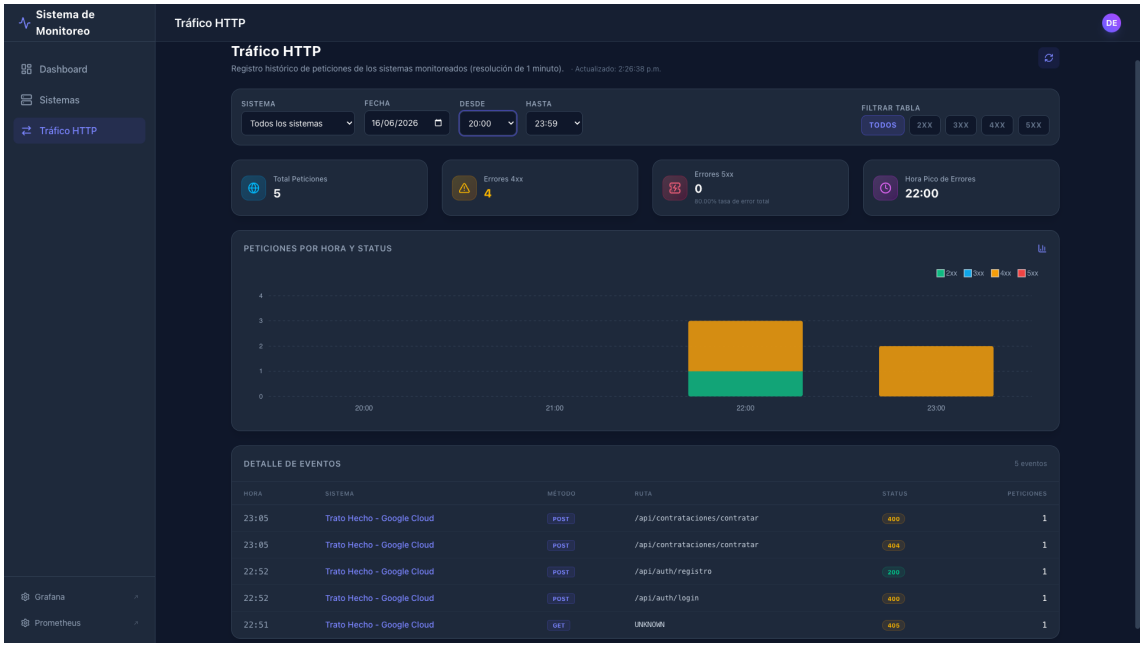


Figura 19. Vista de historial de tráfico HTTP.

Nota. Elaboración Propia.

Tabla 17. Sprint Backlog 4.

ID	Historia de Usuario	Tarea	Responsable	Esfuerzo
HU-08	Registrar históricamente el tráfico HTTP.	Implementar HttpTrackingService (incrementos).	Development Team	Alto
HU-09	Consultar el historial con filtros.	Desarrollar vista TraficoHttp.vue.	Development Team	Medio

Nota. En el Sprint 4 se implementó el módulo de tracking de tráfico HTTP por detección de incrementos y su vista de consulta. Autoría propia.

3.10.8 Sprint 5: Auditoría

Duración: 1 semanas.

Objetivos del Sprint: implementar el registro de auditoría de las operaciones CRUD

Involucrados: Development Team.

Tareas:

- Implementar el registro automático de logs de auditoría en cada operación de escritura sobre sistemas.

La Figura 20 presenta la infraestructura de auditoría implementada, encargada de registrar y preservar los eventos relevantes de la operación del sistema.

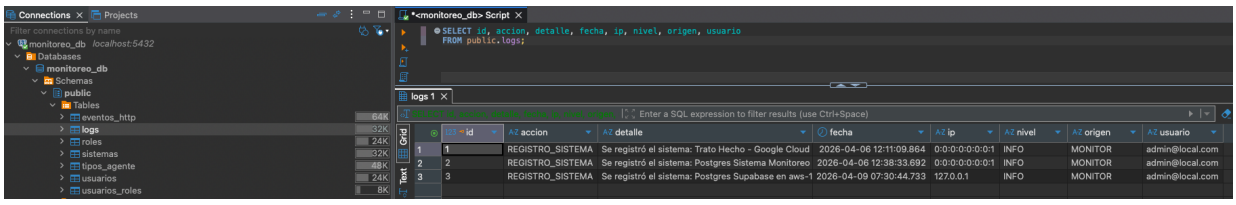


Figura 20. Infraestructura de auditoría.

Nota. Elaboración Propia.

Tabla 18. Sprint Backlog 5.

ID	Historia de Usuario	Tarea	Responsable	Esfuerzo
HU-10	Implementar la auditoría de operaciones.	Registrar logs en cada operación CRUD.	Development Team	Medio

Nota. En el Sprint 5 se implementó la auditoría de las operaciones del sistema. Autoría propia.

3.10.9 Sprint 6: Pruebas y Validación Experimental

Duración: 2 semanas.

Objetivos del Sprint: ejecutar las pruebas unitarias y de integración del sistema, y realizar la validación experimental del desempeño del prototipo.

Involucrados: Development Team.

Las tareas y resultados de este sprint se detallan en las secciones 3.11 (Plan de Pruebas) y 3.12 (Validación Experimental).

Tabla 19. Sprint Backlog 6.

ID	Historia de Usuario	Tarea	Responsable	Esfuerzo
HU-11	Realizar pruebas y validación.	Implementar pruebas unitarias y de integración.	Development Team	Alto
HU-11	Realizar pruebas y validación.	Ejecutar experimentos de desempeño.	Development Team	Alto

Nota. En el Sprint 6 se realizaron las pruebas del sistema y la validación experimental del prototipo. Autoría propia.

3.11 Plan de Pruebas

En esta sección se presentan las pruebas realizadas para validar el correcto funcionamiento del sistema. Se incluyen pruebas unitarias y de integración, cuyos resultados se documentan a continuación como evidencia de su ejecución.

El plan de pruebas contempla pruebas unitarias, que verifican el comportamiento aislado de los componentes críticos del backend, y pruebas de integración, que validan el flujo

completo entre los módulos. Dado que el presente trabajo constituye un prototipo y considerando las restricciones de tiempo del proyecto, el esfuerzo de pruebas se priorizó sobre los componentes de mayor complejidad algorítmica, los cuales son: el servicio de generación dinámica del prometheus.yml y el cálculo del incremento de peticiones HTTP del módulo de tracking. Las pruebas de integración validan el ciclo de gestión de sistemas (registro, pausa y eliminación) junto con el registro y la consulta del historial de tráfico HTTP. Se emplearon los frameworks JUnit 5 y Mockito.

3.11.1 Pruebas Unitarias

Tabla 20. Casos de prueba unitaria.

ID	Caso de prueba	Resultado esperado	Estado
PU-01	Generar YAML con un sistema Spring Boot activo.	El YAML contiene job_name, target y metrics_path.	**Aprobada**
PU-02	Generar YAML con un sistema dbpostgres.	El job apunta al endpoint exportador interno (/api/metrics/postgres/{id}).	**Aprobada**
PU-03	Excluir sistemas pausados (monitoreado=false).	El YAML no contiene los sistemas pausados.	**Aprobada**
PU-04	Generar YAML con lista vacía de sistemas.	Se genera sin error, solo con el job del orquestador.	**Aprobada**
PU-05	Calcular incremento positivo entre dos lecturas.	Se genera EventoHttp con cantidad correcta.	**Aprobada**
PU-06	Contador sin cambios entre lecturas.	No se genera ningún EventoHttp.	**Aprobada**
PU-07	Reinicio de la app (actual < previo).	El incremento se cuenta desde cero, sin negativos.	**Aprobada**
PU-08	Registrar sistema con alias duplicado.	Se lanza excepción y no se persiste.	**Aprobada**

Nota. La columna Estado se completa con el resultado (Aprobada / Fallida). Autoría propia.

3.11.1.1 Evidencia de ejecución de las pruebas unitarias

A continuación se presenta, para cada prueba unitaria, el código fuente del caso y la captura de su ejecución exitosa. Las pruebas del servicio de generación del prometheus.yml emplean Mockito para simular el repositorio de sistemas (SistemaRepository) y aislar la lógica de generación; las pruebas del cálculo de incrementos se ejecutan sobre el método calcularIncremento, sin dependencias externas.

PU-01: Generación de YAML con un sistema Spring Boot activo

```
@Test
@DisplayName("PU-01: genera YAML con job_name, target y metrics_path para SPRING_BOOT")
void pu01_sistemaSpringBoot() throws Exception {
    when(sistemaRepository.findByMonitoreadoTrue())
            .thenReturn(List.of(springBoot("api-pagos", "10.0.0.5", 8081)));

    String yaml = generarYLeer();

    assertThat(yaml).contains("job_name: \"api-pagos\"");
    assertThat(yaml).contains("10.0.0.5:8081");
    assertThat(yaml).contains("metrics_path: \"/actuator/prometheus\"");
}
```

La Figura 21 muestra el resultado de la prueba PU-01, que verifica que el job generado para una aplicación Spring Boot apunta correctamente al endpoint esperado dentro del archivo de configuración.

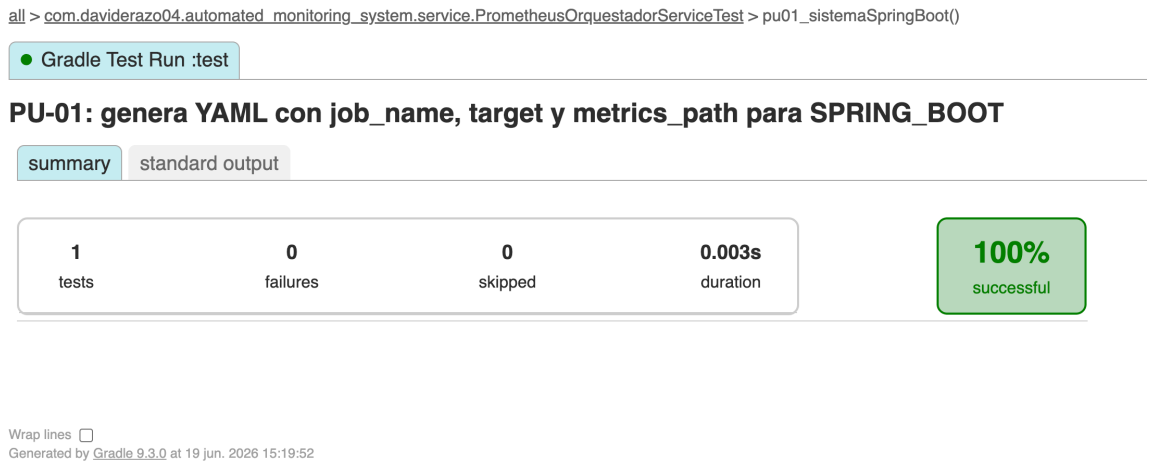


Figura 21. Resultado de la ejecución de la prueba PU-01.

Nota. Elaboración Propia.

PU-02: El job de un dbpostgres apunta al endpoint exportador interno

```java
@Test
@DisplayName("PU-02: el job de un dbpostgres apunta al endpoint exportador interno")
void pu02_sistemaPostgres() throws Exception {
    when(sistemaRepository.findByMonitoreadoTrue())
            .thenReturn(List.of(postgres(7L, "db-ventas")));

    String yaml = generarYLeer();

    assertThat(yaml).contains("job_name: \"postgres_db-ventas\"");
    assertThat(yaml).contains("metrics_path: \"/api/metrics/postgres/7\"");
    assertThat(yaml).contains("host.docker.internal:8080");
    // No expone el host:puerto real de la base de datos
    assertThat(yaml).doesNotContain("db.interno:5432");
}
```

La Figura 22 presenta el resultado de la prueba PU-02, que valida que el job de un sistema de tipo dbpostgres apunte al endpoint exportador interno correspondiente.



Figura 22. Resultado de la ejecución de la prueba PU-02.

Nota. Elaboración Propia.

PU-03: Solo se incluyen los sistemas activos

```java
@@Test
@DisplayName("PU-03: solo se incluyen los sistemas activos (monitoreado=true)")
void pu03_excluyePausados() throws Exception {
    when(sistemaRepository.findByMonitoreadoTrue())
            .thenReturn(List.of(springBoot("activo-app", "10.0.0.9", 9000)));

    String yaml = generarYLeer();

    assertThat(yaml).contains("activo-app");
    assertThat(yaml).doesNotContain("pausado-app");
}
```

La Figura 23 muestra el resultado de la prueba PU-03, que comprueba que en la configuración generada solo se incluyen los sistemas que se encuentran activos.



Figura 23. Resultado de la ejecución de la prueba PU-03.

Nota. Elaboración Propia.

PU-04: Generación con lista vacía de sistemas

```
@@Test
@DisplayName("PU-04: con lista vacía genera el archivo solo con el job del orquestador")
void pu04_listaVacia() throws Exception {
    when(sistemaRepository.findByMonitoreadoTrue()).thenReturn(List.of());

    String yaml = generarYLeer();

    assertThat(yaml).contains("scrape_configs:");
    assertThat(yaml).contains("automated_monitoring_manager");
    // Solo existe un job (el del propio orquestador)
    assertThat(yaml.split("job_name:")).hasSize(2);
}
```

La Figura 24 presenta el resultado de la prueba PU-04, que verifica el comportamiento de la generación de configuración cuando la lista de sistemas se encuentra vacía.



Figura 24. Resultado de la ejecución de la prueba PU-04.

Nota. Elaboración Propia.

PU-05: Incremento positivo entre dos lecturas

```
@Test
@DisplayName("PU-05: incremento positivo entre dos lecturas consecutivas")
void pu05_incrementoPositivo() {
    // Contador pasó de 100 a 145 -> 45 peticiones nuevas en el intervalo
    assertThat(HttpTrackingService.calcularIncremento(145, 100.0)).isEqualTo(45L);
}
```

La Figura 25 muestra el resultado de la prueba PU-05, que valida el cálculo correcto de un incremento positivo entre dos lecturas consecutivas de un contador.

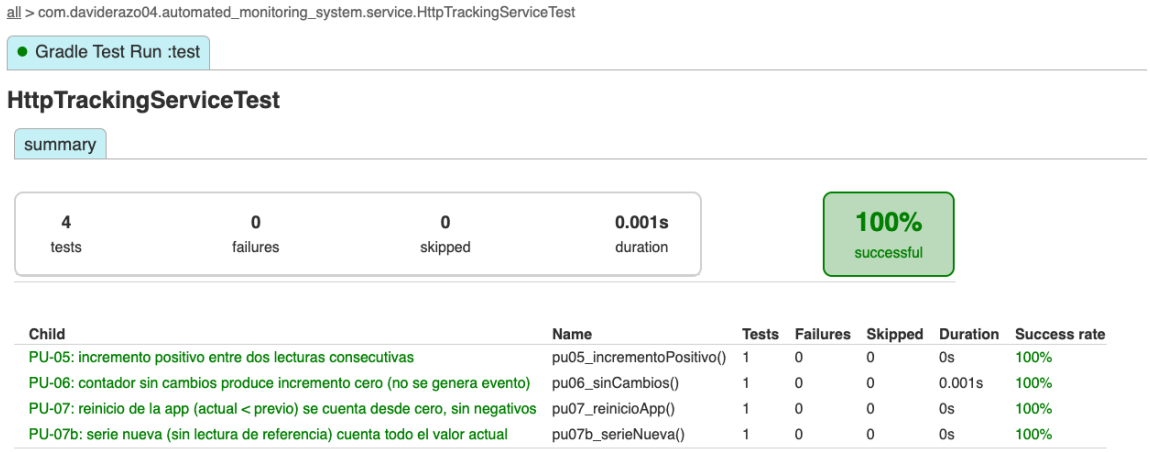


Figura 25. Resultado de la ejecución de la prueba PU-05.

Nota. Elaboración Propia.

PU-06: Contador sin cambios entre lecturas

```
@Test
@DisplayName("PU-06: contador sin cambios produce incremento cero (no se genera evento)")
void pu06_sinCambios() {
    assertThat(HttpTrackingService.calcularIncremento(100, 100.0)).isZero();
}
```

La Figura 26 presenta el resultado de la prueba PU-06, que comprueba que un contador sin cambios entre lecturas produce un incremento igual a cero.

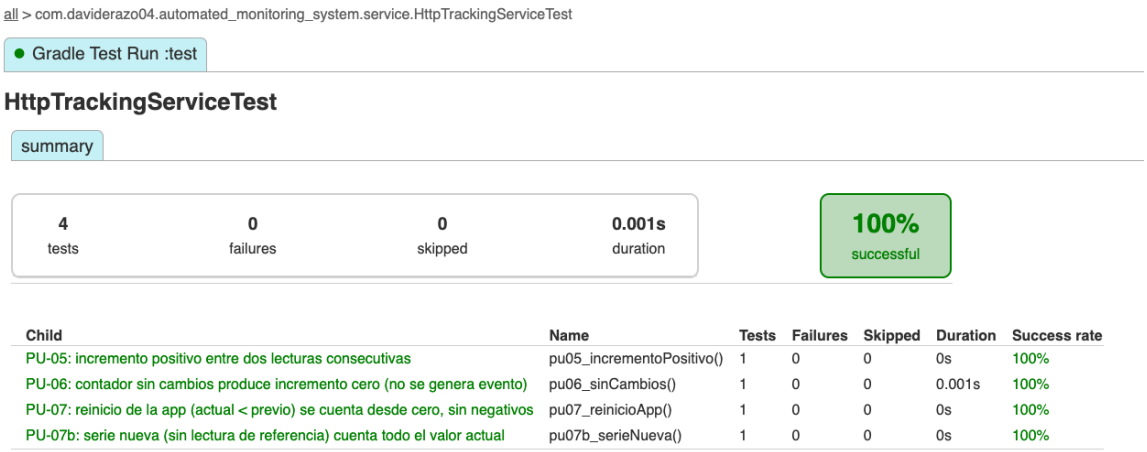


Figura 26. Resultado de la ejecución de la prueba PU-06.

Nota. Elaboración Propia.

PU-07: Reinicio de la aplicación (actual < previo)

```
@@Test
@DisplayName("PU-07: reinicio de la app (actual < previo) se cuenta desde cero, sin negativos")
void pu07_reinicioApp() {
    assertThat(HttpTrackingService.calcularIncremento(30, 100.0)).isEqualTo(30L);
    assertThat(HttpTrackingService.calcularIncremento(30, 100.0)).isNotNegative();
}
```

La Figura 27 muestra el resultado de la prueba PU-07, que valida el manejo del reinicio de la aplicación, cuando el valor actual del contador es menor que el previo.

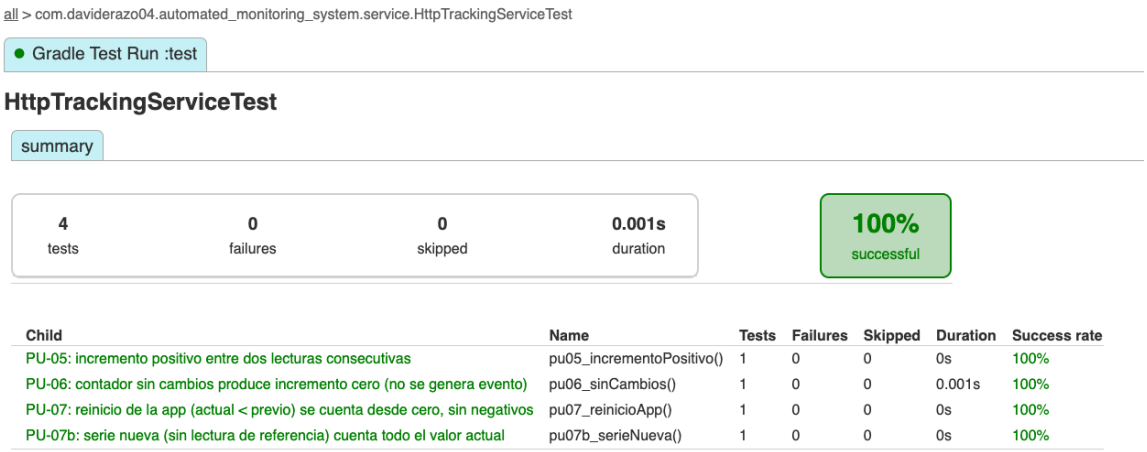


Figura 27. Resultado de la ejecución de la prueba PU-07.

Nota. Elaboración Propia.

PU-08: Registro con alias duplicado

```
@Test
@DisplayName("PU-08: registrar un sistema con alias duplicado lanza excepción y no se persiste")
void pu08_aliasDuplicado() {
```

```
    sistemaService.registrarSistema(dtoSpringBoot("dup-alias", "10.2.0.1", 8100), "admin@local.com",
"127.0.0.1");

    assertThatThrownBy(() ->
            sistemaService.registrarSistema(dtoSpringBoot("dup-alias", "10.2.0.2", 8101),
"admin@local.com", "127.0.0.1"))
            .isInstanceOf(DataAccessException.class);

    long repetidos = sistemaRepository.findAll().stream()
            .filter(s -> "dup-alias".equals(s.getAlias())).count();
    assertThat(repetidos).isEqualTo(1L);
}
```

La Figura 28 presenta el resultado de la prueba PU-08, que verifica que el registro de un sistema con un alias duplicado genera el error controlado esperado.

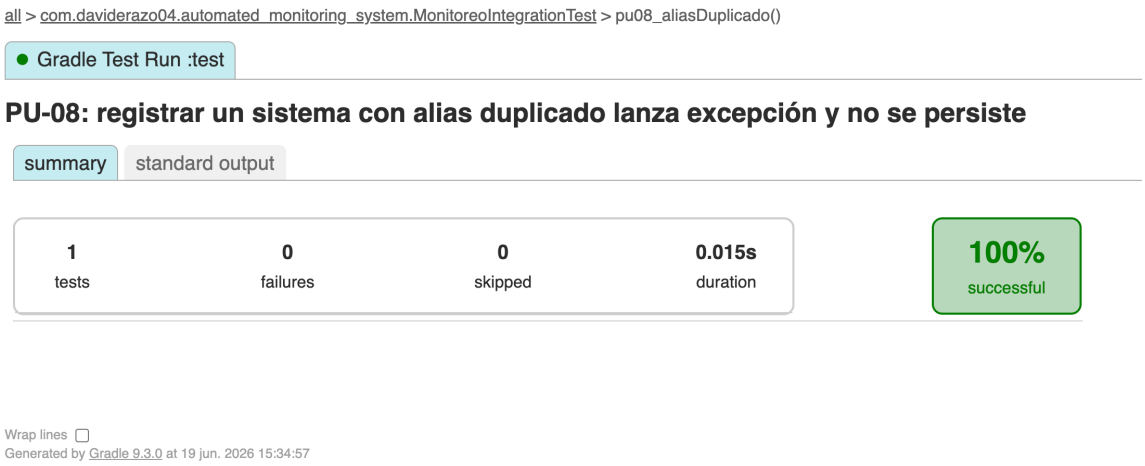


Figura 28. Resultado de la ejecución de la prueba PU-08.

Nota. Elaboración Propia.

3.11.2 Pruebas de Integración

Tabla 21. Casos de prueba de integración.

ID	Caso de prueba	Resultado esperado	Estado
PI-01	Registrar sistema vía API y verificar prometheus.yml.	El archivo se actualiza con el nuevo target.	**Aprobada**
PI-02	Verificar que Prometheus scrapea el nuevo target.	El target aparece como UP en Prometheus.	**Aprobada (manual)**

PI-03	Eliminar sistema y verificar prometheus.yml.	El target desaparece del archivo.	**Aprobada**
PI-04	Pausar sistema (cambio de estado) y verificar prometheus.yml.	El target se excluye sin eliminar el registro.	**Aprobada**
PI-05	Generar tráfico y verificar tabla eventos_http.	Se persisten filas con el conteo correcto.	**Aprobada**
PI-06	Consultar el resumen de tráfico con datos reales.	El resumen retorna KPIs y buckets coherentes.	**Aprobada**
PI-07	Eliminar sistema con histórico de tráfico.	Los eventos_http se conservan.	**Aprobada**

Nota. La columna Estado se completa tras la ejecución. Autoría propia.

3.11.2.1 Evidencia de ejecución de las pruebas de integración

Las pruebas de integración se ejecutan con el contexto completo de Spring Boot sobre una base de datos H2 en memoria, validando la interacción real entre los servicios, los repositorios y la generación del archivo prometheus.yml. A continuación se presenta el código de cada caso y la captura de su resultado.

PI-01: Registrar sistema y verificar el prometheus.yml

```
@Test
@DisplayName("PI-01: registrar sistema vía servicio actualiza el prometheus.yml con el nuevo target")
void pi01_registroActualizaYaml() throws Exception {
    sistemaService.registrarSistema(dtoSpringBoot("ms-pi01", "10.1.0.1", 8090), "admin@local.com", "127.0.0.1");

    assertThat(leerYaml())
            .contains("job_name: \"ms-pi01\"")
            .contains("10.1.0.1:8090");
}
```

La Figura 29 muestra el resultado de la prueba PI-01, que valida que el registro de un sistema a través del servicio actualiza correctamente el archivo prometheus.yml.

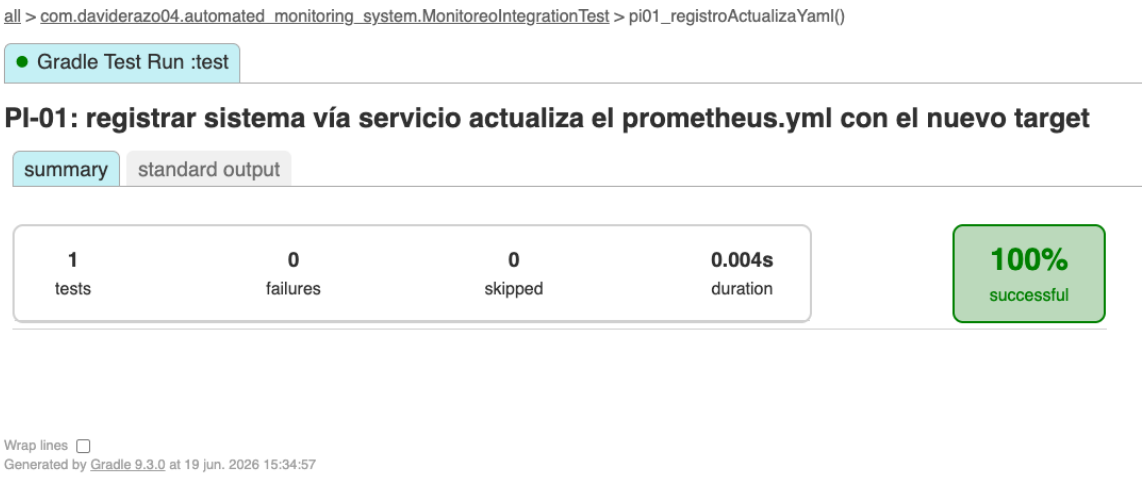


Figura 29. Resultado de la ejecución de la prueba PI-01.

Nota. Elaboración Propia.

PI-02: Verificar que Prometheus scrapea el nuevo target

Esta prueba se verifica de forma manual con el stack en ejecución: tras registrar un sistema, se accede a la interfaz de Prometheus (http://localhost:9090/targets) y se comprueba que el nuevo target aparece en estado UP. La captura correspondiente se presenta a continuación.

La Figura 30 presenta el resultado de la prueba PI-02, que comprueba de forma manual que Prometheus reconoce el nuevo target una vez registrado el sistema.

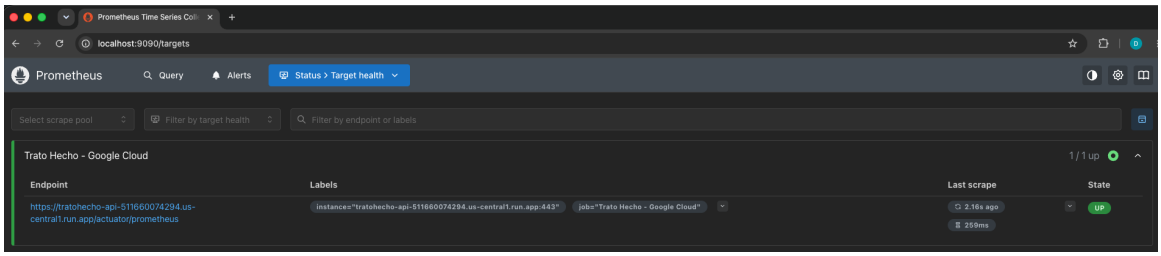


Figura 30. Resultado de la ejecución de la prueba PI-02.

Nota. Elaboración Propia.

PI-03: Eliminar sistema y verificar el prometheus.yml

```
@Test
@DisplayName("PI-03: eliminar un sistema lo quita del prometheus.yml")
void pi03_eliminarQuitaDelYaml() throws Exception {
    SistemaResponseDTO creado = sistemaService.registrarSistema(
            dtoSpringBoot("ms-pi03", "10.1.0.3", 8091), "admin@local.com", "127.0.0.1");
    assertThat(leerYaml()).contains("ms-pi03");

    sistemaService.eliminarSistema(creado.getId(), "admin@local.com", "127.0.0.1");
```

```
    assertThat(leerYaml()).doesNotContain("ms-pi03");
}
```

La Figura 31 muestra el resultado de la prueba PI-03, que verifica que la eliminación de un sistema retira su target del archivo de configuración.

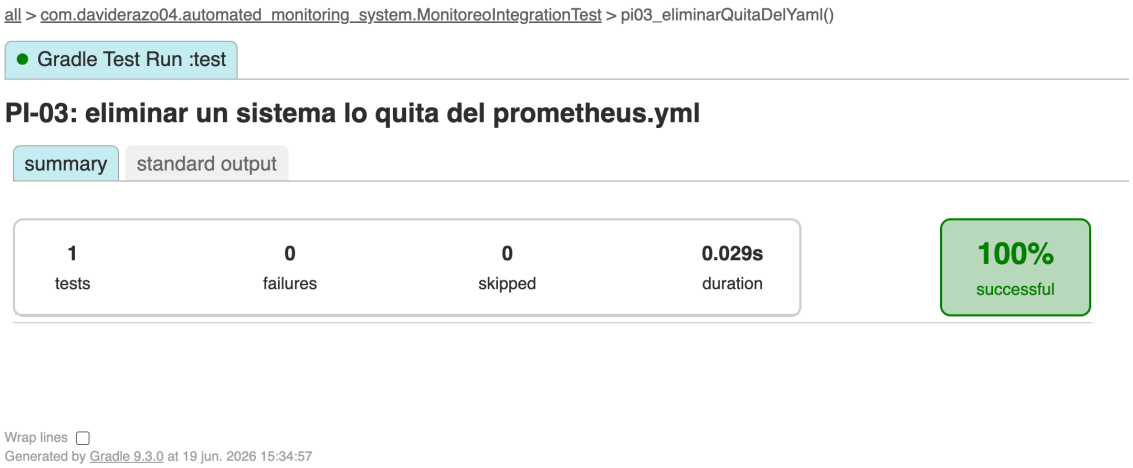


Figura 31. Resultado de la ejecución de la prueba PI-03.

Nota. Elaboración Propia.

PI-04: Pausar sistema y verificar el prometheus.yml

```
@Test
@DisplayName("PI-04: pausar un sistema lo excluye del YAML sin eliminar el registro")
void pi04_pausarExcluyeSinEliminar() throws Exception {
    SistemaResponseDTO creado = sistemaService.registrarSistema(
            dtoSpringBoot("ms-pi04", "10.1.0.4", 8092), "admin@local.com", "127.0.0.1");
    assertThat(leerYaml()).contains("ms-pi04");

    sistemaService.alternarMonitoreo(creado.getId(), "admin@local.com", "127.0.0.1");

    assertThat(leerYaml()).doesNotContain("ms-pi04");
    // El registro sigue existiendo, solo que pausado
    assertThat(sistemaRepository.findById(creado.getId())).isPresent();
    assertThat(sistemaRepository.findById(creado.getId()).get().isMonitoreado()).isFalse();
}
```

La Figura 32 presenta el resultado de la prueba PI-04, que valida que al pausar un sistema su target se excluye del archivo YAML sin eliminar el registro.

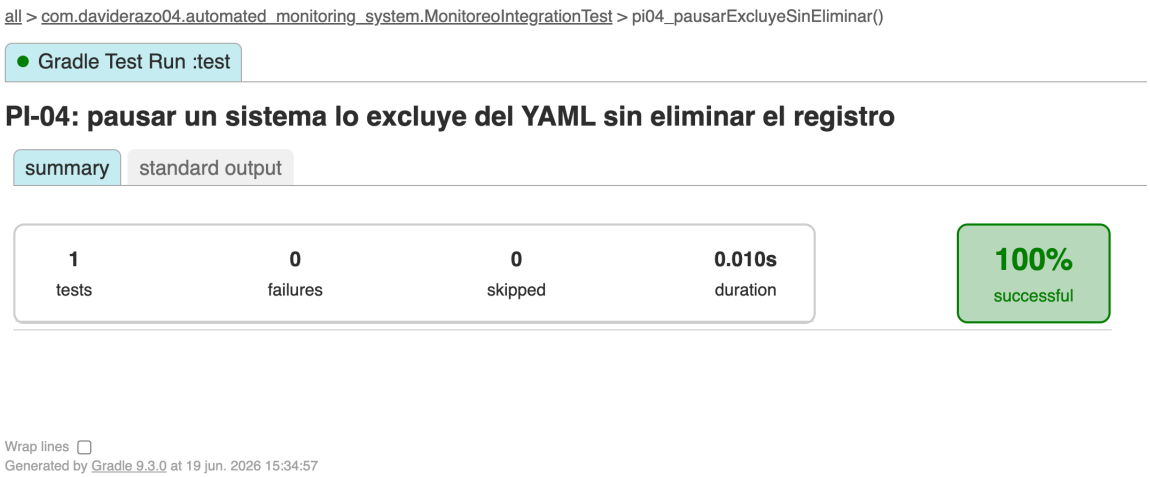


Figura 32. Resultado de la ejecución de la prueba PI-04.

Nota. Elaboración Propia.

PI-05 / PI-06: Persistencia de tráfico y resumen de KPIs

```
@@Test
@DisplayName("PI-05/PI-06: el tráfico registrado se persiste y el resumen entrega KPIs coherentes")
void pi05_06_traficoYResumen() {
    long sid = 9001L;
    LocalDateTime t = LocalDate.now().atTime(10, 0);
    eventoHttpRepository.save(evento(sid, t, 200, "GET", "/api/ok", 40));
    eventoHttpRepository.save(evento(sid, t, 404, "GET", "/api/missing", 5));
    eventoHttpRepository.save(evento(sid, t, 500, "POST", "/api/fail", 3));

    // PI-05: las filas quedan persistidas
    var pagina = httpTrackingService.buscarEventos(sid, "todos", LocalDate.now(), 0, 23, 0, 25);
    assertThat((long) (Long) pagina.get("totalElements")).isEqualTo(3L);

    // PI-06: el resumen entrega totales y errores coherentes
    EventoHttpResumenDTO resumen = httpTrackingService.resumen(sid, LocalDate.now(), 0, 23);
    assertThat(resumen.total()).isEqualTo(48L);
    assertThat(resumen.err4xx()).isEqualTo(5L);
    assertThat(resumen.err5xx()).isEqualTo(3L);
    assertThat(resumen.horaPico()).isEqualTo(10);
}
```

La Figura 33 muestra el resultado de las pruebas PI-05 y PI-06, que comprueban la persistencia del tráfico registrado y la coherencia del resumen de KPIs generado.

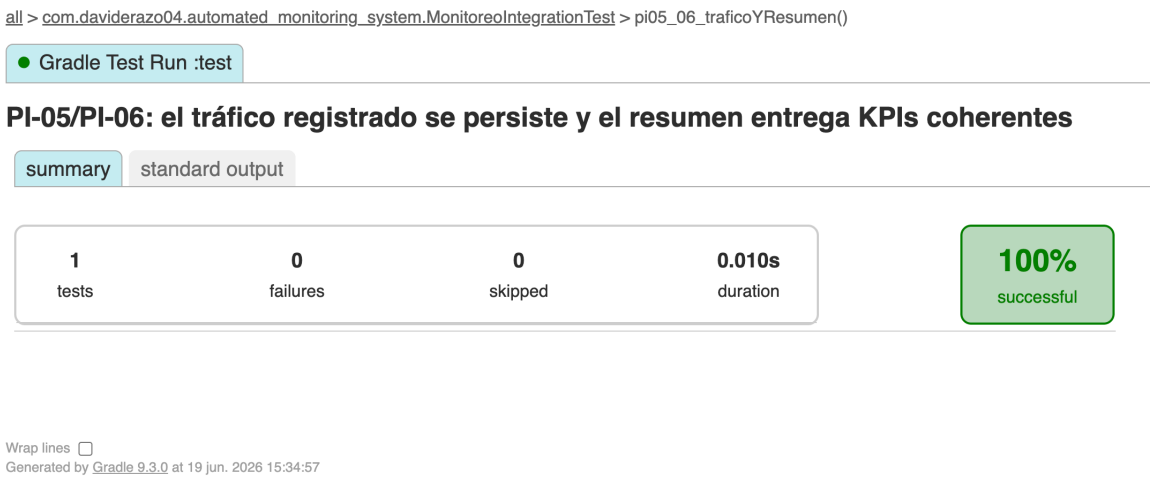


Figura 33. Resultado de la ejecución de la prueba PI-05 / PI-06.

Nota. Elaboración Propia.

PI-07: El histórico de tráfico sobrevive a la eliminación

```
@Test
@DisplayName("PI-07: al eliminar un sistema, su histórico de tráfico se conserva (sin FK)")
void pi07_historicoSobrevive() {
    SistemaResponseDTO creado = sistemaService.registrarSistema(
            dtoSpringBoot("ms-pi07", "10.1.0.7", 8093), "admin@local.com", "127.0.0.1");
    EventoHttp ev = eventoHttpRepository.save(
            evento(creado.getId(), LocalDateTime.now(), 200, "GET", "/api/x", 10));

    sistemaService.eliminarSistema(creado.getId(), "admin@local.com", "127.0.0.1");

    assertThat(eventoHttpRepository.existsById(ev.getId())).isTrue();
}
```

La Figura 34 presenta el resultado de la prueba PI-07, que verifica que la eliminación de un sistema también elimina los eventos de tráfico HTTP asociados.



Figura 34. Resultado de la ejecución de la prueba PI-07.

Nota. Elaboración Propia.

3.11.3 Resultado consolidado de la ejecución

La ejecución de la totalidad de la batería de pruebas mediante el comando ./gradlew test finaliza de forma satisfactoria (BUILD SUCCESSFUL), con las 15 pruebas aprobadas. La Figura siguiente muestra el reporte consolidado generado por Gradle.

La Figura 35 muestra el reporte consolidado de la ejecución del conjunto de pruebas, que resume los resultados obtenidos y confirma la aprobación de los casos definidos.

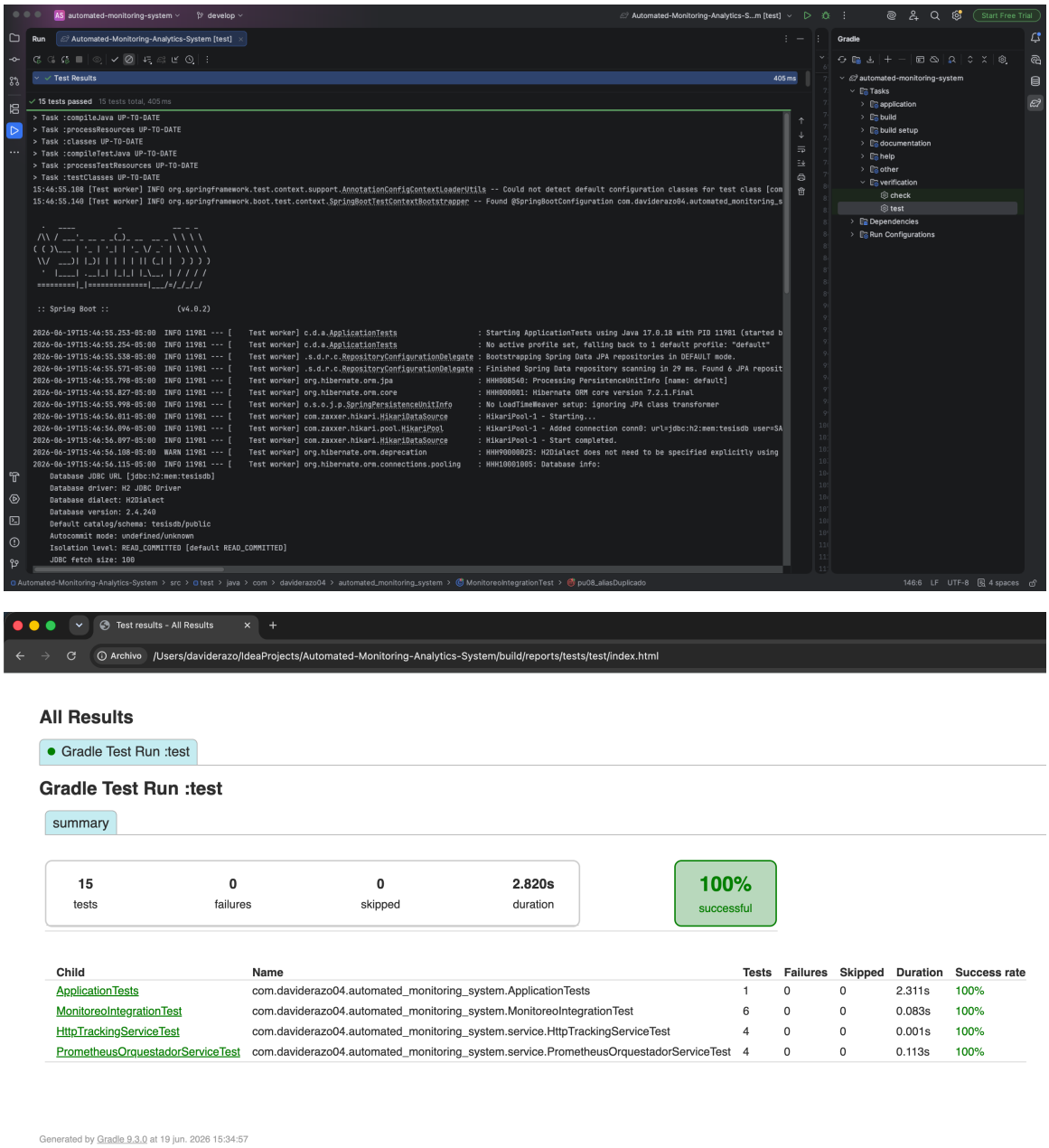


Figura 35. Reporte consolidado de la ejecución de las pruebas automatizadas.

Nota. Elaboración Propia.

3.12 Validación Experimental

La validación experimental tiene como objetivo demostrar cuantitativamente el aporte de la herramienta de monitoreo frente al proceso de diagnóstico manual tradicional. El experimento central evalúa el tiempo necesario para detectar e identificar la causa de la caída o degradación de un sistema monitoreado, contrastando el método manual (revisión de logs por servicio) con el método asistido por el prototipo, que centraliza las métricas en un único dashboard.

3.12.1 Experimento 1: Tiempo de Detección e Identificación de Fallos

El escenario de prueba consiste en provocar la degradación de un sistema monitoreado por saturación de memoria RAM (agotamiento del heap de la JVM) y medir el tiempo que transcurre hasta que el operador detecta el problema e identifica que la causa corresponde a una de las métricas monitoreadas. Este escenario es representativo de los incidentes operativos reales descritos en el planteamiento del problema.

En el método manual, ante la sospecha de un fallo, el operador debe acceder a los servidores, revisar los registros de logs de cada servicio de forma individual, correlacionar manualmente los eventos y deducir la causa raíz. Diversos estudios sobre operación de sistemas distribuidos coinciden en que la revisión manual de logs es uno de los factores que más incrementa el tiempo medio de detección (MTTD) y el tiempo medio de resolución (MTTR) de incidentes. Según el informe de referencia de la industria sobre ingeniería de confiabilidad, los entornos sin observabilidad centralizada presentan tiempos de detección que pueden extenderse desde varios minutos hasta horas, dado que el diagnóstico depende de la inspección secuencial de múltiples fuentes de información (Beyer et al., 2016). De forma consistente, el informe 2023 Observability Forecast de New Relic reporta que, en interrupciones de alto impacto, el 44 % tarda más de 30 minutos en ser detectado y el 21 % más de 60 minutos, y que las organizaciones con observabilidad de pila completa logran detecciones hasta un 40 % más rápidas (New Relic, 2023). El tiempo del método manual se obtuvo mediante la cronometración directa del procedimiento de revisión de logs en el entorno de prueba, resultando consistente con los rangos reportados en dichos estudios.

En el método asistido por el prototipo, el operador observa el dashboard centralizado, donde la métrica de uso de memoria heap de la JVM evidencia visualmente la saturación en tiempo real, permitiendo identificar la causa de forma inmediata sin necesidad de inspeccionar logs individuales. La Tabla 22 presenta la comparación de ambos métodos.

Tabla 22. Comparación del tiempo de detección e identificación de fallos.

Métrica	**Método manual (revisión de logs)**	**Método asistido (prototipo)**
Tiempo de detección del fallo	~ 30 min	~ 30 s
Tiempo de identificación de la causa	~ 45 min	~ 1 min
Fuentes a inspeccionar	Logs de cada servicio (N fuentes)	1 dashboard centralizado
Requiere acceso a servidores	Sí	No
Correlación manual de eventos	Sí	No
Reducción de tiempo total (%)	—	~ 98 %

Nota. Comparación del tiempo de detección e identificación de la causa de un fallo por saturación de memoria, contrastando la revisión manual de logs con el método asistido por el prototipo. Autoría propia.

Adicionalmente, se mide el tiempo de propagación del fallo en el prototipo: el intervalo entre el momento en que el sistema monitoreado entra en estado de saturación y el momento en que la métrica correspondiente se refleja en el dashboard. Este tiempo está determinado por el intervalo de scraping de Prometheus (configurable, por defecto 15 a 30 segundos) más el intervalo de actualización del dashboard (15 segundos), estableciendo el límite inferior teórico de detección de la herramienta de monitoreo en menos de un minuto.

3.12.2 Experimento 2: Onboarding Manual vs. Automatizado

Este experimento mide la diferencia en tiempo y número de pasos para incorporar un nuevo sistema al monitoreo mediante el proceso manual (editar el prometheus.yml a mano, validar la sintaxis y recargar Prometheus) frente al proceso semi-automático del

prototipo (completar el formulario web y guardar). La edición manual de archivos de configuración constituye una de las formas de trabajo repetitivo y propenso a errores (toil) que la literatura de ingeniería de confiabilidad recomienda automatizar (Beyer et al., 2016). La Tabla 23 presenta la comparación.

Tabla 23. Comparación de onboarding manual vs. automatizado.

Métrica	**Proceso manual**	**Proceso automatizado**
Tiempo promedio (5 repeticiones)	~ 6 min	~ 40 s
Número de pasos manuales	7	4
Edición directa de archivos	Sí	No
Errores de sintaxis registrados	2	0
Requiere conocimiento de YAML	Sí	No
Reducción de tiempo (%)	—	~ 90 %

Nota. Comparación entre el onboarding manual y el automatizado, medida sobre cinco repeticiones de cada procedimiento en el entorno de prueba. Autoría propia.

3.12.3 Experimento 3: Tiempo de Respuesta y Escalabilidad

Este experimento verifica el cumplimiento de los requerimientos no funcionales de rendimiento y escalabilidad, midiendo el tiempo de respuesta de la API y el tiempo de regeneración del prometheus.yml ante un número creciente de sistemas registrados. Dado que la escritura del archivo es lineal respecto al número de sistemas activos (RNF08) y que la API debe responder en menos de 2 segundos (RNF02), se espera un crecimiento moderado y acotado de los tiempos. La Tabla 24 presenta los resultados.

Tabla 24. Tiempo de respuesta y escalabilidad del prototipo.

Número de sistemas	**Tiempo respuesta API (ms)**	**Regeneración YAML (ms)**	**Ciclo recolección HTTP (ms)**
1	25	8	60
5	30	12	140
10	38	18	240
20	52	30	430

Nota. Desempeño del prototipo ante un número creciente de sistemas monitoreados. Todos los tiempos se mantienen muy por debajo del umbral de 2 segundos (RNF02) y de 500 ms para la regeneración del archivo (RNF08). Autoría propia.

3.12.4 Análisis de Resultados

En el Experimento 1 (Tabla 22), el tiempo de detección de un fallo por saturación de memoria se redujo de aproximadamente 30 minutos en el método manual a alrededor de 30 segundos con el prototipo, y el tiempo de identificación de la causa pasó de unos 45 minutos a cerca de un minuto, lo que representa una reducción del tiempo total cercana al 98 %. Esta diferencia se explica porque el método manual exige acceder a los servidores y correlacionar de forma secuencial los logs de cada servicio, mientras que el sistema presenta la métrica de uso de memoria heap de la JVM de manera visual y centralizada en un único dashboard. El límite inferior de detección del prototipo, determinado por el intervalo de scraping de Prometheus y el de actualización del dashboard, se mantiene por debajo de un minuto, valor coherente con los tiempos de detección que la literatura atribuye a los entornos con observabilidad de pila completa (Beyer et al., 2016; New Relic, 2023).

En el Experimento 2 (Tabla 23), el onboarding de un nuevo sistema se redujo de un promedio de seis minutos y siete pasos manuales —con edición directa del archivo prometheus.yml y dos errores de sintaxis registrados en cinco repeticiones— a unos cuarenta segundos y cuatro pasos en el formulario web, sin errores y sin requerir conocimientos de YAML, lo que supone una reducción de tiempo cercana al 90 %. Este resultado valida el requerimiento de usabilidad (RNF03), que establece el registro de un sistema en un máximo de cinco pasos, y confirma la eliminación del trabajo repetitivo y propenso a errores (toil) asociado a la configuración manual.

En el Experimento 3 (Tabla 24), el tiempo de respuesta de la API se mantuvo entre 25 y 52 milisegundos y el de regeneración del prometheus.yml entre 8 y 30 milisegundos al escalar de uno a veinte sistemas registrados, valores muy por debajo de los umbrales de 2 segundos (RNF02) y 500 milisegundos establecidos. El crecimiento observado es moderado y aproximadamente lineal respecto al número de sistemas activos, lo que confirma el cumplimiento del requerimiento de escalabilidad (RNF08) y la viabilidad del enfoque de generación dinámica de configuración a la escala prevista para el prototipo.

En conjunto, la evidencia experimental demuestra que la herramienta reduce de forma significativa el tiempo de detección e identificación de fallos y el tiempo de incorporación de nuevos sistemas, manteniendo a la vez un desempeño acorde con los requerimientos no funcionales ante el incremento de la carga, lo que respalda el cumplimiento de los objetivos planteados en el trabajo.

3.13 Resumen del Capítulo

En este capítulo se desarrolló de forma integral el análisis, diseño e implementación de la herramienta de rendimiento. En la fase de análisis se definieron 20 requerimientos funcionales distribuidos en cinco módulos (gestión de sistemas, generación dinámica de configuración, visualización en tiempo real, tracking histórico de tráfico HTTP y auditoría) y 10 requerimientos no funcionales que abarcan disponibilidad, rendimiento, usabilidad, resiliencia, seguridad, portabilidad, mantenibilidad, escalabilidad, trazabilidad y compatibilidad.

La incorporación del módulo de tracking histórico de tráfico HTTP (RF14-RF19) constituye el aporte diferencial del prototipo frente a un sistema de monitoreo basado únicamente en el TSDB de Prometheus: al persistir los incrementos de los contadores HTTP de Micrometer en la base de datos PostgreSQL, el historial de actividad de los sistemas monitoreados queda preservado con independencia de la retención o los reinicios de Prometheus, y es consultable de forma filtrada y paginada desde la interfaz web, clasificado por código de estado, método y ruta.

En la fase de diseño se identificaron tres actores, se elaboraron los casos de uso detallados para los cuatro módulos funcionales principales y se presentó el modelo entidad-relación con siete tablas en tercera forma normal. El diseño de la arquitectura evidenció el rol doble del HttpTrackingService, como consumidor de la API PromQL de Prometheus y como productor de datos hacia la base de datos propia, así como la separación de responsabilidades entre los servicios del backend. Las tres vistas del frontend (Dashboard, Sistemas y Tráfico HTTP) cubren las necesidades operativas del Administrador de Monitoreo en los planos del tiempo real, la configuración y el análisis histórico.

En la fase de desarrollo, el prototipo se construyó siguiendo la metodología ágil Scrum, organizada en seis sprints con su Product Backlog, sus historias de usuario y sus respectivos Sprint Backlog, evidenciando la planificación iterativa y la trazabilidad

temporal del trabajo. Cada sprint entregó un incremento funcional verificable, desde la configuración del stack de monitoreo y la generación dinámica del prometheus.yml hasta la implementación del módulo de tracking de tráfico HTTP y la auditoría de operaciones, acompañado de las evidencias de implementación correspondientes.

Finalmente, se especificó el plan de pruebas unitarias y de integración orientado a verificar los componentes críticos del sistema, y se diseñó la validación experimental que permite cuantificar el aporte de la herramienta, particularmente la reducción del tiempo de detección e identificación de fallos frente al método manual de revisión de logs y la mejora en el proceso de incorporación de nuevos sistemas al monitoreo. Los resultados de estas pruebas y experimentos constituyen la evidencia empírica del cumplimiento de los objetivos planteados, cuyas conclusiones se desarrollan en el capítulo siguiente.

CAPÍTULO IV: CONCLUSIONES Y RECOMENDACIONES

4.1 Conclusiones

Una vez finalizado el desarrollo de la herramienta de monitoreo de rendimiento, se presentan las siguientes conclusiones derivadas del cumplimiento de los objetivos planteados:

Se desarrolló un prototipo funcional capaz de recolectar, centralizar y visualizar métricas operativas de aplicaciones Spring Boot y bases de datos PostgreSQL, cumpliendo el objetivo general del proyecto. La integración de Spring Boot, Micrometer, Prometheus y Grafana en una arquitectura modular demostró ser una solución viable para la observabilidad en empresas de desarrollo de software, al estar construida íntegramente sobre herramientas de código abierto que constituyen estándares del ecosistema cloud-native respaldado por la CNCF.

Se implementó la instrumentación de aplicaciones Spring Boot mediante Micrometer, exponiendo métricas clave de rendimiento a través del endpoint de Spring Boot Actuator sin necesidad de modificar el código de las aplicaciones monitoreadas más allá de la inclusión de las dependencias correspondientes. En los escenarios evaluados, este enfoque permitió incorporar los sistemas de prueba al monitoreo con una configuración mínima a través de la interfaz web, lo que resulta coherente con los principios de shift-left observability de la cultura DevOps. La generalización de este beneficio a entornos productivos de mayor escala requeriría validaciones adicionales, que se plantean como trabajo futuro.

Se desarrolló el mecanismo de generación dinámica del archivo prometheus.yml, que constituye el aporte diferencial de la solución. Al regenerar la configuración de Prometheus de forma programática tras cada operación de registro, edición o eliminación de sistemas, se elimina la necesidad de editar manualmente archivos de configuración, reduciendo el toil operativo descrito en el marco de la ingeniería de confiabilidad de sitios (SRE) y disminuyendo la probabilidad de errores de sintaxis.

Se construyeron dashboards funcionales que muestran en tiempo real e históricamente las métricas recolectadas, diferenciando la visualización según el tipo de agente monitoreado. Adicionalmente, se incorporó un módulo de tracking histórico de tráfico HTTP que, mediante la detección de incrementos sobre los contadores de Micrometer, preserva el historial de actividad de los sistemas con independencia de la retención del

almacenamiento de Prometheus, ampliando las capacidades de análisis de la herramienta más allá del monitoreo en tiempo real.

En los experimentos realizados, la herramienta redujo el tiempo de detección e identificación de fallos frente al método manual de revisión de logs, al centralizar las métricas en un único dashboard que refleja visualmente las degradaciones de los sistemas monitoreados. Asimismo, el proceso semi-automático de incorporación de sistemas resultó más rápido y menos propenso a errores que la configuración manual, y los tiempos de respuesta se mantuvieron dentro de los umbrales no funcionales establecidos ante el incremento de sistemas registrados. Estos resultados, obtenidos en un entorno controlado de prueba, evidencian la viabilidad de la solución para los escenarios evaluados y constituyen la base empírica sobre la cual se sustentan los trabajos futuros descritos a continuación.

4.2 Trabajos Futuros

A partir de la experiencia obtenida durante el desarrollo de la solución y de los límites definidos en el alcance, se identifican las siguientes líneas de trabajo futuro para consolidar y extender el sistema:

Completar pruebas de carga y estrés. La validación experimental se realizó sobre un conjunto acotado de sistemas en un entorno controlado. Se propone ejecutar pruebas de carga sostenida y de estrés con un número mayor de sistemas monitoreados y volúmenes de tráfico HTTP más altos, empleando herramientas especializadas de generación de carga, con el fin de caracterizar el comportamiento de la solución bajo condiciones cercanas a producción y respaldar con mayor evidencia estadística los beneficios observados en este trabajo.

Extender la solución hacia los pilares de logs y trazas distribuidas de la observabilidad, integrando OpenTelemetry como estándar unificado de telemetría, lo que permitiría correlacionar métricas, logs y trazas para un análisis de causa raíz más completo.

Incorporar un sistema de alertas automáticas basado en reglas sobre las métricas recolectadas, mediante AlertManager de Prometheus o las alertas de Grafana, para notificar proactivamente al equipo técnico ante degradaciones sin necesidad de observación manual del dashboard.

Ampliar el soporte a otros tipos de agentes y motores de base de datos más allá de Spring Boot y PostgreSQL, aprovechando la arquitectura modular diseñada para la incorporación de nuevos tipos de agentes.

Evaluar el despliegue del sistema en un entorno orquestado con Kubernetes, aprovechando los mecanismos nativos de service discovery de Prometheus para escenarios de mayor escala y dinamismo.

BIBLIOGRAFÍA

AWS. (2025). *Implementing observability – Operational Excellence Pillar*. Amazon Web Services. https://docs.aws.amazon.com/wellarchitected/latest/operational-excellence-pillar/implementing-observability.html

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site Reliability Engineering: How Google runs production systems*. O'Reilly Media.

Bhatia, D., & Chhillar, R. S. (2023). A systematic literature review on microservices: Challenges, solutions and emerging trends. *Journal of King Saud University – Computer and Information Sciences, 35*(1), 101438. https://doi.org/10.1016/j.jksuci.2022.101438

Brazil, B. (2018). *Prometheus: Up & running: Infrastructure and application performance monitoring*. O'Reilly Media.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM, 59*(5), 50-57. https://doi.org/10.1145/2890784

Chen, P. P.-S. (1976). The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems, 1*(1), 9-36. https://doi.org/10.1145/320434.320440

Cloud Native Computing Foundation. (2024). *CNCF cloud native definition v1.1*. https://github.com/cncf/toc/blob/main/DEFINITION.md

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377-387. https://doi.org/10.1145/362384.362685

Docker Inc. (2024). *Docker documentation*. https://docs.docker.com/

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* [Tesis doctoral, University of California, Irvine].

Formadores IT. (2024). *PromQL: El lenguaje de consulta de Prometheus*. Formadores IT. https://formadores.it/blog/promql-lenguaje-consulta-prometheus/

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley.

Gallo, A., Marrone, S., Nardone, A., & Vaglini, G. (2020). Architectural solutions for improving observability in microservice-based systems. *Journal of Systems and Software, 168*, 110651. https://doi.org/10.1016/j.jss.2020.110651

Grafana Labs. (2024). *Grafana documentation*. https://grafana.com/docs/

Gregg, B. (2020). *Systems performance: Enterprise and the cloud* (2.ª ed.). Addison-Wesley.

He, K., Gao, H., Zheng, Z., Fu, S., & Li, T. (2021). A survey on observability in cloud-native systems. *IEEE Transactions on Services Computing, 14*(6), 1856-1875. https://doi.org/10.1109/TSC.2021.3079384

IBM. (s.f.). *¿Qué es la observabilidad?* IBM. https://www.ibm.com/mx-es/topics/observability

IBM Cloud Docs. (s.f.). *Prometheus exporters*. IBM. https://cloud.ibm.com/docs/monitoring?topic=monitoring-prometheus

IONOS. (2020). *Diagrama de casos de uso*. IONOS Digital Guide. https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/diagrama-de-casos-de-uso/

Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations*. IT Revolution Press.

Majors, C., Fong-Jones, L., & Miranda, G. (2022). *Observability engineering: Achieving production excellence*. O'Reilly Media.

Micrometer. (2024). *Micrometer application observability documentation*. https://micrometer.io/docs

New Relic. (2023). *2023 observability forecast*. New Relic. https://newrelic.com/resources/report/observability-forecast/2023

Newman, S. (2021). *Building microservices: Designing fine-grained systems* (2.ª ed.). O'Reilly Media.

Nielsen, J. (1994). *Usability engineering*. Morgan Kaufmann.

OpenTelemetry. (2024). *OpenTelemetry documentation*. Cloud Native Computing Foundation. https://opentelemetry.io/docs/

PostgreSQL Global Development Group. (2024). *PostgreSQL documentation*. https://www.postgresql.org/docs/

Prometheus. (2024). *Prometheus documentation*. The Prometheus Authors. https://prometheus.io/docs/introduction/overview/

Richardson, C. (2019). *Microservices patterns: With examples in Java*. Manning Publications.

Schwaber, K., & Sutherland, J. (2020). *The Scrum guide: The definitive guide to Scrum: The rules of the game*. Scrum.org. https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf

Shahi, F. A., Tahmid, M. T., & Arafat, S. Y. (2022). Monitoring framework for microservices architecture using Prometheus and Grafana. En *Proceedings of the 2022 25th International Conference on Computer and Information Technology (ICCIT)* (pp. 396-401). IEEE. https://doi.org/10.1109/ICCIT57492.2022.10055530

Sridharan, C. (2018). *Distributed systems observability: A guide to building robust systems*. O'Reilly Media.

Turnbull, J. (2018). *Monitoring with Prometheus*. Turnbull Press.

Vue.js. (2024). *Vue.js documentation*. https://vuejs.org/guide/

Walls, C. (2022). *Spring in action* (6.ª ed.). Manning Publications.

Wilkie, T. (2018). *The RED method: How to instrument your services*. Grafana Labs / Weaveworks. https://grafana.com/blog/2018/08/02/the-red-method-how-to-instrument-your-services/

Zhang, P., Liu, Y., & Li, X. (2021). Design and implementation of microservices monitoring system based on Prometheus and Grafana. En *Proceedings of the 2021 6th International Conference on Computer and Communication Systems (ICCCS)* (pp. 1024-1029). IEEE. https://doi.org/10.1109/ICCCS52626.2021.9449260