# UNIVERSIDAD TÉCNICA DEL NORTE

## FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

## CARRERA DE TECNOLOGÍAS DE LA INFORMACIÓN

### TRABAJO DE INTEGRACIÓN CURRICULAR

**TEMA:**

"AUTOMATIZACIÓN DEL PROCESO DE INTEGRACIÓN CONTINUA Y DESPLIEGUE CONTINUO CON ENFOQUE EN MICROSERVICIOS"

Trabajo de titulación previo a la obtención del título en Ingeniero en Tecnologías de la Información

**Línea de investigación:** Desarrollo, aplicación de software y cyber security (seguridad cibernética)


**AUTOR:**

Wilmer Paolo Guairacaja Ipiales

**DIRECTOR:**

MSc. Pablo Andrés Landeta López


**Ibarra – Ecuador 2026**

# UNIVERSIDAD TÉCNICA DEL NORTE
## DIRECCIÓN DE BIBLIOTECA

## AUTORIZACIÓN DE USO Y PUBLICACIÓN A FAVOR DE LA UNIVERSIDAD TÉCNICA DEL NORTE

### 1. IDENTIFICACIÓN DE LA OBRA

En cumplimiento del Art. 144 de la Ley de Educación Superior, hago la entrega del presente trabajo a la Universidad Técnica del Norte para que sea publicado en el Repositorio Digital Institucional, para lo cual pongo a disposición la siguiente información:

| DATOS DE CONTACTO | |
|---|---|
| **APELLIDOS Y NOMBRES:** | Guairacaja Ipiales Wilmer Paolo |

| DATOS DE LA OBRA | |
|---|---|
| **TÍTULO:** | Automatización del proceso de integración continua y despliegue continuo con enfoque en microservicios |
| **AUTOR (ES):** | Wilmer Paolo Guairacaja Ipiales |
| **FECHA: DD/MM/AAAA** | 04/09/2026 |
| SOLO PARA TRABAJOS DE GRADO | |
| **PROGRAMA:** | ☑ **PREGRADO** ☐ **POSGRADO** |
| **TITULO POR EL QUE OPTA:** | **Ingeniero en Tecnologías de la Información** |
| **DIRECTOR /ASESOR:** | **MSc. Pablo Andrés Landeta López, MSc. Diego Javier Trejo España** |

### 2. CONSTANCIAS

El autor (es) manifiesta (n) que la obra objeto de la presente autorización es original y se la desarrolló, sin violar derechos de autor de terceros, por lo tanto la obra es original y que es (son) el (los) titular (es) de los derechos patrimoniales, por lo que asume (n) la responsabilidad sobre el contenido de la misma y saldrá (n) en defensa de la Universidad en caso de reclamación por parte de terceros.

Ibarra, a los 04 días del mes de septiembre de 2026

**EL AUTOR:**

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
WILMER PAOLO GUAIRACAJA IPIALES

.....

Nombre: Wilmer Paolo Guairacaja Ipiales

# CERTIFICACIÓN DEL DIRECTOR DEL TRABAJO DE INTEGRACIÓN CURRICULAR

Ibarra, 04 de septiembre de 2026.

MSc. Pablo Andrés Landeta López
DIRECTOR DEL TRABAJO DE INTEGRACIÓN CURRICULAR

CERTIFICA:
Haber revisado el presente informe final del trabajo de Integración Curricular, el mismo que se ajusta a las normas vigentes de la Universidad Técnica del Norte; en consecuencia, autorizo su presentación para los fines legales pertinentes.

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
PABLO ANDRES LANDETA LOPEZ

*MSc. Pablo Andrés Landeta López*

*C.C: 1002161055*

# APROBACIÓN DEL COMITÉ CALIFICADOR

El Comité Calificado del trabajo de Integración Curricular “ Automatización del proceso de integración continua y despliegue continuo con enfoque en microservicios ” elaborado por Wilmer Paolo Guairacaja Ipiales , previo a la obtención del título del Ingeniero en Tecnologías de la Información , aprueba el presente informe de investigación en nombre de la Universidad Técnica del Norte:

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
PABLO ANDRES LANDETA LOPEZ

*MSc. Pablo Andrés Landeta López*

*C.C: 1002161055*

Validar únicamente en FirmaEC.
Firmado electrónicamente por:
DIEGO JAVIER TREJO ESPANA

*MSc. Diego Javier Trejo España*

*C.C: 1002149290*

# DEDICATORIA

Dedico esta tesis a mis seres queridos que han estado apoyándome, pero en especial a Dios por iluminar mi camino, brindarme salud, sabiduría y fortaleza para superar cada desafío presentado durante este camino académico.

A mi mamá y papá por apoyarme íntegramente, por su amor incondicional, consejos, esfuerzo y sacrificio por criar a un hijo que, a pesar de sus errores, los honra ayer, hoy y siempre con este logro que ellos siempre han soñado.

A mis colegas de trabajo que me han compartido su sabiduría, por su compañerismo, motivación y apoyo durante el desarrollo de este proyecto, contribuyen con ánimos y experiencias que enriquecieron este proceso.

Este logro no solo representa mi esfuerzo, sino el de todos aquellos que estuvieron presentes a mi lado apoyándome.

Wilmer Paolo Guairacaja Ipiales

## AGRADECIMIENTO

Hoy día expreso mi más sincero agradecimiento a Dios por ayudarme a no rendirme cuando estaba a punto de echar todo por la borda.

A mis padres, por su amor, paciencia, tenacidad, apoyo incondicional y sacrificio constante, quienes han sido y siempre serán el pilar fundamental de mi vida y carrera profesional.

A mis abuelitos, quienes hoy no están, pero sé que están velando por mi bien. También a mis tías y tíos que me vieron crecer y seguir adelante.

A mis colegas de trabajo que han compartido su conocimiento, experiencias y brindarme su apoyo y motivación durante el desarrollo de esta investigación.

De igual manera, agradezco a mis docentes y a la institución educativa por los conocimientos impartidos y las oportunidades brindadas para mi crecimiento profesional.

Finalmente, agradezco a las personas que de una u otra forma contribuyeron al cumplimiento de este objetivo académico.

Wilmer Paolo Guairacaja Ipiales

# ÍNDICE DE CONTENIDOS

DEDICATORIA . . . . . 7
AGRADECIMIENTO . . . . . 8
ÍNDICE DE FIGURAS . . . . . 14
ÍNDICE DE TABLAS . . . . . 15
RESUMEN . . . . . 17
ABSTRACT . . . . . 18
CAPÍTULO I
1.1 Planteamiento del Problema . . . . . 19
1.2 Objetivos . . . . . 20
1.2.1 Objetivo General . . . . . 20
1.2.2 Objetivos Específicos . . . . . 21
1.3 Alcance y delimitación . . . . . 21
1.4 Justificación . . . . . 23
CAPÍTULO II
2.1 Bases teóricas . . . . . 25
2.1.1 Arquitecturas de Software: Evolución hacia Microservicios . . . . . 25
2.1.2 DevOps y Automatización de Ciclos de Vida . . . . . 34
2.1.3 Patrones y Herramientas para CI/CD en Microservicios . . . . . 41
2.1.4 Calidad y Seguridad en Pipelines Automatizados . . . . . 49
2.1.5 Observabilidad y Monitoreo de Microservicios . . . . . 54
2.1.6 CI/CD en Contextos Educativos . . . . . 65
2.2 Antecedentes . . . . . 68

CAPÍTULO III

3.1 Tipo de investigación . . . . . 71
3.1.1 Enfoque metodológico . . . . . 71
3.1.2 Variables de estudio . . . . . 72
3.2 Materiales, herramientas e insumos utilizados . . . . . 74
3.2.1 Herramientas de desarrollo . . . . . 74
3.2.2 Herramientas de contenerización y despliegue . . . . . 75
3.2.3 Herramientas de automatización CI/CD . . . . . 75
3.2.4 Herramientas de observabilidad . . . . . 75
3.2.5 Infraestructura y entorno de ejecución . . . . . 76
3.3 Planificación del proyecto por sprints . . . . . 76
3.3.1 Estructura general de los sprints . . . . . 76
3.3.2 Sprint 1: Análisis de requerimientos y diseño inicial . . . . . 77
3.3.3 Sprint 2: Implementación de microservicios backend . . . . . 77
3.3.4 Sprint 3: Desarrollo del frontend e integración inicial . . . . . 78
3.3.5 Sprint 4: Persistencia, autenticación y gestión de archivos . . . . . 78
3.3.6 Sprint 5: Contenerización y despliegue semi local . . . . . 78
3.3.7 Sprint 6: Automatización del pipeline CI/CD . . . . . 79
3.3.8 Sprint 7: Observabilidad y monitoreo . . . . . 79
3.3.9 Sprint 8: Validación y ajustes finales . . . . . 80
3.4 Definición de Requisitos del Sistema . . . . . 80

3.4.1 Requisitos Funcionales . . . . . 80
3.4.2 Requisitos No Funcionales . . . . . 81
3.5 Diseño de la arquitectura de la solución . . . . . 83
3.5.1 Arquitectura basada en microservicios . . . . . 83
3.5.2 Componentes principales del sistema . . . . . 84
3.5.3 Comunicación entre componentes . . . . . 84
3.5.4 Principios de diseño aplicados . . . . . 85
3.5.5 Diagrama general de la arquitectura . . . . . 85
3.6 Diseño de persistencia y almacenamiento . . . . . 86
3.6.1 Estrategia de poliglotismo de persistencia . . . . . 87
3.6.2 Persistencia relacional con PostgreSQL . . . . . 87
3.6.3 Persistencia documental con MongoDB . . . . . 88
3.6.4 Gestión de materias como entidad de apoyo . . . . . 89
3.6.5 Persistencia de archivos y almacenamiento permanente . . . . . 90
3.6.6 Diagrama lógico de persistencia . . . . . 92
3.7 Diseño e implementación del pipeline CI/CD . . . . . 92

3.7.1 Objetivo del pipeline automatizado. . . . . 92
3.7.2 Diseño del flujo de integración continua. . . . . 93
3.7.3 Automatización de pruebas . . . . . 94
3.7.4 Construcción automatizada de artefactos . . . . . 95
3.7.5 Diseño del flujo de despliegue continuo . . . . . 95
3.7.6 Configuración del workflow en GitHub Actions . . . . . 96
3.7.7 Publicación y versionado de imágenes . . . . . 105
3.7.8 Despliegue automatizado en entorno semi local . . . . . 106
3.7.9 Estrategia de rollback y recuperación . . . . . 107
3.7.10 Beneficios operativos del pipeline implementado . . . . . 107
3.8 Infraestructura de despliegue semilocal e infraestructura como código . . . . . 107
3.8.1 Diseño de la infraestructura de despliegue semilocal . . . . . 108
3.8.2 Manifiestos de Kubernetes . . . . . 108
3.8.3 Estrategia de alta disponibilidad . . . . . 109
3.8.4 Exposición de servicios . . . . . 110
3.8.5 Persistencia y almacenamiento en Kubernetes . . . . . 110
3.8.6 Ventajas del enfoque IaC aplicado . . . . . 111
3.9 Implementación de observabilidad y monitoreo . . . . . 111

3.9.1 Diseño del enfoque de observabilidad . . . . . 111
3.9.2 Recolección de métricas con Prometheus . . . . . 112
3.9.3 Visualización y análisis con Grafana . . . . . 112
3.9.4 Integración entre monitoreo y pipeline . . . . . 113
CAPÍTULO IV
4.1 Resultados obtenidos . . . . . 115
4.1.1 Resultados de la implementación del pipeline CI/CD . . . . . 115
4.1.2 Evaluación de métricas del sistema. . . . . 116
4.2 Análisis de resultados . . . . . 140
4.3 Cumplimiento de objetivos . . . . . 142
4.3.1 Cumplimiento de objetivos específicos. . . . . 142
CONCLUSIONES . . . . . 144
RECOMENDACIONES . . . . . 145
BIBLIOGRAFÍA . . . . . 146
ANEXOS . . . . . 155

# ÍNDICE DE FIGURAS


Figura 1 Diegrama del problema. Elaboración propia . . . . . 20
Figura 2 Elaborado por Montoya y Ocampo de Pragma. Proceso de integracion continua y despliegue continuo [6]. . . . . 22

Figura 3 Ejemplo de una arquitectura monolítica [11]. . . . . 27

Figura 4 Diagrama general de la arquitectura de la solución . . . . . 86
Figura 5 Código PVC para PostgreSQL. Elaboración propia . . . . . 88
Figura 6 Código PVC para MongoDB. Elaboración propia . . . . . 89
Figura 7 Configuración Multer. Elaboración propia. . . . . 91
Figura 8 Configuración para PVC Tareas. Elaboracion propia. . . . . 91
Figura 9 Diagrama lógico sobre cómo interactúa el PVC . . . . . 92
Figura 10 Flujograma del Pipeline CI. Elaboración propia . . . . . 94
Figura 11 Construccion de imagenes. Elaboración propia . . . . . 95
Figura 12 Flujograma del Pipeline CD. Elaboración propia . . . . . 96
Figura 13 Despliegue automatizado. Elaboración propia . . . . . 106
Figura 14 Configuración de descarga. Elaboración propia . . . . . 106
Figura 15 Configuración para crear BD, tablas, roles. Elaboración propia . . . . . 109
Figura 16 Configuración inicial. Elaboración propia. . . . . 112

Figura 17 Cantidad de jobs ejecutados. Elaboración propia. . . . . 118
Figura 18 Prueba inicial de calidad de código. Elaboración propia. . . . . 121
Figura 19 Implementación de test para login. Elaboración propia . . . . . 122
Figura 20 Implementación de test para autenticación. Elaboración propia. . . . . 122
Figura 21 Implementación de test para servicios Auth. Elaboración propia. . . . . 123
Figura 22 Test con las implementaciones para cobertura JEST. Elaboración propia. 123
Figura 23 Consumo de CPU. Elaboración propia. . . . . 126
Figura 24 Consumo de memoria RAM de los microservicios backend (Prometheus + Grafana). Elaboración propia. . . . . 127
Figura 25 Evolución temporal del consumo de CPU de los microservicios backend (vista ampliada). Fuente: Elaboración propia. . . . . 129
Figura 26 Estado de los pods en el namespace default. Fuente: Elaboración propia. 131
Figura 27 Estado de los deployments. Fuente: Elaboración propia. . . . . 132
Figura 28 Servicios expuestos en el clúster. Fuente: Elaboración propia. . . . . 133
Figura 29 PersistentVolumeClaims (PVC) en el clúster. Fuente: Elaboración propia. 134
Figura 30 Obtención de PV funcionales. Fuente: Elaboración propia. . . . . 134
Figura 31 Datos persistentes en PostgreSQL - tabla usuarios (db_usuarios). Fuente: Elaboración propia. . . . . 136
Figura 32 Datos persistentes en PostgreSQL - tabla notas (db_notas). Fuente: Elaboración propia. . . . . 136
Figura 33 Datos persistentes en MongoDB - colección tareas (db_tareas). Fuente: Elaboración propia. . . . . 137
Figura 34 Datos persistentes en MongoDB - colección asistencias (db_asistencia). Fuente: Elaboración propia. . . . . 137
Figura 35 Reinicios de pods en Grafana. Elaboración propia. . . . . 139

Figura 36 Consumo de memoria inicial. Elaboración propia . . . . . 174
Figura 37 Implementación del frontend. Elabora propia . . . . . 174
Figura 38 Habilitación de puerto para VITE. Elaboración propia . . . . . 175
Figura 39 Pruebas de endpoints usando Postman. Elaboración propia. . . . . 175
Figura 40 Instalación de Runner para despliegue. Elaboración propia. . . . . 176
Figura 41 Configuración del runner. Elaboración propia. . . . . 176
Figura 42 Error de imagen en el frontend. Elaboración propia. . . . . 177
Figura 43 Proceso de memoria residente. Elaboración propia. . . . . 177
Figura 44 Proceso de memoria residente global. Elaboración propia. . . . . 178
Figura 45 Commits desde Febrero hasta Mayo. Elaboracion propia. . . . . 178

# ÍNDICE DE TABLAS


Tabla I Comparación entre Arquitectura Monolítica y Arquitectura de Microservicios [11] . . . . . 31

Tabla II Sprint 1 . . . . . 77
Tabla III Sprint 2 . . . . . 77
Tabla IV Sprint 3 . . . . . 78
Tabla V Sprint 4 . . . . . 78
Tabla VI Sprint 5 . . . . . 79
Tabla VII Sprint 6 . . . . . 79
Tabla VIII Sprint 7 . . . . . 80
Tabla IX Sprint 8 . . . . . 80
Tabla X Características de la arquitectura de microservicios . . . . . 83
Tabla XI Componentes principales del sistema . . . . . 84
Tabla XII Principios de diseño aplicados . . . . . 85

Tabla XIII Tiempos de ejecución del pipeline CI/CD en GitHub Actions . . . . . 117
Tabla XIV Métricas de ejecución del pipeline CI/CD . . . . . 119
Tabla XV Comparación del proceso manual vs automatizado . . . . . 120
Tabla XVI Análisis de la métrica de CPU. . . . . 126
Tabla XVII Consumo de memoria por microservicio . . . . . 127
Tabla XVIII Análisis del consumo de memoria . . . . . 128
Tabla XIX Resumen de métricas de rendimiento por microservicio . . . . . 130
Tabla XX Estado de los microservicios y despliegues en Kubernetes . . . . . 131
Tabla XXI Servicios desplegados en Kubernetes . . . . . 133
Tabla XXII Persistent Volume Claims (PVC) utilizados en Kubernetes . . . . . 134
Tabla XXIII Características de los Persistent Volumes y Persistent Volume Claims . . 135
Tabla XXIV Análisis de reinicios de Pods en Kubernetes . . . . . 138
Tabla XXV Resumen de confiabilidad del sistema . . . . . 140

# RESUMEN

La presente investigación tuvo como objetivo implementar un pipeline automatizado de Integración Continua y Despliegue Continuo (CI/CD) para arquitecturas basadas en microservicios, con el propósito de optimizar los procesos de construcción, pruebas y despliegue de aplicaciones modernas. El estudio surge debido a los problemas operativos presentes en arquitecturas distribuidas, tales como despliegues manuales, errores humanos, tiempos elevados de entrega, dificultades de escalabilidad y capacidad limitada de monitoreo. Para abordar esta problemática, se desarrolló una solución empleando buenas prácticas DevOps y tecnologías open source orientadas a la automatización y orquestación de servicios. La metodología empleada contempló el diseño e implementación de una arquitectura de microservicios contenerizada mediante Docker, utilizando GitHub Actions como motor de automatización CI/CD y Kubernetes como plataforma de orquestación de contenedores. Además, se integraron herramientas de observabilidad como Prometheus y Grafana para el monitoreo de métricas en tiempo real, permitiendo evaluar el comportamiento de los servicios, el consumo de recursos y la disponibilidad de la infraestructura. También se implementaron mecanismos de persistencia mediante bases de datos PostgreSQL y MongoDB, junto con volúmenes persistentes en Kubernetes.

Los resultados obtenidos evidenciaron una mejora significativa en la automatización de despliegues, reducción de tiempos operativos y mayor estabilidad del entorno de ejecución. Además, la solución permitió realizar compilaciones, pruebas unitarias y despliegues automáticos de manera continua, garantizando trazabilidad, repetibilidad y facilidad de mantenimiento. Así mismo, las herramientas de observabilidad facilitaron la supervisión centralizada de los microservicios y la detección temprana de anomalías.



**Palabras clave:** CI/CD, microservicios, DevOps, Kubernetes, Docker, GitHub Actions, Prometheus, Grafana.

# ABSTRACT

This research aimed to implement an automated Continuous Integration and Continuous Deployment (CI/CD) pipeline for microservice-based architectures, with the purpose of optimizing the build, test, and deployment processes of modern applications. The study arose from operational problems present in distributed architectures, such as manual deployments, human error, high delivery times, scalability difficulties, and limited monitoring capabilities. To address this issue, a solution was developed using DevOps best practices and open-source technologies focused on service automation and orchestration.

The methodology employed included the design and implementation of a containerized microservices architecture using Docker, utilizing GitHub Actions as the CI/CD automation engine and Kubernetes as the container orchestration platform. Additionally, observability tools such as Prometheus and Grafana were integrated for real-time metrics monitoring, allowing for the evaluation of service behavior, resource consumption, and infrastructure availability. Persistence mechanisms were also implemented using PostgreSQL and MongoDB databases, along with persistent volumes in Kubernetes.

The results obtained demonstrated a significant improvement in deployment automation, reduction in operational times, and greater stability of the execution environment. Furthermore, the solution enabled continuous automated builds, unit tests, and deployments, ensuring traceability, repeatability, and ease of maintenance. Likewise, the observability tools facilitated centralized supervision of microservices and the early detection of anomalies.


**Keywords:** CI/CD, microservices, DevOps, Kubernetes, Docker, GitHub Actions, Prometheus, Grafana.

# CAPÍTULO I

# INTRODUCCIÓN

## 1.1 Planteamiento del Problema

En el actual panorama del desarrollo de software, la arquitectura de microservicios se ha consolidado como la opción preferida por su capacidad de escalar, facilitar el mantenimiento y adaptarse con agilidad a cambios del negocio [1], [2]. Sin embargo, la fase de integración continua y entrega continua (CI/CD) de estos servicios distribuidos sigue demandando una atención especial: muchas empresas aún dependen de procesos manuales o semi controlados que generan cuellos de botella, elevan el riesgo de errores humanos y reducen la trazabilidad de cada versión desplegada [3]. Según el informe DORA, las organizaciones con prácticas maduras de CI/CD y DevOps despliegan hasta 46 veces más rápido y con tasas de fallo notablemente menores; no obstante, gran parte de las empresas no alcanzan estos niveles de madurez, lo que se traduce en retrasos de desarrollo, dificultades para escalar y mayores costos operativos [4]. Además, la ausencia de pruebas automatizadas y despliegues confiables introduce riesgo de caídas en producción, comprometiendo la disponibilidad y la escalabilidad de las aplicaciones [5].

En conclusión, aunque las aplicaciones tengan una arquitectura de microservicios y contengan el potencial de transformar la entrega de software; su verdadero valor queda limitado por la falta de soluciones globales de CI/CD. Es imprescindible diseñar y validar la automatización de un pipeline que no dependa de un solo lenguaje de desarrollo, fácilmente adaptable a cualquier stack tecnológico y replicable en diversos entornos empresariales, abordando tanto los retos técnicos como los culturales y organizacionales que impiden la adopción plena de CI/CD.

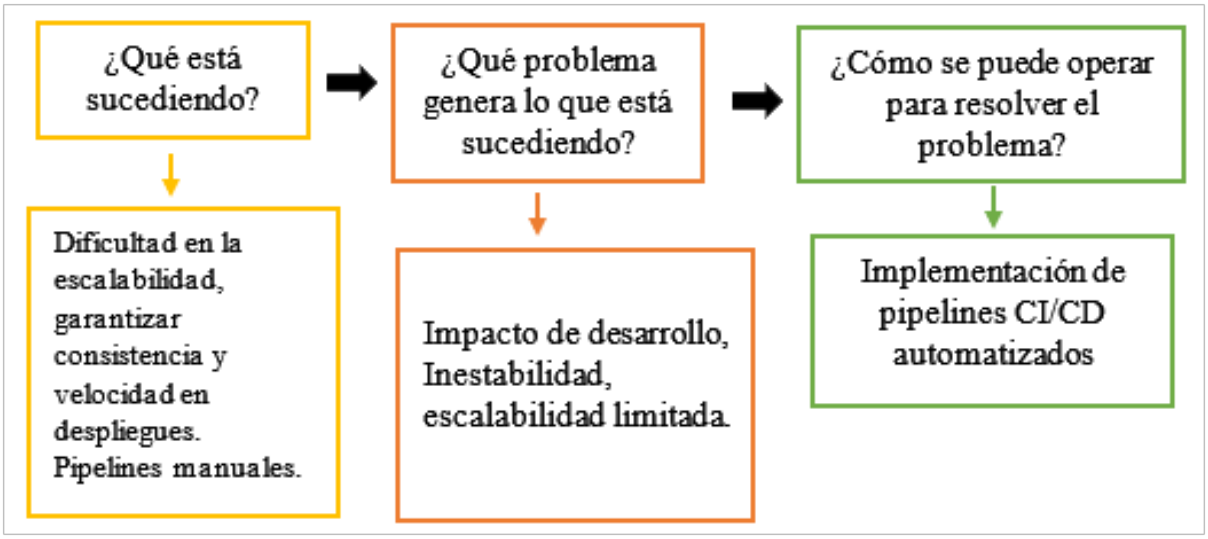



**Figura 1**.
Diegrama del problema. Elaboración propia

Este diagrama ilustra el núcleo de esta investigación sobre automatización de CI/CD para microservicios. Identifica un problema crítico: la dificultad en escalabilidad, consistencia y velocidad en despliegues manuales, que genera inestabilidad y limitaciones en entornos de microservicios. Como solución, propone implementar pipelines CI/CD automatizados, estableciendo una clara relación causa-efecto entre los desafíos operativos actuales y los beneficios de la automatización para arquitecturas de microservicios modernas.

## 1.2 Objetivos

### 1.2.1 Objetivo General

Implementar un pipeline automatizado de Integración Continua y Despliegue Continuo (CI/CD) para microservicios con el fin de optimizar los procesos de pruebas y despliegue de software.

### 1.2.2 Objetivos Específicos

- Realizar un análisis bibliográfico sobre la automatización de pipelines CI/CD y su impacto en el desarrollo de aplicaciones basadas en microservicios.

- Construir la arquitectura del pipeline CI/CD considerando buenas prácticas DevOps.

- Validar el funcionamiento del pipeline automatizado en cuanto a su eficiencia

## 1.3 Alcance y delimitación

El presente trabajo de investigación se centrará en el diseño, implementación y validación de un pipeline CI/CD automatizado, orientado a sistemas construidos bajo la arquitectura de microservicios. La solución se aplicará específicamente en el contexto de proyectos educativos orientados al desarrollo de plataformas de gestión académica, tales como sistemas de seguimiento de tareas, control de asistencia y gestión de notas de los estudiantes. La implementación estará enfocada en un conjunto básico de microservicios que representen funcionalidades comunes en aplicaciones académicas:

• Microservicio usuarios: gestión de registro, autenticación y roles: estudiante, docente, administrador.

• Microservicio tareas: crear, editar y entregar trabajos académicos.

• Microservicio notas: calificación, consultas y edición de evaluaciones.

• Microservicio asistencia: registro y verificación de asistencia a clases.

El enfoque estará limitado al uso de herramientas cloud y de código abierto ampliamente utilizadas en la industria, como GitLab CI, GitHub Actions o Circle CI como motor de CI/CD. Docker para contenedor, Kubernetes como orquestador, Express / FastAPI como Frameworks,

Python / Node.JS como lenguajes de programación y Git / GitHub como gestionador de código. La investigación incluirá el análisis del estado actual de la automatización CI/CD en contextos formativos, la selección de herramientas adecuadas y la construcción de un entorno funcional que permita automatizar el ciclo de vida de despliegue de los microservicios mencionados. La validación se llevará a cabo mediante un caso práctico en entorno controlado, simulando el flujo de trabajo completo desde el desarrollo hasta el despliegue. Se evaluarán métricas como la eficiencia del pipeline, confiabilidad del despliegue y facilidad de mantenimiento, con el objetivo de ofrecer una solución replicable y útil para entornos educativos centrados en la formación de desarrolladores.

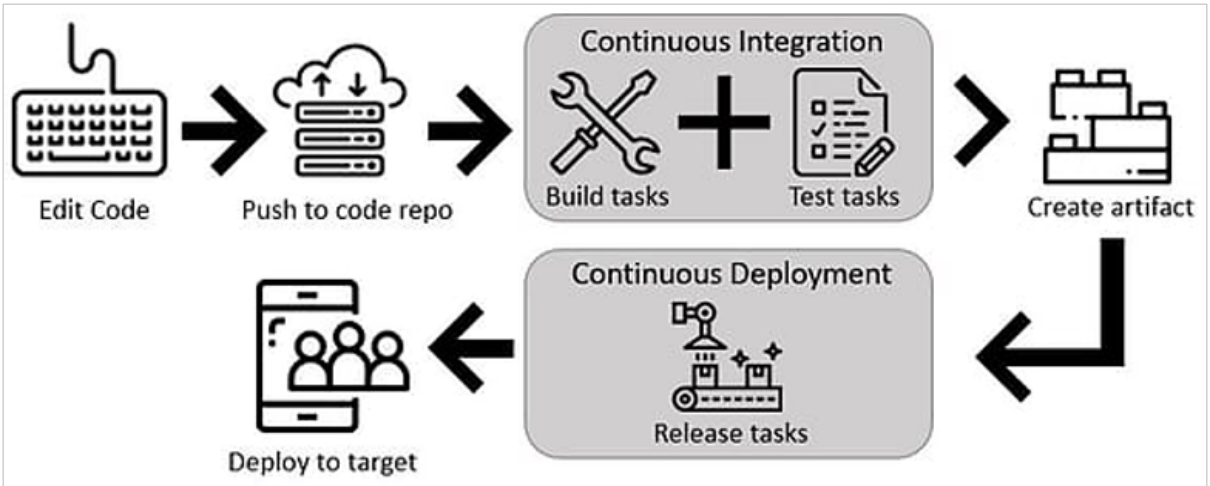



**Figura 2**.
Elaborado por Montoya y Ocampo de Pragma. Proceso de integracion continua y despliegue continuo [6].

La figura [6] representa de forma esquemática el flujo de trabajo que se busca automatizar mediante la implementación del pipeline CI/CD propuesto en esta investigación. El alcance técnico del sistema incluye todas las fases representadas en el ciclo, desde la edición del código fuente hasta el despliegue final del software en el entorno objetivo. Las etapas que abarca este trabajo son:

Edición y versionamiento del código: ”Los desarrolladores realizan cambios en el código fuente y lo suben a un repositorio remoto mediante sistemas de control de versiones como Git”[6].

Integración continua: .El pipeline automatizado debe ejecutar tareas de construcción y prue-

bas, de modo que garantice que el código este funcional y estable tras cada modificación”[6].

Creación de artefactos: Üna vez superadas las pruebas, se generan artefactos, por ejemplo: una imagen de contenedor listo para ser desplegado”[6].

Despliegue continuo (CD): .El artefacto se despliega automáticamente en el entorno de destino, simulando un sistema de entrega continua que reduce la intervención humana”[6].

Este flujo automatizado deberá ser replicado y validado en un entorno académico controlado, mediante el uso de microservicios de un sistema educativo. Con ello, buscamos la forma de demostrar la viabilidad sobre cómo debemos aplicar prácticas CI/CD en proyectos estudiantiles mediante herramientas accesibles y replicables.

## 1.4 Justificación

La automatización robusta de CI/CD en arquitecturas de microservicios no es solo una ventaja competitiva, sino una necesidad crítica en la industria del software moderno. En este trabajo se justifica en tres temas principales: la relevancia académica, la relevancia práctica y la relevancia social, basándose en los desafíos identificados, los objetivos planteados y el alcance definido.

### Relevancia académica

Aunque existen estudios sobre los microservicios [[2] [1]] y prácticas DevOps [3], [4], existe una brecha en la implementación de ambos campos. Uno de las investigaciones fue la de “Automation of Regression Test in Microservice Architecture”desarrollada por Kargar & Hanifizade [5], abordan pruebas automatizadas en microservicios, pero sin implementar un pipeline de CI/CD. Por otro lado, en el informe DORA acronimo de “DevOps Research and Assessment” traducido al espanol como “Investigación y evaluación de DevOps.evidencia que la correlación entre madurez por parte de DevOps y el desempeño organizacional, pero no profundiza en soluciones técnicas específicas para microservicios [4].

**Relevancia práctica**

Las organizaciones enfrentan desafíos tangibles al escalar microservicios: despliegues lentos, errores recurrentes y falta de estandarización [3]. Según el informe DORA, solo el 20 % de las empresas alcanzan un nivel élite en DevOps, mientras que el 45 % se clasifican como de baja madurez [4]. Esto se traduce en costos operativos elevados y riesgos en producción, como caídas de servicios o vulnerabilidades no detectadas [5]

**Relevancia Social**

La dependencia de sistemas software en sectores críticos (banca, telecomunicaciones, salud) exige despliegues rápidos, seguros y auditables. Un error en producción puede traducirse en pérdidas económicas, filtración de datos o incluso riesgos para la vida humana. Este trabajo contribuye a democratizar el acceso a CI/CD pues al utilizar herramientas de open-source el modelo propuesto es accesible para startups, PyMEs, organizaciones, etc., sin presupuestos elevados.

# CAPÍTULO II

# MARCO TEÓRICO

## 2.1 Bases teóricas

### 2.1.1 Arquitecturas de Software: Evolución hacia Microservicios

La evolución de las arquitecturas de software ha dado lugar al auge de los microservicios, un enfoque en el que las aplicaciones se construyen como un conjunto de servicios pequeños, independientes y autónomos, en lugar de como un gran monolito. Esta transición se ha impulsado por la necesidad de conseguir una mayor flexibilidad, escalabilidad y agilidad en el desarrollo y despliegue de aplicaciones modernas [7].

#### 2.1.1.1 Arquitectura Monolítica

La arquitectura monolítica es el enfoque tradicional para el desarrollo de aplicaciones, en el que todos los módulos, como la interfaz de usuario, lógica empresarial y acceso a datos, están integrados en un único bloque ejecutable que se ejecuta como un proceso único. Esta simplicidad facilita el desarrollo, la prueba, el despliegue y la depuración en las etapas iniciales de una aplicación [8]. A continuación, se muestran las caracteristicas, ventajas y limitaciones.

**Características**

La mayoría de las aplicaciones monolíticas poseen las siguientes características:

- **Un solo nivel.** Esto significa que múltiples componentes se combinan en una gran aplicación. En consecuencia, las aplicaciones tienden a tener grandes bases de código, que

puede ser engorroso actualizar y administrar con el tiempo [9].

- **Autónomo y unificado.** Son independientes de otras aplicaciones [9].
- **Base de código única.** Si un componente del programa necesita ser actualizado, otros elementos también pueden requerir reescritura, y toda la aplicación debe ser recompilada, probada e implementada [9].
- **Base de datos única.** Pueden usar un sistema de gestión de bases de datos relacionales como una única fuente de datos [9].

**Ventajas**

El uso de la arquitectura monolítica puede producir grandes beneficios, entre ellos:

- **Desarrollo de aplicaciones sin complicaciones:** Las aplicaciones construidas con una única base de código son más sencillas de construir con un desarrollo más rápido [10].
- **Implementación básica:** La arquitectura monolítica funciona con un archivo o directorio ejecutable, lo que hace que la implementación sea menos difícil. Una arquitectura monolítica también es más fácil de mantener en virtud del uso de menos componentes [10].
- **Depuración simple:** Pruebas y depuración las operaciones están menos involucradas con las arquitecturas monolíticas. Las operaciones de prueba de extremo a extremo se realizan desde un sistema de registro central [10].
- **Seguridad reforzada:** Debido a que una arquitectura monolítica es un sistema cerrado, sus actividades de procesamiento de datos están completamente contenidas y protegidas de ciberamenazas [10].

**Limitaciones**

El usar este tipo de arquitectura monolítica también nos presenta posibles problemas, tales como:

- **Resistente a las nuevas tecnologías:** Debido a que las aplicaciones monolíticas generalmente están estrechamente acopladas, puede ser difícil integrar nuevas tecnologías en ellas [10].
- **Escalabilidad reducida:** Incluso si la cantidad de escala necesaria es relativamente menor (como ajustar una sola función), es posible que tenga que desmantelar y reconstruir el sistema de manera efectiva para reflejar el nuevo cambio. Eso puede resultar lento y laborioso [10].

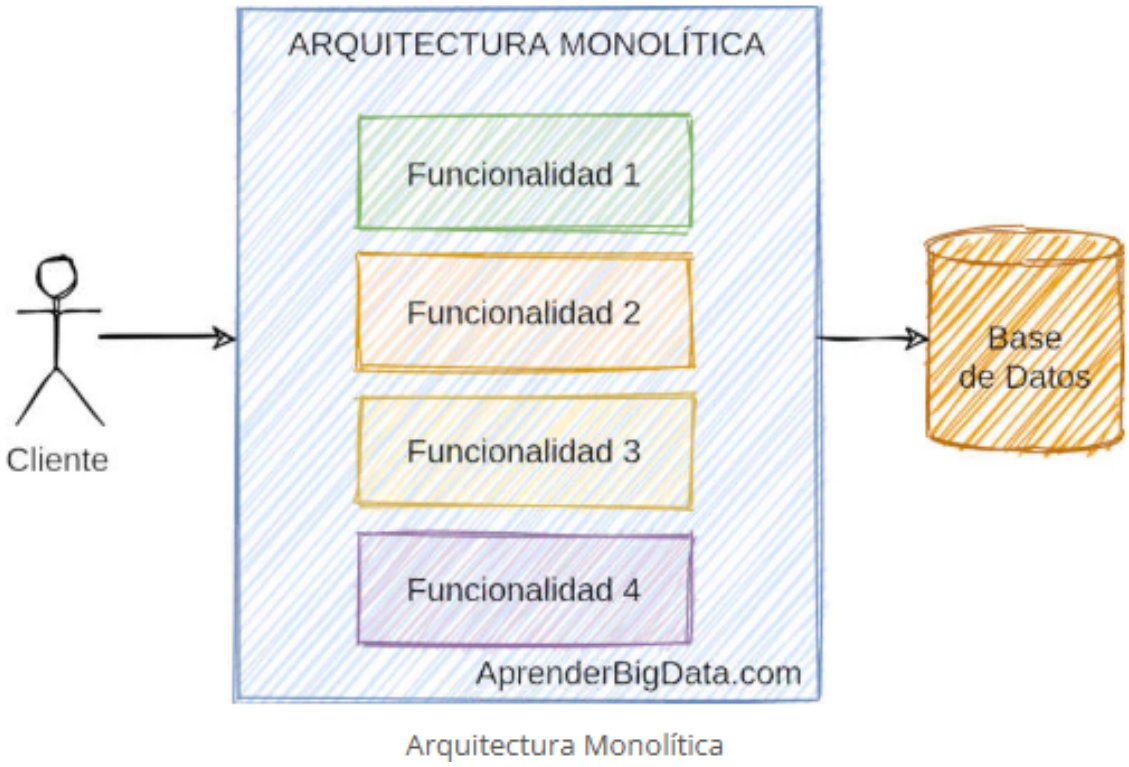



**Figura 3**.
Ejemplo de una arquitectura monolítica [11].

En la esta figura podemos observar un ejemplo de arquitectura monolítica, en la que todas las funcionalidades de la aplicación están integradas en un único sistema que interactúa directamente con el cliente y la base de datos. Pues este modelo destaca por su simplicidad y cohesión, ya que todos los componentes están acoplados en una sola unidad, como es lo usual en los sistemas tradicionales. La imagen refuerza el concepto central de esta arquitectura, un diseño unificado que centraliza la lógica empresarial y el acceso a los datos.

#### 2.1.1.2 Arquitectura de Microservicios

**Concepto**

Los microservicios son un enfoque arquitectónico para el desarrollo de sistemas distribuidos que se caracteriza por la implementación de servicios pequeños, autónomos y especializados, organizados en torno a dominios de negocio. Según Newman, estos servicios pueden ser modificados, implementados y liberados de manera independiente, lo que proporciona flexibilidad para adaptar el sistema a las necesidades cambiantes de los clientes [1]. Cada microservicio es responsable de una funcionalidad específica del sistema y se comunica con otros servicios mediante interfaces bien definidas, generalmente APIs REST o mensajería asincrónica [7].

**Principios clave**

Los principios clave de los microservicios establecen las bases para diseñar sistemas distribuidos y mantenibles. Estos principios se centran en la creación de servicios pequeños y autónomos, cada uno con una responsabilidad específica y la capacidad de operar de forma independiente. [12]. A continuacion, se muestran los siguientes:

- **Separación de responsabilidades:** Cada microservicio debe tener una única responsabilidad principal, centrada en una función o capacidad empresarial concreta. Así se facilita el desarrollo, el mantenimiento y las pruebas individuales de cada servicio [12].
- **Despliegue independiente:** "Cada microservicio se puede desplegar y actualizar de manera independiente sin afectar a otros servicios de la aplicación. Esto proporciona una mayor flexibilidad y agilidad para añadir nuevas funcionalidades"[12].
- **Autonomía:** "Los microservicios operan de forma autónoma y pueden elegir su propia tecnología y base de datos, lo que fomenta la innovación y la adopción de tecnologías específicas para cada servicio"[12].

- **Alta cohesion y bajo acoplamiento:** “Se busca que los microservicios tengan alta cohesión interna, es decir, que los elementos dentro de un servicio estén fuertemente relacionados. Al mismo tiempo, se busca un bajo acoplamiento entre los servicios para minimizar su dependencia mutua” [12].
- **Resiliencia:** “Un error afecta a toda la aplicación en arquitecturas monolíticas. Pero los microservicios son independientes. Un error no afectará a las demás partes de la aplicación.”[13].
- **Escalabilidad:** “Cada microservicio se puede escalar de manera independiente según la demanda, lo que optimiza el uso de recursos y garantiza la disponibilidad”[13].
- **Comunicación mediante APIs:** “La comunicación entre microservicios se realiza a través de interfaces bien definidas, generalmente APIs REST, lo que permite la interoperabilidad y la independencia tecnológica”[13].
- **Organización por capacidad empresarial:** “Los microservicios se organizan en base a capacidades empresariales específicas, lo que facilita la comprensión y gestión del sistema [14]”.
- **Equipos pequeños y autoorganizados:** Se recomienda que cada equipo de desarrollo sea responsable de uno o varios microservicios, lo que fomenta la autonomía y la responsabilidad [14].
- **Infraestructura automatizada:** “La automatización de la infraestructura es fundamental para facilitar el despliegue y la gestión de los microservicios [14]”.

**Caracteristicas**

- **Modularidad y separación:** “Cada servicio encapsula una funcionalidad de negocio específica con límites claros y bien definidos [15]”.
- **Tecnología heterogénea:** “Diferentes servicios pueden utilizar distintos lenguajes de programación, frameworks y bases de datos según sus necesidades específicas [15]”.

- **Propiedad del equipo:** “Cada microservicio es propiedad de un equipo pequeño que se responsabiliza de su desarrollo, despliegue y mantenimiento completo [1]”.
- **Comunicación ligera:** “Utilizan protocolos de comunicación ligeros, como HTTP/REST, gRPC o mensajería asíncrona, para interactuar entre servicios [1]”.
- **Despliegue independiente:** “Cada servicio puede desplegarse, actualizarse y escalarse de forma independiente sin afectar al resto [1]”.
- **Descentralización:** “No existe un punto central de control; cada servicio toma sus propias decisiones y gestiona su estado internamente [1]”.
- **Resiliencia:** “Implementan patrones de tolerancia a fallos, como timeouts, reintentos y circuit breakers, para gestionar dependencias no disponibles [1]”.
- **Observabilidad:** “Requieren herramientas sofisticadas de supervisión, registro distribuido y trazabilidad para mantener la visibilidad del sistema completo [1]”.

Los microservicios se caracterizan por su naturaleza modular y autónoma, donde cada servicio encapsula una funcionalidad específica del negocio y puede utilizar tecnologías heterogéneas según sus necesidades particulares. Cada servicio es propiedad de un equipo pequeño que controla completamente su ciclo de vida, desde el desarrollo hasta el despliegue independiente, lo que permite actualizaciones y escalado individuales sin afectar a otros componentes del sistema [16]. La comunicación entre servicios se realiza mediante protocolos ligeros como HTTP/REST o mensajería asíncrona, lo que permite mantener un bajo acoplamiento mientras se implementan patrones de resiliencia como cortafuegos y temporizadores de salida para manejar fallos de dependencias. Esta arquitectura descentralizada requiere herramientas sofisticadas de observabilidad, como el monitoreo distribuido, el registro centralizado y la trazabilidad de transacciones, para mantener la visibilidad y el control sobre el comportamiento del sistema completo, lo que facilita la detección temprana de problemas y la optimización del rendimiento [15].

#### 2.1.1.3 Comparativa: Monolito vs. Microservicios

**Tabla I**.
Comparación entre Arquitectura Monolítica y Arquitectura de Microservicios [11]

| Aspecto | Arquitectura Monolítica | Arquitectura de Microservicios |
|---|---|---|
| **Escalabilidad** | Limitada al sistema completo | Escalabilidad independiente por servicio |
| **Despliegue** | Centralizado | Descentralizado, automatizable |
| **Pruebas** | Más simples pero propensas a romperse por cambios | Necesita enfoques especializados como pruebas de contratos |
| **Tiempo de entrega** | Lento a medida que crece | Ágil con CI/CD por servicio |
| **Tecnologías** | Homogénea | Heterogénea |

En la presente tabla podemos observamos la comparación entre las arquitecturas monolítica y de microservicios revela profundas diferencias en su diseño y en su funcionamiento. En una arquitectura monolítica, todo el sistema se construye y despliega como una única unidad, lo que facilita el desarrollo inicial y simplifica el proceso de pruebas, ya que los componentes están integrados en el mismo entorno. “No obstante, a medida que las aplicaciones se desarrollan, esta integración se convierte en una desventaja, ya que cualquier cambio o error en el código puede afectar al sistema, lo que dificultaría el mantenimiento, pruebas regresivas y escalabilidad en secciones específicas. Por lo que la arquitectura basada en microservicios nos permite dividir la aplicación en múltiples servicios independientes, cada uno con su propia lógica y almacenamiento de datos, lo que habilita un desarrollo paralelo, escalabilidad granular y despliegues independientes. No obstante, esta flexibilidad introduce desafíos adicionales: la necesidad de coordinar múltiples despliegues, establecer pruebas de integración más sofisticadas (como las pruebas de contratos entre servicios), gestionar la comunicación en red y asegurar la coherencia de datos entre servicios distribuidos. En esencia, mientras el monolito favorece la simplicidad y velocidad en fases tempranas, los microservicios aportan agilidad, resiliencia y escalabilidad a gran escala, aunque requieren una mayor inversión en automatización, supervisión y gobernanza tecnológica”[17].

#### 2.1.1.4 Retos operativos en Microservicios

Si bien los microservicios ofrecen múltiples beneficios, también introducen nuevos desafíos operativos que deben gestionarse cuidadosamente para garantizar el éxito de la implementación:

- **Complejidad del sistema distribuido:** El diseño distribuido aumenta exponencialmente la complejidad operativa y requiere herramientas sofisticadas de observabilidad que incluyan registro distribuido, métricas en tiempo real y trazabilidad de transacciones (distributed tracing). La depuración de problemas se vuelve más compleja cuando involucra múltiples servicios interconectados y la identificación de la causa raíz de fallos puede requerir el análisis de correlación entre diferentes componentes del sistema [1].
- **Gestión de red y comunicación:** La comunicación entre servicios introduce latencia de red, puntos de fallo adicionales y la necesidad de configuraciones robustas para el balanceo de carga, el descubrimiento de servicios y la tolerancia a fallos. Para prevenir cascadas de fallos, es necesario implementar patrones como circuit breakers, timeouts, reintentos con backoff exponencial y bulkhead isolation. Además, la gestión de la seguridad en la red se complica con la necesidad de autenticación y autorización entre servicios [13].
- **Consistencia de datos y transacciones distribuidas:** Al eliminar las bases de datos centralizadas, se pierde la capacidad de utilizar transacciones ACID (es un acrónimo que significa Atomicidad, Consistencia, Aislamiento y Durabilidad) tradicionales. Por tanto, hay que implementar patrones como Eventual Consistency, en los que los datos pueden estar temporalmente inconsistentes, y el patrón Saga para manejar transacciones de larga duración que abarcan múltiples servicios. La sincronización de datos entre servicios requiere estrategias cuidadosas de replicación y versionado [1].
- **Despliegues y versionamiento:** “Aunque los servicios son teóricamente independientes, a menudo comparten contratos API o dependencias que deben mantenerse sincroni-

zadas. Por ello, el versionado de APIs se vuelve crítico para evitar romper la compatibilidad, para lo que se requieren estrategias como el versionado [1]".

- **Gestión del estado y persistencia:** "Aunque se prefiere mantener los microservicios sin estado (stateless) para facilitar el escalamiento, ciertos servicios requieren manejo de estado, lo que complica la gestión de sesiones, la caché distribuida y la recuperación ante fallos. La gestión de los datos de configuración y secretos también se vuelve más compleja en entornos distribuidos [13]".
- **Latencia y rendimiento:** La comunicación de red entre servicios añade latencia adicional en comparación con llamadas locales en aplicaciones monolíticas. Esto requiere optimizaciones como caché distribuido, agregación de llamadas y un diseño cuidadoso de las interfaces para minimizar el número de llamadas entre servicios (interfaces chatty) [1].
- **Seguridad distribuida:** La seguridad se vuelve más compleja con múltiples puntos de entrada y comunicación entre servicios. Por tanto, es necesario implementar una autenticación y una autorización distribuidas, una gestión centralizada de identidades, un cifrado de comunicaciones y una auditoría de accesos a través de toda la arquitectura [1].
- **Monitoreo y alertas:** El monitoreo debe abarcar múltiples dimensiones: la salud individual de los servicios, la comunicación entre ellos, las métricas empresariales y la infraestructura. Se requieren cuadros de mando unificados que proporcionen visibilidad de extremo a extremo y sistemas de alertas inteligentes que puedan correlacionar eventos a través de múltiples servicios [13].

Para afrontar estos retos es necesario una transformación organizativa que incluya la adopción de prácticas DevOps maduras, la automatización exhaustiva de pruebas unitarias, de integración, de contratos y end-to-end, herramientas de orquestación y contenedores como Docker, Kubernetes, y una cultura de observabilidad y mejora continua. La implementación exitosa

de los microservicios no solo es un desafío técnico, sino que también requiere cambios en los procesos, las herramientas y la mentalidad organizativa [11].

### 2.1.2 DevOps y Automatización de Ciclos de Vida

#### 2.1.2.1 Fundamentos de la Cultura DevOps

DevOps es más que una metodología técnica, es una cultura organizacional basada en la colaboración, la automatización de procesos y la mejora continua. Su objetivo es eliminar las barreras tradicionales entre desarrollo (Dev) y operaciones (Ops), fomentando un entorno donde ambos equipos trabajen juntos desde la planificación hasta la entrega de software. Dentro de esto contemplamos tres aspectos como la colaboracion, automatizacion y mejora continua [18].

**Colaboración**

Segun el sitio web Atlassian nos dice que “La premisa clave de DevOps es la colaboración. Los equipos de desarrollo y operaciones se fusionan en un equipo funcional que se comunica, comparte feedback y colabora durante todo el ciclo de desarrollo e implementación. A menudo, esto conlleva que los equipos de desarrollo y operaciones se fusionen en un solo equipo que se encarga de todo el ciclo de vida de la aplicación”[18].

**Automatización**

Segun el sitio web Atlassian nos dice que “Una práctica fundamental de DevOps es automatizar todo el ciclo de vida de desarrollo de software que sea posible. De este modo, los desarrolladores tienen más tiempo para programar y desarrollar nuevas funciones. La automatización es un elemento clave para conseguir una canalización de CI/CD que ayude a reducir los errores humanos y aumentar la productividad del equipo. Con la automatización de los procesos, los equipos consiguen una mejora continua con tiempos de iteración breves que les permiten responden rápidamente a los comentarios de los clientes”[18].

**Mejora continua**

Segun el sitio web Atlassian nos dice que “La mejora continua se estableció como un elemento básico de las prácticas de metodología ágil, así como de la producción lean y el kata de mejora. Es una práctica que consiste en centrarse en la experimentación, minimizar lo superfluo y optimizar la velocidad, los costes y la facilidad de entrega. La mejora continua también está vinculada a la entrega continua, lo que permite a los equipos de DevOps enviar actualizaciones de forma continuada para mejorar la eficiencia de los sistemas de software. Con la canalización constante de nuevas publicaciones, los equipos envían sistemáticamente cambios de código que eliminan lo superfluo, mejoran la eficiencia del desarrollo y aportan más valor a los clientes”[18].

#### 2.1.2.2 Integración Continua (Continuous Integration): Prácticas, beneficios y flujo de trabajo

La Integración Continua (CI conocida por sus siglas Continuous Integration) es una práctica clave de DevOps en la que los desarrolladores integran su código con frecuencia en un repositorio compartido, preferiblemente varias veces al día. Cada integración activa automáticamente los procesos de construcción y las pruebas automatizadas con el fin de detectar errores lo antes posible [19].

Además, la CI se basa en el principio de retroalimentación rápida y la filosofía de ”fallar rápido”(fail fast), lo que permite identificar y corregir problemas en las etapas iniciales del desarrollo. Esta práctica fue popularizada por Martin Fowler y constituye uno de los pilares fundamentales de las metodologías ágiles de desarrollo [20]. Dentro de la CI también tenemos lo que son prácticas, beneficios y flujo de trabajo.

**Prácticas**

Las prácticas de CI incluyen:

- Integración frecuente del código en ramas principales (desarrollo en base a ramas principales) [19].
- Ejecución automática de pruebas unitarias, de integración y funcionales[19].
- “Verificación del estilo de código mediante herramientas de linting y formateo [19]”.
- “Análisis estático del código para detectar vulnerabilidades y errores de diseño [19]”.
- “Construcción automática de artefactos deployables [19]”.
- “Notificaciones inmediatas sobre el estado de las compilaciones [19]”.
- “Mantenimiento de un repositorio único como fuente de verdad [20]”.

**Beneficios**

- “Fomenta una mayor responsabilidad individual en la calidad del código”[19].
- “Reduce el tiempo dedicado a resolver conflictos de integración”[20].
- “Mejora la visibilidad del estado del proyecto para todos los miembros del equipo”[20].
- “Facilita la colaboración entre desarrolladores al mantener siempre estable el código base”[20].

En el marco de DevOps, “la CI es el pilar que permite la entrega rápida: al estandarizar las pruebas y las construcciones, los equipos pueden iterar con confianza, cumpliendo los principios de colaboración y mejora continua [19]”.

**Flujo de trabajo**

Segun el sitio web LightNode el flujo de CI sigue las siguientes etapas:

1. “Compromiso de Código: Los desarrolladores envían los cambios de código a un repositorio compartido, por ejemplo: Git”[21].

2. “Construcción: El código se construye automáticamente en un artefacto ejecutable (por ejemplo, binario, imagen de Docker)”[21].
3. “Pruebas Automatizadas: Se ejecutan pruebas automatizadas (unitarias, de integración y otros tipos) para verificar el código”[21].
4. “Despliegue en Staging: Si las pruebas pasan, el código se despliega en un entorno de staging para más pruebas”[21].
5. “Aprobación Manual o Despliegue Automatizado en Producción: Si las pruebas de staging son exitosas, el código se despliega en producción. Esto puede requerir aprobación manual (en Entrega Continua) o puede ser completamente automatizado (en Despliegue Continuo)”[21].

#### 2.1.2.3 Entrega Continua (CD) y Despliegue Continuo (CD)

**Entrega Continua**

Entrega continua (Continuous Delivery): consiste en mantener el software siempre desplegable. Tras el proceso de integración continua (CI), el código pasa automáticamente a un entorno de prueba, donde puede ser revisado manualmente antes de ser lanzado a producción. Esta metodología garantiza que cada cambio en el código pueda desplegarse de forma segura y fiable, pero mantiene un control humano sobre cuándo realizar el despliegue final [22].

**Despliegue Continuo**

El despliegue continuo (continuous deployment) va un paso más allá y automatiza por completo el proceso de despliegue del código a producción, sin intervención humana. En este enfoque, cada cambio que supera todas las pruebas automatizadas se despliega automáticamente al entorno de producción, eliminando cualquier barrera manual en el proceso [22].

La principal diferencia radica en el punto de control: mientras que la entrega continua requiere una aprobación manual para el despliegue a producción, el despliegue continuo auto-

matiza por completo este paso, confiando por completo en la batería de pruebas automatizadas para garantizar la calidad del software [22].

**Prácticas comunes**

- "Uso de pipelines automatizados que validan el código en distintas etapas progresivas [20]".
- "Implementación de gates de calidad que deben superarse en cada fase [20]".
- "Paralelización de pruebas para optimizar tiempos de ejecución [20]".
- "Rollback automático en caso de fallos detectados [20]".

**Beneficios**

- Reducción del tiempo de entrega de valor al usuario, pasando de ciclos de semanas o meses a horas o días [20].
- Mayor confianza en cada despliegue gracias a las pruebas automatizadas exhaustivas [22].
- Posibilidad de hacer múltiples despliegues diarios sin interrupciones significativas del servicio [22].
- Detección temprana de problemas, reduciendo el costo de corrección de defectos [22].

#### 2.1.2.4 Infraestructura como Código (IaC)

La Infraestructura como Código (IaC) representa un paradigma fundamental en la gestión moderna de la infraestructura de TI, en la que los recursos computacionales se definen, aprovisionan y gestionan mediante archivos de configuración legibles por humanos y procesables

por máquinas. Esta metodología transforma la administración tradicional de la infraestructura de un proceso manual y propenso a errores hacia un enfoque automatizado, versionado y reproducible [23].

**Concepto**

Infraestructura como código (conocida por sus siglas Infrastructure as Code, abreviado IaC) trata sobre la infraestructura como el código fuente de las aplicaciones, aplicando principios de ingeniería de software como control de versiones, pruebas automatizadas, revisiones de código y despliegue continuo. Este enfoque establece la infraestructura como un artefacto inmutable que puede recrearse de manera consistente en cualquier momento y entorno [23].

**Herramientas**

- Según la web oficial de Terraform "es una herramienta declarativa de aprovisionamiento y orquestación de infraestructura que permite a los ingenieros automatizar el aprovisionamiento de todos los aspectos de su infraestructura empresarial basada en la nube y en las instalaciones"[24].
- Según la web oficial de Chef "es una de las herramientas de gestión de la configuración más populares que las organizaciones utilizan en sus procesos continuos de integración y entrega. Chef es una herramienta agnóstica de la nube y trabaja con muchos proveedores de servicios de nube como AWS, Microsoft Azure, Google Cloud Platform, OpenStack y más"[25].
- Según la web oficial de Puppet "es otra herramienta popular de administración de configuración que ayuda a los ingenieros a entregar software continuamente. Usando Puppet, podemos definir el estado final deseado de la infraestructura. Luego Puppet automáticamente refuerza el estado deseado y arregla cualquier cambio incorrecto. Se integra con los principales proveedores de cloud computing como AWS, Azure, Google Cloud y VMware, lo que le permite automatizar en varias nubes"[26].
- Según la web oficial de Ansible "es una herramienta de automatización de infraestructu-

ras creada por Red Hat. Ansible modela su infraestructura describiendo cómo se relacionan sus componentes y el sistema entre sí, en lugar de gestionar los sistemas de forma independiente"[27].

**Ventajas**

- Reproducibilidad: Se pueden crear entornos idénticos para desarrollo, pruebas y producción [28].
- Reducción de errores humanos [28].
- Escalabilidad y consistencia en despliegues complejos [28].
- Ahorro de tiempo en la provisión de infraestructura [28].

"Las herramientas de Infraestructura como Código: Terraform, Chef, Puppet y Ansible, permiten definir y gestionar la infraestructura mediante código. Estas soluciones ofrecen ventajas clave: la reproducibilidad de entornos, la reducción de errores manuales, la escalabilidad en despliegues complejos y la eficiencia en la provisión de recursos, lo que optimiza los ciclos de DevOps" [23].

#### 2.1.2.5 Automatización de Pipelines CI/CD

La automatización de pipelines CI/CD es un componente central de DevOps. Un pipeline representa una serie de etapas secuenciales por las que pasa el código desde que se escribe hasta que se despliega en producción. Cada fase está automatizada y controlada por herramientas como Jenkins, GitLab CI, GitHub Actions, CircleCI, entre otras [29].

**Componentes clave:**

- Repositorio de código fuente: Git.

- Herramientas de integración: Jenkins, GitLab CI.
- Sistemas de pruebas automáticas: JUnit, Selenium.
- Sistemas de despliegue: Docker, Kubernetes.

**Fases típicas:**

- Build (construir): “Compilación del código fuente, generación de artefactos como archivos .jar, contenedores Docker [12]”.
- Test (pruebas): “Ejecución de pruebas unitarias, de integración y de aceptación [30]”.
- Deploy: “Despliegue del artefacto a entornos de staging y/o producción [30]”.

La automatización de pipelines no solo reduce el tiempo de entrega, sino que también mejora la calidad, repetibilidad y trazabilidad de cada entrega. Es particularmente esencial en arquitecturas de microservicios, donde múltiples servicios deben integrarse, testearse y desplegarse de manera orquestada y frecuente [29].

### 2.1.3 Patrones y Herramientas para CI/CD en Microservicios

La implementación de CI/CD en entornos basados en microservicios requiere un conjunto de patrones, herramientas y tecnologías que faciliten la automatización, escalabilidad y mantenimiento del ciclo de vida del software. Entre las prácticas más relevantes se encuentran la contenedorización, la orquestación de contenedores, la adopción de GitOps, los motores de CI/CD y la gestión del código con flujos de trabajo colaborativos. Estos elementos permiten integrar, entregar y desplegar microservicios de forma coherente, repetible y fiable [31].

#### 2.1.3.1 Contenedorización: Docker

La contenedorización es una técnica clave en el proceso de CI/CD para microservicios, ya que permite encapsular aplicaciones junto con sus dependencias en unidades portables y reproducibles llamadas contenedores. Docker es la herramienta más popular en este ámbito [31].

**Docker**

Según el sitio web de Docker "es una plataforma de código abierto que permite a los desarrolladores crear, implementar, ejecutar, actualizar y gestionar aplicaciones en contenedores. Los contenedores son unidades estandarizadas que contienen el código de la aplicación y todas sus dependencias, lo que facilita el despliegue consistente en diferentes entornos"[32].

Algunas de las características de Docker son las siguientes:

- Las imágenes Docker son plantillas inmutables que contienen el sistema operativo base, el entorno de ejecución y la aplicación. Se construyen a partir de un archivo llamado Dockerfile [32].
- Los contenedores son instancias en ejecución de imágenes. Se comportan como máquinas virtuales ligeras, aisladas del sistema anfitrión [32].
- Registros Docker: son repositorios donde se almacenan y distribuyen imágenes. Docker Hub es el registro público más conocido, aunque también existen opciones privadas como Amazon ECR o GitHub Container Registry [32].

#### 2.1.3.2 Orquestación de Contenedores: Kubernetes

**Concepto**

Según el sitio web de Kubernetes "es una plataforma portable y extensible de código abier-

to para administrar cargas de trabajo y servicios. Kubernetes facilita la automatización y la configuración declarativa. Tiene un ecosistema grande y en rápido crecimiento. El soporte, las herramientas y los servicios para Kubernetes están ampliamente disponibles"[33].

**Pods**

En Kubernetes un Pod, es un grupo de uno o más contenedores, con recursos de almacenamiento y red compartidos, y una especificación sobre cómo ejecutarlos. El contenido de un Pod siempre está coubicado y coprogramado, y se ejecuta en un contexto compartido. Un Pod modela un host lógico específico de la aplicación: contiene uno o más contenedores de aplicaciones que están estrechamente acoplados. En contextos distintos de la nube, las aplicaciones ejecutadas en la misma máquina física o virtual son análogas a las aplicaciones en la nube ejecutadas en el mismo host lógico [34].

**Deployments**

Según el sitio web de Kubernetes, un Deployment traducido al espanol como Implementacion, "es un objeto de alto nivel que gestiona el ciclo de vida de las aplicaciones, asegurando que se ejecute el número deseado de réplicas de pods y manejando actualizaciones y reversiones de manera declarativa. Una implementación administra un conjunto de pods para ejecutar una carga de trabajo de aplicación, generalmente una que no mantiene el estado"[35].

**Services**

En Kubernetes, un servicio es un método para exponer una aplicación de red que se ejecuta como una o más Cápsulasen tu clúster. Un objetivo clave de los Servicios en Kubernetes es que no necesite modificar sus servicios existentes aplicación para utilizar un mecanismo de descubrimiento de servicios desconocido. Puede ejecutar código en Pods, ya sea que se trate de un código diseñado para un mundo nativo de la nube o una aplicación más antigua que has contenedorizado. Utiliza un Servicio para que ese conjunto de Pods esté disponible en la red para que los clientes puedan interactuar con ella [36].

#### 2.1.3.3 GitOps: Principios y herramientas

GitOps es un enfoque que utiliza Git como única fuente de verdad para las configuraciones de infraestructura y aplicaciones. Automatiza el despliegue mediante herramientas que sincronizan el estado declarado en Git con el estado real del clúster [37].

**Principios**

1. Declarativo Un sistema administrado por GitOps debe tener su estado deseado expresado declarativamente [38].
2. Versionado e Inmutable El estado deseado se almacena de forma que garantiza la inmutabilidad, el control de versiones y conserva un historial de versiones completo [38].
3. Extraccion Automática Los agentes de software extraen automáticamente las declaraciones del estado deseado del código fuente [38].
4. Conciliado Continuo Los agentes de software observan constantemente el estado real del sistema e intentan aplicar el estado deseado [38].

**Herramientas**

Existe herramientas que se pueden complementar con otras para diseñar un marco de GitOps, como los repositorios de Git, Kubernetes, las herramientas de integración y distribución continuas (CI/CD) y las de gestión de la configuración [37]. A continuación se muestran algunas herramientas.

**Herramientas de CI**

**Tekton**

Según el sitio web de Tekton “es un marco desarrollado para Kubernetes que permite establecer sistemas de CI/CD. Ofrece a los ingenieros y los desarrolladores de plataformas la

flexibilidad, las prácticas recomendadas y la estandarización que necesitan para realizar los diseños, las pruebas y las implementaciones en proveedores en la nube y sistemas locales"[39].

**Jenkins**

Según el sitio web de Jenkins "es un servidor de automatización open source basado en Java. Facilita a los desarrolladores la integración de sus cambios al realizar pruebas permanentes, diseñar e implementar los proyectos de forma sencilla y con una configuración mínima. Jenkins X es una extensión del ecosistema de Jenkins que permite automatizar los procesos de CI/CD en la nube"[40].

**Herramientas de CD**

Según el sitio web de RedHat "la implementación o distribución continua (CD) es un proceso de dos partes que implica la integración, la prueba y la distribución de los cambios en el código. Después de la automatización de las compilaciones y las pruebas de unidad e integración de la CI, la distribución continua automatiza el traslado del código validado hacia un repositorio"[37]. A continuacion se muestran algunas herramientas.

**Argo CD**

Según el sitio web de ArgoCD "es una herramienta declarativa de distribución continua para Kubernetes. Permite que los equipos de desarrollo implementen y gestionen las aplicaciones sin necesidad de poseer amplios conocimientos sobre Kubernetes ni acceso completo a este sistema. Con la supervisión de todo el proceso de ejecución, Argo CD garantiza que sus aplicaciones nunca dejen de estar sincronizadas con los estados deseados"[41].

**Flux**

Según el sitio web de Flux "se trata de una herramienta de GitOps que automatiza la implementación en clústeres de Kubernetes gracias a que mantiene los estados del clúster sincronizados con los repositorios de Git. Se basa en un conjunto de extensiones de la API de Kubernetes y admite la implementación de aplicaciones y la gestión de infraestructuras"[42].

#### 2.1.3.4 Motores de CI/CD: GitHub Actions, GitLab CI

Aunque no existe una definición estandarizada del término 'motor de CI/CD', en la práctica se refiere a un conjunto de herramientas y procesos automatizados que ayudan a los equipos de desarrollo a integrar cambios de código con mayor frecuencia y fiabilidad, y a entregar software de forma continua a los usuarios. A continuación, se muestran algunos motores ejemplares usados en la industria.

**GitHub Actions**

"GitHub Actions es una herramienta que permite reducir la cadena de acciones necesaria para la ejecución de código, mediante la creación de un de flujo de trabajo encargado del Pipeline. Siendo configurable para que GitHub reaccione a ciertos eventos de forma automática según nuestras preferencias"[43].

**GitLab CI**

Según el sitio web de GitLab CI/CD "es una herramienta integrada en GitLab que automatiza el proceso de integración y entrega continua (CI/CD) de software. Permite definir pipelines que ejecutan scripts para construir, probar y desplegar código automáticamente, agilizando el ciclo de desarrollo"[44].

Tanto GitHub Actions como GitLab son motores de integración continua y despliegue continuo (CI/CD) muy utilizados, pero presentan diferencias según el entorno de desarrollo.

"GitHub Actions está integrado de forma nativa en GitHub y permite definir flujos de trabajo "workflows.[en] archivos YAML dentro del repositorio. Su punto fuerte es su enfoque modular mediante acciones reutilizables, que pueden descargarse desde un mercado público o desarrollarse de forma personalizada. Esto permite a los desarrolladores construir pipelines de manera ágil y mantener una alta cohesión con el repositorio principal. GitHub Actions es especialmente adecuado para equipos que ya trabajan en GitHub y desean simplicidad y potencia en un mismo entorno"[43].

“GitLab CI/CD se destaca por su capacidad de personalización y por su detallada interfaz de monitorización, lo cual es valioso para las organizaciones que necesitan visibilidad y control sobre cada etapa del proceso”[44].

En otras palabras, GitHub Actions es la opción más sencilla y directa; además, es ideal para pequeños proyectos o startups, mientras que GitLab CI/CD es más completo y detallado e incluso apropiado para proyectos más complejos y con estructuras jerárquicas de despliegue.

#### 2.1.3.5 Gestión de Código y Colaboración: Git y GitHub

Según el sitio web de Atlassian, “La gestión de código fuente (SCM) se utiliza para hacer un seguimiento de las modificaciones realizadas en un repositorio de código fuente. La SCM hace un seguimiento del historial de cambios activo de un código base y ayuda a resolver conflictos al fusionar modificaciones realizadas por varios colaboradores ”[45].

**Flujos de trabajo**

Según el sitio web de GitHub “el flujo de GitHub conocido tambien como GitFlow es un flujo de trabajo ligero basado en ramas. El flujo de GitHub es útil para todos, no solo para los desarrolladores. Por ejemplo, en GitHub, se usa el flujo de GitHub para la directiva del sitio, la documentación y la hoja de ruta”[46].

A continuación, se menciona el flujo de trabajo segun GitHub:

1. Crear una rama: “se crea una rama con un nombre corto y descriptivo como «increase-test-timeout» o «add-code-of-conduct» que permite a los colaboradores identificar el trabajo en curso de un vistazo”[46].
2. Realizar cambios: “en la nueva rama se realizan los cambios deseados de forma segura, sin afectar a la rama principal. Cada confirmación debe incluir un mensaje descriptivo que detalle los cambios realizados y que estos estén completamente aislados”[46].

3. Crear una solicitud de cambios (pull request): “Se crea una solicitud de cambios para obtener la retroalimentación de los colaboradores, incluyendo un resumen de los cambios y la explicación del problema que solucionan”[46].

4. Abordar comentarios: “Los revisores pueden hacer preguntas, comentarios y sugerencias que deben atenderse para permitir mejoras continuas en el código”[46].

5. Fusionar la solicitud: “Una vez aprobada, la solicitud se fusiona automáticamente con la rama principal, conservando el historial de comentarios y confirmaciones”[46].

6. Motor CDI: “Significado, funcionamiento y ejemplos de motores populares”[46].

7. Eliminar la rama: “Finalmente, se elimina la rama para indicar que el trabajo está completo sin perder la información del historial”[46].

**Trunk-Based Development**

Según el sitio web de Trunk-Based Development “es un modelo de ramificación de control de código fuente donde los desarrolladores colaboran en código en una sola rama llamada ’trunk’, resistiendo cualquier presión para crear otras ramas de desarrollo de larga duración mediante el empleo de técnicas documentadas”[47]

**Comparativa**

Gitflow propone una estrategia basada en múltiples ramas que representan distintos estados del desarrollo: main, develop, feature, release y hotfix. Este modelo permite un control más estricto del ciclo de vida del software, ya que facilita la gestión de versiones planificadas, lanzamientos formales y mantenimiento de correcciones críticas. Es especialmente útil para organizaciones con calendarios de entrega establecidos y con equipos que trabajan en paralelo en varias funcionalidades [46].

“En contraste, Trunk-Based Development promueve una filosofía de simplicidad y rapidez, en la que todos los desarrolladores trabajan directamente sobre una única rama principal

(trunk o main). Los cambios se integran de forma continua y frecuente, lo que reduce los conflictos de integración y fomenta una cultura de despliegue continuo. Este enfoque está alineado con las prácticas modernas de DevOps y CI/CD, donde la prioridad es entregar rápidamente y de manera confiable software en pequeños incrementos"[47].

### 2.1.4 Calidad y Seguridad en Pipelines Automatizados

En un entorno de desarrollo moderno basado en microservicios y despliegue continuo, la calidad y la seguridad del software son pilares fundamentales. Los pipelines automatizados permiten integrar estas prácticas directamente en el flujo de trabajo DevOps, asegurando que cada cambio de código sea verificado no solo funcionalmente, sino también desde una perspectiva de ciberseguridad [20].

#### 2.1.4.1 Pruebas Automatizadas

En el sitio web de Testing IT nos mencionan que "son un conjunto de procesos en los que se utilizan herramientas especializadas para ejecutar pruebas de software sin intervención manualc "su principal objetivo es mejorar la eficiencia y precisión en la detección de errores, asegurando que los productos digitales cumplan con los estándares de calidad antes de su lanzamiento"[48].

**Tipos y Pirámide de Pruebas** La Pirámide de Pruebas es un modelo que representa la proporción ideal de diferentes tipos de pruebas en un sistema de CI/CD:

- "Pruebas Unitarias: Verifican el funcionamiento de unidades individuales de código (como funciones o métodos). Son rápidas, aisladas y representan la base de la pirámide"[49].
- "Pruebas de Integración: Validan la interacción entre distintos módulos o servicios. Aseguran que los componentes funcionen correctamente al interactuar entre sí"[50].

- “Pruebas de Contratos: Aseguran que los servicios en una arquitectura de microservicios mantengan acuerdos definidos (contratos) entre productor y consumidor, como en el caso de APIs REST”[51].
- “Pruebas End-to-End (E2E): Las pruebas E2E evalúan el flujo de trabajo del software simulando las interacciones del usuario con las API, las bases de datos y la interfaz, incluida la finalización del formulario, los clics en los botones y la navegación”[50].

**Frameworks y su integración en CI**

Existen múltiples herramientas y frameworks para implementar estas pruebas en los pipelines CI/CD:

- “Jest (JavaScript) y Pytest (Python) son populares para pruebas unitarias y de integración. Se integran fácilmente con plataformas como Jenkins, GitHub Actions o GitLab CI”[52].
- “Selenium y Cypress son los mas usados para pruebas E2E. Selenium permite la automatización de navegadores con diversos lenguajes, mientras que Cypress ofrece una solución más moderna y rápida para el testing en frontend”[48].

#### 2.1.4.2 Seguridad en CI/CD (DevSecOps)

La integración continua y el despliegue continuo (CI/CD) ha estado evolucionandp fundamentalmente la manera en que las organizaciones desarrollan, prueban y despliegan software. Como señala Thatikonda, “CI/CD se ha posicionado como una práctica influyente dentro de los marcos Agile y DevOps, ayudando a las organizaciones a entregar valor de manera rápida y consistente a sus consumidores”[53].

**Principios Básicos de Seguridad de la Información**

Siguiendo los principios establecidos por la norma ISO/IEC 27001, la seguridad de la información en pipelines CI/CD debe garantizar:

- Confidencialidad: “Protección de la información sensible durante todo el proceso de desarrollo y despliegue, incluyendo código fuente, credenciales, configuraciones y datos de prueba”[54].
- Integridad: “Asegurar que el código y los artefactos no sean modificados de manera involuntaria durante su despliegue por el pipeline”[54].
- Disponibilidad: “Mantener los sistemas y procesos accesibles para usuarios autorizados, evitando interrupciones en el flujo de desarrollo y despliegue”[54].

**Análisis Estático de Seguridad (SAST)**

En el sitio de Testlio menciona que “Las pruebas estáticas son un método utilizado para identificar errores de software sin ejecutar realmente el código. Piense en ello como revisar un documento importante antes de publicarlo, asegurándose de que cada detalle sea preciso y libre de errores”[55].

**Análisis de Dependencias (SCA)**

El análisis de composición de software o Software Composition Analysis (SCA) evalúa las bibliotecas y paquetes externos usados por la aplicación. Herramientas como OWASP Dependency-Check, Snyk o WhiteSource identifican vulnerabilidades conocidas en versiones específicas de dependencias. Esto es vital ya que muchas brechas de seguridad provienen de componentes de terceros [52].

**Análisis de Imágenes de Contenedores**

Segun Kim et al. [56] en su investigacion encontramos que “el análisis de imágenes de contenedores es una medida de seguridad fundamental en los sistemas de contenedorización modernos”. Ademas los autores nos explican que, a diferencia de los métodos tradicionales, que

dependen exclusivamente de gestores de paquetes o del análisis de Dockerfiles, los enfoques más robustos inspeccionan directamente el sistema de archivos del contenedor.

**Gestión de Secretos (Secrets Management)**

En el sitio KeeperSecurity menciona que “La gestión de secretos es el proceso de organizar, gestionar y proteger los secretos de la infraestructura de TI. Permite a las organizaciones almacenar, transmitir y auditar secretos de forma segura”[57].

**Buenas Prácticas de Seguridad en Pipelines**

A continuación, veremos algunos aspectos claves que se contemplan en la buenas prácticas, segun Thatikonda [53].

1. **Automatización de Pruebas de Seguridad**

   La automatización es central tanto para CI/CD como para la seguridad. Las pruebas de seguridad automatizadas son fundamentales en los pipelines CI/CD, ya que ”validan continuamente el código y la infraestructura contra vulnerabilidades conocidas”[53].

2. **Gestión Segura de Credenciales y Secrets**

   La gestión inadecuada de credenciales representa uno de los mayores riesgos en pipelines CI/CD. Prácticas recomendadas:

   - “Utilizar sistemas de gestión de secretos especializados: HashiCorp Vault, AWS Secrets Manager”[53].
   - “Implementar cambio automático de credenciales”[53].
   - “No almacenar secrets en código fuente o archivos de configuración”[53].
   - “Aplicar principio de menor privilegio en accesos”[53].

3. **Monitoreo y Feedback Continuo**

   “El monitoreo en tiempo real es imperativo en CI/CD para identificar anomalías, regresiones de rendimiento o errores de manera inmediata”[53].

Estrategias de monitoreo:

- “Implementar un inicio de sesión centralizado y análisis de logs”[53].
- “Establecer alertas basadas en eventos de seguridad”[53].
- “Monitorear métricas de seguridad en tiempo real”[53].
- “Implementar trazabilidad de cambios”[53].

4. **Estrategias de Despliegue Seguro**

   Las prácticas avanzadas de CD incluyen estrategias que mejoran tanto la seguridad como la confiabilidad:

   - Blue/Green Deployments: “Mantener dos entornos idénticos permite rollbacks rápidos en caso de problemas de seguridad”[58].
   - Canary Releases: “Consiste en implementar características para un subconjunto de usuarios lo cual permite detectar problemas de seguridad antes de un despliegue completo”[58].
   - Feature Flags: “Permite desactivar características complejas sin necesidad de restablecer todo el despliegue”[58].

5. **Gestión de Vulnerabilidades y Respuesta a Incidentes**

   Siguiendo las recomendaciones de las buenas prácticas de seguridad de la información para empresas, es necesario implementar:

   **Metodología de Respuesta a Incidentes:**

   - Establecimiento de procedimientos claros para identificación y clasificación de incidentes.
   - Definición de roles y responsabilidades durante incidentes de seguridad.
   - Implementación de planes de comunicación y escalamiento.
   - Realizar un análisis post-incidente para mejora continua.

   **Gestión de Vulnerabilidades:**

- Establecer diversos procesos para evaluar y priorizar vulnerabilidades.
- Implementar parches y actualizaciones de manera sistemática.
- Mantener inventarios actualizados de componentes y dependencias.

### 2.1.5 Observabilidad y Monitoreo de Microservicios

La operación eficiente de arquitecturas de microservicios requiere una comprensión profunda del comportamiento del sistema en tiempo real. A medida que las aplicaciones monolíticas se descomponen en servicios más pequeños e independientes, la complejidad operacional aumenta exponencialmente, creando la necesidad de estrategias sofisticadas de observabilidad [59].

#### 2.1.5.1 Concepto de Observabilidad

En el libro de Cindy Dtidharan “Observabilidad de sistemas distribuidos”nos menciona que “La observabilidad en sistemas distribuidos se fundamenta en tres pilares complementarios que proporcionan una visión holística del comportamiento del sistema. Esta aproximación multidimensional permite a los equipos de desarrollo y operaciones comprender no solo qué está sucediendo en el sistema, sino por qué está ocurriendo, facilitando la identificación proactiva de problemas y la optimización continua del rendimiento”[60].

**Los Tres Pilares de la Observabilidad**

**Logs (Registros)**

Un registro de eventos es un registro inmutable con marca de tiempo de eventos discretos que ocurrieron a lo largo del tiempo. Los registros de eventos, en general, se presentan en tres formatos, pero son fundamentalmente iguales: una marca de tiempo y una carga útil de contexto. Los tres formatos son:

- Texto plano: Un registro puede ser texto libre. Este también es el formato más común [60].
- Estructurado: Muy popularizado y defendido últimamente. Normalmente, estos registros se emiten en formato JSON [60].
- Binario: Piense en registros en formato Protobuf, binlogs de MySQL utilizados para replicación y recuperación en un momento dado, registros de diario de systemd y el formato pflog utilizado por el firewall BSD, pf, que a menudo sirve como interfaz para tcpdump [60].

La depuración de patologías raras o poco frecuentes en los sistemas suele implicar una depuración con un nivel de granularidad muy alto. Los registros de eventos, en particular, destacan al proporcionar información valiosa, junto con un amplio contexto sobre la larga cola que los promedios y percentiles no revelan. Por ello, los registros de eventos son especialmente útiles para descubrir comportamientos emergentes e impredecibles que presentan los componentes de un sistema distribuido [60].

**Métricas**

Las métricas son una representación numérica de datos medidos a lo largo de intervalos de tiempo. Las métricas pueden aprovechar el poder del modelado matemático y la predicción para obtener conocimiento del comportamiento de un sistema a lo largo de intervalos de tiempo, tanto en el presente como en el futuro. Dado que los números están optimizados para el almacenamiento, el procesamiento, la compresión y la recuperación, las métricas permiten una mayor retención de datos, así como una consulta más sencilla. Esto las hace ideales para crear paneles que reflejen tendencias históricas. Las métricas también permiten una reducción gradual de la resolución de los datos. Tras un cierto período de tiempo, los datos pueden agregarse con una frecuencia diaria o semanal [60].

**Trazas**

Una traza es la representación de una serie de eventos distribuidos, causalmente relacionados que codifican el flujo de solicitudes de extremo a extremo a través de un sistema distribuido [60].

Las trazas son una representación de registros; su estructura de datos se asemeja a la de un registro de eventos. Una sola traza puede proporcionar visibilidad tanto de la ruta recorrida por una solicitud como de su estructura. La ruta de una solicitud permite a los ingenieros de software y a los SRE comprender los diferentes servicios involucrados, y su estructura ayuda a comprender las coyunturas y los efectos de la asincronía en su ejecución [60].

Aunque los debates sobre el traza suelen centrarse en su utilidad en un entorno de microservicios, es justo sugerir que cualquier aplicación suficientemente compleja que interactúe con recursos, o mejor dicho, compita por ellos, como la red, el disco o un mutex de forma significativa, puede beneficiarse de las ventajas que ofrece el traza [60].

#### 2.1.5.2 Monitoreo de Microservicios

En este apartado hablaremos sobre algunas metodologías para monitorizar los microservicios, según expertos.

**Metodología USE (Utilization, Saturation, Errors)**

La metodología USE (traducido como Utilizacion, Saturacion, Errores), desarrollada por Brendan Gregg [61] nos dice que “Es una metodología para analizar el rendimiento de cualquier sistema. Dirige la construcción de una lista de verificación que, en el análisis de servidores, permite identificar rápidamente cuellos de botella o errores en los recursos. Comienza planteando preguntas y luego busca respuestas, en lugar de partir de métricas dadas (respuestas parciales) e intentar trabajar a la inversa. El método USE se basa en tres tipos de métricas y una estrategia para abordar un sistema complejo. He comprobado que resuelve aproximadamente el 80 % de los problemas de servidores con el 5 % del esfuerzo y, como demostraré, puede aplicarse a otros sistemas. Debe considerarse una herramienta, parte de un conjunto más

amplio de metodologías. Hay muchos tipos de problemas que no resuelve, lo que requerirá otros métodos y plazos más largos.”

**Metodologia RED**

Esta metodologia RED (Rate, Errors, Duration el acronimo significa tarifa, errores y duracion) fue desarrollada por Tom Wilkie [62] miembro de Grafana Labs y en una charla nos dice “El método USE no se aplica realmente a los servicios; se aplica al hardware, a los discos de red y cosas así. Realmente queríamos una filosofía de monitoreo orientada a microservicios, así que se nos ocurrió el Método RED.”

Para cada recurso, monitoree:

- Tarifa (el número de solicitudes por segundo)[62].
- Errores (la cantidad de solicitudes que fallan)[62].
- Duración (la cantidad de tiempo que tardan esas solicitudes)[62].

Ademas Tom menciona que “El Método RED es un buen indicador de qué tan felices estarán sus clientes. Si tiene una alta tasa de error, eso básicamente se transmite a sus usuarios y ellos reciben errores de carga de página. Si tienes una duración alta, tu sitio web es lento. Estas son métricas realmente buenas para crear alertas significativas y medir su SLA”[62].

**Las cuatro señales de oro**

Conocido en ingles como “The Four Golden Signals”proporcionado por Google [63] nos dice que ”son la latencia, el tráfico, los errores y la saturación. Si solo puede medir cuatro métricas de su sistema de cara al usuario, concéntrese en ellas.”

**Latencia**

El tiempo que se tarda en atender una solicitud. Es importante distinguir entre la latencia de las solicitudes exitosas y la de las solicitudes fallidas. Por ejemplo, un error HTTP 500 pro-

vocado por la pérdida de conexión a una base de datos u otro backend crítico podría atenderse muy rápidamente; sin embargo, como un error HTTP 500 indica una solicitud fallida, considerar 500 segundos en la latencia general podría generar cálculos erróneos. Por otro lado, ¡un error lento es incluso peor que uno rápido! Por lo tanto, es importante monitorizar la latencia de los errores, en lugar de simplemente filtrarlos [63].

**Tráfico**

Una medida de la demanda que se ejerce sobre su sistema, medida mediante una métrica general específica del sistema. Para un servicio web, esta medición suele ser la cantidad de solicitudes HTTP por segundo, posiblemente desglosadas según la naturaleza de las solicitudes (p. ej., contenido estático o dinámico). Para un sistema de streaming de audio, esta medición podría centrarse en la tasa de E/S de la red o en las sesiones concurrentes. Para un sistema de almacenamiento de clave-valor, esta medición podría ser la cantidad de transacciones y recuperaciones por segundo [63].

**Errores**

La tasa de solicitudes que fallan, ya sea explícitamente (p. ej., HTTP 500), implícitamente (por ejemplo, una respuesta HTTP 200 correcta, pero con contenido incorrecto) o por política (por ejemplo, ”Si se comprometió a tiempos de respuesta de un segundo, cualquier solicitud superior a un segundo es un error”). Cuando los códigos de respuesta del protocolo son insuficientes para expresar todas las condiciones de fallo, pueden ser necesarios protocolos secundarios (internos) para rastrear los modos de fallo parcial. La monitorización de estos casos puede ser radicalmente diferente: detectar HTTP 500 en el balanceador de carga puede ser eficaz para detectar todas las solicitudes que fallan por completo, mientras que solo las pruebas integrales del sistema pueden detectar que se está sirviendo contenido incorrecto [63].

**Saturación**

Qué “completo” es su servicio. Una medida de la fracción de su sistema, enfatizando los recursos que están más restringidos (por ejemplo, en un sistema con memoria restringida, mos-

trar memoria; en un sistema con E/S restringida, mostrar E/S). Tenga en cuenta que el rendimiento de muchos sistemas se degrada antes de alcanzar el 100 % de utilización, por lo que tener un objetivo de utilización es esencial [63].

En sistemas complejos, la saturación se puede complementar con una medición de carga de mayor nivel: ¿puede su servicio manejar adecuadamente el doble de tráfico, manejar solo un 10 % más de tráfico o manejar incluso menos tráfico del que recibe actualmente? Para servicios muy simples que no tienen parámetros que alteren la complejidad de la solicitud (por ejemplo, ”Dame un nonce.º ”Necesito un entero monótono globalmente único”) que rara vez cambian de configuración, un valor estático de una prueba de carga podría ser adecuado. Sin embargo, como se analizó en el párrafo anterior, la mayoría de los servicios necesitan utilizar señales indirectas como la utilización de la CPU o el ancho de banda de la red que tienen un límite superior conocido. Los aumentos de latencia suelen ser un indicador principal de saturación. Medir el tiempo de respuesta del percentil 99 durante una ventana pequeña (por ejemplo, un minuto) puede dar una señal muy temprana de saturación [63].

Por último, la saturación también se ocupa de las predicciones de saturación inminente, como ”Parece que su base de datos llenará su disco duro en 4 horas”[63].

**Herramientas: Prometheus, Grafana**

**Prometheus**

Es un sistema de código abierto Kit de herramientas de monitoreo y alerta creado originalmente en SoundCloud [64].

Las principales características de Prometheus son:

- Un multidimensional modelo de datos con datos de series temporales identificados por nombre métrico y pares clave/valor [64].
- PromQL, a lenguaje de consulta flexible para aprovechar esta dimensionalidad [64].

- No depende del almacenamiento distribuido; los nodos de servidor único son autónomos [64].
- La recopilación de series temporales se realiza a través de un modelo de extracción a través de HTTP [64].
- Empujando series temporales se admite a través de una puerta de enlace intermediaria [64].
- Los objetivos se descubren mediante el descubrimiento de servicios o la configuración estática [64].
- Compatibilidad con múltiples modos de gráficos y paneles de control [64].

**Grafana**

Según la pagina de Grafana nos dice que "El software de código abierto de Grafana (OSS) le permite consultar, visualizar, alertar y explorar sus métricas, registros y trazas donde se almacenen. Los complementos de fuente de datos de Grafana le permiten consultar fuentes de datos, incluidas bases de datos de series temporales como Prometheus y CloudWatch, herramientas de registro como Loki y Elasticsearch, bases de datos NoSQL/SQL como Postgres, herramientas de CI/CD como GitHub y muchos más. Grafana OSS le proporciona herramientas para mostrar esos datos en paneles en vivo con gráficos y visualizaciones perspicaces [65].

#### 2.1.5.3 Gestión de Logs Centralizados

**Concepto**

Segun Elastic, "La gestión de logs es el proceso continuo de recopilación, almacenamiento y procesamiento de datos de logs para análisis futuros. La gestión eficaz de logs es el primer paso para obtener información procesable a partir de los datos de logs, lo que permite una resolución de problemas optimizada y un mejor rendimiento del sistema. Esto da lugar a un análisis forense más detallado y a una gestión de recursos más eficiente"[66].

Ademas menciona que “Un enfoque centralizado de gestión de logs se basa en herramientas que agregan datos de sistemas operativos y otras fuentes en una plataforma unificada. La gestión de logs permite a las organizaciones ordenar y almacenar sus datos de logs. A partir de ahí, los equipos de IT, DevOps y SecOps pueden usar el monitoreo de logs para hacer un seguimiento de la actividad, mientras que el análisis de logs ayuda a descubrir amenazas potenciales de forma proactiva”[66].

En la práctica, “la gestión de logs proporciona acceso centralizado a los datos de logs en un marco de trabajo que se puede ordenar y tiene capacidad de búsqueda. Junto con métricas, rastreos y perfiles, la gestión de logs es una señal fundamental tanto para los equipos de observabilidad como para los de seguridad”[66].

**Herramientas**

Dentro de la gestion de logs existen algunas herramientas, a continuación las siguientes.

**Pila elástica**

Segun Elastic, “Es un conjunto de componentes rápidos y altamente escalables —Elasticsearch, Kibana, Beats, Logstash— y otros, que juntos le permiten tomar datos de forma segura de cualquier fuente, en cualquier formato y luego buscarlos, analizarlos y visualizarlos”[67].

**Elasticsearch**

“Elasticsearch es el motor de búsqueda y análisis distribuido en el corazón de la pila elástica. Proporciona búsqueda y análisis casi en tiempo real para todo tipo de datos. Ya sea que tenga texto estructurado o no estructurado, datos numéricos o datos geoespaciales, Elasticsearch puede almacenarlo e indexarlo de manera eficiente de una manera que admite búsquedas rápidas. Elasticsearch proporciona una API REST que le permite almacenar datos en Elasticsearch y recuperarlos. La API REST también proporciona acceso a las capacidades de búsqueda y análisis de Elasticsearch”[67].

**Kibana**

“Kibana es la herramienta para aprovechar sus datos de elasticsearch y administrar la pila elástica. Úselo para analizar y visualizar los datos almacenados en Elasticsearch. Kibana también es el hogar de las soluciones de búsqueda, observabilidad y seguridad”[67].

**Beats**

“Beats son cargadores de datos que instala como agentes en sus servidores para enviar datos operativos a Elasticsearch. Los ritmos están disponibles para muchos escenarios de datos de observabilidad estándar, incluidos datos de auditoría, archivos de registro y revistas, datos en la nube, disponibilidad, métricas, tráfico de red y registros de eventos de Windows”[67].

**Logstash**

“Logstash es un motor de recopilación de datos con capacidades de tuberías en tiempo real. Puede unificar dinámicamente datos de fuentes dispares y normalizar los datos en destinos de su elección. Logstash admite una amplia gama de complementos de entrada, filtro y salida, con muchos códecs nativos que simplifican aún más el proceso de ingestión”[67].

**APM**

“Elastic APM es un sistema de monitoreo de rendimiento de la aplicación basado en la pila elástica. Le permite monitorear los servicios y aplicaciones de software en tiempo real, mediante la recopilación de información de rendimiento detallada sobre el tiempo de respuesta para solicitudes entrantes, consultas de bases de datos, llamadas a cachés, solicitudes HTTP externas y más. Esto hace que sea fácil identificar y solucionar los problemas de rendimiento rápidamente”[67].

**GrayLog**

Segun Graylog, “Graylog es una potente solución de gestión de eventos e información de seguridad (SIEM) y una plataforma de análisis de registros que centraliza, protege y monitorea datos generados por máquinas en diversas fuentes. Ya sea que se utilice para ciberseguridad, operaciones de TI o cumplimiento, Graylog brinda a los equipos información útil a través de

capacidades rápidas de búsqueda, alerta y visualización”[68].

**Loki (Grafana)**

Segun Grafana, “Loki es un sistema de agregación de registros multiinquilino, altamente disponible y escalable horizontalmente inspirado en Prometheus. Está diseñado para ser muy rentable y fácil de operar. No indexa el contenido de los registros, sino un conjunto de etiquetas para cada flujo de registros”[69].

**Fluentd**

Segun Fluentd, “es un recopilador de datos de código abierto para una capa de registro unificada. Fluentd le permite unificar la recopilación y el consumo de datos para un mejor uso y comprensión de los datos”[70].

**Syslog-ng**

Segun OneIdentity, “La aplicación syslog-ng es una aplicación de registro de sistemas flexible y altamente escalable que es ideal para crear soluciones de registro centralizadas y confiables. Entre otros, syslog-ng OSE le permite lo siguiente”[71].

“La aplicación Syslog-NG OSE le permite enviar los mensajes de registro de sus hosts a servidores remotos utilizando los últimos estándares de protocolo. Puede recopilar y almacenar sus datos de registro centralmente en servidores de registro dedicados. Transferir mensajes de registro utilizando el protocolo TCP asegura que no se pierdan mensajes”[71].

#### 2.1.5.4 Trazas Distribuidas

Según Sridharan en “Distributed Systems Observability”, “una traza distribuida representa el recorrido completo de una solicitud a través de un sistema distribuido y captura información temporal y contextual sobre cada operación realizada. Esta técnica permite visualizar las dependencias entre servicios, identificar cuellos de botella de rendimiento y diagnosticar fallos

en sistemas en los que una única transacción puede involucrar a decenas o cientos de servicios interconectados"[60].

A medida que la arquitectura de microservicios ha ido evolucionando, también han surgido herramientas especializadas para implementar la traza distribuida y facilitar el monitoreo de las solicitudes entre los diferentes servicios. Entre las más conocidas se encuentran Jaeger, Zipkin y OpenTelemetry, las cuales ofrecen distintas funcionalidades y enfoques para adaptarse a diversos escenarios de implementación.

**Jaeger**

Segun Jaeger, "es una plataforma de rastreo distribuida lanzada como código abierto por Uber Technologies en 2016 y donada a Cloud Native Computing Foundation, donde es un proyecto graduado"[72].

Con Jaeger puedes:

1. Monitorear y solucionar problemas de trabajo distribuido [72].
2. Identificar cuellos de botella de rendimiento [72].
3. Rastrear las causas raíz [72].
4. Analizar dependencias de servicios [72].

**Zipkin**

Segun Zipkin, "es un sistema de rastreo distribuido. Ayuda a recopilar datos de tiempo necesarios para solucionar problemas de latencia en las arquitecturas de servicio. Las características incluyen la recopilación y la búsqueda de estos datos"[73].

"Si tiene una ID de rastreo en un archivo de registro, puede saltar directamente a él. De lo contrario, puede consultar según los atributos como el servicio, el nombre de la operación, las

etiquetas y la duración. Se resumirán algunos datos interesantes para usted, como el porcentaje de tiempo dedicado a un servicio y si las operaciones fallaron o no"[73].

**OpenTelemetry**

OpenTelemetry, también conocida como OTEL, "es un marco de observabilidad de código abierto neutral en el proveedor para instrumentar, generar, recopilar y exportar datos de telemetría, como trazas, métricas y registros"[74].

"Como estándar de la industria, OpenTelemetry es compatible con más de 40 proveedores de observabilidad, integrados por muchas bibliotecas, servicios y aplicaciones, y adoptada por numerosos usuarios finales"[74].

### 2.1.6 CI/CD en Contextos Educativos

#### 2.1.6.1 Desafíos específicos en entornos de formación

La adopción de prácticas para implementar CI/CD en entornos educativos plantea varios desafíos. "Uno de los principales es la curva de aprendizaje. Muchos estudiantes y docentes carecen de conocimientos previos sobre herramientas como Git, Jenkins o GitHub Actions, por lo que es necesario proporcionarles una formación previa considerable para poder integrarlas de manera efectiva en los cursos"[75].

"Otro aspecto importante es la limitación de recursos: muchas instituciones educativas carecen de servidores potentes, entornos de integración adecuados o acceso a runners dedicados para ejecutar flujos de trabajo de manera eficiente. Esto limita la posibilidad de simular entornos reales de desarrollo y dificulta la experiencia inmersiva que busca replicar el mundo profesional"[58].

"También existe una resistencia institucional y docente. Algunos profesores no están familiarizados con las metodologías ágiles ni con los conceptos DevOps, lo que supone una barrera

para la aplicación efectiva de CI/CD en el plan de estudios”[75].

#### 2.1.6.2 Beneficios pedagógicos de la automatización CI/CD

En el panorama actual de la tecnología, hemos observado como evoluciona a cada instante, por eso algunas instituciones educativas como los colegios y universidades se están enfocando en adaptarse para mejorar las experiencias educativas y eficiencia operativa. Algunos de los enfoques que mas esta impactando son la adopcion de canales de CI/CD de DevOps.

Varenya Z nos explica en su sitio web oficial que “ Las organizaciones educativas se beneficiarán significativamente al adoptar CI/CD impulsado por DevOps, lo que afectará múltiples dominios operativos y estratégicos”[76]. Algunos de los beneficios que Varenya Z nos menciona se detallan, a continuación:

1. **Tiempo más rápido de comercialización para mejoras de aprendizaje**

   “Permitir la implementación rápida de funciones significa que nuevas herramientas educativas, módulos de aprendizaje o actualizaciones de políticas pueden reflejarse rápidamente en plataformas digitales, siguiendo el ritmo de las tendencias educativas”[76].

2. **Estabilidad y confiabilidad mejoradas del sistema**

   “Las pruebas y el monitoreo automatizados reducen los tiempos de inactividad inesperados de los portales de estudiantes, los sistemas de gestión del aprendizaje (LMS) y las aplicaciones administrativas”[76].

3. **Aumento de los ciclos de retroalimentación e iteración**

   “Con una integración continua, los comentarios de educadores y estudiantes se pueden incorporar rápidamente a las mejoras del software, adaptando el entorno de aprendizaje más estrechamente a las necesidades reales”[76].

4. **Optimización de recursos**

"DevOps reduce las tareas manuales redundantes, lo que permite a los equipos de TI centrarse en la innovación en lugar de en la extinción de incendios, lo cual es crucial dadas las limitaciones presupuestarias en muchos entornos educativos"[76].

#### 2.1.6.3 Experiencias reportadas y mejores prácticas adaptadas a la academia

Dentro de este apartado existen casos reportados y uno de ellos es el caso del Instituto de Tecnología de Georgia, donde se implementó un pipeline CI/CD para validar tareas de aprendizaje automático (Machine Learning). Y gracias a esta estrategia implementada, se logró reducir en un 57 % los reportes de errores en foros estudiantiles, al automatizar pruebas y entregas de forma eficiente mediante GitHub Actions como motor de CI/CD, Docker y scripts de validación [75].

Asi mismo, en el sitio web de Codefresh se resumen 11 buenas prácticas de CI/CD que pueden aplicarse en el entorno educativo, algunos de ellos son:

- **Automatizar pruebas**

  "Las pruebas de calidad automatizadas se dividen en tres categorías: unitarias, de integración y funcionales. Las pruebas unitarias verifican funciones específicas en busca de errores. Las pruebas de integración garantizan el funcionamiento adecuado de múltiples funciones juntas y, a menudo, encuentran problemas inusuales y, finalmente, las pruebas funcionales verifican el funcionamiento final con los resultados esperados"[77].

- **Construir solo una vez**

  "Un CI/CD mal implementado a menudo reconstruirá el mismo artefacto varias veces. Una vez que se espera que los cambios sean definitivos, el artefacto solo debe construirse una vez. Esto evita que diferentes compilaciones obtengan resultados de prueba diferentes. Al trabajar con imágenes de contenedores, estas se pueden promocionar con etiquetas adicionales"[77].

- **Utilice tuberías compartidas (DRY)**

> “La gestión de canalizaciones de CI/CD es un componente central del proceso ci/cd, el uso de una canalización CI/CD diferente para cada aplicación o microservicio genera duplicación innecesaria y complejidad de gestión. En su lugar, utilice canalizaciones compartidas para que usted “no se repita” (DRY). Los pipelines compartidos, como los de Codefresh, dependen de activadores de eventos para establecer el contexto para “hidratar” un pipeline que se puede reutilizar en muchas aplicaciones y microservicios”[77].

## 2.2 Antecedentes

La investigación “Continuous Testing in CI/CD Pipelines”[30] analiza la integración de pruebas automatizadas en pipelines de CI/CD, destacando su papel crítico para garantizar calidad en entornos ágiles. La investigación se centró en resolver desafíos como pruebas inestables conocido como flaky tests, disparidad de entornos y gestión de datos, mediante estrategias como IA para optimización de pruebas, containerización (Docker/Kubernetes) y enfoques shift-left/shift-right. Para validar estas soluciones, se analizaron casos de estudio en empresas como TCS, Comcast y Ahold Delhaize, donde se implementaron pruebas continuas en sistemas distribuidos. En TCS, el uso de marcos de pruebas basados en IA y entornos cloud redujo defectos críticos en un 40 % y aceleró despliegues en un 60 % para un cliente bancario. Comcast integró pruebas self-healing y de regresión automatizadas, disminuyendo fallos en un 35 % en servicios como Xfinity Mobile. Por su parte, Ahold Delhaize logró un 50 % menos tiempo en ciclos de release mediante entornos containerizados y análisis de datos en e-commerce. Los resultados demostraron que la automatización de pruebas continuas, combinada con containerización y paralelización de pruebas, reduce errores en producción y optimiza la entrega de software, especialmente en arquitecturas complejas como microservicios.

El artículo “Integrating Performance Testing into CI/CD Pipelines for Test Automation”[78] se realizo una investigación con el objetivo principal de demostrar cómo se puede integrar de manera efectiva el testing de rendimiento en flujos de trabajo CI/CD automatizados. Para ello,

se definieron los conceptos clave relacionados con las pruebas de rendimiento y su relevancia en el ciclo de vida del desarrollo de software. El desarrollo del trabajo se estructuró en varias etapas. En primer lugar, se establecieron los fundamentos teóricos sobre las pruebas de rendimiento y los pipelines CI/CD. También se exploraron diferentes tipos de pruebas, como las de carga de estrés, de pico y de retención, y su función a la hora de detectar cuellos de botella y problemas de escalabilidad. Se propusieron estrategias para automatizar estas pruebas con herramientas como JMeter, Gatling, Jenkins, GitLab CI, Prometheus y Grafana, que se pueden integrar directamente en las fases del flujo de trabajo. También se incluyeron recomendaciones prácticas para garantizar unas pruebas efectivas, como el uso de datos realistas, la supervisión de recursos y la ejecución en entornos aislados.

Los resultados de la investigación demostraron que integrar las pruebas de rendimiento en los flujos de trabajo de CI/CD permite detectar fallos en una fase temprana del ciclo de desarrollo, lo que contribuye a reducir costos y evitar problemas en producción. Además, se consiguió un ciclo de retroalimentación más rápido, lo que mejoró la eficiencia del equipo de desarrollo. El seguimiento continuo y la visualización de métricas permitieron tomar decisiones más informadas y basadas en datos. De modo que la colaboración entre los equipos de desarrollo, pruebas y operaciones se fortaleció al asumir una responsabilidad compartida sobre el rendimiento del sistema. En conclusión, la integración de las pruebas de rendimiento en CI/CD es una práctica esencial para el desarrollo de software moderno, robusto y de calidad.

El estudio “CI/CD PARA ARQUITECTURAS DE MICROSERVICIOS” [17] el proyecto consistió en implementar una arquitectura de integración continua/entrega continua (CI/CD) para una aplicación basada en microservicios. El proyecto incluyó el diseño, el desarrollo y la automatización del despliegue de una aplicación web con herramientas como Jenkins, Docker, SonarQube y Git. El trabajo se dividió en las siguientes fases: aprendizaje de herramientas, desarrollo de la aplicación, integración de CI/CD con Jenkins, contenedorización con Docker y documentación. Para lo cual se aplicaron metodologías ágiles (Scrum) para la gestión del proyecto, adicional se utilizó integración continua para validar el código con cada cambio y se implementaron pruebas automáticas con JUnit. Se monitorizó el rendimiento con NetDa-

ta y se controló la calidad del código con SonarQube. El sistema CI/CD se logro automatizar por completo el ciclo de vida de desarrollo de la aplicación, lo que redujo los errores manuales, ademas mejoró la calidad del código y agilizó el proceso de despliegue. Se consiguió una infraestructura estable, escalable y más transparente, lo que facilitó futuras ampliaciones y mejoras.

En la investigación “Strategies for Automating Tests in A Microservices-Based Application” [79] se investigaron y propusieron estrategias para automatizar las pruebas en aplicaciones basadas en microservicios, abordando distintos tipos como pruebas unitarias, de integración, de contrato, end-to-end y de rendimiento. Se analizaron los desafíos propios del entorno distribuido de microservicios y se propusieron soluciones con herramientas específicas como JUnit, Postman, Pact, Selenium, JMeter, entre otras. Las estrategias incluyeron virtualización de servicios, orquestación de pruebas, integración con pipelines CI/CD y uso de contratos consumidor-proveedor. Se concluyó que la automatización de pruebas mejora la confiabilidad, escalabilidad y rendimiento de los sistemas con microservicios. Además, aplicar buenas prácticas y herramientas adecuadas permite detectar errores tempranamente, agilizar el desarrollo y fortalecer la calidad del software.

En el estudio “Microservices in Organizations” [80] se abordó la implementación y automatización de arquitecturas basadas en microservicios, así como su impacto estratégico en las organizaciones. En los tres estudios se desarrollaron aplicaciones utilizando microservicios, integrando herramientas como Jenkins, Docker, SonarQube, Pact, Selenium y JMeter para automatizar pruebas (unitarias, de integración, contrato, end-to-end y de rendimiento) dentro de pipelines CI/CD. Los análisis compararon microservicios con arquitecturas monolíticas, destacando ventajas como modularidad, escalabilidad, despliegue independiente y adaptación ágil al cambio. También se identificaron desafíos como la complejidad operativa, costos y necesidad de estructuras organizacionales alineadas. Como resultado, se concluyó que los microservicios permiten a las organizaciones mejorar la calidad del software, acelerar el desarrollo, reducir errores y adaptarse rápidamente a entornos competitivos y disruptivos, siempre que su adopción se realice de forma planificada y con las capacidades técnicas necesarias.

# CAPÍTULO III

# MATERIALES Y MÉTODOS

## 3.1 Tipo de investigación

El presente trabajo de integración curricular se enmarca en una investigación de tipo cuantitativo, debido a que se basa en la medición y análisis de variables relacionadas con la eficiencia del proceso de desarrollo e implementación de software.

Esta investigación se enmarca en este enfoque debido a que busca desarrollar e implementar un pipeline automatizado de Integración Continua y Despliegue Continuo (CI/CD) para una plataforma de gestión académica basada en microservicios. La propuesta surge como respuesta a la necesidad de optimizar los procesos de integración, pruebas y despliegue del software, abordando una problemática presente en el desarrollo de este tipo de aplicaciones. Además de implementar la solución, el estudio permite analizar su viabilidad técnica y evaluar los beneficios obtenidos mediante indicadores como el tiempo de implementación, la frecuencia de integración y la reducción de errores durante el proceso.

### 3.1.1 Enfoque metodológico

Para llevar a cabo el desarrollo de la propuesta se optó por un enfoque experimental, ya que fue necesario implementar la solución en un entorno controlado y analizar su comportamiento a partir de los resultados obtenidos. Este enfoque permitió comprobar el funcionamiento del pipeline CI/CD y determinar si cumple con los objetivos planteados. De forma complementaria, se utilizó un modelo de desarrollo iterativo, mediante el cual la solución se construyó de manera gradual, incorporando mejoras y validaciones en cada etapa antes de avanzar a la

siguiente.

Además, se empleó la metodología ágil SCRUM como marco de gestión del proyecto, lo cual facilitó la organización del trabajo en ciclos cortos de desarrollo (sprints), promoviendo la entrega incremental de funcionalidades y la retroalimentación continua.

La elección de SCRUM se fundamenta en la naturaleza modular de la arquitectura de microservicios, donde cada componente puede desarrollarse, probarse y desplegarse de forma independiente. Este enfoque permitió estructurar el proyecto en sprints de dos semanas, optimizando la gestión del tiempo, la adaptación a cambios y la resolución progresiva de los desafíos técnicos identificados durante la implementación.

### 3.1.2 Variables de estudio

En el presente trabajo se identifican y definen las variables de estudio con el propósito de establecer la relación entre la implementación de la solución propuesta y los resultados obtenidos en el proceso de desarrollo de software basado en microservicios. Estas variables permiten estructurar el análisis y la validación del impacto del pipeline automatizado de Integración Continua y Despliegue Continuo (CI/CD).

**Variable independiente**

La variable independiente corresponde a la automatización del pipeline de Integración Continua y Despliegue Continuo (CI/CD), la cual representa el elemento central de la propuesta desarrollada en esta investigación. Esta variable se enfoca en la implementación de un flujo de trabajo automatizado que integra procesos de construcción, pruebas y despliegue de aplicaciones, reduciendo la intervención manual y estandarizando el ciclo de vida del software.

Dentro de esta variable se consideran las siguientes dimensiones:

- Integración Continua (CI): ejecución automática de procesos de construcción y pruebas

del código fuente tras cada cambio realizado en el repositorio.

- Despliegue Continuo (CD): automatización del proceso de liberación y despliegue de artefactos en entornos de ejecución.
- Automatización de pruebas: incorporación de pruebas unitarias, de integración y validaciones automáticas dentro del pipeline.
- Herramientas tecnológicas: uso de plataformas como GitHub Actions, Docker y Kubernetes para la implementación y orquestación del pipeline.

Estas dimensiones permiten evaluar el nivel de automatización alcanzado y su correcta integración dentro del entorno de microservicios.

**Variables dependientes**

La variable dependiente está relacionada con la efectividad del proceso de desarrollo y despliegue de software en arquitecturas de microservicios, y está directamente afectada por la ejecución del pipeline automatizado.

Esta variable se analiza mediante las siguientes dimensiones:

- Tiempo de implementación: tiempo necesario para pasar una versión del software de su desarrollo a producción.
- Frecuencia de implementaciones: Cantidad de despliegues que se realizan en un tiempo determinado.
- Tasa de errores: cantidad de fallos detectados durante las etapas de integración, prueba o implementación.
- Confiabilidad del sistema: estabilidad del software desplegado y reducción de fallos en producción.
- Facilidad de mantenimiento: capacidad de realizar cambios, correcciones y mejoras de manera ágil y controlada.

Estas métricas facilitan la medición del impacto del pipeline CI/CD en la optimización del ciclo de vida del software, mostrando mejoras en la calidad, velocidad y confiabilidad del proceso de desarrollo.

**Relación de las variables**

La relación entre las variables de estudio se basa en que la implementación de un pipeline automatizado de CI/CD afecta directamente la mejora de la eficiencia del proceso de desarrollo y despliegue de microservicios. En este sentido, se espera que un mayor nivel de automatización ayude a reducir tiempos, minimizar errores y aumentar la confiabilidad del sistema, validando así la propuesta planteada en esta investigación.

## 3.2 Materiales, herramientas e insumos utilizados

### 3.2.1 Herramientas de desarrollo

Para el desarrollo del código y la gestión de versiones, se empleó Visual Studio Code como Entorno de Desarrollo Integrado (conocido por sus siglas IDE) por su compatibilidad con extensiones de contenedores.

Para el control de versiones, se utilizó Git junto con el repositorio en GitHub, lo que permitió guardar los distintos estados del software a través de commits y branches.

Para la gestión de paquetes en el Frontend, se eligió npm (Node Package Manager) junto con Vite, el cual proporciona un servidor de desarrollo rápido y optimizaciones de construcción para la aplicación React.

### 3.2.2 Herramientas de contenerización y despliegue

Para la creación de imágenes de contenedores que encapsulan los microservicios y sus dependencias, se utiliza Docker. Esta herramienta permitió estandarizar el entorno de ejecución, aislar el código y las librerías de la máquina host, lo cual es esencial para la reproducibilidad del sistema.

Para la orquestación y despliegue de estos contenedores, se seleccionó Kubernetes, gestionado a través de Minikube en el entorno local. Esta combinación permite simular un clúster de producción (Minikube) y gestionar pods, servicios y volúmenes de manera declarativa, facilitando el escalamiento horizontal y la alta disponibilidad de la aplicación.

### 3.2.3 Herramientas de automatización CI/CD

La herramienta central para la automatización fue GitHub Actions, el servicio de integración continua de GitHub. Se configuró un flujo de trabajo automatizado que se activa ante eventos como push o pull request. Este flujo de trabajo se encarga de orquestar la ejecución de las pruebas unitarias, la construcción de las imágenes Docker y su posterior subida al registro de contenedores. El uso de Docker Hub o GitHub Container Registry fue necesario para almacenar las imágenes generadas, permitiendo que Kubernetes las descargue automáticamente para el despliegue, completando el ciclo de CI/CD.

### 3.2.4 Herramientas de observabilidad

Para el monitoreo y visualización de métricas de la infraestructura y el comportamiento de la aplicación, se implementó Grafana. Esta herramienta se configuró para consumir métricas de uso de CPU, memoria y red de los pods de Kubernetes y las aplicaciones. A través de Grafana, se crearon paneles visuales que permiten observar en tiempo real el estado de salud de los microservicios Usuarios, Tareas, Notas, Asistencia, permitiendo identificar cuellos de bo-

tella de recursos o errores de lógica de negocio al observar picos inesperados en los gráficos de rendimiento.

### 3.2.5 Infraestructura y entorno de ejecución

El desarrollo del proyecto se ejecutó en un entorno híbrido que combina herramientas locales y en la nube. Para la fase de desarrollo y pruebas locales, se utilizó Minikube como orquestador local dentro de una máquina virtual, aprovechando los recursos de la computadora anfitriona. Sin embargo, para el proceso de construcción automatizada (CI), se dependió de los GitHub Actions Runners entornos de ejecución provisionados por GitHub para la compilación y pruebas. Se requiere una conexión a Internet estable para que GitHub Actions pueda acceder al repositorio, descargar el código, descargar las imágenes base Docker, Node.js, etc; y ejecutar los comandos de construcción y despliegue sin manual de intervención.

## 3.3 Planificación del proyecto por sprints

La planificación del proyecto se estructuró bajo la metodología ágil SCRUM, organizando el desarrollo en ocho sprints de dos semanas cada uno. Esta planificación permitió abordar de manera incremental la construcción del sistema basado en microservicios, así como la implementación del pipeline CI/CD automatizado.

Cada sprint contempló actividades específicas, entregables definidos y objetivos orientados a la construcción progresiva de la solución, garantizando retroalimentación continua y mejora iterativa.

### 3.3.1 Estructura general de los sprints

La estructura general de los sprints se definió considerando las fases del ciclo de vida del desarrollo de software en entornos DevOps, incluyendo análisis, desarrollo, integración, des-

pliegue y validación.

### 3.3.2 Sprint 1: Análisis de requerimientos y diseño inicial

En este sprint se realizó el levantamiento de requerimientos funcionales y no funcionales, así como el diseño de la arquitectura basada en microservicios.

**Tabla II**.
Sprint 1

| | |
|---|---|
| **Objetivo** | Definir los requerimientos del sistema y diseñar la arquitectura base |
| **Actividades** | ▪ Identificación de stakeholders<br>▪ Definición de requerimientos funcionales y no funcionales<br>▪ Diseño de arquitectura de microservicios<br>▪ Modelado inicial del sistema |
| **Entregables** | Documento de requerimientos y diagrama arquitectónico |

### 3.3.3 Sprint 2: Implementación de microservicios backend

Se desarrollaron los microservicios principales del sistema académico.

**Tabla III**.
Sprint 2

| | |
|---|---|
| **Objetivo** | Implementar los microservicios backend |
| **Actividades** | ▪ Desarrollo de microservicio de usuarios<br>▪ Desarrollo de microservicio de tareas<br>▪ Desarrollo de microservicio de notas<br>▪ Desarrollo de microservicio de asistencia |
| **Entregables** | APIs funcionales y documentadas |

### 3.3.4 Sprint 3: Desarrollo del frontend e integración inicial

Se desarrolló la interfaz de usuario e integración con los servicios backend.

**Tabla IV**.
Sprint 3

| **Objetivo** | Construir el frontend e integrar servicios |
|---|---|
| **Actividades** | ▪ Desarrollo de interfaz en VITE<br>▪ Consumo de APIs<br>▪ Pruebas iniciales de integración |
| **Entregables** | Frontend funcional conectado al backend |

### 3.3.5 Sprint 4: Persistencia, autenticación y gestión de archivos

Se implementaron mecanismos de seguridad y almacenamiento.

**Tabla V**.
Sprint 4

| **Objetivo** | Implementar seguridad y persistencia |
|---|---|
| **Actividades** | ▪ Integración con base de datos<br>▪ Implementación de autenticación (JWT)<br>▪ Gestión de archivos |
| **Entregables** | Sistema con autenticación y almacenamiento |

### 3.3.6 Sprint 5: Contenerización y despliegue semi local

Se realizó la contenerización de los servicios y despliegue en entorno local.

**Tabla VI**.
Sprint 5

| **Objetivo** | Contenerizar y desplegar el sistema |
|---|---|
| **Actividades** | ▪ Creación de Dockerfiles<br>▪ Configuración de Docker Compose<br>▪ Despliegue en entorno local (Minikube) |
| **Entregables** | Sistema ejecutándose en contenedores |

### 3.3.7 Sprint 6: Automatización del pipeline CI/CD

Se implementó el pipeline automatizado utilizando herramientas DevOps.

**Tabla VII**.
Sprint 6

| **Objetivo** | Automatizar integración y despliegue |
|---|---|
| **Actividades** | ▪ Configuración de GitHub Actions<br>▪ Automatización de build y test<br>▪ Integración con Docker Registry<br>▪ Despliegue automático en Kubernetes |
| **Entregables** | Pipeline CI/CD funcional |

### 3.3.8 Sprint 7: Observabilidad y monitoreo

Se implementaron herramientas de monitoreo del sistema.

**Tabla VIII**.
Sprint 7

| **Objetivo** | Implementar monitoreo y observabilidad |
|---|---|
| **Actividades** | ▪ Configuración de Prometheus<br>▪ Implementación de Grafana |
| **Entregables** | Dashboard de monitoreo operativo |

### 3.3.9 Sprint 8: Validación y ajustes finales

Se evaluó el sistema y se realizaron mejoras finales.

**Tabla IX**.
Sprint 8

| **Objetivo** | Validar y optimizar el sistema |
|---|---|
| **Actividades** | ▪ Pruebas finales del sistema<br>▪ Evaluación de métricas (tiempo, errores)<br>▪ Ajustes de rendimiento |
| **Entregables** | Sistema validado y optimizado |

## 3.4 Definición de Requisitos del Sistema

Para guiar el diseño e implementación de la solución, se establecieron requisitos funcionales y no funcionales basados en los objetivos del proyecto y las necesidades de un entorno de microservicios.

### 3.4.1 Requisitos Funcionales

Los requisitos funcionales definen las acciones específicas que el sistema y el pipeline deben ejecutar:

- RF1 - Gestión de dominios académicos (Microservicios): El sistema debe exponer funcionalidades a través de microservicios independientes para la gestión de usuarios (autenticación y roles: estudiante, docente, administrador), tareas académicas, calificaciones (notas) y control de asistencias.
- RF2 - Automatización de Integración Continua (CI): El pipeline debe detectar automáticamente cambios en el repositorio de código fuente (GitHub) para ejecutar tareas de compilación, instalación de dependencias y ejecución de pruebas unitarias de manera autónoma.
- RF3 - Construcción y publicación de artefactos: El pipeline debe generar automáticamente imágenes Docker para cada microservicio (backend y frontend) tras la aprobación de las pruebas, y publicarlas en un registro de contenedores.
- RF4 - Despliegue Continuo (CD) en orquestador: El flujo automatizado debe aplicar los manifiestos de Kubernetes (YAML) para desplegar o actualizar las nuevas imágenes en el clúster de forma autónoma, sin intervención manual.
- RF5 - Persistencia de datos: El sistema debe implementar mecanismos de almacenamiento persistente mediante volúmenes (PVCs) para asegurar que la información de las bases de datos relacionales (PostgreSQL) y documentales (MongoDB) no se pierda ante reinicios de los contenedores.

### 3.4.2 Requisitos No Funcionales

Los requisitos no funcionales establecen las características de calidad, rendimiento y diseño arquitectónico:

- RNF1 - Escalabilidad independiente: La arquitectura debe permitir escalar cada microservicio de manera individual aumentando o disminuyendo réplicas en función de la demanda, aprovechando las capacidades nativas de Kubernetes.

- RNF2 - Bajo acoplamiento y alta cohesión: Los microservicios deben comunicarse exclusivamente mediante APIs REST ligeras. Un fallo en un servicio por ejemplo, el de tareas no debe provocar la caída del sistema completo a esto se lo conoce como tolerancia a fallos.
- RNF3 - Observabilidad y monitoreo: La infraestructura debe integrar herramientas de monitoreo Prometheus y Grafana para la recolección y visualización en tiempo real de métricas de rendimiento, consumo de recursos (CPU, memoria) y estado de salud de los pods.
- RNF4 - Trazabilidad y Repetibilidad: Cada ejecución del pipeline CI/CD debe estar asociada a un commit específico, garantizando que el proceso de construcción y despliegue sea repetible, auditable y trazable.
- RNF5 - Eficiencia operativa: El tiempo total de ejecución del pipeline desde el push del código hasta el despliegue en Kubernetes debe ser significativamente menor al tiempo requerido por un proceso de despliegue manual.
- RNF6 - Multi-lenguaje (Portabilidad del Pipeline): El diseño del pipeline debe ser agnóstico al lenguaje de programación, permitiendo en el futuro integrar microservicios desarrollados en distintos stacks tecnológicos sin rediseñar la estructura de automatización.
- RNF7 - Seguridad (Gestión de secretos): El pipeline debe evitar el hardcoding de credenciales, utilizando almacenes de secretos nativos para la gestión de tokens y contraseñas de bases de datos .
- RNF8 - Eficiencia de costos y recursos: El pipeline debe implementar estrategias de paralelismo y uso de caché de dependencias para minimizar el tiempo de cómputo

## 3.5 Diseño de la arquitectura de la solución

El diseño de la arquitectura de la solución se fundamenta en el paradigma de microservicios, el cual permite desarrollar sistemas distribuidos, escalables y altamente mantenibles. Esta arquitectura facilita la integración con buenas prácticas DevOps, especialmente en la implementación de pipelines automatizados de Integración Continua y Despliegue Continuo (CI/CD), objetivo central de la presente investigación.

La solución propuesta se orienta a una plataforma de gestión académica, donde cada funcionalidad se implementa como un servicio independiente, permitiendo su desarrollo, despliegue y escalabilidad de manera autónoma.

### 3.5.1 Arquitectura basada en microservicios

La arquitectura de microservicios se caracteriza por dividir el sistema en múltiples servicios pequeños, independientes y especializados, que se comunican entre sí mediante interfaces bien definidas, generalmente a través de APIs REST.

En este enfoque, cada microservicio encapsula una funcionalidad específica del dominio del sistema, lo que permite mejorar la modularidad, facilitar el mantenimiento y acelerar los ciclos de desarrollo.

**Tabla X**.
Características de la arquitectura de microservicios

| Característica | Descripción |
|---|---|
| Desacoplamiento | Cada servicio es independiente y puede evolucionar sin afectar a otros |
| Escalabilidad | Los servicios pueden escalarse de forma individual según la demanda |
| Despliegue independiente | Cada microservicio puede ser desplegado sin necesidad de reiniciar el sistema completo |
| Mantenibilidad | Facilita la identificación y corrección de errores |

### 3.5.2 Componentes principales del sistema

El sistema se compone de varios elementos que interactúan para garantizar su funcionamiento integral. A continuación, se describen los componentes principales:

**Tabla XI**.
Componentes principales del sistema

| Componente | Descripción |
|---|---|
| Frontend | Aplicación cliente desarrollada en VITE, encargada de la interacción con el usuario |
| Microservicios Backend | Servicios independientes encargados de la lógica de negocio (usuarios, tareas, asistencia, etc.) |
| API Gateway | Punto de entrada único para gestionar solicitudes y enrutar hacia los servicios |
| Base de datos | Sistemas de almacenamiento para cada microservicio |
| Contenedores Docker | Encapsulan los servicios para garantizar portabilidad |
| Kubernetes | Plataforma de orquestación para la gestión de contenedores |
| Pipeline CI/CD | Automatiza la integración, pruebas y despliegue del sistema |

### 3.5.3 Comunicación entre componentes

La comunicación entre el Frontend y los microservicios se realiza mediante protocolo HTTP/JSON, lo que estandariza la interacción y permite el uso de diferentes tecnologías para el Frontend (React) y el Backend (Node.js), manteniendo el débil acoplamiento entre ellos. Cada microservicio expone endpoints específicos que permiten la interacción con otros servicios y con el frontend.

Adicionalmente, la comunicación interna dentro del clúster de Kubernetes se realiza mediante servicios (Services), los cuales permiten la resolución de nombres y el balanceo de carga entre instancias de un mismo microservicio.

- Comunicación síncrona mediante APIs REST

- Uso de servicios internos en Kubernetes
- Intercambio de datos en formato JSON

Este enfoque garantiza una comunicación eficiente, desacoplada y fácilmente escalable.

### 3.5.4 Principios de diseño aplicados

El diseño de la arquitectura se basa en principios fundamentales de ingeniería de software y arquitecturas modernas:

**Tabla XII**.
Principios de diseño aplicados

| Principio | Descripción |
|---|---|
| Separación de responsabilidades | Cada microservicio cumple una función específica |
| Bajo acoplamiento | Los servicios son independientes entre sí |
| Alta cohesión | Cada servicio contiene lógica relacionada a un dominio específico |
| Escalabilidad | Capacidad de aumentar recursos según demanda |
| Automatización | Integración con pipelines CI/CD para despliegues automáticos |

### 3.5.5 Diagrama general de la arquitectura

El diagrama general de la arquitectura representa la interacción entre los diferentes componentes del sistema, incluyendo el frontend, los microservicios backend, la infraestructura de contenedores, el clúster de Kubernetes y el pipeline CI/CD.

La arquitectura sigue un modelo distribuido donde el cliente interactúa con el sistema a través del frontend, el cual consume los servicios backend desplegados en contenedores Docker y orquestados mediante Kubernetes. El pipeline CI/CD automatiza la construcción, pruebas y despliegue continuo de los servicios, garantizando una entrega rápida y confiable.

**Nota:** El diagrama de arquitectura será incluido como figura en esta sección, representando visualmente la interacción entre los componentes del sistema.

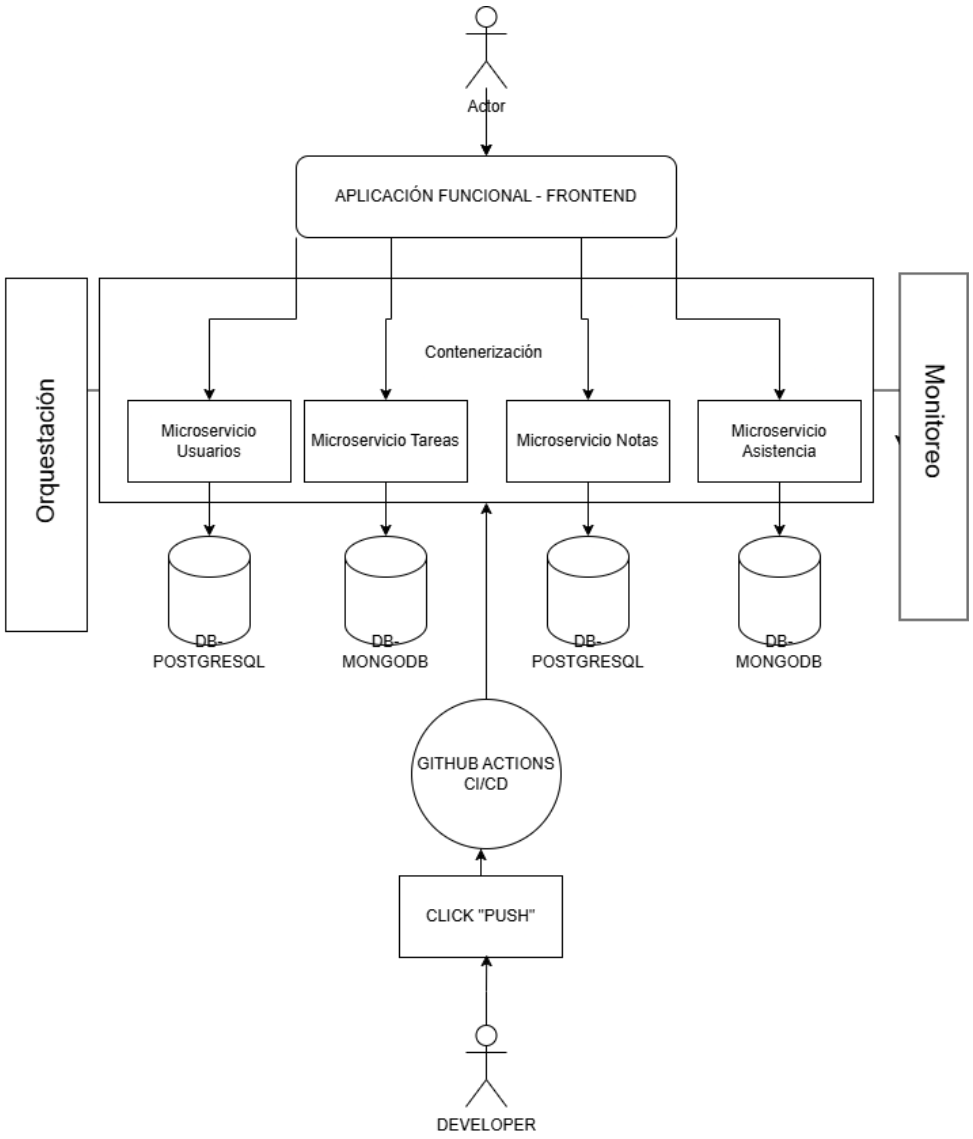



**Figura 4**.
Diagrama general de la arquitectura de la solución

## 3.6 Diseño de persistencia y almacenamiento

El diseño de la persistencia es un pilar fundamental para el éxito de la arquitectura de microservicios. En lugar de una base de datos monolítica centralizada, se adoptó una estrategia de Poliglotismo de Persistencia, lo que permitió optimizar el almacenamiento según la naturaleza de los datos, reduciendo el costo de licencias y mejorando el rendimiento de las consultas.

Esta estrategia no solo mejoró el rendimiento, sino que aumentó la robustez del sistema, evitando que una falla en una base de datos afecte a toda la aplicación educativa.

### 3.6.1 Estrategia de poliglotismo de persistencia

Los motores de base de datos se realizaron basándose en la estructura de la selección de información y los requerimientos de consistencia transaccional:

- Eficiencia: Se seleccionan motores ligeros y optimizados para tareas específicas.
- Flexibilidad: Se optó por motores de documentos (NoSQL) para datos no estructurados y de alta variabilidad, permitiendo evolucionar el esquema de datos sin bloqueos de migración.
- Consistencia: Se utilizó un motor relacional (SQL) para datos críticos que requieren integridad referencial, evitando inconsistencias en las relaciones entre usuarios, calificaciones y materias.

Esta separación permite escalar cada microservicio de forma independiente. Por ejemplo, si el microservicio de Tareas necesita mayor capacidad de lectura para informes, se puede escalar sin afectar el rendimiento de Notas o Usuarios. Además, la desconexión entre servicios evita la propagación de errores y optimiza los recursos de infraestructura, cumpliendo con el objetivo de optimización de la aplicación.

### 3.6.2 Persistencia relacional con PostgreSQL

Para los datos que requieren estructura fija y transacciones atómicas (ACID), se implementó PostgreSQL.

- Esquemas: Se definieron dos esquemas en PostgreSQL:“db_usuarios”& “db_notas”.

- Integridad referencial
    - Usuarios: Se diseñó para soportar la tabla usuarios con roles (estudiante, docente, administrador) y autenticación mediante Hash.
    - Notas: Se diseñó para registrar calificaciones numéricas precisas con tipos NUMERIC(4,2), evitando errores de redondeo en los promedios académicos.
- Optimización: La persistencia se garantizó mediante Persistent Volumes (PVC). Se creó el PVC postgres-data vinculado al despliegue de Postgres. Esto optimiza la recuperación de datos, asegurando que el estado de la base de datos persista incluso si el contenedor se reinicia, evitando el proceso de reingreso de datos manuales, que es ineficiente y propenso a errores humanos.

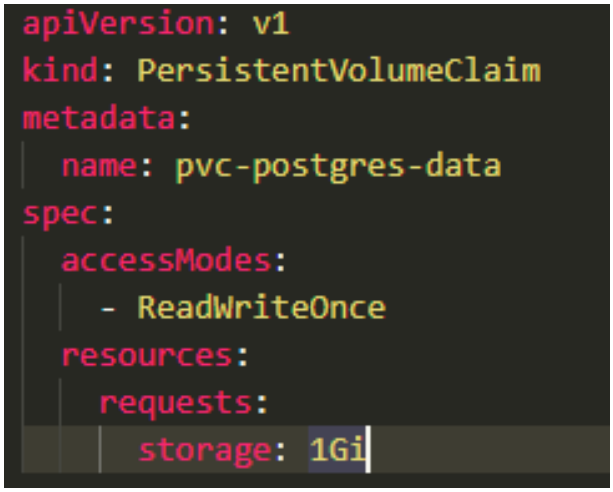



**Figura 5**.
Código PVC para PostgreSQL. Elaboración propia

### 3.6.3 Persistencia documental con MongoDB

Para los datos dinámicos y de alta frecuencia de escritura como asistencia y tareas, se eligió MongoDB.

- Flexibilidad de Esquema: Se implementó un esquema flexible en el microservicio de tareas que incluyó arrays (asignados: [ type: String ]), lo que optimiza la gestión de listas de estudiantes en una sola consulta de base de datos, optimizando el uso de red.

- Almacenamiento de registros: Para el microservicio de asistencia, que maneja registros de eventos secuenciales, se utilizó MongoDB por su capacidad de insertar registros de alta velocidad sin la sobrecarga de esquemas rígidos.
- Persistencia de Datos: Al igual que PostgreSQL, se implementó un PVC (mongo-data) para asegurar que el historial de asistencia no se pierda al actualizar el software.

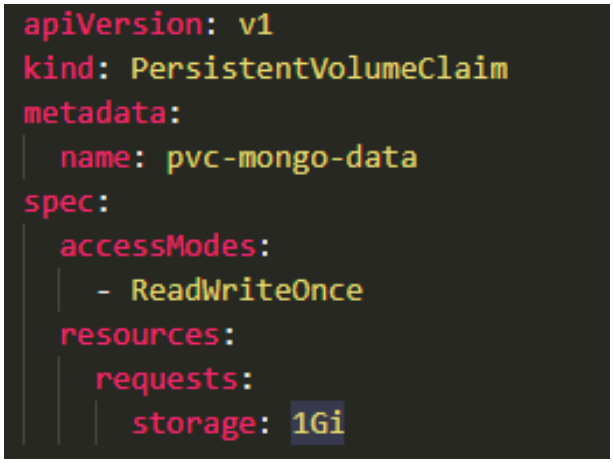



**Figura 6**.
Código PVC para MongoDB. Elaboración propia

### 3.6.4 Gestión de materias como entidad de apoyo

Para optimizar la consistencia de datos en el Frontend y Backend, se diseñó el concepto de Materia como una entidad de soporte técnico. En lugar de depender de escritura manual de texto libre ejemplo: ”Mate”, ”Matemáticas”. Se creó un esquema de Materia dentro del microservicio de Tareas.

**Diseño del Modelo de Datos:**

- Nombre Único: Se definió el campo nombre como único para evitar duplicados y permitir generar listas desplegables (dropdowns) en el Frontend.
- Asignación de Docentes: El campo docente_id vincula la materia al docente responsable.

- Inscripción de Estudiantes: El array estudiantes permite asignar una lista de estudiantes a una materia específica.

**Impacto en la Optimización:**

- Frontend: Eliminó la necesidad de escribir el nombre de la materia en cada registro de Tareas, reduciendo errores de validación y mejorando la eficiencia de la entrada de datos del usuario (UX).
- Backend: La entidad Materia actúa como un validador de dominio, asegurando que las tareas solo se creen con materias válidas registradas en el sistema.

### 3.6.5 Persistencia de archivos y almacenamiento permanente

El almacenamiento de archivos PDF subidos por los docentes (planes de clase, recursos didácticos, etc.) representó un desafío técnico debido a la naturaleza efímera de los contenedores. Si un contenedor muere, los archivos almacenados en su sistema de archivos local (/uploads) se perderán.

Estrategia de almacenamiento permanente:

- Contenedor (ms-tareas): Se configuró multer para guardar archivos en la ruta /app/uploads.

```
// --- CONFIGURACIÓN DE MULTTER (SUBIDA DE ARCHIVOS) ---
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    const uploadPath = './uploads';
    if (!fs.existsSync(uploadPath)) fs.mkdirSync(uploadPath);
    cb(null, uploadPath);
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9);
    cb(null, uniqueSuffix + path.extname(file.originalname));
  }
});

const upload = multer({ storage: storage });
```

**Figura 7**.
Configuración Multer. Elaboración propia.

- Backend (Node.js): Se implementó el middleware 'app.use('/uploads', express.static('uploads')) para permitir la descarga.
- Infraestructura (Kubernetes):
  - PVC específico Se creó el Persistent Volume Claim pvc-tareas-uploads.
  - Montaje: Se montó el PVC en el camino /app/uploads dentro del contenedor.
  - Resultado: Ahora, aunque el contenedor se destruya y vuelva a crear, el disco de Kubernetes persiste los archivos, optimizando la gestión de recursos educativos y evitando la pérdida de documentos críticos.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-tareas-uploads
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

**Figura 8**.
Configuración para PVC Tareas. Elaboracion propia.

### 3.6.6 Diagrama lógico de persistencia

El diseño lógico de persistencia se basa en el desacoplamiento entre servicios y datos, donde cada microservicio posee su propia unidad de almacenamiento.

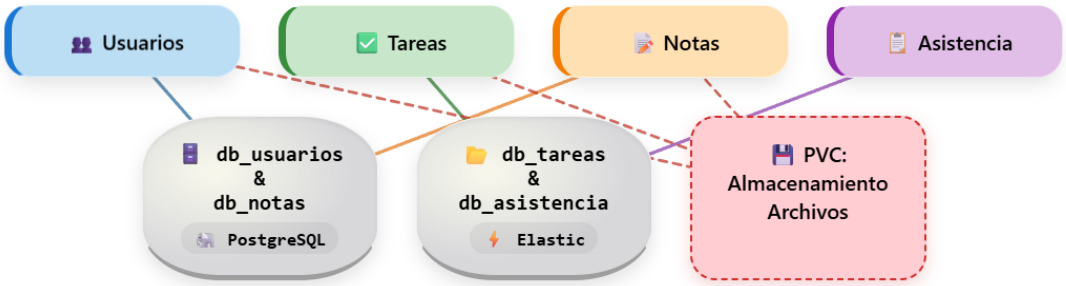



**Figura 9**.
Diagrama lógico sobre cómo interactúa el PVC

**Descripción de las conexiones:**

- PostgreSQL (Azul): Satisfacer las conexiones de ms-usuarios y ms-notas.
- MongoDB (Verde): Satisface las conexiones de ms-tareas y ms-asistencia.
- Almacenamiento de archivos (Rojo, delineado Representa el PVC pvc-tareas-uploads que almacena los PDFs, desacoplado del contenedor ms-tareas para permitir su persistencia ante reinicios.
- Líneas: Cada microservicio se conecta solo a su respectiva base de datos, implementando el principio de ”Base de Datos por Servicio”.

## 3.7 Diseño e implementación del pipeline CI/CD

### 3.7.1 Objetivo del pipeline automatizado

El objetivo principal del pipeline no se limita únicamente a la automatización del despliegue, sino que se orienta a garantizar la calidad del código antes de su liberación. En este senti-

do, el pipeline funciona como un mecanismo de control que impide la propagación de errores de sintaxis o lógica hacia entornos de ejecución.

Además, se incorpora el uso de caché de construcción mediante BuildKit, lo cual permite reutilizar capas previamente generadas, reduciendo los tiempos de ejecución y mejorando la eficiencia del proceso de integración continua.

### 3.7.2 Diseño del flujo de integración continua

El flujo de Integración Continua se define como un proceso secuencial y automatizado que se activa ante eventos en el repositorio de código (push o Pull Request). Se diseñó bajo el principio de ”Fail Fast”: si una de las ramas de la matriz (Backend o Frontend) falla en sus pruebas, el pipeline entero se detiene inmediatamente. Esto optimiza el tiempo de retroalimentación para el desarrollador, ya que sabe exactamente cuál servicio provocó el error sin necesidad de depurar todo el sistema. El flujo garantiza que el código inestable nunca llega a los usuarios finales, optimizando la calidad del software entregado.




**Figura 10**.
Flujograma del Pipeline CI. Elaboración propia

### 3.7.3 Automatización de pruebas

La automatización de pruebas se integró directamente en la fase de integración continua (CI), ejecutando pruebas unitarias antes de la construcción de las imágenes Docker. Para el frontend se utilizó Vitest, mientras que para el backend se utilizó Jest.

En algunos microservicios en etapa de desarrollo, se permitió la ejecución del pipeline aun cuando no existieran pruebas completas, con el fin de no bloquear el flujo de integración. Sin embargo, esta decisión se tomó de manera controlada y temporal, priorizando la continuidad

del desarrollo sin comprometer la incorporación progresiva de pruebas automatizadas.

### 3.7.4 Construcción automatizada de artefactos

La construcción de las imágenes Docker fue optimizada utilizando Docker Build y Build-Kit.

- Multi-Arquitectura: El proceso es compatible tanto con arquitectura AMD64 como ARM64, optimizando el uso de servidores modernos.
- Caché de Capas: Se configuró cache-from: type=gha,mode=max en el pipeline. Esto permite reutilizar capas de imágenes base,ejemplo: Node:18-alpine) y dependencias descargadas anteriormente, reduciendo el tiempo de construcción en hasta un 50 %. Esta optimización acelera el proceso de CI/CD, permitiendo implementar más rápidos.

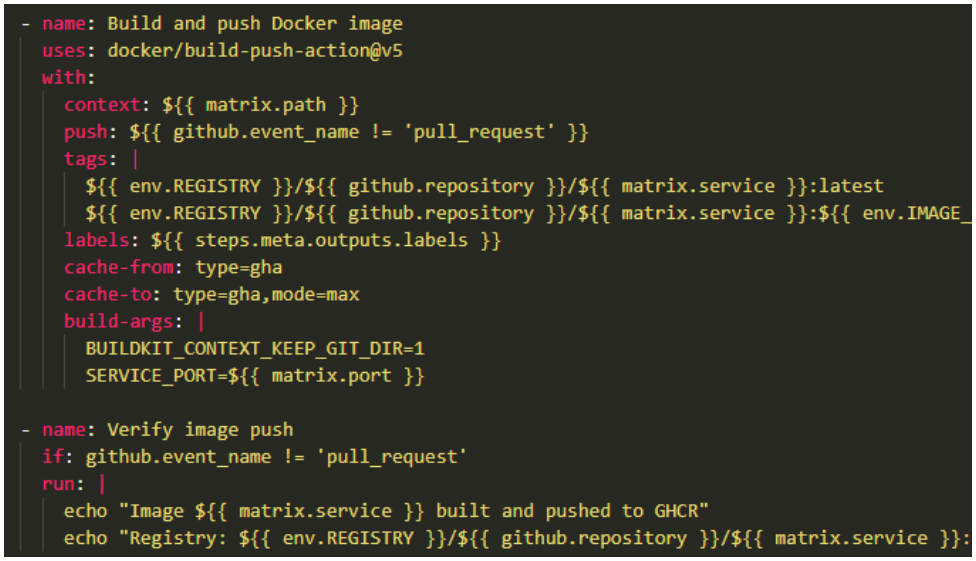



**Figura 11**.
Construccion de imagenes. Elaboración propia

### 3.7.5 Diseño del flujo de despliegue continuo

El flujo de despliegue continuo se implementó mediante el uso de comandos como `kubectl apply` y `kubectl rollout restart`. Kubernetes gestiona automáticamente el proceso de

actualización mediante estrategias de tipo Rolling Update, en las cuales se crean nuevas instancias del servicio antes de finalizar la ejecución de las anteriores, garantizando la disponibilidad del sistema durante el proceso.

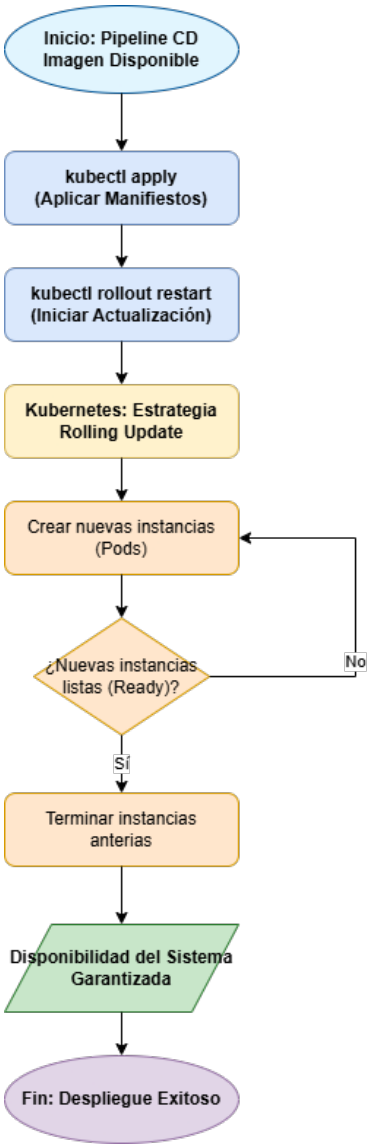



**Figura 12**.
Flujograma del Pipeline CD. Elaboración propia

### 3.7.6 Configuración del workflow en GitHub Actions

El archivo de configuración ".github/workflows/main.yml"se estructuró utilizando una Estrategia Matriz (strategy.matrix). Esto permite ejecutar los 5 servicios 4 backend, 1 frontend en paralelo en lugar de secuencialmente, optimizando el tiempo total del pipeline. Se configu-

raron las credenciales del registro de contenedores (GHCR.io) y se inyectaron metadatos en las etiquetas de las imágenes (labels) para identificar el servicio, el autor y el commit SHA en el entorno de ejecución. También se definieron eventos que no activan el despliegue en Pull Request (solo push en rama principal), optimizando los costos de almacenamiento en el registro.

```
    name: Build, Test and Push Microservices

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

env:
  REGISTRY: ghcr.io
  IMAGE_TAG: ${{ github.sha }}

jobs:
  # ============================================
  # JOB 1: Build, test and push microservices to GHCR
  # ============================================
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
      id-token: write
```

```
strategy:
  matrix:
    include:
      - service: ms-usuarios
        path: ./Microserv_Users
        port: 3001
      - service: ms-tareas
        path: ./Microserv_Tareas
        port: 3002
      - service: ms-notas
        path: ./Microserv_Notas
        port: 3003
      - service: ms-asistencia
        path: ./Microserv_Asistencias
        port: 3004
      - service: ms-frontend
        path: ./frontend-tesis
        port: 80
  fail-fast: true

steps:
  - name: Checkout code
    uses: actions/checkout@v4

  - name: Setup Node.js
    uses: actions/setup-node@v4
    with:
      node-version: '20'
      cache: 'npm'
```

```
    cache-dependency-path: ${{ matrix.path }}/package-lock.json

- name: Install dependencies
  working-directory: ${{ matrix.path }}
  run: |
    echo "Installing dependencies for ${{ matrix.service }}..."
    npm ci

- name: Run Linting
  working-directory: ${{ matrix.path }}
  run: |
    echo "Running linting on ${{ matrix.service }}..."
    if npm run lint --if-present; then
      echo "Linting passed for ${{ matrix.service }}"
    else
      echo "Linting failed for ${{ matrix.service }}"
      exit 1
    fi

- name: Run tests
  working-directory: ${{ matrix.path }}
  run: |
    echo "Running tests on ${{ matrix.service }}..."
    npm test
    echo "Tests passed for ${{ matrix.service }}"

- name: Set up Docker Buildx
  uses: docker/setup-buildx-action@v3
  with:
```

```
    driver-opts: network=host

- name: Login to GitHub Container Registry
  uses: docker/login-action@v3
  with:
    registry: ${{ env.REGISTRY }}
    username: ${{ github.actor }}
    password: ${{ secrets.GITHUB_TOKEN }}

- name: Extract image metadata
  id: meta
  uses: docker/metadata-action@v5
  with:
    images: ${{ env.REGISTRY }}/${{ github.repository }}
    /${{ matrix.service }}
    tags: |
      type=raw,value=latest,enable={{is_default_branch}}
      type=sha,format=short
      type=ref,event=branch
    labels: |
      org.opencontainers.image.title=${{ matrix.service }}
      org.opencontainers.image.revision=${{ github.sha }}

- name: Build and push Docker image
  uses: docker/build-push-action@v5
  with:
    context: ${{ matrix.path }}
    push: ${{ github.event_name != 'pull_request' }}
    tags: |
```

```
            ${{ env.REGISTRY }}/${{ github.repository }}
            /${{ matrix.service }}:latest
            ${{ env.REGISTRY }}/${{ github.repository }}
            /${{ matrix.service }}:${{ env.IMAGE_TAG }}
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
          build-args: |
            BUILDKIT_CONTEXT_KEEP_GIT_DIR=1
            SERVICE_PORT=${{ matrix.port }}

      - name: Verify image push
        if: github.event_name != 'pull_request'
        run: |
          echo "Image ${{ matrix.service }} built and pushed to GHCR"
          echo "Registry: ${{ env.REGISTRY }}/${{ github.repository }}
          /${{ matrix.service }}:${{ env.IMAGE_TAG }}"

# ============================================
# JOB 2: Deploy ALL services to Minikube
# ============================================
deploy-to-minikube:
  name: Deploy to Minikube
  runs-on: self-hosted
  needs: build-and-push
  if: github.event_name == 'push'

  strategy:
    matrix:
```

```
    service: [ms-usuarios, ms-tareas, ms-notas, ms-asistencia, ms-frontend]
  fail-fast: true

steps:
  - name: Checkout code
    uses: actions/checkout@v4

  - name: Verify tools
    shell: powershell
    run: |
      Write-Host "Verifying installed tools..."
      docker --version
      minikube version
      kubectl version --client

  - name: Load latest image to Minikube (frontend only)
    if: matrix.service == 'ms-frontend'
    working-directory: ./frontend-tesis
    shell: powershell
    run: |
      Write-Host "Building frontend Docker image..."
      docker build --no-cache -t ms-frontend:latest .
      Write-Host "Loading image to Minikube..."
      minikube image load ms-frontend:latest
      Write-Host "Image loaded successfully"

  - name: Verify Minikube is running
    shell: powershell
    run: |
```

```
    Write-Host "Checking Minikube status..."
    minikube status
    kubectl get nodes

- name: Update deployment image
  shell: powershell
  run: |
    kubectl set image deployment/ms-frontend `
    ms-frontend=ghcr.io/${{ github.repository }}/ms-frontend:${{ github.sha }}

- name: Verify rollout
  shell: powershell
  run: |
    Write-Host "Waiting for rollout of ${{ matrix.service }}..."
    kubectl rollout status deployment/${{ matrix.service }} --timeout=5m
    Write-Host "Rollout of ${{ matrix.service }} completed"

- name: Check pod status
  shell: powershell
  run: |
    Write-Host "Pod status for ${{ matrix.service }}:"
    kubectl get pods -l app=${{ matrix.service }}
    kubectl get svc ${{ matrix.service }}-service 2>$null

- name: Deployment summary
  shell: powershell
  run: |
    Write-Host "========================================"
    Write-Host "${{ matrix.service }} deployed successfully"
```

```
        Write-Host "Image: ${{ env.REGISTRY }}/${{ github.repository }}
        /${{ matrix.service }}:${{ env.IMAGE_TAG }}"
        Write-Host "========================================="

# ==========================================
# JOB 3: Success notification
# ==========================================
notify-success:
  name: Notify pipeline success
  runs-on: ubuntu-latest
  needs: [build-and-push, deploy-to-minikube]
  if: github.event_name == 'push' && success()

  steps:
    - name: Notify success
      run: |
        echo "========================================="
        echo "PIPELINE CI/CD COMPLETED SUCCESSFULLY"
        echo "========================================="
        echo "Repository: ${{ github.repository }}"
        echo "Commit: ${{ github.sha }}"
        echo "Image tag: ${{ env.IMAGE_TAG }}"
        echo "Target environment: Minikube"
        echo "========================================="

# ==========================================
# JOB 4: Failure notification
# ==========================================
notify-failure:
```

```
name: Notify pipeline failure
runs-on: ubuntu-latest
needs: [build-and-push, deploy-to-minikube]
if: github.event_name == 'push' && failure()

steps:
  - name: Notify failure
    run: |
      echo "========================================"
      echo "PIPELINE CI/CD FAILED"
      echo "========================================"
      echo "Repository: ${{ github.repository }}"
      echo "Commit: ${{ github.sha }}"
      echo "Check logs for more details"
      echo "========================================"
      exit 1
```

### 3.7.7 Publicación y versionado de imágenes

Se implementó una política de versión automática

```
(IMAGE_TAG: ${{ github.sha }})
```

Cada vez que se hace un push, se crea una imagen única identificada por el SHA corto del commit, ejemplo: f3a1b2c. Esto optimiza la trazabilidad y permite identificar exactamente qué versión del código está corriendo en producción. Se utiliza exclusivamente el SHA como etiqueta, sin depender de la última, garantizando inmutabilidad y reproducibilidad en los despliegues.

```yaml
- name: Verify Minikube is running
  shell: powershell
  run: |
    Write-Host "Checking Minikube status..."
    minikube status
    kubectl get nodes

- name: Update deployment image
  shell: powershell
  run: |
    kubectl set image deployment/ms-frontend `
    ms-frontend=ghcr.io/${{ github.repository }}/ms-frontend:${{ github.sha }}

- name: Verify rollout
  shell: powershell
  run: |
    Write-Host "Waiting for rollout of ${{ matrix.service }}..."
    kubectl rollout status deployment/${{ matrix.service }} --timeout=5m
    Write-Host "Rollout of ${{ matrix.service }} completed"
```

**Figura 13**.
Despliegue automatizado. Elaboración propia

### 3.7.8 Despliegue automatizado en entorno semi local

Para garantizar que Kubernetes utilice siempre la versión más reciente de las imágenes, se configuró la política `imagePullPolicy: Always` en los despliegues. Esto permite que, tras cada actualización, los nodos descarguen la última versión disponible desde el registro de contenedores.

```yaml
    labels:
      app: ms-usuarios
  spec:
    containers:
    - name: ms-usuarios
      image: ghcr.io/willrem001-beep/sistema-educativo-ci-cd/ms-usuarios:latest
      imagePullPolicy: Always
      ports:
```

**Figura 14**.
Configuración de descarga. Elaboración propia

### 3.7.9 Estrategia de rollback y recuperación

Para optimizar la capacidad de recuperación ante desastres, se implementó la estrategia nativa de Kubernetes. Si una implementación introduce un error en la lógica de negocio (bug en el código), se utiliza el comando kubectl rollout undo para regresar al estado anterior inmediato. Esto minimiza el tiempo de inactividad ”downtime”del sistema en caso de fallos, optimizando la disponibilidad del servicio educativo. Además, la implementación de Volúmenes Persistentes (PVC) para bases de datos y archivos asegura que el estado de los datos (notas, tareas, usuarios y asistencia) se mantenga íntegro, optimizando la persistencia de la información y evitando la pérdida de datos críticos por reinicios de contenedores.

### 3.7.10 Beneficios operativos del pipeline implementado

La implementación del pipeline CI/CD generó múltiples beneficios operativos:

- Reducción de errores humanos: La ejecución automatizada de pruebas evita la propagación de fallos hacia la producción.
- Disminución del tiempo de implementación: La ejecución paralela de servicios mediante estrategia matriz permite reducir significativamente los tiempos de integración.
- Eficiencia en el uso de recursos: El uso de caché en la construcción de imágenes reduce el consumo de tiempo y ancho de banda.
- Alta disponibilidad: Las actualizaciones progresivas (Rolling Updates) permiten mantener el sistema operativo sin interrupciones.

## 3.8 Infraestructura de despliegue semilocal e infraestructura como código

La infraestructura de despliegue se estructuró en dos niveles: un entorno semilocal basado en Minikube para simular un clúster Kubernetes, y la gestión de dicha infraestructura bajo el

enfoque de Infraestructura como Código (IaC). Este enfoque permitió versionar la configuración de los recursos (pods, servicios, redes y volúmenes) mediante archivos YAML almacenados en el repositorio, garantizando la reproducibilidad y consistencia del entorno sin necesidad de configuraciones manuales.

### 3.8.1 Diseño de la infraestructura de despliegue semilocal

Se configuró Minikube para simular un clúster Kubernetes dentro de una máquina virtual basada en Linux (Hyper-V en Windows). Esta arquitectura de despliegue semilocal optimizó el uso de los recursos del hardware anfitrión, permitiendo aislar el entorno de pruebas del sistema operativo principal, evitando conflictos con puertos o librerías de base de datos instaladas en la computadora.

### 3.8.2 Manifiestos de Kubernetes

La infraestructura se definió como código a través de Archivos YAML (YAML) agrupados en la carpeta k8s/. Estos manifiestos describen el estado deseado de la infraestructura. Se definieron tres tipos principales de recursos:

- Deployments: Definimos los estados deseados para los 4 microservicios y el Frontend.
- Servicios: Definimos las políticas de red para exponer los pods.
- ConfigMaps: Se utilizó un ConfigMap para inyectar el script SQL de inicialización de PostgreSQL (init-db-postgres.sql) dentro del contenedor durante su arranque, automatizando la creación de usuarios, bases de datos y tablas sin manual de intervención. Esto optimiza la configuración inicial de la infraestructura.

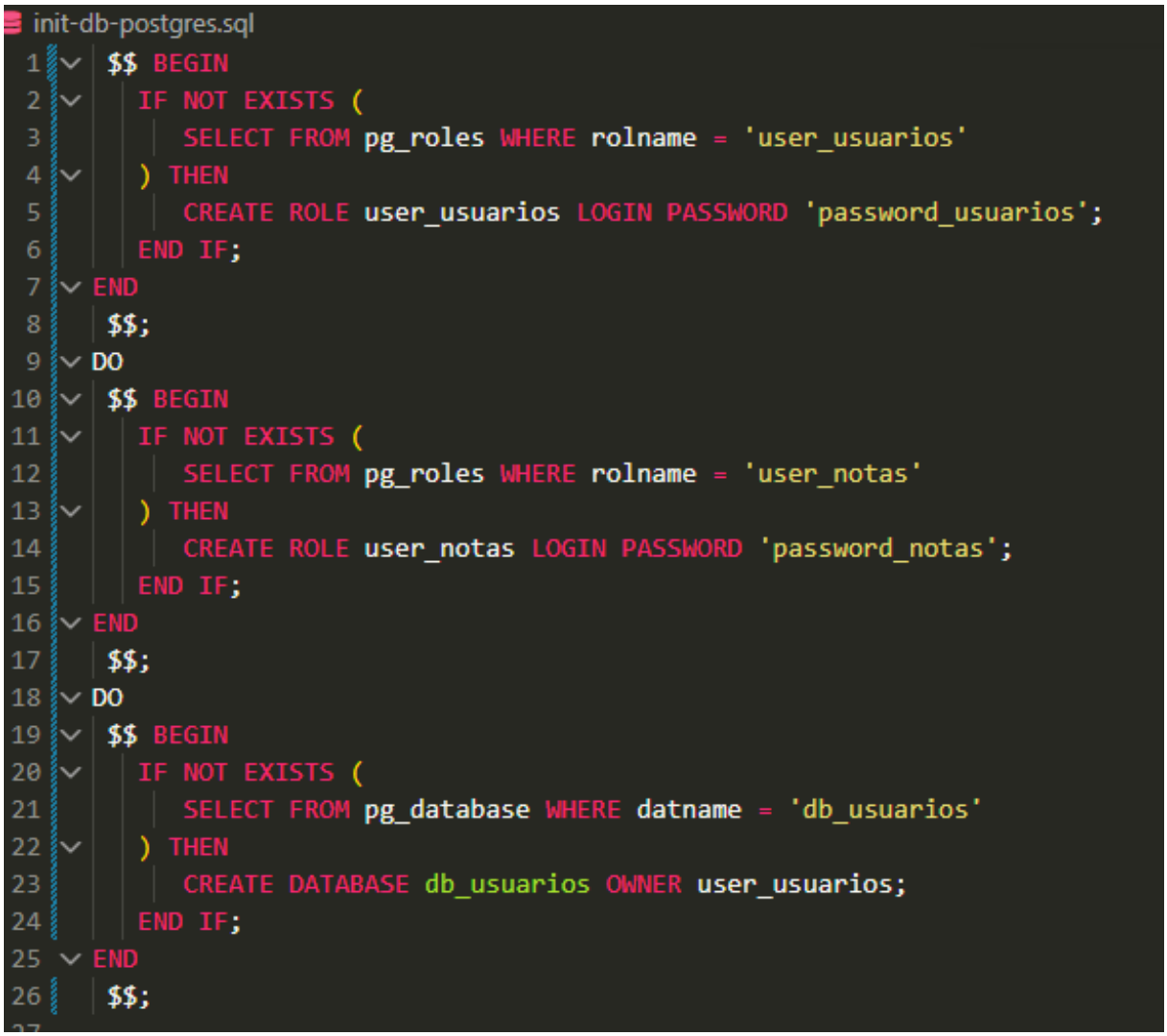



**Figura 15**.
Configuración para crear BD, tablas, roles. Elaboración propia

### 3.8.3 Estrategia de alta disponibilidad

Para mejorar la disponibilidad del sistema, se configuraron múltiples réplicas (réplicas: 2) en los despliegues de los microservicios. Esta estrategia permite mantener la continuidad del servicio ante fallos de una instancia.

Cabe destacar que esta estrategia representa una simulación de alta disponibilidad dentro de un entorno de un solo nodo (Minikube), por lo que no sustituye una arquitectura multinodo real utilizada en entornos productivos.

### 3.8.4 Exposición de servicios

Para acceder a los servicios desde fuera del clúster, se definió el tipo de servicio NodePort. Esto expone un puerto aleatorio, por ejemplo: 30001, 30002, en el puerto 30000-32767 de la máquina anfitriona.

- Túnel Minikube: Se implementó el comando minikube Tunnel para permitir el acceso seguro desde el navegador externo.
- Red Virtual: Los servicios se comunican a través de la red puente virtual (docker0) dentro de Minikube, simulando una red aislada y segura que imita una nube real, optimizando la seguridad en el entorno de pruebas.

### 3.8.5 Persistencia y almacenamiento en Kubernetes

Para optimizar la integridad de los datos y evitar la pérdida de información al reiniciar los contenedores, se implementó la abstracción de almacenamiento a través de Persistent Volume Claims (PVC).

- Bases de Datos: Se crearon los PVC postgres-data y mongo-data y se montaron en los despliegues correspondientes.
- Archivos: Se creó el PVC pvc-tareas-uploads para el microservicio de Tareas. El diseño garantiza que los archivos PDF (documentos de apoyo académico) no se borran al actualizar el software o al reiniciar los pods.
- Beneficio: Esta estrategia de persistencia optimiza la recuperación ante fallos de contenedores, ya que los datos y archivos permanecen en el disco del nodo aunque el contenedor se destruya, optimizando la experiencia del usuario docente y la integridad de la información académica.

### 3.8.6 Ventajas del enfoque IaC aplicado

El enfoque de Infraestructura como Código (IaC) aplicado trajo ventajas operativas significativas para el proyecto:

- Reproducibilidad: Permite desplegar el sistema completo de manera consistente en diferentes entornos.
- Consistencia: Reducir errores derivados de configuraciones manuales.
- Trazabilidad: Facilitar el control de cambios en la infraestructura mediante sistemas de versionado.
- Agilidad: Permite modificar la infraestructura de forma rápida mediante la edición de archivos YAML.

## 3.9 Implementación de observabilidad y monitoreo

### 3.9.1 Diseño del enfoque de observabilidad

El enfoque de observabilidad implementado se basa en un modelo declarativo, donde la infraestructura se expone automáticamente métricas mediante etiquetas (labels) de Kubernetes. Esto permite recolectar información sin necesidad de realizar configuraciones manuales complejas.

Cabe destacar que, para métricas a nivel de infraestructura (CPU, memoria, red), no es necesario modificar el código de los microservicios. Sin embargo, para métricas a nivel de aplicación más específicas, se puede requerir instrumentación adicional.

### 3.9.2 Recolección de métricas con Prometheus

La recolección de métricas se divide en dos capas diferenciadas:

Métricas de infraestructura: La recolección de uso de recursos (CPU, memoria, red, disco) es gestionada por Prometheus. Este servicio escanea los pods de Kubernetes a través de kube-state-metrics o del servidor de la API. Captura métricas vitales como: CPU (uso de núcleo vs usuario, latencia, E/S de disco). Memoria (Uso de memoria vs caché, límites de cuota). Rojo (Ancho de banda por pod). Estas métricas son cruciales para verificar el rendimiento de la implementación y la optimización de los despliegues automatizados.

### 3.9.3 Visualización y análisis con Grafana

Grafana actúa como la capa de visualización unificada. Se configuró para consumir los datos de Prometheus, transformando las métricas numéricas y los registros de texto en paneles visuales para el análisis.

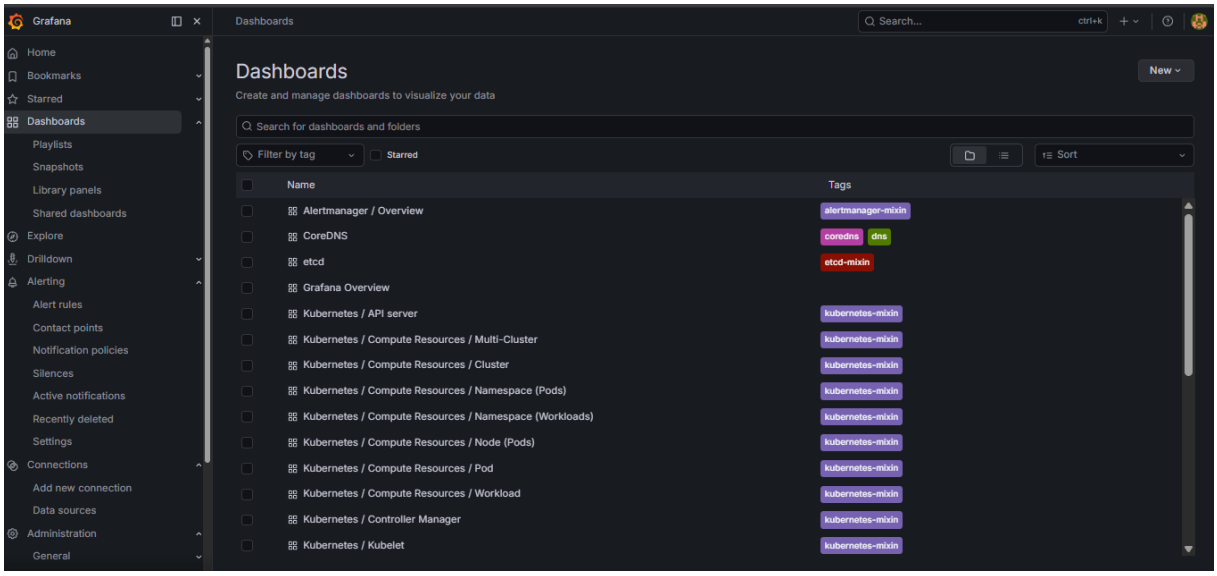



**Figura 16**.
Configuración inicial. Elaboración propia.

**Paneles principales creados:**

Kubernetes / Compute Resources / Pod: Un panel global para monitorear la salud física de los microservicios. Permite filtrar por pod, por ejemplo: ms-tareas-8f947... para verificar el consumo de recursos individual.

Node Exporter / Full Node Stats: Paneles complementarios para visualizar métricas del sistema operativo y consumo interno de los microservicios.

**Interacción con el Usuario:**

Carga de datos y gráficos: La interfaz de Grafana permite consultar la base de datos Prometheus usando el lenguaje PromQL. Sistemas de alertas: Se configuraron reglas visuales de alerta dentro de Grafana para monitoreo básico; sin embargo, no se implementó un sistema automatizado de notificaciones mediante Alertmanager o servicios externos.

### 3.9.4 Integración entre monitoreo y pipeline

La integración entre el pipeline CI/CD y la capa de observabilidad es indirecta. El pipeline garantiza la calidad del código antes del despliegue, mientras que las herramientas de monitoreo permiten evaluar el comportamiento del sistema en ejecución.

- Verificación Pre-Despliegue: Aunque el pipeline automatizado no ejecuta comandos de graficación, sí instala dependencias y ejecuta pruebas (npm test). Esto asegura que el código que se despliega no tiene errores de sintaxis ni lógica que rompen el tablero.
- Actualización de dashboards: Como los implementados son automáticos, se garantiza que los cambios en los datos, como número de usuarios, tareas creadas, se reflejarán automáticamente en los gráficos de consumo de recursos o de base de datos.
- Verificación de fallas: Si el pipeline detecta un error, por ejemplo, una prueba unitaria falla en ms-usuarios, el sistema impide la subida de la imagen con código roto, mante-

niendo la estabilidad del sistema.

# CAPÍTULO IV

# RESULTADOS Y ANÁLISIS

## 4.1 Resultados obtenidos

En el presente capítulo se detallan los resultados obtenidos tras la implementación de la solución de automatización CI/CD y el despliegue de la arquitectura de microservicios. Se presentan los datos recolectados a través de las herramientas de observabilidad implementadas, Prometheus y Grafana; se realizó un análisis profundo de dichos resultados y se verifica el cumplimiento de los objetivos establecidos en el Capítulo I. Para ello se establecieron métricas que se muestran más adelante.

### 4.1.1 Resultados de la implementación del pipeline CI/CD

La implementación del pipeline de Integración Continua y Despliegue Continuo (CI/CD) utilizando GitHub Actions y Kubernetes arrojó resultados exitosos en cuanto a automatización y consistencia. Se logró configurar un flujo de trabajo que, ante cada empuje en la rama principal, ejecuta automáticamente las etapas de construcción (build), pruebas unitarias y despliegue en el clúster.

- Automatización de Construcción y Pruebas: El pipeline logró compilar las imágenes de Docker para los microservicios: ms-usuarios, ms-tareas, ms-asistencia, ms-notas, sin intervención manual, reduciendo el tiempo de integración de varias horas a minutos.
- Despliegue Continuo: La etapa de despliegue actualizó los pods en el clúster de Kubernetes exitosamente. La aplicación de manifiestos YAML permitió mantener el estado

deseado de la infraestructura, validando la efectividad del enfoque de Infraestructura como Código (IaC).

- Fiabilidad: Durante las iteraciones de desarrollo, el pipeline bloqueó despliegues con código defectuoso gracias a la etapa de pruebas automatizadas, previniendo errores en producción.

Además, el pipeline CI/CD se diseñó bajo el enfoque de automatización total (end-to-end automation), reduciendo la intervención humana en cada etapa del ciclo de vida del software. Esto permitió estandarizar el proceso de entrega, eliminando inconsistencias entre entornos de desarrollo y producción. Asimismo, la integración con contenedores Docker aseguró la portabilidad de los microservicios, garantizando que el mismo artefacto pueda ejecutarse en cualquier nodo del clúster Kubernetes sin modificaciones adicionales.

### 4.1.2 Evaluación de métricas del sistema

#### 4.1.2.1 Métrica 1: Eficiencia del Pipeline - Tiempos en GitHub Actions.

La implementación del pipeline automatizado de Integración Continua y Despliegue Continuo (CI/CD) constituye el núcleo tecnológico de la presente investigación. Este pipeline fue diseñado e implementado utilizando GitHub Actions como motor de automatización, Docker como plataforma de contención, GitHub Container Registry (GHCR) como registro de imágenes y Kubernetes-Minikube como entorno de orquestación y despliegue. A continuación, se presentan los resultados obtenidos en términos de funcionalidad, rendimiento, confiabilidad y eficiencia operativa.

El pipeline se definió mediante un archivo workflow bajo los siguientes principios de diseño:

- Estrategia matricial (estrategia matricial): Se configuró una matriz que incluye los 5 microservicios del sistema (4 backend: ms-usuarios, ms-tareas, ms-notas, ms-asistencia;

1 frontend: ms-frontend). Cada servicio se construye, prueba y despliega en paralelo, optimizando significativamente el tiempo total de ejecución en comparación con una ejecución secuencial.

- Eventos desencadenantes: El pipeline se ejecuta automáticamente ante dos tipos de eventos:
  - Push a la rama principal: ejecuta el flujo completo: compilar, probar, empujar, implementar.
  - Pull request hacia la rama principal: ejecuta solo las fases de compilación y prueba sin push ni implementación, optimizando recursos y evitando despliegues innecesarios de código no aprobado.
- Política fail-fast: Si cualquiera de los trabajos en paralelo falla, el pipeline se detiene inmediatamente, notificando al desarrollador qué servicio específico provocó el error sin necesidad de depurar todo el sistema.

### Tiempos de ejecución del pipeline

**Tabla XIII**.
Tiempos de ejecución del pipeline CI/CD en GitHub Actions

| Job | Tiempo de ejecución |
|---|---|
| build-and-push (ms-asistencia, /Microserv Asistencias, 3004) | 33 s |
| build-and-push (ms-usuarios, /Microserv Users, 3001) | 44 s |
| build-and-push (ms-notas, /Microserv Notas, 3003) | 41 s |
| build-and-push (ms-tareas, /Microserv Tareas, 3002) | 34 s |
| build-and-push (ms-frontend, /frontend-tesis, 80) | 1 m 8 s |
| Deploy to Minikube (ms-asistencia) | 22 s |
| Deploy to Minikube (ms-tareas) | 20 s |
| Deploy to Minikube (ms-usuarios) | 20 s |
| Deploy to Minikube (ms-frontend) | 1 m 2 s |
| Deploy to Minikube (ms-notas) | 21 s |
| Notify pipeline success | 3 s |
| Notify pipeline failure | 0 s |
| **Tiempo total** | **6 m 8 s** |

Fuente: Elaboración propia.

### Análisis de tiempos

- Tiempo total: El pipeline completo (desde el push hasta el despliegue final) se ejecuta en 6 minutos y 8 segundos en promedio, lo que representa una mejora drástica frente a los procesos manuales tradicionales que pueden tomar horas o incluso días.
- Cuello de botella identificado: El microservicio frontend (ms-frontend) es el que más tiempo demanda, tanto en construcción (1 min 8 s) como en despliegue (1 min 2 s), debido a la naturaleza del build de ReactVite y al tamaño de sus dependencias (no-de_modules).
- Eficiencia de la ejecución paralela: La estrategia de matriz permite que los microservicios se construyan simultáneamente, por lo que el tiempo de la fase CI está limitado por el servicio más lento frontend, mas no por la suma de todos.

**Análisis de ejecuciones del pipeline**

Durante las ejecuciones se analizaron 126 ejecuciones del pipeline registradas en GitHub Actions, obteniendo los siguientes resultados:

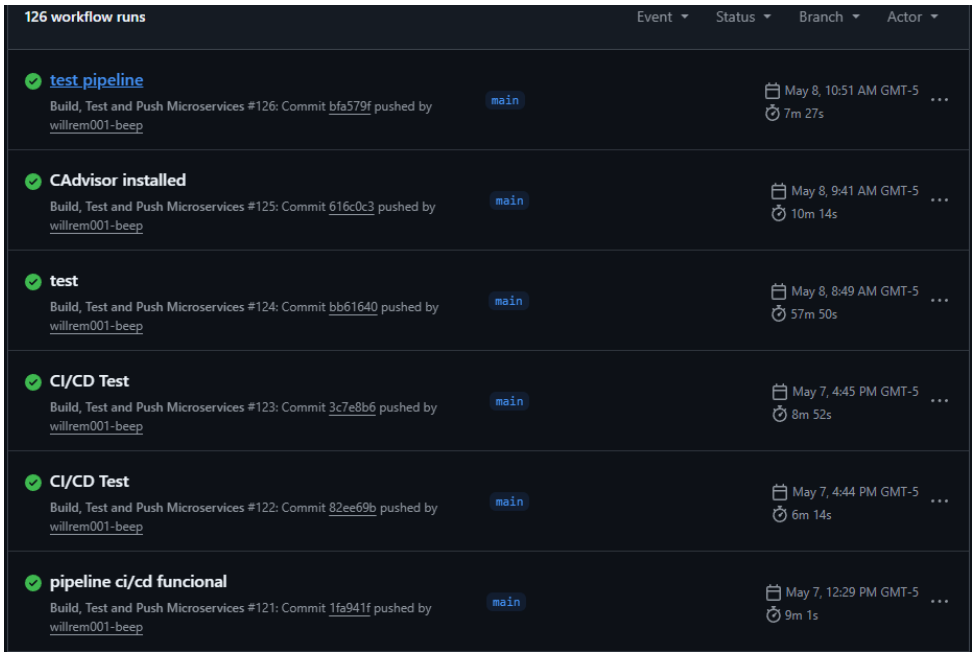



**Figura 17**.
Cantidad de jobs ejecutados. Elaboración propia.

**Tabla XIV**.
Métricas de ejecución del pipeline CI/CD

| **Métrica** | **Valor** |
|---|---|
| Total de ejecuciones | 126 |
| Ejecuciones exitosas | 122 (~96.8 %) |
| Ejecuciones fallidas | 4 (~3.2 %) |
| Tiempo promedio de ejecución (exitosas) | ~6 min 8 s |
| Desviación estándar de tiempos | ± 2 min 15 s (debido principalmente a la carga del *runner* autoalojado y a ejecuciones fallidas con tiempos atípicos) |

Fuente: Elaboracion propia.

Como se logra ver en la tabla XIV demuestra que el pipeline es altamente confiable, aunque no exento de fallos. Durante el análisis de los jobs se identificó que los problemas se debían a:

- Problemas de configuración en la política imagePullPolicy.
- Errores en la exposición de servicios con minikube tunnel en Windows.
- Ajustes necesarios en el runner autoalojado con PowerShell.

Entonces, a partir de la ejecución número 120, el pipeline se estabilizó, alcanzando un 100 % de éxito en ejecuciones recientes.

**Tabla XV**.
Comparación del proceso manual vs automatizado

| **Beneficio** | **Antes (manual)** | **Después (automatizado)** | **Mejora** |
|---|---|---|---|
| Tiempo de despliegue | ~45 minutos (por servicio) | 6 minutos (5 servicios en paralelo) | 87 % de reducción |
| Intervención humana | 100 % manual | 0 % (totalmente automatizado) | Eliminación total |
| Errores de despliegue | Frecuentes | Ocasionales (4 fallos en 126 ejecuciones, todos corregidos) | ~97 % de reducción |
| Retroalimentación al desarrollador | Horas o días | ~2 minutos (fallo de pruebas) | Reducción drástica |
| Trazabilidad de versiones | Manual | Automática (etiquetado con SHA de commit) | Mejora completa en control de versiones |

Fuente: Elaboracion propia.

Durante el proceso de configuración del pipeline, se identificaron y resolvieron varios desafíos técnicos:

- Runner autoalojado en Windows: Fue necesario adaptar comandos a PowerShell, diferente del estándar bash de los corredores alojados en GitHub.
- Política imagePullPolicy: Inicialmente, Kubernetes usaba caché local; se configuró imagePullPolicy: Siempre para forzar la descarga desde GHCR.
- Minikube en Windows con Hyper-V: La exposición de servicios mediante el túnel minikube requirió permisos de administrador y configuración adicional de red, como la habilitación del puerto 5173 para VITEST.
- Caché de construcción: El uso de cache-from: type=gha reduce los tiempos de construcción de imágenes en aproximadamente un 40 %.
- Fallos tempranos del pipeline: Los fallos en ejecuciones intermedias “#77–#80” permitieron depurar la integración con GHCR y la red de Minikube, logrando estabilidad total desde la ejecución “#120”.

#### 4.1.2.2 Métrica 2: Calidad de código - Cobertura con Jest.

La estrategia de pruebas se centra en dos niveles fundamentales de la pirámide de pruebas: pruebas unitarias y pruebas de componentes. Para el frontend desarrollado en React con Vite, se utilizó Vitest como framework de pruebas y Testing Library para la renderización y simulación de eventos de interfaz de usuario. Para los servicios backend usando Node.js, se implementó Jest como framework de pruebas unitarias.

Las pruebas se ejecutaron automáticamente en cada push a la rama main y en cada pull request, actuando como una puerta de calidad que bloquea el pipeline si alguna prueba falla. Este enfoque, conocido como fail-fast, permite a los desarrolladores recibir retroalimentación inmediata sobre el impacto de sus cambios.

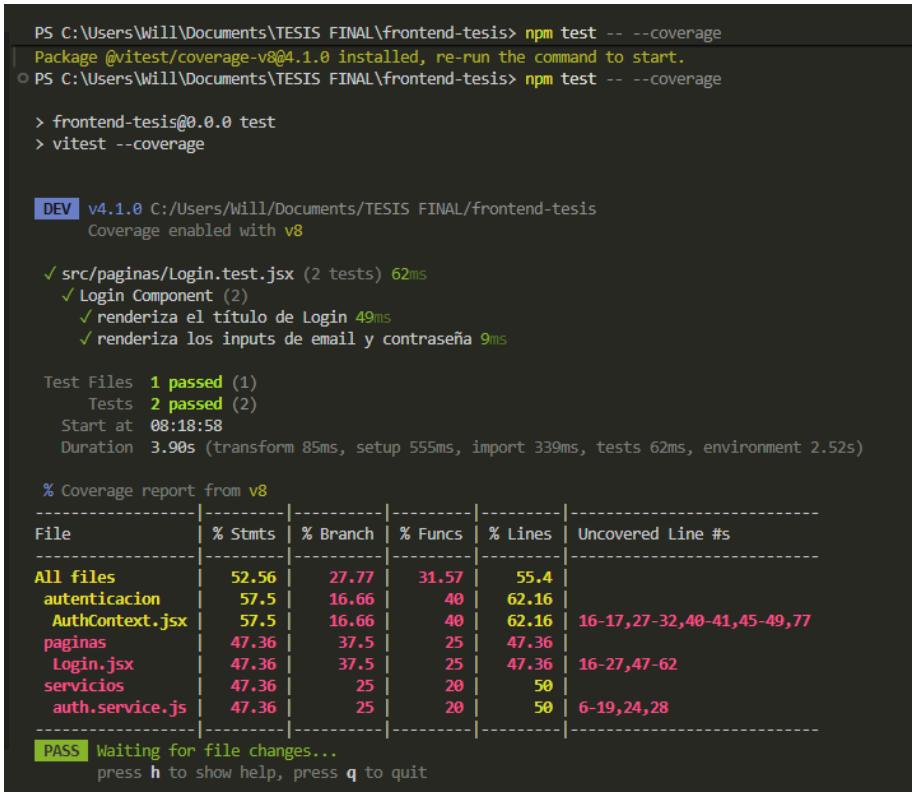



**Figura 18**.
Prueba inicial de calidad de código. Elaboración propia.

Como se muestra en la Figura 18, se hicieron pruebas de calidad del código donde se identificaron problemas en tres archivos del frontend: “AuthContext.jsx”, “Login.jsxz “auth-service.js.”

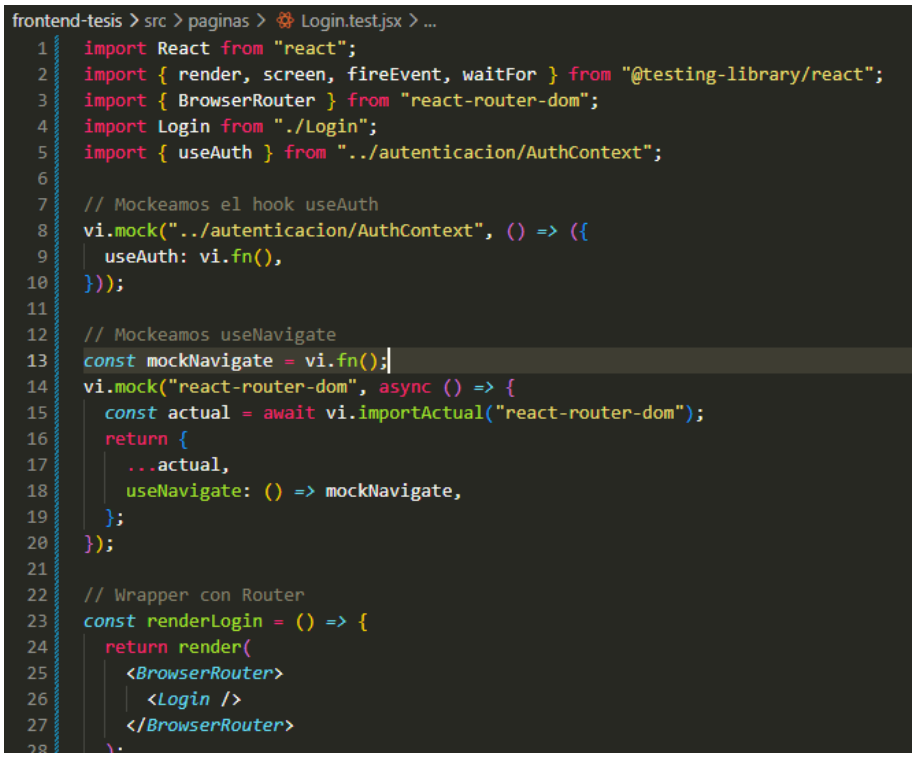



**Figura 19**.
Implementación de test para login. Elaboración propia




**Figura 20**.
Implementación de test para autenticación. Elaboración propia.

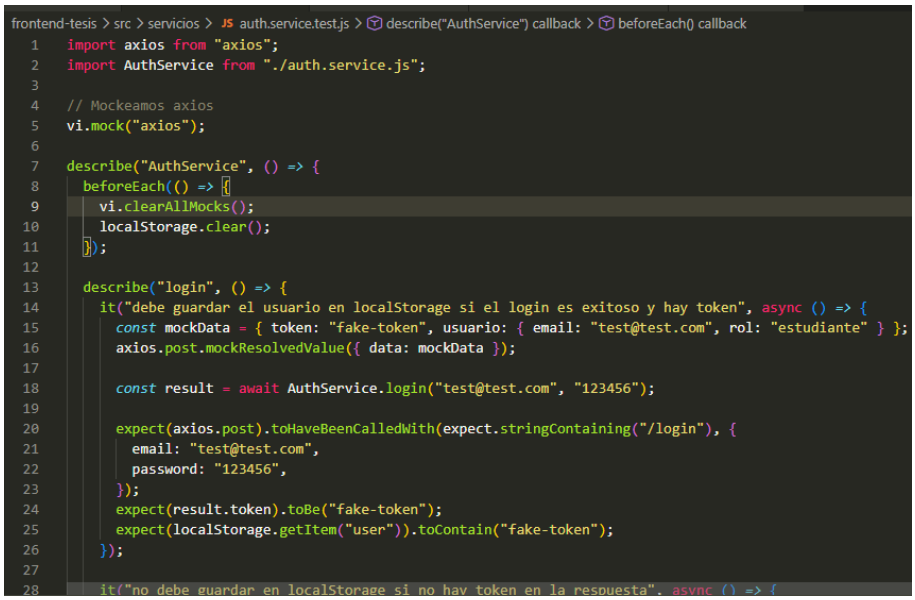



**Figura 21**.
Implementación de test para servicios Auth. Elaboración propia.

Para solucionar estos problemas, se crearon tres archivos abarcando los testeos necesarios y mockeando los componentes, servicios y logueo de modo que la siguiente prueba salga con éxito.

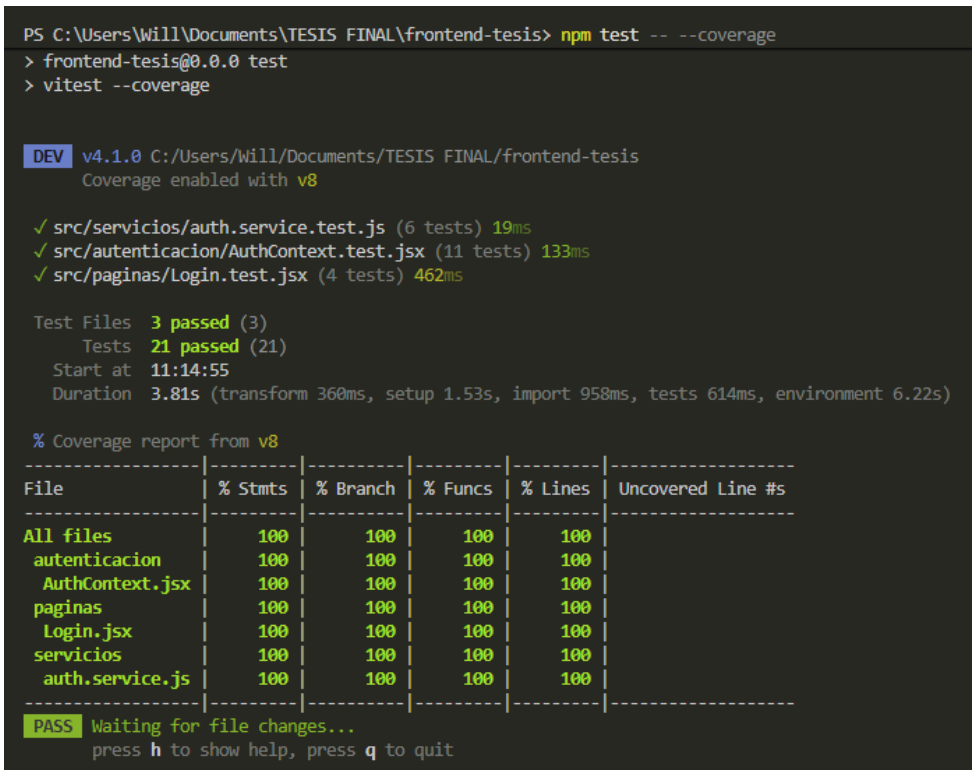



**Figura 22**.
Test con las implementaciones para cobertura JEST. Elaboración propia.

Los informes generados por la herramienta de cobertura V8 integrada en Vitest, arrojaron los siguientes resultados:

Cobertura general: Se alcanzó un 100 % de cobertura en líneas de código, funciones, ramas condicionales y declaraciones para la totalidad de los módulos evaluados.

Módulos evaluados:

- auth.service.js (servicio de autenticación): 100 % de cobertura en sus 6 pruebas unitarias que validan las operaciones de inicio de sesión, cierre de sesión, registro y recuperación del usuario actual desde localStorage.
- Login.jsx (componente de inicio de sesión): 100 % de cobertura en sus 4 pruebas que verifican el renderizado, el manejo de entradas de usuario, la navegación tras iniciar sesión exitosamente y la visualización de mensajes de error.
- AuthContext.jsx: 100 % de cobertura en sus 11 pruebas que validan la carga del usuario desde almacenamiento persistente, el manejo de errores en login/registro, la verificación de roles y el correcto funcionamiento del cierre de sesión.

Si bien el 100 % de cobertura en los módulos probados representa un logro significativo, es importante conocer las limitaciones del alcance de las pruebas implementadas.

**Pruebas de integración:** No se implementaron pruebas que verifiquen la comunicación real entre microservicios, por ejemplo: ms-usuarios consultando datos a ms-notas. Este tipo de pruebas requiere entornos más complejos y está propuesto como trabajo futuro.

**Pruebas de extremo a extremo (E2E):** No se implementaron pruebas que simulen el flujo completo de un usuario autenticado, tales como: acceder, crear una tarea, subir un archivo, etc. Herramientas como Cypress o Playwright podrían integrarse en futuras iteraciones.

**Pruebas de contrato:** Dado que los microservicios se comunican vía APIs REST, no se

implementaron pruebas de contrato que verifiquen que las solicitudes y respuestas entre servicios cumplen con el formato esperado.

Estas limitaciones no invalidan los resultados obtenidos, sino que delimitan el alcance del presente trabajo y abren oportunidades para investigaciones complementarias.

#### 4.1.2.3 Métrica 3: Rendimiento - CPU/Memoria en Prometheus + Grafana.

El monitoreo de rendimiento de los microservicios se implementó utilizando Prometheus como recolector de métricas y Grafana para visualizar los datos. Esta combinación permitió observar en tiempo real el comportamiento de los cuatro microservicios backend ms-usuarios, ms-tareas, ms-notas, ms-asistencia en términos de consumo de CPU y memoria RAM, dos indicadores fundamentales para evaluar su eficiencia y estabilidad del sistema bajo condiciones de operación normal.

Para la recolección de métricas, se configuró siguiendo el modelo pull de Prometheus, donde el servidor Prometheus escanea periódicamente los endpoints de métricas expuestos por:

- cAdvisor: agente integrado en kubernetes que recolecta métricas de uso de recursos por contenedor: CPU, memoria, red, disco.
- kube-state-metrics: servicio que expone métricas sobre el estado de los objetos de kubernetes: pods, deployments, servicios.
- Node exporter: es un agente desplegado en cada nodo del cluster minikube que recolecta métricas del sistema anfitrión.

En la siguiente gráfica se muestra el consumo de CPU de los cuatro microservicios backend durante un periodo de observación de aproximadamente 55 minutos.

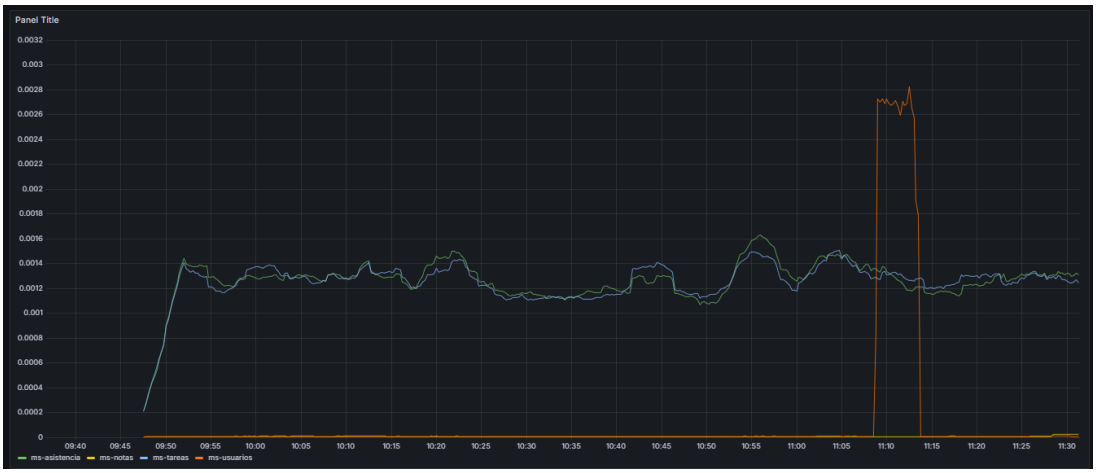



**Figura 23**.
Consumo de CPU. Elaboración propia.

**Tabla XVI**.
Análisis de la métrica de CPU.

| **Microservicio** | **Comportamiento observado** | **Interpretación** |
|---|---|---|
| ms-usuarios | Consumo entre 0.010 y 0.028 núcleos | Mayor carga debido a operaciones de autenticación, hash de contraseñas y gestión de sesiones. |
| ms-tareas | Consumo entre 0.008 y 0.018 núcleos | Carga moderada por operaciones CRUD y gestión de archivos. |
| ms-notas | Consumo entre 0.006 y 0.016 núcleos | Carga moderada-baja, operaciones principalmente de lectura/escritura en PostgreSQL. |
| ms-asistencia | Consumo entre 0.002 y 0.012 núcleos | Carga baja, operaciones simples de registro y consulta en MongoDB. |

Fuente: Elaboración propia.

Como se observa en la tabla, todos los microservicios operan con un consumo de CPU inferior a 0.03 núcleos, lo que representa menos del 3 % de un núcleo completo de CPU. Este resultado indica que el código es altamente eficiente y que el sistema tiene un amplio margen para escalar horizontalmente ante un aumento de la demanda.

En cuanto al microservicio ms-usuarios, presenta un consumo más alto, lo cual es esperable dado que implementa operaciones criptográficas como el hashing de contraseñas con bcrypt y la validación de tokens JWT, operaciones intencionalmente costosas en CPU por razones de seguridad. De otra manera, ninguno de los microservicios muestra picos repentinos o com-

portamientos oscilatorios irregulares. Esta estabilidad sugiere que no existen bucles infinitos, fuga de recursos ni contención por bloques en el código implementado. El bajo consumo de CPU implica que, en un entorno de producción, se podrían ejecutar múltiples réplicas de cada microservicio en el mismo nodo sin saturar los recursos del servidor.

La siguiente gráfica muestra el consumo de memoria RAM en MiB de los cuatro microservicios backend durante el mismo periodo de observación.




**Figura 24**.
Consumo de memoria RAM de los microservicios backend (Prometheus + Grafana). Elaboración propia.

Los valores cuantitativos obtenidos al final del período de observación fueron:

**Tabla XVII**.
Consumo de memoria por microservicio

| Microservicio | Consumo de memoria (MiB) |
|---|---|
| ms-usuarios | 34.47 MiB |
| ms-asistencia | 18.83 MiB |
| ms-tareas | 17.88 MiB |
| ms-notas | 8.03 MiB |

Fuente: Elaboración propia.

**Tabla XVIII**.
Análisis del consumo de memoria

| Microservicio | Consumo observado | Interpretación |
|---|---|---|
| ms-usuarios | ~34.5 MiB | Mayor consumo por gestión de sesiones, caché de usuarios y operaciones de autenticación. |
| ms-asistencia | ~18.8 MiB | Consumo moderado, almacena registros de asistencia en memoria. |
| ms-tareas | ~17.9 MiB | Consumo moderado, maneja archivos subidos y operaciones CRUD. |
| ms-notas | ~8.0 MiB | Consumo más bajo, operaciones principalmente de consulta a base de datos. |

Fuente: Elaboración propia.

En la tabla sobre el consumo de RAM se logra observar que los resultados obtenidos consumen menos de 35MiB de RAM cada uno. En comparación con aplicaciones monolíticas tradicionales que llegan a consumir cientos de MiB o incluso GiB, este perfil es sumamente ligero y eficiente. Cabe recalcar que el consumo de memoria es consistente con el modelo de eventos asíncronos de Node.js, que mantiene una huella de memoria reducida en comparación con otros entornos de ejecución como las JVM de Java o los workers de Python.

Adicionalmente, la estabilidad de las líneas de tendencia sin crecimiento monótono a lo largo del tiempo indica que no existen memory leaks en el código; en caso de que se presentara una fuga de memoria, se observaría un aumento progresivo y constante en el consumo de RAM hasta alcanzar el límite del contenedor, de modo que provocaría un reinicio por OOM-Killed (Out of Memory Killed).

En la siguiente figura se muestra un periodo más extenso que nos permite observar la evolución y estabilidad de las métricas en el tiempo.

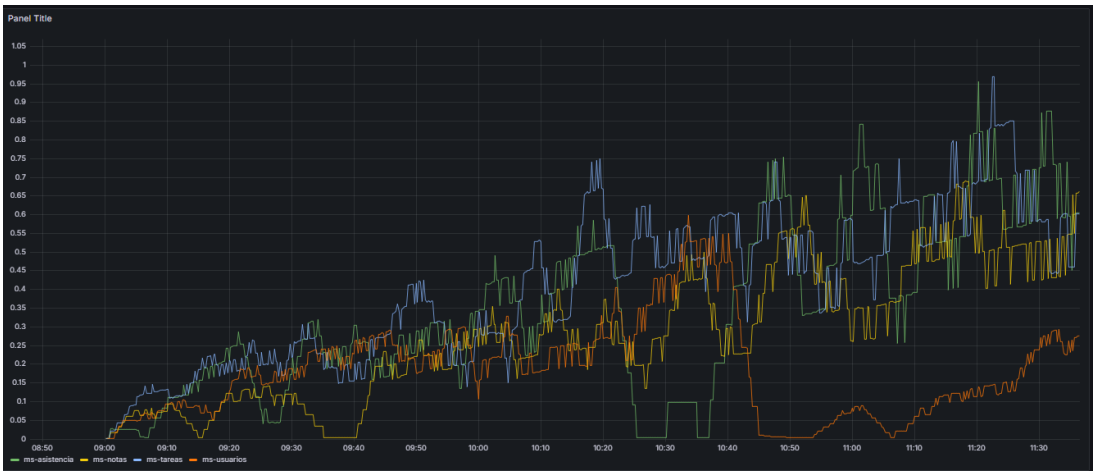



**Figura 25**.
Evolución temporal del consumo de CPU de los microservicios backend (vista ampliada). Fuente: Elaboración propia.

En la figura 25 se logra observar el comportamiento de 55 minutos que se mantiene consistente durante las 2 horas y 40 minutos de monitoreo. No se detectan cambios bruscos de comportamiento ni degradación del rendimiento con el paso del tiempo. Las curvas de consumo son suaves y predecibles, lo que facilita la configuración de sistemas de alerta basados en umbrales estáticos o dinámicos. La diferencia en el consumo entre ms/usuarios es más alta y los demás servicios se mantienen constantes, validando que no hay interferencias ni contención de recursos entre microservicios que coexisten en el mismo clúster.

También es importante destacar que las métricas de rendimiento presentadas corresponden a la versión de los microservicios desplegados automáticamente por el pipeline CI/CD etiquetada con el SHA del commit. Esto significa que cada métrica de rendimiento puede asociarse a una versión específica del código. Si una modificación posterior se introdujera, generaría una degradación en el rendimiento, como por ejemplo el aumento de un 300 % de consumo en el CPU.

Además, las métricas obtenidas confirman que el pipeline no solo despliega correctamente, sino que el código desplegado es operativamente eficiente. Los valores obtenidos constituyen una línea base de rendimiento contra la cual se pueden comparar versiones futuras del sistema.

**Tabla XIX**.
Resumen de métricas de rendimiento por microservicio

| Métrica | ms-usuarios | ms-tareas | ms-notas | ms-asistencia |
|---|---|---|---|---|
| CPU (núcleos) | 0.010 - 0.028 | 0.008 - 0.018 | 0.006 - 0.016 | 0.002 - 0.012 |
| Memoria RAM (MiB) | 34.47 | 17.88 | 8.03 | 18.83 |
| Estabilidad | Alta | Alta | Alta | Alta |
| Riesgo de fuga | No detectado | No detectado | No detectado | No detectado |

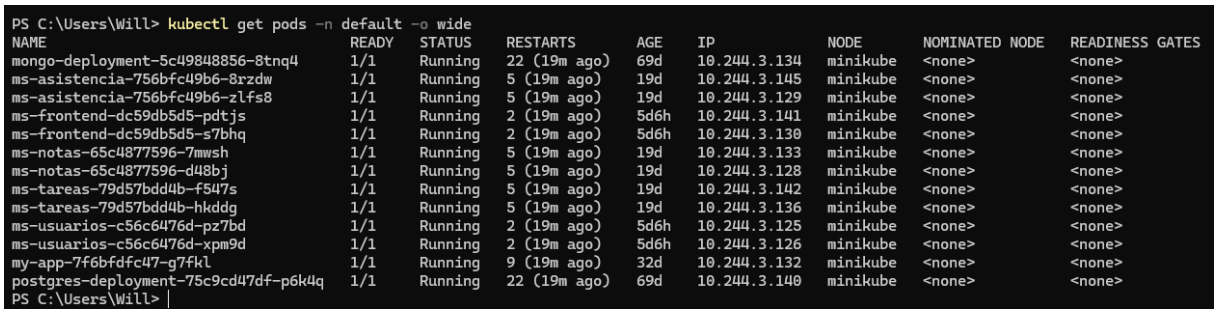
Fuente: Elaboración propia a partir de métricas de Prometheus y Grafana

En resumen, las métricas de rendimiento recolectadas mediante Prometheus y visualizadas en Grafana demuestran que los microservicios implementados son altamente eficientes en el uso de recursos computacionales. El bajo consumo de CPU y memoria, junto con la estabilidad observada durante periodos prolongados, valida la calidad del código generado y la correcta configuración de los contenedores y del orquestador Kubernetes. Finalmente, estas métricas recolectadas son fundamentales para la detección temprana de regresiones en futuras iteraciones del sistema.

#### 4.1.2.4 Métrica 4: Confiabilidad: Estado de Pods y Persistencia.

En esta sección se evalúa la confiabilidad de la solución implementada desde dos perspectivas complementarias: el estado y la disponibilidad de los pods orquestados por Kubernetes y la persistencia de datos ante reinicios y actualizaciones del sistema.

El estado de los pods es el indicador más directo de la salud operativa de un clúster Kubernetes. Un pod en estado runnig indica que todos sus contenedores están en funcionamiento y que el servicio está disponible para procesar solicitudes, en la siguiente captura se muestra el estado detallado de todos los pods desplegados en el clúster.



```
PS C:\Users\Will> kubectl get pods -n default -o wide
NAME                                READY   STATUS    RESTARTS       AGE    IP             NODE       NOMINATED NODE   READINESS GATES
mongo-deployment-5c49848856-8tnq4   1/1     Running   22 (19m ago)   69d    10.244.3.134   minikube   <none>           <none>
ms-asistencia-756bfc49b6-8rzdw      1/1     Running   5 (19m ago)    19d    10.244.3.145   minikube   <none>           <none>
ms-asistencia-756bfc49b6-zlfs8      1/1     Running   5 (19m ago)    19d    10.244.3.129   minikube   <none>           <none>
ms-frontend-dc59db5d5-pdtjs         1/1     Running   2 (19m ago)    5d6h   10.244.3.141   minikube   <none>           <none>
ms-frontend-dc59db5d5-s7bhq         1/1     Running   2 (19m ago)    5d6h   10.244.3.130   minikube   <none>           <none>
ms-notas-65c4877596-7mwsh           1/1     Running   5 (19m ago)    19d    10.244.3.133   minikube   <none>           <none>
ms-notas-65c4877596-d48bj           1/1     Running   5 (19m ago)    19d    10.244.3.128   minikube   <none>           <none>
ms-tareas-79d57bdd4b-f547s          1/1     Running   5 (19m ago)    19d    10.244.3.142   minikube   <none>           <none>
ms-tareas-79d57bdd4b-hkddg          1/1     Running   5 (19m ago)    19d    10.244.3.136   minikube   <none>           <none>
ms-usuarios-c56c6476d-pz7bd         1/1     Running   2 (19m ago)    5d6h   10.244.3.125   minikube   <none>           <none>
ms-usuarios-c56c6476d-xpm9d         1/1     Running   2 (19m ago)    5d6h   10.244.3.126   minikube   <none>           <none>
my-app-7f6bfdfc47-g7fkl             1/1     Running   9 (19m ago)    32d    10.244.3.132   minikube   <none>           <none>
postgres-deployment-75c9cd47df-p6k4q   1/1     Running   22 (19m ago)   69d    10.244.3.140   minikube   <none>           <none>
PS C:\Users\Will>
```

**Figura 26**.
Estado de los pods en el namespace default. Fuente: Elaboración propia.

### Análisis del estado de los pods:

**Tabla XX**.
Estado de los microservicios y despliegues en Kubernetes

| Microservicio | Réplicas | Estado | RESTARTS | Edad | IP asignada |
|---|---|---|---|---|---|
| ms-usuarios | 2 | Running | 2 (hace 19 min) | 5d6h | 10.244.3.125, 10.244.3.126 |
| ms-tareas | 2 | Running | 5 (hace 19 min) | 19d | 10.244.3.142, 10.244.3.136 |
| ms-notas | 2 | Running | 5 (hace 19 min) | 19d | 10.244.3.133, 10.244.3.128 |
| ms-asistencia | 2 | Running | 5 (hace 19 min) | 19d | 10.244.3.145, 10.244.3.129 |
| ms-frontend | 2 | Running | 2 (hace 19 min) | 5d6h | 10.244.3.141, 10.244.3.130 |
| postgres-deployment | 1 | Running | 22 (hace 19 min) | 69d | 10.244.3.140 |
| mongo-deployment | 1 | Running | 22 (hace 19 min) | 69d | 10.244.3.134 |

Fuente: Elaboración propia.

Como se puede observar, todos los pods se encuentran en estado running y con readiness 1/1, lo que indica que el 100 % de los servicios están operativos y listos para recibir tráfico. También los microservicios de aplicación cuentan con 2 réplicas cada uno. Esta configuración garantiza que, si una réplica falla o se reinicia, la otra continúa atendiendo solicitudes, evitando tiempo de inactividad.

A diferencia de los microservicios de aplicación, las bases de datos se ejecutan con una sola réplica. Esta decisión de diseño es adecuada para un entorno de desarrollo y pruebas, aunque en producción se recomendaría configurar clústeres replicados, por ejemplo: PostgreSQL

con replicación primaria-secundaria, MongoDB con ReplicaSet.

A continuación, se presenta el estado de los deployments, que confirma la configuración deseada de réplicas.

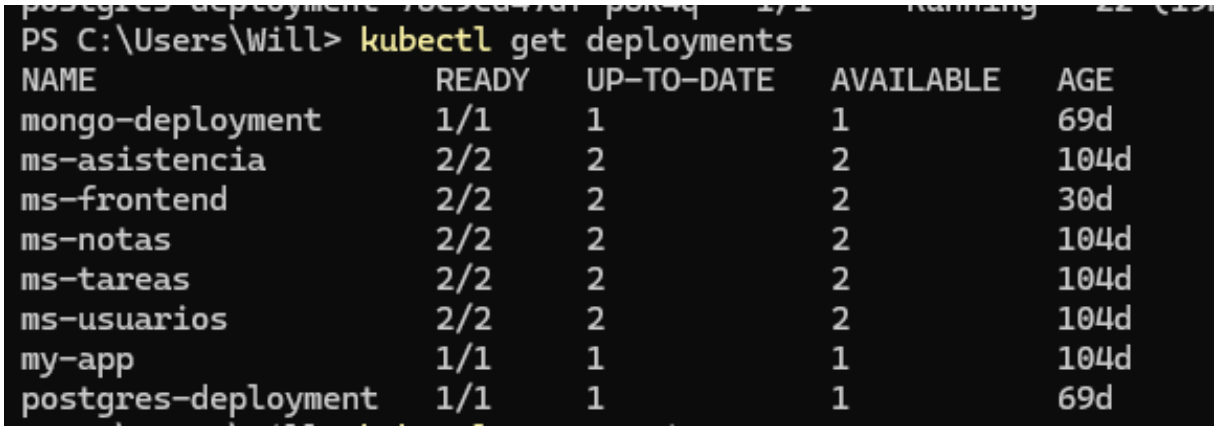



**Figura 27**.
Estado de los deployments. Fuente: Elaboración propia.

Los deployments muestran valores 2/2 en las columnas READY, UP-TO-DATE y AVAILABLE para los microservicios de aplicacion lo que confirma que:

- READY (2/2): De las 2 réplicas definidas, 2 están listas para servir tráfico.
- UP-TO-DATE(2): Las 2 réplicas están ejecutando la versión más reciente del código.
- AVAILABLE(2): Las 2 réplicas están disponibles para los usuarios sin interrupciones.

Ahora, para que los microservicios sean accesibles desde fuera del clúster, se definieron servicios de tipo NodePort y ClusterIP.

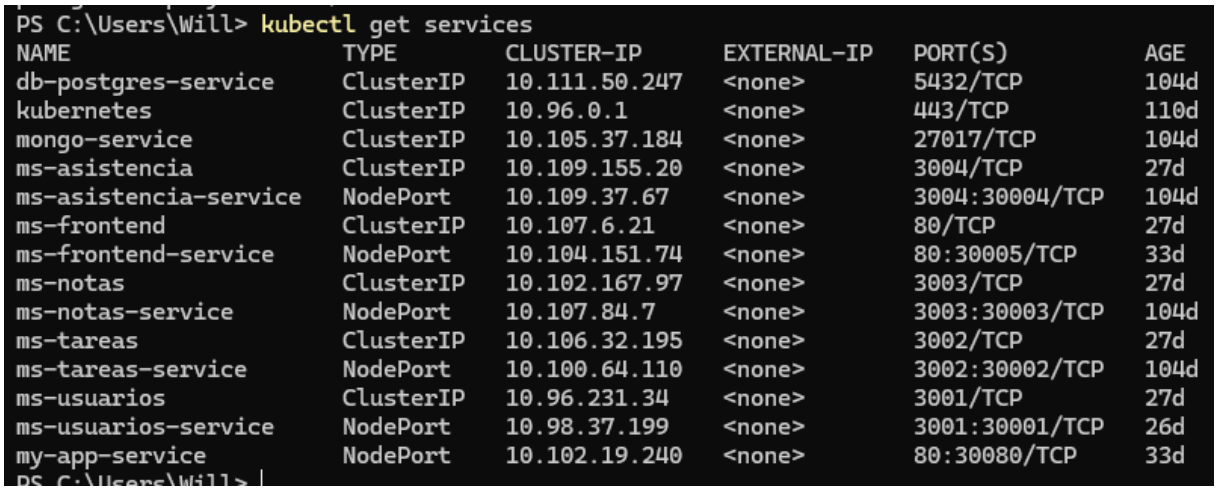



**Figura 28**.

Servicios expuestos en el clúster. Fuente: Elaboración propia.

**Tabla XXI**.

Servicios desplegados en Kubernetes

| Servicio | Tipo | Puerto(s) | Propósito |
|---|---|---|---|
| ms-frontend-service | NodePort | 3003/TCP | Acceso externo a la interfaz web |
| ms-usuarios | ClusterIP | 80:30080/TCP | Comunicación interna con otros microservicios |
| ms-tareas-service | NodePort | 3001/TCP | Acceso externo a API de tareas |
| ms-notas-service | NodePort | 3002/TCP | Acceso externo a API de notas |
| ms-asistencia-service | NodePort | 80/TCP | Acceso externo a API de asistencia |
| db-postgres-service | ClusterIP | 5432/TCP | Comunicación interna con PostgreSQL |
| mongo-service | ClusterIP | 27017/TCP | Comunicación interna con MongoDB |

Fuente: Elaboración propia.

La combinación de servicios ClusterIP para comunicación interna entre microservicios y NodePort para acceso externo desde navegadores o clientes API garantiza que la arquitectura sea accesible y segura, aislando la comunicación interna de la red externa.

La persistencia de datos es un requisito crítico en cualquier sistema de gestión académica, ya que los usuarios esperan que su informacion no se pierda ante reinicios o actualizaciones del sistema. Para garantizar esto, se implementaron PersistentVolumeClaims abreviado PVC que abstraen el almacenamiento subyacente.

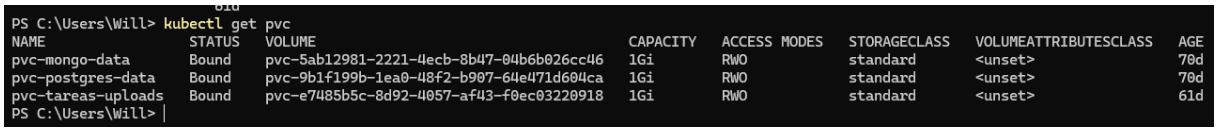



**Figura 29**.

PersistentVolumeClaims (PVC) en el clúster. Fuente: Elaboración propia.

**Tabla XXII**.

Persistent Volume Claims (PVC) utilizados en Kubernetes

| PVC Nombre | Status | Volumen asociado | Propósito |
|---|---|---|---|
| pvc-postgres-data | Bound | pvc-9b1f199b-1ea0-48f2-b907-64e471d604ca | Datos relacionales (usuarios, notas) |
| pvc-mongo-data | Bound | pvc-5ab12981-2221-4ecb-8b47-04b6b026cc46 | Datos documentales (tareas, asistencia) |
| pvc-tareas-uploads | Bound | pvc-e7485b5c-8d92-4057-a4f3-f0ec03220918 | Archivos subidos (PDFs, imágenes) |

PersistentVolumes (PV) físicos. Fuente: Elaboración propia.




**Figura 30**.

Obtención de PV funcionales. Fuente: Elaboración propia.

### Análisis de la configuración de persistencia

**Tabla XXIII**.
Características de los Persistent Volumes y Persistent Volume Claims

| Característica | Valor | Interpretación |
|---|---|---|
| Capacidad | 1Gi cada PV | Almacenamiento suficiente para entorno de pruebas. En producción, este valor debería aumentarse. |
| Access Modes | RWO (ReadWriteOnce) | El volumen puede ser montado en modo lectura-escritura por un solo nodo. Apropiado para bases de datos. |
| Reclaim Policy | Delete | Al eliminar el PVC, se elimina el PV asociado. En producción, se recomienda `Retain` para evitar pérdida de datos. |
| Status | Bound (3/3) | Los 3 PVCs están correctamente vinculados a sus PVs, indicando que el almacenamiento está disponible. |

Fuente: Elaboración propia

En conclusión, los tres PVCs están en estado Bound, lo que confirma que el almacenamiento persistente está correctamente configurado y disponible para los microservicios que lo requieren.

### Verificación de persistencia en PostgreSQL

Para validar que los datos relacionales persisten entre reinicios, se consultaron las tablas usuarios y notas.

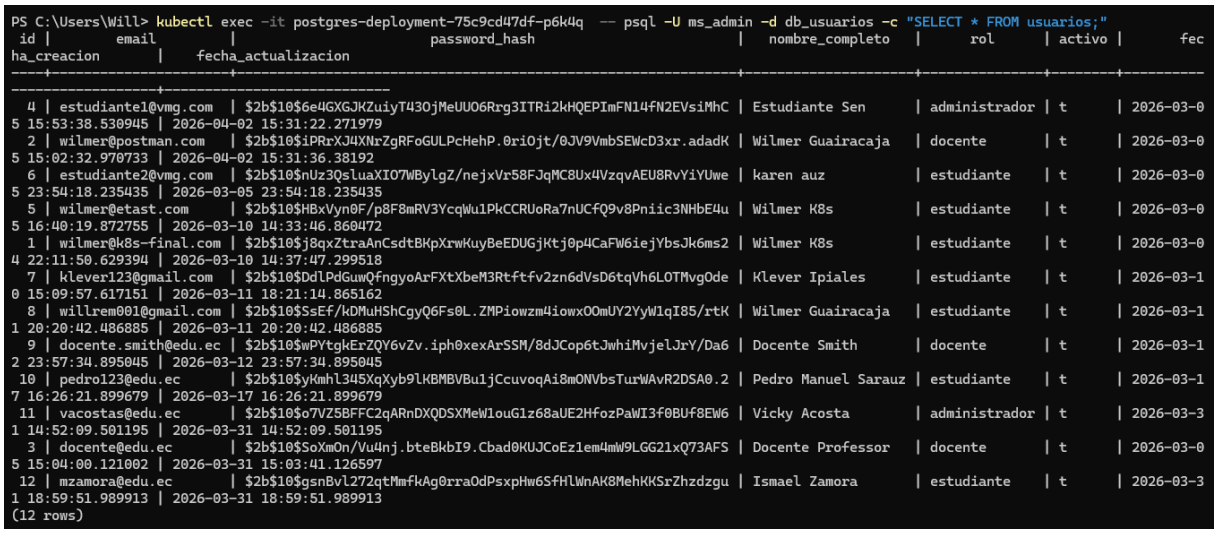



**Figura 31**.
Datos persistentes en PostgreSQL - tabla usuarios (db_usuarios). Fuente: Elaboración propia.

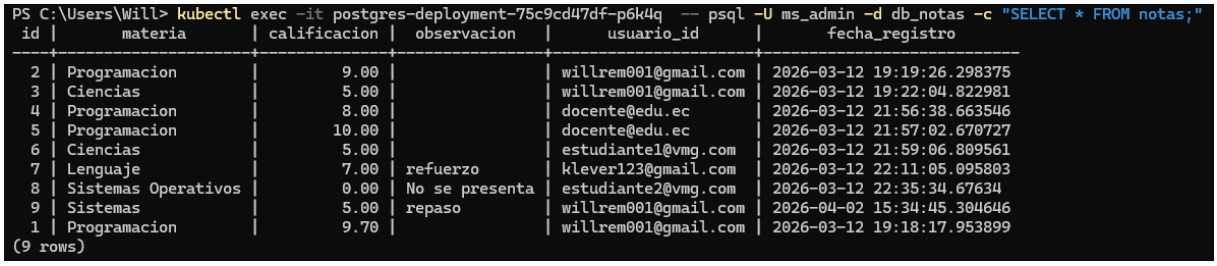



**Figura 32**.
Datos persistentes en PostgreSQL - tabla notas (db_notas). Fuente: Elaboración propia.

Ambos conjuntos de datos han persistido durante 69 días desde la creación del pod postgres-deployment, demostrando que los reinicios del contenedor, que se registraron 22 reinicios, no han afectado la integridad de los datos.

Para validar que los datos no relacionales persisten entre reinicios, se consultaron las colecciones tareas y asistencias.

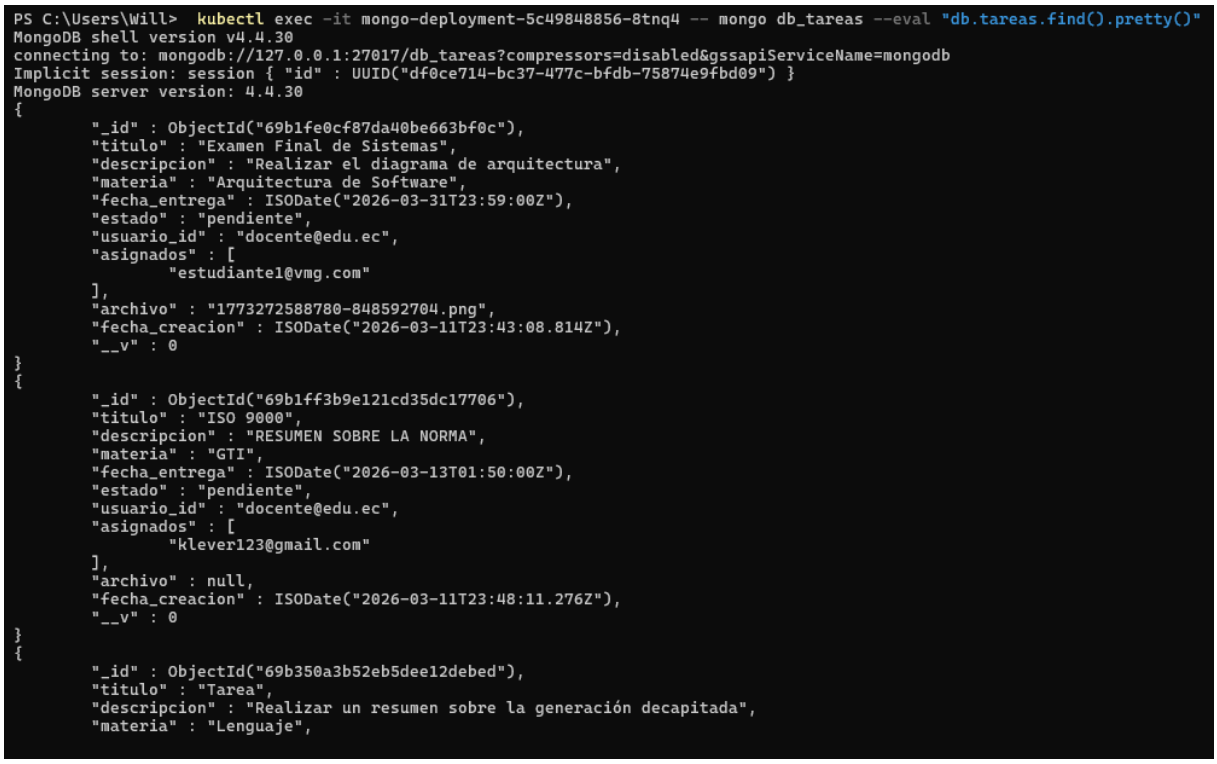



**Figura 33**.
Datos persistentes en MongoDB - colección tareas (db_tareas). Fuente: Elaboración propia.




**Figura 34**.
Datos persistentes en MongoDB - colección asistencias (db_asistencia). Fuente: Elaboración propia.

Los documentos en ambas colecciones han persistido durante toda la vida del clúster, validando que la estrategia de persistencia documental es efectiva. La flexibilidad del esquema JSON ha permitido evolucionar los datos sin migraciones complejas.

### Resiliencia: Auto-recuperación de pods

En la figura 27 en la columna RESTARTS en la salida de kubectl get pods muestra eventos de reinicio en varios contenedores:

**Tabla XXIV**.
Análisis de reinicios de Pods en Kubernetes

| Pod | Reinicios | Último reinicio | Interpretación |
|---|---|---|---|
| postgres-deployment | 22 | hace 19 días | Posibles fallos transitorios de recursos o *liveness probes*. |
| mongo-deployment | 22 | hace 19 días | Similar a PostgreSQL. |
| ms-asistencia (2 pods) | 5 | hace 19 días | Reinicios controlados por Kubernetes. |
| ms-notas (2 pods) | 5 | hace 19 días | Reinicios controlados por Kubernetes. |
| ms-tareas (2 pods) | 5 | hace 19 días | Reinicios controlados por Kubernetes. |
| ms-usuarios (2 pods) | 2 | hace 19 días | Menos reinicios, mayor estabilidad. |
| ms-frontend (2 pods) | 2 | hace 19 días | Menos reinicios, mayor estabilidad. |

Fuente:Elaboración propia.

**Análisis de los reinicios**

- Auto-recuperación efectiva: A pesar de los reinicios, todos los pods se encuentran actualmente en estado running. Esto demuestra que Kubernetes ha gestionado automáticamente estos fallos, recreando los contenedores sin intervención manual.
- Causas probables de reinicios: En un entorno de desarrollo local con Minikube, los reinicios pueden deberse a:
    - Liveness probes fallidas: Kubernetes reinicia contenedores que no responden a las sondas de salud.
    - Límites de recursos OOMKilled: Si un contenedor excede su límite de memoria, es terminado y reiniciado.
    - Fallos transitorios de red: Perdidas temporales de conectividad entre servicios.
    - Actualizaciones del pipeline: Cada despliegue automatizado del pipeline CI/CD recrea los pods con nuevas imágenes.

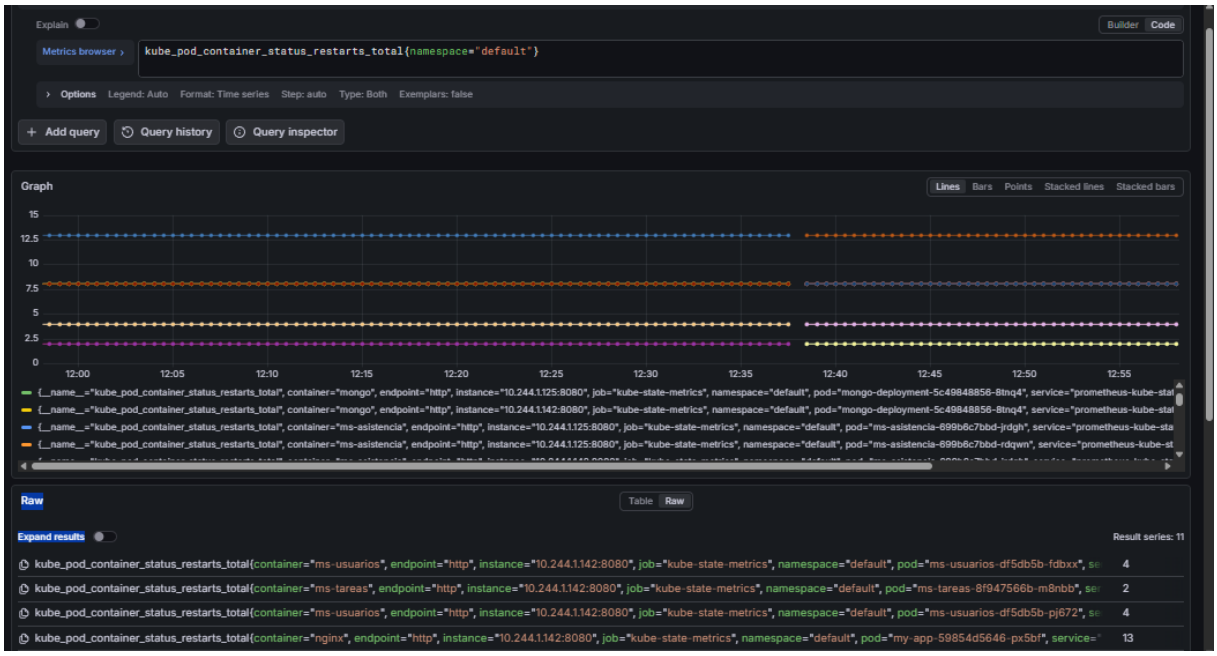



**Figura 35**.
Reinicios de pods en Grafana. Elaboración propia.

- Correlación con métricas de Prometheus: La métrica kube_pod_container_status_restarts_total como se ve en la figura 35 confirma estos reinicios y proporciona una visión histórica de la estabilidad del sistema.
- Resiliencia demostrada: El hecho de que los microservicios hayan operado durante 104 días con múltiples reinicios, sin pérdida de datos y manteniendo la disponibilidad, valida la resiliencia de la arquitectura.

### Resumen de confiabilidad

**Tabla XXV**.
Resumen de confiabilidad del sistema

| Aspecto evaluado | Resultado | Conclusión |
|---|---|---|
| Estado de pods | 100 % en Running | Sistema operativo y disponible. |
| Alta disponibilidad | 2 réplicas por microservicio | Tolerancia a fallos de instancias individuales. |
| PersistentVolumes (PV) | 3 PVs de 1Gi, estado Bound | Almacenamiento persistente configurado. |
| PersistentVolumeClaims (PVC) | 3 PVCs, estado Bound | Microservicios pueden reclamar almacenamiento. |
| Persistencia PostgreSQL | 12 usuarios + 9 notas persistidos | Datos relacionales intactos tras 22 reinicios. |
| Persistencia MongoDB | Documentos de tareas y asistencia intactos | Datos documentales intactos tras 22 reinicios. |
| Resiliencia | Pods se autorrecuperan tras reinicios | Kubernetes gestiona fallos automáticamente. |
| Tiempo operativo continuo | 104 días (microservicios), 69 días (BD) | Estabilidad probada en el tiempo. |

Fuente:Elaboración propia.

En conclusión, la métrica de confiabilidad demuestra que la solución implementada es altamente confiable, con disponibilidad total de servicios, persistencia garantizada de datos ante reinicios y capacidad de autorecuperación gestionada por Kubernetes. Estos resultados validan la elección de Kubernetes como orquestador y la estrategia de persistencia basada en PVCs, cumpliendo con los requisitos de un sistema de gestión académica donde la integridad de los datos es crítica.

## 4.2 Análisis de resultados

Los resultados obtenidos corresponden a un entorno controlado semilocal basado en Minikube, por lo que el comportamiento en ambientes productivos distribuidos puede variar dependiendo de la infraestructura y carga operativa, validando cada una de las variables de estudio planteadas en el Capítulo III.

- Eficiencia del Pipeline CI/CD: La implementación del pipeline reduce el tiempo de despliegue de aproximadamente 45 minutos por servicio (manual) a 6 minutos y 8 segundos para 5 servicios en paralelo, lo que representa una mejora del 87 % Tabla XV. Si bien la tasa de éxito general fue del 96.8 % (122 ejecuciones exitosas de 126), el análisis de los 4 fallos resultó crucial para depurar la integración con GHCR, las políticas imagePullPolicy y la red de Minikube, logrando una estabilidad del 100 % en las ejecuciones recientes desde la #120. Esto demuestra que el pipeline no solo es rápido, sino que su confiabilidad ha sido perfeccionada iterativamente.

- Calidad y Cobertura de Código: Alcanzar un 100 % de cobertura en pruebas unitarias de los módulos críticos del frontend "auth.service, Login, AuthContext.[es] un hito significativo. Esto garantiza que la lógica de negocio fundamental está libre de errores de regresión. Si bien la ausencia de pruebas de integración y E2E es una limitación reconocida, la cobertura actual establece una base de confianza absoluta sobre la cual se pueden construir pruebas más complejas en el futuro.

- Comportamiento de la Arquitectura (Microservicios): El análisis de rendimiento valida las decisiones tecnológicas y de diseño. El bajo consumo de CPU <0.03 núcleos y memoria <35 MiB por servicio confirma que Node.js es una plataforma excepcionalmente ligera para microservicios. El mayor consumo de ms-usuarios 34.47 MiB es esperable por el costo computacional del hashing bcrypt y la gestión de JWT, demostrando que la carga está justificada y distribuida correctamente.

- Estrategia de Persistencia : La elección de PostgreSQL para datos relacionales usuarios, notas y MongoDB para datos documentales tareas, asistencia fue acertada. La alta memoria de MongoDB  200 MB fuera de las métricas de la Tabla XIX, verificada en paneles anteriores no es un problema, sino una característica de su motor de caché. La persistencia de 12 usuarios, 9 notas y documentos de tareas tras 22 reinicios de los pods de bases de datos demuestra que la estrategia de PersistentVolumeClaims (PVCs) es robusta y garantiza la integridad de los datos.

- Resiliencia y observabilidad Los reinicios detectados, 22 en BD, 5 en microservicios,

son la prueba más valiosa de la implementación. Sin la integración de Prometheus y Grafana, estos eventos pasarían desapercibidos. Gracias a la observabilidad, no solo se identificaron, sino que se confirma que Kubernetes los gestionó automáticamente auto-recuperación, manteniendo la disponibilidad del servicio. Esto demuestra que el sistema no solo es funcional, sino tolerante a fallos y estable durante períodos prolongados 104 días de operación de los microservicios.

## 4.3 Cumplimiento de objetivos

Se implementó exitosamente un pipeline automatizado de Integración Continua y Despliegue Continuo (CI/CD) utilizando GitHub Actions, Docker y Kubernetes (Minikube). El pipeline optimiza los procesos de prueba (ejecutando pruebas con 100 % de cobertura en módulos críticos y bloqueando código defectuoso) y implementando (actualizando 5 microservicios en paralelo en un tiempo promedio de 6 minutos y 8 segundos), cumpliendo una cabalidad con el objetivo general de optimizar los procesos de pruebas e implementación de software en arquitecturas de microservicios.

### 4.3.1 Cumplimiento de objetivos específicos

- Realizar un análisis bibliográfico: Se construyó en el Capítulo II, estableciendo un marco teórico sólido sobre la evolución de las arquitecturas de software hacia microservicios, los fundamentos de DevOps, los patrones de CI/CD (GitHub Actions, GitOps), la importancia de la calidad/seguridad y la observabilidad con Prometheus/Grafana. Este análisis guió la selección de todas las herramientas utilizadas en la implementación práctica.
- Construir la arquitectura del pipeline CI/CD: Se construyó una arquitectura de pipeline basada en una estrategia matricial (strategy.matrix) que sigue las buenas prácticas DevOps, incluyendo:

- Automatización total: Desde el empuje hasta el despliegue.
- Ejecución paralela: Construcción y despliegue simultáneo de 5 servicios.
- Infraestructura como Código (IaC): Gestión de todo el clúster implementaciones, servicios, PVCs mediante manifiestos YAML versionados en Git.
- Política fail-fast: Detención inmediata del pipeline ante fallos, acelerando la retro-alimentación al desarrollador.

- Validar el funcionamiento y eficiencia del ducto: La validación se realizó de manera cuantitativa y objetiva mediante las métricas recolectadas:
  - Eficiencia del Pipeline: Tiempo de ejecución promedio de 6m 8s y tasa de éxito estabilizada en 100 %.
  - Calidad de Código: 100 % de cobertura de pruebas unitarias en módulos críticos de autenticación y frontend.
  - Rendimiento: Consumo de CPU <0,03 núcleos y memoria <35 MiB por microservicio, demostrando alta eficiencia.
  - Confiabilidad y Resiliencia: Estado Running del 100 % de los pods, persistencia de datos verificada tras 22 reinicios de las bases de datos, y auto-recuperación demostrada por Kubernetes.

# CONCLUSIONES

La implementación del pipeline CI/CD utilizando herramientas de código abierto como GitHub Actions y Kubernetes demostró ser altamente eficiente, permitiendo reducir enormemente los tiempos de integración y despliegue en comparación con procesos manuales. La automatización de pruebas y construcción garantizó que solo el código verificado llegue al entorno de producción.

El desarrollo de la plataforma con una arquitectura de microservicios (Usuarios, Tareas, Asistencia, Notas) probó que es una arquitectura viable y escalable para proyectos académicos. La modularidad permitió actualizar servicios específicos sin afectar el funcionamiento global del sistema.

La integración de Prometheus y Grafana fue fundamental para el éxito del proyecto. El monitoreo continuo permitió detectar comportamientos anómalos, como los reinicios en el pod ms-usuarios, y visualizar la distribución de carga, evidenciando que la base de datos MongoDB es el componente más demandante en términos de memoria, mientras que los microservicios operan con un bajo perfil de consumo.

A pesar de los reinicios observados en algunos contenedores, la infraestructura de Kubernetes demostró robustez al auto-recuperar los servicios, asegurando la disponibilidad del sistema. Esto valida la elección de Kubernetes como orquestador para gestionar la complejidad operativa de los microservicios.

Los análisis de CPU y memoria revelaron que, bajo las condiciones de prueba, el sistema es ligero y eficiente. El consumo de CPU cercano a cero y la estabilidad en la memoria de los pods de aplicación indican que la infraestructura ”semi localïmplementada es suficiente para entornos de desarrollo y pruebas de carga moderada.

# RECOMENDACIONES

El presente trabajo se ejecutó sobre un clúster de un solo nodo (Minikube) en un entorno semilocal. Para entornos productivos, se recomienda implementar la solución sobre un clúster Kubernetes de múltiples nodos con balanceadores de carga reales, lo que mejoraría la alta disponibilidad, la tolerancia a fallos y la capacidad de escalamiento horizontal de efectivo.

Si bien se alcanzó una cobertura del 100 % en pruebas unitarias de frontend, no se implementaron pruebas de integración entre microservicios ni pruebas E2E que simulen el flujo completo de un usuario autenticado. Se recomienda integrar herramientas como Cypress o Playwright para pruebas E2E, y Pact para pruebas de contrato entre servicios, con el fin de validar la comunicación real entre los microservicios y detectar regresiones en las API.

Actualmente, las credenciales de bases de datos y tokens JWT se gestionan mediante variables de entorno en los manifiestos de Kubernetes. En entornos productivos, se recomienda implementar un sistema de gestión de secretos como HashiCorp Vault o Secrets Store CSI Driver para Kubernetes, lo que eliminaría la exposición de información sensible en archivos YAML.

Aunque se configuró Prometheus y Grafana para la visualización de métricas, no se implementaron sistemas de alertas automáticas. Se recomienda configurar Alertmanager con reglas basadas en umbrales críticos, por ejemplo: consumo de CPU >80 %, memoria cercana al límite, pods en estado CrashLoopBackOff, y conectar las alertas a canales de notificación como Slack o Telegram para una respuesta inmediata ante incidentes.

El pipeline actual utiliza kubectl set image para actualizar los despliegues de forma imperativa. Se recomienda migrar a un enfoque GitOps utilizando herramientas como Argo CD oFlux, donde el estado deseado del clúster se almacena en el repositorio Git y un agente sincroniza automáticamente el clúster con dicho estado. Esto mejoraría la trazabilidad, auditabilidad y seguridad de los despliegues.

# BIBLIOGRAFÍA


[1] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2021.

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente et al., *Microservices: Yesterday, Today, and Tomorrow*, http://arxiv.org/abs/1606.04036, Accedido: 10 mayo 2025, 2016.

[3] G. Kim, J. Humble, P. Debois y J. Willis, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2016.

[4] N. Forsgren, J. Humble y G. Kim, *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 2018.

[5] M. J. Kargar y A. Hanifizade, «Automation of Regression Test in Microservice Architecture,» en *2018 4th International Conference on Web Research (ICWR)*, 2018, págs. 133-137. DOI: 10.1109/ICWR.2018.8387249.

[6] S. Montoya y M. A. Ocampo, *Conoce 5 herramientas para integración y entrega continua con DevOps*, https://www.pragma.co/es/blog/conoce-cinco-herramientas-para-integracion-y-entrega-continua-con-devops, Accedido: 10 mayo 2025, mar. de 2022.

[7] P. Baddula, *The evolution of software architecture: Monolithic to microservices*, en, https://medium.com/@phanindra208/the-evolution-of-software-architecture-monolithic-to-microservices-cb62fcd7aa94, Accessed: 2025-06-20, jul. de 2023.

[8] G. Blinowski, A. Ojdowska y A. Przybyłek, «Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation,» *IEEE Access*, vol. 10, págs. 20 357-20 372, 2022. DOI: 10.1109/ACCESS.2022.3152803.

[9] R. Awati e I. Wigmore. «What is monolithic architecture in software?» WhatIs. (oct. de 2024), dirección: https://www.techtarget.com/whatis/definition/monolithic-architecture (visitado 24-06-2025).

[10] P. Powell e I. Smalley. «Monolithic vs. Microservices architecture.» (abr. de 2025), dirección: https://www.ibm.com/think/topics/monolithic-vs-microservices (visitado 20-06-2025).

[11] O. Fernández. «Microservicios: Introducción a la Arquitectura,» Aprender BIG DATA. (14 de ene. de 2025), dirección: https://aprenderbigdata.com/microservicios/ (visitado 17-06-2025).

[12] S. Pittet. «Continuous integration vs. delivery vs. deployment, Learn the differences between these continuous practices,» Atlassian. (2025), dirección: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment (visitado 17-06-2025).

[13] Atlassian. «Qué son los microservicios.» (2023), dirección: https://www.atlassian.com/es/microservices (visitado 22-06-2025).

[14] J. Parra. «Team Organization in Microservices Environments: A Detailed Approach.» Blog post, Medium. (2024), dirección: https://medium.com/@josueparra2892/team-organization-in-microservices-environments-a-detailed-approach-8d24786fac17 (visitado 19-07-2025).

[15] Microsoft, *Estilo de arquitectura de microservicios*, Accessed: 2025-06-17, Azure Architecture Center, 2025. dirección: https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices.

[16] Á. d. S. Moura, F. G. Rocha y M. S. Soares, «Microservices Patterns Recommendation based on Information Retrieval,» *Journal of Universal Computer Science*, vol. 30, n.º 11, págs. 1455-1483, 2024. DOI: 10.3897/jucs.108974.

[17] A. F. Cosmineanu, *CI/CD para arquitecturas de microservicios*, https://docta.ucm.es/entities/publication/19bab462-2eb4-407b-a7e3-1f6c316b0619, Accedido: 10 mayo 2025, 2022.

[18] Atlassian. «Los principios de DevOps.» Guía sobre los 5 principios fundamentales de DevOps: colaboración, automatización, mejora continua, acción centrada en el cliente y crear con el objetivo en mente, Atlassian. (2024), dirección: https://www.atlassian.com/es/devops/what-is-devops (visitado 22-07-2025).

[19] GlobalAppTesting. «What is Continuous Integration?» (2023), dirección: https://www.globalapptesting.com/blog/what-is-continuous-integration (visitado 22-06-2025).

[20] M. Fowler. «Continuous Integration.» (2006), dirección: https://martinfowler.com/articles/continuousIntegration.html (visitado 22-06-2025).

[21] LightNode. «¿Qué es CI/CD? Una guía completa sobre integración continua y entrega continua.» LightNode Tech Blog, LightNode. (2024), dirección: https://go.lightnode.com/es/tech/what-is-ci-cd#5-cu%C3%A1l-es-un-flujo-de-trabajo-t%C3%ADpico-de-un-pipeline-de-cicd (visitado 22-07-2025).

[22] Atlassian. «Continuous Integration vs. Delivery vs. Deployment.» (2023), dirección: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment (visitado 22-06-2025).

[23] R. Hat. «¿Qué es la Infraestructura como Código (IaC)?» (2023), dirección: https://www.redhat.com/es/topics/automation/what-is-infrastructure-as-code-iac (visitado 22-06-2025).

[24] HashiCorp. «Terraform by HashiCorp,» HashiCorp. (2023), dirección: https://developer.hashicorp.com/terraform (visitado 22-06-2025).

[25] C. Software. «Chef Infra,» Chef Software, Inc. (2023), dirección: https://www.chef.io/products/chef-infra (visitado 22-06-2025).

[26] I. Puppet. «Puppet Infrastructure Automation,» Puppet, Inc. (2023), dirección: https://www.puppet.com/ (visitado 22-06-2025).

[27] R. Hat. «Aprender Ansible: Tutorial básico,» Red Hat, Inc. (2023), dirección: https://www.redhat.com/es/topics/automation/learning-ansible-tutorial (visitado 22-06-2025).

[28] Apiumhub. «Infraestructura como Código: Beneficios y Herramientas.» (2023), dirección: https://apiumhub.com/es/tech-blog-barcelona/infraestructura-como-codigo-beneficios-y-herramientas/ (visitado 21-06-2025).

[29] IBM. «What is a CI/CD Pipeline?» IBM. (2023), dirección: https://www.ibm.com/think/topics/ci-cd-pipeline (visitado 21-06-2025).

[30] V. Jain, «Continuous Testing in CI/CD Pipelines,» *International Journal of Innovative Research and Creative Technology*, vol. 1, 2023. DOI: 10.5281/zenodo.14883221.

[31] Atlassian. «Patrones de diseño de microservicios,» Atlassian. (2023), dirección: https://www.atlassian.com/es/microservices/cloud-computing/microservices-design-patterns (visitado 25-06-2025).

[32] Docker Docs. «Docker Overview.» Accessed: 2025-06-16. (2024), dirección: https://docs.docker.com/get-started/overview/.

[33] Kubernetes Docs. «Kubernetes Concepts.» Accessed: 2025-06-16. (2024), dirección: https://kubernetes.io/docs/concepts/.

[34] The Kubernetes Authors, *Kubernetes Documentation: Pods*, 2023. dirección: https://kubernetes.io/docs/concepts/workloads/pods/ (visitado 25-06-2025).

[35] Kubernetes Authors. «Deployments, Kubernetes Workloads,» The Linux Foundation. (2023), dirección: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visitado 25-06-2025).

[36] Kubernetes Authors. «Services, Kubernetes Networking Concepts,» The Linux Foundation. (2023), dirección: https://kubernetes.io/docs/concepts/services-networking/service/ (visitado 25-06-2025).

[37] Red Hat, Inc. «¿Qué es GitOps?» (2023), dirección: https://www.redhat.com/es/topics/devops/what-is-gitops (visitado 26-06-2025).

[38] Cloud Native Computing Foundation. «GitOps Principles.» (2024), dirección: https://opengitops.dev/ (visitado 22-06-2025).

[39] The Tekton Authors. «Tekton Documentation,» Continuous Delivery Foundation. (2023), dirección: https://tekton.dev/docs/ (visitado 25-06-2025).

[40] Jenkins Community. «Jenkins Automation Server,» Linux Foundation. (2023), dirección: https://www.jenkins.io/ (visitado 25-06-2025).

[41] Argo Project Authors. «Argo CD Documentation, Declarative GitOps CD for Kubernetes,» Cloud Native Computing Foundation. (2023), dirección: https://argo-cd.readthedocs.io/en/stable/ (visitado 25-06-2025).

[42] Flux Maintainers. «Flux CD Documentation, GitOps Toolkit for Kubernetes,» Cloud Native Computing Foundation. (2023), dirección: https://fluxcd.io/ (visitado 25-06-2025).

[43] GitHub Docs. «About GitHub Actions.» Accessed: 2025-06-17. (2024), dirección: https://docs.github.com/es/actions/about-github-actions/understanding-github-actions.

[44] GitLab. «GitLab CI/CD Documentation.» (2023), dirección: https://docs.gitlab.com/ci/ (visitado 25-06-2025).

[45] Atlassian, *Source Code Management with Git*, Accessed: 2025, 2023. dirección: https://www.atlassian.com/es/git/tutorials/source-code-management.

[46] GitHub, Inc. «Flujo de GitHub,» GitHub. (2025), dirección: https://docs.github.com/es/get-started/using-github/github-flow (visitado 01-07-2025).

[47] TrunkBasedDevelopment.com. «Trunk Based Development.» (2025), dirección: https://trunkbaseddevelopment.com/ (visitado 01-07-2025).

[48] TestingIT. «Pruebas automatizadas: Cinco mejores herramientas,» TestingIT. (2024), dirección: https://www.testingit.com.mx/blog/pruebas-automatizadas-cinco-mejores-herramientas (visitado 2025).

[49] N. Moore. «Revisiting the Test Pyramid in 2024,» Qase. (feb. de 2024), dirección: https://qase.io/blog/test-pyramid/ (visitado 2025).

[50] A. Sharma. «The Modern Testing Pyramid: A Practical Approach,» TestGrid. (jul. de 2024), dirección: https://testgrid.io/blog/testing-pyramid/ (visitado 2025).

[51] J. Schmitt. «Understanding the Testing Pyramid,» CircleCI. (dic. de 2024), dirección: https://circleci.com/blog/testing-pyramid/ (visitado 2025).

[52] A. Goala. «Breaking Down the Test Automation Pyramid,» Testlio. (feb. de 2022), dirección: https://testlio.com/blog/breaking-down-the-test-automation-pyramid/ (visitado 2025).

[53] V. K. Thatikonda, «Beyond the Buzz: A Journey Through CI/CD Principles and Best Practices,» *European Journal of Theoretical and Applied Sciences*, vol. 1, n.º 5, págs. 334-340, 2023. DOI: 10.59324/ejtas.2023.1(5).24.

[54] Normaiso27001.es, *Normaiso27001.es*, https://www.normaiso27001.es, Recuperado el 15 de mayo de 2026, s/f.

[55] Testlio. «Static Testing: Definition, Types, Tools, Advantages and Disadvantages.» inglés. Testlio Blog, Testlio. (2024), dirección: https://testlio.com/blog/static-testing/ (visitado 22-07-2025).

[56] Y. Kim, C. Park y D. Hong, «A Robust Framework for Comprehensive Container Image Vulnerability Assessment,» *IEEE Access*, vol. 13, págs. 35 837-35 847, 2025. DOI: 10.1109/ACCESS.2025.3545124.

[57] Keeper Security. «¿Qué es la gestión de secretos? Beneficios y mejores prácticas.» Glosario IAM, Keeper Security. (2024), dirección: https://www.keepersecurity.com/es_ES/resources/glossary/what-is-secrets-management/ (visitado 22-07-2025).

[58] M. Shahin, M. A. Babar y L. Zhu, «Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices,» *IEEE Access*, vol. 5, págs. 3909-3943, 2017. DOI: 10.1109/ACCESS.2017.2685629. dirección: https://doi.org/10.1109/ACCESS.2017.2685629.

[59] C. M. L. F.-J. Miranda, «Observability Engineering: Achieving Production Excellence,» mayo de 2022.

[60] C. Sridharan, *Distributed Systems Observability A Guide to Building Robust Systems*. mayo de 2018, ISBN: 9781492033400. dirección: http://oreilly.com/safari.

[61] B. Gregg, *The USE Method*, Brendangregg.com, ago. de 2017. dirección: https://www.brendangregg.com/usemethod.html.

[62] Grafana. «The RED method: How to instrument your services.» (mar. de 2018), dirección: https://www.youtube.com/watch?v=zk77VS98Em8&t=6s (visitado 15-07-2025).

[63] Google. «Monitoring Distributed Systems.» Chapter from Site Reliability Engineering book, section on Golden Signals. (), dirección: https://sre.google/sre-book/monitoring-distributed-systems/ (visitado 15-07-2025).

[64] Prometheus. «Introduction - Overview.» Official Prometheus documentation overview. (), dirección: https://prometheus.io/docs/introduction/overview/ (visitado 15-07-2025).

[65] G. Labs. «Grafana Documentation.» Official Grafana documentation, latest version. (), dirección: https://grafana.com/docs/grafana/latest/ (visitado 15-07-2025).

[66] Elastic. «¿Qué son los archivos de log?» Guía de Elastic sobre archivos de log y su gestión. (), dirección: https://www.elastic.co/es/what-is/log-files (visitado 15-07-2025).

[67] Elastic. «Get started with the Elastic Stack.» Documentación oficial del Elastic Stack (Elasticsearch, Logstash, Kibana). (), dirección: https://www.elastic.co/docs/get-started/the-stack (visitado 15-07-2025).

[68] Graylog. «What Is Graylog?» Documentación oficial de Graylog sobre sus características y funcionalidades. (), dirección: https://go2docs.graylog.org/current/what_is_graylog/what_is_graylog.htm (visitado 15-07-2025).

[69] G. Labs. «Loki.» Sistema de agregación de logs horizontalmente escalable, inspirado en Prometheus. (), dirección: https://grafana.com/oss/loki/ (visitado 15-07-2025).

[70] Fluentd. «Fluentd Documentation.» Documentación oficial de Fluentd, recolector de datos unificado para logging. (), dirección: https://docs.fluentd.org/ (visitado 15-07-2025).

[71] O. Identity. «Syslog-ng Open Source Edition Administration Guide.» Guía de administración oficial de syslog-ng Open Source Edition. (), dirección: https://support.oneidentity.com/technical-documents/syslog-ng-open-source-edition/3.17/administration-guide/3 (visitado 15-07-2025).

[72] Jaeger Project, «Jaeger Documentation,» Cloud Native Computing Foundation, Technical Documentation, ver. 2.8, 2024. dirección: https://www.jaegertracing.io/docs/2.8/.

[73] Zipkin Project, *Zipkin Documentation*, Apache Software Foundation, 2024. dirección: https://zipkin.io/.

[74] OpenTelemetry Community, *OpenTelemetry Documentation*, Cloud Native Computing Foundation, 2024. dirección: https://opentelemetry.io/docs/.

[75] G. Deshpande, S. Cheekati, S. Patel et al., «Transforming CS Education with DevOps: Streamlined Assignment Validation and Delivery @ Scale,» en *Proceedings of the Eleventh ACM Conference on Learning @ Scale (L@S '24)*, Atlanta, GA, USA: ACM, 2024, págs. 259-262. DOI: 10.1145/3657604.3664676. dirección: https://doi.org/10.1145/3657604.3664676.

[76] Varenyaz Technologies, *Unlocking Business Potential with DevOps CI/CD Implementation in Education*, https://varenyaz.com/unlocking-business-potential-with-devops-ci-cd-implementation-in-education/, Accessed: 2025-07-16, 2023.

[77] Codefresh, *11 CI/CD Best Practices for DevOps Success*, https://codefresh.io/learn/ci-cd/11-ci-cd-best-practices-for-devops-success/, Accessed: 2025-07-16, 2023.

[78] N. K. Ale, «Integrating Performance Testing into CI/CD Pipelines for Test Automation,» *Journal of Scientific and Engineering Research*, vol. 6, págs. 272-278, 2020. DOI: 10.5281/zenodo.12754783.

[79] N. K. Ale, «Strategies for Automating Tests in a Microservices-Based Application,» *European Journal of Advances in Engineering and Technology*, vol. 6, págs. 84-90, 2020. DOI: 10.5281/zenodo.12754689.

[80] A. Ganje, «Microservices in Organizations,» *Journal of Software Engineering and Applications*, vol. 18, n.º 2, págs. 76-86, 2025. DOI: 10.4236/jsea.2025.182005.

# ANEXOS

## Código de Manifiestos, Usuarios y Notas

```
    apiVersion: apps/v1
kind: Deployment
metadata:
  name: ms-usuarios
spec:
  replicas: 2
  selector:
    matchLabels:
      app: ms-usuarios
  template:
    metadata:
      labels:
        app: ms-usuarios
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "3001"
        prometheus.io/path: "/metrics"
    spec:
      containers:
      - name: ms-usuarios
        image: ghcr.io/willrem001-beep/sistema-educativo-ci-cd/ms-usuarios:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 3001
        env:
        - name: DB_HOST
```

```
          value: "db-postgres-service"  # Nombre del servicio K8s
        - name: DB_PORT
          value: "5432"
        - name: DB_NAME
          value: "db_usuarios"
        - name: JWT_EXPIRES_IN
          value: "3h"
        - name: DB_USER
          valueFrom:
            secretKeyRef:
              name: db-secrets
              key: db-user
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secrets
              key: db-password
        - name: JWT_SECRET
          valueFrom:
            secretKeyRef:
              name: jwt-secrets
              key: jwt-secret
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ms-notas
spec:
  replicas: 2
  selector:
```

```
  matchLabels:
    app: ms-notas
template:
  metadata:
    labels:
      app: ms-notas
    annotations:
      prometheus.io/scrape: "true"
      prometheus.io/port: "3003"
      prometheus.io/path: "/metrics"
  spec:
    containers:
    - name: ms-notas
      image: ghcr.io/willrem001-beep/sistema-educativo-ci-cd/ms-notas:latest
      imagePullPolicy: Always
      ports:
      - containerPort: 3003
      env:
      - name: DB_HOST
        value: "db-postgres-service"
      - name: DB_PORT
        value: "5432"
      - name: DB_NAME
        value: "db_notas"
      - name: DB_USER
        valueFrom:
          secretKeyRef:
            name: db-secrets
            key: db-user
```

```
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secrets
              key: db-password
```

**Código de Manifiestos, Tareas y Asistencia**

```
    apiVersion: apps/v1
kind: Deployment
metadata:
  name: ms-tareas
spec:
  replicas: 2
  selector:
    matchLabels:
      app: ms-tareas
  template:
    metadata:
      labels:
        app: ms-tareas
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "3002"
        prometheus.io/path: "/metrics"
    spec:
      containers:
      - name: ms-tareas
        image: ghcr.io/willrem001-beep/sistema-educativo-ci-cd/
        ms-tareas:latest
```

```
        imagePullPolicy: Always
        ports:
        - containerPort: 3002
        env:
        - name: MONGO_URI
          value: "mongodb://mongo-service:27017/db_tareas"
        volumeMounts:
        - name: uploads-storage
          mountPath: /app/uploads
      volumes:
      - name: uploads-storage
        persistentVolumeClaim:
          claimName: pvc-tareas-uploads
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ms-asistencia
spec:
  replicas: 2
  selector:
    matchLabels:
      app: ms-asistencia
  template:
    metadata:
      labels:
        app: ms-asistencia
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "3004"
```

```
        prometheus.io/path: "/metrics"
    spec:
      containers:
      - name: ms-asistencia
        image: ghcr.io/willrem001-beep/sistema-educativo-ci-cd/ms-asistencia:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 3004
        env:
        - name: MONGO_URI
          value: "mongodb://mongo-service:27017/db_asistencia"
```

**Código de Manifiestos PostgreSQL y MongoDB**

```
    apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
```

```
    image: postgres:13-alpine
    ports:
    - containerPort: 5432
    env:
    - name: POSTGRES_USER
      valueFrom:
        secretKeyRef:
          name: db-secrets
          key: postgres-user
    - name: POSTGRES_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-secrets
          key: postgres-password
    volumeMounts:
    # Montamos sobre la carpeta de inicialización, no sobre el archivo
    - name: init-db
      mountPath: /docker-entrypoint-initdb.d
    # montaje para que los datos se guarden permanentemente
    - name: postgres-storage
      mountPath: /var/lib/postgresql/data
  volumes:
  - name: init-db
    configMap:
      name: postgres-configmap
      #volumen para que los datos no se borren tras un reinicio
  - name: postgres-storage
    persistentVolumeClaim:
      claimName: pvc-postgres-data
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: mongo
        image: mongo:4.4
        ports:
        - containerPort: 27017
        volumeMounts:
          - name: mongo-storage
            mountPath: /data/db
      volumes:
      - name: mongo-storage
        persistentVolumeClaim:
          claimName: pvc-mongo-data
```

**Script de inicialización para crear credenciales, BD, tablas**

```
#reemplazamos las variables
```

```
     $$ BEGIN
   IF NOT EXISTS (
      SELECT FROM pg_roles WHERE rolname = 'user_usuarios'
   ) THEN
      CREATE ROLE user_usuarios LOGIN PASSWORD 'password_usuarios';
   END IF;
END
 $$;
DO
 $$ BEGIN
   IF NOT EXISTS (
      SELECT FROM pg_roles WHERE rolname = 'user_notas'
   ) THEN
      CREATE ROLE user_notas LOGIN PASSWORD 'password_notas';
   END IF;
END
 $$;
DO
 $$ BEGIN
   IF NOT EXISTS (
      SELECT FROM pg_database WHERE datname = 'db_usuarios'
   ) THEN
      CREATE DATABASE db_usuarios OWNER user_usuarios;
   END IF;
END
 $$;


DO
 $$ BEGIN
```

```
   IF NOT EXISTS (
      SELECT FROM pg_database WHERE datname = 'db_notas'
   ) THEN
      CREATE DATABASE db_notas OWNER user_notas;
   END IF;
END
 $$;
-- Asignar privilegios
GRANT ALL PRIVILEGES ON DATABASE db_usuarios TO user_usuarios;
GRANT ALL PRIVILEGES ON DATABASE db_notas TO user_notas;
```

**Configuración para recolección de métricas Prometheus y configuración de Promtail**

```
#Reemplazamos la seccion targets por enlaces de la app
    global:
  scrape_interval: 15s
  evaluation_interval: 15s
  external_labels:
    monitor: 'tesis-final'

scrape_configs:
  # 1. Prometheus (Se auto-monitoriza)
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

  # 2. Microservicio de Usuarios
  # IMPORTANTE: Usa la IP de Minikube y el NodePort (30001)
  - job_name: 'ms-usuarios'
    metrics_path: /metrics
```

```
  static_configs:
    - targets: ['host.docker.internal:56085']
      labels:
        app: 'usuarios'

# 3. Microservicio de Tareas
- job_name: 'ms-tareas'
  metrics_path: /metrics
  static_configs:
    - targets: ['host.docker.internal:56121']
      labels:
        app: 'tareas'

# 4. Microservicio de Notas
- job_name: 'ms-notas'
  metrics_path: /metrics
  static_configs:
    - targets: ['host.docker.internal:56131']
      labels:
        app: 'notas'

# 5. Microservicio de Asistencia
- job_name: 'ms-asistencia'
  metrics_path: /metrics
  static_configs:
    - targets: ['host.docker.internal:56138']
      labels:
        app: 'asistencia'
```

```
   # CADVISOR
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']

  # NODE EXPORTER
  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']

server:
  http_listen_port: 9080
  grpc_listen_port: 0

clients:
  - url: http://loki:3100/loki/api/v1/push

scrape_configs:
  - job_name: docker_logs
    docker_sd_configs:
      - host: unix:///var/run/docker.sock
        refresh_interval: 5s

    relabel_configs:
      # Generar la ruta del archivo de log para Docker
      - source_labels: ['__meta_docker_container_id']
        regex: '(.*)'
        target_label: '__path__'
        replacement: '/var/lib/docker/containers/$1/${1}-json.log'
```

```
      # Etiquetas útiles para Grafana
      - source_labels: ['__meta_docker_container_name']
        regex: '/(.*)'
        target_label: 'container'

      - source_labels: ['__meta_docker_container_label
      _com_docker_compose_service']
        target_label: 'service'

      - source_labels: ['__meta_docker_container_label
      _com_docker_compose_project']
        target_label: 'project'

      - source_labels: ['__meta_docker_container_label
      _com_docker_compose_container_number']
        target_label: 'container_number'
```

**Código para orquestar y desplegar automáticamente todos los servicios y microservicios**

```
services:
  # BASE DE DATOS: POSTGRESQL
  db-postgres:
    image: postgres:13-alpine
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    ports:
      - "5432:5432"
```

```
  volumes:
    - ./init-db-postgres.sql:/docker-entrypoint-initdb.d/init.sql
    - postgres-data:/var/lib/postgresql/data
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 10s
    retries: 5

# BASE DE DATOS: MONGODB
db-mongo:
  image: mongo:4.4
  ports:
    - "27017:27017"
  volumes:
    - mongo-data:/data/db
  healthcheck:
    test: echo 'db.runCommand("ping").ok' | mongo localhost:27017/test --quiet
    interval: 10s
    retries: 5

# MICROSERVICIO 1: USUARIOS
servicio-usuarios:
  build: ./Microserv_Users
  container_name: ms-usuarios
  ports:
    - "3001:3001"
  environment:
    DB_HOST: db-postgres
    DB_PORT: 5432
```

```
    DB_NAME: db_usuarios
    DB_USER: ${DB_USER}
    DB_PASSWORD: ${DB_PASSWORD}
    JWT_SECRET: ${JWT_SECRET}
  depends_on:
    db-postgres:
      condition: service_healthy

# MICROSERVICIO 2: TAREAS
servicio-tareas:
  build: ./Microserv_Tareas
  container_name: ms-tareas
  ports:
    - "3002:3002"
  environment:
    MONGO_URI: mongodb://db-mongo:27017/db_tareas
  depends_on:
    db-mongo:
      condition: service_healthy

# MICROSERVICIO 3: NOTAS
servicio-notas:
  build: ./Microserv_Notas
  container_name: ms-notas
  ports:
    - "3003:3003"
  environment:
    DB_HOST: db-postgres
    DB_PORT: 5432
```

```
    DB_NAME: db_notas
    DB_USER: ${DB_USER}
    DB_PASSWORD: ${NOTAS_DB_PASSWORD}
  depends_on:
    db-postgres:
      condition: service_healthy

# MICROSERVICIO 4: ASISTENCIA
servicio-asistencia:
  build: ./Microserv_Asistencias
  container_name: ms-asistencia
  ports:
    - "3004:3004"
  environment:
    MONGO_URI: mongodb://db-mongo:27017/db_asistencia
  depends_on:
    db-mongo:
      condition: service_healthy

 # LOKI
loki:
  image: grafana/loki:3.0.0
  container_name: loki
  ports:
    - "3100:3100"
  # Usamos el config default interno, no necesitamos montar un archivo externo para
  command: -config.file=/etc/loki/local-config.yaml
  restart: unless-stopped
```

```
# MONITOREO: PROMTAIL
promtail:
  image: grafana/promtail:3.0.0
  container_name: promtail
  volumes:
    - ./promtail-config.yml:/etc/promtail/config.yml:ro
    - /var/lib/docker/containers:/var/lib/docker/containers:ro
    - /var/run/docker.sock:/var/run/docker.sock:ro
  command: -config.file=/etc/promtail/config.yml
  restart: unless-stopped

# MONITOREO: GRAFANA
grafana:
  image: grafana/grafana:11.2.2
  container_name: grafana
  ports:
    - "3000:3000"
  environment:
    - GF_INSTALL_PLUGINS=grafana-lokiexplore-app
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=admin
  volumes:
    - grafana-data:/var/lib/grafana
  depends_on:
    - loki
    - prometheus
  restart: unless-stopped

# MONITOREO: PROMETHEUS
```

```
prometheus:
  image: prom/prometheus:latest
  container_name: prometheus
  ports:
    - "9090:9090"
  volumes:
    - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml:ro
    - prometheus_data:/prometheus
  # Añadimos depends_on para intentar conectar después de que inicien los microservi
  depends_on:
    - servicio-usuarios
    - servicio-tareas
    - servicio-notas
    - servicio-asistencia
  command:
    - '--config.file=/etc/prometheus/prometheus.yml'
    - '--storage.tsdb.path=/prometheus'
    - '--web.console.libraries=/etc/prometheus/console_libraries'
    - '--web.console.templates=/etc/prometheus/consoles'
  restart: unless-stopped
  # MONITOREO: CADVISOR
cadvisor:
  image: gcr.io/cadvisor/cadvisor:latest
  container_name: cadvisor
  ports:
    - "8081:8080"
  privileged: true
  devices:
    - /dev/kmsg
```

```
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker:/var/lib/docker:ro
      - /dev/disk:/dev/disk:ro
    restart: unless-stopped
    # MONITOREO: NODE EXPORTER
  node-exporter:
    image: prom/node-exporter:latest
    container_name: node-exporter
    ports:
      - "9100:9100"
    restart: unless-stopped

volumes:
  postgres-data:
  mongo-data:
  loki-data:
  grafana-data:
  prometheus_data:
```

Este código contiene la configuración dockercompose.yml, en el cual se define la orquestación completa de la arquitectura de microservicios del sistema educativo. El código automatiza el despliegue y gestión de los cuatro servicios principales: usuarios, tareas, notas y asistencia, junto con sus respectivas bases de datos, PostgreSQL y MongoDB, gestionando la persistencia de datos a través de volúmenes y asegurando la disponibilidad mediante healthchecks. Adicionalmente, integra una pila completa de observabilidad y monitoreo compuesta por Prometheus para la recolección de métricas, Loki y Promtail para la centralización de registros (logs), cAdvisor y Node Exporter para el monitoreo de recursos, y Grafana como panel de vi-

sualización unificado, proporcionando así un entorno controlado, replicable y autónomo para el análisis y ejecución del sistema.

### Dashboards: Recolección de métricas

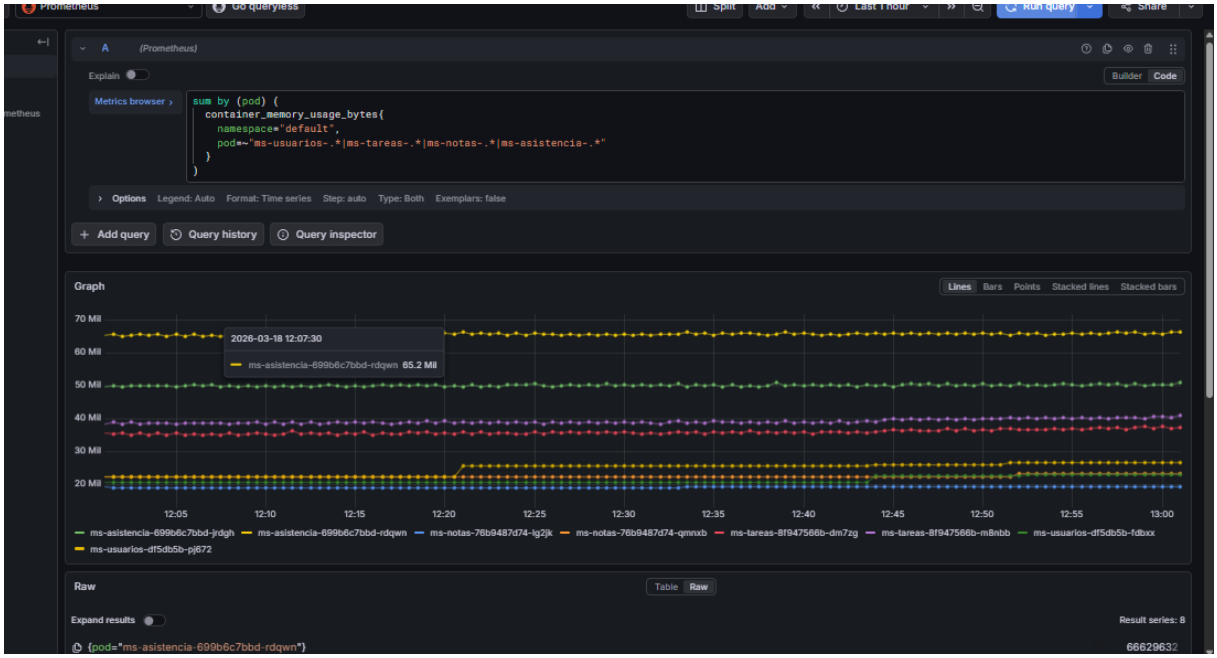



**Figura 36**.
Consumo de memoria inicial. Elaboración propia

Este dashboard hace referencia a los primeros monitoreos desde su despliegue del pipeline. Mostrando que se mantiene constante sin sufrir picos de consumo de memoria.

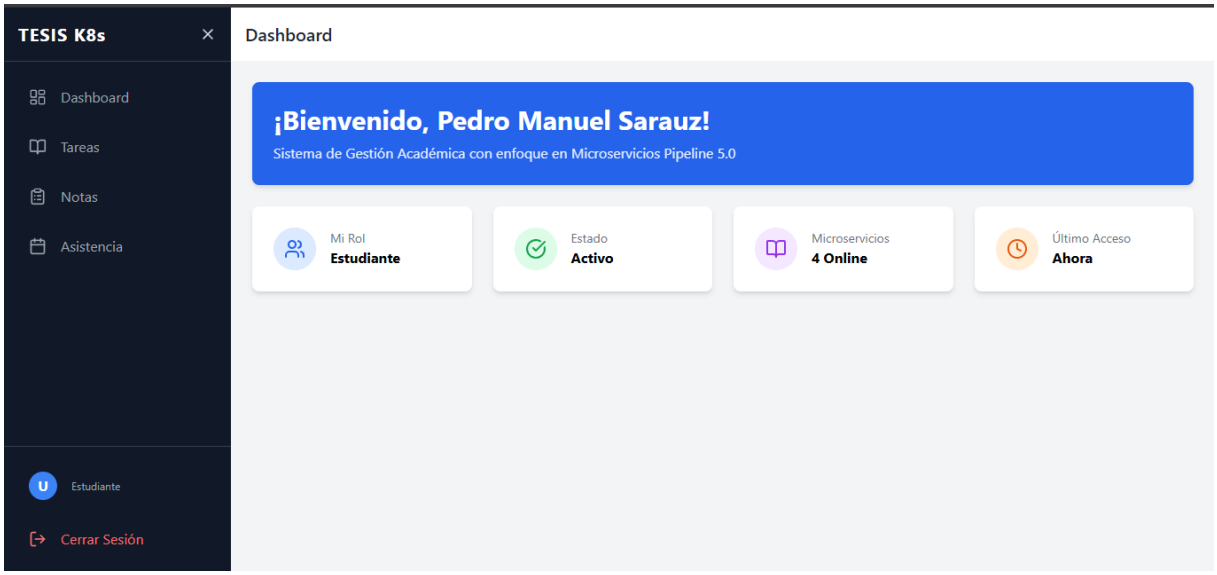



**Figura 37**.
Implementación del frontend. Elabora propia

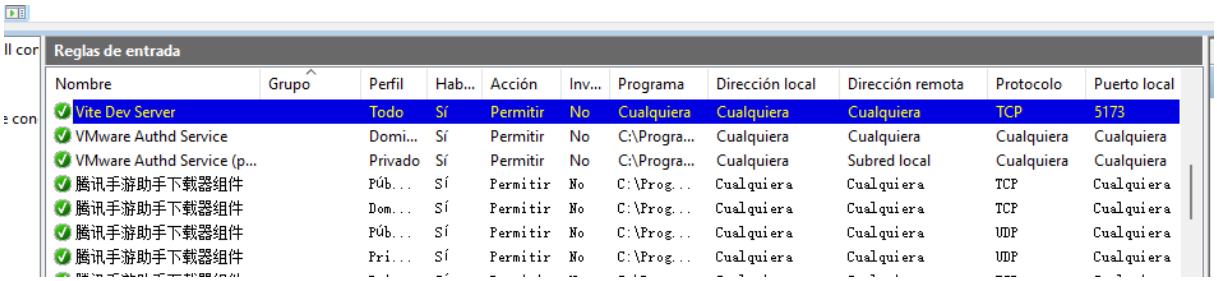



**Figura 38**.
Habilitación de puerto para VITE. Elaboración propia

Para visualizar el frontend, fue necesario habilitar un puerto TCP para VITE que por defecto es 5173. Esto permitió validar URLs y previamente conectar con Prometheus para la recolección de métricas; también se logró hacer pruebas de los endpoints usando Postman.

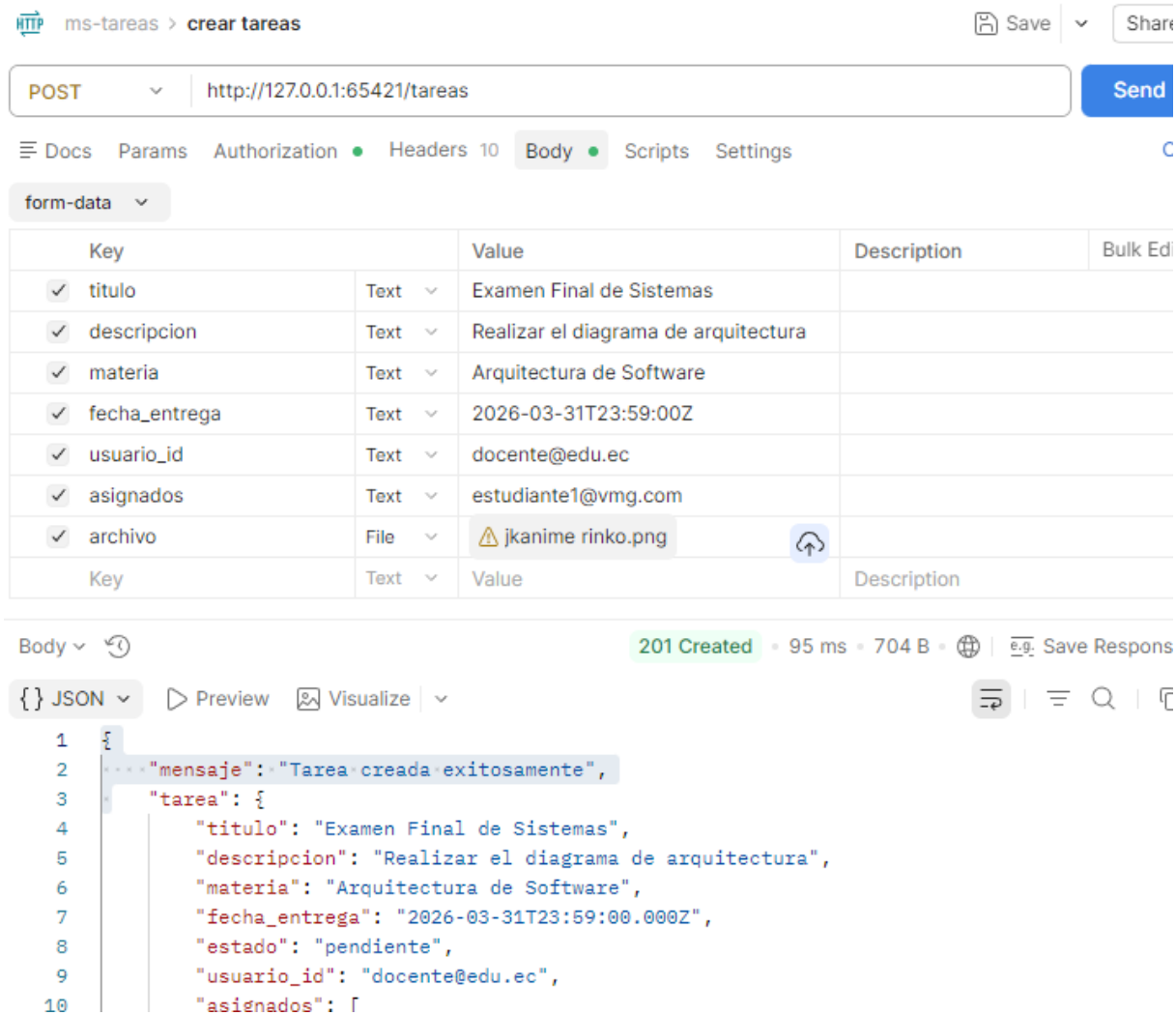



**Figura 39**.
Pruebas de endpoints usando Postman. Elaboración propia.

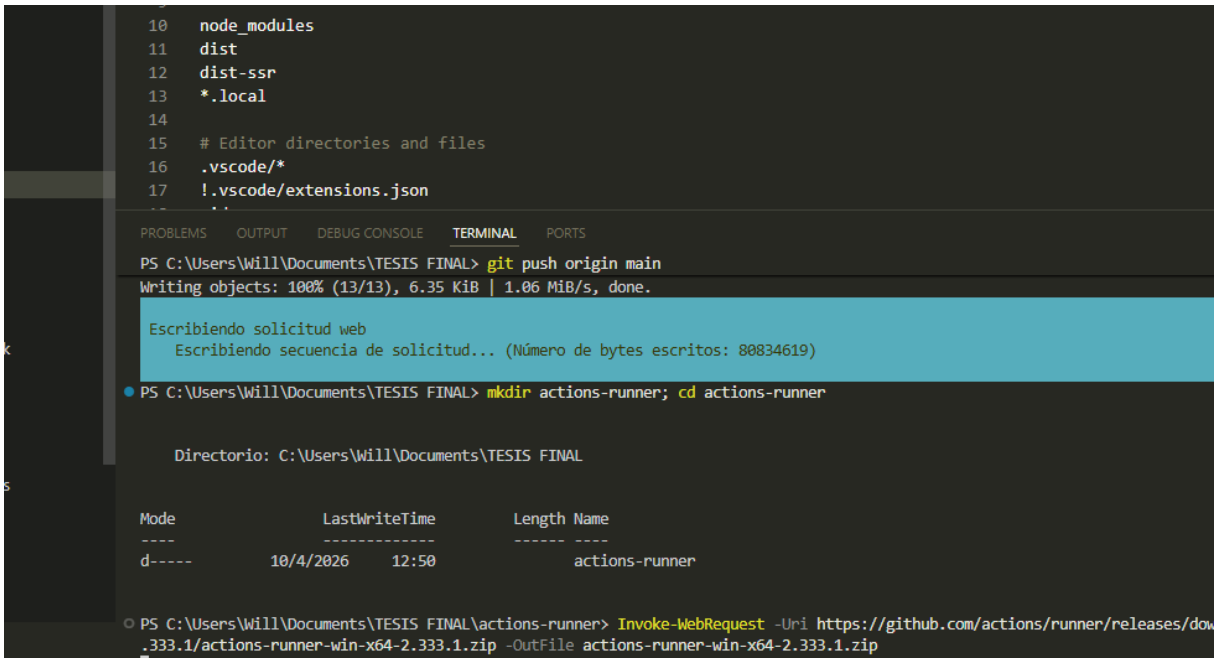



**Figura 40**.
Instalación de Runner para despliegue. Elaboración propia.




**Figura 41**.
Configuración del runner. Elaboración propia.

Para poder automatizar compilaciones, pruebas, empaquetado y despliegue de aplicaciones, especialmente aquellas que requieren un entorno .NET o herramientas específicas de Windows se instaló un Runner de GitHub Actions y posteriormente se realizó la configuración.

```
PS C:\Users\Will> minikube ip
192.168.49.2
PS C:\Users\Will> kubectl get svc my-app-service
NAME             TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
my-app-service   NodePort   10.102.19.240   <none>        80:30080/TCP   15s
PS C:\Users\Will> kubectl get svc my-app-service
NAME             TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
my-app-service   NodePort   10.102.19.240   <none>        80:30080/TCP   31m
PS C:\Users\Will> kubectl get pods -l app=ms-frontend
NAME                           READY   STATUS             RESTARTS   AGE
ms-frontend-748d5fcdd6-qn8gh   0/1     ImagePullBackOff   0          6m19s
ms-frontend-97dc68d96-pbdbg    1/1     Running            0          11m
ms-frontend-97dc68d96-sg4fn    1/1     Running            0          11m
PS C:\Users\Will> kubectl get pods -l app=ms-frontend --show-labels
NAME                           READY   STATUS         RESTARTS   AGE     LABELS
ms-frontend-748d5fcdd6-qn8gh   0/1     ErrImagePull   0          6m26s   app=ms-frontend,pod-template-hash=748d5fcdd6
ms-frontend-97dc68d96-pbdbg    1/1     Running        0          11m     app=ms-frontend,pod-template-hash=97dc68d96
ms-frontend-97dc68d96-sg4fn    1/1     Running        0          11m     app=ms-frontend,pod-template-hash=97dc68d96
```

**Figura 42**.
Error de imagen en el frontend. Elaboración propia.

En la figura 42, se muestra el listado de los Pods del sistema, donde se observa el estado de cada contenedor; algunos aparecen como “running”, mientras que otros muestran el estado ImagePullBackOff, lo que indica un error al intentar descargar la imagen del contenedor correspondiente.




**Figura 43**.
Proceso de memoria residente. Elaboración propia.

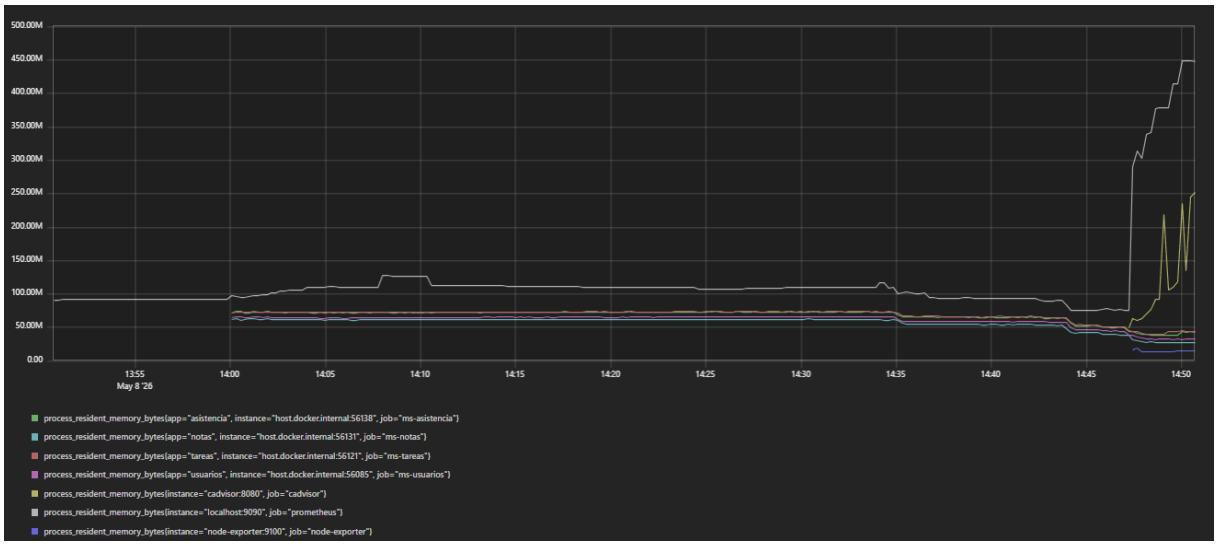



**Figura 44**.
Proceso de memoria residente global. Elaboración propia.

En las figuras 43 y 44 se observa la barra de consulta, la cual calcula la memoria residente de los procesos (process_resident_memory_bytes) filtrando por los microservicios üsuarios", "notas", "tareasz .ªsistencia", y convirtiendo los valores a MB. El gráfico central es un diagrama de líneas de series temporales que muestra la fluctuación de este consumo en el tiempo. Cada línea de color representa una instancia específica de una aplicación identificada en la leyenda con sus etiquetas app y pod, permitiendo comparar el uso de recursos entre los diferentes microservicios durante el periodo mostrado.

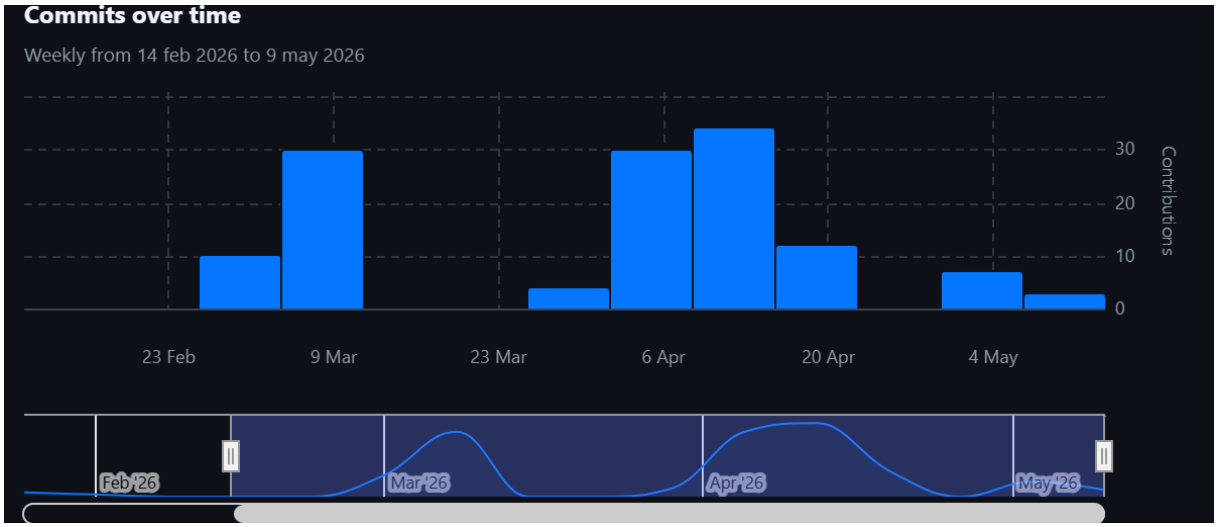



**Figura 45**.
Commits desde Febrero hasta Mayo. Elaboracion propia.

Esta imagen muestra un gráfico de barras que visualiza la actividad de desarrollo de código a lo largo del tiempo. El gráfico cubre el periodo semanal desde febrero de 2026 hasta mayo de 2026. En el eje vertical se indica la cantidad de çontributions”(commits), mientras que el eje horizontal muestra la línea de tiempo con hitos mensuales: Febrero, Marzo, April, Mayo. Las barras azules de diferentes alturas representan el volumen de confirmaciones de código realizadas cada semana, permitiendo identificar picos de actividad y la evolución del trabajo en el proyecto durante ese trimestre.