# Estrategias de despliegue, monitoreo y observabilidad para la plataforma Smart Campus UIS

Cristian Manuel Hernández Delgado y Oscar Iván Torrens Bustamante

Trabajo de grado para optar al título de Ingeniero de Sistemas e Informática

Director

Gabriel Rodrigo Pedraza Ferreira

Doctor en Ciencias de la Computación

Codirector

Henry Andres Jimenez Herrera

Magíster en Ingeniería de Sistemas e Informática

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingeniería de Sistemas e Informática

Ingeniería de sistemas

Bucaramanga

2026

## Dedicatoria

Dedico este trabajo a mi familia, especialmente a mis padres y a mis tías, quienes me brindaron su apoyo incondicional durante mi vida universitaria. Gracias por acompañarme en este camino, por creer en mí y por todo el esfuerzo, el cariño y la dedicación que me han brindado. Los quiero profundamente y les estaré eternamente agradecido por todo lo que han hecho por mí. Este logro también les pertenece, porque sin ustedes el camino habría sido muy distinto.

**Cristian Manuel Hernández Delgado**

**Agradecimientos**

A mis padres y a mis tías, por su amor incondicional, por estar presentes en cada etapa de mi vida y por brindarme siempre su apoyo, incluso en los momentos más difíciles. Gracias por creer en mí, por darme fuerzas cuando las necesitaba y por acompañarme durante este camino con su cariño y comprensión. Les agradezco profundamente cada esfuerzo, cada consejo y cada gesto que me ayudó a seguir adelante, tanto en mi vida universitaria como fuera de ella. Este logro representa también una parte de todo lo que ustedes me han dado, y me llena de alegría poder compartirlo con ustedes. Los quiero mucho y les estaré eternamente agradecido por todo lo que han hecho por mí.

Al profesor Henry Jiménez, por su disposición, orientación y apoyo durante el desarrollo de este proyecto, y por acompañarnos a lo largo de todo el proceso.

Al profesor Gabriel Pedraza, por confiar en nosotros y brindarnos la oportunidad de llevar a cabo este proyecto, permitiéndonos asumir este reto y poner en práctica nuestros conocimientos.

A Jas y a todos mis amigos, por su amistad, compañía y apoyo, que me ayudaron a sobrellevar esta etapa de mi vida. Gracias por hacer más llevadero el camino, por los momentos compartidos y por estar presentes durante este proceso.

**Cristian Manuel Hernández Delgado**

## Dedicatoria

A mis padres, por enseñarme a sobreponerme a las dificultades; a mis tías, por enseñarme que la fortaleza no está reñida con la ternura; y a quienes, sin tener la obligación de hacerlo, hicieron que el camino fuera menos solitario.

A los maestros que dejaron en mí algo más que conocimientos, y a los amigos con quienes las dificultades pesaron menos y las alegrías significaron más.

Este logro lleva mi nombre, pero no se construyó únicamente con mis esfuerzos. También pertenece a quienes creyeron en mí, me tendieron la mano y, de distintas maneras, contribuyeron a que llegara hasta aquí.

**Oscar Iván Torrens Bustamante**

**Agradecimientos**

Esta profesión, bien desempeñada, exige cierto pragmatismo, y ese pragmatismo no siempre se lleva bien con la sensibilidad que requiere expresar gratitud. Por eso, si en estas líneas no logro rendir los honores como corresponde, ofrezco de antemano mis disculpas.

A la existencia de este documento, como a la asignación de clases un sábado por la mañana, le surgieron infinitos detractores: la vida, la muerte, el desempleo, la desidia, la burocracia y hasta la ineficiencia técnica del sector privado colombiano. Si aun así existe, es gracias a ustedes, Oscar Enrique Torrens Gaona y Bertha Cecilia Bustamante López, mis padres, quienes me enseñaron que sus apellidos juntos significan sobreponerse a la dificultad y triunfar incluso con el mundo en contra.

Mis padres me enseñaron lo bueno y lo malo, pero fueron mis tías, Mary y Fidian, quienes me enseñaron a ser humano. Me protegieron de la soledad todo lo que pudieron e hicieron que mi infancia fuese memorable. Lo poco que sé de dar afecto y de querer se lo debo a ambas. Quienes me enseñaron que soñar no estaba mal fueron ustedes.

La vida de foráneo, aunque sea regalada, es cara. La familia, lejos. Los amigos, aunque pocos, lejos también. Ya no existe la posibilidad de salir del cuarto e ir a verlos. Quien quiera enfrentarse a sus demonios, que decida ser foráneo; ahí entenderá lo que es estar a solas con ellos. A pesar de lo anterior, siempre hubo alguien que, dentro de sus posibilidades, evitó que batallara solo con ellos: el ingeniero Juan Carlos Escobar, que me trató como a un hijo, me dio consejos e incluso me ayudó a sortear las crisis. Celebraba mis logros como propios y me corregía siempre que fuese necesario,

incluso cuando sabía que no iba a hacer caso. Seré muchas cosas, pero no un mal hijo. Honor a quien honor merece.

Las clases comenzaron y todo iba lento: ir, escuchar, ir a otro salón y repetir; esperar al día siguiente por lo mismo. Siempre he sido alguien de números y académicamente me he considerado decente; nunca supe lo que es perder una materia. Quien rompió la monotonía y recompensó debidamente mi deseo de aprender fue el profesor Oscar Lozano Mantilla. Creyó en mis capacidades y valoró el esfuerzo que vio en mí. Se podría decir que fue la primera persona que supo reconocer mis talentos en la universidad.

Pasaban las materias interesantes y empezaban a llegar las de «relleno»: Dirección Empresarial. Siendo honesto, la vi como un mal que tenía que aguantar; no esperaba absolutamente nada. En mi mente iban a ser un par de horas semanales que había que tolerar para pasar la página lo más pronto posible. Fue un error de juicio que agradezco, porque me encontré con alguien a quien algunos tildan de excéntrica solo porque no conocen bien el significado de la palabra genio: la profesora Angélica María Díaz Gómez. Si con alguien he tenido charlas interesantes, he encontrado problemas nuevos y he descubierto que no existe un eufemismo más grande que la expresión «soft skill», ha sido con ella. Es la única persona que siempre encuentra la forma de hacerme reflexionar sobre cosas que creía resueltas.

A medida que avanzaban las materias de carrera, me encontré con un profesorado de lo más variopinto; no por nada llegué a acuñar la frase: «En la UIS no hay personas, solo personajes». Encontré de todo: profesores reservados, otros de trato difícil e incluso algunos que se sentían en la cúspide… de su propia megalomanía. Entre tantos personajes, conocí a un profesor que se destacaba

por su rigurosidad y, a la vez, por su calidez. Sus problemas de clase resultaban más complejos que los de aquellos que se creían en la cúspide, solo que sin tanto bombo y platillo. Me refiero al profesor Jonnathan Ramos Chaux, de quien hoy, gracias a esa calidez, puedo decir que es mi amigo.

Entre más nos acercábamos al final de la carrera, mayores eran los obstáculos para culminarla. No queríamos cualquier final; queríamos salir a la industria con conocimiento sólido y habilidades técnicas envidiables. Si hubo alguien que nos permitió tener el final que queríamos, que tuvo la paciencia para estar a nuestro lado e incluso llegó a abogar por nosotros cuando nadie quería recibirnos, ese fue el profesor Henry Jiménez. Profesor, todavía no alcanzo a explicarme por qué hizo lo que hizo; sin embargo, una vez más se lo digo: mil gracias.

Profesores Oscar, Angélica, Chaux y Henry: espero que llegue el día en que la UIS reconozca que la verdadera acreditación de calidad no la da un documento, sino la cantidad de estudiantes que pueden decir que hubo un antes y un después de ustedes. Su labor vale mucho más de lo que un contrato de cátedra puede reflejar, y por eso siempre estaré en deuda con ustedes.

Si alguien sabe, además de mí, que la parte más fácil del proceso de formación es el proceso de formación mismo, ese es mi compañero Cristian Manuel Hernández Delgado. Cada problema lo resolvimos codo a codo, incluso cuando parecían sacados de una novela de Victor Hugo. En el mundo existen duplas irrepetibles y esta, sin temor a equivocarme, es una de ellas. Hoy puedo decir que, más que mi compañero de trabajo de grado, es mi amigo.

**Oscar Iván Torrens Bustamante**

**Tabla de contenido**

Introducción 25
1. Planteamiento y justificación del problema 26
2. Objetivos 30
2.1 Objetivo general 30
2.2 Objetivos específicos 30
3. Marco de referencia 31
3.1 Fundamentos Teóricos 31
3.1.1 Internet de las Cosas 31
3.1.2 Arquitecturas de Microservicios 32
3.1.3 Contenedores y Orquestación 34
3.1.4 Brokers de Mensajería y Protocolos IoT 36
3.1.5 Observabilidad en Sistemas Distribuidos 37
3.1.6 DevOps y Estrategias de Despliegue 39
3.2 Contexto Tecnológico 40
3.2.1 Plataformas IoT Basadas en Microservicios 40
3.2.2 Ecosistema de Herramientas de Observabilidad 41
3.2.3 Desafíos Operacionales en Implementaciones IoT Distribuidas 42
3.3 Revisión del estado del arte 43
3.3.1 Fuentes Consultadas 44
3.3.1.1 Fuentes Académicas 44
3.3.1.2 Fuentes Técnicas 46
3.3.2 Estrategias de Despliegue 49
3.3.3 Monitoreo y Observabilidad 50
3.3.4 Síntesis de Hallazgos 51
3.4 Herramientas Evaluadas 52
3.4.1 Plataformas de Orquestación 52
3.4.2 Herramientas de Métricas 53
3.4.3 Herramientas de Logs 53
3.4.4 Herramientas de Trazas Distribuidas 54
3.4.5 Herramientas de Visualización 55
4. Metodología 56
4.1 Fase 1: Caracterización de la Plataforma Smart Campus UIS 57
4.1.1 Análisis de la arquitectura actual 58
4.1.2 Diagnóstico del sistema de despliegue, monitoreo y observabilidad actual 58
4.1.3 Mapeo de dependencias e identificación de componentes críticos 58

4.1.4 Establecimiento de requerimientos funcionales 59
4.1.5 Especificación de requerimientos no funcionales 59
4.2 Fase 2: Establecimiento de Estrategia Integrada 59
4.2.1 Revisión del estado del arte 60
4.2.2 Establecimiento de criterios de evaluación 60
4.2.3 Identificación de integraciones tecnológicas 60
4.2.4 Evaluación comparativa de alternativas 61
4.2.5 Selección justificada de estrategias y herramientas 61
4.3 Fase 3: Adaptación de Servicios Smart Campus UIS 61
4.3.1 Análisis de requerimientos de adaptación 62
4.3.2 Adaptación de servicios para estrategia de despliegue 62
4.3.3 Instrumentación de servicios para monitoreo y observabilidad 62
4.4 Fase 4: Implementación de Estrategia Integrada 63
4.4.1 Diseño de la arquitectura de despliegue 63
4.4.2 Diseño del sistema de monitoreo 63
4.4.3 Diseño del sistema de observabilidad 64
4.4.4 Configuración de la infraestructura 64
4.4.5 Implementación del sistema de monitoreo 64
4.4.6 Implementación del sistema de observabilidad 64
4.4.7 Implementación de la estrategia de despliegue 65
4.4.8 Migración de servicios 65
4.5 Fase 5: Validación Experimental 65
4.5.1 Validación de estrategias de despliegue 66
4.5.2 Validación de sistemas de monitoreo y observabilidad 66
4.5.3 Validación de configuraciones 66
4.5.4 Análisis de resultados y trabajo futuro 66
5. Caracterización de la plataforma Smart Campus UIS 67
5.1 Análisis de la arquitectura actual 67
5.1.1 Frontend de Administración 69
5.1.2 API Gateway 70
5.1.3 Microservicio de Administración 70
5.1.4 Microservicio de Datos 70
5.1.5 Sistemas de Almacenamiento 71
5.1.6 Brokers de Mensajería 71
5.1.7 Flujo de Datos de Mensajería 72
5.1.7.1 Flujo MQTT (Dispositivos IoT) 72
5.1.7.2 Flujo AMQP (Mensajería Empresarial) 72
5.1.8 Casos de Uso 73

5.1.9 Diagrama de componentes 76
5.2 Diagnóstico del sistema de despliegue, monitoreo y observabilidad actual 77
5.2.1 Estado Actual del Despliegue 78
5.2.1.1 Mecanismos Existentes 78
5.2.1.2 Sin Pipeline de CI/CD 78
5.2.1.3 Sin Health Checks 78
5.2.1.4 Gestión de Secretos Insegura 78
5.2.1.5 Sin Versionamiento de Imágenes 79
5.2.1.6 Sin Estrategias de Despliegue 79
5.2.2 Estado Actual del Monitoreo 81
5.2.2.1 Componentes Existentes 81
5.2.2.2 Grafana Sin Configuración 81
5.2.2.3 Spring Actuator Sin Exponer 82
5.2.2.4 Sin Prometheus ni Métricas 82
5.2.2.5 Sin Sistema de Alertas 82
5.2.3 Estado Actual de la Observabilidad 82
5.2.3.1 Componentes Existentes 83
5.2.3.2 Logging No Estructurado 83
5.2.3.3 Sin Tracing Distribuido 83
5.2.3.4 Manejo de Excepciones Deficiente 83
5.3 Mapeo de dependencias e identificación de componentes críticos 86
5.4 Establecimiento de requerimientos funcionales 87
5.5 Especificación de requerimientos no funcionales 91
5.6 Síntesis de Hallazgos 93
6. Establecimiento de la estrategia integrada 94
6.1 Establecimiento de Criterios de Evaluación 94
6.1.1 Criterios de Evaluación para Patrones de Despliegue 94
6.1.2 Criterio Excluyente: Licenciamiento Open Source 95
6.1.3 Criterios de Evaluación para Herramientas (Escala 1-5) 96
6.1.4 Justificación de la Escala de Puntuación 97
6.1.4.1 Escala para Herramientas 97
6.1.4.2 Escala para Patrones de Despliegue 100
6.2 Evaluación Comparativa de Alternativas 102
6.2.1 Evaluación Comparativa de Patrones de Despliegue 103
6.2.2 Plataformas de Orquestación 104
6.2.3 Herramientas de Métricas 105
6.2.4 Herramientas de Logs 105
6.2.5 Herramientas de Trazas Distribuidas 106

6.2.6 Herramientas de Visualización 107
6.3 Identificación de Integraciones Tecnológicas 108
6.3.1 Integración Spring Boot y Observabilidad 109
6.3.2 Integración EMQX y Observabilidad 109
6.3.3 Flujo de Datos de Observabilidad 109
6.4 Selección Justificada de Estrategias y Herramientas 110
6.4.1 Arquitectura de la Estrategia Integrada 110
6.4.2 Stack Seleccionado 113
7. Adaptación de servicios Smart Campus UIS 114
7.1 Análisis de mejoras de adaptación 114
7.1.1 Mejoras de adaptación para despliegue 114
7.1.2 Mejoras de instrumentación para monitoreo y observabilidad 115
7.2 Adaptación de servicios para estrategia de despliegue 116
7.2.1 Optimización de imágenes de contenedor 116
7.2.1.1 Optimización del Microservicio de Datos 116
7.2.1.2 Optimización del Dockerfile 117
7.2.2 Exposición de endpoints de monitoreo y health checks 119
7.3 Instrumentación de servicios para monitoreo y observabilidad 119
7.3.1 Integración de dependencias de monitoreo 119
7.3.2 Configuración de OpenTelemetry 120
7.3.3 Implementación de logging estructurado 121
8. Implementación de estrategia integrada 122
8.1 Implementación de los mecanismos de despliegue 122
8.1.1 Registro de imágenes en DockerHub 123
8.1.2 Configuración Docker Compose 124
8.1.3 Gestión del despliegue con Helm 125
8.1.4 Deployment y StatefulSet 126
8.1.5 Services e Ingress 127
8.1.6 Gestión de credenciales 128
8.1.7 Arranque y salud de los servicios 128
8.1.8 Flujo de configuración 129
8.2 Implementación del sistema de monitoreo 130
8.2.1 Métricas y alertas 131
8.2.2 Despliegue del stack 132
8.2.3 Grafana Alloy 132
8.2.4 Prometheus 134
8.2.5 Grafana 134
8.2.6 Alertas 135

8.3 Implementación del sistema de observabilidad 135
8.3.1 Grafana Alloy 135
8.3.2 Loki 136
8.3.3 Trazas distribuidas 136
8.3.4 Logs estructurados 138
8.3.5 Consideraciones de configuración del stack 139
8.4 Dashboards 140
8.4.1 Configuración de Grafana para dashboards 140
8.4.2 Dashboards de monitoreo 140
8.4.3 Dashboards de Observabilidad 141
8.4.3.1 Smart Campus - Overview 142
8.4.3.2 Smart Campus - Microservicios 143
8.4.3.3 Smart Campus - Infraestructura 145
8.4.3.4 Smart Campus - Trazas y Latencia 146
8.5 Migración de servicios 147
9. Evaluación de la estrategia 149
9.1 Hardware para la evaluación 149
9.2 Plan de Prueba 150
9.2.1 Caso 1: Comparativa de configuraciones de despliegue 154
9.2.1.1 Infraestructura para el Caso 1 154
9.2.1.2 Plan de Pruebas para el Caso 1 154
9.2.2 Caso 2: Rolling Update y Rollback 155
9.2.2.1 Infraestructura para el Caso 2 155
9.2.2.2 Plan de Pruebas para el Caso 2 155
9.2.3 Caso 3: Despliegue del stack de monitoreo 156
9.2.3.1 Infraestructura para el Caso 3 156
9.2.3.2 Plan de Pruebas para el Caso 3 156
9.2.4 Caso 4: Logs de infraestructura (Capa 1) 157
9.2.4.1 Infraestructura para el Caso 4 157
9.2.4.2 Plan de Pruebas para el Caso 4 158
9.2.5 Caso 5: Instrumentación de microservicios (Capa 2) 159
9.2.5.1 Infraestructura para el Caso 5 159
9.2.5.2 Plan de Pruebas para el Caso 5 159
9.2.6 Caso 6: Correlación cross-signal (Capa 3) 161
9.2.6.1 Infraestructura para el Caso 6 161
9.2.6.2 Plan de Pruebas para el Caso 6 161
9.2.7 Caso 7: Sistema de alertas (Capa 4) 162
9.2.7.1 Infraestructura para el Caso 7 162

9.2.7.2 Plan de Pruebas para el Caso 7 163
10. Análisis de resultados 165
10.1 Resultados del Caso 1 — Comparativa de configuraciones de despliegue 165
10.2 Resultados del Caso 2 — Rolling Update y Rollback 166
10.3 Resultados del Caso 3 — Despliegue del stack de monitoreo y observabilidad 169
10.4 Resultados del Caso 4 — Logs de infraestructura 169
10.5 Resultados del Caso 5 — Instrumentación de microservicios 170
10.6 Resultados del Caso 6 — Correlación cross-signal 171
10.7 Resultados del Caso 7 — Sistema de alertas 175
10.8 Cobertura de observabilidad 181
10.9 Capacidad de diagnóstico 182
10.10 Cumplimiento de requerimientos 185
11. Conclusiones 186
12. Trabajo futuro 189
Referencias Bibliográficas 192
Apéndices 204

## Lista de tablas

Tabla 1. Fuentes académicas consultadas 44
Tabla 2. Fuentes técnicas consultadas 47
Tabla 3. Síntesis de hallazgos, revisión estado del arte 51
Tabla 4. Casos de uso relevantes 73
Tabla 5. Mecanismos existentes 78
Tabla 6. Estado actual de monitoreo 81
Tabla 7. Estado actual observabilidad 83
Tabla 8. Mapeo de dependencias 86
Tabla 9. Requerimientos funcionales 88
Tabla 10. Requerimientos no funcionales 92
Tabla 11. Criterios de evaluación, patrones de despliegue 95
Tabla 12. Criterios de evaluación, herramientas 96
Tabla 13. Justificación escala de puntuación, herramientas 97
Tabla 14. Justificación escala de puntuación, patrones de despliegue 101
Tabla 15. Evaluación comparativa, patrones de despliegue 103
Tabla 16. Evaluación comparativa, plataformas de orquestación 104
Tabla 17. Evaluación comparativa, herramientas de métricas 105
Tabla 18. Evaluación comparativa, herramientas de logs 106
Tabla 19. Evaluación comparativa, herramientas de trazas 107
Tabla 20. Evaluación comparativa, herramientas de visualización 108
Tabla 21. Integración Spring Boot y observabilidad 109
Tabla 22. Flujo de datos de observabilidad 110
Tabla 23. Stack seleccionado 113
Tabla 24. Requerimientos de adaptación para despliegue 115
Tabla 25. Requerimientos de instrumentación para monitoreo y observabilidad 116
Tabla 26. Clases de JavaCV en el microservicio de datos 117
Tabla 27. Dependencias integradas para monitoreo 119
Tabla 28. Configuraciones según el entorno 123
Tabla 29. Tipo de recurso de Kubernetes por componente para despliegue 126
Tabla 30. Métricas relevantes y alertas 131
Tabla 31. Operaciones instrumentadas para trazas 137
Tabla 32. Logs implementados 139
Tabla 33. Paneles, dashboards de monitoreo 141
Tabla 34. Dashboards para observabilidad 142

Tabla 35. Paneles dashboard Smart Campus - Overview 143
Tabla 36. Paneles dashboard Smart Campus - Microservicios 144
Tabla 37. Paneles dashboard Smart Campus - Infraestructura 145
Tabla 38. Paneles dashboard trazas y latencia 147
Tabla 39. Fases de implementación 148
Tabla 40. Características del hardware para el entorno de evaluación 150
Tabla 41. Resumen de los siete casos del plan de prueba 152
Tabla 42. Comparativa primer despliegue 166
Tabla 43. Cobertura de observabilidad 181
Tabla 44. Capacidad de respuestas a consultas de diagnóstico 182
Tabla 45. Análisis configuración docker compose 185
Tabla 46. Análisis configuración k3s 185
Tabla A.1. Casos de uso, microservicio de administración 204
Tabla A.2. Casos de uso — recepción de datos, microservicio de datos 208
Tabla A.3. Casos de uso — consulta de datos, microservicio de datos 208
Tabla A.4. Casos de uso — gestión de archivos, microservicio de datos 210
Tabla A.5. Casos de uso — gestión de cámaras, microservicio de datos 211
Tabla A.6. Casos de uso, API Gateway 212

## Lista de figuras

Figura 1. Metodología de investigación 57
Figura 2. Diagrama de arquitectura de smart-campus uis 69
Figura 3. Diagrama de componentes de Smart Campus UIS 77
Figura 4. Actualización del microservicio de datos con Docker Compose (estado inicial) 80
Figura 5. Operación de la plataforma con Docker Compose (estado inicial) 85
Figura 6. Arquitectura de la estrategia integrada 111
Figura 7. Configuración OpenTelemetry en Dockerfile 121
Figura 8. Cuenta de DockerHub semillero CAGE 124
Figura 9. Estructura de un Helm chart 125
Figura 10. Diagrama de configuración de un Helm chart 130
Figura 11. Configuración de recolección de métricas de Grafana Alloy 133
Figura 12. Configuración de Prometheus 134
Figura 13. Configuración de Tempo 138
Figura 14. Arquitectura de las cuatro capas de observabilidad evaluadas en el plan de prueba 152
Figura 15. Rolling update, microservicio de datos 167
Figura 16. Rollback, microservicio de datos 167
Figura 17. Actualización del microservicio de datos con K3s y Helm (Rolling Update y rollback) 168
Figura 18. Estado de monitoreo y observabilidad 169
Figura 19. Dashboard de infraestructura 170
Figura 20. Dashboard de microservicios 171
Figura 21. Exemplars en Prometheus 172
Figura 22. Traza completa en Tempo 173
Figura 23. Navegación de log a traza 174
Figura 24. Navegación de traza a logs 175
Figura 25. Alerta de errores HTTP 176
Figura 26. Alerta de latencia P95 177
Figura 27. Alerta de memoria heap crítica 178
Figura 28. Alerta de servicio caído 179
Figura 29. Alerta de CPU alta sostenida 180
Figura 30. Timeline de alertas simultáneas 180
Figura 31. Operación de la plataforma con K3s, Helm y stack LGTM 184
Figura B.1. Dockerfile original del microservicio de datos 212
Figura C.1. Dockerfile optimizado del microservicio de datos 214
Figura D.1. Archivo docker-compose.yml completo 215
Figura E.1. Patrón de salida — logback-spring.xml 220

Figura E.2. Patrón de formato de salida, logback-spring.xml 220
Figura E.3. Patrón de nivel de log, application.properties / application.yml 220
Figura F.1. Values.yaml 221
Figura F.2. Values-custom.yaml 222
Figura G.1. Ejemplo de init containers en templates de despliegue 223
Figura H.1. Configuración de Alloy para metricas 224
Figura H.2. Configuración básica de Prometheus 225
Figura H.3. Configuración de Grafana para métricas 225
Figura I.1. Configuración de reglas de alerta 226
Figura J.1. Configuración de Alloy para Loki 227
Figura J.2. Configuración de Alloy para Tempo 228
Figura J.3. Configuración Loki 228
Figura J.4. Configuración Tempo 229
Figura J.5. Configuración de Grafana para observabilidad 230
Figura J.6. Configuración values custom 230
Figura K.1. Configuración de Grafana para Dashboards 231
Figura K.2. Configuración de un Dashboard 231

## Lista de apéndices

Apéndice A. Casos de uso detallados 204
Apéndice B. Dockerfile original del microservicio de datos 212
Apéndice C. Dockerfile después de la optimización 213
Apéndice D. docker-compose.yml completo 215
Apéndice E. Patrones de logging estructurado 219
Apéndice F. Archivos de configuración centralizada 221
Apéndice G. Ejemplo de init containers en templates (Deploymen/ Resultset) 223
Apéndice H. Configuraciones de métricas 224
Apéndice I. Configuración de alertas 226
Apéndice J. Configuraciones de logs 227
Apéndice K. Dashboards 231
Apéndice L. Link del repositorio 232

**Glosario**

**Agente Alloy:** componente de recolección de observabilidad utilizado para recibir, procesar y enviar datos operacionales, como registros y trazas, hacia los sistemas correspondientes.

**API (Application Programming Interface):** interfaz que define las operaciones y los mecanismos mediante los cuales un componente de software puede comunicarse con otro.

**Contenedor:** unidad de ejecución que empaqueta una aplicación y sus dependencias para ejecutarla de manera aislada y reproducible en distintos entornos.

**Despliegue:** proceso mediante el cual una aplicación y su configuración se instalan y ponen en funcionamiento en un entorno de ejecución.

**Docker:** plataforma de contenedorización utilizada para construir, distribuir y ejecutar aplicaciones empaquetadas en imágenes.

**Docker Compose:** herramienta que permite definir y ejecutar aplicaciones compuestas por varios contenedores mediante un archivo de configuración.

**DockerHub:** registro de imágenes de contenedores que permite almacenarlas y distribuirlas para su utilización durante el despliegue.

**EMQX:** plataforma de intermediación de mensajes que implementa el protocolo MQTT y permite la comunicación entre dispositivos y aplicaciones.

**Endpoint:** punto de acceso de una interfaz de software mediante el cual se expone una operación o un recurso, generalmente a través de una dirección y un método HTTP.

**Grafana:** plataforma de visualización que permite consultar y representar datos operacionales mediante paneles y gráficos.

**Helm:** herramienta de gestión de paquetes para Kubernetes que permite definir, configurar e instalar aplicaciones mediante charts.

**HTTP (Hypertext Transfer Protocol):** protocolo de comunicación utilizado para intercambiar solicitudes y respuestas entre clientes y servidores.

**HTTPS (Hypertext Transfer Protocol Secure):** versión de HTTP que protege la comunicación mediante cifrado TLS.

**InfluxDB:** sistema de almacenamiento y consulta de datos de series temporales, utilizado en la plataforma para gestionar mediciones asociadas a los dispositivos y sensores.

**IoT (Internet of Things):** paradigma tecnológico que conecta objetos físicos, como sensores y actuadores, a través de redes de comunicación para recopilar, intercambiar y utilizar datos.

**K3s:** distribución ligera de Kubernetes diseñada para simplificar la instalación y operación de clústeres de orquestación de contenedores.

**Kubernetes:** plataforma de orquestación de contenedores que permite gestionar su despliegue, disponibilidad, configuración y ciclo de vida.

**Latencia:** tiempo transcurrido entre el inicio de una operación y la obtención de su respuesta. En la evaluación de la plataforma, permite caracterizar el comportamiento temporal de los servicios y sus dependencias.

**Loki:** sistema de agregación y consulta de registros que permite centralizar los logs de los componentes de una aplicación distribuida.

**Métrica:** medición cuantitativa de una característica del sistema a lo largo del tiempo, como el uso de CPU, el consumo de memoria, la tasa de solicitudes o la duración de las operaciones.

**Microservicio:** componente de una aplicación estructurada como un servicio con responsabilidades específicas, que puede desarrollarse, desplegarse y ejecutarse de manera relativamente independiente.

**Mimir:** sistema de almacenamiento escalable de métricas compatible con Prometheus, considerado como alternativa para conservar métricas a largo plazo.

**MinIO:** sistema de almacenamiento de objetos compatible con la API de Amazon S3, utilizado para almacenar datos y artefactos que requieren persistencia.

**MongoDB:** sistema de gestión de bases de datos orientado a documentos, utilizado para almacenar información en estructuras flexibles.

**MQTT (Message Queuing Telemetry Transport):** protocolo ligero de mensajería basado en el modelo de publicación y suscripción, empleado habitualmente en entornos IoT.

**Namespace:** mecanismo de Kubernetes que permite organizar y aislar lógicamente los recursos de un clúster.

**Observabilidad:** capacidad de inferir el estado interno y el comportamiento de un sistema a partir de los datos que expone durante su ejecución, principalmente métricas, registros y trazas.

**OpenTelemetry:** conjunto de herramientas, bibliotecas y estándares para instrumentar aplicaciones y generar, recopilar y exportar datos de observabilidad.

**OTLP (OpenTelemetry Protocol):** protocolo utilizado para transportar datos de telemetría, como métricas, registros y trazas, entre componentes compatibles con OpenTelemetry.

**P95 (percentil 95):** valor por debajo del cual se encuentra el 95 % de las observaciones de una distribución. Aplicado a la latencia, permite caracterizar el comportamiento de la mayoría de las solicitudes sin limitarse al promedio.

**Pod:** unidad mínima desplegable de Kubernetes, que agrupa uno o varios contenedores que comparten recursos y un contexto de ejecución.

**PostgreSQL:** sistema de gestión de bases de datos relacional utilizado para almacenar y consultar información estructurada.

**Prometheus:** sistema de monitoreo y alertamiento que recopila métricas de los componentes instrumentados y permite consultarlas mediante su lenguaje de consulta.

**RabbitMQ:** intermediario de mensajería que permite intercambiar mensajes entre componentes mediante colas y mecanismos de enrutamiento.

**Rolling Update:** estrategia de actualización progresiva en Kubernetes que reemplaza las instancias de una aplicación por otras nuevas de manera controlada, según la configuración del despliegue.

**StatefulSet:** recurso de Kubernetes diseñado para gestionar aplicaciones con estado que requieren identidades estables y, cuando corresponde, almacenamiento persistente asociado a cada réplica.

**Tempo:** sistema de almacenamiento y consulta de trazas distribuidas integrado en el ecosistema de Grafana.

**Traza distribuida:** representación del recorrido de una solicitud a través de los distintos componentes de un sistema distribuido. Se compone de operaciones o spans relacionados que permiten analizar la secuencia y duración de las interacciones.

**YAML (YAML Ain't Markup Language):** formato de serialización de datos legible para las personas, utilizado para expresar configuraciones de aplicaciones y recursos de infraestructura.

## Resumen

**TÍTULO:** Estrategias de despliegue, monitoreo y observabilidad para la plataforma Smart Campus UIS. *

**AUTOR:** Cristian Manuel Hernández Delgado, Oscar Ivan Torrens Bustamante. **

**PALABRAS CLAVE:** IoT, Microservicios, Observabilidad, Monitoreo, Despliegue, Sistemas Distribuidos

**DESCRIPCIÓN:** Este trabajo presenta el diseño e implementación de una estrategia integrada de despliegue, monitoreo y observabilidad para la plataforma IoT Smart Campus UIS, con el objetivo de mejorar sus capacidades operacionales y facilitar la detección y el diagnóstico de fallos en un entorno distribuido. Se caracteriza el estado inicial de la plataforma y se identifican los requerimientos necesarios para formular una estrategia compatible con sus componentes y necesidades operativas.

A partir de este análisis, se establece una estrategia fundamentada en la revisión del estado del arte y en la evaluación comparativa de herramientas bajo criterios definidos. Asimismo, se adaptan los servicios de la plataforma para su despliegue en entornos orquestados, incorporando mecanismos de actualización controlada y monitoreo continuo. La estrategia seleccionada se implementa mediante la integración de herramientas para la recolección y visualización de métricas, logs y trazas distribuidas, con el propósito de facilitar el seguimiento del comportamiento de los servicios y el análisis de sus interacciones.

La estrategia se valida mediante escenarios de prueba que evalúan su comportamiento en condiciones operacionales y ante fallos controlados. Estas pruebas permiten examinar el funcionamiento de los mecanismos de despliegue y la capacidad de las herramientas integradas para presentar información útil sobre el estado de la plataforma. Los resultados evidencian una mejora en la visibilidad operacional del sistema y en la capacidad de analizar su comportamiento, lo que facilita la detección de fallos y el análisis de las interacciones entre sus componentes.

* Trabajo de investigación.

** Facultad de Ingenierías Fisicomecánicas. Escuela de Ingeniería de Sistemas e Informática. Director: Gabriel Rodrigo Pedraza Ferreira, Doctor en Ciencias de la Computación. Codirector: Henry Andres Jimenez Herrera, Magíster en Ingeniería de Sistemas e Informática.

**Abstract**

**TITLE:** Deployment, monitoring, and observability strategies for the Smart Campus UIS platform. *

**AUTHOR:** Cristian Manuel Hernández Delgado, Oscar Ivan Torrens Bustamante. **

**KEYWORDS:** IoT, Microservices, Observability, Monitoring, Deployment, Distributed Systems

**DESCRIPTION:** This work presents the design and implementation of an integrated deployment, monitoring, and observability strategy for the Smart Campus UIS IoT platform, with the aim of improving its operational capabilities and facilitating fault detection and diagnosis in a distributed environment. The platform's initial state is characterized, and the requirements needed to formulate a strategy compatible with its components and operational needs are identified.

Based on this analysis, a strategy is established through a review of the state of the art and a comparative evaluation of tools according to defined criteria. The platform services are also adapted for deployment in orchestrated environments, incorporating controlled update mechanisms and continuous monitoring. The selected strategy is implemented by integrating tools for collecting and visualizing metrics, logs, and distributed traces, with the purpose of facilitating service behavior monitoring and the analysis of interactions among services.

The strategy is validated through test scenarios that evaluate its behavior under operational conditions and in the presence of controlled failures. These tests examine the operation of deployment mechanisms and the ability of the integrated tools to present useful information about the platform's state. The results demonstrate improved operational visibility and an enhanced ability to analyze system behavior, thereby facilitating fault detection and the analysis of interactions among its components.

* Research work

** Faculty of Physics-Mechanics Engineering. School of Systems Engineering and Informatics. Advisor: Gabriel Rodrigo Pedraza Ferreira, Doctor in Computer Science. Co-advisor: Henry Andres Jimenez Herrera, Master in Systems Engineering and Informatics.

## Introducción

Los sistemas de Internet de las Cosas (IoT) han experimentado un crecimiento exponencial, consolidándose como una de las fuerzas transformadoras más significativas en la economía digital global. Para 2024, se estimaban aproximadamente 19 mil millones de dispositivos IoT conectados mundialmente, con proyecciones cercanas a 40 mil millones de dispositivos para 2030 (DemandSage, 2025; IoT Analytics, 2024). Este ecosistema tecnológico ha generado un impacto económico tangible, con McKinsey Global Institute estimando que el IoT podría generar entre $5.5 y $12.6 billones USD en valor económico global para 2030 (McKinsey & Company, 2024a).

La adopción masiva de tecnologías IoT ha sido especialmente notable en sectores como manufactura, donde se proyecta un potencial de generación de hasta $3.3 billones USD para 2030 (McKinsey & Company, 2024a). Sin embargo, esta expansión viene acompañada de desafíos técnicos significativos: el 75% de los proyectos IoT fallan en cumplir expectativas, mientras que el 80% fallan al intentar escalar más allá de las fases piloto (McKinsey & Company, 2024b).

Smart Campus UIS es una plataforma integral de IoT desarrollada en la Universidad Industrial de Santander, diseñada como una solución de propósito general para el monitoreo, análisis y gestión de datos en sistemas distribuidos inteligentes (Jiménez et al., 2020). Su arquitectura basada en microservicios permite la conexión eficiente de dispositivos IoT diversos, facilitando la recolección continua de datos desde sensores distribuidos en tiempo real. Esta característica de propósito general la posiciona como una plataforma versátil capaz de adaptarse a múltiples contextos y aplicaciones más allá

del ámbito universitario.

No obstante, Smart Campus UIS enfrenta los mismos desafíos técnicos que limitan el éxito de muchas implementaciones IoT contemporáneas. Si bien la plataforma utiliza tecnologías de contenedorización que proporciona cierto grado inherente de observabilidad, y cuenta con Grafana como herramienta de visualización, estos componentes no han sido integrados en un mecanismo cohesivo que pueda escalar apropiadamente con la arquitectura. Esta falta de integración y escalabilidad en los mecanismos de despliegue, monitoreo y observabilidad se vuelve crítica a medida que el sistema crece: sin la capacidad de observar el comportamiento de los componentes de manera integral, se dificulta la detección temprana de problemas, la identificación de cuellos de botella, y la planificación de capacidad.

Este trabajo de grado propone el diseño e implementación de una estrategia integrada que aborde estos desafíos operacionales, incorporando prácticas modernas de DevOps y observabilidad que buscan mejorar la resiliencia, mantenibilidad y capacidad de evolución de la plataforma Smart Campus UIS. La investigación aportará un caso de estudio sobre la adaptación e implementación de prácticas de la industria en despliegue, monitoreo y observabilidad para plataformas IoT, proporcionando evidencia cuantitativa de su impacto operacional y sirviendo como referencia para sistemas distribuidos con desafíos similares.

## 1. Planteamiento y justificación del problema

La implementación exitosa de plataformas IoT distribuidas enfrenta desafíos críticos

relacionados con la gestión operacional, especialmente en arquitecturas basadas en microservicios. La literatura especializada documenta que las arquitecturas de microservicios, aunque ofrecen escalabilidad y flexibilidad, introducen complejidades significativas como puntos de falla distribuidos, cuellos de botella de rendimiento y gestión inadecuada de recursos (Faseeha et al., 2025). Estos desafíos se intensifican en el contexto de sistemas IoT donde la confiabilidad operacional es fundamental para el éxito de la plataforma. Smart Campus UIS, como plataforma integral de IoT basada en microservicios, ejemplifica estos desafíos sistémicos, presentando tres problemáticas críticas asociadas al despliegue, monitoreo y observabilidad que limitan su capacidad operacional y potencial de adopción.

La primera problemática corresponde a estrategias de despliegue rudimentarias con impacto operacional significativo. La utilización exclusiva de tecnologías de contenedorización para el despliegue representa una limitación técnica que impacta directamente la eficiencia operacional, con tiempos de despliegue elevados que comprometen la disponibilidad del servicio durante las actualizaciones. Esta problemática es particularmente relevante considerando que las mejores prácticas en DevOps documentan estrategias de despliegue avanzadas como blue/green, canary y rolling deployments que minimizan o eliminan completamente el tiempo de inactividad durante actualizaciones (Rajković et al., 2022). La tecnología de contenedores, aunque útil para entornos de desarrollo y despliegues simples, presenta limitaciones inherentes para entornos de producción, ya que no replica de manera realista el entorno productivo donde típicamente se utilizan recursos de Kubernetes como Deployments o StatefulSets (Okteto, 2023). Esta discrepancia entre entornos de desarrollo y producción incrementa la probabilidad de errores difíciles de identificar y compromete la estabilidad del sistema.

La segunda problemática es que la infraestructura de monitoreo actual no aprovecha adecuadamente los beneficios arquitecturales del proyecto y de las tecnologías de contenedorización, lo cual genera una "ceguera operacional" que impide la detección proactiva de problemas. Esta situación

refleja uno de los desafíos más documentados en arquitecturas de microservicios: la dificultad inherente para obtener una visión clara de la salud, rendimiento y comportamiento general de la aplicación cuando múltiples servicios operan concurrentemente, creando una red compleja de procesos que vuelve obsoletas muchas herramientas tradicionales utilizadas en arquitecturas monolíticas (Cerbos, 2024). En el contexto específico de Smart Campus UIS, esta limitación se traduce en la necesidad de simular el envío de datos para verificar el funcionamiento básico del sistema, una práctica que indica la falta de herramientas automatizadas para evaluar el estado operacional de los microservicios. Sin un monitoreo integrado que escale con la arquitectura, los problemas solo se detectan cuando los usuarios finales reportan fallos, resultando en tiempos de detección y resolución prolongados que comprometen la experiencia del usuario y la confiabilidad del sistema.

La tercera problemática es la carencia de un marco integrado de observabilidad y sus implicaciones en sistemas distribuidos. Aunque la plataforma cuenta con Grafana, este no ha sido incorporado efectivamente en una estrategia integrada que aproveche los tres pilares fundamentales de observabilidad: métricas, logs y trazas distribuidas, esenciales para comprender el comportamiento de sistemas distribuidos complejos (Lumigo, 2022). Sin esta integración estratégica, la plataforma enfrenta lo que la literatura describe como "silos de datos" donde cada microservicio puede monitorearse de forma aislada, pero sin lograr una comprensión completa de cómo interactúan como un sistema integrado, dificultando la depuración y el entendimiento de la causa raíz de los problemas (Cerbos, 2024). Esta limitación es particularmente problemática en arquitecturas de microservicios donde una sola solicitud de usuario puede atravesar múltiples servicios, y la falta de trazabilidad distribuida impide la identificación de la causa raíz cuando ocurren fallos, resultando en lo que los especialistas denominan "ceguera de rastreo distribuido".

Las limitaciones operacionales identificadas han resultado en oportunidades perdidas de

expansión y adopción de Smart Campus UIS, patrón documentado en la literatura donde el 75% de los proyectos IoT fallan en cumplir expectativas y el 80% fallan al intentar escalar más allá de las fases piloto (McKinsey & Company, 2024b). Sin una infraestructura operacional robusta que provea métricas de rendimiento, detección proactiva de cuellos de botella y visibilidad integral del sistema, resulta imposible demostrar la confiabilidad necesaria para justificar su adopción en entornos críticos o planificar adecuadamente su escalamiento. Dado el carácter académico de la plataforma, los recursos disponibles para su despliegue varían considerablemente entre sus usuarios, por lo que la estrategia debe ser adaptable a diferentes escenarios de uso.

Pregunta de investigación: ¿Cómo implementar una estrategia integrada de despliegue, monitoreo y observabilidad para la plataforma Smart Campus UIS que sea adaptable a diferentes escenarios de uso?

# 2. Objetivos

## 2.1 Objetivo general

Implementar una estrategia de despliegue, monitoreo y observabilidad de los componentes de la plataforma Smart Campus UIS que ofrezca flexibilidad para adaptarse a la necesidad de los usuarios y a los recursos tecnológicos disponibles.

## 2.2 Objetivos específicos

Caracterizar el estado operacional actual de Smart Campus UIS para identificar los casos de uso de la plataforma, las limitaciones y oportunidades de mejora en su proceso actual de despliegue y monitoreo.

Establecer una estrategia integrada de despliegue, monitoreo y observabilidad para Smart Campus UIS, fundamentada en prácticas de la industria, que aborde las limitaciones identificadas, seleccione las herramientas apropiadas y permita realizar actualizaciones controladas con visibilidad operacional del sistema.

Adaptar los servicios de Smart Campus UIS para integrarlos con la estrategia establecida y las herramientas de despliegue, monitoreo y observabilidad seleccionadas.

Implementar la estrategia integrada de despliegue, monitoreo y observabilidad establecida,

configurando las herramientas y mecanismos necesarios para realizar actualizaciones controladas del sistema y obtener visibilidad operacional.

Validar experimentalmente la estrategia integrada de despliegue, monitoreo y observabilidad mediante la medición del desempeño de la estrategia utilizando un conjunto de métricas identificadas en la industria, y la evaluación de las capacidades de detección y diagnóstico mediante escenarios de fallo simulados.

# 3. Marco de referencia

La presente sección establece los fundamentos conceptuales y técnicos necesarios para comprender el ecosistema en el que opera Smart Campus UIS y los desafíos que este trabajo busca resolver.

## 3.1 Fundamentos Teóricos

### *3.1.1 Internet de las Cosas*

El Internet de las Cosas (IoT, por sus siglas en inglés) es un paradigma tecnológico que describe la interconexión de objetos físicos (sensores, actuadores, dispositivos embebidos) a través de redes de comunicación, permitiéndoles recolectar, intercambiar y actuar sobre datos del entorno sin intervención humana directa. El factor habilitador central de este paradigma es la convergencia de tecnologías de identificación, redes de sensores inalámbricos, protocolos de comunicación y

distribución de inteligencia en objetos cotidianos (Atzori et al., 2010).

Una plataforma IoT es la infraestructura de software que media entre los dispositivos físicos y las aplicaciones que consumen sus datos. Sus responsabilidades incluyen la gestión de conexiones de dispositivos heterogéneos, la recepción y enrutamiento de mensajes, el almacenamiento de datos según su tipo y temporalidad, y la exposición de interfaces para que aplicaciones externas consulten y actúen sobre la información recolectada. A medida que el número de dispositivos conectados crece (con proyecciones que superan los 40 mil millones para 2030 (DemandSage, 2025)), estas plataformas deben escalar tanto en capacidad de procesamiento como en capacidad operacional.

Las características propias del entorno IoT imponen restricciones particulares sobre las plataformas que lo soportan: los dispositivos son heterogéneos en capacidad de cómputo y protocolo de comunicación, los volúmenes de datos pueden ser elevados y continuos, y la distribución geográfica de los sensores exige que la infraestructura sea resiliente ante fallos parciales (Atzori et al., 2010). Estas restricciones hacen que las plataformas IoT de propósito general, como Smart Campus UIS (Jiménez et al., 2020), requieran decisiones arquitecturales específicas para mantener una operación confiable y sostenible a lo largo del tiempo.

### *3.1.2 Arquitecturas de Microservicios*

Los microservicios representan un paradigma arquitectónico que estructura aplicaciones como una colección de servicios débilmente acoplados, implementables de forma independiente y organizados alrededor de capacidades específicas de negocio (El Akhdar et al., 2024). Esta aproximación contrasta fundamentalmente con las arquitecturas monolíticas tradicionales, donde todas las funcionalidades están integradas en una sola unidad desplegable que debe ser construida, probada y desplegada como un

todo (El Akhdar et al., 2024).

La literatura académica reciente documenta que las arquitecturas de microservicios facilitan la construcción de sistemas altamente flexibles y escalables, superando las metodologías centralizadas complejas en términos de escalabilidad, mantenibilidad y extensibilidad (Ouyang et al., 2023). Un estudio sistemático que examinó 4.388 publicaciones entre 2010 y 2024 identificó doce beneficios principales de la adopción de microservicios en sistemas IoT, incluyendo mejoras en modularidad, escalabilidad independiente de servicios y facilidad de integración con dispositivos heterogéneos (El Akhdar et al., 2024).

Las características descentralizadas de los escenarios IoT se alinean de forma natural con esta arquitectura: los microservicios permiten que diferentes componentes evolucionen de forma independiente, que equipos distintos trabajen en paralelo, y que cada servicio utilice el lenguaje, base de datos o protocolo más adecuado para su función específica (Ferreira, 2023). Esta flexibilidad es particularmente valiosa en plataformas IoT de propósito general, donde la diversidad de tipos de sensores, protocolos y requisitos de procesamiento demanda componentes capaces de adaptarse sin afectar al resto del sistema (Ferreira, 2023).

Sin embargo, esta arquitectura introduce complejidades operacionales significativas. Cuando una solicitud de usuario puede atravesar múltiples servicios antes de completarse, la detección y diagnóstico de fallos se vuelve considerablemente más difícil que en un monolito (Faseeha et al., 2025). Cada servicio es una unidad de fallo independiente y, en ausencia de mecanismos que permitan observar el comportamiento del sistema en conjunto (no solo de cada servicio por separado), los problemas solo se detectan cuando los usuarios finales los reportan (Cerbos, 2024).

### *3.1.3 Contenedores y Orquestación*

Un contenedor es una unidad de software que empaqueta el código de una aplicación junto con todas sus dependencias (librerías, archivos de configuración, entorno de ejecución) de forma que pueda ejecutarse de manera uniforme en cualquier entorno que soporte la tecnología de contenedores, independientemente del sistema operativo subyacente. A diferencia de las máquinas virtuales, los contenedores comparten el kernel del sistema operativo anfitrión, lo que los hace significativamente más ligeros en consumo de recursos y más rápidos en tiempo de arranque (Docker Inc., 2024).

Docker es la plataforma de contenedores más ampliamente adoptada en la industria. Introduce el concepto de imagen: una plantilla inmutable y versionada que describe exactamente qué contiene el contenedor y cómo debe ejecutarse. Esta inmutabilidad resuelve el problema clásico del entorno inconsistente entre desarrollo y producción: si la imagen funciona en un entorno, funcionará igual en cualquier otro. En el contexto de microservicios, cada servicio se empaqueta como una imagen independiente, lo que permite desplegarlo, actualizarlo y escalarlo de forma individual sin afectar a los demás (Docker Inc., 2024).

A medida que el número de contenedores crece, gestionarlos manualmente se vuelve inviable. Un orquestador de contenedores es un sistema que automatiza el despliegue, la distribución, el escalamiento y la recuperación de contenedores sobre un conjunto de máquinas. Kubernetes es el orquestador estándar de la industria, originado en el sistema interno de Google denominado Borg (Verma et al., 2015), que gestionaba cientos de miles de trabajos sobre clústeres de decenas de miles de máquinas (Verma et al., 2015). Kubernetes heredó de Borg el principio de orquestación declarativa: el operador describe el estado deseado del sistema y el orquestador se encarga de alcanzarlo y mantenerlo de forma autónoma, independientemente de los fallos que ocurran (Verma et al., 2015).

Kubernetes organiza las aplicaciones en torno a tres abstracciones fundamentales. El pod es la unidad mínima de despliegue: un grupo de uno o más contenedores que comparten red y almacenamiento, y que Kubernetes trata como una unidad indivisible. El Deployment gestiona un conjunto de pods idénticos sin estado, controlando cuántas réplicas deben estar activas en todo momento y coordinando las actualizaciones entre ellas. El StatefulSet cumple una función análoga para componentes que requieren identidad estable y almacenamiento persistente, como bases de datos y brokers de mensajería, garantizando que cada instancia mantenga su estado entre reinicios (Kubernetes Authors, 2024b).

Uno de los mecanismos más importantes que Kubernetes implementa son los **health checks** (verificaciones de salud periódicas que el orquestador realiza a cada contenedor para determinar si está operando correctamente). Kubernetes define tres tipos: el **liveness probe** determina si el contenedor está vivo (si falla, el orquestador lo reinicia); el **readiness probe** determina si el contenedor está listo para recibir tráfico (si falla, el orquestador deja de enviarle solicitudes sin reiniciarlo); y el **startup probe** protege a contenedores con inicialización lenta de ser reiniciados prematuramente antes de que terminen de arrancar. Sin health checks, un orquestador no puede distinguir entre un servicio que está arrancando, uno degradado y uno que ha fallado completamente (Kubernetes Authors, 2024a).

Helm es el gestor de paquetes estándar para Kubernetes. Introduce el concepto de chart: un conjunto de plantillas y valores que describe todos los recursos necesarios para desplegar una aplicación (Deployments, StatefulSets, Services, ConfigMaps, Secrets) como una unidad coherente y versionada. Esto permite instalar, actualizar y revertir despliegues completos con un único comando, y parametrizar la configuración según el entorno sin modificar las plantillas base (Helm Authors, s.f.).

La combinación de contenedores y orquestación establece la base técnica sobre la que operan

las estrategias de despliegue modernas: al gestionar múltiples instancias de cada servicio, el orquestador puede reemplazarlas gradualmente durante una actualización, manteniendo el servicio disponible en todo momento (Mthembu et al., 2023). Este concepto se desarrolla como estrategia de despliegue en la sección 3.1.6 y se evalúa en detalle en la Fase 2 del presente trabajo.

### *3.1.4 Brokers de Mensajería y Protocolos IoT*

En un sistema IoT distribuido, los dispositivos físicos necesitan comunicar sus datos a la plataforma de forma asíncrona: el dispositivo publica un mensaje cuando tiene datos disponibles, sin requerir que haya un consumidor activo en ese preciso instante (Ferreira, 2023). Un **broker de mensajería** es el componente intermediario que recibe estos mensajes de los productores y los entrega a los consumidores, desacoplándolos temporalmente y garantizando que los mensajes no se pierdan aunque el consumidor no esté disponible en el momento de la publicación (Banks et al., 2019).

El patrón de comunicación dominante en los brokers IoT es **publicar/suscribir** (publish/subscribe): los productores publican mensajes en un canal temático denominado topic, y los consumidores se suscriben a los topics que les interesan. El broker enruta los mensajes del productor a todos los suscriptores del topic correspondiente, sin que productor y consumidor necesiten conocerse mutuamente ni estar activos simultáneamente (Banks et al., 2019).

En el ecosistema IoT dos protocolos de mensajería son especialmente relevantes por sus características complementarias. **MQTT** (Message Queuing Telemetry Transport) es un protocolo publicar/suscribir diseñado específicamente para dispositivos con recursos limitados y redes con ancho de banda reducido. Define tres niveles de calidad de servicio (QoS): entrega sin confirmación, entrega al menos una vez, y entrega exactamente una vez, permitiendo que el protocolo se adapte al balance

entre rendimiento y garantías de entrega que requiera cada caso de uso. Es estándar OASIS e ISO/IEC 20922 (Banks et al., 2019).

**AMQP** (Advanced Message Queuing Protocol) es un protocolo abierto de mensajería para entornos empresariales que define un protocolo binario de nivel de cable para el intercambio confiable de mensajes entre procesos. A diferencia de MQTT, AMQP está diseñado para escenarios que requieren garantías de entrega robustas, enrutamiento flexible, soporte de transacciones y alta disponibilidad. Es estándar OASIS e ISO/IEC 19464:2014 (Godfrey et al., 2012).

La coexistencia de ambos protocolos en una plataforma IoT como Smart Campus UIS no es redundante sino complementaria: MQTT atiende dispositivos físicos con recursos limitados, mientras que AMQP atiende integraciones de software donde la pérdida de mensajes es inaceptable (Banks et al., 2019; Godfrey et al., 2012). El broker que implementa cada protocolo es un componente crítico de la plataforma cuya disponibilidad y comportamiento deben ser monitoreados de forma continua (IoT For All, 2024).

### *3.1.5 Observabilidad en Sistemas Distribuidos*

El monitoreo tradicional se basa en la verificación de umbrales predefinidos: si una métrica supera un valor, se genera una alerta. Este enfoque funciona razonablemente bien en sistemas simples donde los modos de fallo son conocidos y anticipables, pero en arquitecturas de microservicios los problemas raramente se manifiestan como un único umbral superado (Cerbos, 2024). En cambio, emergen como combinaciones sutiles de comportamientos anómalos distribuidos entre servicios, que el monitoreo tradicional no puede correlacionar.

La **observabilidad** es la capacidad de comprender el estado interno de un sistema basándose

en sus salidas externas (China, 2025). A diferencia del monitoreo, que responde a la pregunta "¿está fallando el sistema?", la observabilidad responde a "¿por qué está fallando y dónde?". Este concepto trasciende la detección de problemas para incorporar capacidades de diagnóstico de causa raíz que permiten no solo detectar que algo está mal, sino entender exactamente qué ocurrió y cómo prevenirlo (Lumigo, 2022).

La observabilidad moderna se construye sobre tres pilares fundamentales que proporcionan perspectivas complementarias del comportamiento del sistema (Sharif, 2024; Tozzi, 2024). Las **métricas** son medidas cuantitativas agregadas en el tiempo que describen el rendimiento y utilización de recursos del sistema. Permiten establecer líneas base de comportamiento normal y detectar desviaciones mediante alertas automáticas (Sharif, 2024). Los **logs** son registros inmutables y con marca de tiempo de eventos discretos que ocurren en el sistema. Proporcionan contexto granular sobre el comportamiento del sistema, incluyendo información sobre errores, transacciones específicas y eventos importantes (Tozzi, 2024). Las **trazas distribuidas** registran el flujo completo de una solicitud a través de múltiples servicios, permitiendo reconstruir el camino exacto que siguió, identificar en qué servicio se introdujo la latencia o el error, y visualizar las dependencias entre componentes (Sridharan, 2018).

La clave de la observabilidad no está en cada pilar por separado sino en su correlación: una alerta de métricas señala que algo está mal, los logs de ese período revelan qué eventos ocurrieron, y las trazas muestran exactamente en qué servicio y operación se originó el problema (Sematext, 2025). La literatura académica enfatiza que aunque tener acceso a los tres pilares no garantiza automáticamente que los sistemas sean más observables, estas son herramientas que, cuando se comprenden adecuadamente, desbloquean la capacidad de construir y operar mejores sistemas (Sridharan, 2018).

En arquitecturas de microservicios IoT, la observabilidad se vuelve crítica precisamente porque una sola solicitud puede atravesar múltiples servicios geográficamente distribuidos antes de alcanzar su destino (Sematext, 2025). Sin trazas distribuidas, diagnosticar una degradación de rendimiento requiere revisar manualmente los logs de cada servicio por separado (un proceso que puede tomar horas y que se vuelve inviable a medida que el número de servicios crece) (Cerbos, 2024).

### *3.1.6 DevOps y Estrategias de Despliegue*

DevOps es un movimiento cultural y técnico que propone eliminar la separación tradicional entre los equipos de desarrollo (responsables de construir el software) y los equipos de operaciones (responsables de ejecutarlo en producción). En el modelo tradicional, estas dos funciones operan con objetivos en tensión: desarrollo quiere cambiar el software rápidamente, mientras que operaciones quiere mantenerlo estable. Esta tensión genera ciclos de entrega lentos, despliegues riesgosos y una cultura donde los fallos en producción son eventos traumáticos (Kim et al., 2016).

DevOps propone resolver esta tensión mediante tres principios fundamentales: el **flujo** (hacer que los cambios fluyan de desarrollo a producción de forma rápida y continua, reduciendo el tamaño de cada entrega para reducir el riesgo asociado a cada despliegue); la **retroalimentación** (establecer mecanismos que permitan detectar problemas lo antes posible en el ciclo de vida del software); y la **mejora continua** (crear una cultura donde los fallos se analizan sin culpabilizar y sus aprendizajes se incorporan sistemáticamente al proceso) (Kim et al., 2016).

En el contexto de plataformas IoT que evolucionan continuamente (incorporando nuevos tipos de dispositivos, casos de uso y funcionalidades), la capacidad de desplegar cambios de forma frecuente y confiable es un requisito operacional fundamental (IoT For All, 2024). Una plataforma que requiere

tiempo de inactividad para cada actualización compromete la continuidad del servicio para los dispositivos conectados y limita la velocidad a la que el equipo puede incorporar mejoras (Kim et al., 2016).

Una **estrategia de despliegue** es el mecanismo mediante el cual una nueva versión de un componente de software reemplaza a la versión anterior en un entorno de producción (Rajković et al., 2022). La elección de estrategia implica compromisos entre variables operacionales como: el tiempo de inactividad durante la transición, la velocidad de rollback ante un fallo, el consumo de recursos adicionales durante el proceso y la capacidad de validar la nueva versión con tráfico real antes de completar la transición (Rajković et al., 2022). En sistemas distribuidos con orquestadores de contenedores, estas estrategias pueden implementarse de forma automatizada, reduciendo el riesgo operacional de cada actualización (Mthembu et al., 2023). La evaluación comparativa de las estrategias disponibles para Smart Campus UIS se desarrolla en la Fase 2 del presente trabajo.

## 3.2 Contexto Tecnológico

Esta sección describe el ecosistema tecnológico en el que se inscribe Smart Campus UIS, con el objetivo de mostrar el panorama de plataformas, herramientas y desafíos operacionales que caracterizan los sistemas IoT distribuidos basados en microservicios.

### *3.2.1 Plataformas IoT Basadas en Microservicios*

La adopción de arquitecturas de microservicios en plataformas IoT ha sido documentada en la literatura como una tendencia consolidada que responde a las demandas de escalabilidad, flexibilidad y mantenibilidad en entornos distribuidos (Siddiqui et al., 2023). Sin embargo, esta adopción no elimina

los desafíos operacionales: los estudios indican que la mayoría de las implementaciones presentan debilidades en la gestión de la disponibilidad y en la capacidad de recuperarse ante fallos, aspectos que requieren atención específica (Siddiqui et al., 2023).

Las aplicaciones IoT basadas en microservicios típicamente se conectan a múltiples ubicaciones físicas (computadoras domésticas, dispositivos GPS, líneas de manufactura, sensores universitarios), requiriendo arquitecturas que soporten endpoints físicos heterogéneos, gateways API como puntos de entrada únicos, infraestructura de almacenamiento especializada según el tipo de dato, y brokers de mensajería para la comunicación asíncrona entre dispositivos y servicios (Ferreira, 2023). La implementación exitosa de estas arquitecturas demanda herramientas de monitoreo y observabilidad capaces de identificar fallos en cualquier punto de esta cadena (IoT For All, 2024).

### *3.2.2 Ecosistema de Herramientas de Observabilidad*

El ecosistema de herramientas de observabilidad para sistemas distribuidos ha madurado hasta ofrecer soluciones especializadas para cada uno de los tres pilares fundamentales (Cerbos, 2024). Comprender este ecosistema es necesario para entender el contexto en el que se enmarcan las decisiones tecnológicas evaluadas en la Fase 2 de este trabajo.

**Prometheus** se ha establecido como el estándar de facto para recolección y almacenamiento de métricas en sistemas distribuidos, ofreciendo un modelo de datos de series temporales y un lenguaje de consulta flexible (PromQL) (Cerbos, 2024). **Grafana** complementa Prometheus proporcionando capacidades de visualización interactiva y sistemas de alertas, y ha evolucionado hacia una plataforma unificada de observabilidad capaz de integrar métricas, logs y trazas en una sola interfaz (Cerbos, 2024).

Para la agregación y análisis de logs, el stack ELK (Elasticsearch, Logstash, Kibana) ha sido históricamente la solución más establecida en la industria, ofreciendo capacidades de indexación full-text sobre grandes volúmenes de datos (China, 2025). Alternativas más recientes como Loki adoptan un enfoque diferente: indexan únicamente etiquetas de metadatos en lugar del contenido completo, resultando en un consumo de almacenamiento significativamente menor a cambio de capacidades de búsqueda más acotadas (Cerbos, 2024).

En el ámbito de trazabilidad distribuida, herramientas como Jaeger y Zipkin implementan el modelo de trazas con diferentes compromisos en términos de almacenamiento e interfaz (China, 2025). Ambas son compatibles con **OpenTelemetry**, el framework unificador promovido por la CNCF que estandariza la recolección, procesamiento y exportación de datos de telemetría (métricas, logs y trazas), proporcionando APIs y SDKs consistentes para instrumentar aplicaciones independientemente del backend de observabilidad que se utilice (OpenTelemetry, 2025). OpenTelemetry es especialmente relevante en el contexto de este trabajo porque su agente para Java permite instrumentar aplicaciones Spring Boot de forma automática, sin modificar el código fuente de los servicios (OpenTelemetry, 2025).

### *3.2.3 Desafíos Operacionales en Implementaciones IoT Distribuidas*

Las plataformas IoT basadas en microservicios comparten un conjunto de desafíos operacionales que caracterizan el problema que este trabajo busca abordar. Las arquitecturas de microservicios introducen complejidades como puntos de falla distribuidos, cuellos de botella de rendimiento y gestión inadecuada de recursos (Faseeha et al., 2025), de las cuales tres son especialmente relevantes para Smart Campus UIS.

La **fragmentación de la observabilidad en silos** representa un problema fundamental: cuando cada microservicio se monitorea de forma aislada, se obtiene información útil sobre ese servicio específico, pero se pierde la capacidad de entender cómo los servicios interactúan como sistema integrado (Cerbos, 2024). Una degradación que atraviesa múltiples servicios es invisible cuando cada uno se monitorea por separado, lo que impide detectar el problema hasta que el usuario lo reporta (Faseeha et al., 2025).

La **complejidad creciente del monitoreo** escala proporcionalmente al número de servicios: las mismas herramientas que funcionan bien para monitorear un monolito o dos microservicios se vuelven inmanejables cuando el sistema tiene cinco o más servicios, cada uno con sus propias métricas, logs y dependencias (Cerbos, 2024). Sin una estrategia unificada de observabilidad, cada nuevo servicio incrementa la deuda operacional del sistema.

La **ausencia de estándares de instrumentación** agrava estos problemas: en sistemas donde cada componente expone métricas, formatea logs o propaga contexto de traza de forma independiente, es prácticamente imposible correlacionar información entre servicios de manera automática (Cerbos, 2024). La adopción de estándares como OpenTelemetry es una condición necesaria (aunque no suficiente) para superar este desafío (OpenTelemetry, 2025).

## 3.3 Revisión del estado del arte

Se realizó una revisión de la literatura técnica y académica sobre estrategias de despliegue, monitoreo y observabilidad en sistemas IoT distribuidos basados en microservicios, con el objetivo de establecer el contexto tecnológico en el que se enmarca el presente trabajo. La revisión combina fuentes académicas (IEEE, ACM, Springer) con fuentes técnicas de implementación práctica.

### *3.3.1 Fuentes Consultadas*

La revisión consultó fuentes académicas indexadas en IEEE Xplore, ACM Digital Library y Springer, complementadas con artículos técnicos de implementación práctica provenientes de blogs oficiales, documentación de herramientas y publicaciones de la comunidad DevOps. La selección de fuentes se rigió por tres criterios: similitud temática (relevancia directa para despliegue de microservicios, monitoreo y observabilidad en sistemas IoT distribuidos), contexto tecnológico (aplicabilidad al stack identificado en la Fase 1: Spring Boot, Docker, Kubernetes/K3s, EMQX) y actualidad (preferencia por publicaciones de los últimos cinco años, con excepciones justificadas cuando una fuente más antigua establece conceptos fundacionales vigentes). Las tablas 1 y 2 resumen las fuentes consideradas.

#### 3.3.1.1 Fuentes Académicas.

**Tabla 1**

*Fuentes académicas consultadas*

| # | Referencia | Aporte al trabajo |
|---|---|---|
| A1 | Nguyen et al. (2019). IEEE International Congress on Internet of Things. | Revisión sistemática de 17 estudios sobre despliegue y orquestación IoT. Identifica que la mayoría de enfoques se concentran en las capas edge y cloud, con escasa cobertura del nivel de dispositivo IoT. Establece una taxonomía de patrones declarativos vs. imperativos. Fundamenta la elección de orquestadores ligeros para entornos fog con recursos moderados. |

| # | Referencia | Aporte al trabajo |
|---|---|---|
| A2 | Mthembu et al. (2023). ICARTI 2023. | Revisión sistemática de herramientas de orquestación de contenedores para despliegue IoT en fog computing. Compara Kubernetes, Docker Swarm y Apache Mesos. También evalúa herramientas propietarias como Amazon ECS, Azure Kubernetes Service (AKS) y Google Kubernetes Engine (GKE), concluyendo que Kubernetes es la opción más robusta; K3s se destaca como alternativa ligera para nodos fog con recursos limitados. |
| A3 | Decimavilla-Alarcón y Marcillo-Franco (2025). | Revista Episteme & Praxis. Análisis de arquitecturas de microservicios basadas en contenedores para despliegue ágil de aplicaciones IoT en la nube. Identifica tres estrategias principales: microservicios ligeros, escalamiento elástico dinámico y aislamiento de servicios. Cuantifica mejoras del 40% en latencia y 55% en escalabilidad horizontal frente a arquitecturas monolíticas. Referencia herramientas de monitoreo distribuido como Prometheus, Grafana y Jaeger. |
| A4 | Giamattei et al. (2024). Journal of Systems and Software. | Revisión sistemática de 71 herramientas de monitoreo para microservicios y DevOps. Clasifica herramientas según métricas, trazas y logs. Identifica Prometheus, Jaeger y ELK Stack como referencias. Evalúa herramientas propietarias como Datadog, New Relic , Splunk, Dynatrace , AWS CloudWatch y Sumo Logic , concluyendo que el 73% de herramientas son open source y que las propietarias tienen mayor longevidad pero generan dependencia de proveedor. |

| # | Referencia | Aporte al trabajo |
|---|---|---|
| A5 | Costa et al. (2024). MobiQuitous 2023 (Springer). | Evaluación empírica de observabilidad en un entorno fog real usando herramientas open source (Prometheus, ElasticSearch, Jaeger, Grafana). Despliega los tres pilares (métricas, logs y trazas) en un caso de uso smart city con dispositivos Raspberry Pi. Demuestra que es viable lograr observabilidad en fog con overhead controlado (menos del 12% CPU en dispositivos IoT), ajustando configuraciones por defecto según las características de la aplicación. |
| A6 | Araujo et al. (2024). arXiv (preprint). | Preprint sobre observabilidad en fog computing. Define formalmente los tres pilares (métricas, logs y trazas) y su ciclo de vida de datos (ODLC). Presenta OpenTelemetry como estándar CNCF vendor-neutral para los tres dominios de instrumentación. Documenta el stack de herramientas open source (Prometheus, Fluentd, Jaeger, Grafana) y compara topologías centralizada, descentralizada y distribuida para sistemas fog. |

**3.3.1.2 Fuentes Técnicas.** Complementando las fuentes académicas, se analizaron 34 artículos técnicos de implementación práctica, de los cuales 21 fueron considerados aplicables al contexto de Smart Campus UIS, abarcando estrategias de despliegue, monitoreo, observabilidad e integración IoT. La tabla 2 resume las fuentes técnicas consideradas en esta revisión:

**Tabla 2**

*Fuentes técnicas consultadas*

| # | Fuente Técnica | Herramientas Identificadas |
|---|---|---|
| T1 | Mustufa, S. (2025). Spring Boot Observability Stack. Medium. | Prometheus, Loki, Tempo, Grafana, OpenTelemetry Java Agent, Alertmanager |
| T2 | Emon, N.K. (2025). Production-Ready Observability with PLG Stack. Medium. | Prometheus, Loki, Grafana Alloy, queries PromQL/LogQL |
| T3 | Zehri, M. (2025). Full Observability Stack with Grafana LGTM. Medium. | Loki, Grafana, Tempo, Mimir, Fluent Bit, OpenTelemetry Collector |
| T4 | EMQX Team. (2024). OpenTelemetry for MQTT and IoT. EMQX Blog. | EMQX + OpenTelemetry, métricas MQTT nativas |
| T5 | Bector, H. (2025). Distributed Tracing with Micrometer and OTel. Medium. | Micrometer Tracing, OpenTelemetry bridge, Spring Boot integration |
| T6 | Olawale, A. (2025). Monitoring with Prometheus and Grafana. Medium. | Prometheus exporters, Grafana dashboards, alert rules |
| T7 | Ayogun. (2024). Comparison of ELK vs Jaeger. Medium. | ELK Stack, Jaeger, Zipkin, análisis comparativo de trazas |
| T8 | Grafana Labs. (2024). LGTM Stack Documentation: Loki, Grafana, Tempo, Mimir. | Documentación oficial del stack LGTM: configuración, integración y datasources de Grafana |
| T9 | Rancher Labs. (2024). K3s Documentation: Lightweight Kubernetes. | Documentación oficial de K3s: instalación, arquitectura y compatibilidad con Kubernetes |
| T10 | OpenTelemetry Authors. (2024). OpenTelemetry Java Instrumentation Documentation. | Documentación oficial de instrumentación Java con OpenTelemetry para Spring Boot |

| # | Fuente Técnica | Herramientas Identificadas |
|---|---|---|
| T11 | Platform Engineers. (2025). Microservices Deployment Strategies. Medium. | Blue-Green, Canary, Rolling Updates en microservicios con Kubernetes |
| T12 | Lesmana, D. (2023). Microservices — Deployment Patterns. Medium. | Canary, Blue-Green, Dark Launching, A/B Testing en microservicios |
| T13 | Glushach, R. (2023). Building for Scalability: Microservice Deployment Patterns. Medium. | Patrones de despliegue por contenedor/VM; referencia a Amazon ECS y Datadog como alternativas comerciales |
| T14 | Doubletapp. (2023). Overview of Monitoring System with Prometheus and Grafana. Medium. | Prometheus exporters, Grafana dashboards, alertas y visualización de métricas |
| T15 | George, L. (2025). Building a Factory machine/sensor monitoring IoT pipeline. Medium. | Pipeline IoT industrial con sensores, Kafka, Prometheus, Grafana, InfluxDB |
| T16 | Hettiarachchi, Y. (2024). Distributed Request Tracing with OpenTelemetry. Medium. | OpenTelemetry Collector, Jaeger, Zipkin, Datadog, Honeycomb como backends de trazas |
| T17 | Savle, J. (2025). Distributed Tracing with Zipkin & OpenTelemetry. Medium. | Zipkin, OpenTelemetry SDK, trazas distribuidas en Spring Boot |
| T18 | Cloud Native Daily. (2023). 7 Best Tracing Tools for Microservices. Medium. | Jaeger, Zipkin, Datadog, New Relic, Splunk, Honeycomb — comparativa de herramientas de trazas |
| T19 | Ferreira, I. (2023). Microservices as a Solution for IoT Architecture Challenges. Medium. | Microservicios como solución a desafíos de escalabilidad, heterogeneidad y latencia en IoT |

| # | Fuente Técnica | Herramientas Identificadas |
|---|---|---|
| T20 | Javeri, P. (2019). Architectural Patterns For IoT — Why use Microservices? Medium. | Patrones arquitectónicos IoT: justificación de microservicios frente a monolitos para dispositivos IoT. Incluida como excepción al criterio de actualidad por establecer fundamentos conceptuales sobre arquitectura IoT que permanecen vigentes y no han sido superados por publicaciones más recientes. |
| T21 | Borgp, S. I. (2022). Exploring the Applications of IoT with Kubernetes. Medium. | Kubernetes para IoT: casos de uso, orquestación de dispositivos, despliegue en edge |
| T22 | Hartmann, V. (2025). Migrating from Promtail to Grafana Alloy in Kubernetes. Medium. | Deprecación de Promtail (EOL mar. 2026); Grafana Alloy como colector unificado open source compatible con OTel |

### *3.3.2 Estrategias de Despliegue*

La revisión identificó a Kubernetes como el orquestador de referencia en entornos distribuidos, ampliamente respaldado en fuentes académicas y técnicas (Mthembu et al., 2023; Nguyen et al., 2019), con K3s como alternativa preferida para nodos con recursos moderados en contextos edge y fog, y Docker Swarm como opción para equipos que priorizan simplicidad operacional (Borgp, 2022; Mthembu et al., 2023). En cuanto a patrones de despliegue, la literatura converge en tres estrategias principales para microservicios: Rolling Update, Blue-Green y Canary, cada una con diferentes compromisos entre complejidad de implementación, consumo de recursos, velocidad de rollback y capacidad de validación en producción (Decimavilla-Alarcón y Marcillo-Franco, 2025; Lesmana, 2023; Platform Engineers, 2025).

El Rolling Update reemplaza las instancias de la versión anterior de forma gradual, levantando nuevas instancias y eliminando las antiguas de manera incremental hasta completar la transición, manteniendo el servicio disponible en todo momento. Es el patrón de menor complejidad operativa y el único soportado de forma nativa por Kubernetes sin configuración adicional.

El despliegue Blue-Green mantiene dos entornos idénticos en producción: uno activo y uno en espera. La nueva versión se despliega en el entorno inactivo y, una vez validada, el tráfico se redirige completamente hacia él, permitiendo un rollback inmediato a costa de duplicar los recursos durante la transición.

El despliegue Canary introduce la nueva versión de forma progresiva, enrutando inicialmente un porcentaje reducido del tráfico real hacia ella y ampliándolo gradualmente a medida que se confirma su estabilidad, lo que permite validar el comportamiento en producción antes de completar la transición.

La evaluación comparativa detallada de estos patrones se desarrolla en la sección 6.

### *3.3.3 Monitoreo y Observabilidad*

La revisión reveló un consenso creciente en torno al stack LGTM (Prometheus, Loki, Tempo y Grafana) como la combinación más referenciada para unificar métricas, logs y trazas en implementaciones cloud-native y de microservicios (Emon, 2025; Mustufa, 2025; Zehri, 2025). OpenTelemetry emergió como el estándar vendor-neutral de la CNCF para instrumentación, con capacidad de auto-instrumentación en aplicaciones Java y Spring Boot y compatibilidad con múltiples backends, siendo adoptado junto a Prometheus por el 70% de las organizaciones a nivel global (Araujo et al., 2024; Grafana Labs, 2025). El marco de los 4 Golden Signals de Google SRE (latencia, tráfico,

errores y saturación) apareció como referente teórico recurrente para definir qué métricas observar en sistemas distribuidos (Emon, 2025; Olawale, 2025), y fuentes como George (2025) y Ferreira (2023) documentan pipelines IoT con sensores físicos que demuestran la viabilidad del stack open source en producción. Las integraciones concretas con el stack de Smart Campus UIS se detallan en la sección 6.

### *3.3.4 Síntesis de Hallazgos*

La revisión del estado del arte permite establecer las siguientes conclusiones en la tabla 3:

**Tabla 3**

*Síntesis de hallazgos, revisión estado del arte*

| # | Hallazgo |
|---|---|
| 1 | El stack LGTM (Prometheus + Loki + Tempo + Grafana) con OpenTelemetry representa el estado del arte en observabilidad para microservicios (Emon, 2025; Mustufa, 2025; Zehri, 2025). |
| 2 | OpenTelemetry se consolida como el estándar CNCF para instrumentación, ofreciendo vendor-neutrality y auto-instrumentación (Bector, 2025; EMQX Team, 2024; Mustufa, 2025). El 70% de organizaciones a nivel global combina Prometheus y OpenTelemetry en sus stacks de observabilidad (Grafana Labs, 2025). |
| 3 | La literatura identifica Rolling Update como el patrón de menor complejidad operativa, el de menor consumo de recursos adicionales durante el despliegue y el único que las implementaciones de Kubernetes soportan de forma nativa sin configuración adicional (Decimavilla-Alarcón y Marcillo-Franco, 2025; Lesmana, 2023; Platform Engineers, 2025). |
| 4 | La correlación de señales (métricas + logs + trazas) mediante traceId es fundamental para diagnóstico eficiente (Mustufa, 2025; Zehri, 2025) |

| # | Hallazgo |
|---|---|
| 5 | EMQX 5.2+ ofrece integración nativa con OpenTelemetry, simplificando el monitoreo del broker MQTT (EMQX Team, 2024). |

## 3.4 Herramientas Evaluadas

### *3.4.1 Plataformas de Orquestación*

K8s (Kubernetes): Plataforma de orquestación de contenedores líder en la industria, desarrollada originalmente por Google y actualmente mantenida por la CNCF. Ofrece auto-escalado, auto-recuperación, service discovery y un ecosistema extenso de herramientas, siendo identificado en la literatura como referente para entornos fog y la base sobre la que se desarrolló K3s (Mthembu et al., 2023).

K3s: Distribución ligera de Kubernetes certificada por CNCF, desarrollada por Rancher Labs, que con aproximadamente 512 MB RAM mantiene compatibilidad total con la API de Kubernetes. Reemplaza componentes pesados por alternativas ligeras (SQLite en lugar de etcd, Traefik como ingress por defecto) y hereda el ecosistema completo de extensiones, Helm charts y Operators de Kubernetes, lo que la posiciona como alternativa apropiada para entornos edge y fog computing en implementaciones IoT (Borgp, 2022; Mthembu et al., 2023).

Docker Swarm: Solución de orquestación nativa de Docker, con integración directa con Docker Compose y una curva de aprendizaje suave para equipos familiarizados con el ecosistema Docker (Mirantis, 2025b). Actualmente mantenido por Mirantis con soporte garantizado hasta al menos 2030, su ecosistema de integraciones es más acotado que el de Kubernetes, siendo esta última

la opción preferida cuando la integración con herramientas de observabilidad y la estandarización a largo plazo son prioridades (Mirantis, 2025a; Mthembu et al., 2023).

### *3.4.2 Herramientas de Métricas*

Prometheus: Sistema de monitoreo y alertas open source, proyecto graduado de CNCF, que implementa un modelo pull-based mediante scraping de endpoints con almacenamiento time-series local y lenguaje de consultas PromQL. Su integración nativa con Spring Boot se realiza mediante Micrometer, y cuenta con Prometheus Operator y el Helm chart oficial kube-prometheus-stack, que automatizan el descubrimiento de servicios en Kubernetes mediante CRDs como ServiceMonitor y PodMonitor (Prometheus Community, 2024), consolidándolo como estándar de facto para métricas en microservicios (Decimavilla-Alarcón y Marcillo-Franco, 2025; Emon, 2025; Mustufa, 2025).

InfluxDB: Base de datos time-series open source optimizada para métricas y eventos, que implementa un modelo push-based con lenguaje de consultas Flux/InfluxQL y retención configurable. Aunque Smart Campus UIS ya la utiliza para datos de sensores IoT, la versión 3.0 introduce una ruptura relevante al no incluir soporte nativo para Flux en su motor de consultas, con la capa de compatibilidad disponible presentando limitaciones de rendimiento documentadas (Decimavilla-Alarcón y Marcillo-Franco, 2025; InfluxData, 2023).

### *3.4.3 Herramientas de Logs*

Loki: Sistema de agregación de logs de Grafana Labs que, a diferencia de soluciones basadas en indexación full-text, indexa únicamente labels, lo que lo hace altamente eficiente en almacenamiento. Su lenguaje de consultas LogQL comparte sintaxis con PromQL, reduciendo la curva de aprendizaje

para equipos que ya trabajan con Prometheus, y ofrece integración nativa con Grafana para correlación log↔traza. Para la recolección de logs puede utilizarse Promtail o Grafana Alloy, este último recomendado como colector unificado desde 2025 (Grafana Labs, 2024; Hartmann, 2025; Mustufa, 2025).

ELK Stack: Solución de logging más establecida en la industria, compuesta por Elasticsearch para indexación full-text, Logstash para procesamiento y transformación, y Kibana para visualización. Su despliegue requiere un mínimo aproximado de 4 GB RAM solo para Elasticsearch, y su integración con herramientas de observabilidad basadas en el modelo de métricas de Prometheus requiere adaptadores no documentados oficialmente (Ayogun, 2024; Decimavilla-Alarcón y Marcillo-Franco, 2025).

### *3.4.4 Herramientas de Trazas Distribuidas*

Tempo: Backend de trazas distribuidas de Grafana Labs que utiliza object storage para un almacenamiento eficiente y es compatible con los principales protocolos de trazas: OpenTelemetry, Jaeger y Zipkin. Su integración nativa con Grafana permite la correlación automática de trazas con logs y métricas mediante derived fields, consolidándolo como componente clave del stack LGTM (Grafana Labs, 2024; Mustufa, 2025; Zehri, 2025).

Jaeger: Sistema de tracing distribuido desarrollado por Uber y proyecto graduado de CNCF, con amplia adopción en la industria y una interfaz propia para visualización de trazas. Su despliegue en producción requiere obligatoriamente un backend de almacenamiento externo (Elasticsearch, Cassandra u OpenSearch), ya que el almacenamiento en memoria no está previsto para entornos productivos, lo que introduce un componente adicional de infraestructura que incrementa la complejidad

operativa (Hettiarachchi, 2024; Jaeger Authors, 2024).

Zipkin: Sistema de tracing distribuido desarrollado por Twitter, precursor de muchas implementaciones modernas, con una interfaz simple y múltiples opciones de almacenamiento incluyendo in-memory, MySQL y Elasticsearch. A diferencia de Jaeger, no cuenta con el respaldo de una organización reconocida ni con un Operator oficial para Kubernetes, y su actividad de desarrollo está orientada principalmente al mantenimiento de estabilidad antes que a nuevas funcionalidades (Cloud Native Daily, 2023; Savle, 2025).

### *3.4.5 Herramientas de Visualización*

Grafana: Plataforma de visualización open source líder en la industria, compatible con más de 100 datasources incluyendo Prometheus, Loki y Tempo. Ofrece dashboards interactivos, sistema de alertas integrado y correlación de señales mediante derived fields, y cuenta con Grafana Operator y Helm chart oficial dentro del ecosistema CNCF, lo que permite incorporar nuevos datasources y nodos sin rediseñar la infraestructura (Emon, 2025; Grafana Labs, 2024; Mustufa, 2025).

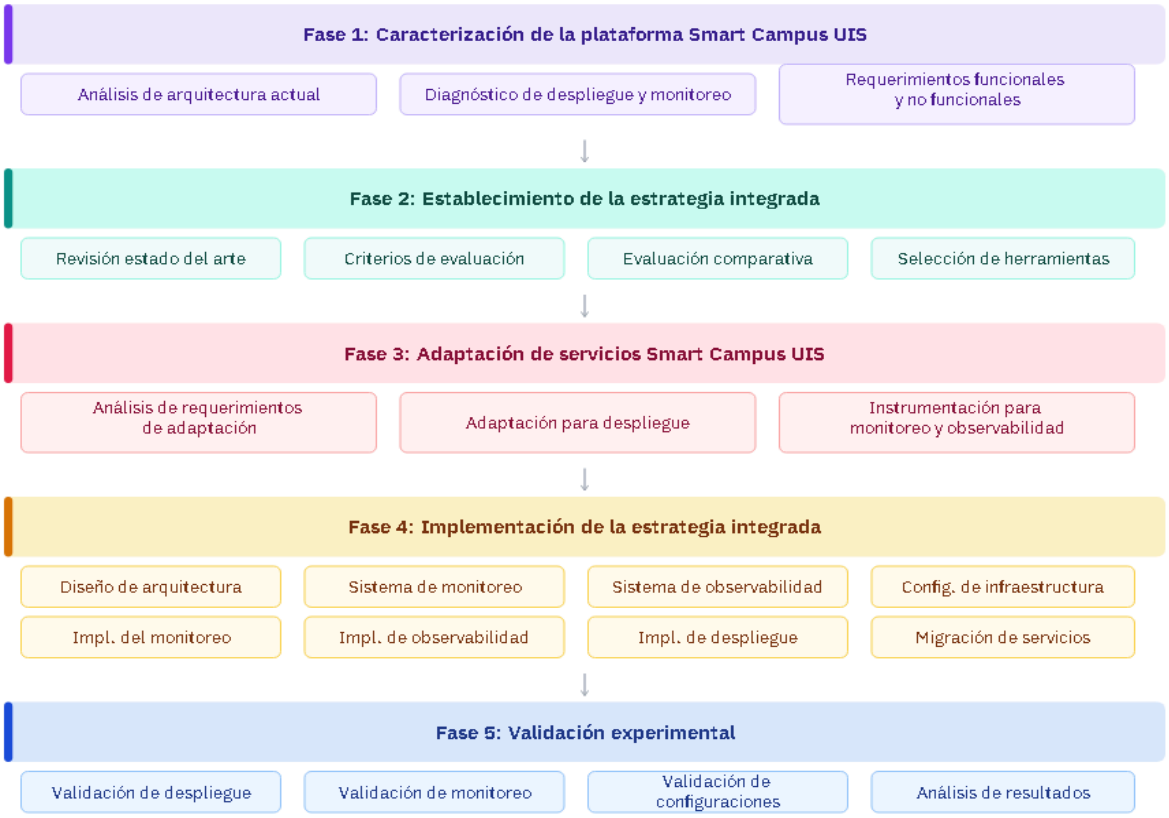
Kibana: Interfaz de visualización del stack ELK desarrollada por Elastic, con visualizaciones potentes y canvas para reportes personalizados. Requiere Elasticsearch como dependencia obligatoria, elevando el consumo total del par a un mínimo aproximado de 4 GB RAM, y fuera del ecosistema Elastic no dispone de integración nativa con herramientas basadas en el modelo de métricas de Prometheus (Ayogun, 2024).

# 4. Metodología

La metodología se estructura siguiendo un enfoque secuencial basado en el modelo en cascada, adaptado al contexto de investigación aplicada en ingeniería de software. Cada fase tiene como objetivo generar entregables que sirven de insumo para la siguiente, asegurando una transición ordenada desde el análisis conceptual hasta la validación de la solución. La figura 1 muestra un diagrama que ilustra esto.

**Figura 1**

*Metodología de investigación*




## 4.1 Fase 1: Caracterización de la Plataforma Smart Campus UIS

En esta fase se busca lograr un entendimiento profundo del funcionamiento interno de la aplicación Smart Campus UIS, identificando sus componentes arquitecturales, las interacciones entre microservicios, los casos de uso de la plataforma, y las limitaciones del sistema de despliegue, monitoreo y observabilidad actual. Se establecerán también los requerimientos funcionales y no funcionales que debe cumplir la solución a implementar.

### *4.1.1 Análisis de la arquitectura actual*

Se realizará un estudio detallado de la arquitectura de microservicios de Smart Campus UIS, documentando cada componente del sistema, su propósito específico y las tecnologías utilizadas. Se identificarán las dependencias entre servicios, bases de datos, sistemas de mensajería y APIs expuestas. Este análisis incluirá la revisión del código fuente, archivos de configuración y documentación técnica existente.

### *4.1.2 Diagnóstico del sistema de despliegue, monitoreo y observabilidad actual*

Se evaluará críticamente el proceso de despliegue y las capacidades de monitoreo y observabilidad actuales del sistema, identificando sus limitaciones operacionales y oportunidades de mejora. Se documentarán aspectos relevantes del proceso actual, como tiempos de despliegue, procedimientos de actualización, mecanismos de recuperación disponibles, herramientas de monitoreo existentes y capacidades de observabilidad disponibles. Si existen registros de incidentes previos, estos serán analizados para identificar patrones de problemas recurrentes y áreas que requieren atención.

### *4.1.3 Mapeo de dependencias e identificación de componentes críticos*

Se mapearán las interacciones y dependencias funcionales entre componentes del sistema, documentando cómo se relacionan los servicios para soportar diferentes funcionalidades de la plataforma. Se identificarán los casos de uso de Smart Campus UIS y los conjuntos de servicios necesarios para cada caso de uso. Se identificarán también los servicios cuya disponibilidad es fundamental para el funcionamiento del sistema, aquellos cuyo comportamiento impacta

significativamente el desempeño general de la plataforma, y posibles cuellos de botella que deban ser considerados al momento de establecer estrategias de despliegue, monitoreo y observabilidad.

### *4.1.4 Establecimiento de requerimientos funcionales*

Con base en las limitaciones identificadas en el diagnóstico, las características de la arquitectura actual y los casos de uso identificados, se definirán los requerimientos funcionales que debe cumplir la solución de despliegue, monitoreo y observabilidad a implementar. Estos requerimientos especificarán las capacidades operacionales necesarias para abordar las limitaciones identificadas y soportar adecuadamente los diferentes casos de uso de la plataforma Smart Campus UIS.

### *4.1.5 Especificación de requerimientos no funcionales*

Se especificarán los requerimientos no funcionales que deberá cumplir la solución implementada, derivados del análisis de la arquitectura, los casos de uso y los componentes críticos identificados. Estos requerimientos definirán los atributos de calidad esperados del sistema de despliegue, monitoreo y observabilidad. Estos requerimientos servirán como criterios de validación en la fase final del proyecto.

## 4.2 Fase 2: Establecimiento de Estrategia Integrada

En esta fase se busca establecer una estrategia integrada de despliegue, monitoreo y observabilidad fundamentada en prácticas de la industria, que aborde las limitaciones identificadas en la fase anterior. Se realizará un estudio comparativo de las prácticas y herramientas disponibles, y se

seleccionarán las más adecuadas para el contexto de Smart Campus UIS. Las actividades propuestas para esta fase son: Revisión del estado del arte, Establecimiento de criterios de evaluación, Identificación de integraciones tecnológicas, Evaluación comparativa de alternativas, Selección justificada de estrategias y herramientas.

### *4.2.1 Revisión del estado del arte*

Se llevará a cabo una revisión sistemática de literatura técnica y académica sobre estrategias de despliegue, estrategias de monitoreo y herramientas de monitoreo y observabilidad, documentando sus características, ventajas, desventajas y aplicabilidad en sistemas IoT distribuidos. Se priorizarán fuentes actualizadas, documentación oficial y casos de estudio relevantes.

### *4.2.2 Establecimiento de criterios de evaluación*

Con base en las prácticas documentadas, los requerimientos establecidos en la fase de caracterización y los casos de uso identificados, se definirá un conjunto de criterios ponderados que servirán para evaluar las alternativas identificadas. Estos criterios permitirán evaluar las estrategias de despliegue, las estrategias de monitoreo y las herramientas de monitoreo y observabilidad, considerando su capacidad para abordar las limitaciones identificadas, soportar los diferentes casos de uso, permitir actualizaciones controladas y proporcionar visibilidad operacional del sistema.

### *4.2.3 Identificación de integraciones tecnológicas*

Se identificarán las combinaciones de herramientas que son compatibles y funcionan de manera

integrada, documentando qué conjuntos de herramientas de monitoreo y observabilidad se integran efectivamente para soportar las estrategias de despliegue identificadas en plataformas IoT distribuidas.

#### *4.2.4 Evaluación comparativa de alternativas*

Se realizará una evaluación sistemática de las alternativas documentadas mediante una matriz de decisión basada en los criterios establecidos previamente. Esta evaluación considerará las estrategias de despliegue, estrategias de monitoreo, herramientas de monitoreo y herramientas de observabilidad identificadas durante la revisión del estado del arte.

#### *4.2.5 Selección justificada de estrategias y herramientas*

Con base en la evaluación comparativa y los requerimientos establecidos en la fase de caracterización, se seleccionará una estrategia de despliegue, una estrategia de monitoreo y una estrategia de observabilidad, junto con las herramientas que las soportan. Se definirán las diferentes configuraciones de la estrategia integrada que permitirán desplegar y monitorear diferentes conjuntos de servicios según los casos de uso y necesidades operacionales identificadas. Cada decisión será documentada con su respectiva justificación técnica, estableciendo la estrategia integrada que se implementará en las fases posteriores.

### 4.3 Fase 3: Adaptación de Servicios Smart Campus UIS

En esta fase se realizarán las adaptaciones necesarias en los servicios de Smart Campus UIS para prepararlos para su integración con la estrategia establecida y las herramientas seleccionadas. Se

modificarán configuraciones, se agregarán capacidades de instrumentación y se prepararán los servicios para su despliegue en la nueva infraestructura. Las actividades propuestas para esta fase son: Análisis de requerimientos de adaptación, Adaptación de servicios para estrategia de despliegue, Instrumentación de servicios para monitoreo y observabilidad.

### *4.3.1 Análisis de requerimientos de adaptación*

Se identificarán las modificaciones necesarias en cada servicio de Smart Campus UIS para integrarlo con las herramientas y estrategias seleccionadas. Se documentarán los cambios requeridos en configuraciones, dependencias, capacidades de instrumentación para monitoreo y observabilidad, y adaptaciones necesarias para soportar la estrategia de despliegue seleccionada.

### *4.3.2 Adaptación de servicios para estrategia de despliegue*

Se realizarán las modificaciones necesarias en los servicios para hacerlos compatibles con la estrategia de despliegue seleccionada. Se implementarán los endpoints de health check y readiness necesarios, se prepararán los archivos de configuración y variables de entorno requeridos, y se ajustarán las dependencias para el despliegue en la nueva infraestructura.

### *4.3.3 Instrumentación de servicios para monitoreo y observabilidad*

Se agregarán las capacidades de instrumentación necesarias en los servicios para habilitar la recolección de métricas, trazas y logs según las herramientas de monitoreo y observabilidad seleccionadas. Se implementarán los mecanismos de exposición de información requeridos por las

herramientas seleccionadas, asegurando que los servicios proporcionen la visibilidad operacional necesaria.

## 4.4 Fase 4: Implementación de Estrategia Integrada

En esta fase se implementará la estrategia integrada de despliegue, monitoreo y observabilidad establecida en la segunda fase, configurando las herramientas y mecanismos seleccionados. Se preparará la infraestructura necesaria, se configurarán los sistemas de monitoreo y observabilidad, y se implementarán los mecanismos de despliegue que permitirán realizar actualizaciones controladas con visibilidad operacional. Las actividades propuestas para esta fase son: Diseño de la arquitectura de despliegue, Diseño del sistema de monitoreo, Diseño del sistema de observabilidad, Configuración de la infraestructura, Implementación del sistema de monitoreo, Implementación del sistema de observabilidad, Implementación de la estrategia de despliegue, Migración de servicios.

### *4.4.1 Diseño de la arquitectura de despliegue*

Se diseñará la arquitectura de despliegue definiendo cómo se organizarán y desplegarán los servicios de Smart Campus UIS según la estrategia seleccionada y las diferentes configuraciones establecidas. Se especificarán los componentes necesarios, su configuración y los procedimientos para realizar actualizaciones del sistema para cada configuración definida.

### *4.4.2 Diseño del sistema de monitoreo*

Se diseñará el sistema de monitoreo especificando qué métricas serán recolectadas, cómo serán

procesadas y qué alertas se generarán. Se definirán los componentes del sistema de monitoreo y su integración con los servicios de Smart Campus UIS.

### *4.4.3 Diseño del sistema de observabilidad*

Se diseñará el sistema de observabilidad especificando qué información de trazas y logs será recolectada, cómo será procesada y visualizada. Se definirán los mecanismos de visualización y diagnóstico que facilitarán la comprensión del comportamiento del sistema.

### *4.4.4 Configuración de la infraestructura*

Se configurará la infraestructura base necesaria para soportar la estrategia integrada, preparando los recursos requeridos según las especificaciones del diseño y las herramientas seleccionadas en la fase de establecimiento de estrategia.

### *4.4.5 Implementación del sistema de monitoreo*

Se configurarán e implementarán las herramientas de monitoreo seleccionadas, estableciendo los mecanismos de recolección de métricas desde los servicios de Smart Campus UIS. Se definirán e implementarán las métricas clave, alertas y umbrales que permitan detectar problemas operacionales.

### *4.4.6 Implementación del sistema de observabilidad*

Se configurarán e implementarán las herramientas de observabilidad seleccionadas, estableciendo los mecanismos de recolección de trazas y logs desde los servicios. Se implementarán los

mecanismos de visualización y diagnóstico que faciliten la comprensión del comportamiento del sistema.

### *4.4.7 Implementación de la estrategia de despliegue*

Se implementará la estrategia de despliegue seleccionada, configurando los mecanismos y procedimientos necesarios para realizar actualizaciones del sistema según lo especificado en la estrategia. Se prepararán los procedimientos operacionales para la ejecución de despliegues.

### *4.4.8 Migración de servicios*

Se realizará la migración de los servicios adaptados de Smart Campus UIS hacia la nueva infraestructura de manera gradual y controlada, aplicando la estrategia de despliegue implementada. La migración se ejecutará por conjuntos de servicios según las configuraciones definidas en la estrategia integrada, permitiendo validar cada configuración antes de proceder con la siguiente.

## 4.5 Fase 5: Validación Experimental

En esta fase se validará que la solución implementada cumple con los objetivos y requerimientos establecidos, mediante pruebas experimentales y evaluación de los resultados obtenidos. Se validarán las diferentes configuraciones de la estrategia integrada, se medirán métricas operacionales del sistema, y se evaluarán las capacidades de los sistemas de monitoreo y observabilidad. Las actividades propuestas para esta fase son: Validación de estrategias de despliegue, Validación de sistemas de monitoreo y observabilidad, Validación de configuraciones, Análisis de resultados y trabajo futuro.

### *4.5.1 Validación de estrategias de despliegue*

Se ejecutarán pruebas para validar el correcto funcionamiento de la estrategia de despliegue implementada, midiendo métricas operacionales como tiempo de despliegue y tiempo de inactividad durante actualizaciones. Se validará el funcionamiento de los mecanismos definidos en la estrategia de despliegue, verificando que permiten realizar actualizaciones controladas del sistema.

### *4.5.2 Validación de sistemas de monitoreo y observabilidad*

Se evaluará la efectividad de los sistemas de monitoreo y observabilidad mediante escenarios de fallo simulados, verificando su capacidad para detectar y diagnosticar problemas operacionales. Se validará la correcta generación de alertas, la precisión de la información recolectada y la utilidad de los mecanismos de visualización y diagnóstico implementados.

### *4.5.3 Validación de configuraciones*

Se validarán las diferentes configuraciones de la estrategia integrada establecidas según los casos de uso identificados. Se verificará que cada configuración cumple con los requerimientos funcionales y no funcionales correspondientes, y que permite desplegar y monitorear adecuadamente los conjuntos de servicios definidos para cada caso de uso.

### *4.5.4 Análisis de resultados y trabajo futuro*

Se analizarán los resultados obtenidos de las pruebas de validación, comparando el desempeño

de la estrategia integrada implementada respecto al sistema anterior. Se identificarán oportunidades de mejora y posibles trabajos futuros relacionados con la plataforma Smart Campus UIS.

# 5. Caracterización de la plataforma Smart Campus UIS

Esta fase documenta el funcionamiento interno de Smart Campus UIS, identificando sus componentes arquitecturales, las interacciones entre microservicios, los casos de uso de la plataforma y las limitaciones del sistema de despliegue, monitoreo y observabilidad actual. A partir de este análisis, se establecen los requerimientos funcionales y no funcionales que debe cumplir la solución implementada. La fase se estructura en cinco secciones: análisis de la arquitectura actual, diagnóstico del sistema de despliegue, monitoreo y observabilidad, mapeo de dependencias e identificación de componentes críticos, requerimientos funcionales y requerimientos no funcionales.

La caracterización se realizó mediante revisión exhaustiva del código fuente de cada componente, sus archivos de configuración y los manifiestos de despliegue Docker Compose disponibles en el repositorio del proyecto. El microservicio de datos constituye una excepción parcial: su arquitectura interna se verificó contra el código fuente actual y se complementó con el trabajo de grado de Lozada Garavito y Cañon Castillo (2020), quienes documentaron el diseño original de ese módulo en el contexto del semillero CAGE.

## 5.1 Análisis de la arquitectura actual

Esta sección caracteriza la arquitectura de Smart Campus UIS a partir del análisis directo de su

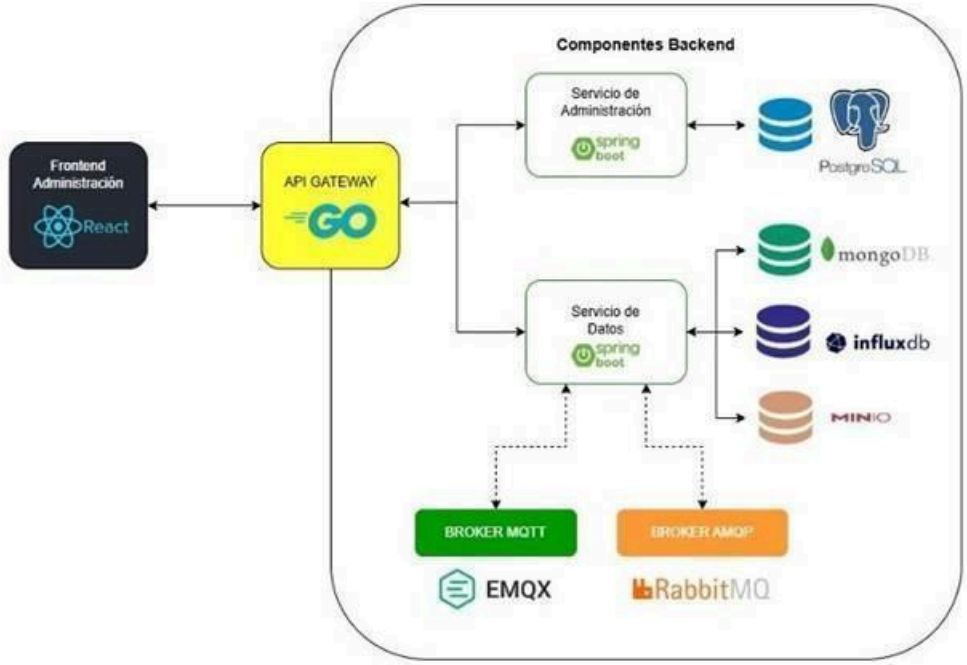
código fuente y configuración. Se describe cada componente del sistema, su propósito, las tecnologías empleadas y las dependencias entre servicios, bases de datos, sistemas de mensajería y APIs expuestas.

Smart Campus UIS está compuesto por seis componentes principales: un frontend de administración, un API Gateway, un microservicio de administración con base de datos PostgreSQL, y un microservicio de datos con tres sistemas de almacenamiento especializados (MongoDB, InfluxDB y MinIO). La comunicación asíncrona se gestiona a través de dos brokers de mensajería: EMQX (MQTT) y RabbitMQ (AMQP).

El siguiente diagrama (figura 2), tomado del trabajo de grado de Lozada Garavito y Cañon Castillo (2020) y verificado como fiel al estado actual de la plataforma, ilustra las relaciones entre los componentes de Smart Campus UIS: microservicios, bases de datos, brokers de mensajería y frontend. Estos componentes serán descritos de manera individual en esta sección.

**Figura 2**

*Diagrama de arquitectura de smart-campus uis*




*Nota.* Tomado de *Extensión del módulo de gestión de datos de la plataforma Smart Campus UIS para el soporte de diversos protocolos y almacenamiento de datos*, por J. C. Lozada Garavito y J. J. Cañon Castillo, 2020, Universidad Industrial de Santander (https://noesis.uis.edu.co/).

### *5.1.1 Frontend de Administración*

El frontend de administración es la interfaz visual de la plataforma, desarrollada en React 18 con React Router DOM para la navegación y Bootstrap como biblioteca de estilos. Permite a los administradores gestionar aplicaciones, dispositivos, modelos de servicio y recursos del sistema a través

de un navegador web, y es servida mediante Nginx en un contenedor Docker. Delega toda comunicación con el backend al API Gateway.

### *5.1.2 API Gateway*

El API Gateway actúa como punto de entrada único para todas las solicitudes provenientes del frontend, implementado en Go utilizando la biblioteca estándar net/http. Aplica un patrón de proxy inverso que enruta cada petición hacia el microservicio correspondiente según el prefijo de la ruta, sin lógica de negocio propia, exponiendo únicamente dos rutas hacia los microservicios internos.

### *5.1.3 Microservicio de Administración*

El microservicio de administración gestiona la configuración y metadatos del sistema IoT, incluyendo el registro de aplicaciones, dispositivos, recursos y modelos de servicio. Desarrollado en Java 11 con Spring Boot y Spring Data JPA, expone una API REST para operaciones CRUD sobre estas entidades y utiliza PostgreSQL como única base de datos persistente. Los cambios sobre dispositivos se propagan hacia la infraestructura IoT mediante llamadas REST.

### *5.1.4 Microservicio de Datos*

El microservicio de datos gestiona la recepción, almacenamiento y consulta de datos generados por dispositivos IoT. Su caracterización se realizó contrastando el código fuente actual con la documentación del trabajo de grado que dio origen a este componente (Lozada Garavito y Cañon Castillo, 2020). Implementa una arquitectura hexagonal con módulos separados para aplicación,

dominio, servicio y persistencia. Se integra con múltiples sistemas de almacenamiento según el tipo de dato: MongoDB para mensajes con esquema flexible, InfluxDB para series temporales de sensores y MinIO para archivos binarios como grabaciones de vídeo. La comunicación con dispositivos se gestiona a través de MQTT mediante EMQX y AMQP mediante RabbitMQ, y expone controladores REST para consultas de series temporales, gestión de documentos, manejo de archivos y gestión de cámaras.

### *5.1.5 Sistemas de Almacenamiento*

El sistema de almacenamiento fue seleccionado según el tipo de dato que persiste: metadatos relacionales con PostgreSQL (PostgreSQL Global Development Group, 2024), mensajes con esquema flexible con MongoDB (MongoDB, Inc., 2024), series temporales de sensores con InfluxDB (InfluxData, 2024) y archivos binarios con MinIO (MinIO, Inc., 2024). PostgreSQL es utilizado exclusivamente por el microservicio de administración, mientras que los tres restantes son gestionados por el microservicio de datos.

### *5.1.6 Brokers de Mensajería*

La comunicación asíncrona entre dispositivos IoT y microservicios se gestiona a través de dos brokers complementarios: EMQX (EMQ Technologies, 2024) recibe los mensajes publicados por dispositivos IoT mediante MQTT, mientras que RabbitMQ (VMware Tanzu, 2024) gestiona la comunicación entre servicios internos mediante AMQP, donde se requieren garantías de entrega robustas.

### *5.1.7 Flujo de Datos de Mensajería*

Los mensajes generados por dispositivos IoT siguen rutas definidas a través de los brokers hasta su persistencia en las bases de datos. A continuación se describen los dos flujos principales según el protocolo de mensajería utilizado.

#### 5.1.7.1 Flujo MQTT (Dispositivos IoT).

- Dispositivos IoT publican mensajes en el topic: device/messages
- EMQX recibe y distribuye los mensajes a los suscriptores
- EmqxListener (Data Microservice) recibe el mensaje
- El mensaje se deserializa a objeto Message
- Se persiste simultáneamente en MongoDB e InfluxDB

La persistencia dual en MongoDB e InfluxDB garantiza que cada mensaje quede disponible tanto para consultas por entidad y rango de tiempo (MongoDB) como para análisis de series temporales y agregaciones (InfluxDB), sin que el dispositivo emisor deba conocer la topología de almacenamiento.

#### 5.1.7.2 Flujo AMQP (Mensajería Empresarial).

- Productores publican mensajes en RabbitMQ
- RabbitListener consume mensajes de cola anónima
- El mensaje se procesa y persiste en los repositorios
- Soporta confirmación de mensajes y reintento

A diferencia del flujo MQTT, el flujo AMQP ofrece garantías de entrega mediante confirmación de mensajes (acknowledgment) y soporte de reintento, lo que lo hace adecuado para integraciones

empresariales donde la pérdida de mensajes es inaceptable.

### *5.1.8 Casos de Uso*

La tabla 4 consolida los principales casos de uso que la plataforma Smart Campus UIS soporta, especificando para cada uno el microservicio responsable, el endpoint involucrado, el método de comunicación, las bases de datos y brokers utilizados, y los componentes externos que participan. El catálogo completo de casos de uso, incluyendo todas las variantes de cada operación, se encuentra disponible en el Apéndice A.

**Tabla 4**

*Casos de uso relevantes*

| Caso de uso | Microservicio | Método | Endpoint | Base de datos / Brokers | Otros componentes |
|---|---|---|---|---|---|
| Listar dispositivos | Admin | GET | /device | PostgreSQL | - |
| Crear dispositivo con sincronización | Admin | POST | /device | PostgreSQL, Redis | Gateway IoT (REST) |
| Eliminar dispositivo con sincronización | Admin | DELETE | /device/{id} | PostgreSQL, Redis | Gateway IoT (REST) |

| Caso de uso | Microservicio | Método | Endpoint | Base de datos / Brokers | Otros componentes |
|---|---|---|---|---|---|
| Recibir mensajes de sensores vía MQTT | Data | MQTT Subscribe | device/message | InfluxDB, MongoDB | EMQX, Gateway IoT |
| Recibir mensajes de sensores vía AMQP | Data | AMQP Consume | device-message | InfluxDB, MongoDB | RabbitMQ, Gateway IoT |
| Re-encolar mensaje a subtópico | Data | MQTT/AMQP Publish | {subtopic} | InfluxDB, MongoDB | EMQX, RabbitMQ |
| Obtener últimas N mediciones de métrica | Data | GET | /influx/measurement/{m}/last | InfluxDB | - |
| Obtener mediciones por rango temporal | Data | GET | /influx/by-time-range | InfluxDB | - |
| Obtener promedio de métrica en rango | Data | GET | /influx/measurement/{m}/average | InfluxDB | - |
| Obtener valor mínimo y máximo de métrica | Data | GET | /influx/measurement/{m}/min\|max | InfluxDB | - |

| Caso de uso | Microservicio | Método | Endpoint | Base de datos / Brokers | Otros componentes |
|---|---|---|---|---|---|
| Obtener mensajes por dispositivo | Data | GET | /mongo/deviceId/{deviceId} | MongoDB | - |
| Obtener mensajes por ubicación | Data | GET | /mongo/location/{location} | MongoDB | - |
| Obtener mensajes por rango de fechas | Data | GET | /mongo/by-time-range | MongoDB | - |
| Registrar nueva cámara | Data | POST | /camera/add | MongoDB | - |
| Iniciar grabación de cámara | Data | GET | /camera/start?IdCamera={id} | MinIO, MongoDB | Cámara RTSP |
| Iniciar transmisión en vivo | Data | GET | /camera/stream?IdCamera={id} | MinIO | Cámara RTSP |
| Enrutamiento de peticiones HTTP | Gateway | Proxy HTTP | /admin/* \| /data/* | - | Admin MS, Data MS |

El análisis conjunto de los casos de uso revela que el microservicio de datos concentra la mayor diversidad de dependencias de infraestructura, siendo el único componente que puede involucrar simultáneamente hasta tres sistemas de almacenamiento y dos brokers en un mismo flujo. Esta característica lo convierte en el componente con mayor superficie de fallo de la plataforma. El API Gateway, aunque el más simple estructuralmente, representa el punto único de fallo desde la perspectiva

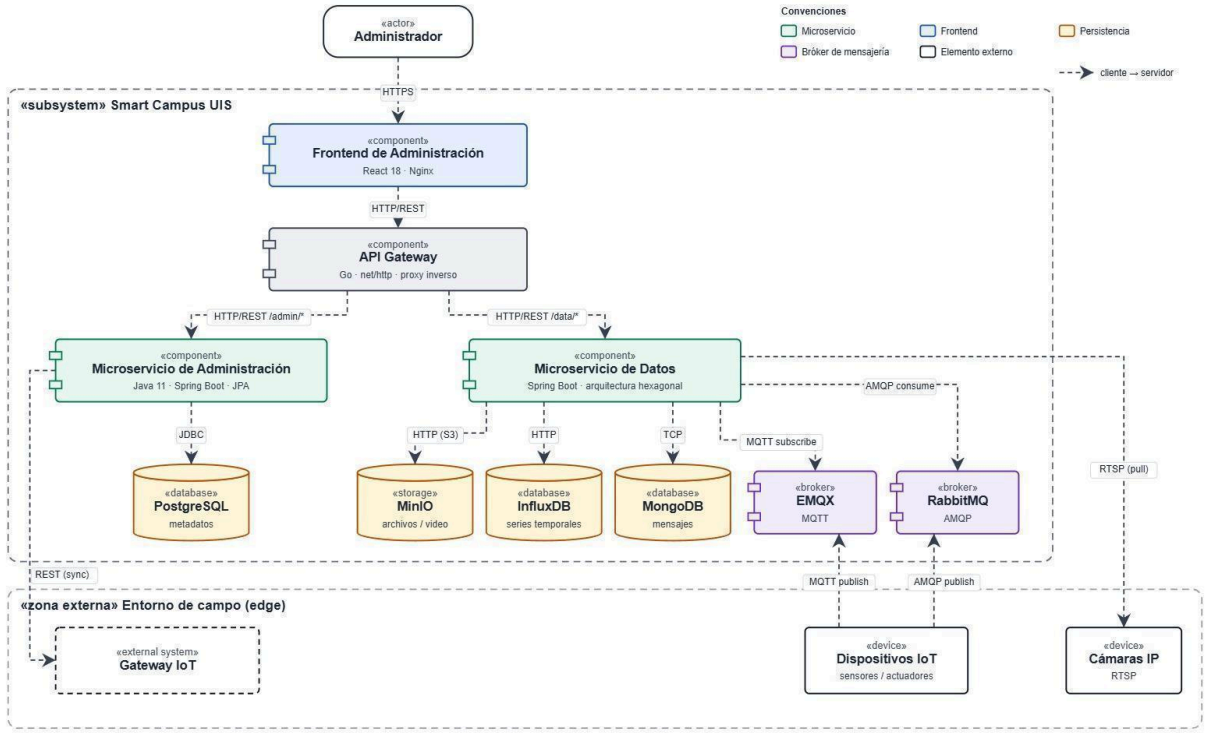
de disponibilidad HTTP.

### *5.1.9 Diagrama de componentes*

La Figura 3 presenta el diagrama de componentes de Smart Campus UIS construido a partir de la caracterización anterior. A diferencia de la Figura 2, explicita el protocolo de cada interacción y separa los elementos internos de la plataforma de los elementos del entorno de campo: los dispositivos IoT publican mensajes tanto en EMQX (MQTT) como en RabbitMQ (AMQP), el microservicio de datos consume de ambos brokers y obtiene el video de las cámaras IP mediante RTSP, y el microservicio de administración se sincroniza con el Gateway IoT mediante REST. Todas las relaciones se representan en sentido cliente → servidor.

**Figura 3**

*Diagrama de componentes de Smart Campus UIS*




## 5.2 Diagnóstico del sistema de despliegue, monitoreo y observabilidad actual

A partir de la caracterización arquitectural de la sección anterior, esta sección analiza el estado actual de las capacidades operacionales de la plataforma en tres dimensiones: despliegue, monitoreo y observabilidad. El análisis se realizó mediante revisión directa del repositorio, el código fuente y los archivos de configuración existentes.

### *5.2.1 Estado Actual del Despliegue*

El proceso de despliegue de Smart Campus UIS se basa íntegramente en Docker Compose. La tabla 5 resume los mecanismos de despliegue implementados, seguida del análisis de las deficiencias identificadas durante la revisión del repositorio y la configuración actual.

#### 5.2.1.1 Mecanismos Existentes.

**Tabla 5**

*Mecanismos existentes*

| Componente | Estado | Observaciones |
|---|---|---|
| Docker Compose | **Implementado** | Orquestación de 11 servicios |
| Dockerfiles | **Implementado** | Multi-stage builds |
| Restart Policy | **Implementado** | restart: always en servicios |
| Red Docker | **Implementado** | Bridge network: iot_campus_uis |
| Volúmenes | **Implementado** | Persistencia en ./data/* |

**5.2.1.2 Sin Pipeline de CI/CD.** No existen archivos de configuración para GitHub Actions, GitLab CI, Jenkins ni herramientas equivalentes. El despliegue es completamente manual mediante docker-compose up, sin automatización de pruebas, construcción ni distribución.

**5.2.1.3 Sin Health Checks.** Docker Compose no define healthchecks para ningún servicio, por lo que no es posible detectar automáticamente si un servicio está degradado. La política restart: always puede reiniciar servicios rotos indefinidamente sin notificar el fallo.

**5.2.1.4 Gestión de Secretos Insegura.** Las credenciales se almacenan en el archivo .env en

texto plano, y el archivo application.yml del admin_microservice contiene credenciales hardcodeadas. No se utiliza Docker Secrets, Vault ni ninguna herramienta de gestión de secretos.

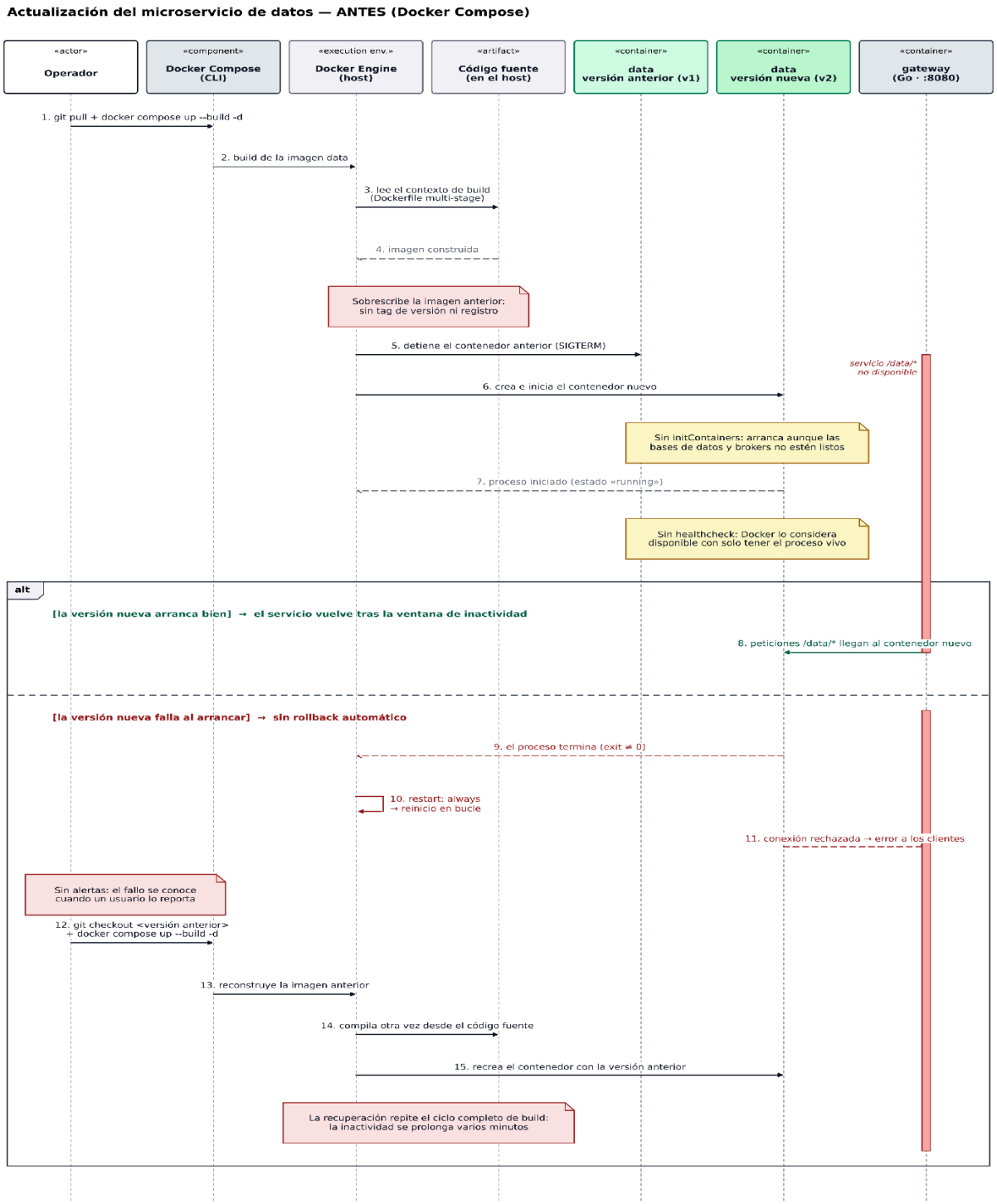
**5.2.1.5 Sin Versionamiento de Imágenes.** Las imágenes Docker no tienen tags de versión y no hay registro privado configurado, lo que hace imposible realizar rollback a versiones anteriores en caso de fallo.

**5.2.1.6 Sin Estrategias de Despliegue.** No hay implementación de Blue/Green, Canary ni Rolling Update. Cualquier despliegue implica tiempo de inactividad, ya que no existe mecanismo para reemplazar instancias sin interrumpir el servicio.

La Figura 4 ilustra, mediante un diagrama de secuencia, cómo se actualiza el microservicio de datos bajo este esquema. La actualización exige reconstruir la imagen a partir del código fuente en el host y detener el contenedor anterior antes de que el nuevo esté disponible, lo que abre una ventana de inactividad del servicio /data/*. Si la nueva versión falla al arrancar, no existe rollback automático ni alertas que adviertan el fallo: la recuperación depende de que un usuario lo reporte y de repetir el ciclo completo de construcción con la versión anterior.

**Figura 4**

*Actualización del microservicio de datos con Docker Compose (estado inicial)*



Las deficiencias en el proceso de despliegue evidencian una operación fundamentalmente manual y sin automatización: sin pipeline de CI/CD, sin health checks funcionales y sin estrategias de despliegue que eviten downtime. Esta combinación implica que cualquier actualización de la plataforma requiere intervención manual y conlleva riesgo de interrupción del servicio. Esta misma característica se refleja en las capacidades de monitoreo de la plataforma, analizadas a continuación.

### *5.2.2 Estado Actual del Monitoreo*

Las capacidades de monitoreo de la plataforma se encuentran en un estado parcial: algunos componentes están desplegados pero sin configuración funcional. La tabla 6 detalla el estado de cada herramienta y las brechas identificadas.

#### 5.2.2.1 Componentes Existentes.

**Tabla 6**

*Estado actual de monitoreo*

| Componente | Estado | Observaciones |
|---|---|---|
| Grafana 11.5 | **Parcial** | Desplegado pero sin dashboards |
| Spring Actuator | **Parcial** | Dependencia en data_microservice, sin configurar |
| Prometheus | **Ausente** | No implementado |
| Alertas | **Ausente** | No implementado |
| Métricas de negocio | **Ausente** | No implementado |

**5.2.2.2 Grafana Sin Configuración.** No existen dashboards pre-configurados para ningún

servicio, no hay datasources configurados automáticamente y se requiere configuración manual después de cada despliegue de la plataforma.

**5.2.2.3 Spring Actuator Sin Exponer.** La dependencia spring-boot-starter-actuator existe en data_microservice pero no hay configuración en application.properties para exponer sus endpoints: /health, /metrics e /info permanecen deshabilitados. El admin_microservice no tiene Spring Actuator.

**5.2.2.4 Sin Prometheus ni Métricas.** No hay servicio Prometheus en docker-compose.yml ni dependencia micrometer-registry-prometheus en los microservicios, por lo que no es posible recolectar métricas de CPU, memoria, threads, latencia de requests ni tasas de error.

**5.2.2.5 Sin Sistema de Alertas.** No hay AlertManager ni sistema de notificación equivalente. Los fallos del sistema pasan desapercibidos hasta que un usuario los reporta, ya que no existen canales de notificación por email, Slack u otros medios.

El monitoreo parcial identificado evidencia que la plataforma carece de visibilidad sobre su propio estado en tiempo de ejecución: sin Prometheus no hay métricas de CPU, memoria ni latencia; sin dashboards no hay forma de detectar degradación progresiva; sin alertas los fallos solo se conocen cuando un usuario los reporta. Esta limitación se profundiza al analizar las capacidades de observabilidad, presentadas a continuación.

### *5.2.3 Estado Actual de la Observabilidad*

La observabilidad comprende la capacidad de entender el estado interno del sistema a partir de sus salidas externas: logs, trazas y métricas. El análisis del código fuente reveló ausencias significativas en los tres dominios, detalladas en la tabla 7.

#### 5.2.3.1 Componentes Existentes.

**Tabla 7**

*Estado actual observabilidad*

| Componente | Estado | Observaciones |
|---|---|---|
| Logging estructurado | **Ausente** | Solo System.out.println |
| Tracing distribuido | **Ausente** | Sin OpenTelemetry/Jaeger/Zipkin |
| Correlación de logs | **Ausente** | Sin trace IDs entre servicios |
| Documentación API | **Parcial** | Swagger solo en data_microservice |
| Manejo de excepciones | **Básico** | Sin logging estructurado |

**5.2.3.2 Logging No Estructurado.** Se encontraron 14+ instancias de System.out.println y e.printStackTrace() en el código fuente. No hay archivos logback.xml ni log4j2.xml, los logs van únicamente a stdout sin formato JSON estructurado y sin niveles de log configurados (DEBUG, INFO, WARN, ERROR). No existe agregación centralizada de logs.

**5.2.3.3 Sin Tracing Distribuido.** No hay dependencias de OpenTelemetry, Jaeger ni Zipkin, por lo que es imposible rastrear una petición a través de múltiples servicios. La ausencia de trace IDs y span IDs en los logs hace que el diagnóstico de problemas en producción sea extremadamente difícil.

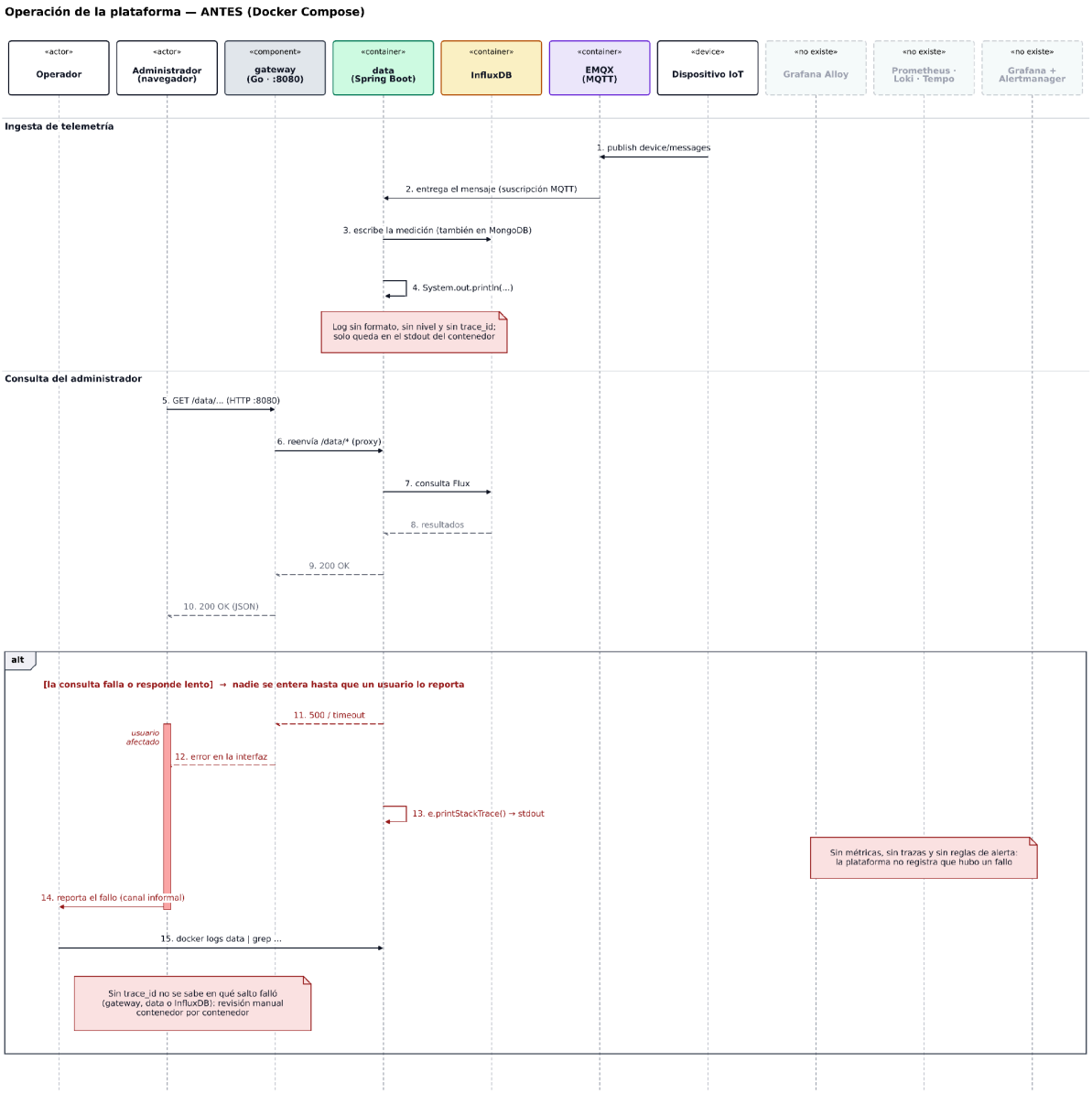
**5.2.3.4 Manejo de Excepciones Deficiente.** Los @ExceptionHandler no registran logs de las excepciones, no hay métricas de conteo de errores por tipo y los stack traces se pierden porque se usa e.printStackTrace() en lugar de logger.error(), ocultando errores que deberían generar alertas operacionales

La Figura 5 integra las deficiencias de monitoreo y observabilidad en el flujo de operación

cotidiana de la plataforma. La ingesta de telemetría y las consultas del administrador funcionan, pero cuando una consulta falla o responde con lentitud la plataforma no registra métricas, trazas ni alertas: el fallo se conoce solo cuando un usuario lo reporta, y el diagnóstico exige revisar manualmente los logs de cada contenedor sin un identificador de traza que permita ubicar el componente que falló.

**Figura 5**

*Operación de la plataforma con Docker Compose (estado inicial)*




Las deficiencias identificadas en despliegue, monitoreo y observabilidad revelan un sistema con capacidad operativa limitada para detectar y responder a fallos: el logging no estructurado impide

búsquedas eficientes, la ausencia de trazas distribuidas hace inviable el diagnóstico de problemas entre servicios, y el manejo de excepciones deficiente oculta errores que deberían generar alertas. Para priorizar adecuadamente las mejoras, la siguiente sección mapea las dependencias entre componentes e identifica aquellos cuya disponibilidad es crítica para el funcionamiento de la plataforma.

## 5.3 Mapeo de dependencias e identificación de componentes críticos

La tabla 8 presenta el mapeo de dependencias entre los componentes de la plataforma, identificando el protocolo de comunicación y el propósito de cada relación.

**Tabla 8**

*Mapeo de dependencias*

| Componente | Depende de | Protocolo / Mecanismo | Propósito |
|---|---|---|---|
| Frontend de administración | API Gateway | HTTP/REST | Acceso a funcionalidades de gestión de la plataforma |
| API Gateway | Microservicio de administración | HTTP/REST | Enrutamiento de peticiones con prefijo /admin/* |
| API Gateway | Microservicio de datos | HTTP/REST | Enrutamiento de peticiones con prefijo /data/* |
| Microservicio de administración | PostgreSQL | JDBC | Persistencia de metadatos y configuración del sistema |
| Microservicio de datos | MongoDB | TCP | Almacenamiento de mensajes y documentos |

| Componente | Depende de | Protocolo / Mecanismo | Propósito |
|---|---|---|---|
| Microservicio de datos | InfluxDB | HTTP | Almacenamiento y consulta de series temporales de sensores |
| Microservicio de datos | MinIO | HTTP | Almacenamiento de archivos binarios y grabaciones de video |
| Microservicio de datos | EMQX | MQTT (TCP) | Recepción de mensajes publicados por dispositivos IoT |
| Microservicio de datos | RabbitMQ | AMQP | Consumo de mensajes desde cola de eventos internos |

El microservicio de datos es el componente de mayor criticidad operacional, por lo que sus dependencias — InfluxDB, MongoDB, MinIO, EMQX y RabbitMQ — adquieren igual importancia operacional, siendo RabbitMQ la más crítica entre los brokers por las garantías de entrega que ofrece. El microservicio de administración es el segundo componente crítico, arrastrando a PostgreSQL como su dependencia directa. El API Gateway, aunque estructuralmente simple, representa el punto único de entrada HTTP de la plataforma, por lo que su caída deja inaccesibles todos los servicios independientemente de su estado interno.

## 5.4 Establecimiento de requerimientos funcionales

Con base en las limitaciones identificadas en el diagnóstico (sección 5.2) y los casos de uso documentados (sección 5.1.8), esta sección establece los requerimientos funcionales que debe cumplir

la solución. Los requerimientos se organizan en tres categorías que reflejan las tres dimensiones del problema: despliegue (ausencia de health checks, migraciones automáticas y perfiles de configuración), monitoreo (ausencia de métricas de negocio, alertas y dashboards) y observabilidad (ausencia de logging estructurado, trazas distribuidas y correlación entre servicios). Cada requerimiento se codifica con un identificador que permite trazabilidad directa hacia los criterios de validación experimental. Los requerimientos funcionales se muestran en la tabla 9.

**Tabla 9**

*Requerimientos funcionales*

| ID | Categoría | Requerimiento | Descripción |
|---|---|---|---|
| **RF-DEPL-01** | Despliegue | Health Checks | Los componentes deben exponer endpoints que verifiquen: estado de la aplicación, conectividad con dependencias (bases de datos, brokers, servicios externos), capacidad de procesar requests |
| **RF-DEPL-02** | Despliegue | Despliegue Selectivo de Dependencias | El sistema debe permitir despliegue opcional de sus dependencias (bases de datos, brokers, microservicios) con detección automática y fallback cuando no estén disponibles |
| **RF-DEPL-03** | Despliegue | Perfiles de Despliegue | El sistema debe incluir configuraciones predefinidas para diferentes escenarios de despliegue |

| ID | Categoría | Requerimiento | Descripción |
|---|---|---|---|
| **RF-DEPL-04** | Despliegue | Health Check Condicional | Los health checks deben reportar estados: UP (funcional), DOWN (no funcional), DEGRADED (funcional con capacidad reducida), con detalles de componentes opcionales disponibles/no disponibles |
| **RF-MON-01** | Monitoreo | Métricas de Infraestructura - Admin | El Admin debe exponer: operaciones CRUD ejecutadas por entidad, errores de sincronización con Gateway IoT, tiempo de respuesta por endpoint, conexiones activas a PostgreSQL y Redis, y tasa de operaciones por segundo. |
| **RF-MON-02** | Monitoreo | Métricas de Negocio - Data | El Data debe exponer: mensajes recibidos por protocolo, mensajes almacenados por BD, videos grabando/almacenados, archivos subidos, cámaras registradas, tasa de mensajes/segundo |
| **RF-MON-03** | Monitoreo | Métricas de Dependencias | Los componentes deben exponer: mensajes consumidos por broker, errores de conexión a brokers/BDs, lag del consumidor, queries ejecutadas por BD, latencia de queries |
| **RF-MON-04** | Monitoreo | Configuración de Alertas | El sistema debe permitir configurar alertas para: servicio caído, uso de recursos alto, tasa de errores alta, dependencias inaccesibles, métricas de negocio fuera de rango esperado |

| ID | Categoría | Requerimiento | Descripción |
|---|---|---|---|
| **RF-MON-05** | Monitoreo | Dashboards | El sistema debe incluir dashboards para: infraestructura (CPU, memoria, disco, red), aplicación (requests/s, latencia, errores), negocio (dispositivos activos, mensajes/s), estado general (UP/DOWN de componentes, alertas activas) |
| **RF-OBS-01** | Observabilidad | Niveles de Log Configurables | Los componentes deben permitir cambiar nivel de log en runtime sin reiniciar |
| **RF-OBS-02** | Observabilidad | Trazas Distribuidas | Los componentes deben generar trazas que permitan: seguir requests a través de servicios, medir latencia por componente, identificar cuellos de botella |
| **RF-OBS-03** | Observabilidad | Propagación de Contexto | Los componentes deben propagar correlation IDs a través de: headers HTTP, mensajes en brokers, logs |
| **RF-OBS-04** | Observabilidad | Logs de Auditoría | Los componentes deben registrar operaciones críticas: inicio/detención de servicios, CRUD de dispositivos, sincronización con Gateway IoT, control de grabación de cámaras, errores de conexión, cambios en configuración |
| **RF-OBS-05** | Observabilidad | Agregación y Búsqueda | El sistema debe centralizar logs en sistema de agregación que permita búsqueda por: rango de tiempo, nivel, servicio, traceId, contexto de negocio, palabras clave |

Los requerimientos funcionales establecen qué debe hacer la solución. Los requerimientos no funcionales presentados a continuación complementan ese conjunto definiendo los atributos de calidad que la solución debe cumplir: containerización, configuración por entornos, seguridad en la gestión de credenciales, orquestación, exposición y recolección de métricas, visualización y observabilidad estructurada. Estos atributos servirán como criterios de aceptación en la fase de validación. Los requerimientos no funcionales se muestran en la siguiente tabla.

## 5.5 Especificación de requerimientos no funcionales

Esta sección establece los requerimientos no funcionales de la solución, derivados del análisis arquitectural y el contexto operacional de Smart Campus UIS: una plataforma universitaria mantenida por estudiantes del semillero CAGE, desplegada en infraestructura con recursos moderados y en evolución continua. A diferencia de los requerimientos funcionales, que definen qué debe hacer la solución, los requerimientos no funcionales definen cómo debe comportarse: con qué mecanismo de despliegue, bajo qué modelo de configuración por entornos, con qué garantías de seguridad en la gestión de credenciales y con qué capacidades de visualización y agregación de señales. Estos atributos de calidad servirán como criterios de aceptación en la fase de validación. Los requerimientos no funcionales se muestran en la tabla 10.

**Tabla 10**

*Requerimientos no funcionales*

| ID | Categoría | Requerimiento | Descripción |
|---|---|---|---|
| **RNF-DEPL-01** | Despliegue | Containerización | Todos los componentes deben ser desplegables como contenedores |
| **RNF-DEPL-02** | Despliegue | Configuración por Entornos | Los componentes deben soportar configuración específica por entorno sin reconstruir imágenes |
| **RNF-DEPL-03** | Despliegue | Orquestación | El sistema debe incluir archivos de orquestación para desarrollo local y configuraciones para diferentes escenarios |
| **RNF-DEPL-04** | Despliegue | Documentación de Despliegue | El sistema debe incluir documentación clara de cómo desplegar cada componente y sus dependencias |
| **RNF-MON-01** | Monitoreo | Exposición de Métricas | Los componentes deben exponer métricas en formato estándar |
| **RNF-MON-02** | Monitoreo | Recolección de Métricas | El sistema debe incluir servidor que recolecte métricas |
| **RNF-MON-03** | Monitoreo | Visualización | El sistema debe incluir plataforma de visualización con dashboards básicos configurados |
| **RNF-OBS-01** | Observabilidad | Framework de Observabilidad | Los componentes deben usar framework que soporte: trazas distribuidas, métricas, logs estructurados |

| ID | Categoría | Requerimiento | Descripción |
|---|---|---|---|
| **RNF-OBS-02** | Observabilidad | Formato Estructurado | Los logs deben usar formato estructurado con correlación a trazas |
| **RNF-OBS-03** | Observabilidad | Agregación Centralizada | Los logs deben centralizarse en sistema de agregación y búsqueda |

## 5.6 Síntesis de Hallazgos

Smart Campus UIS es una plataforma funcional con una arquitectura de microservicios bien definida, pero con una brecha operacional significativa entre su capacidad de procesamiento y su capacidad de ser operada, monitoreada y desplegada de forma confiable. El microservicio de datos concentra la mayor parte de la lógica de la plataforma y depende de cinco sistemas de infraestructura simultáneos, ninguno de los cuales cuenta con health checks, métricas ni trazas que permitan detectar su degradación. El despliegue es manual, sin automatización ni estrategias que eviten downtime. Los logs no son estructurados ni centralizados, y no existe ningún mecanismo que correlacione eventos entre servicios. Los requerimientos establecidos en esta fase (funcionales y no funcionales) definen el conjunto mínimo de capacidades que debe incorporar la solución para cerrar esa brecha. La Fase de revisión de establecimiento de la estrategia integrada toma esos requerimientos como punto de partida para evaluar alternativas tecnológicas, seleccionar el stack de herramientas y definir la estrategia integrada de despliegue, monitoreo y observabilidad.

# 6. Establecimiento de la estrategia integrada

Con base en la caracterización de la plataforma realizada en la sección 5 y los hallazgos de la revisión de literatura presentados en la sección 3, esta sección establece la estrategia integrada de despliegue, monitoreo y observabilidad para Smart Campus UIS. El proceso se estructura en cuatro etapas: el establecimiento de criterios de evaluación derivados de los requerimientos identificados, la evaluación comparativa de alternativas disponibles, la identificación de integraciones tecnológicas viables con el stack existente, y la selección justificada de las herramientas y estrategias que conforman la propuesta final.

## 6.1 Establecimiento de Criterios de Evaluación

Con base en el análisis del estado del arte, los requerimientos identificados en la Fase 1, y el contexto específico de Smart Campus UIS, se definen los siguientes criterios de evaluación. Las herramientas se evalúan en escala de 1 a 5, excepto el criterio de licenciamiento open source que es excluyente.

### *6.1.1 Criterios de Evaluación para Patrones de Despliegue*

Basándose en los trade-offs identificados en la literatura (Glushach, 2023; Lesmana, 2023; Platform Engineers, 2025), se establecen los criterios para evaluar los patrones de despliegue en la tabla 11:

**Tabla 11**

*Criterios de evaluación, patrones de despliegue*

| Criterio | Descripción |
|---|---|
| **Complejidad–de Implementación** | Esfuerzo técnico requerido para implementar el patrón en la infraestructura existente |
| **Recursos Requeridos** | Cantidad de recursos adicionales (CPU, memoria, instancias) necesarios durante el despliegue |
| **Velocidad de Rollback** | Tiempo necesario para revertir a la versión anterior en caso de fallo |
| **Zero-Downtime** | Capacidad de realizar el despliegue sin interrupción del servicio |
| **Validación en Producción** | Capacidad de validar la nueva versión con tráfico real antes del despliegue completo |

### *6.1.2 Criterio Excluyente: Licenciamiento Open Source*

El licenciamiento open source es un requisito excluyente (Sí/No). Las herramientas que no cumplan este criterio son descartadas automáticamente de la selección final, independientemente de sus méritos técnicos. Sin embargo, se evaluarán en todos los demás criterios para fines comparativos.

El criterio excluyente de licenciamiento open source responde a tres razones complementarias: restricción presupuestaria absoluta (el proyecto no dispone de fondos para licencias comerciales), contexto académico (se requiere transparencia tecnológica para facilitar el aprendizaje y la transferencia de conocimiento entre cohortes), y sostenibilidad a largo plazo (se evita dependencia de proveedores comerciales en un proyecto universitario con horizonte de operación extendido).

### *6.1.3 Criterios de Evaluación para Herramientas (Escala 1-5)*

Los criterios se evalúan en escala de 1 (deficiente) a 5 (excelente) en la tabla 12:

**Tabla 12**

*Criterios de evaluación, herramientas*

| Criterio | Descripción |
|---|---|
| **Curva de Aprendizaje** | Evalúa la complejidad conceptual y operacional de la herramienta en relación con el perfil del equipo: estudiantes de ingeniería de sistemas con experiencia en Docker y Spring Boot, pero sin formación específica en operaciones de infraestructura avanzada. |
| **Documentación y Comunidad** | Al no contar con soporte comercial, la documentación oficial y la comunidad son la única red de seguridad técnica disponible. Evalúa la calidad, exhaustividad y actualización de la documentación, y la actividad de la comunidad en foros y repositorios como mecanismo de soporte alternativo. |
| **Madurez y Estabilidad** | Evalúa el tiempo en producción de la herramienta, la estabilidad de su API y versiones, su adopción en entornos reales, y el respaldo de organizaciones reconocidas (CNCF, Apache) que garanticen mantenimiento a largo plazo. |
| **Compatibilidad e Integración** | Evalúa la existencia de integraciones oficiales o ampliamente documentadas con el stack existente (Spring Boot, Docker, EMQX, PostgreSQL, MongoDB, InfluxDB) y el esfuerzo de configuración requerido para lograr una integración funcional sin modificaciones estructurales. |
| **Adecuación al Tamaño** | Smart Campus opera con 2–5 microservicios y hasta 100 dispositivos IoT. Evalúa si la herramienta es apropiada para esta escala y si su modelo operacional se mantiene manejable a medida que el sistema crece dentro del rango esperado. |

| Criterio | Descripción |
|---|---|
| **Consumo de Recursos** | La infraestructura disponible corresponde a servidores universitarios con recursos moderados y sin posibilidad de escalar bajo demanda. Evalúa el overhead en CPU, RAM y almacenamiento que introduce la herramienta en condiciones normales de operación. |
| **Escalabilidad Operacional** | Evalúa con qué facilidad se pueden incorporar nuevos microservicios, nodos o herramientas complementarias, considerando la compatibilidad con el ecosistema cloud-native (CNCF, Helm charts, Operators) y el esfuerzo operacional requerido para cada extensión. |

### *6.1.4 Justificación de la Escala de Puntuación*

**6.1.4.1 Escala para Herramientas.** Los criterios de evaluación de herramientas mostrados en la tabla 13 consideran las características técnicas y operacionales de las soluciones analizadas. Las condiciones están definidas para el contexto de Smart Campus UIS: infraestructura universitaria con recursos moderados, equipo con experiencia en Spring Boot y Docker, y un stack de 2 a 5 microservicios con hasta 100 dispositivos IoT. Ante ambigüedad entre dos valores adyacentes, se adopta el criterio conservador.

**Tabla 13**
*Justificación escala de puntuación, herramientas*

| Criterio | 1 — Inaceptable | 2 — Deficiente | 3 — Aceptable | 4 — Bueno | 5 — Excelente |
|---|---|---|---|---|---|
| **Curva de Aprendizaje** | Requiere conocimientos especializados no disponibles en el | Requiere formación significativa que el equipo no posee; documentación | Curva considerable; aprendizaje posible pero | El equipo puede dominarla en 1–2 semanas con documentación | El equipo puede operar la herramienta en menos de una |

| Criterio | 1 — Inaceptable | 2 — Deficiente | 3 — Aceptable | 4 — Bueno | 5 — Excelente |
|---|---|---|---|---|---|
| | equipo cuya adquisición excede el alcance del proyecto. | insuficiente para autoaprendizaje. | requiere varias semanas de dedicación o formación guiada. | oficial; curva manejable dado el conocimiento existente en Docker/Spring Boot. | semana con documentación oficial; no requiere formación adicional ni experiencia específica previa. |
| **Documentación y Comunidad** | Sin documentación oficial o totalmente abandonada; comunidad inexistente. | Documentación oficial escasa o desactualizada; comunidad pequeña con respuesta nula o muy lenta. | Documentación oficial existente con lagunas notables; comunidad activa pero sin cobertura del contexto IoT/microservicios. | Documentación oficial completa con guías de inicio rápido; comunidad activa con cobertura del contexto. Algunas áreas poco documentadas. | Documentación oficial exhaustiva y versionada con ejemplos del contexto (Spring Boot, Kubernetes, IoT); comunidad muy activa con alta cobertura en foros y repositorios. |
| **Madurez y Estabilidad** | Herramienta abandonada, sin mantenimiento activo o con historial de cambios breaking que comprometen la estabilidad. | En desarrollo activo pero con versiones inestables, cambios disruptivos frecuentes o sin adopción significativa en producción. | Estable con versiones productivas, pero sin respaldo de organización reconocida o con cambios disruptivos ocasionales. | Estable con varios años en producción y comunidad consolidada; respaldo institucional presente o en proceso de graduación (e.g., incubación | Graduada por CNCF u organización equivalente; múltiples años de adopción en producción a escala con historial de versiones estables y soporte a largo |

| Criterio | 1 — Inaceptable | 2 — Deficiente | 3 — Aceptable | 4 — Bueno | 5 — Excelente |
|---|---|---|---|---|---|
| | | | | CNCF). | plazo. |
| **Compatibilidad e Integración** | Sin integración conocida ni documentada con ninguno de los componentes del stack de Smart Campus UIS. | Integración posible en teoría pero requiere desarrollo de adaptadores propios sin documentación de referencia. | Integración documentada con parte del stack mediante conectores de terceros; configuración no trivial requerida. | Integración documentada con los componentes principales del stack (Spring Boot, Docker, EMQX) mediante adaptadores estándar bien mantenidos. | Integración nativa o de primera clase con los componentes del stack; documentación oficial específica con ejemplos funcionales verificados. |
| **Adecuación al Tamaño** | Diseñada exclusivamente para sistemas enterprise de gran escala; introduce over-engineering inviable en el contexto. | Orientada a sistemas medianos-grandes; su uso en 2–5 microservicios implica overhead que no aporta valor proporcional. | Puede usarse en el contexto pero con funcionalidades subutilizadas o configuración que excede la necesidad real. | Diseñada para sistemas pequeños–medianos; escala correctamente sin imponer complejidad innecesaria. | Validada explícitamente para entornos de recursos moderados y sistemas pequeños–medianos; documentación o casos de uso en contexto IoT o edge disponibles. |
| **Consumo de Recursos** | Consumo que hace inviable su despliegue en la infraestructura universitaria | Consumo que presiona significativamente la infraestructura; coexistencia con el | Consumo notable que requiere ajustes de configuración para coexistir con el | Consumo moderado y predecible; coexiste con el stack sin | Consumo mínimo relativo a su categoría funcional; diseñada para |

| Criterio | 1 — Inaceptable | 2 — Deficiente | 3 — Aceptable | 4 — Bueno | 5 — Excelente |
|---|---|---|---|---|---|
| | disponible incluso con optimizaciones. | stack completo requiere compromisos inaceptables. | stack; viable pero con margen ajustado. | comprometer otros servicios, aunque requiere planificación de recursos. | entornos con recursos limitados o su overhead es despreciable respecto al valor aportado. |
| **Escalabilidad Operacional** | Añadir servicios, nodos o herramientas requiere rediseño estructural de la infraestructura; incompatible con el ecosistema estándar. | Crecimiento posible, pero requiere trabajo manual significativo para cada nuevo componente; no hay mecanismos estándar de extensión. | Puede crecer con esfuerzo considerable; compatibilidad parcial con herramientas estándar o ecosistema limitado. | Crecimiento bien documentado con mecanismos estándar; la mayoría de las herramientas del ecosistema son compatibles con configuración menor. | Crecimiento nativo y sin fricción; ecosistema amplio (CNCF, Helm, Operators) permite añadir servicios, nodos y herramientas con mecanismos estandarizados y documentados. |

**6.1.4.2 Escala para Patrones de Despliegue.** Los criterios de evaluación de patrones de despliegue mostrados en la tabla 14 evalúan características operacionales del patrón, no de una herramienta de software. Las condiciones a continuación están ancladas al contexto de Smart Campus UIS: infraestructura universitaria con recursos moderados y tolerancia a interrupciones breves en servicios no críticos.

**Tabla 14**

*Justificación escala de puntuación, patrones de despliegue*

| Criterio | 1 — Inaceptable | 2 — Deficiente | 3 — Aceptable | 4 — Bueno | 5 — Excelente |
|---|---|---|---|---|---|
| **Complejidad de Implementación** | Requiere infraestructura especializada (service mesh, traffic splitting avanzado) no disponible en el stack. | Requiere configuración significativa fuera del orquestador base; múltiples dependencias externas. | Configurable dentro del orquestador con pasos manuales adicionales documentados. | Implementable con configuración estándar del orquestador; sin dependencias externas. | Soportado nativamente por el orquestador sin configuración adicional; usa parámetros por defecto. |
| **Recursos Requeridos** | Requiere duplicar o triplicar la infraestructura activa; inviable en el contexto universitario. | Requiere mantener una copia completa del entorno de producción activa durante el despliegue. | Requiere recursos adicionales variables según tráfico desplazado; overhead notable pero no duplicación. | Recursos adicionales menores y temporales durante la transición; overhead predecible y acotado. | No requiere recursos adicionales significativos; el despliegue reemplaza instancias dentro de la capacidad existente. |
| **Velocidad de Rollback** | Rollback implica redespliegue completo; puede tardar minutos con intervención manual. | Rollback gradual reemplazando instancias una a una; tarda varios minutos según número de réplicas. | Rollback posible con pasos manuales adicionales antes de completarse. | Rollback rápido en segundos mediante ajuste del porcentaje de tráfico; no es instantáneo. | Rollback instantáneo mediante redirección del balanceador al entorno anterior; sin tiempo de convergencia. |

| Criterio | 1 — Inaceptable | 2 — Deficiente | 3 — Aceptable | 4 — Bueno | 5 — Excelente |
|---|---|---|---|---|---|
| **Zero-Downtime** | El despliegue causa interrupción del servicio; usuarios experimentan errores durante la transición. | Puede causar degradación o interrupciones breves durante momentos críticos de la transición. | Zero-downtime posible pero condicionado a configuración cuidadosa de health checks. | Zero-downtime garantizado en condiciones normales con health checks correctamente configurados. | Zero-downtime garantizado por diseño del patrón; no depende de configuración adicional. |
| **Validación en Producción** | No permite validar con tráfico real; validación ocurre solo en ambientes de prueba previos. | Validación limitada o en paralelo sin redirigir tráfico real a la nueva versión. | Permite validación post-switch; la nueva versión ya está en producción cuando se detectan problemas. | Permite validar con un subconjunto controlado de tráfico real antes de completar el despliegue. | Permite validación progresiva con tráfico real (1%→10%→100%); máxima visibilidad sobre comportamiento en producción. |

## 6.2 Evaluación Comparativa de Alternativas

Esta sección presenta la evaluación de herramientas candidatas para cada componente del stack de observabilidad. Se incluyen tanto herramientas open source como propietarias, evaluando todas en los criterios técnicos para fines comparativos. Las herramientas propietarias son descartadas al final por no cumplir el criterio excluyente.

*Nota.* Las herramientas propietarias se incluyen como referencia de mercado para contextualizar la evaluación y demostrar que la selección open source es competitiva técnicamente. La información de productos propietarios proviene de su documentación oficial (Amazon Web Services, 2024a, 2024b;

Datadog, 2024; Dynatrace, 2024; Microsoft Azure, 2024; New Relic, 2024; Splunk, 2024; Sumo Logic, 2024). Adicionalmente, datos del sector confirman que en entornos enterprise reales el 73% de las integraciones de observabilidad son con herramientas propietarias, siendo Datadog, New Relic, Splunk y Dynatrace las más adoptadas, lo que refuerza su relevancia como referencia de mercado (BigPanda, 2025).

### *6.2.1 Evaluación Comparativa de Patrones de Despliegue*

En la tabla 15 se evalúan los tres patrones utilizando una escala de 1 a 5, donde 5 representa la mejor puntuación:

**Tabla 15**

*Evaluación comparativa, patrones de despliegue*

| **Criterio** | **Rolling Update** | **Blue-Green** | **Canary** |
|---|---|---|---|
| Complejidad de Implementación | 5 - Baja | 3 - Media | 2 - Alta |
| Recursos Requeridos | 5 - Mínimos | 2 - Duplicados | 3 - Variables |
| Velocidad de Rollback | 3 - Gradual | 5 - Instantáneo | 4 - Rápido |
| Zero-Downtime | 5 - Sí | 5 - Sí | 5 - Sí |
| Validación en Producción | 2 - Limitada | 3 - Post-switch | 5 - Completa |
| **PROMEDIO** | **4.0** | **3.6** | **3.8** |

Rolling Update obtiene la mayor puntuación promedio (4.0) y es el único patrón soportado de forma nativa por K3s sin configuración adicional, lo que lo convierte en la estrategia más adecuada para el contexto de Smart Campus UIS.

### *6.2.2 Plataformas de Orquestación*

En la tabla 16 se evalúan las plataformas de orquestación utilizando una escala de 1 a 5, donde 5 representa la mejor puntuación:

**Tabla 16**

*Evaluación comparativa, plataformas de orquestación*

| **Criterio** | **K3s** | **Swarm** | **K8s** | **ECS** | **ACI** |
|---|---|---|---|---|---|
| Open Source | Sí | Sí | Sí | No | No |
| Curva de Aprendizaje | 4 | 5 | 2 | 3 | 4 |
| Documentación | 4 | 3 | 5 | 5 | 4 |
| Madurez | 4 | 4 | 5 | 5 | 4 |
| Compatibilidad | 5 | 3 | 5 | 3 | 3 |
| Adecuación Tamaño | 5 | 4 | 3 | 3 | 4 |
| Consumo Recursos | 5 | 4 | 2 | 4 | 5 |
| Escalabilidad Operacional | 5 | 2 | 5 | 3 | 3 |
| **PROMEDIO** | **4.6** | **3.6** | **3.9** | **3.7** | **3.9** |

K3s obtiene la mayor puntuación entre las alternativas open source (4.6), superando a Docker Swarm en compatibilidad y escalabilidad operacional, y a K8s estándar en adecuación al tamaño y consumo de recursos. Se selecciona como plataforma de orquestación para Smart Campus UIS.

### *6.2.3 Herramientas de Métricas*

En la tabla 17 se evalúan las herramientas de métricas utilizando una escala de 1 a 5, donde 5 representa la mejor puntuación:

**Tabla 17**

*Evaluación comparativa, herramientas de métricas*

| **Criterio** | **Prometheus** | **InfluxDB** | **Datadog** | **New Relic** |
|---|---|---|---|---|
| Open Source | Sí | Sí | No | No |
| Curva de Aprendizaje | 4 | 3 | 5 | 5 |
| Documentación | 5 | 4 | 5 | 5 |
| Madurez | 5 | 3 | 5 | 5 |
| Compatibilidad | 5 | 3 | 5 | 4 |
| Adecuación Tamaño | 5 | 4 | 3 | 3 |
| Consumo Recursos | 4 | 4 | 5 | 5 |
| Escalabilidad Operacional | 4 | 2 | 1 | 1 |
| **PROMEDIO** | **4.6** | **3.3** | **4.1** | **4.0** |

Prometheus obtiene la mayor puntuación entre las alternativas open source (4.6), destacando especialmente en compatibilidad con el stack existente y escalabilidad operacional. Se selecciona como herramienta de métricas para Smart Campus UIS.

### *6.2.4 Herramientas de Logs*

En la tabla 18 se evalúan las herramientas de logs utilizando una escala de 1 a 5, donde 5 representa la mejor puntuación:

**Tabla 18**

*Evaluación comparativa, herramientas de logs*

| Criterio | Loki | ELK | Splunk | Sumo Logic |
|---|---|---|---|---|
| Open Source | Sí | Sí | No | No |
| Curva de Aprendizaje | 5 | 3 | 3 | 4 |
| Documentación | 4 | 5 | 5 | 4 |
| Madurez | 4 | 5 | 5 | 4 |
| Compatibilidad | 5 | 3 | 3 | 3 |
| Adecuación Tamaño | 5 | 3 | 1 | 3 |
| Consumo Recursos | 5 | 1 | 3 | 4 |
| Escalabilidad Operacional | 4 | 3 | 3 | 3 |
| **PROMEDIO** | **4.6** | **3.1** | **3.3** | **3.6** |

Loki obtiene la mayor puntuación entre las alternativas open source (4.6), con ventajas significativas en consumo de recursos y compatibilidad con el ecosistema Grafana Labs. Se selecciona como herramienta de agregación de logs para Smart Campus UIS.

### *6.2.5 Herramientas de Trazas Distribuidas*

En la tabla 19 se evalúan las herramientas de trazas utilizando una escala de 1 a 5, donde 5 representa la mejor puntuación:

**Tabla 19**

*Evaluación comparativa, herramientas de trazas*

| Criterio | Tempo | Jaeger | Zipkin | X-Ray | Dynatrace |
|---|---|---|---|---|---|
| Open Source | Sí | Sí | Sí | No | No |
| Curva de Aprendizaje | 4 | 3 | 4 | 3 | 5 |
| Documentación | 4 | 5 | 4 | 5 | 5 |
| Madurez | 4 | 5 | 3 | 5 | 5 |
| Compatibilidad | 5 | 4 | 3 | 2 | 4 |
| Adecuación Tamaño | 5 | 3 | 4 | 3 | 1 |
| Consumo Recursos | 5 | 3 | 4 | 4 | 3 |
| Escalabilidad Operacional | 4 | 4 | 3 | 2 | 3 |
| **PROMEDIO** | **4.4** | **3.9** | **3.4** | **3.4** | **3.7** |

Tempo obtiene la mayor puntuación entre las alternativas open source (4.4), con ventajas en adecuación al tamaño, consumo de recursos e integración nativa con Grafana. Se selecciona como backend de trazas distribuidas para Smart Campus UIS.

### *6.2.6 Herramientas de Visualización*

En la tabla 20 se evalúan las herramientas de visualización utilizando una escala de 1 a 5, donde 5 representa la mejor puntuación:

**Tabla 20**

*Evaluación comparativa, herramientas de visualización*

| Criterio | Grafana | Kibana |
|---|---|---|
| Open Source | Sí | Sí |
| Curva de Aprendizaje | 4 | 3 |
| Documentación | 5 | 4 |
| Madurez | 5 | 5 |
| Compatibilidad | 5 | 2 |
| Adecuación Tamaño | 5 | 2 |
| Consumo Recursos | 4 | 1 |
| Escalabilidad Operacional | 5 | 2 |
| **PROMEDIO** | **4.7** | **2.7** |

Grafana obtiene la mayor puntuación de todas las categorías evaluadas (4.7), con ventajas determinantes en compatibilidad, escalabilidad operacional y consumo de recursos frente a Kibana. Se selecciona como plataforma de visualización para Smart Campus UIS.

## 6.3 Identificación de Integraciones Tecnológicas

Esta sección documenta las integraciones tecnológicas viables entre las herramientas seleccionadas y el stack existente de Smart Campus UIS, con base en la documentación oficial de cada componente y fuentes técnicas de implementación (Bector, 2025; EMQX Team, 2024; Mustufa, 2025; Zehri, 2025).

### *6.3.1 Integración Spring Boot y Observabilidad*

Las integraciones que se muestran en la tabla 21 están documentadas en (Mustufa, 2025) y (Bector, 2025):

**Tabla 21**

*Integración Spring Boot y observabilidad*

| Integración | Componentes | Beneficio |
|---|---|---|
| Micrometer + Prometheus | micrometer-registry-prometheus + /actuator/prometheus | Métricas JVM, HTTP, custom automáticos (Mustufa, 2025) |
| OTel Java Agent | opentelemetry-javaagent.jar como -javaagent | Auto-instrumentación sin cambios código (Bector, 2025; Mustufa, 2025) |
| Logback + JSON | logstash-logback-encoder | Logs estructurados con traceId/spanId (Mustufa, 2025) |

### *6.3.2 Integración EMQX y Observabilidad*

EMQX 5.2+ ofrece integración nativa con el stack de observabilidad seleccionado mediante dos mecanismos complementarios: exposición de métricas del broker MQTT en formato Prometheus, y exportación directa de telemetría al OTel Collector mediante el protocolo gRPC OTLP (EMQX Team, 2024).

### *6.3.3 Flujo de Datos de Observabilidad*

La tabla 22 sintetiza el pipeline completo de observabilidad, mostrando cómo cada señal

(métricas, trazas y logs) fluye desde su origen en los componentes del stack hasta su visualización en Grafana (Mustufa, 2025; Zehri, 2025):

**Tabla 22**

*Flujo de datos de observabilidad*

| Señal | Origen | Collector | Backend |
|---|---|---|---|
| Metrics | Micrometer, node_exporter, EMQX | Prometheus (pull/scrape) | Prometheus TSDB → Grafana |
| Traces | OTel Java Agent, EMQX OTel | OTel Collector (OTLP gRPC) | Tempo → Grafana |
| Logs | Logback JSON, Container logs | Grafana Alloy | Loki → Grafana |

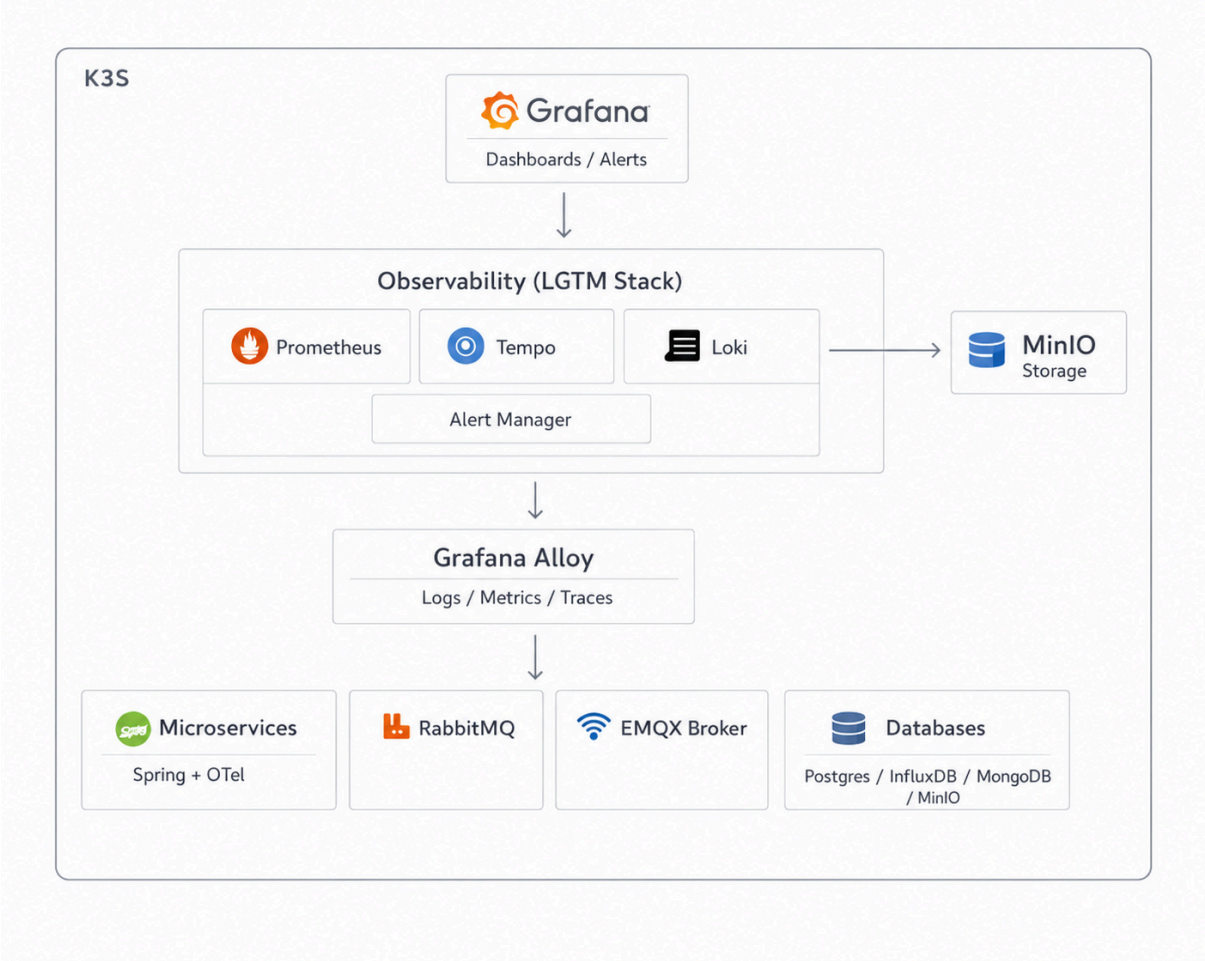
## 6.4 Selección Justificada de Estrategias y Herramientas

La estrategia integrada para Smart Campus UIS combina un stack de observabilidad unificado con una estrategia de despliegue sin tiempo de inactividad, sobre una infraestructura de orquestación ligera adecuada al contexto universitario. El diagrama de arquitectura (Figura 6) ilustra cómo los componentes seleccionados se articulan en un sistema coherente.

### *6.4.1 Arquitectura de la Estrategia Integrada*

**Figura 6**

*Arquitectura de la estrategia integrada*




El stack de observabilidad se estructura alrededor del ecosistema Grafana Labs: Prometheus recolecta métricas mediante scraping, Grafana Alloy actúa como colector unificado de telemetría consolidando la recolección de logs, trazas y métricas en un único agente, Loki almacena los logs y Tempo las trazas distribuidas, y Grafana centraliza la visualización y correlación de las tres señales mediante derived fields con el traceId como enlace común. La coherencia ecosistémica entre estos componentes reduce la fricción de configuración y garantiza compatibilidad entre versiones (Grafana Labs, 2024; Mustufa, 2025; Zehri, 2025). La instrumentación de los microservicios Spring Boot se

realiza mediante el OTel Java Agent, que genera trazas distribuidas sin modificar el código fuente, y Micrometer, que expone métricas JVM y HTTP en formato Prometheus (Bector, 2025).

K3s orquesta el stack completo sobre la infraestructura universitaria disponible. Su huella reducida (~512 MB RAM) permite que los microservicios de Smart Campus UIS y el stack de observabilidad coexistan sin comprometer los recursos disponibles, y su compatibilidad total con la API de Kubernetes habilita de forma nativa los mecanismos que el stack requiere: ServiceMonitor CRDs para el descubrimiento automático de servicios por Prometheus, DaemonSets para desplegar Alloy en cada nodo, y Rolling Update como estrategia de despliegue sin configuración adicional (Borgp, 2022; Mthembu et al., 2023; Rancher Labs, 2024).

Rolling Update se selecciona como patrón de despliegue por ser el de menor complejidad operativa, consumo de recursos mínimo durante la transición y zero-downtime garantizado por diseño: Canary requiere infraestructura de routing avanzada (service mesh) que excede el alcance del proyecto, y Blue-Green implica duplicar la infraestructura activa, lo cual no es viable en el contexto universitario (Decimavilla-Alarcón y Marcillo-Franco, 2025; Lesmana, 2023). En cuanto al colector de telemetría, Grafana Alloy se selecciona sobre Promtail no solo porque este último alcanzó su fin de vida en marzo de 2026, sino porque Alloy es la distribución de Grafana Labs del OpenTelemetry Collector, capaz de consolidar la recolección de métricas, logs y trazas en un único proceso y reducir así el número de agentes necesarios en el clúster (Hartmann, 2025). Finalmente, Mimir, aunque contemplado en el stack LGTM como capa de retención de métricas a largo plazo, no se incorpora en esta etapa dado que el volumen generado por 2 a 5 microservicios y hasta 100 dispositivos IoT no justifica su overhead operativo, quedando documentado como opción de escalamiento futuro (Grafana Labs, 2024; Zehri, 2025).

### *6.4.2 Stack Seleccionado*

La tabla 23 presenta el stack de herramientas y patrones de despliegue seleccionados con base en la evaluación realizada.

**Tabla 23**

*Stack seleccionado*

| Componente | Herramienta | Puntuación | Fuente de Referencia |
|---|---|---|---|
| Orquestación | K3s | 4.6 | (Mthembu et al., 2023) |
| Métricas | Prometheus | 4.6 | (Emon, 2025; Mustufa, 2025; Olawale, 2025), (Decimavilla-Alarcón y Marcillo-Franco, 2025) |
| Logs | Loki + Grafana Alloy | 4.6 | (Emon, 2025; Mustufa, 2025; Zehri, 2025), (Decimavilla-Alarcón y Marcillo-Franco, 2025) |
| Trazas | Tempo + OpenTelemetry | 4.4 | (Bector, 2025; Mustufa, 2025; Zehri, 2025) |
| Visualización | Grafana | 4.7 | (Ayogun, 2024; Bector, 2025; Emon, 2025; EMQX Team, 2024; Mustufa, 2025; Zehri, 2025) |
| Despliegue | Rolling Update | 4.0 | (Decimavilla-Alarcón y Marcillo-Franco, 2025) |

# 7. Adaptación de servicios Smart Campus UIS

La sección de adaptación constituye el puente entre la estrategia definida en la sección 6 y su implementación sobre la infraestructura de Smart Campus UIS. Antes de desplegar las herramientas seleccionadas, fue necesario preparar los servicios existentes para que pudieran integrarse con ellas: esto implicó identificar las brechas entre el estado actual de cada servicio y los requerimientos de la estrategia, y aplicar las modificaciones correspondientes. El proceso se estructuró en dos dimensiones: adaptaciones orientadas a la estrategia de despliegue, que abordaron problemas en las imágenes de contenedor y la configuración del entorno, e instrumentación de los servicios para habilitar la recolección de métricas, trazas y logs según las herramientas seleccionadas.

## 7.1 Análisis de mejoras de adaptación

Se identificaron las modificaciones necesarias en los servicios de Smart Campus UIS para su integración con la estrategia definida. El análisis se estructuró en dos dimensiones: adaptaciones requeridas para la estrategia de despliegue e instrumentación necesaria para monitoreo y observabilidad.

### *7.1.1 Mejoras de adaptación para despliegue*

Se evaluaron los servicios para identificar puntos que dificultan la implementación de la estrategia de despliegue y el uso de K3s. Los hallazgos se concentraron en el microservicio de datos y

se detallan en la Tabla 24.

**Tabla 24**

*Requerimientos de adaptación para despliegue*

| Servicio | Hallazgo | Impacto |
|---|---|---|
| Microservicio de datos | Imagen Docker de 2GB por importación completa de JavaCV, .jar con peso de 1GB | Tiempos de despliegue elevados, consumo excesivo de almacenamiento y complicaciones para usar repositorios de imágenes. |
| Microservicio de datos | Dockerfile ineficiente teniendo en cuenta la estructura del microservicio | Tiempos de despliegue elevados, consumo parcial de almacenamiento |

Ver Apéndice B para el Dockerfile original del microservicio de datos.

### *7.1.2 Mejoras de instrumentación para monitoreo y observabilidad*

Se determinaron las modificaciones de instrumentación requeridas en los microservicios de datos y administración, por ser los componentes con mayor relevancia operacional de la plataforma, con base en el diagnóstico de las secciones 5.2.2 y 5.2.3 como lo muestra la tabla 25.

**Tabla 25**

*Requerimientos de instrumentación para monitoreo y observabilidad*

| Servicio | Modificación requerida | Capacidad habilitada |
|---|---|---|
| Microservicio de datos y administración | Agregar spring-boot-actuator y micrometer-prometheus | Exposición de métricas para Prometheus |
| Microservicio de datos y administración | Incorporar agente OpenTelemetry | Trazas distribuidas |
| Microservicio de datos y administración | Implementar logging estructurado | Correlación de logs por trace_id y span_id |

## 7.2 Adaptación de servicios para estrategia de despliegue

A partir de los hallazgos identificados en la sección 7.1.1, se realizaron las modificaciones necesarias en el microservicio de datos para garantizar su compatibilidad con la estrategia de despliegue Rolling Update sobre K3s.

### *7.2.1 Optimización de imágenes de contenedor*

La optimización de imágenes de contenedor se abordó en dos niveles: la reducción del artefacto generado por el microservicio de datos mediante el reemplazo de dependencias JavaCV, y la refactorización del Dockerfile para mejorar la eficiencia del proceso de construcción.

**7.2.1.1 Optimización del Microservicio de Datos.** El microservicio de datos utiliza las clases de JavaCV mostradas en la tabla 26:

**Tabla 26**

*Clases de JavaCV en el microservicio de datos*

| Importación | Uso |
|---|---|
| FFmpegFrameGrabber | Capturar frames de video/audio |
| FFmpegFrameRecorder | Grabar o codificar frames |
| Frame | Representar un frame |
| FrameGrabber | Base para capturar frames |
| Java2DFrameConverter | Convertir a/desde BufferedImage |
| avcodec | Constantes de codecs FFmpeg |

La dependencia javacv-platform incluye binarios nativos para todas las plataformas soportadas por la librería, lo que elevaba el peso del artefacto generado a aproximadamente 1 GB. Se reemplazó por tres dependencias específicas (javacv, ffmpeg y javacpp), limitando los binarios incluidos a los estrictamente requeridos por el microservicio. De forma complementaria, se actualizaron las importaciones en las clases que utilizan JavaCV para referenciar únicamente los seis módulos efectivamente utilizados.

**7.2.1.2 Optimización del Dockerfile.** Se refactorizó el Dockerfile del microservicio de datos aplicando tres mejoras principales sobre la construcción anterior: uso de un multi-stage build, caché de dependencias Maven y reducción de la imagen de runtime de JDK a JRE. A continuación se documentan cada uno de estos cambios.

Cambio 1 — Imagen de runtime: JDK a JRE. La imagen de runtime original utilizaba eclipse-temurin:21-jdk-jammy, que incluye el compilador y las herramientas de desarrollo de Java. En

producción estas herramientas no son necesarias: lo único requerido es el entorno de ejecución, es decir, la máquina virtual que interpreta y ejecuta el artefacto compilado. Se reemplazó por eclipse-temurin:21-jre-jammy, que contiene exclusivamente ese entorno, resultando en una imagen significativamente más liviana.

Cambio 2 — Caché de dependencias Maven. El Dockerfile original ejecutaba mvn clean package sin caché, descargando todas las dependencias en cada construcción. Se introdujo un stage dedicado a la resolución de dependencias aprovechando la instrucción --mount=type=cache, separando el proceso en dos pasos: primero la resolución de dependencias sobre los archivos pom.xml, y luego la compilación del artefacto. De este modo, Maven solo descarga dependencias nuevas o modificadas entre construcciones.

Cambio 3 — Instalación mínima de ffmpeg. La instalación original de ffmpeg no restringía los paquetes adicionales que apt instala por defecto junto con una dependencia: documentación, utilidades sugeridas y paquetes opcionales que no son requeridos para el funcionamiento del sistema. Se agregó la bandera --no-install-recommends para limitar la instalación estrictamente a ffmpeg y sus dependencias esenciales, y se eliminó la caché de apt tras la instalación para no incrementar el tamaño de la imagen.

Como resultado de estas tres optimizaciones, el tamaño de la imagen se redujo de aproximadamente 2 GB a 700 MB, lo que se traduce directamente en tiempos de construcción y distribución de la imagen menores. Ver Apéndices [B,C] para los Dockerfiles completos antes y después de la optimización.

### *7.2.2 Exposición de endpoints de monitoreo y health checks*

La dependencia spring-boot-starter-actuator se agregó a ambos microservicios, habilitando los endpoints de salud y métricas requeridos por la estrategia de monitoreo. La configuración habilita tres endpoints: /actuator/health para verificación de disponibilidad, /actuator/info para metadatos del servicio, y /actuator/prometheus para la exposición de métricas en formato compatible con Prometheus. El tag application permite identificar el origen de las métricas en las consultas PromQL.

## 7.3 Instrumentación de servicios para monitoreo y observabilidad

A partir de los hallazgos identificados en la sección 7.1.2, se realizaron las siguientes adaptaciones en los microservicios de datos y administración para habilitar la recolección de métricas, trazas distribuidas y logs estructurados.

### *7.3.1 Integración de dependencias de monitoreo*

Se agregaron las siguientes dependencias a ambos microservicios, se muestra en la tabla 27:

**Tabla 27**

*Dependencias integradas para monitoreo*

| Dependencia | Versión | Propósito |
|---|---|---|
| spring-boot-starter-actuator | Gestionada por Spring Boot BOM | Exposición de endpoints de salud y métricas para health checks |

| Dependencia | Versión | Propósito |
|---|---|---|
| micrometer-registry-prometheus | Gestionada por Spring Boot BOM | Exportación de métricas en formato Prometheus (scraping por Prometheus) |
| opentelemetry-javaagent | 2.11.0 | Instrumentación automática para trazas distribuidas |
| micrometer-tracing-bridge-otel | Gestionada por Spring Boot BOM | Puente entre Micrometer y OpenTelemetry para emisión de exemplars |

La adición de micrometer-tracing-bridge-otel habilitó la emisión de exemplars en las métricas de Prometheus: puntos de muestra individuales que llevan consigo el identificador de la traza que los originó, permitiendo navegar directamente desde un valor de latencia hacia la solicitud concreta que lo produjo en Tempo. El muestreo se configuró mediante la propiedad management.tracing.sampling.probability=1.0 en el archivo de propiedades de cada servicio.

### *7.3.2 Configuración de OpenTelemetry*

Se incorporó el agente de instrumentación automática de OpenTelemetry Java (versión 2.11.0) en el Dockerfile de ambos microservicios. El agente se descarga en tiempo de construcción desde el repositorio oficial y se activa mediante la bandera -javaagent en el punto de entrada del contenedor, sin requerir modificaciones en el código fuente ni dependencias adicionales en el pom.xml. En la figura 7 se muestra un ejemplo de la configuración del agente en un Dockerfile.

**Figura 7**

*Configuración OpenTelemetry en Dockerfile*

```
ARG OTEL_AGENT_VERSION=2.11.0
ADD
https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases/downlo
ad/v${OTEL_AGENT_VERSION}/opentelemetry-javaagent.jar \
    /app/opentelemetry-javaagent.jar

ENTRYPOINT ["java", \
    "-XX:+UseContainerSupport", \
    "-XX:MaxRAMPercentage=75.0", \
    "-javaagent:/app/opentelemetry-javaagent.jar", \
    "-jar", "app.jar"]
```

La configuración del endpoint de exportación de trazas se realiza mediante variables de entorno definidas en los manifiestos de despliegue, documentadas en la sección 8.

### *7.3.3 Implementación de logging estructurado*

Se implementó logging estructurado en ambos microservicios mediante dos cambios complementarios: la adición de logs SLF4J en controllers y services, y la configuración de un archivo logback-spring.xml. Este archivo define el formato de salida e incluye en cada línea el trace_id y el span_id: el primero identifica de forma única una solicitud a lo largo de todos los servicios que la procesan; el segundo identifica una operación específica dentro de esa solicitud. De forma complementaria, se configuró el patrón de nivel de log en los archivos de propiedades de cada servicio para incluir los mismos identificadores. Los patrones de configuración completos se encuentran en el Apéndice E.

Como todos los logs de una misma solicitud comparten el mismo trace_id, es posible desde Loki filtrar por ese identificador y reconstruir exactamente qué ocurrió en el sistema, sin importar

cuántos servicios la procesaron. Los niveles de log se distribuyeron por capa siguiendo una convención uniforme: INFO en el controller para registrar cada operación recibida con sus parámetros, DEBUG en el service para detallar la ejecución interna, WARN para condiciones anómalas sin ser errores fatales, y ERROR para excepciones que interrumpen el flujo normal.

# 8. Implementación de estrategia integrada

Este capítulo describe la implementación de la estrategia integrada de despliegue, monitoreo y observabilidad establecida en la sección 6, aplicada sobre los servicios adaptados en la sección 7. Se configuraron e implementaron los mecanismos de despliegue, los sistemas de monitoreo y observabilidad, y se realizó la puesta en marcha de los servicios sobre la nueva infraestructura.

## 8.1 Implementación de los mecanismos de despliegue

El diseño de la arquitectura de despliegue contempló dos configuraciones según el entorno de uso y los recursos disponibles. Cada configuración responde a necesidades distintas y puede emplearse de forma independiente como se muestra en la tabla 28.

**Tabla 28**

*Configuraciones según el entorno*

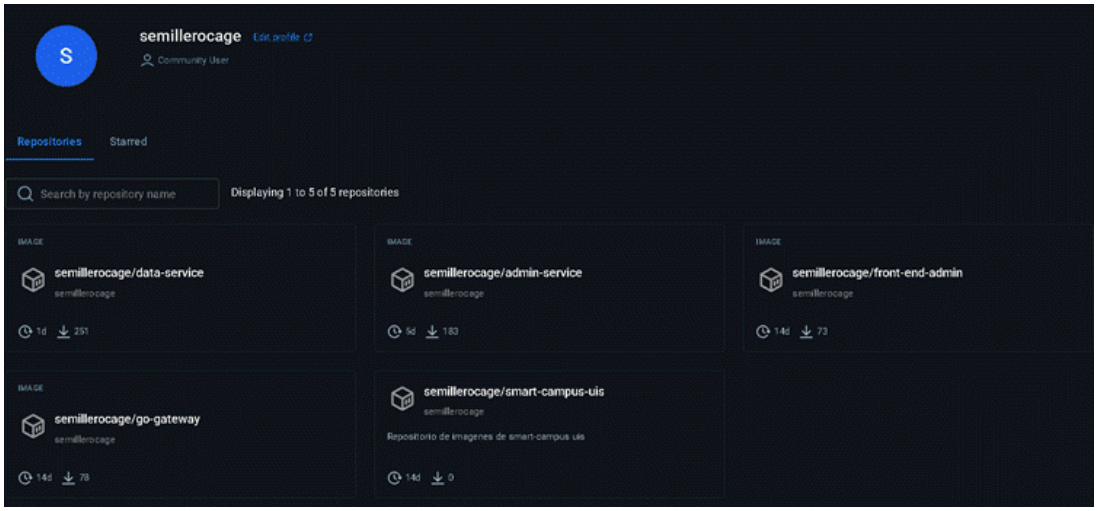
| Configuración | Descripción | Contexto de uso |
|---|---|---|
| Docker Compose | Despliegue simple con imágenes publicadas en DockerHub. Sin observabilidad. | Entornos de desarrollo, recursos limitados, despliegue rápido sin infraestructura adicional. |
| K3s | Despliegue completo con orquestación, Rolling Update, monitoreo y observabilidad. | Entorno de producción o validación completa de la estrategia integrada. |

### *8.1.1 Registro de imágenes en DockerHub*

Para centralizar la gestión de imágenes del proyecto, se creó una cuenta en DockerHub a nombre del semillero CAGE desde la cual se publican todas las imágenes de Smart Campus UIS, garantizando su acceso a ellas, independientemente de quién las construya pues son publicas. La figura 8 muestra la creación de la cuenta de Docker Hub así como los repositorios.

**Figura 8**

*Cuenta de DockerHub semillero CAGE*




### *8.1.2 Configuración Docker Compose*

Docker Compose permite levantar el sistema completo con un único comando, sin requerir un orquestador ni infraestructura adicional. Esta configuración está orientada a entornos donde la complejidad de Kubernetes no está justificada: desarrollo local, pruebas funcionales o máquinas con recursos limitados. Al no incorporar orquestación, tampoco incluye las capacidades de monitoreo y observabilidad implementadas en este trabajo. La configuración toma como base el archivo existente en el repositorio y lo adapta para referenciar las imágenes publicadas en DockerHub, eliminando la necesidad de construirlas localmente. Las credenciales de todos los servicios se centralizan en un archivo .env que el operador configura antes del despliegue. Ver Apéndice D para el docker-compose.yml completo.

### *8.1.3 Gestión del despliegue con Helm*

Se eligió Helm como gestor de despliegue por tres razones principales. La portabilidad permite desplegar un chart en cualquier clúster K3s sin modificar los manifiestos, adaptando únicamente el archivo de valores al entorno de destino. El control selectivo de componentes permite levantar únicamente los necesarios en cada momento, reduciendo el consumo de recursos y excluyendo servicios que ya existen en el entorno de destino. Los perfiles de configuración permiten superponer múltiples archivos de valores al momento del despliegue, habilitando configuraciones distintas sin duplicar manifiestos ni mantener variantes manuales.

Toda la configuración parametrizable se centraliza en el archivo values.yaml, que actúa como fuente de verdad base del chart. La Figura 9 ilustra la organización típica de un chart, que sirvió como referencia para la estructura adoptada en Smart Campus UIS.

**Figura 9**

*Estructura de un Helm chart*

```
smart-campus/
├── Chart.yaml
├── Chart.lock
├── values.yaml
├── values-custom.yaml
├── charts/
└── templates/
    ├── 00-namespace.yaml
    ├── 01-data.yaml
    └── 02-admin.yaml
```

El archivo values-custom.yaml contiene únicamente los parámetros que difieren del base; Helm fusiona ambos al momento del despliegue. Para Smart Campus UIS este mecanismo se utiliza para habilitar o deshabilitar componentes según el entorno, y para alternar entre imágenes publicadas en

DockerHub (imageSource: remote) e imágenes construidas localmente (imageSource: local). Ver Apéndice F para el values-custom.yaml completo.

### *8.1.4 Deployment y StatefulSet*

Los microservicios de administración y datos se desplegaron como Deployments, el recurso de Kubernetes para aplicaciones sin estado: sus pods son intercambiables y toda la persistencia reside en los sistemas de almacenamiento externos, lo que los hace aptos para este tipo de recurso. El Deployment además soporta nativamente la estrategia Rolling Update definida en la sección 6. Las bases de datos y brokers (PostgreSQL, MongoDB, InfluxDB, MinIO, EMQX y RabbitMQ) se desplegaron como StatefulSets, que a diferencia del Deployment asigna a cada pod una identidad de red estable y un volumen de almacenamiento propio. Aunque en la configuración actual se trabaja con una sola réplica por componente, se eligió StatefulSet sobre Deployment precisamente porque deja preparada la arquitectura para escalar horizontalmente en el futuro sin necesidad de migrar el tipo de recurso. La tabla 29 muestra el componente junto con el tipo de recurso asignado.

**Tabla 29**

*Tipo de recurso de Kubernetes por componente para despliegue*

| Componente | Tipo de recurso | Justificación |
|---|---|---|
| Frontend de administración | Deployment | Sin estado, instancias intercambiables |
| Microservicio de administración | Deployment | Sin estado, escala horizontalmente |
| Microservicio de datos | Deployment | Sin estado, escala horizontalmente |

| Componente | Tipo de recurso | Justificación |
|---|---|---|
| Ingress (Traefik) | Ingress Controller | Enrutamiento nativo de K3s, reemplaza el API Gateway |
| PostgreSQL | StatefulSet | Base de datos relacional, requiere persistencia e identidad estable |
| MongoDB | StatefulSet | Base de datos documental, requiere persistencia |
| InfluxDB | StatefulSet | Base de datos de series temporales, requiere persistencia |
| MinIO | StatefulSet | Almacenamiento de objetos, requiere persistencia |
| EMQX | StatefulSet | Broker MQTT, requiere identidad de red estable |
| RabbitMQ | StatefulSet | Broker AMQP, requiere identidad de red estable |

### *8.1.5 Services e Ingress*

La comunicación entre componentes se gestiona mediante Services, que exponen cada pod bajo una dirección estable dentro del clúster. Para los microservicios se utiliza ClusterIP (acceso interno al clúster únicamente); para las bases de datos y brokers, un Service headless que provee resolución DNS directa a cada pod individual, dándole a cada instancia una identidad de red estable. El acceso externo, originalmente gestionado por un microservicio Gateway escrito en Go, se delegó al IngressRoute de Traefik (Traefik Labs, 2024), el Ingress Controller incluido por defecto en K3s. Dado

que el Gateway original se limitaba a enrutar peticiones por prefijo de path hacia los microservicios de administración y datos, su comportamiento se replicó exactamente mediante dos reglas en el IngressRoute, eliminando un pod innecesario y delegando el enrutamiento a infraestructura nativa del orquestador.

### *8.1.6 Gestión de credenciales*

Las credenciales de acceso a bases de datos y brokers se centralizan en un recurso Secret de Kubernetes denominado db-credentials. En la configuración actual, los valores provienen directamente del values.yaml, por lo que el Secret no agrega seguridad real. Su valor es arquitectural: actúa como capa de indirección entre la fuente de las credenciales y el Deployment, que siempre las consume desde el Secret sin importar su origen. Esto significa que en el futuro la fuente puede cambiar, ya sea un archivo no versionado pasado con -f, External Secrets Operator o un Secret creado manualmente antes del despliegue, sin realizar modificaciones significativas a ningún manifiesto del Deployment.

### *8.1.7 Arranque y salud de los servicios*

Cada Deployment incorpora dos mecanismos complementarios para garantizar la disponibilidad de los servicios. El primero son los initContainers: contenedores que se ejecutan antes del arranque del servicio principal y verifican que cada dependencia esté disponible, evitando que un microservicio intente conectarse a una base de datos o broker que aún no está listo. El segundo son los health checks (livenessProbe y readinessProbe), ambos apuntando al endpoint /actuator/health: el primero reinicia el pod si detecta que el servicio dejó de responder; el segundo lo saca de la rotación del Service sin reiniciarlo si aún no está listo para recibir tráfico. Esta distinción es relevante para el Rolling Update:

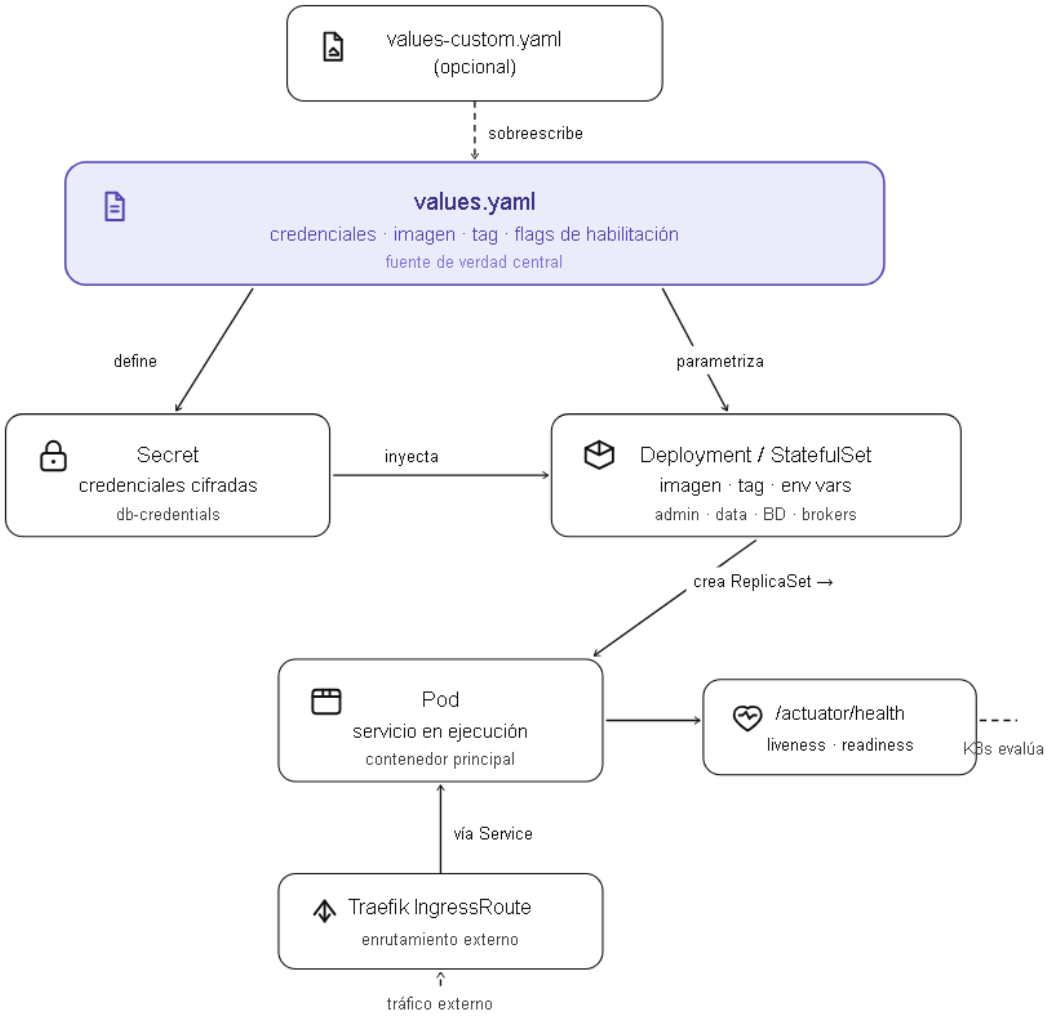
Kubernetes no redirige tráfico a un pod nuevo hasta que su readinessProbe responde satisfactoriamente, garantizando que el servicio permanezca disponible durante toda la transición.

### *8.1.8 Flujo de configuración*

La figura 10 sintetiza cómo se articula la configuración de un componente en K3s: desde el values.yaml como fuente de verdad central, pasando por el Secret y el Deployment, hasta el Pod en ejecución con sus health checks.

**Figura 10**

*Diagrama de configuración de un Helm chart*




## 8.2 Implementación del sistema de monitoreo

El sistema de monitoreo recolecta métricas desde los servicios instrumentados hacia Prometheus, con visualización centralizada en Grafana. Grafana Alloy actúa como colector intermediario, descubriendo automáticamente los pods a monitorear y enviando las métricas

recolectadas a Prometheus mediante remote_write. El stack completo se despliega como una unidad mediante el chart smart-campus-monitoring, que agrupa Prometheus, Grafana y Alloy cómo dependencias coordinadas.

### *8.2.1 Métricas y alertas*

De las métricas disponibles a través del endpoint /actuator/prometheus, se seleccionaron las más relevantes para el monitoreo operacional de la plataforma, priorizando indicadores de disponibilidad, consumo de recursos y comportamiento bajo carga (incluyendo latencia en el percentil 95), y definiendo umbrales de alerta para cada una. La Tabla 30 consolida las métricas seleccionadas con sus umbrales y propósito, y la Tabla 33 describe los paneles del dashboard Smart Campus - Alertas y rendimiento. Los paneles correspondientes a estas métricas se describen en la sección 8.4.2.

**Tabla 30**

*Métricas relevantes y alertas*

| Métrica | Servicio origen | Umbral de alerta | Propósito |
|---|---|---|---|
| Disponibilidad del servicio (up) | admin, data | Pod caído por más de 1 minuto | Detectar cuando un microservicio deja de responder o desaparece del clúster |
| Uso de CPU del proceso Java (process_cpu_usage) | admin, data | Mayor al 80% por 5 minutos | Detectar carga sostenida que pueda afectar el rendimiento |

| Métrica | Servicio origen | Umbral de alerta | Propósito |
|---|---|---|---|
| Uso de heap JVM (jvm_memory_used_bytes) | admin, data | Mayor al 90% por 5 minutos | Detectar presión de memoria o posibles memory leaks |
| Tasa de errores HTTP 4xx/5xx | admin, data | Mayor a 0.1 req/s por 2 minutos | Identificar endpoints con fallos recurrentes |
| Latencia P95 (http_server_requests_seconds) | admin, data | Mayor a 500ms por 5 minutos | Detectar degradación sostenida en el tiempo de respuesta de los servicios |

### *8.2.2 Despliegue del stack*

El stack de monitoreo se despliega mediante el chart smart-campus-monitoring, que declara como dependencias los charts oficiales de Prometheus, Grafana y Grafana Alloy. Un único comando de Helm levanta todos los componentes de forma coordinada, sin necesidad de gestionar cada herramienta de forma independiente. Los microservicios de administración y datos exponen métricas en formato Prometheus a través del endpoint /actuator/prometheus, habilitado mediante las adaptaciones descritas en la sección 7, que constituyen el prerrequisito para el scraping automático de métricas desde Alloy.

### *8.2.3 Grafana Alloy*

Grafana Alloy actúa como colector centralizado de métricas mediante un pipeline de tres etapas: descubrimiento de pods, filtrado por anotaciones y envío a Prometheus. En la etapa de descubrimiento, Alloy consulta la API de Kubernetes para obtener todos los pods activos en el clúster.

A continuación filtra únicamente los pods del namespace smart-campus que declaran la anotación prometheus.io/scrape: "true", leyendo el puerto y el path de scraping desde las propias anotaciones del pod. Esto permite que cada microservicio declare sus propios valores sin modificar la configuración de Alloy. Las métricas recolectadas se envían a Prometheus mediante remote_write. La Figura 11 muestra una versión simplificada del pipeline para ilustrar su estructura; la configuración completa se encuentra en el Apéndice H.

**Figura 11**

*Configuración de recolección de métricas de Grafana Alloy*

```
discovery.relabel "spring_metrics" {
  targets = discovery.kubernetes.pods.targets
  rule {
    source_labels = ["__meta_kubernetes_namespace"]
    regex         = "smart-campus"
    action        = "keep"
  }
  // más reglas de filtrado...
}

prometheus.scrape "spring" {
  targets       = discovery.relabel.spring_metrics.output
  forward_to    = [prometheus.remote_write.local.receiver]
  scrape_interval = "30s"
}

prometheus.remote_write "local" {
  endpoint {
    url = "http://prometheus.monitoring.svc.cluster.local/api/v1/write"
  }
}
```

### *8.2.4 Prometheus*

Prometheus se configuró para recibir métricas mediante remote_write desde Alloy, con retención de 15 días sobre un PersistentVolume de 8 GiB. Los componentes del chart que no son necesarios en este contexto (Pushgateway, kube-state-metrics y node-exporter) se deshabilitaron para reducir el consumo de recursos, dado que la recolección de métricas se delega completamente a Alloy. La Figura 12 muestra una versión simplificada de la configuración para ilustrar los componentes habilitados; la configuración completa se encuentra en el Apéndice H.

**Figura 12**

*Configuración de Prometheus*

```yaml
prometheus:
  server:
    retention: "15d"
    persistentVolume:
      enabled: true
  alertmanager:
    enabled: true
  pushgateway:
    enabled: false
  kube-state-metrics:
    enabled: false
  node-exporter:
    enabled: false
```

### *8.2.5 Grafana*

Grafana se desplegó con el datasource de Prometheus preconfigurado mediante el values.yaml del chart, de modo que no requiere configuración manual después del despliegue. Esto garantiza que

cualquier reinstalación del stack reproduce el mismo estado sin intervención adicional. La configuración completa del datasource se encuentra en el Apéndice H.

### *8.2.6 Alertas*

Las cuatro alertas definidas en la Tabla 30 se implementaron de forma declarativa en el values.yaml del chart mediante el campo serverFiles, de modo que Prometheus las carga automáticamente al desplegarse sin configuración adicional. Prometheus evalúa las reglas cada 30 segundos y cuando una alerta se dispara, Alertmanager la agrupa por nombre y servicio antes de hacerla visible en el panel de alertas activas del dashboard Smart Campus - Alertas y rendimiento. Las expresiones PromQL completas de cada alerta se encuentran en el Apéndice I.

## 8.3 Implementación del sistema de observabilidad

El sistema de observabilidad recolecta dos señales complementarias: trazas distribuidas y logs estructurados. Las trazas permiten reconstruir el camino completo de una solicitud a través de los servicios; los logs registran los eventos relevantes de cada operación con el identificador de traza incluido. Ambas señales convergen en Grafana, donde pueden correlacionarse para diagnosticar problemas con precisión.

### *8.3.1 Grafana Alloy*

Grafana Alloy actúa como colector unificado de telemetría, centralizando en un único proceso la recolección de logs y trazas. Al ser la distribución oficial de Grafana Labs del OpenTelemetry Collector,

es compatible de forma nativa con los backends del stack LGTM sin adaptadores adicionales, lo que simplifica la arquitectura respecto a mantener agentes separados por señal.

### *8.3.2 Loki*

Loki actúa como backend de almacenamiento de logs. Se desplegó en modo SingleBinary, donde todos sus componentes internos corren en un único pod, apropiado para un clúster de un nodo. Los logs se almacenan en una instancia de MinIO dedicada al stack de observabilidad, independiente de la instancia de MinIO de la aplicación, garantizando que el almacenamiento de logs no compita con los datos de la plataforma. La retención se configuró en 7 días; sin esta configuración los logs crecen indefinidamente. La creación del bucket en MinIO se automatizó mediante un initContainer que garantiza su disponibilidad antes de que Loki arranque, sin requerir intervención manual. La configuración completa se encuentra en el Apéndice J.

### *8.3.3 Trazas distribuidas*

Las operaciones instrumentadas para trazas y los spans que el agente genera automáticamente se detallan en la Tabla 31. El agente OTel captura cada operación en el microservicio, genera un span con su duración y estado, y lo envía a través de Alloy hasta Tempo, donde queda disponible para consulta desde Grafana.

**Tabla 31**

*Operaciones instrumentadas para trazas*

| Operación | Servicio | Spans generados |
|---|---|---|
| Petición HTTP entrante (REST) | data, admin | HTTP request, procesamiento del controller, respuesta HTTP |
| Consumo de mensaje MQTT | data | Recepción del mensaje, deserialización, escritura en MongoDB, escritura en InfluxDB |
| Consumo de mensaje AMQP | data | Recepción del mensaje, deserialización, escritura en repositorios |
| Reencolado de mensaje | data | Búsqueda de aplicaciones suscritas, publicación MQTT, publicación AMQP |
| Operación de almacenamiento | data, admin | Escritura o lectura en MongoDB, InfluxDB o MinIO (data) / escritura o lectura en PostgreSQL (admin) |

Antes de reenviar los spans a Tempo, Alloy aplica un filtro de tail sampling que descarta las trazas cuya ruta HTTP corresponde a los endpoints del actuator (/actuator/health, /actuator/prometheus). Esta decisión responde a dos razones: las trazas del actuator no aportan valor diagnóstico en flujos de negocio, y su volumen genera un aumento notable en el consumo de memoria de Tempo en entornos con recursos limitados. La configuración completa se encuentra en el Apéndice J.

Tempo se desplegó en modo monolítico con almacenamiento local en un PersistentVolume de 5 GiB y retención de 72 horas. La Figura 13 ilustra la configuración de retención y almacenamiento; la configuración completa se encuentra en el Apéndice J.

**Figura 13**

*Configuración de Tempo*

```
tempo:
  tempo:
    retention: 72h
    storage:
      trace:
        backend: local
        local:
          path: /var/tempo/traces
        block:
          version: vParquet4
  persistence:
    enabled: true
    storageClassName: "local-path"
    size: 5Gi
```

### *8.3.4 Logs estructurados*

Los logs estructurados implementados siguen el esquema de niveles detallado en la Tabla 32. Cada log escrito en stdout incluye el trace_id y span_id de la solicitud en curso; Alloy los recolecta limitándose a los namespaces smart-campus y monitoring, y los envía a Loki etiquetados con el nombre del servicio. Loki corre en modo SingleBinary con retención de 7 días, almacenando en una instancia de MinIO dedicada e independiente de la de la aplicación. La configuración completa se encuentra en el Apéndice J.

**Tabla 32**

*Logs implementados*

| Nivel | Contexto de uso | Campos incluidos |
|---|---|---|
| INFO | Operaciones completadas correctamente: entrada de peticiones, creación y actualización de entidades | timestamp, nivel, servicio, trace_id, span_id, operación, parámetros relevantes |
| WARN | Condiciones anómalas sin fallo: entidades no encontradas, estados inconsistentes | timestamp, nivel, servicio, trace_id, span_id, descripción de la condición |
| ERROR | Excepciones que interrumpen el flujo normal: fallos de conexión, errores de persistencia | timestamp, nivel, servicio, trace_id, span_id, mensaje de error, stack trace |

La correlación entre señales es el valor central del sistema: desde un log, el trace_id permite saltar directamente a la traza en Tempo; desde una traza, es posible filtrar los logs por ese mismo identificador en Loki, pasando de la detección de un problema a su diagnóstico sin cambiar de herramienta.

### *8.3.5 Consideraciones de configuración del stack*

El stack de monitoreo y observabilidad incluye configuraciones orientadas a garantizar que en escenarios de presión de recursos la plataforma tenga prioridad sobre las herramientas de monitoreo, y a dejar preparada la arquitectura para incorporar infraestructura dedicada en el futuro sin modificar los manifiestos. La configuración completa se encuentra en el Apéndice J.

## 8.4 Dashboards

Los dashboards de la plataforma se organizan en dos grupos según su propósito: monitoreo operacional y observabilidad. Todos se definen de forma declarativa como ConfigMaps de Kubernetes dentro del chart, y Grafana los descubre y carga automáticamente mediante un sidecar sin requerir reinicio ni intervención manual.

### *8.4.1 Configuración de Grafana para dashboards*

Cada dashboard se define como un ConfigMap de Kubernetes con la etiqueta grafana_dashboard: "1", que el sidecar de Grafana detecta automáticamente en el namespace monitoring. Esto garantiza que cualquier reinstalación del stack reproduce el mismo conjunto de dashboards sin intervención adicional, y que agregar un nuevo dashboard es tan simple como añadir un ConfigMap con esa etiqueta. La configuración completa del mecanismo de descubrimiento se encuentra en el Apéndice K.

### *8.4.2 Dashboards de monitoreo*

El dashboard Smart Campus — Alertas y rendimiento centraliza el estado operacional de los microservicios en una sola pantalla, permitiendo detectar rápidamente condiciones anómalas en disponibilidad, consumo de recursos, tasa de errores y latencia. Cubre las métricas y umbrales definidos en la Tabla 30, y sus paneles se detallan en la Tabla 33.

**Tabla 33**

*Paneles, dashboards de monitoreo*

| Panel | Tipo | Fuente | Descripción |
|---|---|---|---|
| Estado de servicios | stat | Prometheus | Indicador rojo/verde de disponibilidad por servicio |
| CPU del proceso Java | timeseries | Prometheus | Uso de CPU con umbrales warning 60% y crítico 80% |
| Heap JVM usado (%) | timeseries | Prometheus | Uso de heap con umbrales warning 70% y crítico 90% |
| Errores HTTP 4xx/5xx | timeseries | Prometheus | Tasa de errores agrupada por endpoint y código de estado |
| Alertas activas | alertlist | Alertmanager | Alertas en estado firing, pending o error |
| Latencia P95 | timeseries | Prometheus | Latencia P95 por endpoint con exemplars enlazados a Tempo |

### *8.4.3 Dashboards de Observabilidad*

Los dashboards de observabilidad cubren las distintas capas de la plataforma: rendimiento de microservicios, vista general del sistema, diagnóstico por servicio e infraestructura. Cada uno combina señales de Loki, Tempo y Prometheus según su propósito como se puede evidenciar en la tabla 34.

**Tabla 34**

*Dashboards para observabilidad*

| Dashboard | Descripción |
|---|---|
| Smart Campus - Overview | Vista general del sistema: volumen de logs, errores por servicio, distribución por pod |
| Smart Campus - Microservicios | Actividad, errores y logs de los microservicios admin y data |
| Smart Campus - Infraestructura | Actividad y logs de bases de datos y brokers (PostgreSQL, MongoDB, InfluxDB, MinIO, EMQX, RabbitMQ) |
| Smart Campus - Trazas y latencia | Trazas recientes por servicio, latencia por endpoint, tasa de peticiones |

**8.4.3.1 Smart Campus - Overview.** Este dashboard proporciona una vista general del sistema, orientada a detectar rápidamente anomalías en el volumen de logs y la distribución de errores entre servicios. Combina paneles de series temporales, gráficos de barras y un visor de logs para consolidar el estado del namespace smart-campus en una sola pantalla. La tabla 35 muestra una descripción más detallada de los paneles.

**Tabla 35**

*Paneles dashboard Smart Campus - Overview*

| Panel | Tipo | Fuente | Descripción |
|---|---|---|---|
| Volumen de logs por minuto | timeseries | Loki | Tasa de generación de logs y errores por minuto en el namespace smart-campus |
| Errores por servicio | barchart | Loki | Cantidad de errores agrupados por aplicación en el rango seleccionado |
| Distribución de logs por pod | piechart | Loki | Proporción de logs generados por cada pod en el rango seleccionado |
| Estadísticas rápidas | stat | Loki | Total de logs generados en el rango seleccionado |
| Errores totales | stat | Loki | Total de errores generados en el rango seleccionado |
| Actividad de logs por servicio | timeseries | Loki | Actividad apilada por aplicación a lo largo del tiempo |
| Últimos errores y warnings | logs | Loki | Logs recientes con nivel ERROR o WARN de todos los servicios |

**8.4.3.2 Smart Campus - Microservicios.** Este dashboard está orientado al diagnóstico de los microservicios de administración y datos. Permite comparar su actividad, identificar errores, detectar conexiones fallidas a dependencias y visualizar excepciones Java con sus stack traces completos. La tabla 36 muestra una descripción más detallada de los paneles.

**Tabla 36**

*Paneles dashboard Smart Campus - Microservicios*

| Panel | Tipo | Fuente | Descripción |
|---|---|---|---|
| Comparación de actividad: admin vs data | timeseries | Loki | Volumen de logs generados por cada microservicio a lo largo del tiempo |
| Errores en admin | stat | Loki | Total de errores del microservicio de administración en el rango seleccionado |
| Errores en data | stat | Loki | Total de errores del microservicio de datos en el rango seleccionado |
| Conexiones fallidas a bases de datos | stat | Loki | Intentos de conexión fallidos detectados en los logs de ambos microservicios |
| Reinicios detectados | stat | Loki | Logs que indican inicio de aplicación, indicativo de reinicios del servicio |
| Logs de admin | logs | Loki | Logs en tiempo real del microservicio de administración |
| Logs de data | logs | Loki | Logs en tiempo real del microservicio de datos |
| Excepciones Java / Stack Traces | logs | Loki | Logs con excepciones y stack traces de ambos microservicios Spring Boot |

| Panel | Tipo | Fuente | Descripción |
|---|---|---|---|
| Tasa de peticiones | timeseries | Prometheus | Volumen de tráfico por servicio en requests por segundo |
| Errores 5xx y 4xx | timeseries | Prometheus | Tasa de errores 5xx y 4xx por servicio |
| Latencia P95 | timeseries | Prometheus | Latencia P95 por endpoint con exemplars enlazados a Tempo |

**8.4.3.3 Smart Campus - Infraestructura.** Este dashboard cubre los servicios de infraestructura de la plataforma: bases de datos, brokers de mensajería y almacenamiento de objetos. Cada componente cuenta con un panel de logs propio con variables de filtrado por nivel de severidad, permitiendo acotar la visualización según el tipo de evento de interés. La tabla 37 muestra una descripción más detallada de los paneles.

**Tabla 37**

*Paneles dashboard Smart Campus - Infraestructura*

| Panel | Tipo | Fuente | Descripción |
|---|---|---|---|
| Volumen de logs por componente | barchart | Loki | Total de logs generados por cada servicio de infraestructura en el rango seleccionado |
| Errores en infraestructura | barchart | Loki | Errores por componente con umbrales de color según severidad |
| PostgreSQL - Logs | logs | Loki | Logs de PostgreSQL filtrables por nivel: error, warning, info, queries lentas y conexiones |

| Panel | Tipo | Fuente | Descripción |
|---|---|---|---|
| MongoDB - Logs relevantes | logs | Loki | Logs de MongoDB filtrables por nivel: error, warning, queries lentas, conexiones e info |
| RabbitMQ - Logs | logs | Loki | Logs de RabbitMQ filtrables por nivel: error, warning e info |
| EMQX (MQTT) - Logs | logs | Loki | Logs de EMQX filtrables por nivel: error, warning e info |
| InfluxDB - Logs | logs | Loki | Logs de InfluxDB filtrables por nivel: error, warning e info |
| MinIO - Logs | logs | Loki | Logs de MinIO filtrables por nivel: error, warning e info, excluyendo health checks |

**8.4.3.4 Smart Campus - Trazas y Latencia.** Este dashboard está orientado al diagnóstico de rendimiento de los microservicios. Combina datos de Tempo y Prometheus en cinco paneles. Las rutas del actuator se excluyen de los paneles de latencia y tasa de peticiones para reducir ruido visual, consistente con el filtro aplicado en Alloy al nivel de las trazas. La tabla 38 muestra una descripción más detallada de los paneles.

**Tabla 38**

*Paneles dashboard trazas y latencia*

| Panel | Tipo | Fuente | Descripción |
|---|---|---|---|
| Trazas recientes - admin | table | Tempo | Últimas 15 trazas del microservicio de administración |
| Trazas recientes - data | table | Tempo | Últimas 15 trazas del microservicio de datos |
| Latencia P95 por endpoint - admin | timeseries | Prometheus | Latencia P95 por endpoint excluyendo rutas de actuator |
| Latencia P95 por endpoint - data | timeseries | Prometheus | Latencia P95 por endpoint excluyendo rutas de actuator |
| Tasa de peticiones por servicio | timeseries | Prometheus | Volumen de tráfico por servicio en requests por segundo |

## 8.5 Migración de servicios

La adopción de K3s con Helm como mecanismo de despliegue implicó trasladar la definición del sistema de Docker Compose a los Helm charts desarrollados en la sección 8.1. Para validar que la nueva infraestructura funcionaba correctamente, el despliegue se realizó de forma gradual, levantando y verificando cada conjunto de componentes antes de proceder con el siguiente. Este orden no fue arbitrario: cada fase depende de que la anterior esté operativa, ya que los microservicios requieren las bases de datos y brokers, el Ingress requiere los microservicios, y el stack de observabilidad requiere que haya servicios corriendo para tener qué monitorear. La tabla 39 ilustra las fases de implementación.

**Tabla 39**

*Fases de implementación*

| Fase | Servicios | Validación | Resultado |
|---|---|---|---|
| 1 | PostgreSQL, MongoDB, InfluxDB, MinIO, EMQX, RabbitMQ | Pod en estado Running, livenessProbe y readinessProbe en OK, conexión desde microservicios confirmada | OK |
| 2 | Microservicio admin, microservicio data | /actuator/health en OK, /actuator/prometheus expuesto, métricas visibles en Grafana (up{namespace="smart-campus"} = 1) | OK |
| 3 | Frontend de administración, IngressRoute Traefik | Navegación en frontend operativa, peticiones enrutadas por Traefik a /admin y /data correctamente | OK |
| 4 | Loki, Tempo, Prometheus, Grafana Alloy, Grafana, Alertmanager, MinIO de observabilidad | Métricas JVM y HTTP en Prometheus, logs estructurados con trace_id en Loki, trazas distribuidas en Tempo, correlación log↔traza operativa en Grafana | OK |

Al completar la fase 4, la plataforma Smart Campus UIS se encontraba completamente operativa bajo el nuevo mecanismo de despliegue, con monitoreo y observabilidad activos sobre todos los servicios.

# 9. Evaluación de la estrategia

La evaluación de la estrategia integrada para la plataforma Smart Campus UIS persigue verificar que la estrategia de despliegue, monitoreo y observabilidad cumple con los requerimientos definidos en un entorno real de ejecución. Se aborda en cuatro componentes complementarios: la especificación del hardware sobre el cual se ejecutan las pruebas, el plan de prueba estructurado en siete Casos independientes (uno por dimensión de la estrategia integrada), la ejecución de dichos Casos mediante scripts reproducibles, y el análisis de los resultados obtenidos. La intención es demostrar que la estrategia integrada permite desplegar, monitorear y observar la plataforma Smart Campus UIS como una sola unidad operacional coherente, y que las decisiones tomadas durante el diseño se sostienen al confrontarlas con la operación real del sistema.

## 9.1 Hardware para la evaluación

La evaluación experimental se ejecuta sobre un único nodo en un entorno de desarrollo basado en Windows Subsystem for Linux 2 (WSL2), el cual proporciona un kernel Linux completo sobre el sistema operativo Windows. Esta configuración permite reproducir el comportamiento de un clúster Kubernetes minimalista (K3s) sobre hardware de uso general, manteniendo aislamiento respecto al sistema operativo anfitrión y conservando trazabilidad de las configuraciones aplicadas. La tabla 40 presenta las características del entorno utilizado durante la totalidad de las pruebas.

**Tabla 40**

*Características del hardware para el entorno de evaluación*

| Componente | Especificación |
|---|---|
| **Procesador** | AMD Ryzen 7 8745HX (16 núcleos lógicos, arquitectura x86_64) |
| **Memoria RAM disponible** | 15 GiB asignados al kernel WSL2 (≈14 GiB disponibles durante la operación) |
| **Almacenamiento** | Volumen ext4 de 1 TB para /, con 942 GiB libres (≈2% utilizado) |
| **Sistema operativo (anfitrión)** | Microsoft Windows 11 con WSL 2.6.3.0 (WSLg 1.0.71) |
| **Distribución Linux (WSL2)** | Ubuntu 24.04.4 LTS |
| **Versión del kernel** | 6.6.87.2-microsoft-standard-WSL2 |
| **Orquestador** | K3s v1.35.4 (single-node) |
| **Gestor de paquetes** | Helm v3.21.0 |

## 9.2 Plan de Prueba

El plan de prueba aplicado a la plataforma Smart Campus UIS se organiza en siete casos, cada uno orientado a validar una dimensión específica de la estrategia integrada definida en el capítulo anterior. La estrategia integrada articula los componentes de despliegue, monitoreo y observabilidad en un único chart Helm, y su validación se aborda en dos ámbitos complementarios: tres casos dedicados al despliegue del stack, que verifican la correcta instalación de la infraestructura, la estrategia de Rolling Update y Rollback, y el despliegue del stack de monitoreo, y cuatro casos dedicados a las capas funcionales del componente de monitoreo y observabilidad (logs de infraestructura, instrumentación de microservicios, correlación cross-signal y sistema de alertas), detallados en la Tabla 41. Esta

organización permite aislar las validaciones por capa, evitar dependencias circulares entre las pruebas y asociar cada caso con un conjunto de scripts ejecutables reproducibles, conservados en el repositorio del proyecto bajo el directorio tests.

La Figura 14 presenta la arquitectura de las cuatro capas evaluadas, junto con sus componentes y las relaciones de flujo de datos entre ellas. Las capas inferiores generan señales de telemetría (logs, métricas y trazas) que se almacenan en backends especializados (Loki, Tempo y Prometheus respectivamente), y se exponen al usuario final a través de Grafana. Las capas superiores construyen sobre las inferiores: la Capa 3 establece relaciones entre las tres señales mediante exemplars y derived fields, mientras que la Capa 4 utiliza las métricas recolectadas por la Capa 2 para evaluar reglas de alerta. Los tres casos de despliegue, no representados en el diagrama, abarcan desde la comparativa de configuraciones hasta la instalación completa del stack mediante una única operación de Helm.

**Figura 14**

*Arquitectura de las cuatro capas de observabilidad evaluadas en el plan de prueba*

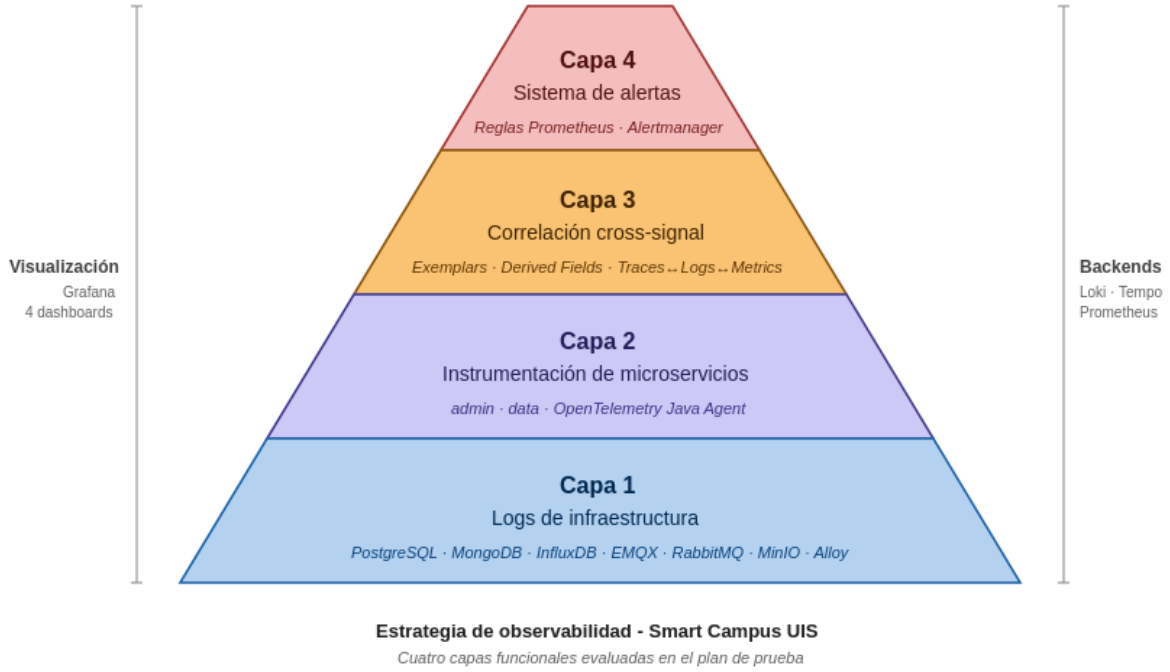



**Tabla 41**

*Resumen de los siete casos del plan de prueba*

| # | Caso | Objetivo de validación | Scripts asociados |
|---|---|---|---|
| 1 | Comparativa de configuraciones de despliegue | Verificar que las tres configuraciones se despliegan correctamente y comparar sus tiempos de arranque. | Comandos docker compose y helm. |

| # | Caso | Objetivo de validación | Scripts asociados |
|---|---|---|---|
| 2 | Rolling Update y Rollback | Validar que Kubernetes reemplaza instancias de forma gradual sin interrumpir el servicio y que el rollback restaura la versión anterior | Comandos kubectl y helm. |
| 3 | Despliegue del stack | Verificar que el chart Helm se instala correctamente y que los pods del stack quedan en estado Running. | Comandos kubectl y helm. |
| 4 | Capa 1 — Logs de infraestructura | Validar que cada componente de infraestructura emite logs, que estos llegan a Loki y que son visibles en los dashboards de Grafana. | test-postgres-logs.sh, test-mongo-logs.sh, test-emqx-logs.sh, test-influxdb-logs.sh, test-minio-logs.sh, test-rabbitmq-logs.sh, test-error-induction.sh. |
| 5 | Capa 2 — Instrumentación de microservicios | Validar que los microservicios admin y data generan trazas, métricas y logs, y que el trace_id se propaga al MDC y a Loki. | test-admin-trace.sh, test-data-trace.sh. |
| 6 | Capa 3 — Correlación cross-signal | Validar el cableado bidireccional entre logs, trazas y métricas mediante exemplars, derived fields, traces-to-logs y traces-to-metrics. | Validación visual sobre el dashboard Smart Campus — Tempo. No requiere scripts ejecutables independientes. |

| # | Caso | Objetivo de validación | Scripts asociados |
|---|---|---|---|
| 7 | Capa 4 — Sistema de alertas | Validar el ciclo completo inactive → pending → firing → inactive de las cinco reglas de alerta configuradas. | test-alert-errores-http.sh, test-alert-latencia-p95.sh, test-alert-memoria-heap.sh, test-alert-cpu-alta.sh, test-alert-servicio-caido.sh. |

### *9.2.1 Caso 1: Comparativa de configuraciones de despliegue*

**9.2.1.1 Infraestructura para el Caso 1.** Para la comparativa de configEste caso valida que las tres configuraciones de despliegue definidas en la estrategia (Docker Compose con build local, Docker Compose con imágenes de DockerHub y K3s con Helm) son funcionales y comparables en términos de tiempo de arranque. Para ello se requieren tres entornos limpios equivalentes: el primero con el código fuente del proyecto y Docker con BuildKit habilitado para la configuración con build local; el segundo con Docker y conectividad de salida hacia DockerHub para la configuración con imágenes publicadas; el tercero con un clúster K3s operativo con Helm 3.21 y conectividad de salida hacia los registros públicos de imágenes para la configuración con K3s.

**9.2.1.2 Plan de Pruebas para el Caso 1.** La prueba consiste en ejecutar cada configuración tres veces sobre un entorno limpio y reportar el promedio como tiempo representativo. La secuencia es la siguiente:

- Ejecutar el comando de arranque de cada configuración: docker compose up --build -d para build local, docker compose up -d para imágenes de DockerHub, y helm install smart-campus. -n smart-campus --create-namespace -f values-custom.yaml para K3s con Helm.

- Registrar el tiempo transcurrido desde el comando de arranque hasta que el último componente reporta estado operativo.
- Verificar que todos los componentes están disponibles y los endpoints de los microservicios responden correctamente.

Las condiciones de éxito son equivalentes en las tres configuraciones. La primera es que todos los contenedores o pods reporten estado operativo, verificable con docker compose ps o kubectl get pods -n smart-campus según corresponda. La segunda es que los endpoints de los microservicios respondan correctamente.ponda. La segunda es que los endpoints de los microservicios respondan correctamente.

### *9.2.2 Caso 2: Rolling Update y Rollback*

**9.2.2.1 Infraestructura para el Caso 2.** Este caso valida que la estrategia de actualización controlada funciona correctamente sobre K3s, verificando tanto el reemplazo gradual de instancias sin interrupción del servicio como la capacidad de revertir a una versión anterior ante un fallo. Requiere un clúster K3s con Smart Campus UIS desplegado y operativo, y dos versiones de la imagen del microservicio de datos publicadas en DockerHub.

**9.2.2.2 Plan de Pruebas para el Caso 2.** La prueba válida los dos mecanismos de gestión de versiones disponibles en la configuración K3s con Helm. La secuencia es la siguiente:

- Ejecutar helm upgrade smart-campus. -n smart-campus -f values-custom.yaml con el archivo de valores actualizado para referenciar una nueva versión de la imagen del microservicio de datos publicada en DockerHub.

- Monitorear el reemplazo gradual del pod mediante kubectl rollout status deployment/data -n smart-campus, verificando que el servicio permanece disponible durante toda la transición.
- Una vez completado el upgrade, verificar que el nuevo pod responde correctamente en /actuator/health.
- Ejecutar helm rollback smart-campus -n smart-campus y verificar que Kubernetes restaura la versión anterior manteniendo la disponibilidad del servicio durante la reversión.

Las condiciones de éxito son tres. La primera es que el upgrade no genere interrupción del servicio durante la transición. La segunda es que el pod de la nueva versión responda correctamente tras el upgrade. La tercera es que el rollback restaure satisfactoriamente la versión anterior sin interrupción del servicio.

### *9.2.3 Caso 3: Despliegue del stack de monitoreo*

**9.2.3.1 Infraestructura para el Caso 3.** Este caso valida que el stack de monitoreo y observabilidad se instala correctamente como una unidad mediante una única operación de Helm. Requiere el clúster K3s con Smart Campus UIS desplegado y operativo, y el namespace monitoring inexistente al iniciar la prueba, de modo que la operación de instalación pueda crearlo desde cero.

**9.2.3.2 Plan de Pruebas para el Caso 3.** La prueba consiste en instalar el chart smart-campus-monitoring sobre el clúster K3s y verificar las condiciones de éxito al cabo de un periodo razonable de espera. La secuencia es la siguiente:

- Ejecutar helm install monitoring. -n monitoring --create-namespace -f values-custom.yaml desde el directorio raíz del chart, proporcionando el archivo de valores específico del entorno.

- Esperar a que el conjunto completo de pods alcance el estado Running y verificar que todos sus contenedores están listos.

Las condiciones de éxito son dos. La primera es que todos los pods del namespace monitoring se encuentren en estado Running, comprobable con kubectl get pods -n monitoring. La segunda es que los servicios principales (Grafana, Prometheus, Loki y Tempo) sean accesibles a través del Ingress configurado en Traefik, verificable mediante una solicitud HTTP de prueba a la dirección externa del clúster.

### *9.2.4 Caso 4: Logs de infraestructura (Capa 1)*

**9.2.4.1 Infraestructura para el Caso 4.** Una vez completado el despliegue del stack, el caso de Capa 1 requiere que la totalidad de los componentes de infraestructura del namespace smart-campus se encuentre operacional. Estos componentes son PostgreSQL, MongoDB, InfluxDB, EMQX, RabbitMQ y MinIO. Cada uno de ellos se despliega como StatefulSet con su volumen persistente y su servicio asociado, manteniendo configuraciones por defecto representativas de un entorno de evaluación. La condición previa es que los pods correspondientes se encuentren en estado Running y que el componente Alloy del stack de monitorización esté efectivamente recolectando los logs del namespace smart-campus para enviarlos a Loki.

La infraestructura de validación para este Caso no requiere recursos adicionales más allá de los pods efímeros que cada script invoca para inducir eventos. Los scripts utilizan pods de tipo --rm --restart=Never con imágenes oficiales de cada componente (mongosh, psql, curlimages/curl, eclipse-mosquitto, quay.io/minio/mc y python:3.12-slim para RabbitMQ), de modo que cada inducción de evento se ejecuta en un contexto aislado, sin contaminar el estado de los pods principales del

namespace.

**9.2.4.2 Plan de Pruebas para el Caso 4.** El plan de pruebas del Caso 4 consta de siete scripts, organizados en dos grupos. El primer grupo, integrado por seis scripts (uno por componente de infraestructura), busca verificar la captura natural de logs en cada categoría de severidad: error, warning, info, queries lentas y conexiones, según aplique a cada componente. El segundo grupo, integrado por el script test-error-induction.sh, se enfoca específicamente en componentes silenciosos por diseño (MinIO, EMQX, MongoDB e InfluxDB) que no emiten errores en operación normal, induciendo errores reales mediante pods efímeros mal configurados para confirmar el vocabulario de severidad de cada uno.

Cada script del primer grupo sigue un patrón común estructurado en cuatro a cinco pruebas atómicas, según las características del componente:

- Una o varias pruebas de error real, induciendo condiciones que el componente registra con su término propio de severidad de error.
- Una o varias pruebas de warning real, mediante condiciones legítimas que disparan advertencias del componente (por ejemplo, la alarma de memoria en RabbitMQ o el takeover de client-id en EMQX).
- Pruebas de info, frecuentemente provocadas mediante el reinicio del pod para generar mensajes de startup y shutdown.
- Cuando aplica, pruebas adicionales para queries lentas (MongoDB, PostgreSQL) y conexiones (todos los componentes con tráfico de cliente).

El proceso de validación de cada script consta de dos fases. En la primera fase se ejecuta el script y se revisa la salida cruda en consola para confirmar que el componente emitió los términos de

severidad esperados (por ejemplo, ERROR: en PostgreSQL, [error] en EMQX, "s":"E" en MongoDB, FATAL en MinIO, lvl=error en InfluxDB). En la segunda fase se accede al dashboard correspondiente en Grafana, se selecciona el rango temporal que incluye la ejecución, y se verifica que los paneles filtrados por nivel muestren las entradas inducidas.

### *9.2.5 Caso 5: Instrumentación de microservicios (Capa 2)*

**9.2.5.1 Infraestructura para el Caso 5.** El Caso 5 requiere que los microservicios admin y data del namespace smart-campus se encuentren desplegados con las imágenes finales del proyecto (semillerocage/admin-service:1.4 y semillerocage/data-service:1.5), las cuales incluyen el OpenTelemetry Java Agent versión 2.11.0 y el bridge micrometer-tracing-bridge-otel. Esta configuración garantiza que cada solicitud HTTP recibida por los microservicios genere automáticamente un span en formato OTLP, una métrica HTTP en formato Prometheus y una entrada de log con el trace_id propagado a través del Mapped Diagnostic Context (MDC) de Logback.

La configuración del archivo logback-spring.xml de ambos microservicios incluye el patrón %X{trace_id:-} y %X{span_id:-}, utilizando la notación con guion bajo característica del Java Agent de OpenTelemetry. La probabilidad de muestreo de trazas se establece en 1.0 (muestreo completo) mediante la propiedad management.tracing.sampling.probability, lo cual es apropiado para un entorno de evaluación pero requeriría ajustarse a la baja en producción.

**9.2.5.2 Plan de Pruebas para el Caso 5.** El plan se compone de dos scripts simétricos, uno por microservicio (test-admin-trace.sh y test-data-trace.sh). Cada script realiza un conjunto de solicitudes HTTP a través de Traefik con un encabezado traceparent generado por el cliente, siguiendo la especificación W3C Trace Context. El objetivo es validar siete condiciones de instrumentación

correcta:

- El microservicio responde a las solicitudes HTTP entregadas por el ingress (Traefik).
- El microservicio honra el traceparent inyectado por el cliente, propagando el trace_id a los spans generados.
- El microservicio genera un trace_id propio cuando no recibe encabezado traceparent.
- El trace_id se propaga al MDC y, en consecuencia, queda registrado en los logs enviados a Loki.
- La traza generada queda almacenada en Tempo, verificable mediante consulta a la API /api/traces/{id}.
- Las métricas HTTP del microservicio (http_server_requests_seconds_count) son capturadas por Prometheus, agrupadas por uri, method y status.
- Los spans incluyen las llamadas descendentes a las bases de datos correspondientes (PostgreSQL para admin, MongoDB e InfluxDB para data), demostrando la instrumentación automática de JDBC, MongoDB Driver y el cliente HTTP del agente OTel.

El proceso de validación para cada microservicio consta de tres pasos automáticos y tres pasos manuales. Los pasos automáticos verifican mediante kubectl y consultas a las APIs de Tempo y Prometheus la presencia del trace_id en los logs del pod, la disponibilidad de la traza en Tempo y la captura de las métricas HTTP. Los pasos manuales corresponden a la verificación visual en Grafana: una consulta LogQL sobre Loki filtrada por el trace_id de la ejecución, una consulta directa a Tempo por el mismo trace_id para visualizar el árbol completo de spans, y la verificación de los paneles del dashboard de microservicios para confirmar la presencia de las métricas RED (Rate, Errors, Duration) generadas.

### *9.2.6 Caso 6: Correlación cross-signal (Capa 3)*

**9.2.6.1 Infraestructura para el Caso 6.** El Caso 6 se construye sobre el Caso anterior y requiere las mismas condiciones previas: microservicios instrumentados con el OpenTelemetry Java Agent y backends de telemetría operativos. Adicionalmente requiere que las configuraciones de cableado cross-signal estén presentes en el datasource provisioning de Grafana, lo cual incluye cuatro mecanismos específicos.

El primer mecanismo es la configuración exemplarTraceIdDestinations en el datasource Prometheus, con name: trace_id en notación snake_case y datasource Tempo como destino. El segundo es la configuración derivedFields en el datasource Loki, con una expresión regular que extrae el trace_id de las líneas de log (\\[[\\w-]+,([a-fA-F0-9]{16,32}),). El tercer y cuarto mecanismos son tracesToLogsV2 y tracesToMetrics en el datasource Tempo, configurados para navegar desde un span en Tempo a los logs correspondientes en Loki y a las métricas correspondientes en Prometheus respectivamente. Adicionalmente, Prometheus debe estar arrancado con el flag --enable-feature=exemplar-storage para soportar el almacenamiento de exemplars.

**9.2.6.2 Plan de Pruebas para el Caso 6.** A diferencia de los Casos anteriores, el Caso 6 no se valida mediante scripts ejecutables sino mediante verificación visual sobre el dashboard Smart Campus — Tempo y los datasources de Grafana. La razón es que el cableado cross-signal es por naturaleza una característica de configuración, no de comportamiento dinámico: si los datasources están correctamente configurados, las navegaciones entre señales funcionan; si no lo están, no funcionan. La validación consiste en ejecutar una sucesión de navegaciones que demuestran las cuatro relaciones esperadas.

El procedimiento de validación parte de la generación previa de tráfico hacia los microservicios mediante el script test-data-trace.sh, lo cual produce trazas, métricas y logs correlacionados. Posteriormente se realizan cuatro verificaciones secuenciales en Grafana:

- Verificación 1 (exemplar → traza): se abre un panel de métrica HTTP en el dashboard de microservicios, se hace clic sobre un punto exemplar (representado como un punto de color sobre la curva de la métrica), y se confirma que aparece el popup con el botón Ver traza en Tempo, el cual al pulsarse abre la traza correspondiente en el panel de Tempo.
- Verificación 2 (derivedFields → traza): se abre un panel de logs en el dashboard de overview, se expande una línea de log que contenga un trace_id, y se confirma que aparece el botón Tempo asociado al campo trace_id extraído por el derivedField, el cual al pulsarse abre la traza.
- Verificación 3 (tracesToLogsV2 → logs): se abre una traza en Tempo, se hace clic en un span, y se confirma que en el panel lateral aparece la opción Logs for this span, la cual al pulsarse ejecuta una consulta LogQL sobre Loki filtrada por el trace_id de la traza.
- Verificación 4 (tracesToMetrics → métricas): mismo procedimiento de la verificación anterior pero seleccionando la opción Related metrics, la cual ejecuta consultas sobre Prometheus filtradas por el span.

### *9.2.7 Caso 7: Sistema de alertas (Capa 4)*

**9.2.7.1 Infraestructura para el Caso 7.** El Caso 7 requiere los mismos componentes que el Caso 6, ya que el sistema de alertas opera sobre las métricas generadas por los microservicios instrumentados. Adicionalmente requiere que Prometheus tenga cargadas y activas las cinco reglas de alerta definidas en el archivo values.yaml del chart, lo cual se verifica con la API /api/v1/rules de

Prometheus. Las cinco reglas son ServicioCaido, MemoriaHeapCritica, CPUAltaSostenida, ErroresHTTP y LatenciaP95Alta, cada una con su expresión PromQL, su periodo de espera (for) y su severidad asociada.

Una particularidad de este Caso, documentada explícitamente como decisión metodológica, es que tres de las cinco reglas (MemoriaHeapCritica, CPUAltaSostenida y LatenciaP95Alta) operan con umbrales deliberadamente bajos para la validación experimental, dado que el sistema en operación normal mantiene valores muy por debajo de los umbrales productivos estándar (heap entre 0.7 y 1.5 %, CPU del proceso Java inferior al 5 %, percentil 95 de latencia entre 3 y 5 ms). Los umbrales productivos quedan documentados en la descripción de cada regla para su restauración en un despliegue real. Las dos reglas restantes (ServicioCaido y ErroresHTTP) se validan con umbrales productivos al provocar las degradaciones reales correspondientes.

**9.2.7.2 Plan de Pruebas para el Caso 7.** El plan de pruebas del Caso 7 se compone de cinco scripts, uno por regla. Cada script sigue una estructura común de cinco a siete fases: la primera fase captura el estado inicial de la alerta y de la métrica subyacente, la segunda fase provoca la condición de disparo durante un tiempo suficiente para superar el periodo for de la regla, las fases intermedias monitorean el estado de la alerta a intervalos regulares, una fase confirma el firing y registra el RESULTADO de la ejecución, una fase de captura visual permite documentar el estado en Grafana, y, cuando aplica, una fase final restaura las condiciones del sistema y verifica la transición firing → inactive.

Las cinco reglas se validan mediante las siguientes estrategias de provocación específicas:

- ErroresHTTP: bombardeo sostenido de tres minutos al endpoint /data/influx/measurement/temperature/last cada 0.5 segundos, aprovechando un bug real de tipo

IndexOutOfBoundsException presente en el controlador. Esto genera una tasa sostenida de errores 500 que supera el umbral productivo de 0.1 req/s.

- LatenciaP95Alta: treinta solicitudes en paralelo durante seis minutos contra el endpoint /data/camera/list, suficientes para sostener la P95 por encima del umbral de validación de 2 ms.
- MemoriaHeapCritica: cien solicitudes en paralelo durante ocho minutos contra endpoints que generan presión de memoria (deserialización JSON y consultas Mongo). La condición se cumple naturalmente con la carga base del sistema dado el umbral de validación de 0.5 %.
- CPUAltaSostenida: diez solicitudes en paralelo durante siete minutos, suficientes para mantener el process_cpu_usage por encima del umbral de validación de 2 %.
- ServicioCaido: escalado del deployment admin a cero réplicas durante 90 segundos, lo cual provoca que Prometheus pierda el target y se cumpla la condición up == 0. Tras el periodo for de un minuto, la alerta dispara con severidad critical. Posteriormente se restaura admin a una réplica y se verifica la transición firing → inactive.

El proceso de validación de cada regla consta de cuatro pasos. En el primer paso se ejecuta el script y se verifica en su salida el ciclo inactive → pending → firing → inactive. En el segundo paso se accede al dashboard Smart Campus — Alertas y rendimiento durante la fase de firing y se captura una imagen del panel Alertas activas, donde la alerta debe figurar con su severidad (badge rojo para critical, badge naranja para warning), su tiempo en estado y la totalidad de sus labels (app, uri, status según aplique). En el tercer paso se captura el panel Histórico de alertas (timeline) que registra la transición naranja → roja correspondiente al ciclo pending → firing. En el cuarto paso se confirma en la API de Prometheus que la alerta efectivamente cambió de estado a firing y posteriormente a inactive.

# 10. Análisis de resultados

Esta sección presenta los resultados obtenidos al ejecutar el plan de pruebas definido en la sección 9, organizados por caso y consolidados en tres análisis transversales: cobertura de observabilidad, capacidad de diagnóstico y cumplimiento de requerimientos.

## 10.1 Resultados del Caso 1 — Comparativa de configuraciones de despliegue

Las tres configuraciones se desplegaron correctamente. El tiempo de arranque se midió tres veces por configuración sobre un entorno limpio y se reporta el promedio. La configuración Docker Compose con build local resultó la más lenta (7:36 min) por la compilación de imágenes en cada ejecución. La configuración con imágenes de DockerHub fue la más rápida (2:27 min) al eliminar el paso de build. K3s con Helm registró un tiempo intermedio (5:22 min), atribuible a la descarga de imágenes, la ejecución de los initContainers de cada pod y la carga del orquestador. Los tiempos de arranque de las tres configuraciones se resumen en la tabla 42.

**Tabla 42**

*Comparativa primer despliegue*

| Aspecto | Docker Compose (build local) | Docker Compose (DockerHub) | K3s + Helm |
|---|---|---|---|
| **Comando** | docker compose up --build -d | docker compose up -d | helm install smart-campus ... -f values-custom.yaml |
| **Requiere código fuente** | Si | No | No |
| **Requiere herramientas de build** | Si (Docker BuildKit) | No | No |
| **Orquestación y Rolling Update** | No | No | Si |
| **Monitoreo y observabilidad** | No | No | Si |
| **Tiempo promedio de arranque medido** | 7:36 minutos | 2:27 minutos | 5:22 minutos |

## 10.2 Resultados del Caso 2 — Rolling Update y Rollback

El mecanismo de Rolling Update se validó reiniciando el deployment del microservicio de datos con una imagen deliberadamente errónea, lo que forzó un estado CrashLoopBackOff en el pod entrante. Como se aprecia en la Figura 15, durante la transición coexisten dos instancias del pod: la nueva en estado de fallo y la anterior aún en servicio, evidenciando que Kubernetes no elimina el pod anterior hasta confirmar que el nuevo está listo. Ante el fallo, se ejecutó un rollback mediante kubectl rollout undo, restaurando la versión anterior del pod. La Figura 16 muestra el pod recuperado, cuyo identificador coincide con el de la versión previa al upgrade, confirmando la reversión exitosa.

**Figura 15**

*Rolling update, microservicio de datos*

```
NAME                        READY   STATUS             RESTARTS      AGE
admin-585f86dd99-ckw78      1/1     Running            0             146m
data-577797ffbb-4268w       1/1     Running            0             69m
data-658886f476-dtxxs       0/1     CrashLoopBackOff   2 (19s ago)   58s
```

**Figura 16**

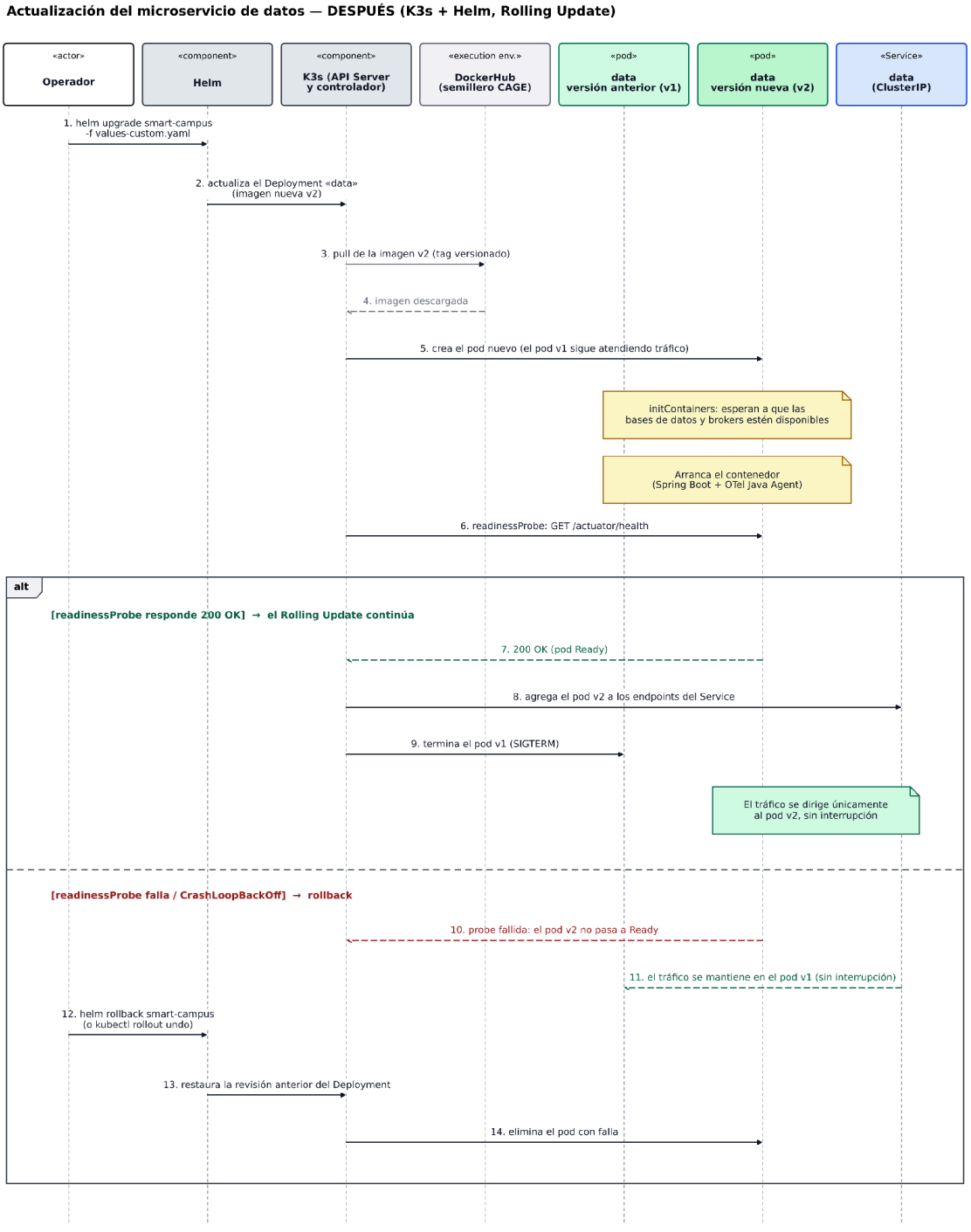
*Rollback, microservicio de datos*

```
root@Cristian:/home/cristian/proyectov3/smart-campus-charts/stack-completo/smart-campus# kubectl rollout undo deployment/data -n smart-cam
pus
deployment.apps/data rolled back
root@Cristian:/home/cristian/proyectov3/smart-campus-charts/stack-completo/smart-campus# nano values.yaml kubectl get pods -n smart-campus
ampus -w
NAME                        READY   STATUS    RESTARTS   AGE
admin-585f86dd99-ckw78      1/1     Running   0          148m
data-577797ffbb-4268w       1/1     Running   0          71m
emqx-0                      1/1     Running   0          148m
influxdb-0                  1/1     Running   0          148m
minio-0                     1/1     Running   0          148m
mongodb-0                   1/1     Running   0          148m
postgres-0                  1/1     Running   0          148m
rabbitmq-0                  1/1     Running   0          148m
```

La Figura 17 sintetiza el comportamiento validado en este caso como diagrama de secuencia, para contrastarlo con el proceso de actualización inicial de la Figura 4 en la sección 5.2.1. Con K3s y Helm, la imagen versionada se descarga desde DockerHub, los initContainers esperan a que las dependencias estén disponibles y la readinessProbe determina si el pod nuevo recibe tráfico. Mientras el pod nuevo no pase a estado Ready, el pod anterior continúa atendiendo las solicitudes, y la reversión se realiza con helm rollback sin reconstruir imágenes, eliminando la ventana de inactividad observada en el estado inicial.

**Figura 17**

*Actualización del microservicio de datos con K3s y Helm (Rolling Update y rollback)*



### 10.3 Resultados del Caso 3 — Despliegue del stack de monitoreo y observabilidad

El stack de monitoreo y observabilidad se desplegó correctamente mediante una única operación de Helm. La Figura 18 muestra el resultado de kubectl get pods -n monitoring, donde todos los pods del namespace se encuentran en estado Running con la totalidad de sus contenedores listos.

**Figura 18**

*Estado de monitoreo y observabilidad*

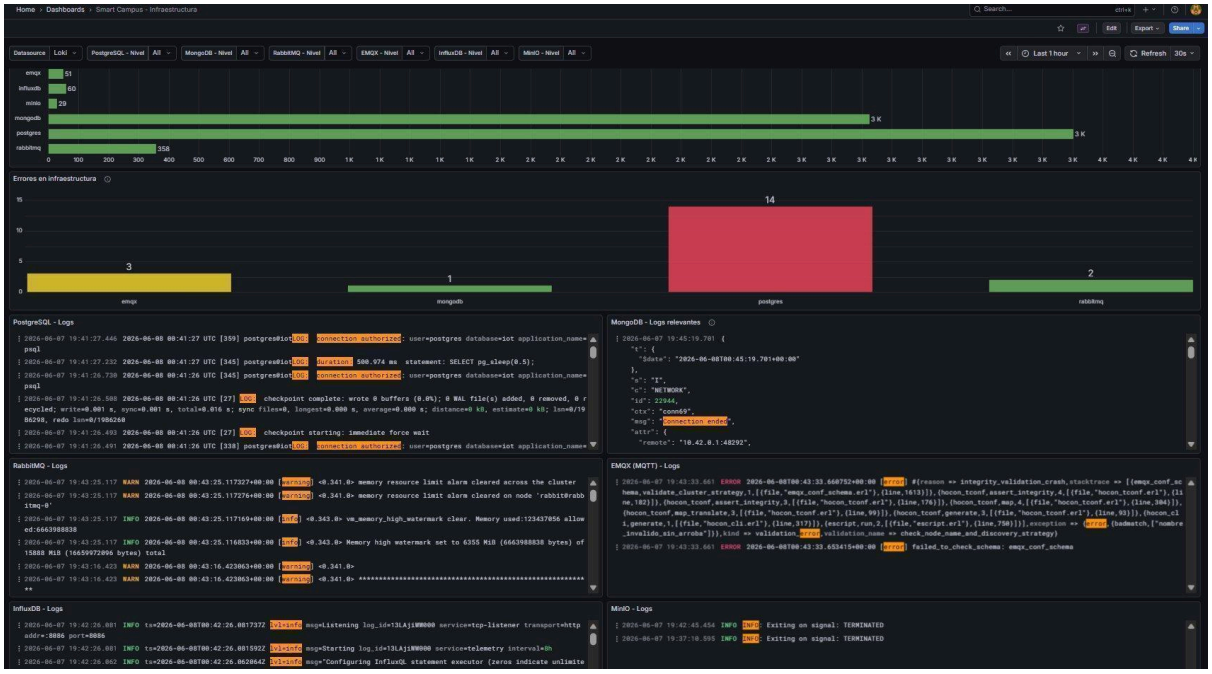
```
NAME                                          READY   STATUS    RESTARTS   AGE
minio-monitoring-0                            1/1     Running   0          3h9m
monitoring-alertmanager-0                     1/1     Running   0          3h9m
monitoring-alloy-qbzfp                        2/2     Running   0          3h9m
monitoring-grafana-6cd49ddccd-5f7gn           2/2     Running   0          3h9m
monitoring-loki-0                             2/2     Running   0          3h9m
monitoring-prometheus-server-684fd8c6fb-5tf6l   2/2     Running   0          3h9m
monitoring-tempo-0                            1/1     Running   0          3h9m
```

### 10.4 Resultados del Caso 4 — Logs de infraestructura

El dashboard de infraestructura consolidó la actividad de logs de los brokers y bases de datos del sistema, mostrando tanto el volumen total de eventos por componente como los errores detectados en cada uno. La Figura 19 muestra el estado del dashboard tras la ejecución de los scripts de prueba, donde todos los componentes registraron actividad coherente con las cargas sintéticas inyectadas. Cabe señalar que el vocabulario de severidad no es homogéneo entre componentes, por lo que la ausencia de errores en alguno de ellos no necesariamente indica un comportamiento anómalo del dashboard sino una convención de clasificación propia del componente.

**Figura 19**

*Dashboard de infraestructura*

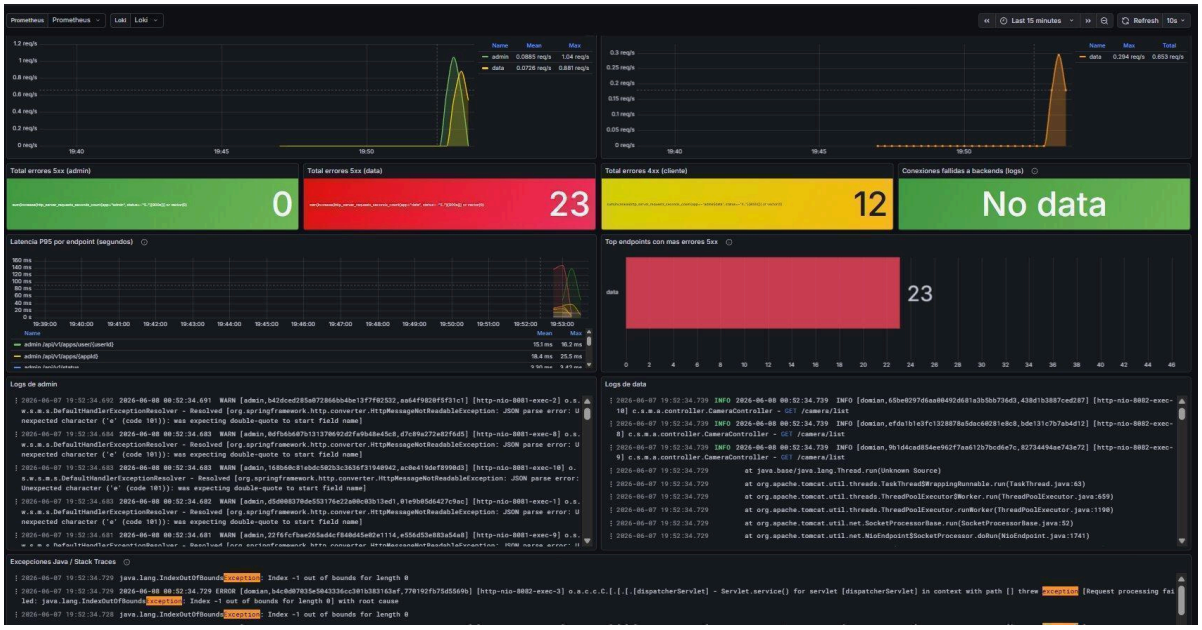



## 10.5 Resultados del Caso 5 — Instrumentación de microservicios

El dashboard de microservicios expuso las métricas RED de los servicios de administración y datos durante la ejecución de la carga sintética. La Figura 20 muestra la tasa de peticiones, la distribución de errores 4xx y 5xx, y la latencia P95 (que permite identificar el comportamiento en el peor caso sin distorsiones por valores extremos), junto con los logs de ambos servicios. El conjunto confirma que la instrumentación con OpenTelemetry capturó correctamente la actividad generada en cada componente.

**Figura 20**

*Dashboard de microservicios*

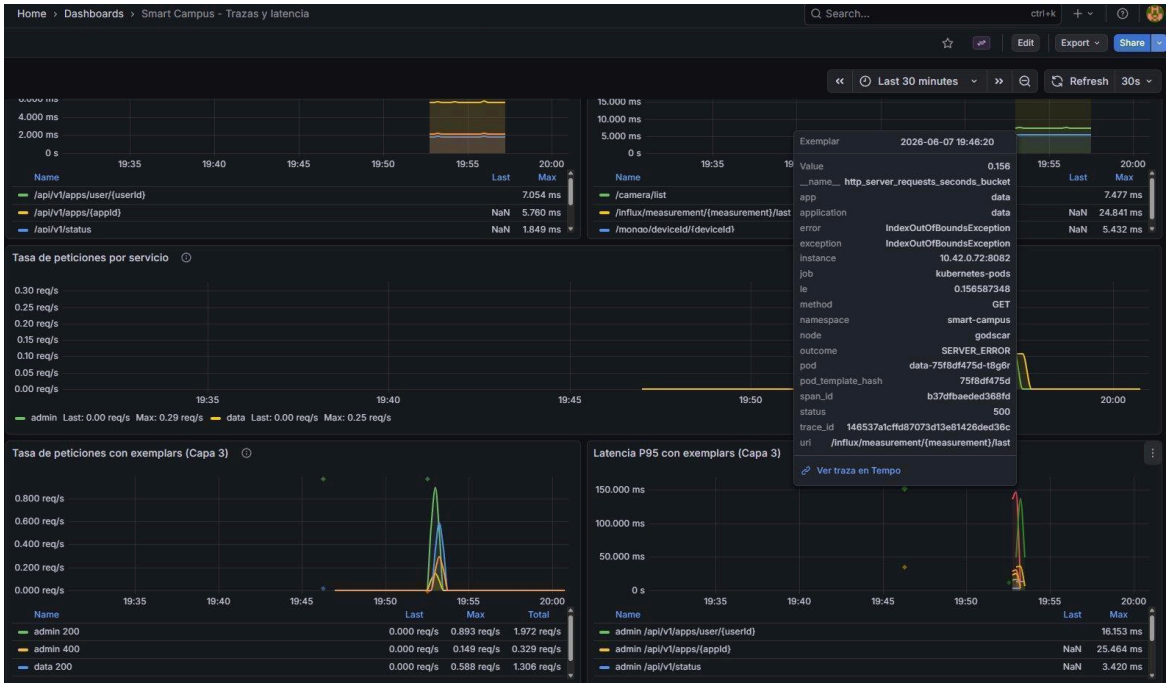



## 10.6 Resultados del Caso 6 — Correlación cross-signal

La Figura 21 muestra el panel de métricas con exemplars activos en Prometheus, donde cada punto del gráfico de latencia permite navegar directamente hacia la traza que lo originó.

**Figura 21**

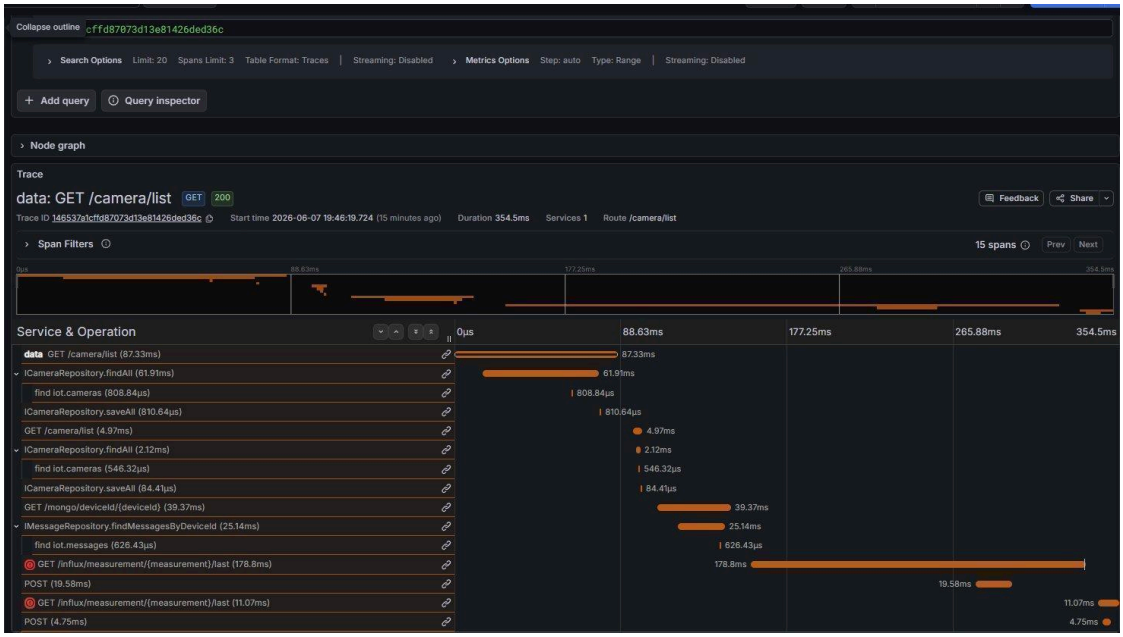
*Exemplars en Prometheus*




La Figura 22 presenta la traza completa en Tempo, descompuesta en spans que permiten seguir el recorrido de una solicitud a través de los distintos componentes del sistema e identificar en cuál de ellos se produjo el fallo.

**Figura 22**

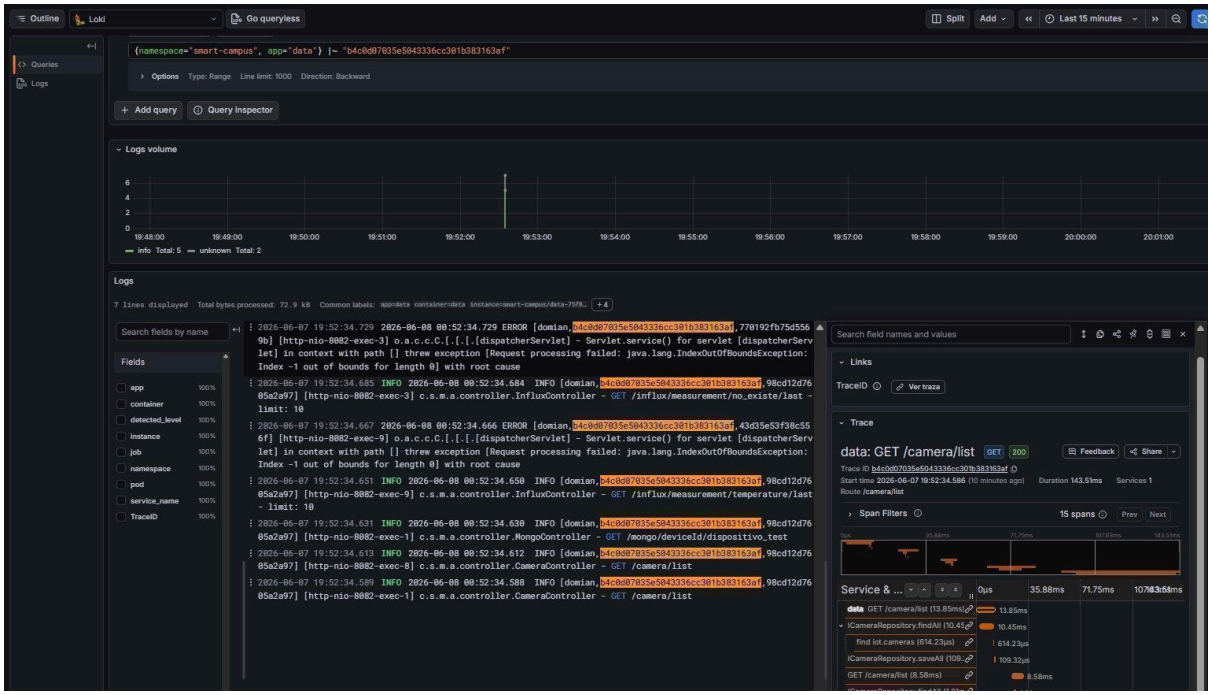
*Traza completa en Tempo*




La Figura 23 muestra la vista de Loki donde, al seleccionar un log que contiene un identificador de traza, Grafana reconoce ese identificador y abre automáticamente la traza correspondiente en Tempo en panel dividido, sin necesidad de copiar ni buscar manualmente.

**Figura 23**

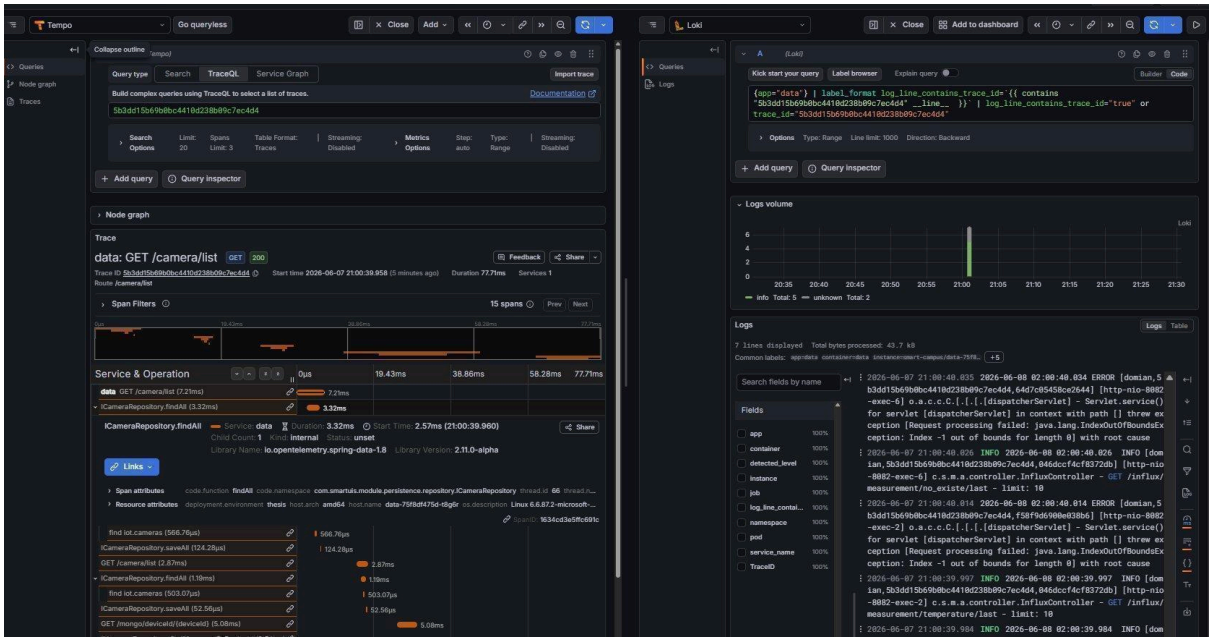
*Navegación de log a traza*




La Figura 24 presenta el recorrido inverso: desde un span seleccionado en Tempo, Loki genera automáticamente la consulta filtrada por ese identificador y muestra los logs asociados a esa solicitud específica.

**Figura 24**

*Navegación de traza a logs*




## 10.7 Resultados del Caso 7 — Sistema de alertas

La Figura 25 muestra la alerta de errores HTTP en estado firing, sostenida durante varios minutos con severidad crítica, evidenciando que el sistema detectó y mantuvo la condición de error durante el período de carga.

**Figura 25**

*Alerta de errores HTTP*




La Figura 26 presenta la alerta de latencia P95 activa con severidad de advertencia, disparada tras superar el umbral configurado durante el tiempo de evaluación establecido en la regla.

**Figura 26**

*Alerta de latencia P95*




La Figura 27 muestra la alerta de memoria heap crítica activa en ambos microservicios de forma simultánea, confirmando que la regla evalúa cada instancia de forma independiente.

**Figura 27**

*Alerta de memoria heap crítica*




La Figura 28 presenta la alerta de servicio caído, donde el panel de estado refleja visualmente la diferencia entre el servicio afectado y el que continuó operando con normalidad.

**Figura 28**

*Alerta de servicio caído*




La Figura 29 muestra la alerta de CPU alta sostenida con severidad de advertencia, disparada tras superar el umbral definido durante el tiempo mínimo requerido por la regla.

**Figura 29**

*Alerta de CPU alta sostenida*




La Figura 30 presenta el timeline acumulado con las cinco alertas disparadas de forma simultánea, confirmando que el motor de alertas procesó la carga concurrente sin omitir ninguna regla.

**Figura 30**

*Timeline de alertas simultáneas*



## 10.8 Cobertura de observabilidad

Los microservicios de datos y administración fueron instrumentados en las tres señales (métricas, logs y trazas), mientras que los servicios de infraestructura exponen únicamente logs de contenedor recolectados por Alloy. Esto representa una cobertura total de logs del 100% (9/9 componentes), y una cobertura de métricas y trazas del 22% (2/9), concentrada en los componentes de mayor relevancia operacional. La tabla 43 presenta el detalle de la cobertura de observabilidad.

**Tabla 43**

*Cobertura de observabilidad*

| Servicio | Métricas | Logs estructurados | Trazas |
|---|---|---|---|
| **Microservicio de datos (data)** | Si | Si | Si |
| **Microservicio de administración (admin)** | Si | Si | Si |
| **PostgreSQL** | No | Si (logs de contenedor) | No |
| **MongoDB** | No | Si (logs de contenedor) | No |
| **InfluxDB** | No | Si (logs de contenedor) | No |
| **MinIO** | No | Si (logs de contenedor) | No |
| **EMQX** | No | Si (logs de contenedor) | No |
| **RabbitMQ** | No | Si (logs de contenedor) | No |
| **Frontend** | No | Si (logs de contenedor) | No |

| Servicio | Métricas | Logs estructurados | Trazas |
|---|---|---|---|
| **Cobertura** | **2/9 (22%)** | **9/9 (100%)** | **2/9 (22%)** |

## 10.9 Capacidad de diagnóstico

El sistema respondió satisfactoriamente las 8 consultas de diagnóstico evaluadas (100%), cubriendo desde la identificación del servicio afectado hasta la reconstrucción del flujo completo de una solicitud fallida. La tabla 44 muestra la capacidad de respuesta del sistema a cada una de las consultas.

**Tabla 44**

*Capacidad de respuestas a consultas de diagnóstico*

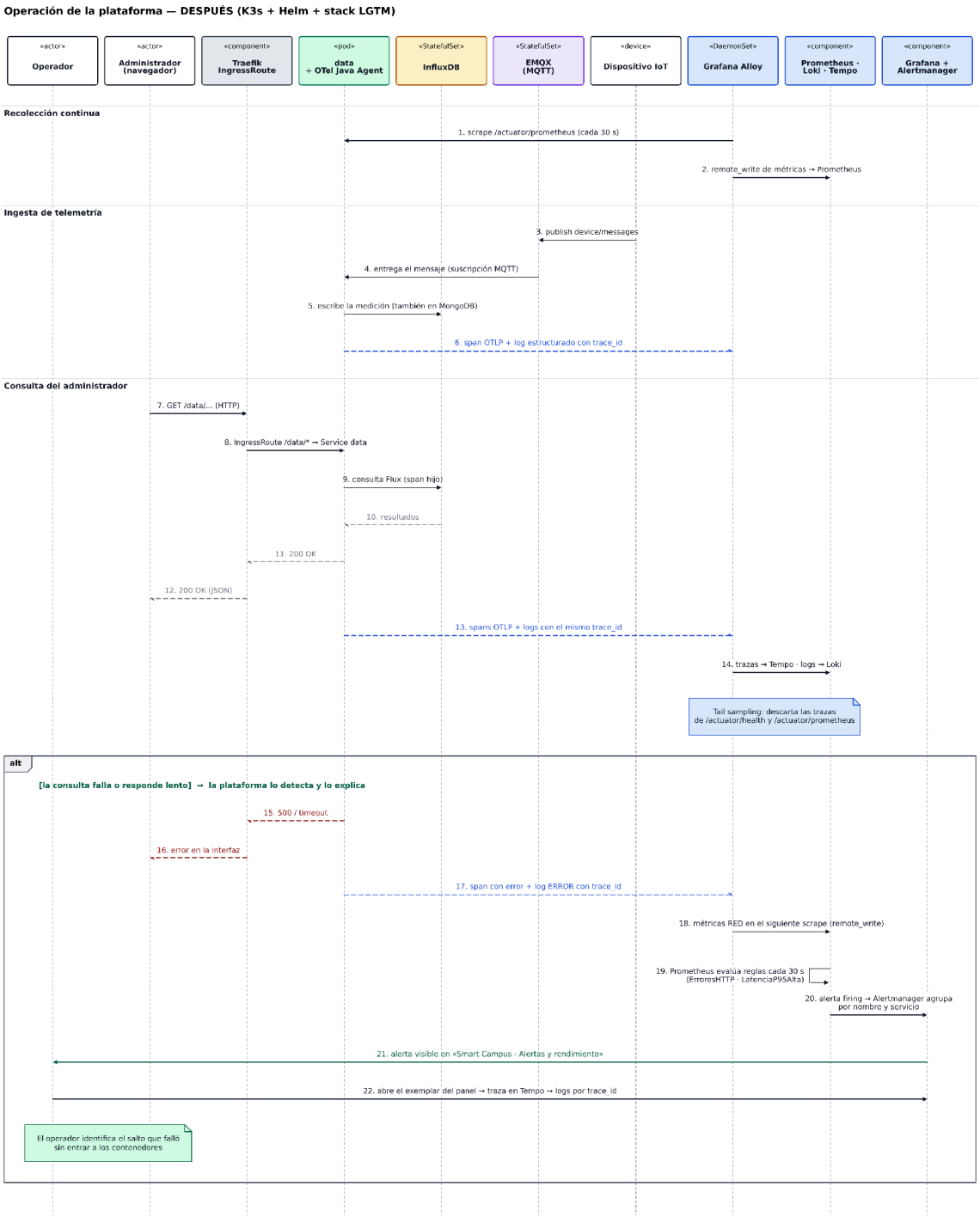
| Consulta de diagnóstico | Respondida (1/0) | Fuente |
|---|---|---|
| ¿Qué servicio está afectado? | 1 | Dashboard Smart Campus - Alertas y rendimiento |
| ¿Está el servicio caído o solo degradado? | 1 | Panel Estado de servicios + alertas activas |
| ¿Desde cuándo ocurre el problema? | 1 | Serie temporal en los paneles de métricas |
| ¿Qué endpoint específico está fallando? | 1 | Panel Errores HTTP por endpoint |

| Consulta de diagnóstico | Respondida (1/0) | Fuente |
|---|---|---|
| ¿Qué operación interna falló dentro del microservicio? | 1 | Traza distribuida en Tempo |
| ¿Qué logs generó el servicio en el momento del fallo? | 1 | Loki filtrado por trace_id desde la traza |
| ¿Qué dependencia externa involucró la operación fallida? | 1 | Spans hijos en la traza |
| ¿El problema es de memoria o de CPU? | 1 | Paneles JVM heap y CPU del proceso Java |
| **Capacidad de respuesta** | **8/8 (100%)** | |

La Figura 31 contrasta el flujo de operación resultante con el estado inicial presentado en la Figura 5. Ante el mismo escenario de una consulta que falla o responde con lentitud, el microservicio emite el span con error y el log con su trace_id, Prometheus evalúa las reglas de alerta y la alerta se hace visible en el dashboard Smart Campus - Alertas y rendimiento. Desde el exemplar del panel, el operador navega hasta la traza en Tempo y los logs asociados en Loki, e identifica el componente que falló sin acceder a los contenedores.

**Figura 31**

*Operación de la plataforma con K3s, Helm y stack LGTM*



## 10.10 Cumplimiento de requerimientos

La implementación cubre la totalidad de los requerimientos funcionales y no funcionales definidos en la sección 5, distribuidos entre las capacidades de despliegue, monitoreo y observabilidad. Las tablas 45 y 46 presentan el análisis para las configuraciones Docker Compose y K3s, respectivamente.

**Tabla 45**

*Análisis configuración docker compose*

| Capacidad | RF que cumple | RNF que cumple |
|---|---|---|
| **Despliegue de la plataforma completa** | RF-DEPL-02, RF-DEPL-03 | RNF-DEPL-01, RNF-DEPL-03, RNF-DEPL-04 |
| **Acceso al frontend y gateway** | — | — |
| **Persistencia de datos** | — | — |
| **Configuración de credenciales** | RF-DEPL-03 | — |

**Tabla 46**

*Análisis configuración k3s*

| Capacidad | RF que cumple | RNF que cumple |
|---|---|---|
| **Despliegue orquestado** | RF-DEPL-02, RF-DEPL-03 | RNF-DEPL-01, RNF-DEPL-04 |
| **Enrutamiento vía Traefik** | RF-DEPL-03 | — |
| **Persistencia con PersistentVolumes** | RNF-DEPL-01 | — |

| Capacidad | RF que cumple | RNF que cumple |
|---|---|---|
| **Gestión de credenciales con Secrets** | RF-DEPL-03 | RNF-DEPL-01, RNF-DEPL-02 |
| **Actualizaciones controladas** | — | RNF-DEPL-02, RNF-DEPL-03 |
| **Health checks en microservicios** | RF-DEPL-01, RF-DEPL-04 | — |
| **Exposición de métricas JVM y HTTP** | RF-MON-01, RF-MON-02, RF-MON-03 | RNF-MON-01 |
| **Recolección de métricas con Prometheus** | — | RNF-MON-02 |
| **Alertas y dashboards de monitoreo** | RF-MON-01, RF-MON-02, RF-MON-03, RF-MON-04, RF-MON-05 | RNF-MON-03 |
| **Logs estructurados con trace_id y span_id** | RF-OBS-01, RF-OBS-02, RF-OBS-03, RF-OBS-04 | RNF-OBS-01, RNF-OBS-02 |
| **Trazas distribuidas con OpenTelemetry** | RF-OBS-02, RF-OBS-03 | RNF-OBS-01 |
| **Agregación centralizada de logs con Loki** | RF-OBS-05 | RNF-OBS-03 |

## 11. Conclusiones

El presente trabajo logró diseñar e implementar una estrategia integrada de despliegue, monitoreo y observabilidad para la plataforma Smart Campus UIS, cerrando la brecha operacional

identificada en la caracterización inicial entre la capacidad funcional de la plataforma y su capacidad de ser operada, actualizada y diagnosticada de forma confiable.

La caracterización del estado inicial reveló que Smart Campus UIS operaba con un proceso de despliegue completamente manual, sin estrategias que evitaran la interrupción del servicio durante las actualizaciones, sin métricas recolectadas, sin logs estructurados y sin ningún mecanismo de trazabilidad distribuida. Esta condición dificultaba significativamente el diagnóstico de fallos, ya que cualquier problema requería intervención manual revisando la salida de cada servicio por separado, sin herramientas que correlacionaran la información entre componentes. La identificación de estas limitaciones, junto con el mapeo de dependencias y la clasificación de componentes críticos, permitió establecer un conjunto de requerimientos funcionales y no funcionales que sirvieron de criterio de evaluación a lo largo de todo el trabajo.

La revisión del estado del arte y la evaluación comparativa de alternativas condujeron a la selección de un stack cohesivo y open source: K3s como orquestador ligero, Rolling Update como patrón de despliegue, y el conjunto Prometheus, Loki, Tempo y Grafana con Grafana Alloy como colector unificado de telemetría. Esta selección estuvo guiada por criterios verificables anclados al contexto del proyecto: infraestructura universitaria con recursos moderados, un equipo de estudiantes sin formación especializada en operaciones, y un horizonte de operación extendido que exige sostenibilidad tecnológica. La coherencia ecosistémica del stack seleccionado, donde Loki, Tempo, Grafana y Alloy pertenecen al mismo ecosistema de Grafana Labs, redujo la fricción de integración y garantizó compatibilidad entre componentes sin necesidad de adaptadores adicionales.

La adaptación de los servicios demostró que es posible instrumentar aplicaciones Spring Boot existentes para los tres pilares de observabilidad sin modificaciones estructurales al código de negocio.

Las dependencias y configuraciones necesarias se incorporaron como capas sobre la aplicación existente. De forma complementaria, se optimizó la construcción y el tamaño de la imagen del microservicio de datos, reduciendo el impacto que esta tenía sobre los tiempos de despliegue y el almacenamiento disponible.

La implementación de la estrategia produjo dos configuraciones de despliegue complementarias, diseñadas para adaptarse a diferentes contextos y niveles de recursos. La configuración Docker Compose permite levantar la plataforma completa con un único comando, sin requerir conocimientos de orquestación ni infraestructura adicional, siendo la opción natural para desarrollo y pruebas. La configuración K3s con Helm incorpora orquestación, Rolling Update, monitoreo y observabilidad, y está diseñada para entornos donde se requiere visibilidad operacional y actualizaciones controladas.

Ambas configuraciones son portables y están completamente definidas como código, por lo que pueden reproducirse en cualquier entorno compatible ajustando únicamente los valores de configuración. El uso del mecanismo de superposición de archivos de Helm permite además habilitar o deshabilitar componentes de infraestructura de forma selectiva, adaptando el despliegue a las necesidades del equipo en cada momento.

La validación experimental evaluó la estrategia en sus tres dimensiones. La naturaleza declarativa del Rolling Update en K3s garantiza por diseño que las actualizaciones reemplacen instancias de forma gradual sin interrumpir el servicio, y el mecanismo de revisiones de Helm hace posible revertir cualquier despliegue a un estado anterior con un único comando. La cobertura de instrumentación alcanzada —los tres pilares en los microservicios de negocio y logs de contenedor en todos los servicios de infraestructura— asegura que los componentes con mayor relevancia operacional

emiten señales suficientes para detectar y diagnosticar fallos. Los casos de fallo simulados permitieron verificar que las alertas configuradas responden a las condiciones operacionales definidas, y que el sistema de observabilidad es capaz de responder las consultas de diagnóstico que un integrante del equipo enfrentaría ante un incidente real, reduciendo significativamente las interacciones manuales necesarias respecto al estado anterior de la plataforma.

La estrategia implementada dota a Smart Campus UIS de una base operacional sobre la que la plataforma puede seguir evolucionando con mayor confianza y visibilidad sobre su comportamiento.

## 12. Trabajo futuro

A partir de los resultados obtenidos y las decisiones de alcance tomadas durante el desarrollo del trabajo, se proponen las siguientes líneas de trabajo futuro para continuar con la evolución de la plataforma Smart Campus UIS:

Seguridad e HTTPS: Smart Campus UIS no cuenta actualmente con mecanismos de seguridad implementados. Una línea de trabajo natural es incorporar autenticación y autorización sobre los endpoints de la plataforma, reflejando los estados de acceso denegado en los dashboards de observabilidad para tener visibilidad sobre intentos de acceso no autorizados. De forma complementaria, se puede habilitar HTTPS en el Ingress de Traefik para cifrar el tráfico entre clientes y la plataforma.

Gestión de secretos: En la implementación actual las credenciales se centralizan en Kubernetes Secrets, que representa una mejora respecto al estado anterior pero no constituye una solución de

gestión de secretos completa. A futuro se puede integrar una herramienta dedicada de gestión de secretos que habilite rotación automática de credenciales y auditoría de accesos sobre los componentes de la plataforma.

Escalabilidad de StatefulSets: La arquitectura actual despliega una réplica de cada base de datos y broker, lo cual es adecuado para el contexto presente de la plataforma. Escalar horizontalmente los microservicios en K3s es directo, pero escalar los StatefulSets de bases de datos y brokers implica manejar concurrencia y sincronización entre réplicas, lo que requiere configuraciones específicas por tecnología que quedaron fuera del alcance de este trabajo. A futuro se puede explorar la configuración de múltiples réplicas para estos componentes, aprovechando las capacidades de identidad estable y almacenamiento persistente individual que los StatefulSets proveen.

Retención de métricas a largo plazo: Prometheus en su configuración actual es suficiente para el tamaño presente de la plataforma. Si el proyecto crece en número de servicios o dispositivos conectados, se puede incorporar Grafana Mimir como capa de almacenamiento distribuido de métricas a largo plazo, manteniendo compatibilidad total con el stack existente.

Registro de imágenes independiente: DockerHub es adecuado para el contexto actual de Smart Campus UIS, pero introduce una dependencia externa sobre la que el proyecto no tiene control. Como trabajo futuro se recomienda evaluar alternativas de registro privado que eliminen esa dependencia y ofrezcan mayor autonomía sobre el ciclo de vida de las imágenes.

Observabilidad completa de infraestructura: La estrategia implementada permite inferir el comportamiento de brokers y bases de datos a través de las trazas de los microservicios, incluyendo los tiempos de respuesta de cada dependencia. Sin embargo, una instrumentación directa sobre PostgreSQL, MongoDB, InfluxDB, MinIO, EMQX y RabbitMQ quedó fuera del alcance del trabajo

por el costo computacional adicional que implica y por requerir configuraciones específicas por tecnología. Profundizar en esta línea permitiría completar la cobertura de observabilidad de la plataforma.

Automatización CI/CD: El proceso de construcción y publicación de imágenes es actualmente manual. Incorporar un pipeline de integración y despliegue continuo automatizaría este proceso, reduciendo el riesgo de errores manuales y acortando el ciclo entre un cambio en el código y su despliegue en el clúster.

## Referencias Bibliográficas

Amazon Web Services. (2024a). *Amazon Elastic Container Service documentation* [Documentación de Amazon Elastic Container Service]. Amazon Web Services Documentation. https://docs.aws.amazon.com/ecs/

Amazon Web Services. (2024b). *AWS X-Ray documentation* [Documentación de AWS X-Ray]. Amazon Web Services Documentation. https://docs.aws.amazon.com/xray/

Araujo, A., Costa, B., Bachiega, J. J., Carvalho, L. R. y Buyya, R. (2024). *Observability in fog computing* [Observabilidad en la computación en la niebla]. arXiv. https://arxiv.org/abs/2411.17753

Atzori, L., Iera, A. y Morabito, G. (2010). The Internet of Things: A survey [El Internet de las cosas: una revisión]. *Computer Networks*, *54*(15), 2787-2805. https://doi.org/10.1016/j.comnet.2010.05.010

Ayogun. (2024). *Comparison of ELK vs Jaeger for distributed systems* [Comparación de ELK frente a Jaeger para sistemas distribuidos]. Medium. https://medium.com/@ayogun/comparison-of-elk-vs-jaeger-b920c2a65327

Banks, A., Briggs, E., Borgendale, K. y Gupta, R. (Eds.). (2019). *MQTT version 5.0* [MQTT versión 5.0]. OASIS Standard. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html

Bector, H. (2025). *Distributed tracing with Micrometer Tracing and OpenTelemetry* [Trazabilidad distribuida con Micrometer Tracing y OpenTelemetry]. Medium. https://medium.com/@bectorhimanshu/distributed-tracing-with-micrometer-tracing-and-opentel

emetry-exporting-to-otel-collector-and-7059a3c1834a

BigPanda. (2025). *Monitoring and observability tool effectiveness for IT event management* [Efectividad de las herramientas de monitoreo y observabilidad para la gestión de eventos de TI]. BigPanda. https://www.bigpanda.io/resource/report/tool-effectiveness-for-it-event-management

Borgp, S. I. (2022). *Exploring the applications of IoT with Kubernetes* [Exploración de las aplicaciones de IoT con Kubernetes]. Medium. https://medium.com/@borgpsi/exploring-the-applications-of-iot-with-kubernetes-f244a0b4ee99

Cerbos. (2024). *The value of monitoring and observability in microservices, and associated challenges* [El valor del monitoreo y la observabilidad en microservicios, y los desafíos asociados]. Cerbos. https://www.cerbos.dev/blog/monitoring-and-observability-microservices

China, C. (2025). *Three pillars of observability: Logs, metrics and traces* [Tres pilares de la observabilidad: registros, métricas y trazas]. IBM. https://www.ibm.com/think/insights/observability-pillars

Cloud Native Daily. (2023). *7 best tracing tools for microservices* [Las 7 mejores herramientas de trazabilidad para microservicios]. Medium. https://medium.com/cloud-native-daily/7-best-tracing-tools-for-microservices-27a5e3bc4b9c

Costa, B., Banerjee, A., Jayaraman, P. P., Carvalho, L. R., Bachiega, J. J. y Araujo, A. (2024). Achieving observability on fog computing with the use of open-source tools [Logro de la observabilidad en la computación en la niebla mediante herramientas de código abierto]. En *Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2023)*. Springer. https://doi.org/10.1007/978-3-031-63992-0_21

Datadog. (2024). *Datadog documentation and pricing* [Documentación y precios de Datadog]. Datadog Documentation. https://docs.datadoghq.com/

Decimavilla-Alarcón, D. C. y Marcillo-Franco, P. F. (2025). Arquitectura de microservicios basada en contenedores para despliegue ágil de aplicaciones IoT en la nube. *Revista Episteme & Praxis*, *3*(1), 35-49.

DemandSage. (2025). *How many IoT devices are there [2025 statistics]* [¿Cuántos dispositivos IoT existen? (estadísticas de 2025)]. DemandSage. https://www.demandsage.com/number-of-iot-devices/

Docker Inc. (2024). *Docker overview* [Descripción general de Docker]. Docker Documentation. https://docs.docker.com/get-started/docker-overview/

Doubletapp. (2023). *Overview of monitoring system with Prometheus and Grafana* [Descripción general de un sistema de monitoreo con Prometheus y Grafana]. Medium. https://doubletapp.medium.com/overview-of-monitoring-system-with-prometheus-and-grafana-9ce6501eff88

Dynatrace. (2024). *Dynatrace documentation and pricing* [Documentación y precios de Dynatrace]. Dynatrace Documentation. https://www.dynatrace.com/support/doc/

El Akhdar, A., Baidada, C., Kartit, A., Hanine, M., Osorio García, C., Garcia Lara, R. y Ashraf, I. (2024). Exploring the potential of microservices in Internet of Things: A systematic review of security and prospects [Exploración del potencial de los microservicios en el Internet de las cosas: una revisión sistemática de la seguridad y las perspectivas]. *Sensors*, *24*(20), Artículo 6771. https://doi.org/10.3390/s24206771

Emon, N. K. (2025). *Production-ready observability with Prometheus, Loki & Grafana* [Observabilidad lista para producción con Prometheus, Loki y Grafana]. Medium.

https://medium.com/@neamulkabiremon/production-ready-observability-with-prometheus-loki-grafana-2ce1ba9f7423

EMQ Technologies. (2024). *EMQX documentation* [Documentación de EMQX]. EMQX Documentation. https://docs.emqx.com/en/emqx/latest/

EMQX Team. (2024). *OpenTelemetry: The basics & benefits for MQTT and IoT observability* [OpenTelemetry: fundamentos y beneficios para la observabilidad de MQTT e IoT]. EMQX Official Blog. https://www.emqx.com/en/blog/opentelemetry-mqtt-iot

Faseeha, U., Syed, H. J., Samad, F. y Zehra, S. (2025). Observability in microservices: An in-depth exploration of frameworks, challenges, and deployment paradigms [Observabilidad en microservicios: una exploración en profundidad de marcos de trabajo, desafíos y paradigmas de despliegue]. *IEEE Access*, *13*, 72011-72039. https://doi.org/10.1109/ACCESS.2025.3562125

Ferreira, I. (2023). *Microservices as a solution for IoT architecture challenges* [Los microservicios como solución a los desafíos de la arquitectura IoT]. Medium. https://isabellafer.medium.com/microservices-as-a-solution-for-iot-architecture-challenges-fb5dfa12422c

George, L. (2025). *Building a factory machine/sensor monitoring IoT pipeline* [Construcción de un pipeline IoT de monitoreo de máquinas y sensores de fábrica]. Medium. https://medium.com/@georgelza/building-a-factory-machine-sensor-monitoring-iot-pipeline-d2bd4fa0eb7d

Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S., Malavolta, I., Islam, T., Dînga, M., Koziolek, A., Singh, S., Armbruster, M., Gutierrez-Martinez, J. M., Caro-Alvaro, S., Rodriguez, D., Weber, S., Henss, J., Fernandez-Vogelin, E. y Simon-Panojo, F. (2024). Monitoring tools for

DevOps and microservices: A systematic grey literature review [Herramientas de monitoreo para DevOps y microservicios: una revisión sistemática de literatura gris]. *The Journal of Systems and Software*, *208*, Artículo 111906. https://doi.org/10.1016/j.jss.2023.111906

Glushach, R. (2023). *Building for scalability: Microservice deployment patterns* [Construir para la escalabilidad: patrones de despliegue de microservicios]. Medium. https://romanglushach.medium.com/building-for-scalability-microservice-deployment-patterns-4c9f82b04a6d

Godfrey, R., Ingham, D. y Schloming, R. (Eds.). (2012). *OASIS advanced message queuing protocol (AMQP) version 1.0* [Protocolo avanzado de colas de mensajes (AMQP) de OASIS, versión 1.0]. OASIS Standard. https://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html

Grafana Labs. (2024). *LGTM stack documentation: Loki, Grafana, Tempo, Mimir* [Documentación del stack LGTM: Loki, Grafana, Tempo, Mimir]. Grafana Documentation. https://grafana.com/docs/

Grafana Labs. (2025, 25 de marzo). *Grafana Labs unveils 2025 observability survey findings and open source updates at KubeCon Europe* [Grafana Labs presenta los resultados de su encuesta de observabilidad 2025 y novedades de código abierto en KubeCon Europa]. Grafana. https://grafana.com/press/2025/03/25/grafana-labs-unveils-2025-observability-survey-findings-and-open-source-updates-at-kubecon-europe/

Hartmann, V. (2025). *Migrating from Promtail to Grafana Alloy in Kubernetes* [Migración de Promtail a Grafana Alloy en Kubernetes]. Medium. https://medium.com/@vincenthartmann/migrating-from-promtail-to-grafana-alloy-in-kubernetes-

53ce4c5b7556

Helm Authors. (s.f.). *Helm documentation* [Documentación de Helm]. Helm Documentation. Recuperado en 2024, de https://helm.sh/docs/

Hettiarachchi, Y. (2024). *Observability — Distributed request tracing with OpenTelemetry* [Observabilidad: trazabilidad distribuida de solicitudes con OpenTelemetry]. Medium. https://medium.com/@hettiarachchi.yashodha/observability-distributed-request-tracing-with-opentelemetry-c95e83795f6e

InfluxData. (2023). *Is Flux being deprecated with InfluxDB 3.0?* [¿Se está deprecando Flux con InfluxDB 3.0?]. InfluxData Community Forums. https://community.influxdata.com/t/is-flux-being-deprecated-with-influxdb-3-0/30992

InfluxData. (2024). *InfluxDB OSS v2 documentation* [Documentación de InfluxDB OSS v2]. InfluxDB Documentation. https://docs.influxdata.com/influxdb/v2/

IoT Analytics. (2024). *State of IoT 2024: Number of connected IoT devices growing 13% to 18.8 billion globally* [Estado del IoT 2024: el número de dispositivos IoT conectados crece 13 % hasta 18.800 millones a nivel mundial]. IoT Analytics. https://iot-analytics.com/number-connected-iot-devices/

IoT For All. (2024). *Scaling IoT monitoring and observability solutions* [Escalamiento de soluciones de monitoreo y observabilidad para IoT]. IoT For All. https://www.iotforall.com/scaling-iot-monitoring-observability-solutions

Jaeger Authors. (2024). *Deployment — Jaeger documentation* [Despliegue: documentación de Jaeger]. Jaeger Documentation. https://www.jaegertracing.io/docs/1.76/deployment/

Javeri, P. (2019). *Architectural patterns for IoT — Why use microservices?* [Patrones arquitectónicos para IoT: ¿por qué usar microservicios?]. Medium.

https://medium.com/@prashunjaveri/architectural-patterns-for-iot-why-use-micro-services-3154bfbafcce

Jiménez, H., Cárcamo, E. y Pedraza, G. (2020). Plataforma software extensible para campus inteligente basada en microservicios. *RISTI - Revista Ibérica de Sistemas e Tecnologias de Informação*, (E38), 270-282. https://dialnet.unirioja.es/servlet/articulo?codigo=8576444

Kim, G., Humble, J., Debois, P. y Willis, J. (2016). *The DevOps handbook: How to create world-class agility, reliability, and security in technology organizations* [El manual de DevOps: cómo crear agilidad, confiabilidad y seguridad de clase mundial en las organizaciones tecnológicas]. IT Revolution Press. https://itrevolution.com/product/the-devops-handbook/

Kubernetes Authors. (2024a). *Pod lifecycle — Container probes* [Ciclo de vida de los Pods: sondas de contenedor]. Kubernetes Documentation. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

Kubernetes Authors. (2024b). *Workloads — Deployments, StatefulSets, and Services* [Cargas de trabajo: Deployments, StatefulSets y Services]. Kubernetes Documentation. https://kubernetes.io/docs/concepts/workloads/

Lesmana, D. (2023). *Microservices — Deployment patterns* [Microservicios: patrones de despliegue]. Medium. https://dennylesmana.medium.com/microservices-deployment-patterns-ca1343c89e13

Lozada Garavito, J. C. y Cañon Castillo, J. J. (2020). *Extensión del módulo de gestión de datos de la plataforma Smart Campus UIS para el soporte de diversos protocolos y almacenamiento de datos* (Tesis de pregrado). Universidad Industrial de Santander, Bucaramanga, Colombia. https://noesis.uis.edu.co/

Lumigo. (2022). *Microservices observability: 3 pillars and 6 patterns* [Observabilidad de

microservicios: 3 pilares y 6 patrones]. Lumigo. https://lumigo.io/microservices-monitoring/microservices-observability/

McKinsey & Company. (2024a). *IoT value set to accelerate through 2030: Where and how to capture it* [El valor del IoT se acelerará hasta 2030: dónde y cómo capturarlo]. McKinsey & Company. https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/iot-value-set-to-accelerate-through-2030-where-and-how-to-capture-it

McKinsey & Company. (2024b). *What is the Internet of Things (IoT)?* [¿Qué es el Internet de las cosas (IoT)?] McKinsey & Company. https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-the-internet-of-things

Microsoft Azure. (2024). *Azure Container Instances documentation* [Documentación de Azure Container Instances]. Microsoft Azure Documentation. https://docs.microsoft.com/azure/container-instances/

MinIO, Inc. (2024). *MinIO documentation* [Documentación de MinIO]. MinIO Documentation. https://min.io/docs/minio/linux/index.html

Mirantis. (2025a). *Mirantis extends Swarm support another five years* [Mirantis extiende el soporte de Swarm cinco años más]. Mirantis Press Center. https://www.mirantis.com/company/press-center/company-news/mirantis-extends-swarm-support-another-five-years/

Mirantis. (2025b). *What is Docker Swarm / Swarm mode orchestration?* [¿Qué es Docker Swarm y la orquestación en modo Swarm?]. Mirantis Cloud Native Concepts. https://www.mirantis.com/cloud-native-concepts/understanding-swarm/what-is-swarm-mode-orchestration/

MongoDB, Inc. (2024). *MongoDB manual* [Manual de MongoDB]. MongoDB Documentation. https://www.mongodb.com/docs/manual/

Mthembu, S. J., Ezeji, I. N. y Adigun, M. (2023). Orchestration tools for efficient deployment of IoT applications in fog computing [Herramientas de orquestación para el despliegue eficiente de aplicaciones IoT en la computación en la niebla]. En *ICARTI 2023*. https://doi.org/10.59200/ICARTI.2023.021

Mustufa, S. (2025). *Spring Boot observability stack (OpenTelemetry, Prometheus, Loki, Tempo, Grafana, Alertmanager)* [Stack de observabilidad para Spring Boot (OpenTelemetry, Prometheus, Loki, Tempo, Grafana, Alertmanager)]. Medium. https://medium.com/@mustufa589/spring-boot-observability-stack

New Relic. (2024). *New Relic documentation and pricing* [Documentación y precios de New Relic]. New Relic Documentation. https://docs.newrelic.com/

Nguyen, P. H., Ferry, N., Erdogan, G., Song, H., Lavirotte, S., Tigli, J.-Y. y Solberg, A. (2019). Advances in deployment and orchestration approaches for IoT - A systematic review [Avances en los enfoques de despliegue y orquestación para IoT: una revisión sistemática]. En *IEEE International Congress on Internet of Things* (pp. 53-60). IEEE. https://www.sintef.no/en/publications/publication/1775272

Okteto. (2023). *Five challenges with developing locally using Docker Compose* [Cinco desafíos al desarrollar localmente con Docker Compose]. Okteto. https://www.okteto.com/blog/five-challenges-with-developing-locally-using-docker-compose/

Olawale, A. (2025). *Part 16: Monitoring with Prometheus and Grafana - Comprehensive guide* [Parte 16: monitoreo con Prometheus y Grafana, guía completa]. Medium. https://medium.com/@Adekola_Olawale/part-16-monitoring-with-prometheus-and-grafana-e3

304355055d

OpenTelemetry. (2025). *What is OpenTelemetry?* [¿Qué es OpenTelemetry?] OpenTelemetry. https://opentelemetry.io/docs/what-is-opentelemetry/

OpenTelemetry Authors. (2024). *OpenTelemetry Java instrumentation documentation* [Documentación de la instrumentación de Java de OpenTelemetry]. OpenTelemetry Documentation. https://opentelemetry.io/docs/instrumentation/java/

Ouyang, R., Wang, J., Xu, H., Chen, S., Xiong, X., Tolba, A. y Zhang, X. (2023). A microservice and serverless architecture for secure IoT system [Una arquitectura de microservicios y sin servidor para un sistema IoT seguro]. *Sensors*, *23*(9), 4559. https://doi.org/10.3390/s23094559

Platform Engineers. (2025). *Microservices deployment strategies: Blue-green, canary and rolling updates* [Estrategias de despliegue de microservicios: Blue-Green, Canary y actualizaciones progresivas]. Medium. https://medium.com/@platform.engineers/microservices-deployment-strategies-blue-green-canary-and-rolling-updates-ead4617ecbf3

PostgreSQL Global Development Group. (2024). *PostgreSQL documentation* [Documentación de PostgreSQL]. PostgreSQL Documentation. https://www.postgresql.org/docs/current/

Prometheus Community. (2024). *prometheus-operator/prometheus-operator: Prometheus Operator for Kubernetes* [prometheus-operator/prometheus-operator: Prometheus Operator para Kubernetes]. GitHub. https://github.com/prometheus-operator/prometheus-operator

Rajković, P., Stefanović, D. y Milosavljević, B. (2022). Hybrid software deployment strategy for complex distributed systems [Estrategia híbrida de despliegue de software para sistemas distribuidos complejos]. *Electronics*, *11*(14), 2186.

Rancher Labs. (2024). *K3s documentation: Lightweight Kubernetes* [Documentación de K3s:

Kubernetes ligero]. K3s Documentation. https://k3s.io/

Savle, J. (2025). *Observability made easy: Distributed tracing with Zipkin & OpenTelemetry* [Observabilidad simplificada: trazabilidad distribuida con Zipkin y OpenTelemetry]. Medium. https://medium.com/@jayeshsavle2026/observability-made-easy-a-practical-guide-to-distributed-tracing-with-zipkin-opentelemetry-6dcee9df3109

Sematext. (2025). *Three pillars of observability: Logs, metrics & traces defined* [Tres pilares de la observabilidad: definición de registros, métricas y trazas]. Sematext. https://sematext.com/glossary/three-pillars-of-observability/

Sharif, A. (2024). *The three pillars of observability: Logs, metrics, and traces* [Los tres pilares de la observabilidad: registros, métricas y trazas]. CrowdStrike. https://www.crowdstrike.com/en-us/cybersecurity-101/observability/three-pillars-of-observability/

Siddiqui, H., Khendek, F. y Toeroe, M. (2023). Microservices based architectures for IoT systems — State-of-the-art review [Arquitecturas basadas en microservicios para sistemas IoT: revisión del estado del arte]. *Internet of Things*, *23*, Artículo 100854. https://doi.org/10.1016/j.iot.2023.100854

Splunk. (2024). *Splunk documentation and pricing* [Documentación y precios de Splunk]. Splunk Documentation. https://docs.splunk.com/

Sridharan, C. (2018). The three pillars of observability [Los tres pilares de la observabilidad]. En *Distributed systems observability* [Observabilidad de sistemas distribuidos]. O'Reilly Media. https://www.oreilly.com/library/view/distributed-systems-observability/9781492033431/ch04.html

Sumo Logic. (2024). *Sumo Logic documentation* [Documentación de Sumo Logic]. Sumo Logic

Documentation. https://help.sumologic.com/

Tozzi, C. (2024). *The 3 pillars of observability: Logs, metrics and traces* [Los 3 pilares de la observabilidad: registros, métricas y trazas]. TechTarget. https://www.techtarget.com/searchitoperations/tip/The-3-pillars-of-observability-Logs-metrics-and-traces

Traefik Labs. (2024). *Traefik documentation* [Documentación de Traefik]. Traefik Documentation. https://doc.traefik.io/traefik/

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E. y Wilkes, J. (2015). Large-scale cluster management at Google with Borg [Gestión de clústeres a gran escala en Google con Borg]. *Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*, Article 18. https://doi.org/10.1145/2741948.2741964

VMware Tanzu. (2024). *RabbitMQ documentation* [Documentación de RabbitMQ]. RabbitMQ Documentation. https://www.rabbitmq.com/docs

Zehri, M. (2025). *Full observability stack with Grafana, Loki, Fluent Bit, Mimir, and OpenTelemetry* [Stack de observabilidad completo con Grafana, Loki, Fluent Bit, Mimir y OpenTelemetry]. Medium. https://medium.com/@montasser.zehri/build-a-full-observability-stack-with-grafana-loki-fluent-bit-mimir-and-opentelemetry-5714d95afbe6

# Apéndices

## Apéndice A. Casos de uso detallados

Este apéndice presenta el catálogo completo de casos de uso de la plataforma Smart Campus UIS, organizados por microservicio y funcionalidad. Complementa la tabla resumen incluida en la sección 5.1.8.

### Tabla A.1

*Casos de uso, microservicio de administración*

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Listar todos los dispositivos | Admin | /device | GET | PostgreSQL | - | - |
| Listar dispositivos con paginación | Admin | /device/pageable | GET | PostgreSQL | - | - |
| Crear dispositivo con sincronización | Admin | /device | POST | PostgreSQL, Redis | - | Gateway IoT (REST) |
| Obtener dispositivo por ID | Admin | /device/{id} | GET | PostgreSQL | - | - |

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Obtener representación de gateway | Admin | /device/gateway/{id} | GET | PostgreSQL | - | - |
| Actualizar dispositivo | Admin | /device/{id} | PUT | PostgreSQL, Redis | - | Gateway IoT (REST) |
| Eliminar dispositivo con sincronización | Admin | /device/{id} | DELETE | PostgreSQL, Redis | - | Gateway IoT (REST) |
| Agregar recurso a dispositivo | Admin | /device/resource/{device_id} | POST | PostgreSQL | - | - |
| Eliminar recurso de dispositivo | Admin | /device/resource/{device_id} | DELETE | PostgreSQL | - | - |
| Agregar propiedad a dispositivo | Admin | /device/property/{device_id} | POST | PostgreSQL | - | - |
| Eliminar propiedad de dispositivo | Admin | /device/property/{device_id} | DELETE | PostgreSQL | - | - |
| Listar todas las aplicaciones | Admin | /apps | GET | PostgreSQL | - | - |
| Crear nueva aplicación | Admin | /apps | POST | PostgreSQL | - | - |
| Eliminar aplicación | Admin | /apps/{id} | DELETE | PostgreSQL | - | - |

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Asociar dispositivo a aplicación | Admin | /apps/device/{app_id} | POST | PostgreSQL | - | - |
| Desasociar dispositivo de aplicación | Admin | /apps/device/{app_id} | DELETE | PostgreSQL | - | - |
| Asociar servicio a aplicación | Admin | /apps/service/{app_id} | POST | PostgreSQL | - | - |
| Desasociar servicio de aplicación | Admin | /apps/service/{app_id} | DELETE | PostgreSQL | - | - |
| Crear nuevo recurso | Admin | /resource | POST | PostgreSQL | - | - |
| Obtener recurso por ID | Admin | /resource/{id} | GET | PostgreSQL | - | - |
| Actualizar recurso | Admin | /resource/{id} | PUT | PostgreSQL | - | - |
| Eliminar recurso | Admin | /resource/{id} | DELETE | PostgreSQL | - | - |
| Agregar propiedad a recurso | Admin | /resource/property/{resource_id} | POST | PostgreSQL | - | - |
| Eliminar propiedad de recurso | Admin | /resource/property/{resource_id} | DELETE | PostgreSQL | - | - |

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Crear nueva propiedad | Admin | /property | POST | PostgreSQL | - | - |
| Obtener propiedad por ID | Admin | /property/{id} | GET | PostgreSQL | - | - |
| Actualizar propiedad | Admin | /property/{id} | PUT | PostgreSQL | - | - |
| Eliminar propiedad | Admin | /property/{id} | DELETE | PostgreSQL | - | - |
| Listar modelos de servicio | Admin | /services | GET | PostgreSQL | - | - |
| Crear nuevo modelo de servicio | Admin | /services | POST | PostgreSQL | - | - |
| Eliminar modelo de servicio | Admin | /services/{id} | DELETE | PostgreSQL | - | - |
| Asociar recurso a servicio | Admin | /services/resource/{service_id} | POST | PostgreSQL | - | - |
| Desasociar recurso de servicio | Admin | /services/resource/{service_id} | DELETE | PostgreSQL | - | - |

**Tabla A.2**

*Casos de uso — recepción de datos, microservicio de datos*

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Recibir mensajes de sensores vía MQTT | Data | device/message | MQTT Subscribe | InfluxDB, MongoDB | EMQX (MQTT) | Gateway IoT |
| Recibir mensajes de sensores vía AMQP | Data | device-message | AMQP Consume | InfluxDB, MongoDB | RabbitMQ (AMQP) | Gateway IoT |
| Re-encolar mensaje a subtópico | Data | {subtopic} | MQTT/AMQP Publish | InfluxDB, MongoDB | EMQX, RabbitMQ | Aplicaciones suscritas |

**Tabla A.3**

*Casos de uso — consulta de datos, microservicio de datos*

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Obtener valor mínimo de métrica | Data | /influx/measurement/{m}/min | GET | InfluxDB | - | - |
| Obtener valor máximo de métrica | Data | /influx/measurement/{m}/max | GET | InfluxDB | - | - |
| Obtener últimas N mediciones | Data | /influx/measurement/{m}/last | GET | InfluxDB | - | - |

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Obtener promedio de métrica en rango | Data | /influx/measurement/{m}/average | GET | InfluxDB | - | - |
| Obtener mediciones agrupadas por tiempo | Data | /influx/date/units/{time} | GET | InfluxDB | - | - |
| Obtener todas las métricas en rango | Data | /influx/by-time-range | GET | InfluxDB | - | - |
| Obtener métrica específica en rango | Data | /influx/by-time-range/measurement/{m} | GET | InfluxDB | - | - |
| Obtener últimos N mensajes de métrica | Data | /mongo/measurement/last | GET | MongoDB | - | - |
| Obtener mensajes de métrica en rango | Data | /mongo/measurement/by-time-range | GET | MongoDB | - | - |
| Obtener mensajes por ubicación | Data | /mongo/location/{location} | GET | MongoDB | - | - |

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Obtener mensajes por dispositivo | Data | /mongo/deviceId/{deviceId} | GET | MongoDB | - | - |
| Obtener mensajes agrupados por tiempo | Data | /mongo/date/units/{time} | GET | MongoDB | - | - |

**Tabla A.4**

*Casos de uso — gestión de archivos, microservicio de datos*

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Subir y almacenar archivo | Data | /file/save | POST | MinIO | - | - |

**Tabla A.5**

*Casos de uso — gestión de cámaras, microservicio de datos*

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Registrar nueva cámara | Data | /camera/add | POST | MongoDB | - | - |
| Listar cámaras registradas | Data | /camera/list | GET | MongoDB | - | - |
| Eliminar cámara | Data | /camera/delete | DELETE | MongoDB | - | - |
| Iniciar transmisión en vivo | Data | /camera/stream?IdCamera={id} | GET | MinIO | - | Cámara RTSP |
| Iniciar grabación de cámara | Data | /camera/start?IdCamera={id} | GET | MinIO, MongoDB | - | Cámara RTSP |
| Pausar grabación de cámara | Data | /camera/pause?IdCamera={id} | GET | MinIO, MongoDB | - | Cámara RTSP |
| Reanudar grabación de cámara | Data | /camera/resume?IdCamera={id} | GET | MinIO, MongoDB | - | Cámara RTSP |
| Detener grabación de cámara | Data | /camera/stop?IdCamera={id} | GET | MinIO, MongoDB | - | Cámara RTSP |

**Tabla A.6**

*Casos de uso, API Gateway*

| Caso de Uso | Microservicio | Endpoint | Método | Base de Datos | Brokers | Otros Componentes |
|---|---|---|---|---|---|---|
| Enrutar peticiones a Admin | Gateway | /admin/* | Proxy HTTP | - | - | Admin Microservice |
| Enrutar peticiones a Data | Gateway | /data/* | Proxy HTTP | - | - | Data Microservice |

## Apéndice B. Dockerfile original del microservicio de datos

Se presenta el Dockerfile original del microservicio de datos previo a las optimizaciones descritas en la sección 7.2.1.2. Este archivo constituye la línea base sobre la que se aplicaron las tres mejoras documentadas: reemplazo de JDK por JRE, caché de dependencias Maven e instalación mínima de ffmpeg.

**Figura B.1**

*Dockerfile original del microservicio de datos*

```
#Utilizamos la imagen de maven para crear el artefacto desplegable (jar) del
proyecto
FROM maven:3.9.9-eclipse-temurin-21-jammy AS builder

#Instalamos git para clonar el repositorio
RUN apt-get update \
  && apt-get install -y git \
  && rm -rf /var/lib/apt/lists/*
```

```
#Clonamos el repositorio data_microservice
COPY . /data_microservice

# Establecemos el directorio de trabajo dentro del contenedor
WORKDIR /data_microservice

# Compilamos el proyecto y generamos el archivo .jar del modulo application,
omitiendo los tests
RUN mvn clean package -DskipTests -pl application -am

# Usamos una imagen base mas liviana con JDK 21 para ejecutar el .jar
FROM eclipse-temurin:21-jdk-jammy

# Instalamos ffmpeg (necesario para JavaCV)
RUN apt-get update \
    && DEBIAN_FRONTEND=noninteractive apt-get install -y ffmpeg \
    && rm -rf /var/lib/apt/lists/ \
    && which ffmpeg

# Establecemos el directorio de trabajo de la aplicacion
WORKDIR /app
RUN mkdir -p /app/application

# Copiamos el .jar generado desde la imagen 'builder'
COPY --from=builder /data_microservice/application/target/*.jar app.jar

# Exponemos el puerto 8080, donde se ejecuta el microservicio
EXPOSE 8080

# Arrancamos el microservicio
ENTRYPOINT["java","-jar","app.jar"]
```

**Apéndice C. Dockerfile después de la optimización**

Se presenta el nuevo Dockerfile del microservicio de datos para facilitar la comparación directa entre la versión original y la versión optimizada. El Dockerfile original corresponde al estado del repositorio previo a este trabajo; el optimizado incorpora las tres mejoras descritas en la sección 7.2.1.2.

**Figura C.1**

*Dockerfile optimizado del microservicio de datos*

```
# syntax=docker/dockerfile:1.4

##########################
# Stage 1: Build
##########################
FROM maven:3.9.9-eclipse-temurin-21-jammy AS builder
WORKDIR /build

# 1. Copiar solo los POMs de todos los modulos
COPY pom.xml ./
COPY application/pom.xml application/
COPY domain/pom.xml domain/
COPY persistence/pom.xml persistence/
COPY service/pom.xml service/

# 2. Descargar dependencias en cache
RUN --mount=type=cache,target=/root/.m2 \
    mvn -B -ntp dependency:go-offline -pl application -am

# 3. Copiar el codigo fuente completo
COPY . .

# 4. Compilar solo el modulo application (y los necesarios)
RUN --mount=type=cache,target=/root/.m2 \
    mvn -B -ntp clean package -DskipTests -pl application -am

##########################
# Stage 2: Runtime
##########################
FROM eclipse-temurin:21-jre-jammy

# Instalar ffmpeg (minimo posible, sin paquetes recomendados)
RUN apt-get update \
 && DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends ffmpeg
\
 && rm -rf /var/lib/apt/lists/*

WORKDIR /app
COPY --from=builder /build/application/target/*.jar /app/app.jar

EXPOSE 8080
```

```
ENTRYPOINT ["java", "-XX:+UseContainerSupport", "-XX:MaxRAMPercentage=75.0", "-jar",
"/app/app.jar"]
```

## Apéndice D. docker-compose.yml completo

Se presenta el archivo docker-compose.yml completo de Smart Campus UIS, adaptado para usar las imágenes publicadas en DockerHub. Este archivo levanta el sistema completo con un único comando (docker compose up -d) sin requerir construcción local de imágenes ni instalación de dependencias adicionales.

**Figura D.1**

*Archivo docker-compose.yml completo*

```
services:
  postgres:
    image: postgres:16
    restart: always
    shm_size: 128mb
    env_file: .env
    ports:
     - 5432:5432
    environment:
      - POSTGRES_USER=${POSTGRESQL_USER}
      - POSTGRES_DB=${POSTGRESQL_DB}
      - POSTGRES_PASSWORD=${POSTGRESQL_PASSWORD}
    volumes:
    - ./data/postgres-data:/var/lib/postgresql/data
    networks:
      - iot_campus_uis

  influxdb:
    image: influxdb:2.7.11
    restart: always
    env_file:
      - .env
    ports:
      - 8086:8086
    environment:
```

```
      - DOCKER_INFLUXDB_INIT_MODE=setup
      - DOCKER_INFLUXDB_INIT_USERNAME=${INFLUXDB_USER}
      - DOCKER_INFLUXDB_INIT_PASSWORD=${INFLUXDB_PASSWORD}
      - DOCKER_INFLUXDB_INIT_ORG=${INFLUXDB_ORG}
      - DOCKER_INFLUXDB_INIT_BUCKET=${INFLUXDB_BUCKET}
      - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=${INFLUXDB_TOKEN}
    volumes:
      - ./data/influxdb/data:/var/lib/influxdb2
      - ./data/influxdb/config:/etc/influxdb2
    networks:
      - iot_campus_uis

  mongodb:
    image: mongo
    restart: always
    env_file:
      - .env
    ports:
      - 27017:27017
    environment:
      - MONGO_INITDB_ROOT_USERNAME=${MONGO_USERNAME}
      - MONGO_INITDB_ROOT_PASSWORD=${MONGO_PASSWORD}
      - MONGO_INITDB_DATABASE=${MONGO_DATABASE}
    volumes:
      - ./data/mongodb/data:/data/db
    networks:
      - iot_campus_uis

  minio:
    image: quay.io/minio/minio
    restart: always
    env_file:
      - .env
    ports:
      - "9000:9000"
      - "9001:9001"
    environment:
      - MINIO_ROOT_USER=${MINIO_ROOT_USER}
      - MINIO_ROOT_PASSWORD=${MINIO_ROOT_PASSWORD}
    volumes:
      - ./data/minio/data:/data
    command: server /data --console-address ":9001"
    networks:
      - iot_campus_uis

  emqx:
    user: root
```

```
    image: "emqx/emqx:5.8"
    restart: always
    container_name: emqx
    ports:
      - 1883:1883
      - 8083:8083
      - 8084:8084
      - 8883:8883
      - 18083:18083
    volumes:
      - ./data/emqx/data:/opt/emqx/data
      - ./data/emqx/log:/opt/emqx/log
    networks:
      - iot_campus_uis

  rabbitmq:
    image: rabbitmq:4.0-management
    restart: always
    env_file:
      - .env
    ports:
      - 5672:5672
      - 15672:15672
    environment:
      - RABBITMQ_DEFAULT_USER=${RABBITMQ_USERNAME}
      - RABBITMQ_DEFAULT_PASS=${RABBITMQ_PASSWORD}
    networks:
      - iot_campus_uis

  admin:
    image: semillerocage/admin-service:1.0
    restart: always
    depends_on:
      - postgres
    environment:
      - POSTGRESQL_HOST=postgres
      - POSTGRESQL_PORT=${POSTGRESQL_PORT}
      - POSTGRESQL_DB=${POSTGRESQL_DB}
      - POSTGRESQL_USER=${POSTGRESQL_USER}
      - POSTGRESQL_PASSWORD=${POSTGRESQL_PASSWORD}
      - ADMIN_SERVICE_PORT=${ADMIN_SERVICE_PORT}
    networks:
      - iot_campus_uis

  data:
    image: semillerocage/data-service:1.0
    restart: always
```

```
    env_file:
      - .env
    ports:
      - 8082:8082
    environment:
      - RABBITMQ_HOST=${RABBITMQ_HOST}
      - RABBITMQ_PORT=${RABBITMQ_PORT}
      - RABBITMQ_USERNAME=${RABBITMQ_USERNAME}
      - RABBITMQ_PASSWORD=${RABBITMQ_PASSWORD}
      - MQTT_BROKER_URL=${MQTT_BROKER_URL}
      - MQTT_BROKER_CLIENT_ID=${MQTT_BROKER_CLIENT_ID}
      - TOPIC_LISTENER=${TOPIC_LISTENER}
      - INFLUXDB_URL=${INFLUXDB_URL}
      - INFLUXDB_TOKEN=${INFLUXDB_TOKEN}
      - INFLUXDB_ORG=${INFLUXDB_ORG}
      - INFLUXDB_BUCKET=${INFLUXDB_BUCKET}
      - MONGO_HOST=${MONGO_HOST}
      - MONGO_PORT=${MONGO_PORT}
      - MONGO_DATABASE=${MONGO_DATABASE}
      - MONGO_AUTH_DB=${MONGO_AUTH_DB}
      - MONGO_USERNAME=${MONGO_USERNAME}
      - MONGO_PASSWORD=${MONGO_PASSWORD}
      - MINIO_URL=${MINIO_URL}
      - MINIO_ACCES_KEY=${MINIO_ROOT_USER}
      - MINIO_SECRET_KEY=${MINIO_ROOT_PASSWORD}
      - VIDEO_DURATION_MINUTES=${VIDEO_DURATION_MINUTES}
    depends_on:
      - influxdb
      - mongodb
      - minio
      - rabbitmq
      - emqx
    networks:
      - iot_campus_uis
    extra_hosts:
      - "host.docker.internal:host-gateway"

  gateway:
    restart: always
    image: semillerocage/go-gateway:1.0
    depends_on:
      - admin
    ports:
      - 8080:8080
    environment:
      - ADMIN_HOST=http://admin:${ADMIN_SERVICE_PORT}
      - DATA_HOST=http://data:${DATA_SERVICE_PORT}
```

```
      - GATEWAY_PORT=${GATEWAY_PORT}
    networks:
      - iot_campus_uis

  frontend:
    restart: always
    image: semillerocage/front-end-admin:1.0
    depends_on:
      - admin
      - gateway
      - data
    ports:
      - 4000:80
    networks:
      - iot_campus_uis

  grafana:
    image: "grafana/grafana:11.5.0"
    container_name: grafana
    user: "$UID:$GID"
    ports:
      - "3000:3000"
    volumes:
      - '$PWD/data/grafana:/var/lib/grafana'
    networks:
      - iot_campus_uis
    depends_on:
      - influxdb

networks:
  iot_campus_uis:
    driver: bridge
```

## Apéndice E. Patrones de logging estructurado

Se presentan los patrones de configuración de logging estructurado implementados en ambos microservicios, descritos en la sección 7.3.3.

**Figura E.1**

*Patrón de salida — logback-spring.xml*

Patrón de formato de salida configurado en logback-spring.xml para ambos microservicios:

```
<pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %5level
[${appName},%X{trace_id:-},%X{span_id:-}] [%thread] %logger{36} - %msg%n</pattern>
```

**Figura E.2**

*Patrón de formato de salida, logback-spring.xml*

Patrón de nivel de log configurado en los archivos de propiedades de cada servicio:

```
logging.pattern.level=%5p [${spring.application.name},%X{trace_id:-},%X{span_id:-}]
```

**Figura E.3**

*Patrón de nivel de log, application.properties / application.yml*

```
management.endpoints.web.exposure.include=health,info,prometheus
management.metrics.tags.application=data
```

```
management:
  endpoints:
    web:
      exposure:
        include: health, info, prometheus
  metrics:
       tags:
       application: admin
```

## Apéndice F. Archivos de configuración centralizada

Se presentan los archivos de valores de Helm utilizados para el despliegue de Smart Campus UIS, descritos en la sección 8.1.

**Figura F.1**

*Values.yaml*

```yaml
# ── Credenciales ──────────────────────────────────────────
postgres:
  enabled: true
  user: postgres
  password: IoTPass1234
  db: iot

mongo:
  enabled: true
  username: root
  password: password
  database: iot

influxdb:
  enabled: true
  user: admin
  password: admin1234
  org: smart-campus
  bucket: messages
  token:
"8_kiyghJzhqlZENCLnTbc-90hkNfq1U5DS4pRCdMPQdhLY48LW33hsM73nQl1zs_sLVc5IPvJNWsUHZHimZfJw=="

minio:
  enabled: true
  rootUser: admin
  rootPassword: password

rabbitmq:
  enabled: true
  username: admin
  password: password

emqx:
```

```
  enabled: true

# ── Origen de imágenes ──────────────────────────────────────
# Fallback global -- aplica si un servicio no define su propio imageSource.
# local  -- imagen en el containerd del nodo  (imagePullPolicy: Never)
# remote -- imagen desde DockerHub             (imagePullPolicy: IfNotPresent)
imageSource: remote

# ── Imágenes ────────────────────────────────────────────────
dataService:
  enabled: true
  image: semillerocage/data-service
  tag: "1.2"
  imageSource: remote

adminService:
  enabled: true
  image: semillerocage/admin-service
  tag: "1.2"
  imageSource: remote
```

**Figura F.2**

*Values-custom.yaml*

```
postgres:
  enabled: true
mongo:
  enabled: true
influxdb:
  enabled: true
minio:
  enabled: true
rabbitmq:
  enabled: true
emqx:
  enabled: true
dataService:
  enabled: true
adminService:
  enabled: true

# imageSource: local  -- usa imagen del containerd del nodo
# imageSource: remote -- descarga desde DockerHub
dataService:
  imageSource: local
```

```
  image: data-service
  tag: latest

adminService:
  imageSource: local
  image: admin-service
  tag: latest
```

## Apéndice G. Ejemplo de init containers en templates (Deploymen/ Resultset)

Se presenta un ejemplo representativo de la configuración de init containers utilizada en los templates de despliegue para garantizar el orden de arranque de las dependencias.

**Figura G.1**

*Ejemplo de init containers en templates de despliegue*

```
      initContainers:
        - name: wait-for-rabbitmq
          image: busybox
          command: ['sh', '-c', 'until nc -z rabbitmq 5672; do echo waiting; sleep
2; done']
        - name: wait-for-emqx
          image: busybox
          command: ['sh', '-c', 'until nc -z emqx 1883; do echo waiting; sleep 2;
done']
        - name: wait-for-mongodb
          image: busybox
          command: ['sh', '-c', 'until nc -z mongodb 27017; do echo waiting; sleep
2; done']
        - name: wait-for-influxdb
          image: busybox
          command: ['sh', '-c', 'until nc -z influxdb 8086; do echo waiting; sleep
2; done']
        - name: wait-for-minio
          image: busybox
          command: ['sh', '-c', 'until nc -z minio 9000; do echo waiting; sleep 2;
done']
```

## Apéndice H. Configuraciones de métricas

Se presentan las configuraciones relevantes para la recolección y visualización de métricas, incluyendo Alloy, Prometheus y Grafana, descritas en la sección 8.2.

### Figura H.1

*Configuración de Alloy para metricas*

```
discovery.relabel "spring_metrics" {
  targets = discovery.kubernetes.pods.targets
  rule {
    source_labels = ["__meta_kubernetes_namespace"]
    regex         = "smart-campus"
    action        = "keep"
  }
  rule {
    source_labels = ["__meta_kubernetes_pod_annotation_prometheus_io_scrape"]
    regex         = "true"
    action        = "keep"
  }
  rule {
    source_labels = ["__meta_kubernetes_pod_ip",
                     "__meta_kubernetes_pod_annotation_prometheus_io_port"]
    separator     = ":"
    target_label  = "__address__"
  }
}

prometheus.scrape "spring" {
  targets         = discovery.relabel.spring_metrics.output
  forward_to      = [prometheus.remote_write.local.receiver]
  scrape_interval = "30s"
}

prometheus.remote_write "local" {
  endpoint {
    url =
"http://monitoring-prometheus-server.monitoring.svc.cluster.local/api/v1/write"
  }
}
```

**Figura H.2**

*Configuración básica de Prometheus*

```yaml
prometheus:
  server:
    retention: "15d"
    persistentVolume:
      enabled: true
      size: 8Gi
    extraArgs:
      web.enable-remote-write-receiver: null
  alertmanager:
    enabled: true
  prometheus-pushgateway:
    enabled: false
  kube-state-metrics:
    enabled: false
  prometheus-node-exporter:
    enabled: false
```

**Figura H.3**

*Configuración de Grafana para métricas*

```yaml
grafana:
  datasources:
    datasources.yaml:
      apiVersion: 1
      datasources:
        - name: Prometheus
          type: prometheus
          uid: prometheus
          url: http://monitoring-prometheus-server.monitoring.svc.cluster.local
          access: proxy
       isDefault: false
```

## Apéndice I. Configuración de alertas

Se presenta la configuración completa de las reglas de alerta definidas en el archivo de valores del chart, descritas en la sección 8.2.6.

**Figura I.1**

*Configuración de reglas de alerta*

```
prometheus:
  serverFiles:
    alerting_rules.yml:
      groups:
       - name: smart-campus
       interval: 30s
       rules:

              - alert: ServicioCaido
              expr: absent(up{app="admin"}) or up{app="admin"} == 0
                     or absent(up{app="data"}) or up{app="data"} == 0
              for: 1m
              labels:
              severity: critical
              annotations:
              summary: "Servicio caido"
              description: "Un pod de admin o data lleva mas de 1 minuto sin
responder."

              - alert: MemoriaHeapCritica
              expr: >
              sum by (app) (jvm_memory_used_bytes{area="heap", app=~"admin|data"})
              /
              sum by (app) (clamp_min(jvm_memory_max_bytes{area="heap",
app=~"admin|data"}, 1))
              > 0.90
              for: 5m
              labels:
              severity: critical
              annotations:
              summary: "Heap JVM critico"
              description: "El heap de un microservicio esta por encima del 90%."
```

```
              - alert: CPUAltaSostenida
              expr: avg by (app) (process_cpu_usage{app=~"admin|data"}) > 0.80
              for: 5m
              labels:
              severity: warning
              annotations:
              summary: "CPU alta sostenida"
              description: "Un microservicio lleva 5 minutos usando mas del 80% de
CPU."

              - alert: ErroresHTTP
              expr: >
              sum by (app, uri, status)
                  (rate(http_server_requests_seconds_count{app=~"admin|data",
status=~"4..|5.."}[5m]))
              > 0.1
              for: 2m
              labels:
              severity: warning
              annotations:
              summary: "Errores HTTP detectados"
              description: "Revisar labels app, uri y status para identificar el
endpoint afectado."
```

## Apéndice J. Configuraciones de logs

Se presentan las configuraciones relevantes para la recolección de logs y trazas distribuidas, incluyendo Alloy, Loki, Tempo y Grafana, descritas en la sección 8.3.

### Figura J.1

*Configuración de Alloy para Loki*

```
discovery.relabel "pod_logs" {
  targets = discovery.kubernetes.pods.targets
  rule {
    source_labels = ["__meta_kubernetes_namespace"]
    regex         = "smart-campus|monitoring"
    action        = "keep"
  }
}
```

```
loki.source.kubernetes "pods" {
  targets    = discovery.relabel.pod_logs.output
  forward_to = [loki.write.default.receiver]
}
```

**Figura J.2**

*Configuración de Alloy para Tempo*

```
otelcol.processor.tail_sampling "drop_actuator" {
  decision_wait             = "5s"
  num_traces                = 1000
  expected_new_traces_per_sec = 10

  policy {
    name = "drop-actuator"
    type = "string_attribute"
    string_attribute {
      key            = "http.route"
      values         = ["/actuator/health", "/actuator/prometheus",
                        "/actuator/health/liveness", "/actuator/health/readiness"]
      invert_match = true
       }
  }
}
```

**Figura J.3**

*Configuración Loki*

```
loki:
  deploymentMode: SingleBinary
  loki:
    limits_config:
      retention_period: 168h
    compactor:
      working_directory: /var/loki/compactor
      compaction_interval: 10m
      retention_enabled: true
      delete_request_store: s3
    storage:
```

```
      type: s3
      s3:
        endpoint: http://minio-monitoring.monitoring.svc.cluster.local:9000
  singleBinary:
    initContainers:
      - name: create-loki-bucket
        image: minio/mc:latest
        command:
          - sh
          - -c
          - |
              until mc alias set local
http://minio-monitoring.monitoring.svc.cluster.local:9000 admin password; do
              echo "waiting for minio...";
              sleep 3;
              done;
              mc mb --ignore-existing local/loki;
              echo "bucket loki ready"
  gateway:
    affinity: {}
```

**Figura J.4**

*Configuración Tempo*

```
tempo:
  tempo:
    retention: 72h
    storage:
      trace:
        backend: local
        local:
          path: /var/tempo/traces
        block:
          version: vParquet4
  persistence:
    enabled: true
    storageClassName: "local-path"
    size: 5Gi
```

**Figura J.5**

*Configuración de Grafana para observabilidad*

```
grafana:
  datasources:
    datasources.yaml:
      apiVersion: 1
      datasources:
        - name: Loki
          type: loki
          uid: loki
          url: http://monitoring-loki.monitoring.svc.cluster.local:3100
          jsonData:
              derivedFields:
              - name: TraceID
              matcherRegex: '\[[\w-]+,([a-fA-F0-9]{16,32}),'
              datasourceUid: tempo
        - name: Tempo
          type: tempo
          uid: tempo
          url: http://monitoring-tempo.monitoring.svc.cluster.local:3100
          jsonData:
              tracesToLogsV2:
              datasourceUid: loki
              filterByTraceID: true
```

**Figura J.6**

*Configuración values custom*

```
observability:
  lowPriority: true   # asigna PriorityClass de valor 100 al stack

# nodeAffinity preferred -- sin efecto en un nodo único,
# pero permite migración automática a nodo role=observability en el futuro
alloy:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
```

```
            matchExpressions:
            - key: role
            operator: In
            values:
                  - observability
```

## Apéndice K. Dashboards

Se presenta la configuración del mecanismo de descubrimiento automático de dashboards en Grafana y un ejemplo representativo de la definición de un dashboard como ConfigMap, descritos en la sección 8.4.1.

**Figura K.1**

*Configuración de Grafana para Dashboards*

```
grafana:
   sidecar:
       dashboards:
       enabled: true
       label: grafana_dashboard
       labelValue: "1"
       searchNamespace: monitoring
```

**Figura K.2**

*Configuración de un Dashboard*

```
apiVersion: v1
 kind: ConfigMap
 metadata:
   name: dashboard-smart-campus-alerts
   namespace: monitoring
   labels:
       grafana_dashboard: "1"
 data:
```

```
  smart-campus-alerts.json: |
      {
      "title": "Smart Campus - Alertas y rendimiento",
      "uid": "smart-campus-alerts",
      "panels": [
      {
             "id": 1,
             "title": "Estado de servicios",
             "type": "stat",
             "datasource": { "type": "prometheus", "uid": "prometheus" },
             "targets": [
             { "expr": "max by (app) (up{app=~\"admin|data\"})" }
             ],
             "fieldConfig": {
             "defaults": {
             "mappings": [
             { "options": { "0": { "text": "CAIDO", "color": "red" },
                              "1": { "text": "OK", "color": "green" } },
                    "type": "value" }
             ]
             }
             }
      }
      ]
      }
```

## Apéndice L. Link del repositorio

El código fuente de los charts de Helm y de las configuraciones de la plataforma Smart Campus UIS se encuentra disponible en el siguiente repositorio de GitHub:

https://github.com/SimpleCroasan/smart-campus-charts