UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

CARRERA DE TECNOLOGÍAS DE LA INFORMACIÓN

TRABAJO DE INTEGRACIÓN CURRICULAR

TEMA:

" AUTOMATIZACIÓN DE CI/CD PARA MICROSERVICIOS SPRING BOOT EN AWS CON ELASTIC BEANSTALK ENFOCADO EN EL AMBIENTE DE PRUEBAS. "

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:

Cristian Santiago Avila Flores

DIRECTOR:

Msc. Pablo Andrés Landeta López

Ibarra – Ecuador 2025

UNIVERSIDAD TÉCNICA DEL NORTE
BIBLIOTECA UNIVERSITARIA

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			
CÉDULA DE IDENTIDAD:	1725049827		
APELLIDOS Y NOMBRES:	CRISTIAN SANTIAGO AVILA FLORES		
DIRECCIÓN:	QUITO -SAN JOSE DE MINAS		
EMAIL:	csavilaf@utn.edu.ec - csavilaf@gmail.com		
TELÉFONO FIJO:	0986826471	**TELÉFONO MÓVIL:**	0963738659

DATOS DE LA OBRA	
TÍTULO:	AUTOMATIZACIÓN DE CI/CD PARA MICROSERVICIOS SPRING BOOT EN AWS CON ELASTIC BEANSTALK ENFOCADO EN EL AMBIENTE DE PRUEBAS
AUTOR (ES):	CRISTIAN SANTIAGO AVILA FLORES
FECHA: DD/MM/AAAA	09/05/2025
SOLO PARA TRABAJOS DE GRADO	
PROGRAMA:	☑ **PREGRADO** ☐ **POSGRADO**
TITULO POR EL QUE OPTA:	**INGENIERO EN TECNOLOGÍAS DE LA INFORMACIÓN.**
ASESOR /DIRECTOR:	**MSC. PABLO ANDRÉS LANDETA LÓPEZ**

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 05 días del mes de septiembre de 2025

EL AUTOR:

(Firma).....
Nombre: Cristian Santiago Avila Flores

CERTIFICACIÓN DEL DIRECTOR DEL TRABAJO DE INTEGRACIÓN CURRICULAR

Ibarra a las 04 días del mes de septiembre de 2025

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.

MSc. Pablo Andrés Landeta López

C.C: 1002161055

APROBACIÓN DEL COMITÉ CALIFICADOR

El Comité Calificado del trabajo de Integración Curricular " Automatización de CI/CD para Microservicios Spring Boot en AWS con Elastic Beanstalk Enfocado en el ambiente de Pruebas. " elaborado por Cristian Santiago Avila Flores , 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:

MSc. Pablo Andrés Landeta López

C.C: 1002161055

MSc. Trejo España Diego Javier

C.C: 1002149290

DEDICATORIA

Con gran gratitud y emoción, dedico esta tesis, titulada ”Automatización de CI/CD para Microservicios Spring Boot en AWS con Elastic Beanstalk Enfocado en el Ambiente de Pruebas”, a cada una de las personas que han estado con migo en este proceso tan largo.

En primer lugar, dedico este logro a mi hijo, Mayquel Gabriel, quien es la mayor inspiración de mi vida.

A a mis padres, Cecilia Flores y Pablo Ávila, su acompañamiento absoluto y guía han ejecutado un rol fundamental en mi formación.

Por otra parte, este éxito adquiere dimensión colectiva al compartirlo con mi hermano Fernando Ávila. Sus orientaciones y fe inquebrantable han constituido un soporte estructural para cada decisión.

De igual manera, extiendo este reconocimiento al núcleo familiar , cuya presencia constante ha operado como variable significativa en el proceso.

Cristian Santiago Avila Flores

AGRADECIMIENTO

Bueno, aquí estamos al fin. Después de tantas noches sin dormir y cafés fríos, puedo decir que esta tesis sobre automatización en AWS ”Automatización de CI/CD para Microservicios Spring Boot” (sí, el título es un trabalenguas).

Y mira, no hubiera sido posible sin la ayuda de un montón de gente increíble.

¿Quién diría que llegar hasta aquí sería tan difícil, pero a la vez tan gratificante?

O sea, primero que nada, tengo que agradecer a mi director, el MSc. Pablo López Landeta.

Este hombre tuvo la paciencia y eso que yo le debo haber sacado más de una cana con mis dudas de última hora.

Recuerdo cuando me atoré con la configuración de Elastic Beanstalk.. Bueno, sin su experiencia y esos consejos que parecían , no habría salido adelante. ¡Gracias por no dejarme botado!

A mis profesores, ¿qué puedo decir? Ustedes fueron los que me enseñaron desde lo básico hasta esos trucos que solo se aprenden en la práctica.

Desde mi experiencia, son detalles que uno no olvida.

Ahora que miro atrás, hasta los errores como aquel deploy que borró medio ambiente de pruebas y que terminaron siendo lecciones el cual en este me llenan de gran satisfacción.

Así que, a todos los que estuvieron en este camino: ¡mil gracias! Sin ustedes, esta meta sería solo un sueño.

Cristian Santiago Avila Flores

ÍNDICE DE CONTENIDOS

List of Figures

List of Tables

RESUMEN EJECUTIVO

Esta investigación analiza la automatización de procesos CI/CD para microservicios desarrollados con Spring Boot, implementando la entrega continua mediante AWS Elastic Beanstalk en entornos de validación. El estudio se centra en la construcción de un flujo de trabajo automatizado que integre, valide y despliegue microservicios basados en Spring Boot, utilizando Elastic Beanstalk como plataforma de implementación dentro de entornos de prueba. La solución busca optimizar los ciclos de desarrollo, mejorar la calidad del software y minimizar las fallas derivadas de la intervención manual.

Jenkins se emplea como la herramienta principal para configurar y gestionar el pipeline, incorporando validaciones automatizadas que verifican la funcionalidad y estabilidad de cada microservicio antes de su implementación. El estudio también examina los beneficios de la automatización de CI/CD en términos de escalabilidad, mantenibilidad y consistencia del desarrollo de software en arquitecturas distribuidas.

La investigación genera un estudio de caso replicable para organizaciones que buscan implementar procesos automatizados de CI/CD en ecosistemas de microservicios, evidenciando las ventajas de integrar tecnologías contemporáneas como Spring Boot y AWS Elastic Beanstalk.

Palabras clave: Microservicios, AWS Elastic Beanstalk, DevOps, Pipeline

ABSTRACT

This research investigates CI/CD process automation for microservices developed with Spring Boot, implementing continuous deployment through AWS Elastic Beanstalk in validation environments. The study focuses on building an automated workflow that integrates, validates, and deploys Spring Boot-based microservices, utilizing Elastic Beanstalk as implementation platform within testing environments. The solution seeks to optimize development cycles, enhance software quality, and minimize failures derived from manual intervention. Jenkins serves as the primary tool for configuring and managing the pipeline, incorporating automated validations that verify functionality and stability of each microservice prior to implementation. The study also examines CI/CD automation benefits in scalability, maintainability, and consistency of software development for distributed architectures. The research generates a replicable case study for organizations seeking to implement automated CI/CD processes in microservice ecosystems, evidencing the advantages of integrating contemporary technologies such as Spring Boot and AWS Elastic Beanstalk.

Keywords: Microservicios, AWS Elastic Beanstalk, DevOps, Pipeline

CAPÍTULO I

INTRODUCCIÓN

1.1 Planteamiento del Problema

En la actualidad las tendencias tecnológicas y las organizaciones están migrando hacia arquitecturas basadas en microservicios con el objetivo de construir sistemas más escalables, adaptables y sencillos de mantener[1].

No obstante, esta aproximación conlleva complejidades consustanciales que requieren atención meticulosa y constituye un factor de alto impacto por la ausencia de mecanismos robustos para validar la calidad dado que despliegues con fallos no detectados podrían comprometer la operatividad global de la plataforma.

Durante las pruebas que realicé, noté una mejora importante en el rendimiento general del sistema, sobre todo cuando se trabajaba con muchas solicitudes al mismo tiempo; sin embargo, ese avance también trajo sus propios retos.

Algo que me quedó claro es que, al usar microservicios, todo queda muy conectado; si no se gestionan bien las actualizaciones, por ejemplo usando versionado semántico es fácil que un cambio en un servicio afecte a los demás[2].

En lo personal, trabajar con herramientas CI/CD en AWS Elastic Beanstalk fue más complicado de lo que pensé al inicio.

Aunque lograr una mayor eficiencia es sin duda una ventaja importante, también trae consigo ciertos desafíos, algo que me llamó bastante la atención durante el desarrollo fue la fuerte dependencia entre los microservicios.

Esto me llevó a implementar protocolos bastante estrictos para gestionar las actualizaciones, como el versionado semántico que resulta fundamental para evitar errores o conflictos entre servicios que están conectados entre sí [2].

Cuando empecé a trabajar con herramientas de automatización, especialmente al intentar integrarlas con plataformas como AWS Elastic Beanstalk, me di cuenta de que no era una tarea tan sencilla como pensaba; no se trata solo de instalar y configurar, sino que se necesita tener muy claro cómo funciona cada herramienta y cómo se relacionan.

El reto técnico me exige no solo conocimientos específicos, sino también una buena dosis de planificación y pruebas constantes, también entendí que al conocer a todo y cada uno de los componentes que son parte del proceso CI/CD es clave; esto ayuda bastante a reducir errores y a optimizar los tiempos de entrega [3].

De hecho muchas veces el simple hecho de no entender bien una herramienta puede generar retrasos o fallos que se podrían haber evitado [4].

La conexión entre microservicios también obliga a ser muy preciso con las actualizaciones, cualquier pequeño cambio en uno puede tener consecuencias en otros, así que se vuelve fundamental contar con estrategias claras para garantizar la coherencia del sistema [2].

En mi experiencia, la integración de herramientas CI/CD con AWS Elastic Beanstalk demanda más que conocimientos técnicos. Requiere atención al detalle, una configuración cuidadosa y sobre todo muchas pruebas, incluso un error mínimo puede afectar el funcionamiento del sistema en producción.

Por eso, para que todo fluya correctamente, es imprescindible entender bien cómo se conectan e interactúan todas las herramientas dentro del pipeline. Solo así es posible minimizar fallos y mantener la eficiencia a lo largo del desarrollo [4].

1.2 Preguntas De Investigación

- ¿Qué componentes y flujos de trabajo debe incluir un pipeline de CI/CD optimizado para la integración, prueba y despliegue automatizado de microservicios Spring Boot en AWS Elastic Beanstalk?
- ¿Cómo influye la elección entre herramientas de CI/CD como GitHub o Jenkins en la eficiencia, confiabilidad y escalabilidad del proceso de despliegue de microservicios en plataformas como AWS Elastic Beanstalk?
- ¿Qué ventajas ofrece la implementación de entornos de pruebas automatizados frente a los manuales en el desarrollo de microservicios, especialmente en términos de velocidad, detección de errores y consistencia en despliegues?

1.3 Objetivos

1.3.1 Objetivo General

La automatización de procesos de integración y despliegue continuos, al construir una arquitectura eficiente para microservicios basados en Spring Boot requiere una planificación minuciosa[5].

1.3.2 Objetivos Específicos

- Construir un pipeline de CI/CD automatizado que integre las fases de compilación, análisis y estático con pruebas unitarias y despliegue de controladores de microservicios Spring Boot en AWS Elastic Beanstalk, garantizando un flujo eficiente desde el repositorio hasta el entorno productivo.
- Una parte importante del proyecto fue montar un entorno de pruebas automatizado que me permitiera asegurar que cada microservicio funcionara bien por separado y también

en conjunto.

- Verificar tiempos de despliegue

1.4 Alcance y delimitación

Durante el desarrollo de mi tesis me di cuenta de que mejorar la eficiencia en los procesos de desarrollo de software no es tan simple como parece.

Surgen varios retos técnicos en el camino, y uno de los más complicados para mí fue lograr implementar un sistema de integración y despliegue continuo que funcionara de forma completamente automatizada, no solo implicó entender bien cada herramienta, sino también asegurarme de que todas trabajaran juntas sin errores.

Esta combinación tecnológica me presentó obstáculos significativos que requerían soluciones especializadas.

Cuando empecé con el tema de mi tesis, honestamente pensé que optimizar el desarrollo de software iba a ser más directo. La cosa se complicó rápido.

Implementar CI/CD desde cero no es solo instalar Jenkins tuve que aprender que cada herramienta tiene sus mañas, y hacer que trabajen juntas... bueno, eso ya es otro tema completamente diferente.

En la Universidad Técnica del Norte nos habían dado las bases teóricas, pero la práctica resultó ser mucho más compleja de lo que esperaba.

Mientras avanzaba en la investigación, me di cuenta de que optimizar los ciclos de desarrollo de software no es una tarea sencilla.

Lo que más me costó y aquí fue donde realmente me di cuenta de la magnitud del proyecto) fue conectar Jenkins con AWS Elastic Beanstalk.

Recuerdo pasar tardes enteras en el laboratorio intentando que la configuración funcionara.

El problema no era solo técnico aunque configurar el pipeline sí que me dio dolores de cabeza.

Era más bien que cada vez que creía tener todo controlado, aparecía algo nuevo. Un día los despliegues funcionaban perfecto, al siguiente fallaban sin explicación aparente.

Tuve que replantear todo mi enfoque. No bastaba con seguir tutoriales online; necesitaba entender realmente cómo funcionaba cada componente del ecosistema CI/CD.

Fue un proceso de prueba y error bastante frustrante al principio, pero gradualmente empecé a ver los patrones. Aparecen varios desafíos en el camino, y uno de los que más destacó en mi caso fue la necesidad de construir un flujo de CI/CD que estuviera completamente automatizado.

Fue ahí donde comprendí lo complejo que puede ser integrar todos los componentes sin que haya fallos en el proceso.

Lo más importante que aprendí fue que la automatización real no es solo conectar herramientas. Es crear un sistema donde cada parte sepa exactamente qué hacer sin que tengas que estar supervisando constantemente. Y créeme, llegar a ese punto requirió muchas horas de debugging y ajustes finos en la configuración.

1.5 Justificación

Para ser honesto, cuando empecé a leer sobre este tema, no esperaba encontrar tanta información fragmentada.

Revisé papers sobre CI/CD, y algo que me llamó mucho la atención fue que casi todos hablaban de los beneficios, pero pocos mostraban implementaciones reales con números concretos.

El trabajo de Rodríguez et al. sobre arquitectura basada en microservicios y DevOps[6]

fue uno de los que más me ayudó a entender el panorama actual. Pero incluso ahí, cuando buscaba ejemplos específicos de Spring Boot con AWS Elastic Beanstalk, las referencias eran limitadas[7].

Los equipos de desarrollo pueden integrar cambios de código continuamente gracias a un pipeline de CI/CD automatizado. Cada modificación y esto es clave se prueba y despliega rápidamente. El resultado: menos tiempo para llegar al mercado.

Todos los desarrolladores trabajan en el mismo entorno estandarizado eso es lo que garantiza la integración continua. Jenkins automatiza pruebas y despliegues.

Cada microservicio sigue procesos uniformes, lo cual es fundamental porque estos servicios necesitan interactuar de forma coherente entre sí.

Menos intervención manual significa menos errores humanos - eso logra la automatización de CI/CD. Las pruebas automatizadas detectan problemas antes de producción[7].

AWS Elastic Beanstalk ofrece una plataforma escalable para microservicios. Las organizaciones escalan aplicaciones eficientemente cuando combinan CI/CD con Elastic Beanstalk, adaptándose así a mercados cambiantes[8].

Este proyecto mejora procesos internos de desarrollo. También alinea la organización con mejores prácticas industriales.

Este proyecto funciona como caso de estudio. Otras organizaciones pueden replicarlo y adaptarlo para implementar CI/CD en microservicios,la documentación y resultados que aportan lecciones y valiosas e mejores prácticas compartibles con la comunidad de desarrollo[6].

CI/CD para microservicios Spring Boot en AWS Elastic Beanstalk optimiza flujos de trabajo y mejora calidad del software. Reduce errores humanos también facilita transformación digital organizacional. El proyecto constituye inversión estratégica en eficiencia y competitividad a largo plazo.

CAPÍTULO II

MARCO TEÓRICO

2.1 Antecedentes

Automatización de pruebas en CI y desarrollo de software el objetivo de estudio en diversas disciplinas e ingeniería de software, gestión de proyectos, calidad del software.

Marco teórico se fundamenta en investigaciones previas se exploró relación entre madurez de automatización de pruebas, calidad del producto y ciclo de lanzamiento.

Ingeniería de software, gestión de proyectos, calidad del software. Este marco teórico usa investigaciones previas. Exploran relaciones entre madurez de automatización de pruebas, calidad del producto y ciclos de lanzamiento. Pruebas automatizadas mejoran calidad del software y reducen tiempo de lanzamiento en la premisa base de la automatización de pruebas. Hilton et al. (2022) reportan que el 60% de costos CI van a desarrollo y ejecución de automatización de pruebas, esto subraya la importancia de optimizar procesos para mejorar eficiencia en el ciclo de vida del desarrollo[2].

Además, la literatura sugiere que la madurez de la automatización de pruebas está relacionada con la calidad del producto, llevaron a cabo estudios recientes que verifican el impacto positivo de la madurez de la automatización de pruebas en la calidad del software, lo que proporciona un fundamento teórico sólido para esta investigación[2].

Varios estudios abordan la relación automatización de pruebas en calidad del software,Jenkins[9] analizó cómo la madurez de automatización afecta calidad en proyectos de código abierto, usaron minería de repositorios de software, esta metodología ha sido poco utilizada antes, resaltando la novedad del enfoque.

Otro estudio investiga impacto de automatización de pruebas en ciclos de lanzamiento de mayor madurez en automatización de pruebas ciclos más cortos y eficientes y cruciales para CI.

El trabajo incluye visión histórica evolución de automatización de pruebas e impacto en calidad del software con la cual establece contexto para entender desarrollo de prácticas actuales desde enfoques anteriores.

Definición de variables de estudio.

A continuación, se presentan las definiciones relacionadas con las variables de estudio:

- Madurez de automatización de pruebas: Nivel de implementación y efectividad de pruebas automatizadas en proyectos de software.
- Calidad del producto: Grado en que el software cumple requisitos y expectativas de usuarios. Se mide con métricas como cantidad de defectos y satisfacción del cliente.
- Esfuerzo de automatización de pruebas: Combina esfuerzo de desarrollo y ejecución de pruebas automatizadas.
- Ciclo de lanzamiento: Tiempo desde inicio del desarrollo hasta entrega del producto final. Variable crucial para evaluar eficiencia de procesos CI y su relación con madurez de automatización[2].

2.2 Bases teóricas

2.2.1 Integración Continua (CI) y Entrega Continua (CD)

Prácticas de mejora de software para aumentar la calidad y acelerar entrega a usuarios finales. Se centran en automatizar procesos desarrollo, prueba y despliegue[10]

2.2.2 Integración Continua (CI)

Integración Continua fusiona cambios de código frecuentemente en repositorio compartido.

Cada cambio se verifica con pruebas automatizadas en la detecta errores temprano en el ciclo de desarrollo y mantiene calidad del software y reduce tiempo para integrar nuevas características[3],[11].

2.2.3 Objetivos de CI

Integración Continua (CI) automatiza y mejora desarrollo de software en la cual la integración de frecuentes de cambios en código metas específicas[8]:

1. Detectar errores rápidamente
2. Asegurar código siempre funcional
3. Facilitar colaboración entre desarrolladores

2.2.4 Entrega Continua (CD)

Entrega Continua extiende la integración Continua permite despliegue automático en producción después de pasar las pruebas a cualquier cambio que pase en las pruebas pueden liberarse a producción en cualquier momento y reduce tiempo de entrega y mejora respuesta a necesidades del cliente.

2.2.5 Objetivos de CD

Despliegue Continuo garantiza procesos automáticos y mínimamente manuales. Reduce errores durante lanzamiento de nuevas versiones. Asegura código validado correctamente antes

de producción[12]. Implementar cambios graduales y frecuentes reduce riesgo de grandes lanzamientos. Permite mejor gestión de incidentes y despliegues más controlados[4].

1. Automatizar proceso de despliegue
2. Asegurar software siempre listo para liberación
3. Minimizar riesgo de despliegue de nuevas versiones

2.2.6 Proceso De CI/CD

La Integración Continua (IC) es un proceso fundamental en el desarrollo ágil de software que busca mejorar la calidad del código y reducir el tiempo necesario para realizar cambios. El proceso comienza con la compilación del código, donde se transforma el código fuente en un artefacto ejecutable. Crucial porque asegura construcción correcta del código. Identifica errores de sintaxis en etapas tempranas[12].

Despliegue Continuo (DC) se basa en IC. Permite procesos de despliegue automatizados y mínimamente manuales. Código validado en entorno de pruebas se despliega automáticamente en producción o entorno simulado. Reduce riesgo de errores humanos[7].

Después del despliegue se realizan pruebas de aceptación en producción. Confirman funcionamiento correcto del software en entorno real[13].

Proceso CI/CD incluye estas etapas[14]:

1. Desarrollo: Desarrolladores realizan cambios en código y los envían al repositorio.
2. Construcción: Compilación del código y generación de artefactos (archivos ejecutables).
3. Pruebas: Ejecución de pruebas automatizadas para verificar funcionamiento del código.
4. Despliegue: Código se despliega en producción o prueba si las pruebas son exitosas.

5. Monitoreo: Seguimiento del rendimiento del software en producción para detectar problemas.

2.2.6.1 Herramientas comunes para CI/CD

Herramientas comunes para CI y CD aparecen en varios documentos. Destacan su importancia en automatización de procesos de desarrollo de software.

Un documento sobre automatización con Jenkins expone que Integración Continua aporta retroalimentación casi instantánea sobre cada entrega de código al repositorio. Automatiza proceso de construcción del software. Identifica errores de integración en fases tempranas. Incrementa calidad del software entregado[8].

Existe variedad de herramientas CI/CD para gestionar y orquestar estos procesos - Jenkins, GitLab CI, Travis CI.

Estas herramientas permiten compilar, probar y desplegar código eficientemente[4].

Facilitan integración de cambios de código y automatización de pruebas y despliegues. Mejora colaboración y calidad del producto final[15].

Varias herramientas facilitan implementación de CI/CD:

1. JeJenkins: Herramienta popular para automatización CI/CD. Permite crear pipelines para gestionar flujo de trabajo de desarrollo[7],[7],[9].
2. GitLab CI: Integrado en GitLab - automatiza pruebas y despliegues.
3. GitHub Actions: Define flujos de trabajo automatizados directamente en repositorios GitHub[16].
4. Servicio CI: Se integra con GitHub para ejecutar pruebas automáticamente[7].

2.3 Aplicaciones de CI/CD en arquitecturas de microservicios.

Despliegue continuo agiliza tiempo de comercialización y elimina desfase entre codificación y valor al cliente. Automatiza pruebas de regresión y reduce intervenciones manuales.

Mejora eficiencia del proceso de entrega de software. Minimiza errores. Facilita comunicación más efectiva entre desarrolladores.

Integración continua aumenta productividad del equipo - reduce tareas manuales. Asegura calidad constante con pruebas frecuentes. Posibilita entregas más rápidas a clientes. Favorece monitorización constante de métricas de calidad del proyecto. Resulta en menores costos y mejor colaboración entre equipos[13],[7].

Integración continua es práctica esencial en desarrollo de software - mejora calidad y eficiencia del proceso. Permite integrar código frecuentemente y de forma automatizada. Esta metodología libera desarrolladores de tareas manuales. Detecta errores de integración tempranamente. Reduce problemas a largo plazo significativamente[17].

Integración continua facilita pruebas más frecuentes - asegura calidad constante del código. Permite actualizaciones más rápidas para clientes, quienes reciben nuevas funciones en tiempo mínimo. Promueve mejor cooperación entre equipos involucrados.

Contribuye a flujo de trabajo más ágil y eficiente[8].

2.4 Comparación de herramientas de CI/CD: Jenkins, GitHub y AWS

2.4.1 Microservicios

Definición:

Los microservicios son un estilo arquitectónico que estructura una aplicación como un con-

junto de servicios pequeños y autónomos, cada uno de los cuales se ejecuta en su propio proceso y se comunica a través de interfaces bien definidas, generalmente utilizando protocolos ligeros como HTTP/REST o mensajería **cortellessa2022model**.

2.4.2 Objetivos de los Microservicios

Los objetivos de los microservicios son proporcionar un estilo arquitectónico que facilite el desarrollo de aplicaciones mediante un conjunto de servicios independientes, escalables y colaborativos. Esta arquitectura busca mejorar la flexibilidad y la adaptabilidad de las aplicaciones en ecosistemas complejos, permitiendo que cada microservicio sea desarrollado, implementado y escalado de manera independiente, sin afectar al sistema en su conjunto[18][1].

Además fomenta el uso de tecnologías y herramientas específicas que pueden ser optimizadas para cada servicio, resultando en una mayor eficiencia y un tiempo de respuesta más rápido ante los cambios del mercado y las necesidades de los usuarios", para mayor naturalidad [6],[1].

2.4.3 Características de los microservicios

Los microservicios dividen aplicaciones en servicios independientes y pequeños que facilitan desarrollo ágil e implementación aislada de cada componente.

La arquitectura modular permite escalado independiente donde cada microservicio se ajusta según demanda específica.

Además, los microservicios promueven la flexibilidad en el uso de diferentes tecnologías y lenguajes de programación, permitiendo a los equipos elegir las herramientas más adecuadas para cada microservicio. Otra característica clave es la comunicación a través de interfaces ligeras, que mantiene la independencia entre los servicios mientras asegura una interacción efectiva[1].

Microservicios suelen ser resilientes - diseñados para manejar fallos eficientemente. Fomentan automatización de pruebas y despliegues. Mejora calidad y continuidad del servicio[19],[6].

1. Descomposición: Aplicaciones se dividen en servicios más pequeños. Desarrollados, desplegados y escalados independientemente[2].
2. Autonomía: Cada microservicio maneja funcionalidad específica. Se desarrolla en diferentes lenguajes y tecnologías[13].
3. Escalabilidad: Permite escalar partes de aplicación independientemente - mejora eficiencia en uso de recursos[3].
4. Resiliencia: Falla de un microservicio no afecta toda la aplicación. Mejora disponibilidad general del sistema[20].
5. Despliegue independiente: Equipos despliegan cambios en microservicio sin desplegar toda la aplicación[21].

2.4.4 Ventajas de los microservicios

Microservicios ofrecen ventajas clave, mejoran eficiencia y flexibilidad de aplicaciones. Permiten escalabilidad independiente de partes de aplicación. Optimiza uso de recursos y mejora rendimiento.

Cada microservicio se despliega y actualiza autónomamente. Facilita implementación de nuevas características sin afectar sistema completo, su arquitectura permite usar diferentes tecnologías para cada componente mayor flexibilidad.

Contribuyen a resiliencia - falla de un microservicio no impacta necesariamente a los demás. Permiten desarrollo más ágil porque equipos trabajan en paralelo. Estos sistemas son más fáciles de mantener. Se adaptan mejor a cambios en negocio o entorno tecnológico [3].

1. Escalabilidad: Microservicios escalan independientemente cada componente según demanda. Optimiza uso de recursos y mejora rendimiento general[6].

2. Mantenibilidad: Servicios más pequeños son más fáciles de entender, mantener y actualizar. Reduce complejidad y facilita gestión del código[13].

3. Resiliencia: Arquitectura permite que falla de un servicio no afecte toda la aplicación. Mejora disponibilidad y confiabilidad del sistema[19].

2.4.5 Desafíos de los microservicios

Los microservicios presentan varios desafíos, como la complejidad en la gestión y e monitoreo, ya que coordinar múltiples servicios puede requerir herramientas avanzadas. Esto se debe a que, en una arquitectura de microservicios, cada componente (o microservicio) opera de manera independiente, pero debe interactuar con otros, lo que complica la supervisión de su rendimiento y la detección de problemas[18].

La comunicación entre microservicios puede introducir latencias, ya que estos dependen de la red para intercambiar datos, y cualquier fallo en una de las conexiones puede afectar el funcionamiento general de la aplicación. Además, gestionar las dependencias entre microservicios es crítico; si un servicio cambia, puede requerir que otros servicios que dependen de él también se actualicen, lo que aumenta la complejidad en el proceso de despliegue y en la garantía de compatibilidad.

En resumen, aunque los microservicios ofrecen muchas ventajas, su implementación y mantenimiento eficaz demandan un enfoque cuidadoso y el uso de herramientas especializadas para gestionar esta complejidad[19],[4],[22].

1. Complejidad en la gestión: La orquestación y gestión de múltiples microservicios puede ser compleja, ya que cada servicio debe ser monitoreado y administrado de manera independiente[23].

2. Comunicación entre servicios: Interacción entre microservicios introduce latencias y puntos de fallo. Establecer protocolos de comunicación y manejar errores de red complica diseño del sistema [24].

3. Pruebas y monitoreo: Requiere enfoque diferente comparado con aplicaciones monolíticas. Pruebas de integración y extremo a extremo son más complejas por naturaleza distribuida de microservicios[25].

Estos desafíos requieren enfoque cuidadoso en diseño, desarrollo y operación de sistemas basados en microservicios. Maximiza beneficios y minimiza riesgos asociados.

2.4.6 Modelado de arquitecturas de microservicios

Modelado de arquitecturas de microservicios implica diseñar sistemas de múltiples servicios independientes que se comunican entre sí. Cada microservicio maneja funcionalidad específica se desarrolla, despliega y escala autónomamente[21][1]:

Aspectos clave del modelado:

1. Descomposición de servicios: Identificar y definir microservicios basados en dominios de negocio, funcionalidades o capacidades específicas. Implica analizar dominio del problema y dividirlo en servicios independientes, desarrollados y desplegados autónomamente.

2. Interacción entre servicios: Definir comunicación entre microservicios. Incluye APIs REST, mensajería asíncrona (RabbitMQ, Kafka) y otros protocolos. Crucial establecer contratos claros para interacción entre servicios.

3. Gestión de datos: Decidir manejo de datos en entorno de microservicios. Incluye elección de bases de datos específicas para cada microservicio o enfoque de base compartida. Consistencia de datos y gestión de transacciones distribuidas son consideraciones importantes.

Modelado efectivo de arquitecturas de microservicios es crucial para éxito en implementación y operación de sistemas distribuidos[26].

2.4.7 Spring Boot

Spring Boot es marco de desarrollo Java para simplificar creación de aplicaciones. Ofrece configuración automática y permite ejecución autónoma de aplicaciones empaquetadas como archivos JAR que facilita la implementación[5].

Su diseño es especialmente adecuado para arquitecturas de microservicios, facilita creación de servicios RESTful, permite integración fluida con otras herramientas del ecosistema Spring boot[27].

Este subsistema habilita monitorización y gestión de aplicaciones en producción. Usa end-points que proporcionan información sobre salud y rendimiento[7].

Spring Boot fomenta entorno de desarrollo ágil que promueve convenciones en lugar de configuraciones detalladas. Esto lo convierte en elección popular entre desarrolladores[5].

1. Configuración simplificada: Spring Boot elimina necesidad de configuraciones extensas y complicadas. Usa convenciones sobre configuraciones predeterminadas funcionales en mayoría de casos. Permite a desarrolladores concentrarse en lógica de negocio. Se logra mediante configuración automática que detecta dependencias en classpath y configura beans necesarios automáticamente.

2. Desarrollo rápido: Con Spring Boot, desarrolladores inician proyectos rápidamente gracias a enfoque en productividad. Proporciona ”starters” y conjuntos de dependencias preconfiguradas para integrar funcionalidades como acceso a bases de datos, seguridad, mensajería. Acelera proceso de desarrollo y reduce tiempo de configuración[16].

3. Microservicios: Spring Boot es especialmente adecuado para desarrollo de microservicios. Permite crear aplicaciones independientes desplegables y escalables autónoma-

mente. Cada microservicio se desarrolla como aplicación Spring Boot que facilita la implementación de arquitecturas distribuidas[7].

Spring Boot es marco poderoso y versátil para crear aplicaciones Java rápida y eficientemente. Aprovecha ventajas de arquitectura moderna y facilita desarrollo de microservicios[1]. Su enfoque en simplicidad y productividad lo convierte en opción popular entre desarrolladores Java[5].

2.4.8 Beneficios de Spring Boot

Spring Boot ofrece múltiples beneficios, elección preferida para desarrollo de aplicaciones Java, especialmente en arquitecturas de microservicios.

Principales beneficios[5]:

1. Desarrollo rápido: Spring Boot permite iniciar proyectos rápidamente gracias a ”starters” y colecciones de dependencias preconfiguradas. Facilita integración de funcionalidades como acceso a bases de datos y seguridad. Acelera proceso de desarrollo y reduce tiempo de configuración.

2. Facilidad de uso: Reduce cantidad de configuración necesaria. Spring Boot permite concentrarse en lógica del negocio en lugar de infraestructura del proyecto. Mejora productividad y permite desarrollo más ágil[5].

2.4.9 AWS (Amazon Web Services)

Amazon Web Services (AWS) es plataforma de servicios en la nube que ofrece amplia gama de servicios de computación, almacenamiento, bases de datos, redes, análisis, inteligencia artificial, IoT, seguridad y aplicaciones empresariales[12]. AWS permite acceso a recursos informáticos escalables y flexibles a través de Internet. Elimina necesidad de invertir en infraestructura

física[12].

Escalabilidad y flexibilidad

AWS permite escalar recursos fácilmente hacia arriba o abajo según demanda, ideal para aplicaciones con variaciones en tráfico. Capacidad de ajustar recursos según carga garantiza que organizaciones paguen solo por lo utilizado. Mejora eficiencia operativa y optimiza costos. Infraestructura elástica de AWS es fundamental para soportar aplicaciones que requieren disponibilidad constante y rendimiento óptimo sin interrupciones, independientemente de picos de tráfico[12].

Seguridad y cumplimiento

AWS prioriza la seguridad como eje central. Ofrece herramientas específicas para cumplir regulaciones de seguridad y privacidad empresariales. La plataforma cifra datos en reposo y tránsito. AWS Identity and Access Management (IAM) gestiona identidades y accesos de forma avanzada. Estas características permiten a organizaciones implementar controles de seguridad robustos. Protegen datos sensibles de manera efectiva, Crucial en la actualidad. Los entornos están más regulados que nunca las empresas requieren mayor conciencia sobre privacidad de datos[12].

2.4.10 Objetivo de AWS

Amazon Web Services persigue un objetivo claro y proporcionar una plataforma de computación en la nube con recursos informáticos flexibles, escalables y seguros. AWS facilita innovación empresarial, Impulsa transformación digital para desarrolladores y organizaciones.

Las empresas pueden concentrarse en desarrollo y crecimiento empresarial. Sin inversiones en infraestructuras físicas complejas. AWS garantiza alto nivel de seguridad. Cumplimiento normativo en operaciones[21].

2.4.11 AWS Elastic Beanstalk

AWS Elastic Beanstalk funciona como plataforma de servicio (PaaS). Simplifica despliegue, gestión y escalabilidad de aplicaciones en la nube. Aspectos clave sobre Elastic Beanstalk[28]:

1. Despliegue simplificado: Desarrolladores despliegan aplicaciones rápidas y sencillamente. Programadores cargan código a Elastic Beanstalk maneja la implementación automática. Provisión de infraestructura subyacente hasta configuración del entorno, Reduce tiempo y esfuerzo significativamente[29].[29].
2. Soporte múltiples lenguajes: Compatible con variedad de lenguajes y frameworks. Java, .NET, PHP, Node.js, Python, Ruby y Go.Desarrolladores utilizan tecnología más cómoda. Se adapta mejor a necesidades específicas.
3. Gestión automática: Plataforma gestiona recursos necesarios automáticamente a servidores, balanceadores de carga, bases de datos, almacenamiento, utiliza Auto Scaling y Elastic Load Balancing, ajusta capacidad según la demanda y garantiza el rendimiento óptimo en tráfico de variable.

AWS Elastic Beanstalk representa solución PaaS. Facilita despliegue y gestión de aplicaciones en la nube, desarrolladores se enfocan en creación de software sin preocuparse por infraestructura subyacente y automatiza la gestión de recursos,soporta múltiples lenguajes y se integra con otros servicios AWS, opción atractiva para empresas y desarrolladores, implementación eficiente de aplicaciones en la nube[12].

2.4.12 Objetivo de Elastic Beanstalk

El objetivo de Elastic Beanstalk es proporcionar una plataforma de servicios en la nube que facilite el despliegue y la gestión de aplicaciones sin que los desarrolladores tengan que preocuparse por la infraestructura subyacente.

Elastic Beanstalk gestiona automáticamente la implementación desde que los usuarios suben su código, mientras maneja provisión de recursos, configuración y escalado para que desarrolladores se enfoquen en desarrollo de software y optimicen tiempo y recursos. Esta plataforma soporta múltiples lenguajes de programación y herramientas, además proporciona a empresas implementación y escalado eficiente de aplicaciones en la nube de manera rápida y efectiva[28].

2.4.13 Configuraciones de Elastic Beanstalk

Las configuraciones de Elastic Beanstalk incluyen diversas opciones para personalizar y optimizar la implementación de aplicaciones. Estas configuraciones permiten a los desarrolladores gestionar aspectos como las variables de entorno, la seguridad, el balanceo de carga y el autoescalado. A través de la consola de Elastic Beanstalk, la CLI de AWS, o archivos de configuración, los usuarios pueden definir parámetros tales como:

1. Variables de entorno: Para configurar cadenas de conexión a bases de datos y otras credenciales sin incluirlas directamente en el código.
2. Seguridad: Utilizando roles de IAM para gestionar permisos y accesos a recursos de AWS.
3. Escalado automático: Configurando el autoescalado para ajustar automáticamente el número de instancias en función del tráfico y la carga.
4. Balanceo de carga: Implementando Elastic Load Balancing para distribuir el tráfico entre las instancias de la aplicación.
5. Configuraciones de la aplicación: Ajustando configuraciones específicas de la plataforma de programación utilizada (por ejemplo, PHP, Java, Node.js).

La configuración ofrece gran flexibilidad y personalización en gestión de aplicaciones en la nube mientras facilita el desarrollo[12].

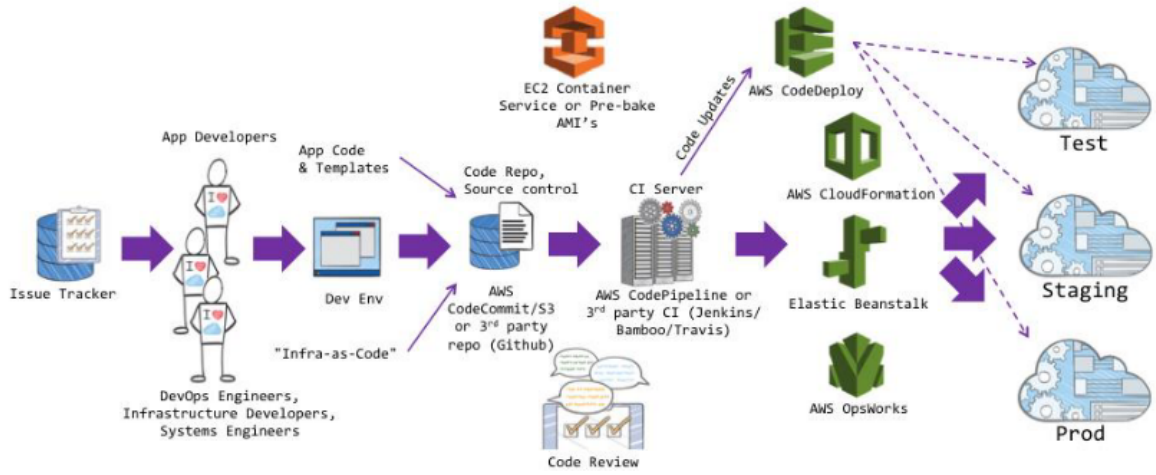

Figure 1.
Descripción general de los servicios de implementación [12].

2.4.14 Amazon S3

Amazon S3 (Simple Storage Service) funciona como servicio de almacenamiento en la nube altamente escalable y duradero donde usuarios almacenan y recuperan cualquier cantidad de datos desde la web en cualquier momento. Resulta ideal para almacenamiento de archivos, copias de seguridad y recuperación ante desastres, mientras se integra fácilmente con otros servicios AWS para distribución de contenido y análisis de datos. Los datos se organizan en ”buckets” que actúan como contenedores donde cada objeto se accede mediante URL única, además S3 incluye características avanzadas como gestión de versiones, encriptación de datos y políticas de ciclo de vida para gestionar efectivamente datos en la nube[28],[12].

2.4.15 Almacenamiento escalable en Amazon S3

Almacena grandes volúmenes de datos de manera flexible y eficiente mientras se adapta a necesidades cambiantes de aplicaciones y negocios.

S3 cuenta con arquitectura que se ajusta automáticamente a necesidades del usuario sin requerir predefinir capacidad de almacenamiento, mientras simplifica la gestión de datos a gran escala.

Este servicio garantiza alta durabilidad y disponibilidad de objetos almacenados mientras los protege frente a pérdidas y ofrece acceso rápido y confiable. Con gestión de versiones y diferentes clases de almacenamiento adaptadas a diversas necesidades de acceso y costo, Amazon S3 representa una solución fundamental para almacenamiento de datos en la nube[3],[28].

2.4.16 Integración de servicios

Amazon S3 se integra con amplia variedad de servicios AWS mientras amplía funcionalidad y permite crear arquitecturas de nube complejas y eficientes. S3 funciona como almacenamiento de datos para Amazon EC2, además proporciona acceso rápido y duradero para aplicaciones en la nube[30].

S3 se emplea junto con Amazon Lambda para ejecutar funciones sin servidor en respuesta a eventos como carga de nuevos objetos en buckets, mientras se asocia con Amazon CloudFront como red de distribución de contenido (CDN) para facilitar entrega rápida de contenido almacenado a usuarios mundiales. La integración permite que S3 funcione como componente central en soluciones de datos a gran escala, además respalda capacidades analíticas y procesamiento en tiempo real[28],[12].

Amazon S3 representa un servicio de almacenamiento de objetos robusto y flexible con escalabilidad, durabilidad y seguridad, mientras su integración con otros servicios AWS y modelo de precios basado en uso lo convierte en opción popular para empresas y desarrolladores que requieren solución de almacenamiento en la nube eficiente y confiable[21],[28].

2.4.17 Git Hub

Git funciona como sistema de control de versiones distribuido donde desarrolladores gestionan y realizan seguimiento de cambios en código fuente de proyectos, mientras su arquitectura distribuida permite que cada desarrollador tenga copia local completa del repositorio para facilitar trabajo en equipo y colaboración mediante creación de ramas para desarrollo paralelo de nuevas

características sin afectar la base de código [31].

Git registra historial completo de cambios mientras facilita identificación y reversión de errores, así como fusión de cambios de diferentes ramas. Su integración con plataformas como GitHub y GitLab ofrece gestión de proyectos, revisión de código y automatización de flujos de trabajo DevOps[4].

2.4.18 Características principales

Git cuenta con comandos esenciales como commit, push y pull para control de versiones, mientras commit guarda cambios en repositorio local y asegura que cada modificación esté documentada con mensaje descriptivo que explique la naturaleza del cambio[20].

Push envía commits desde repositorio local a repositorio remoto mientras facilita colaboración entre múltiples desarrolladores al actualizar versión compartida del proyecto, además pull integra cambios realizados en repositorio remoto hacia copia local para asegurar que desarrolladores trabajen con versión más actualizada[20]. Ambas funcionalidades gestionan desarrollo de software de manera eficiente mientras garantizan historial de cambios claro y colaboración fluida[21],[7].

2.4.19 Ventajas del Uso de Git hub

GitHub aporta ventajas a desarrolladores y equipos de software mientras facilita colaboración y gestión de proyectos. Como plataforma basada en Git permite alojar repositorios de código en la nube y colaborar en tiempo real, además mejora comunicación y revisión de código mediante pull requests[32][20].

GitHub ofrece herramientas para seguimiento de problemas y planificación de proyectos mediante sistema de gestión que organiza tareas y asigna responsabilidades dentro del equipo, mientras soporta automatización de flujos de trabajo con GitHub Actions para implementar CI/CD

de manera sencilla y efectiva. Esto optimiza tiempo de desarrollo y despliegue de aplicaciones mientras GitHub amplía capacidades de Git y facilita entorno de trabajo más colaborativo y eficiente para desarrollo de software[7],[4].

2.4.20 Uso de Jenkins para CI/CD

Jenkins representa una de las herramientas más populares para gestión de construcciones de integración continua y canalizaciones de entrega, mientras permite a desarrolladores construir y probar software de manera continua para aumentar eficiencia y calidad del desarrollo. Facilita automatización de tareas repetitivas mientras los equipos se centran en desarrollo de nuevas características y mejora del software existente[7].

Conceptos clave

1. Integración continua (CI): Práctica de fusionar cambios de código en un repositorio compartido varias veces al día. Jenkins permite la ejecución automática de pruebas para detectar errores rápidamente.
2. Entrega continua (CD): Extensión de CI que permite que el software se despliegue automáticamente en producción después de pasar las pruebas. Jenkins puede automatizar el proceso de despliegue[7].

2.4.21 Características principales del uso de Jenkins

Jenkins cuenta con capacidad para automatizar ciclo de vida del desarrollo de software desde construcción hasta despliegue, mientras funciona como servidor de integración continua de código abierto donde desarrolladores gestionan construcciones de software de manera automática y programada, además facilita pruebas continuas e integración de cambios en el código.

Jenkins configura pipelines que definen proceso de entrega desde codificación hasta producción mientras mejora transparencia y control, además cuenta con extensa biblioteca de plugins

que extienden funcionalidad y permiten integraciones con diversas herramientas y tecnologías. Su naturaleza extensible y comunidad activa proporcionan actualizaciones constantes y soporte técnico mientras contribuyen a su popularidad y eficacia en entornos DevOps[7].

2.4.22 Proceso de configuración de Jenkins para CI/CD

Jenkins se configura para CI/CD (Integración Continua/Entrega Continua) instalando el software en servidor que soporte Java, mientras la configuración inicial desbloquea Jenkins e instala plugins recomendados. Después de la instalación se crea nuevo trabajo o ”job” donde se configura repositorio de código, generalmente basado en Git.

Se define pipeline mediante archivo Jenkinsfile que detalla etapas de construcción, pruebas y despliegue del proyecto, mientras se establecen disparadores automáticos para iniciar construcciones ante cambios en repositorio, además se integran pruebas de calidad y notificaciones sobre estado de construcciones mediante email o aplicaciones de mensajería. La configuración del despliegue automatizado garantiza que cada cambio pase a producción de manera controlada y eficiente mientras convierte a Jenkins en herramienta clave dentro del entorno DevOps[7],[28].

1. Instalación de Jenkins
    - Jenkins se puede instalar en diversas plataformas utilizando paquetes nativos, Docker, o ejecutándolo en cualquier máquina con Java Runtime Environment (JRE).
    - Se debe acceder a la interfaz web de Jenkins a través de un navegador tras completar la instalación.
2. Configuración inicial
    - Al acceder por primera vez, Jenkins requiere una configuración inicial donde se debe ingresar una clave de desbloqueo proporcionada durante la instalación.
    - Después del desbloqueo se instalan plugins recomendados mientras estos amplían las capacidades esenciales de Jenkins.

3. Creación de un nuevo proyecto

    - Los nuevos trabajos o ”jobs” se crean seleccionando ”Nuevo Item” en el menú principal mientras se elige tipo de proyecto adecuado como Freestyle Project o Pipeline.
    - La fuente de código se configura conectando a repositorio Git donde se aloja el código del proyecto.

4. Configuración del Pipeline

    - Pipeline define archivo Jenkinsfile que describe etapas del proceso de construcción desde compilación hasta pruebas y despliegue.
    - Jenkinsfile se almacena en el mismo repositorio del proyecto mientras facilita su mantenimiento.

5. Configuración de disparadores

    - Disparadores automatizan construcciones mediante webhooks que se activan al realizar push o programación temporal.

6. Integración de pruebas y calidad

    - Cada fase del pipeline incluye etapas para ejecutar pruebas unitarias y de integración mientras asegura que cambios no rompan funcionalidad existente.
    - Jenkins se integra con herramientas de calidad de código mientras estas generan reportes detallados.

7. Despliegue automatizado

    - Jenkins configura etapa de despliegue mientras envía aplicación a servidor o servicio de producción.
    - Se pueden utilizar plugins para facilitar la conexión con plataformas de nube como AWS o Azure.

8. Monitoreo y notificaciones

- Se puede configurar el monitoreo del estado de las construcciones y activar notificaciones a través de correo electrónico o aplicaciones de mensajería como Slack, para alertar al equipo ante fallos o finalización de construcciones.

2.4.23 Configuración de pipeline en Jenkins

Para crear un pipeline con jenkins, primero debes asegurarte de que jenkins esté instalado en un servidor accesible y luego desbloquearlo tras la instalación inicial. A continuación, en el panel principal, seleccionas ”Nuevo elemento” para crear un nuevo proyecto de tipo ”Pipeline”. Es crucial configurar el repositorio de código, normalmente en Git, ingresando la URL y las credenciales necesarias.

Jenkinsfile se define en raíz del repositorio mientras especifica etapas del pipeline utilizando sintaxis de Jenkins como ”Construir”, ”Probar” y ”Desplegar”.

Disparadores automáticos configuran pipeline para ejecutarse con cada cambio en repositorio, mientras notificaciones mantienen al tanto del estado tras guardar y ejecutar. Esto implementa procesos de integración y entrega continua de forma efectiva mediante capacidades de automatización de Jenkins[7],[22].

1. Definir el Pipelinesfile

    - CJenkinsfile se crea en raíz del repositorio mientras contiene definición del pipeline utilizando sintaxis de pipeline de Jenkins, además define etapas como ”Construir”, ”Probar” y ”Desplegar”.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building...'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying...'
            }
        }
    }
}
```

Figure 2.
Ejemplo De Código: Pipeline File[7].

2. Definir el Jenkinsfile

   - Jenkinsfile define pipeline que ejecuta tres etapas: ”Build”, ”Test” y ”Deploy”, mientras construcción usa Maven para empaquetar aplicación, pruebas ejecutan automatización y despliegue copia archivo generado a servidor remoto utilizando scp. Esta automatización facilita flujo de trabajo de desarrollo mientras asegura que cada cambio en código se construya, pruebe y despliegue de manera controlada y eficiente, además resulta fundamental en entorno DevOps[7].

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building the application...'
                sh 'mvn clean package' // Comando para constru
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests...'
                sh 'mvn test' // Comando para ejecutar pruebas
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying the application...'
                sh 'scp target/myapp.jar user@server:/path/to/
            }
        }
    }
}
```

Figure 3.
Ejemplo De Código: Jenkins file[7].

CAPÍTULO III

MATERIALES Y MÉTODOS

3.1 Tipo de Investigación

Esta investigación se lleva a cabo con un enfoque cuantitativo.

Considero que el enfoque cuantitativo es esencial para medir y analizar el impacto de la integración y entrega continua (CI/CD) en entornos de microservicios.

Variables medibles como tiempo de despliegue, cantidad de errores y eficiencia del pipeline automatizado ofrecen evidencia concreta sobre mejoras que CI/CD aporta mientras se compara con procesos más manuales[33].

Análisis sobre madurez de automatización de pruebas demuestra la importancia de métricas cuantitativas para evaluar rendimiento y calidad del software, mientras recopila datos de proyectos de código abierto y examina cómo automatización afecta calidad del producto y ciclo de lanzamiento. Esto demuestra importancia de enfoque cuantitativo en práctica del desarrollo de software[33].

Jenkins automatiza tareas y procesos CI mientras optimiza entornos de microservicios, complementando enfoques experimentales para solucionar desafíos prácticos del desarrollo de software[34].

3.1.1 Enfoque Metodológico

El proyecto emplea metodología Scrum, ampliamente reconocida en la industria de software por gestionar proyectos complejos y adaptarse efectivamente a cambios durante el desarrollo.

Scrum emplea un enfoque iterativo e incremental dividiendo el proyecto en sprints de duración fija que facilitan planificación y entrega continua de valor a los interesados. Los sprints establecen objetivos claros enfocados en funcionalidades específicas, mejorando visibilidad del progreso y fomentando colaboración entre miembros del equipo[34].

El proyecto establece reuniones diarias de seguimiento y revisiones de sprint que evalúan entregables y ajustan prioridades conforme surgen nuevas necesidades.

La retroalimentación y adaptación continuas permiten abordar desviaciones del proyecto en tiempo real, generando desarrollo más efectivo y alineado con objetivos estratégicos del usuario final[34].

Scrum facilita organización y planificación del trabajo mientras mejora comunicación entre miembros del equipo e interesados, promoviendo cultura de colaboración y responsabilidad compartida[7].

Este enfoque optimiza proceso de desarrollo mientras asegura que entregables cumplan con estándares de calidad establecidos al inicio del proyecto[33].

3.1.2 Varibales de Estudio

Variables Independientes

1. Implementación del pipeline CI/CD automatizado

    - La implementación del pipeline CI/CD automatizado utiliza herramientas como Jenkins

- integración, pruebas y despliegue, mientras sus indicadores incluyen configuración del pipeline mediante YAML y Jenkinsfile. Grado de automatización (parcial/total).

2. Metodología ágil (Scrum)

   - Definición: Aplicación de sprints, ceremonias (planning, retrospectivas) y roles (Scrum Master) para gestionar la implementación del CI/CD.
   - Indicadores: Número de sprints dedicados a la automatización.

     Retroalimentación del equipo en ceremonias.

3. Herramientas específicas

   - Definición: Uso de Jenkins (orquestación), GitHub (repositorio) y AWS Elastic Beanstalk (despliegue).
   - Indicadores: Integración exitosa entre herramientas.
   - Configuración óptima de entornos (dev/test/prod).

Variables Dependientes

1. Tiempo de despliegue

   - Definición: Duración desde el commit de código hasta el despliegue exitoso en AWS.
   - Métrica: Minutos/segundos por despliegue (comparativo antes/después de la automatización).

2. Número de errores en integración/despliegue

   - Definición: Fallos detectados durante la compilación, pruebas o despliegue.
   - Métrica: Porcentaje de despliegues fallidos.
   - Errores críticos (ej.: timeout, dependencias rotas).

Variables de Control

- Tecnología base: Microservicios en Spring Boot
- Entorno AWS: Configuración fija de Elastic Beanstalk
- Complejidad del proyecto: Número de microservicios y sus interdependencias.

3.2 Planificación del Proyecto

La implementación del pipeline de CI/CD para microservicios Spring Boot en AWS Elastic Beanstalk se divide en módulos que cubren desde el análisis inicial hasta la validación y optimización del sistema. Cada uno de estos módulos desempeña un papel clave para garantizar que el proceso de automatización sea eficiente y de alta calidad.

La Tabla 1 detalla estos módulos:

Tabla 1.
Planificación de módulos del proyecto
Nota. Fuente: Elaboración propia.

Módulo	Modelo	Descripción	Submedidas / Componentes
Módulo 1	Análisis de Requerimientos y Selección de Herramientas	Identificación de necesidades del sistema, estudio de conceptos clave sobre CI/CD y microservicios, y selección de herramientas mediante análisis comparativo técnico.	• Investigación teórica • Comparativa de herramientas
Módulo 2	Configuración de Infraestructura y Entorno CI/CD	Instalación y configuración de Jenkins, creación de repositorios en GitHub, integración de plugins y preparación de entornos en AWS (Elastic Beanstalk, S3).	• Configuración de Jenkins • Repositorios GitHub • Entornos AWS
Módulo 3	Desarrollo y Automatización del Pipeline	Diseño e implementación del pipeline CI/CD utilizando Jenkinsfile para automatizar los procesos de construcción, pruebas y despliegue.	• Creación del Jenkinsfile • Integración de pruebas • Despliegue en AWS

3.2.1 Sprints y Módulos de Desarrollo

La metodología ágil aborda específicamente el concepto de sprints y organización del desarrollo en módulos.

Esta práctica mejora colaboración y eficiencia en equipos de desarrollo mientras permite

manejo más flexible y adaptativo de proyectos de software.

La metodología ágil aborda específicamente el concepto de sprints y organización del desarrollo en módulos mientras mejora colaboración y eficiencia en equipos de desarrollo, además permite manejo más flexible y adaptativo de proyectos de software[7].

El análisis selecciona primer grupo de herramientas relevantes y útiles para el proyecto.

Prototipos funcionales se desarrollan en entorno de prueba mediante enfoque ágil.

Esta fase implementa características básicas mientras evalúa rendimiento de cada herramienta en situaciones reales de desarrollo.

Pruebas de herramientas antes de implementación final identifican ventajas y desventajas en contextos prácticos[4].

3.3 Módulo 1: Análisis de Requerimientos y Selección de Herramientas

Análisis de Requerimientos y Selección de Herramientas

Este módulo analiza requisitos funcionales y no funcionales esenciales para implementar pipeline de Integración Continua y Entrega Continua (CI/CD) para microservicios desarrollados en Spring Boot, mientras asegura que sistema cumpla con expectativas y necesidades de stakeholders del proyecto[33].

El primer sprint identifica necesidades del sistema mediante investigación sobre conceptos y herramientas relacionados con CI/CD y microservicios, mientras revisa mejores prácticas internacionales en automatización de procesos de desarrollo y despliegue de software para alinear objetivos del proyecto con estándares globales que elevan calidad del software[15].

Investigación sobre mejores prácticas internacionales en CI/CD y microservicios integra enfoques y técnicas reconocidas.

Esto mejora eficacia y eficiencia del pipeline mientras garantiza que métodos utilizados se alineen con tendencias del sector[4].

En el segundo sprint se realiza una comparación técnica detallada entre las herramientas del mercado, abarcando Jenkins, GitHub y AWS Pipeline.

La evaluación permitió seleccionar soluciones apropiadas para el proyecto considerando facilidad de integración, escalabilidad, costos y compatibilidad tecnológica.

El módulo incluye análisis detallado de requisitos y justificación técnica para herramientas seleccionadas del pipeline, respaldando decisiones con análisis exhaustivo y fundamentado[19].

Resumen de Actividades por Sprint

Tabla 2.
Planificación de Sprints y Actividades

Sprint	**Actividades**	**Resultados Esperados**
Sprint 1	- Análisis de necesidades del sistema. - Investigación sobre CI/CD y microservicios. - Revisión de mejores prácticas en automatización.	Documento de requerimientos funcionales y no funcionales.
Sprint 2	- Comparativa entre Jenkins, GitHub, AWS Pipeline. - Selección y justificación de herramientas (Jenkins, GitHub, AWS Elastic Beanstalk).	Herramientas seleccionadas y justificadas técnicamente.

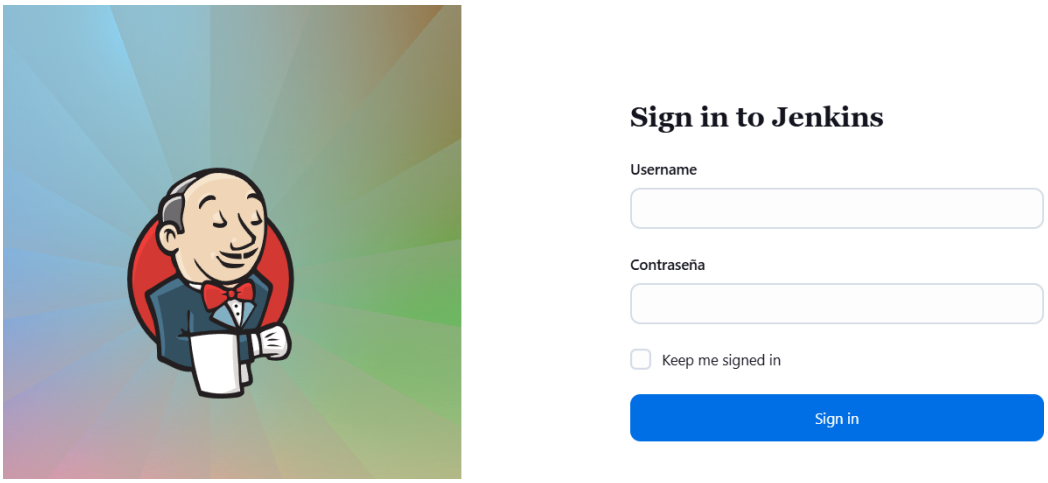

Figure 4.
Sprint 2:Selección y justificación de herramientas Jenkins
Nota. Fuente: Elaboración propia.

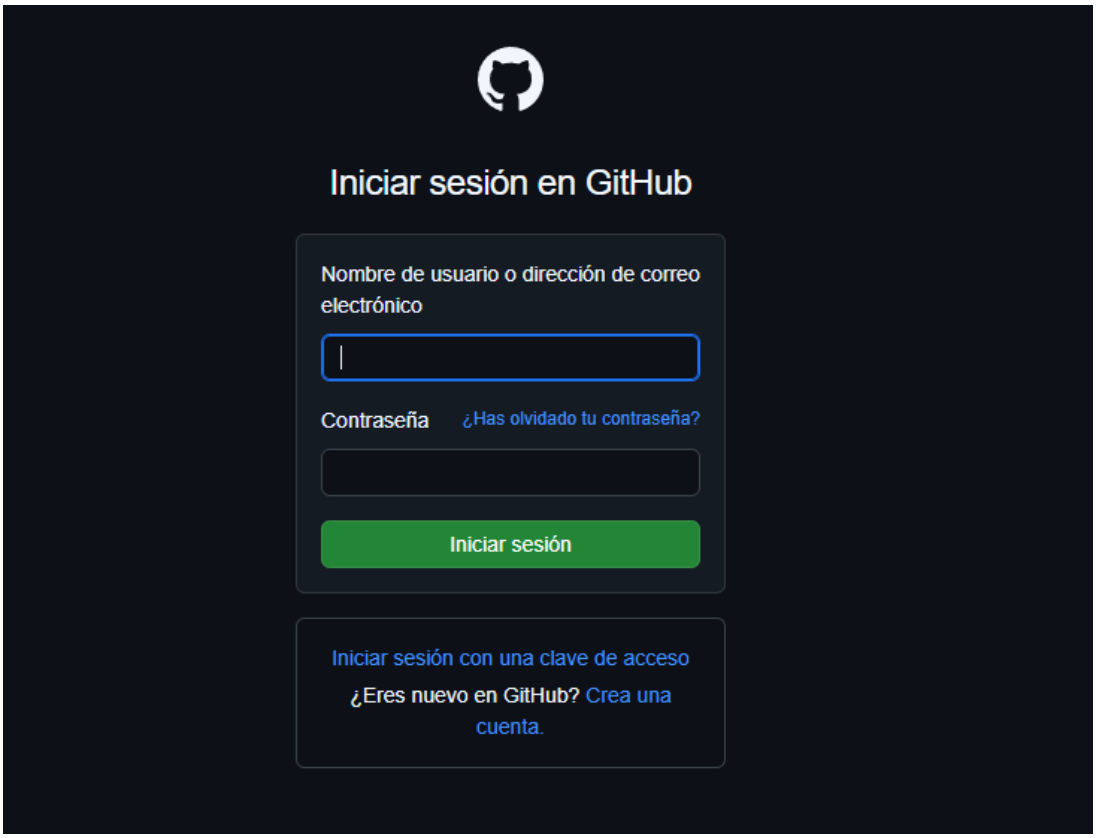

Figure 5.
Sprint 2:Selección y justificación de herramientas GitHub
Nota. Fuente: Elaboración propia.

3.4 Módulo 2: Configuración de Infraestructura y Entorno de CI/CD

El módulo establece la infraestructura requerida para procesos CI/CD de microservicios configurando Jenkins en equipos locales o servidores con parámetros optimizados, además de crear repositorios GitHub para gestión centralizada y segura del código fuente[8].

Se instalan plugins esenciales en Jenkins para facilitar integración con GitHub y servicios de despliegue, estableciendo base fundamental del flujo de trabajo ágil. AWS Elastic Beanstalk configura el entorno de despliegue definiendo entornos de prueba, políticas de seguridad y almacenamiento S3 para gestión eficiente de archivos de despliegue[21].

Tabla 3.
Implementación Técnica: Jenkins, GitHub y AWS

Sprint	**Actividades**	**Resultados Esperados**
Sprint 3	- Instalación y configuración de Jenkins en equipo o local. - Creación de repositorios en GitHub. - Instalación de plugins necesarios en Jenkins.	Jenkins funcional, repositorio GitHub preparado.
Sprint 4	- Configuración de AWS Elastic Beanstalk - Definición de entornos de pruebas. - Configuración de claves y permisos en AWS. - Configuración de AWS S3.	Entorno de despliegue preparado y accesible.

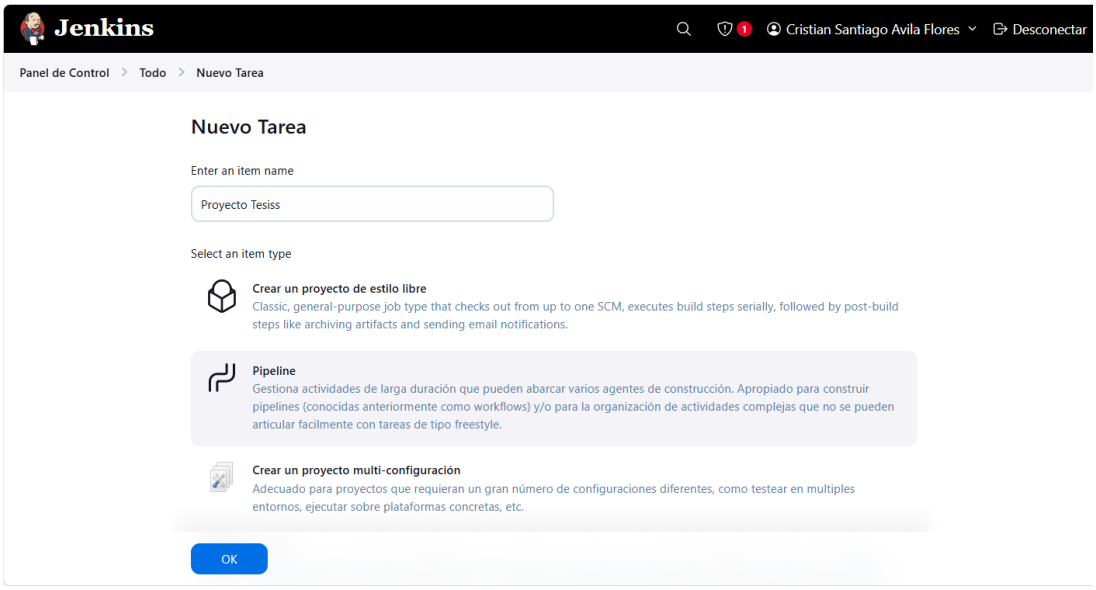

Figure 6.

Sprint 3:Configuración de Jenkins

Nota. Fuente: Elaboración propia.

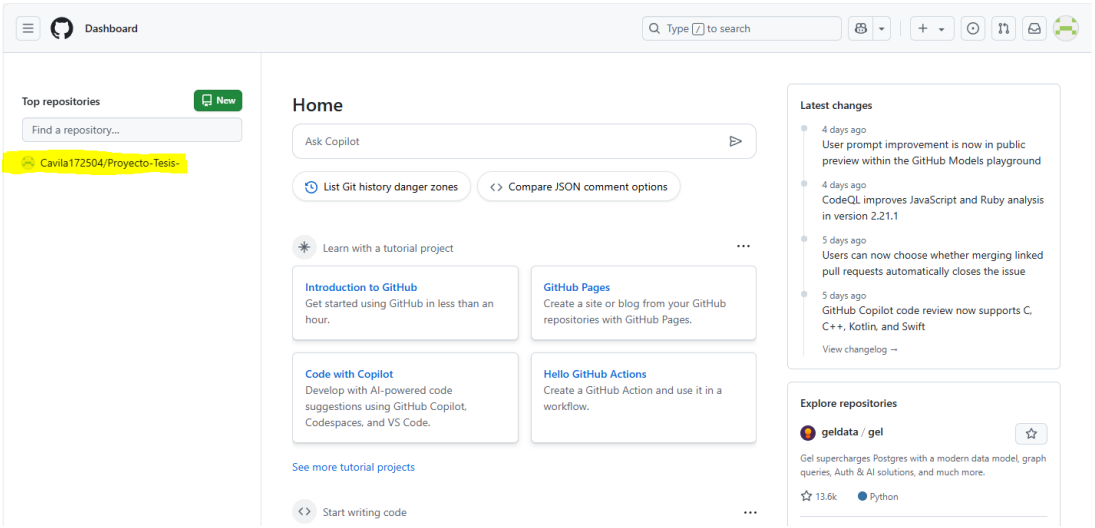

Figure 7.

Sprint 3:Repositorios en GitHub

Nota. Fuente: Elaboración propia.

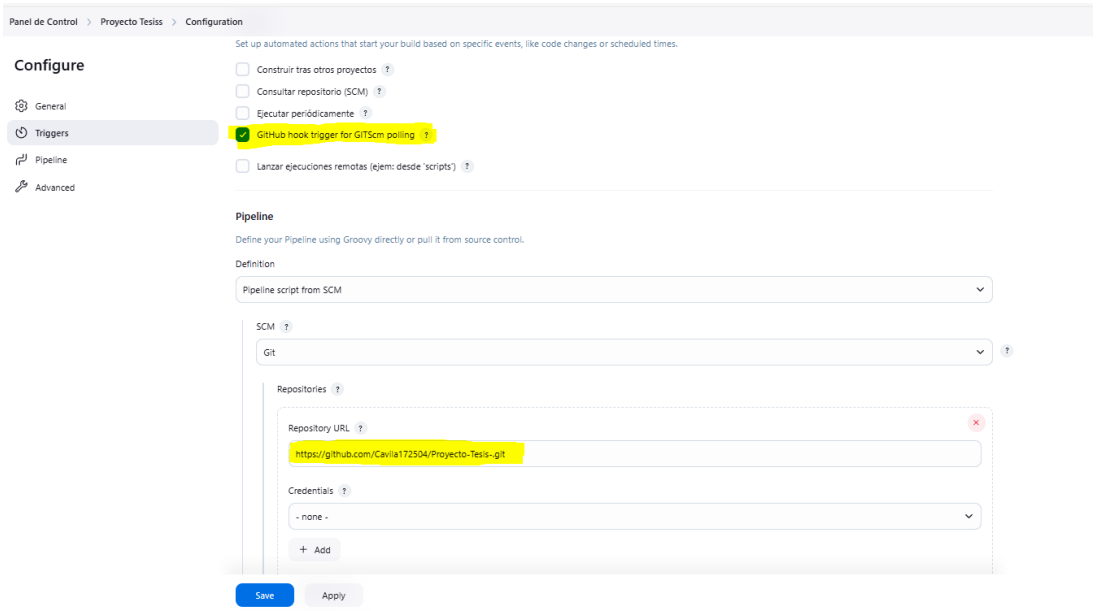

Figure 8.

Sprint 3:Integrar GitHub con Jenkins

Nota. Fuente: Elaboración propia.

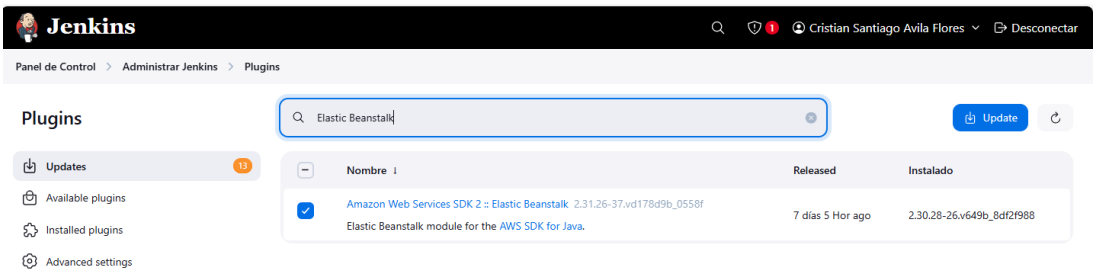

Figure 9.

Sprint 3:Instalación de plugins

Nota. Fuente: Elaboración propia.

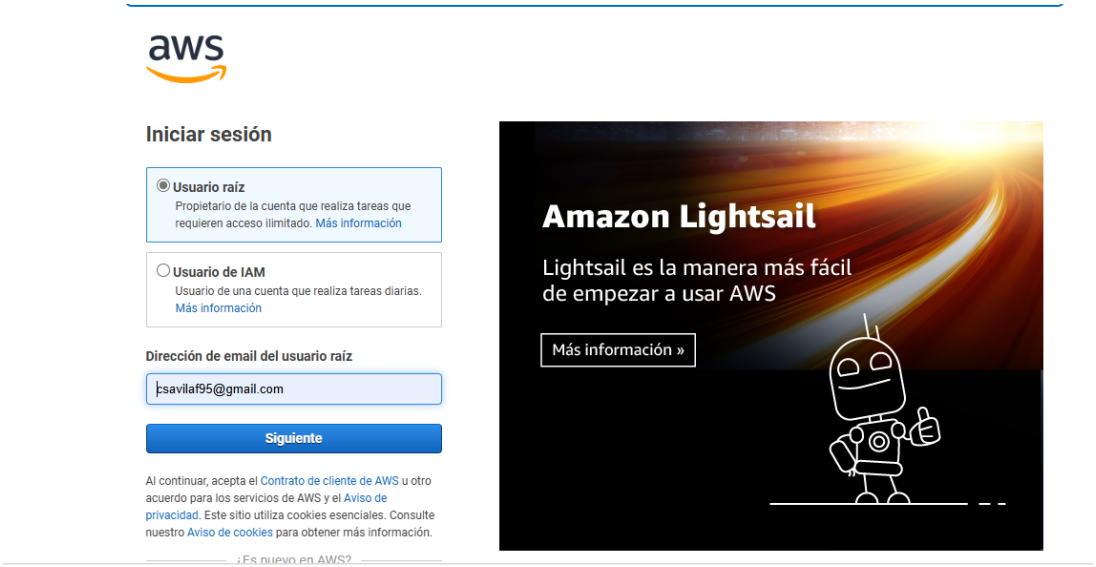

Figure 10.

Sprint 4:Inicio secion AWS

Nota. Fuente: Elaboración propia.

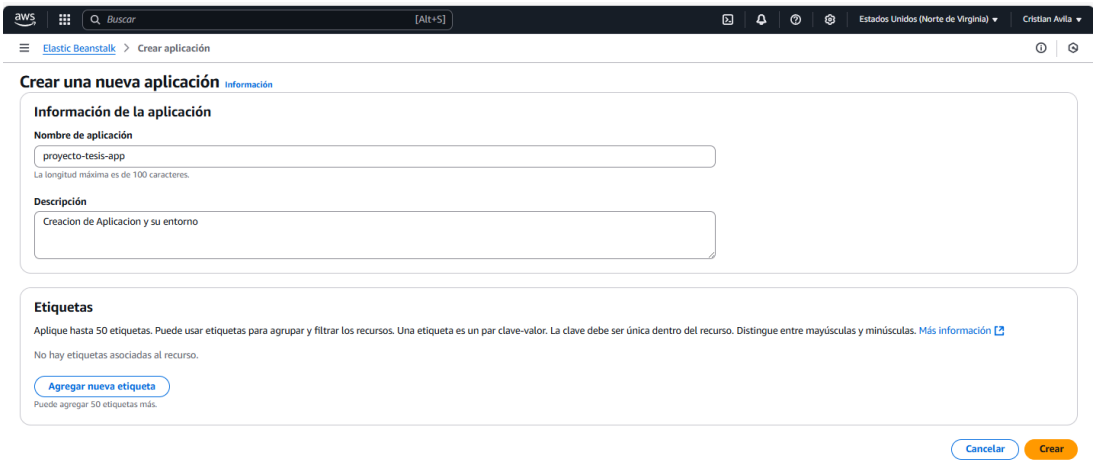

Figure 11.

Sprint 4:Configuración de AWS Elastic Beanstalk.

Nota. Fuente: Elaboración propia.

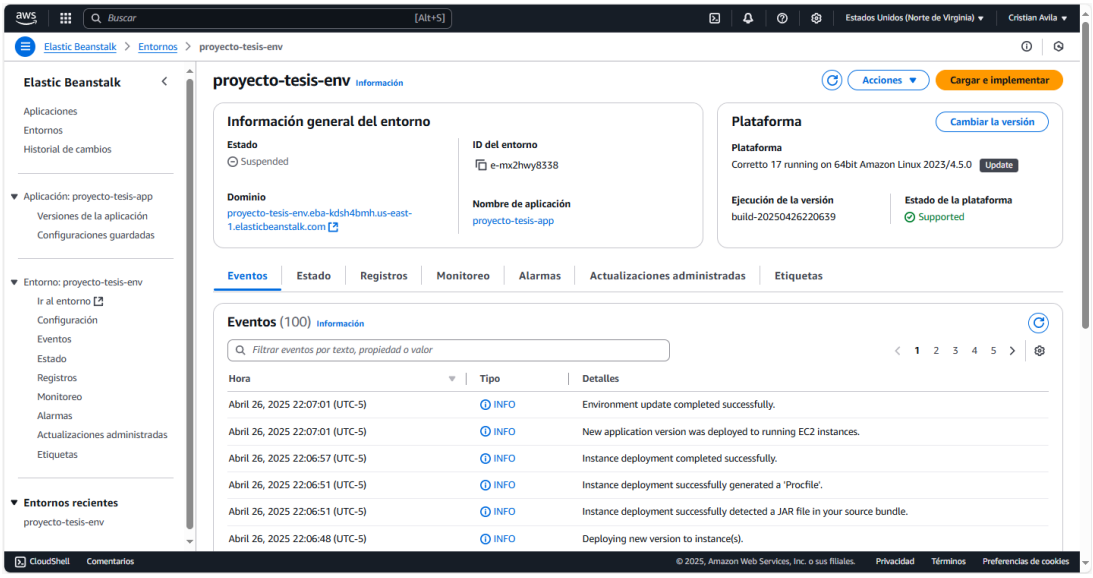

Figure 12.

Sprint 4:Definición de entornos de pruebas

Nota. Fuente: Elaboración propia.

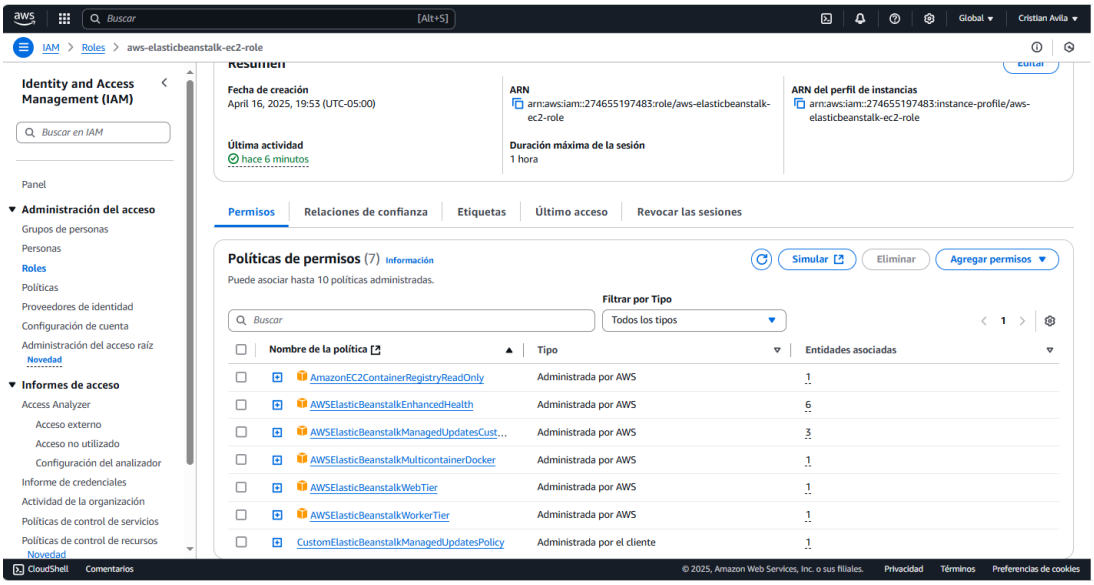

Figure 13.

Sprint 4:Configuración de claves y permisos en AWS

Nota. Fuente: Elaboración propia.

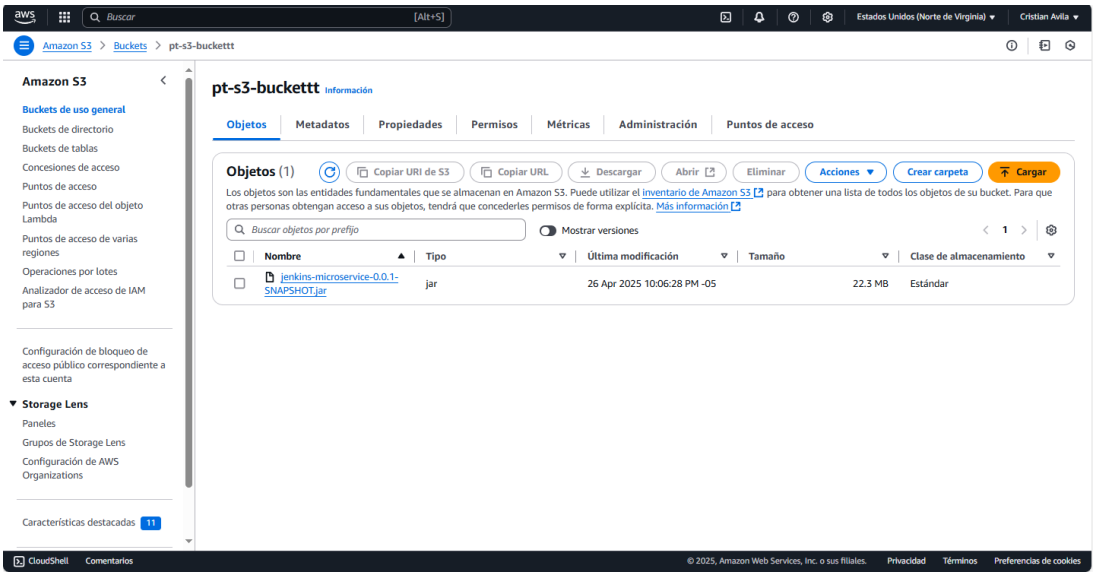

Figure 14.
Sprint 4:Configuración de AWS S3.
Nota. Fuente: Elaboración propia.

3.5 Módulo 3: Desarrollo y Automatización del Pipeline

El Módulo 3 se enfocó en desarrollar a fondo un pipeline de Integración Continua y Entrega Continua (CI/CD), diseñado especialmente para facilitar el despliegue automatizado de microservicios creados con Spring Boot.

La meta principal de este módulo fue garantizar que cada cambio en el código fuente activara un proceso que permitiera la compilación, las pruebas y el despliegue automáticos en la infraestructura en la nube que ya había sido configurada[22].

Sprint 5 crea archivo Jenkinsfile mientras define etapas del pipeline incluyendo construcción del proyecto, ejecución de pruebas unitarias y despliegue automatizado, además configura parámetros de integración con GitHub para que pipeline se active automáticamente con cada commit en repositorio[7],[8].

Sprint 6 mejora pipeline añadiendo pruebas automáticas y ajustando entorno mientras detecta errores y fallos de manera temprana, además implementa lógica para despliegue automático en AWS Elastic Beanstalk para asegurar que versión más reciente del microservicio esté siempre

disponible para usuarios finales[14].

Este módulo automatiza tareas clave del ciclo de vida del software mientras reduce notablemente tiempos de entrega y aumenta fiabilidad de despliegues[8].

Tabla 4.
Desarrollo del Pipeline de Automatización

Sprint	**Actividades**	**Resultados Esperados**
Sprint 5	- Creación del archivo `Jenkinsfile`. - Definición de etapas del pipeline: build, test, deploy. - Integración con GitHub para activación automática.	Pipeline básico funcional, conectado al repositorio.
Sprint 6	- Integración de pruebas automáticas. - Validación del flujo de trabajo. - Configuración del despliegue automático en AWS Elastic Beanstalk.	Pipeline completo funcional, con despliegue automático.

3.6 Diseño del pipeline de CI/CD para microservicios

Pipeline de CI/CD para microservicios considera arquitectura distribuida que caracteriza este enfoque.

Microservicios requieren método más detallado para implementar y gestionar componentes del software comparado con aplicaciones monolíticas.

Cada fase del ciclo de vida del pipeline facilita integración y entrega continuas de manera eficiente y confiable.

Pipeline se organiza en etapas de construcción, pruebas, despliegue y monitoreo mientras maneja cada microservicio como componente independiente alineado con objetivo general de la aplicación[21],[15].

3.6.1 Fase de Construcción

Fase de construcción representa la etapa primera y más importante del pipeline de CI/CD.

Herramientas facilitan creación de entornos estandarizados y replicables para ejecutar microservicios.

Herramientas facilitan creación de entornos estandarizados y replicables para ejecutar microservicios.

Automatización de esta fase integra sistemas de control de versiones como Git y plataformas de automatización CI como Jenkins o GitHub mientras activan construcción con cada cambio en código[6],[35].

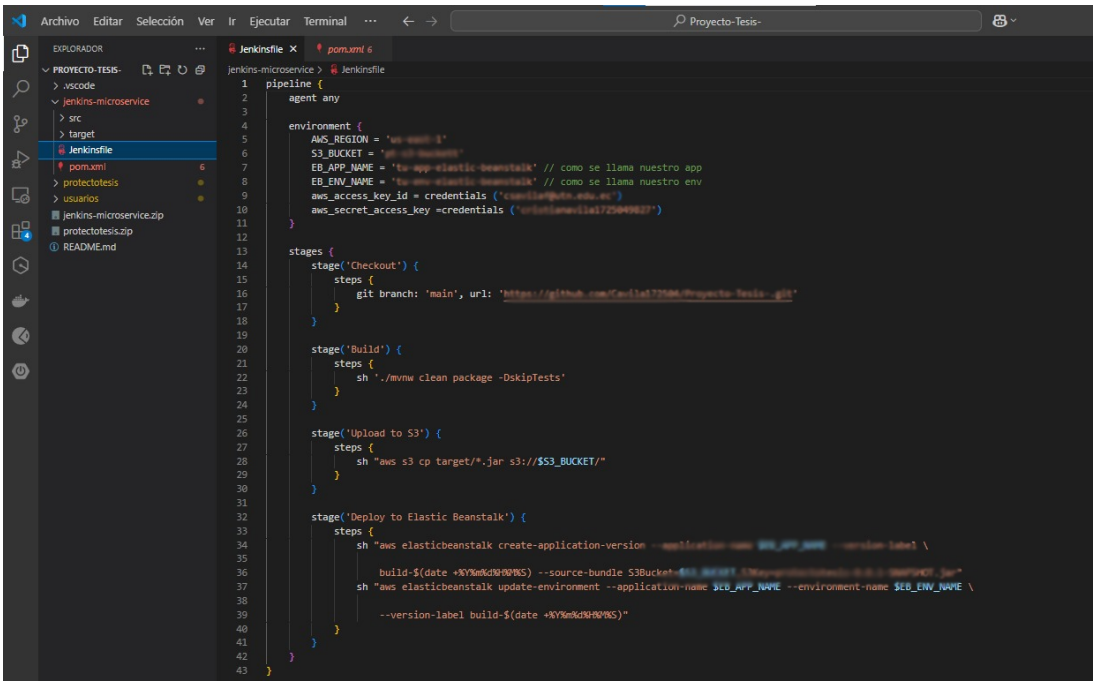

Figure 15.
Construcción del pipeline de CI/CD.
Nota. Fuente: Elaboración propia.

3.6.2 Fase de Despliegue

La fase de despliegue es clave en la entrega continua de software, ya que se trata de implementar de manera automatizada los microservicios en un entorno de producción [21].

Esta etapa asegura transición suave mientras minimiza riesgo de interrupciones en servicio para usuarios finales.

3.6.3 Despliegue con AWS Elastic Beanstalk

El despliegue emplea AWS Elastic Beanstalk, servicio que simplifica la implementación y gestión de aplicaciones en la nube[28].

Elastic Beanstalk permite que los desarrolladores se enfoquen en el código mientras maneja automáticamente la infraestructura.

La plataforma aprovisiona recursos automáticamente y soporta múltiples tecnologías y lenguajes de programación[12].

1. Configuración del Entorno: Elastic Beanstalk establece el entorno de ejecución adaptándose a las necesidades específicas de cada aplicación[28].

   Esto abarca desde seleccionar el tipo de instancia de S3 y gestionar la red. Además, los recursos son escalables, lo que facilita el manejo de picos en la demanda sin complicaciones[4].

2. Proceso de Despliegue: Cuando la aplicación está desarrollada y validada, se despliega en Elastic Beanstalk mediante un proceso directo.

   Proceso de despliegue crea diferentes versiones de aplicación mientras facilita gestión e implementación[8].

3. Monitoreo y Gestión: Elastic Beanstalk proporciona herramientas de monitoreo mientras

supervisa rendimiento de aplicación y recursos que consume[12].

Notificaciones sobre problemas de rendimiento y métricas clave aseguran funcionamiento óptimo de aplicación en producción.

4. Actualizaciones y Mantenimiento: Elastic Beanstalk simplifica actualización de aplicaciones existentes[12].

   Un clic. Los desarrolladores lanzan versiones nuevas, Simple.

   Resulta útil cuando aparecen nuevas funciones frecuentemente o cuando se requiere corregir errores de manera regular.

3.6.4 Consideraciones y Mejores Prácticas

Para despliegues exitosos en AWS Elastic Beanstalk conviene seguir estas prácticas clave:

1. Revisión de la Configuración:

   Previo al despliegue conviene revisar la configuración del entorno para asegurar compatibilidad.

   Esto garantiza adaptación a las necesidades de la aplicación mientras evita recursos innecesarios[21].

2. Backups: Realizar copias de seguridad de sistemas y bases de datos antes de cada despliegue previene pérdida de datos ante posibles inconvenientes[4].

3. Pruebas Post-Despliegue: Ejecutar pruebas automatizadas tras el despliegue asegura que la nueva versión funcione correctamente y cumpla los requisitos funcionales establecidos[12].

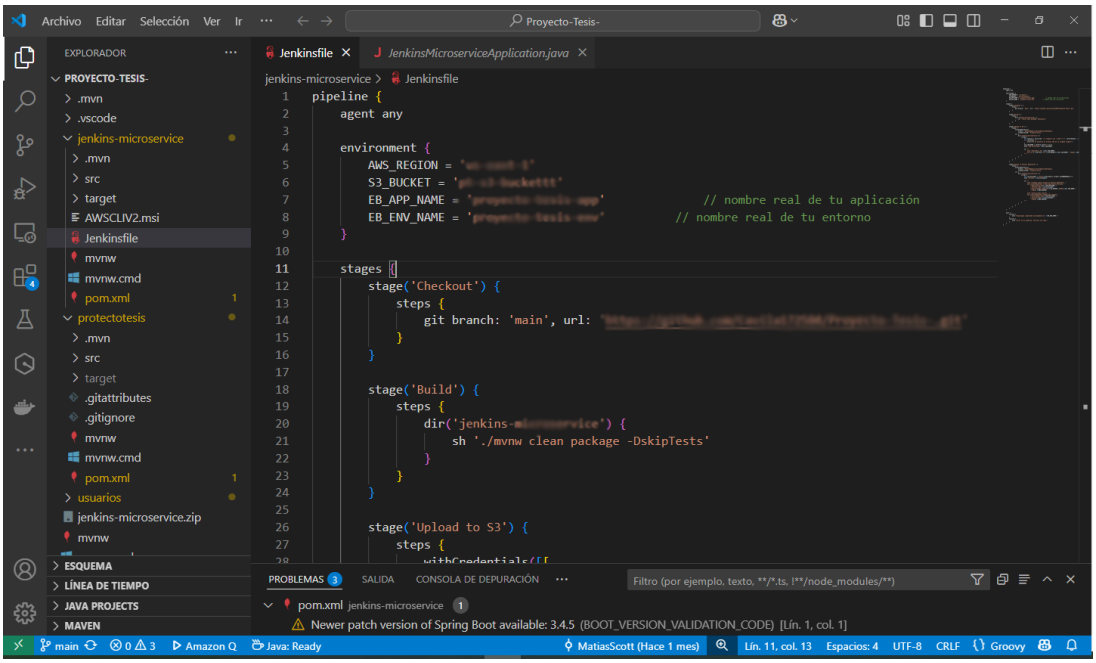

Figure 16.

Sprint 5:Creación de Jenkinsfile para CI/CD

Nota. Fuente: Elaboración propia.

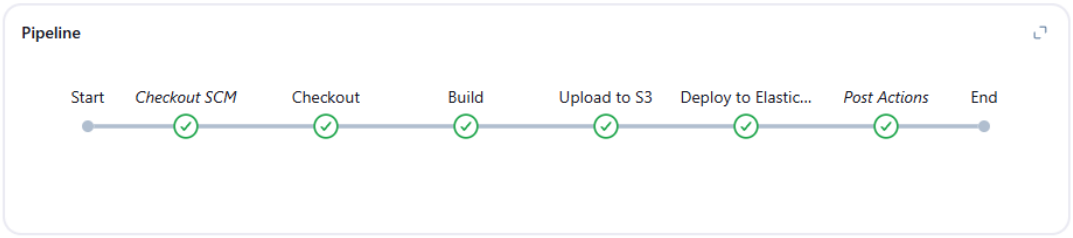

Figure 17.

Sprint 5:Definición de etapas build, test, deploy

Nota. Fuente: Elaboración propia.

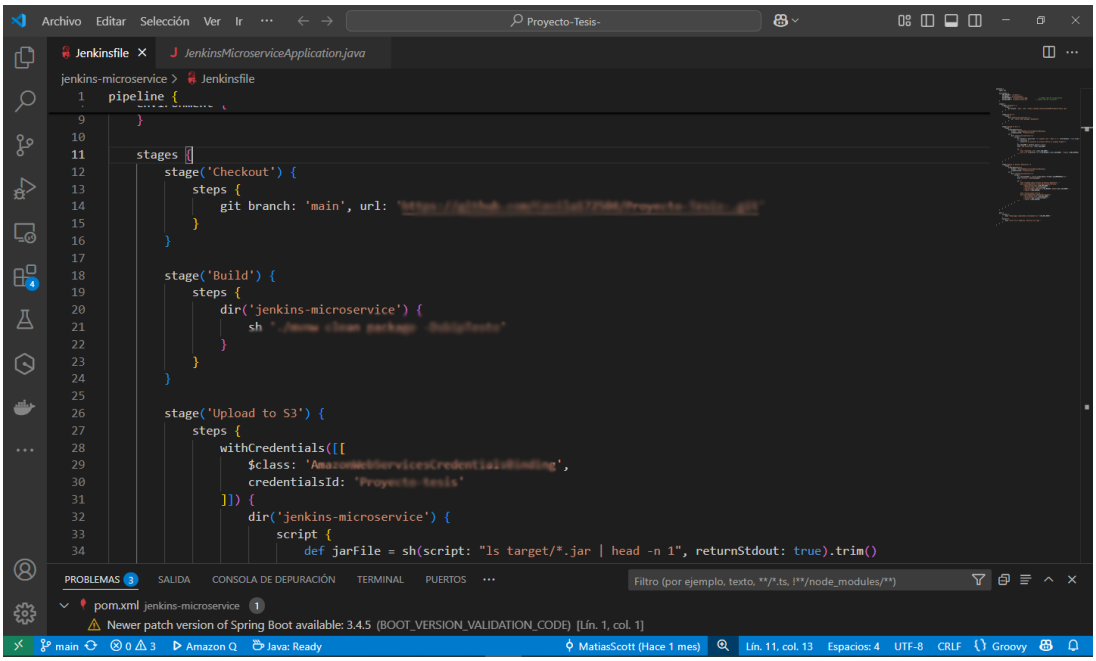

Figure 18.

Sprint 5:Automatización de construcción de microservicios

Nota. Fuente: Elaboración propia.

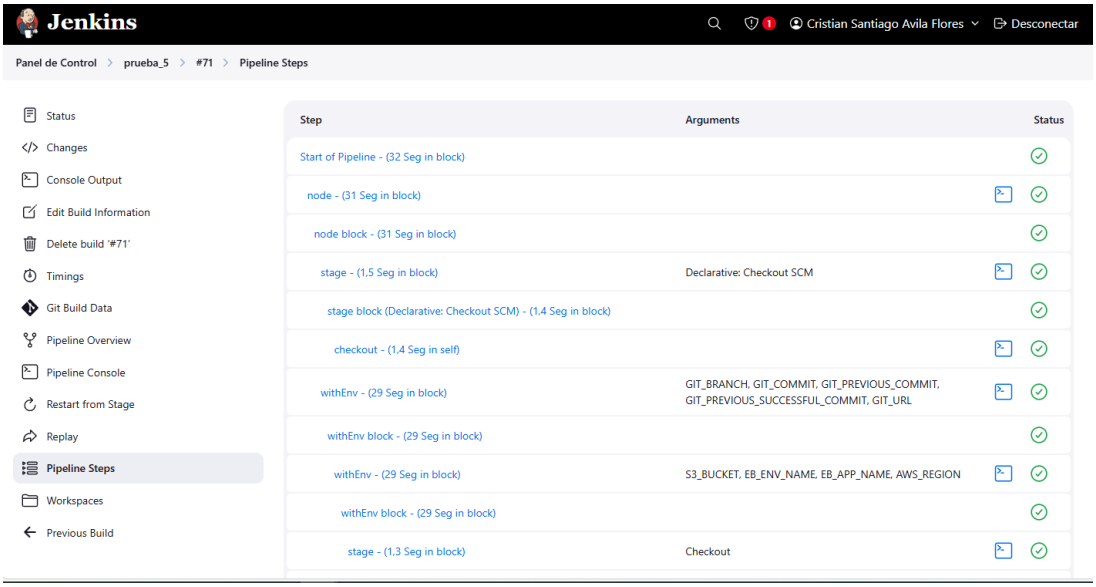

Figure 19.

Sprint 6:Integración de pruebas automáticas

Nota. Fuente: Elaboración propia.

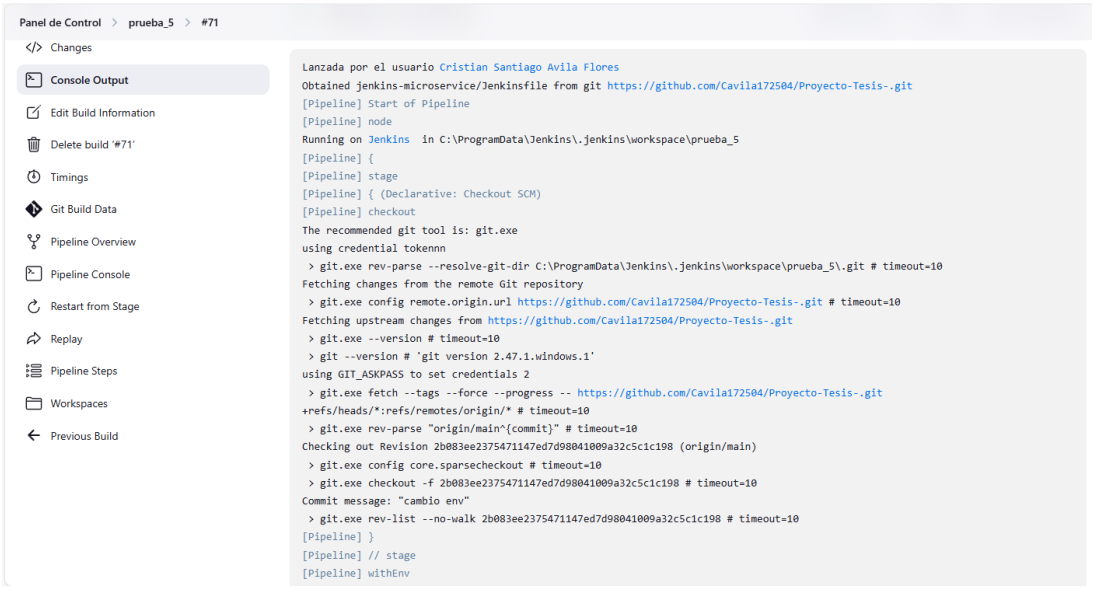

Figure 20.

Sprint 6:Automatización de despliegue en AWS

Nota. Fuente: Elaboración propia.

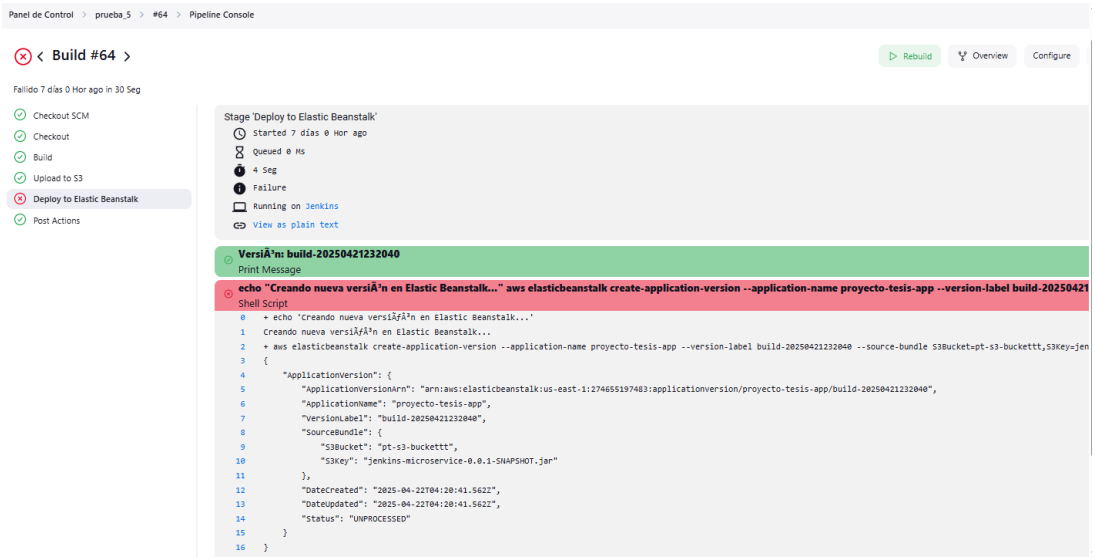

Figure 21.

Sprint 6:Corrección de errores iniciales en pipeline

Nota. Fuente: Elaboración propia.

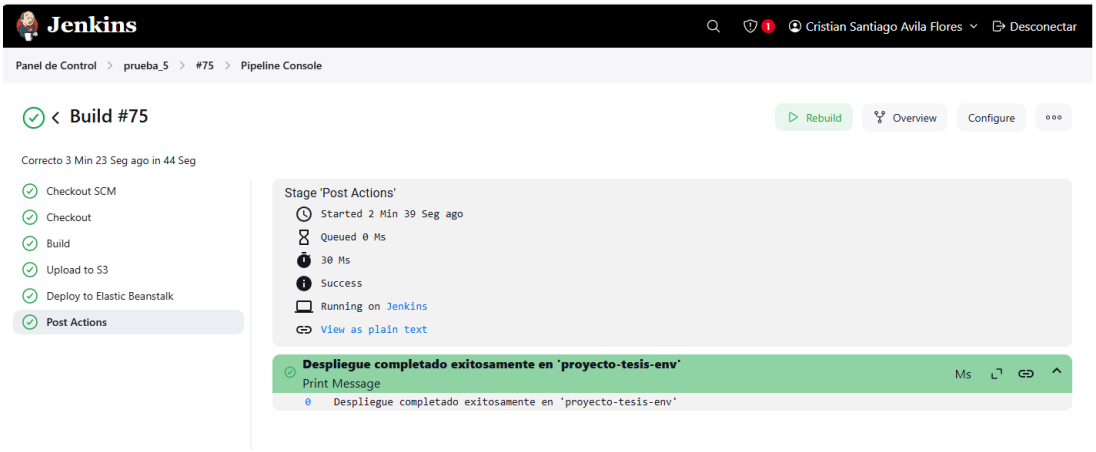

Figure 22.
Sprint 6:Pipeline de CI-CD completo y en funcionamiento
Nota. Fuente: Elaboración propia.

3.7 Implementación del Pipeline en AWS Elastic Beanstalk

El pipeline utiliza AWS Elastic Beanstalk, que ofrece un entorno ideal para proyectos de microservicios[7].

Elastic Beanstalk simplifica la gestión de aplicaciones web permitiendo que los desarrolladores suban código mientras el servicio maneja automáticamente la infraestructura, escalabilidad y monitoreo de la aplicación.

Esta capacidad resulta clave para microservicios al permitir descomponer aplicaciones en componentes independientes que se gestionan y despliegan por separado, reduciendo tiempos de comercialización mientras mejora la agilidad del equipo de desarrollo[4].

3.7.1 Configuración del Entorno

La configuración del entorno en Elastic Beanstalk[3] requirió crear una aplicación y definir un entorno específico con instancias de S3, grupos de escalado automático y balanceadores de carga.

La configuración incluye variables de entorno necesarias como cadenas de conexión a bases de datos, claves API y parámetros críticos que deben mantenerse separados del código de la aplicación.

Configurar el entorno para reflejar condiciones de producción asegura una implementación fluida de la aplicación cuando esté lista[12].

3.7.2 Integración de Herramientas de CI/CD

Las herramientas CI/CD resultan clave para el desarrollo de software moderno.

Permiten automatizar múltiples fases del desarrollo, abarcando la codificación completa hasta la implementación final.

Jenkins y Git mejoran la eficiencia del desarrollo mientras aseguran calidad del software mediante automatización de pruebas y despliegues[13].

1. Jenkins como Motor de Integración Continua: Esta plataforma de código abierto automatiza y simplifica la adopción de prácticas CI/CD en proyectos de desarrollo[7].

    - Construcciones Automáticas: Jenkins ejecuta builds del código cuando ocurren commits en los repositorios Git mediante trabajos configurables[7].

      Los cambios nuevos se prueban al instante, detectando errores tempranamente y manteniendo estabilidad del código[4].

    - Pipeline as Code: Jenkins define flujos de trabajo mediante archivos de configuración (Jenkinsfile) almacenados con el código del proyecto, facilitando replicación y versionado del proceso de construcción y despliegue. La herramienta ofrece notación declarativa o scripts para crear pipelines complejos adaptados a requisitos específicos del proyecto[7].

    - Jenkins tiene modularidad y flexibilidad útiles. Su ecosistema de plugins extiende las funciones base - desde testing tools hasta notificaciones y code analysis, más

otras extensiones disponibles [20].

Cada proyecto tiene necesidades diferentes, y Jenkins se adapta bien a esto[13].

- Monitoreo y reportes: Jenkins tiene una interfaz que te ayuda a ver qué está pasando con tus builds y el sistema en general.

  Te muestra stats y reportes de tus builds y tests, así puedes ver cómo va tu pipeline y trackear si algo se rompe durante el build[13].

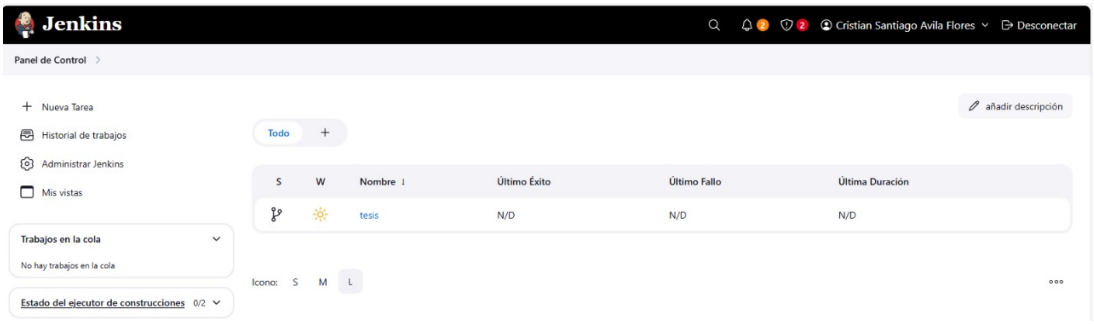

Figure 23.
Jenkins como Motor de Integración Continua.
Nota. Fuente: Elaboración propia.

2. Git para la Gestión de Versiones

   Git es un VCS distribuido que hace que trabajar en equipo sea más fácil. Para CI/CD, las ventajas principales son:

   - Ramas: Puedes hacer branches para trabajar en features nuevas mientras main se queda intacto.

     Cuando tu feature está lista y testeada, haces merge a main.

     Jenkins corre tests automáticamente cuando integras código resultante[7].

   - Historial de Cambios: Git trackea todo lo que cambias en tu código con cada commit, así puedes ver la historia del proyecto y hacer rollback si algo se rompe[4].

     Tu codebase se mantiene estable y no se rompe tan fácil.

   - Colaboración mejorada: Trabajar en equipo es mucho más fácil con Git - puedes hacer PRs para que otros revisen tu código antes de mergearlo.

     Los code reviews te ayudan a catchear bugs antes de merge, plus generas discusiones técnicas chéveres que al final hacen que tu producto sea mejor[19].

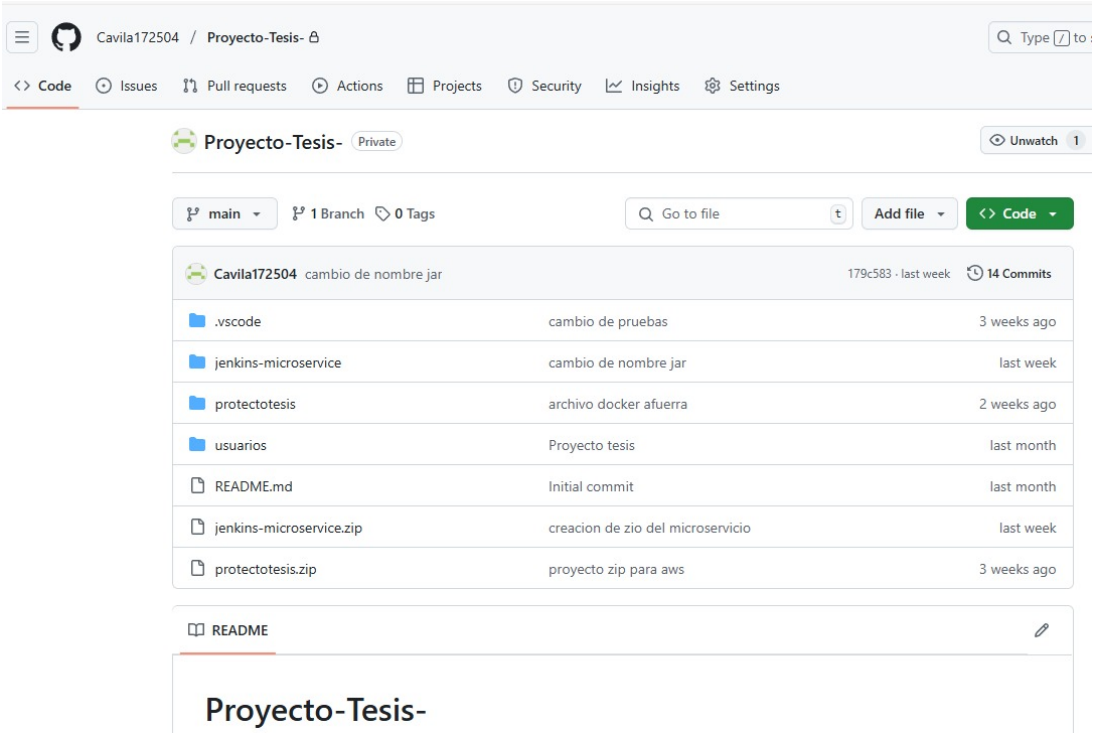

Figure 24.
Jenkins como Motor de Integración Continua.
Nota. Fuente: Elaboración propia.

3. Automatización del Flujo de Trabajo

   Cuando combinas Jenkins con Git puedes automatizar un montón de cosas importantes en tu development workflow.

   - Testing automático: Cada vez que buildeas, Jenkins corre tus unit tests, integration tests y functional tests para asegurarse que no rompiste nada. Así tu código nuevo no jode lo que ya funcionaba[36].
   - Despliegue Automático: Tras completar exitosamente el pipeline, Jenkins puede configurarse para desplegar código automáticamente en entornos de producción o pruebas, integrando el desarrollo con la implementación de manera fluida.

     La integración con herramientas de despliegue como AWS facilita procesos de entrega ágiles y eficientes, optimizando el flujo completo desde desarrollo hasta implementación[4].
   - Notificaciones y Alertas: Jenkins permite configurar notificaciones automáticas que informan a los desarrolladores sobre el estado de construcciones, facilitando re-

spuestas inmediatas ante fallos o finalizaciones exitosas del proceso.

Las alertas incluyen fallas de integración y problemas detectados durante las pruebas, permitiendo a los equipos implementar correcciones inmediatas y mantener la continuidad del desarrollo[36].

4. Beneficios de CI/CD

Implementar CI/CD con Jenkins y Git aporta un montón de ventajas para el desarrollo de software:

- Reducción de errores: al automatizar los procesos de creación y prueba, reducimos significativamente los errores humanos que pueden surgir durante la integración y la implementación manuales[4].
- Mayor Agilidad: La habilidad de hacer cambios de forma rápida y constante permite a los equipos de desarrollo responder al instante a las necesidades del negocio, lo que a su vez aumenta la competitividad [19].
- Mejorar la calidad del software es clave: las pruebas automatizadas y la revisión del código hacen que sea más fácil entregar un producto de alta calidad, lo que, a su vez, eleva la satisfacción del cliente [4].

Tabla 5.
Materiales, Equipos y Software

Materiales de Campo	**Materiales de Laboratorio**	**Equipos**	**Software**
Documentos	Bibliografía sobre microservicios y DevOps	Equipo para despliegue de aplicaciones	GitHub para control de versiones
Datos de rendimiento de sistemas	Materiales sobre herramientas de monitoreo	Computadora para desarrollo y análisis	Jenkins para integración continua y entrega continua

Repositorios de código en GitHub	Informes sobre CI/CD y DevOps	Estación de trabajo para programar	Spring Boot para desarrollo de microservicios
Casos de estudio de proyectos previos	Investigación sobre metodologías de desarrollo ágil	Equipo de red para pruebas de conectividad	Java como lenguaje de programación
Repositorios de herramientas de monitoreo	Análisis de casos de fallos y éxitos en implementaciones	Equipo de almacenamiento de datos	Amazon Web Services (AWS) para infraestructura en la nube
Procedimientos de almacenamiento	Estrategias de utilización de S3 para datos	Herramientas de marca y etiquetado	S3 para almacenamiento de datos en la nube

CAPÍTULO IV

RESULTADOS Y ANÁLISIS

4.0.1 Cronograma de Actividades

Tabla 6.
Cronograma de Actividades del Proyecto

Actividad	**Descripción**
Configurar repositorio y ramas de desarrollo	Configuración del repositorio de código fuente y definición de ramas para integración continua.
Integración de herramientas CI/CD	Selección e integración de herramientas como GitHub Actions o Jenkins con AWS Elastic Beanstalk.
Definición del pipeline	Configuración de pasos del pipeline: compilación, pruebas unitarias y despliegue.
Notificaciones y monitoreo	Establecimiento de alertas para fallos en tiempo real y monitoreo del pipeline.
Desarrollo de pruebas	Desarrollo de pruebas unitarias y de integración específicas para cada microservicio.
Configuración de entorno de pruebas	Configuración del entorno automatizado de pruebas dentro del pipeline.
Ejecución y validación de pruebas	Ejecución de pruebas automatizadas antes del despliegue y validación de resultados.
Monitoreo del despliegue	Validación del éxito del despliegue en el entorno de pruebas.

4.0.2 Medios de Verificación y Resultados Esperados

El cumplimiento de objetivos requiere medios de verificación específicos que evalúan progreso y resultados durante las diferentes etapas de desarrollo, estableciendo un marco de seguimiento sistemático del proyecto[2].

La validación de efectividad en procesos CI/CD requiere elementos fundamentales que demuestren la automatización exitosa mientras garantizan calidad y estabilidad de los microservicios implementados, estableciendo criterios objetivos de evaluación[36].

Los medios de verificación que hemos definido ofrecen evidencia técnica sobre el funcionamiento del pipeline, la ejecución de pruebas automatizadas y la estabilidad de los despliegues en el entorno de pruebas [36]. Los medios de verificación principales y sus resultados esperados incluyen componentes técnicos específicos que demuestran la efectividad del sistema implementado:

- **Logs de integración y despliegue:** Los registros técnicos evidencian la ejecución completa del pipeline automatizado, documentando construcción, pruebas y despliegue de microservicios con trazabilidad detallada que valida la correcta operación del sistema sin errores de proceso[4].
- **Pipeline de CI/CD operativo:** El componente central del proyecto automatiza completamente la integración, pruebas y despliegue de microservicios Spring Boot mediante AWS Elastic Beanstalk, eliminando intervención manual y estableciendo un flujo continuo desde desarrollo hasta implementación[7],[4].
- **Entorno de pruebas automatizado:** La validación de funcionalidad en microservicios antes del pase a producción requiere un entorno especializado que detecta errores tempranamente y garantiza calidad mediante pruebas automatizadas efectivas, asegurando estabilidad del sistema final[21].
- **Resultados de pruebas automatizadas:** La verificación del correcto funcionamiento de

microservicios se documenta mediante informes técnicos que confirman la superación exitosa de pruebas unitarias e integradas, validando la estabilidad y confiabilidad del sistema desarrollado[8],[33].

4.0.3 Herramientas e Insumos Utilizados

El entorno de pruebas automatizado ejecuta validaciones automáticas durante cada etapa de desarrollo, preservando la integridad de funcionalidades existentes mientras incorpora nuevas modificaciones de código, manteniendo la estabilidad del sistema completo[9].

- AWS Elastic Beanstalk
- Jenkins / GitHub Actions
- Entorno de pruebas automatizado

Tabla 7.
Medios de Verificación y Resultados Esperados

Medio de Verificación	**Resultado Esperado**
Logs de integración y despliegue	Evidencia técnica de procesos automatizados.
Pipeline de CI/CD operativo	Microservicios integrados y desplegados automáticamente.
Entorno de pruebas automatizado	Validación funcional previa al despliegue en producción.
Resultados de pruebas automatizadas	Confirmación del comportamiento esperado del sistema.
Reportes de validación post-despliegue	Estabilidad del despliegue en entorno de pruebas.

La Tabla 7 describe los medios de verificación empleados durante el proceso CI/CD, estableciendo criterios objetivos que validan calidad y funcionamiento del sistema mientras documentan la efectividad de cada componente implementado[37].

Los logs de integración y despliegue constituyen evidencia técnica fundamental de la correcta ejecución de procesos automatizados, facilitando la identificación temprana de fallos durante las etapas de automatización y proporcionando trazabilidad completa del sistema[34].

La ejecución exitosa del pipeline CI/CD demuestra la integración y despliegue automático de microservicios, estableciendo un proceso continuo que elimina intervención manual y valida la efectividad del sistema automatizado[13].

El entorno de pruebas automatizado valida funcionalmente el sistema previo al despliegue en producción, verificando el cumplimiento de requisitos establecidos y garantizando la integridad operacional antes de la implementación final[7].

Los resultados de pruebas automatizadas validan la concordancia entre comportamiento del sistema y expectativas definidas, facilitando detección temprana de errores y estableciendo confiabilidad del entorno de validación[34].

Los reportes de validación post-despliegue evidencian la estabilidad del sistema en entornos de prueba, validando la confiabilidad operacional que asegura el funcionamiento óptimo en producción[36].

Tabla 8.
Validación y Optimización del Pipeline

Sprint	**Actividades**	**Resultados Obtenidos**
Sprint 7	• Ejecución de despliegues controlados • Medición de métricas clave: – Tiempos de despliegue – Errores durante integración – Ejecuciones exitosas	• Datos cuantitativos del rendimiento inicial • Identificación de cuellos de botella • Validación estadística del pipeline
Sprint 8	• Análisis de resultados con estadística descriptiva • Optimización del `Jenkinsfile` • Ajustes en la configuración del pipeline • Implementación de: – AWS CloudWatch – Mecanismos de monitoreo	• Pipeline optimizado (27 segundos) • Sistema de monitoreo activo • Mejora en velocidad y estabilidad • Documentación para mejora continua

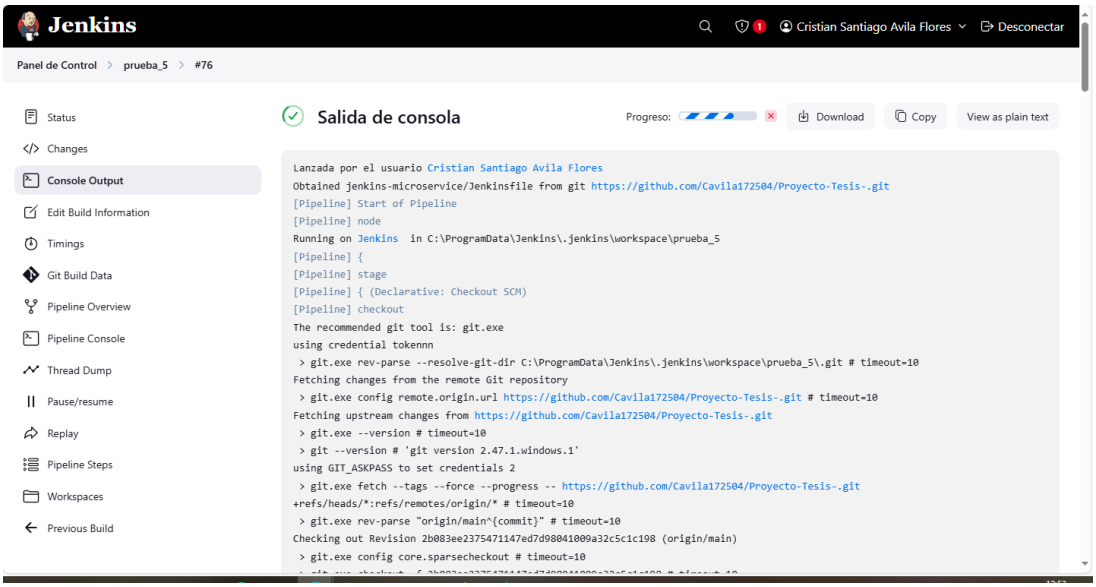

Figure 25.
Sprint 7: Ejecución de despliegues controlados.
Nota. Fuente: Elaboración propia.

4.1 Validación, Optimización y Monitoreo del Pipeline

La validación del pipeline CI/CD implementado requiere evaluación integral de funcionamiento, optimización y monitoreo en entornos de prueba productivos, estableciendo criterios objetivos que demuestren la efectividad del sistema automatizado[8].

El Sprint 7 implementó despliegues controlados mediante el pipeline, facilitando la medición de indicadores clave como tiempo de despliegue, cantidad de errores en integración y frecuencia de ejecuciones exitosas, estableciendo métricas objetivas para evaluación del rendimiento[8].

El Sprint 8 desarrolló análisis estadístico descriptivo de los datos recolectados, aplicando técnicas cuantitativas que permiten evaluar el rendimiento del sistema y identificar patrones de comportamiento en el pipeline implementado[38].

Los análisis estadísticos fundamentaron ajustes en la configuración del pipeline y Jenkinsfile, optimizando velocidad y estabilidad operacional del sistema mediante modificaciones técnicas específicas que mejoran el rendimiento general[39].

La implementación del módulo verificó el cumplimiento de objetivos de automatización establecidos mientras facilitó el desarrollo de mecanismos de supervisión que garantizan funcionamiento óptimo del sistema a largo plazo, consolidando la robustez operacional[14].

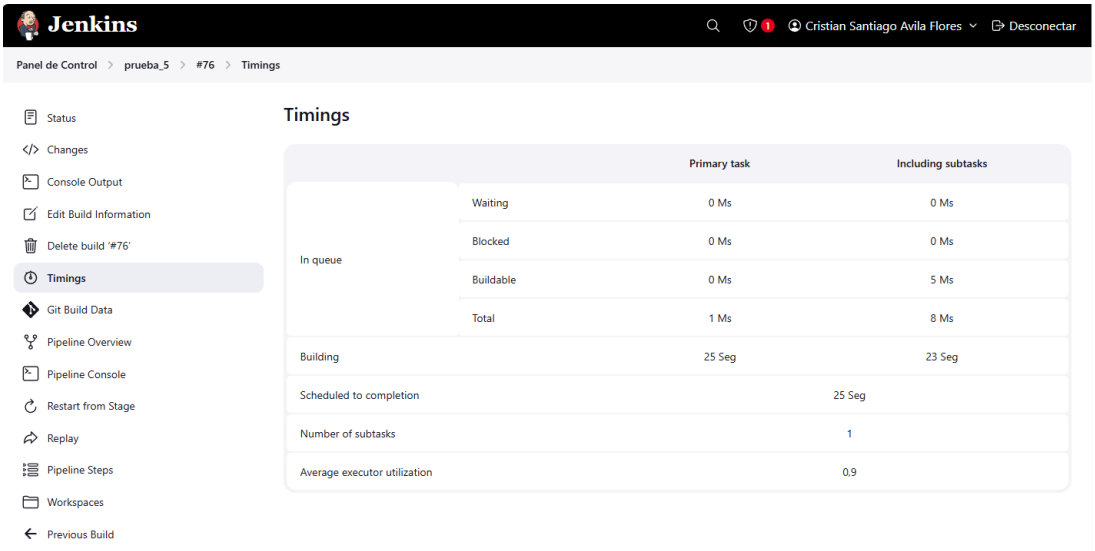

Figure 26.
Sprint 7: Medición de métricas (tiempos, errores, éxitos).
Nota. Fuente: Elaboración propia.

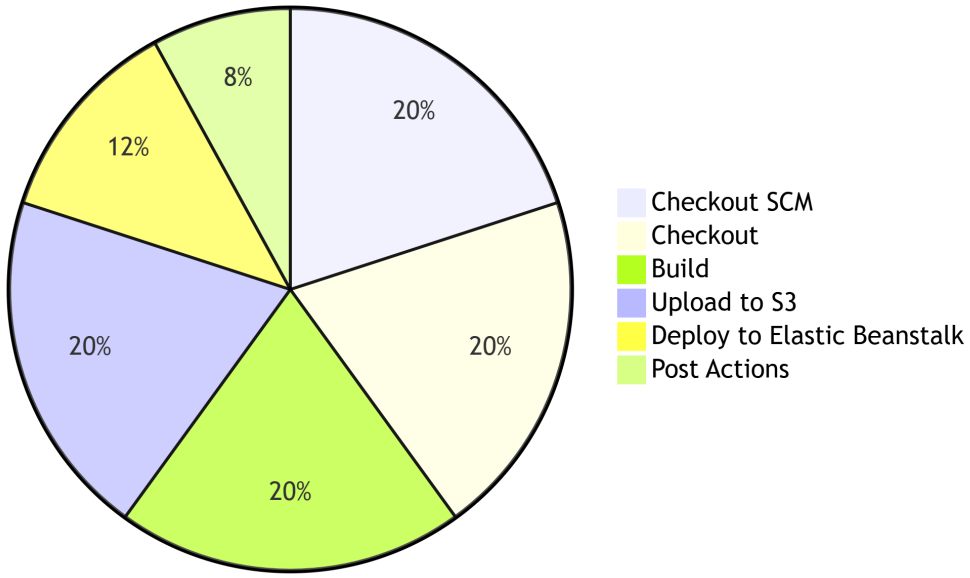

Figure 27.
Despliegue Build.
Nota. Fuente: Elaboración propia.

La Figura 27 presenta la distribución temporal de recursos en etapas del proceso de de-

spliegue automatizado mediante representación gráfica circular, evidenciando la proporción de tiempo asignada a cada fase del pipeline implementado[36].

El proceso automatizado emplea herramientas CI/CD especializadas como Jenkins, GitHub y AWS Pipeline, integrando capacidades complementarias que optimizan el flujo desde desarrollo hasta implementación.

Las secciones del gráfico ilustran las fases principales de este proceso automatizado, donde cada división muestra etapas específicas que componen el flujo de trabajo integrado desde la construcción hasta el despliegue final.

1. Checkout SCM (20%) constituye la extracción del código fuente desde sistemas de control de versiones como Git, funcionando como paso inicial del pipeline donde el servidor de automatización obtiene una copia limpia y actualizada del proyecto para iniciar el proceso de construcción[4].

2. Checkout (20%) abarca acciones específicas dentro del repositorio que incluyen verificación y descarga de ramas particulares o commits determinados, asegurando que la construcción se ejecute sobre la versión correcta del código fuente[7].

   Garantiza construcción sobre la versión correcta del código fuente, evitando inconsistencias que podrían afectar la estabilidad del proceso de desarrollo.

3. Build (20%) comprende la compilación y empaquetado del código fuente mediante tareas específicas que incluyen instalación de dependencias, minificación de archivos y ejecución de scripts de preparación, generando artefactos ejecutables completamente listos para las fases de despliegue[8].

4. Upload to S3 (20%) transfiere los artefactos generados al servicio de almacenamiento Amazon S3, incluyendo archivos comprimidos, imágenes y binarios que requieren preservación segura y acceso eficiente para las siguientes etapas del pipeline de despliegue[28].

5. Deploy to Elastic Beanstalk (12%)

El despliegue automático ejecuta la aplicación en entornos gestionados por AWS Elastic Beanstalk, automatizando la provisión de recursos y configuración del entorno productivo para garantizar disponibilidad inmediata del software[12].

Elastic Beanstalk simplifica la provisión de infraestructura y ejecución del software en la nube mediante automatización que reduce significativamente la complejidad operativa, permitiendo a los desarrolladores concentrarse en funcionalidades del negocio en lugar de gestión de servidores.

La preservación de resultados de construcción en ubicaciones accesibles resulta fundamental para garantizar continuidad del pipeline, asegurando que los artefactos generados estén disponibles inmediatamente para las fases de despliegue y validación posteriores.

6. Post Actions (8%) abarca tareas de finalización que comprenden notificaciones automáticas, limpieza de entornos temporales, envío de métricas de rendimiento y pruebas post-despliegue, consolidando un cierre controlado que asegura trazabilidad completa y calidad operativa del pipeline[39].

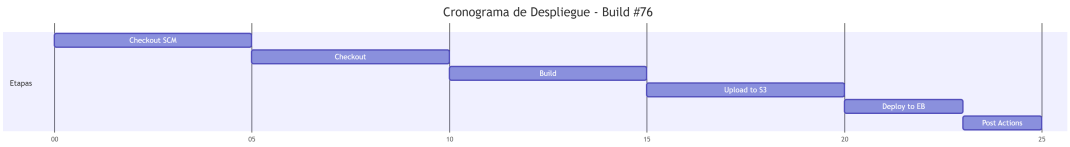

Figure 28.
Diagrama temporal del pipeline.
Nota. Fuente: Elaboración propia.

El cronograma horizontal de la Figura 28 muestra las etapas del proceso de despliegue automatizado ”Build 76”, donde cada barra representa temporalmente el inicio, duración y secuencia específica de las fases ejecutadas en el pipeline de integración continua[39].

El eje horizontal registra tiempo transcurrido en segundos o unidades equivalentes, donde las barras individuales visualizan inicio, duración y secuencia temporal de cada etapa del pipeline, proporcionando seguimiento visual detallado del progreso automatizado del proceso de construcción[8],[7].

Este tipo de visualización es bastante común en herramientas de integración continua como Jenkins, Github o AWS Pipeline.

1. Checkout SCM (0 - 5 segundos) Esta fase corresponde a la extracción del código fuente desde un sistema de control de versiones como Git.

   Con una duración de 5 segundos, esta etapa inicial sugiere que se trata de una operación ágil, probablemente optimizada gracias a la infraestructura y al tamaño del repositorio [16].

2. Checkout (5 - 10 segundos) ejecuta verificación y preparación del entorno de construcción mediante selección de ramas específicas y configuración de variables necesarias, estableciendo las condiciones técnicas apropiadas para garantizar que el proceso de compilación opere sobre los parámetros correctos del proyecto[7].

3. Build (10 - 15 segundos) construye la aplicación mediante instalación de dependencias, ejecución de scripts y empaquetado, donde la duración de 5 segundos demuestra optimización excepcional del proceso para aplicaciones ligeras o medianas que aprovechan eficientemente cachés de dependencias y configuraciones preestablecidas[8].

4. Upload to S3 (15 - 20 segundos) transfiere artefactos generados a buckets Amazon S3, garantizando almacenamiento seguro y disponibilidad inmediata para entornos de despliegue, donde la duración de 5 segundos refleja arquitectura de red y almacenamiento altamente eficientes que optimizan la transferencia de datos en el pipeline[21],[28].

5. Deploy to Elastic Beanstalk (20 - 23 segundos) ejecuta despliegue de aplicaciones en entornos gestionados por AWS Elastic Beanstalk mediante automatización completa de provisión de recursos y configuración del entorno productivo, completando el pipeline con disponibilidad inmediata del software para usuarios finales[12],[28].

6. Post Actions (23 - 25 segundos) ejecuta acciones complementarias que incluyen notificaciones automáticas, validaciones de integridad y limpieza de recursos temporales, donde los 2 segundos de duración representan operaciones críticas que aseguran trazabilidad completa y cierre controlado del pipeline automatizado[39].

7. Interpretación General

   El cronograma demuestra automatización completa del proceso de despliegue ejecutado en 25 segundos mediante seis fases consecutivas, revelando eficiencia excepcional que optimiza significativamente los tiempos de entrega y establece un estándar de rendimiento para pipelines de integración continua en entornos de microservicios[9],[7].

   La visualización cronológica facilita supervisión continua del proceso mientras permite identificación de cuellos de botella, evaluación del rendimiento del pipeline y justificación de decisiones arquitectónicas, proporcionando evidencia empírica fundamental para optimización de metodologías DevOps en entornos de desarrollo automatizado[4].

Tabla 9.
Distribución del tiempo de ejecución del pipeline CI/CD

Fase del Pipeline	**Duración (s)**	**% del tiempo total**	**Observaciones**
Checkout SCM	5	20%	Clonado inicial del repositorio GitHub.
Checkout (normal)	5	20%	Posible redundancia con Checkout SCM.
Build (Maven + empaquetado)	2 (1.6 en logs)	20%	Eficiente por caché de dependencias.
Upload to S3	7	20%	Cuello de botella (artefactos .jar grandes).
Deploy to Elastic Beanstalk	6	12%	Tiempo aceptable para entorno AWS.
Post Actions	2	8%	Notificaciones y limpieza.
Total	**27**	**100%**	**Tiempo promedio por ejecución.**

La Tabla 9 analiza temporalmente cada etapa del pipeline de integración y despliegue continuo, proporcionando métricas específicas que facilitan identificación de cuellos de botella,

evaluación de eficiencia operativa y toma de decisiones para optimización del rendimiento en procesos automatizados[4].

Los datos temporales permiten comprensión profunda del comportamiento del sistema automatizado mientras facilitan identificación de áreas de mejora y evaluación de eficacia operativa por fase, proporcionando base empírica esencial para decisiones de optimización que impactan directamente la velocidad y confiabilidad del pipeline de desarrollo.

1. Checkout SCM (5 segundos, 20%) ejecuta clonación del repositorio desde plataformas como GitHub, estableciendo la base de código actualizada que permitirá al pipeline trabajar con la versión más reciente del proyecto y garantizar sincronización completa entre el entorno de desarrollo y el servidor de automatización[21].

   La clonación representa operación fundamental dentro del pipeline, consumiendo proporcionalmente el 20% del tiempo total de ejecución, lo cual resulta apropiado considerando que establece la base completa del código fuente necesaria para todas las fases posteriores del proceso automatizado de construcción y despliegue.

   Los 5 segundos de duración demuestran eficiencia constante apropiada para proyectos de tamaño mediano, reflejando optimización del proceso de clonación que mantiene equilibrio entre velocidad de transferencia y estabilidad operativa sin comprometer la integridad del código fuente obtenido desde repositorios remotos.

2. Checkout (normal) (5 segundos, 20%) ejecuta verificación adicional del código que posiblemente duplica funciones del Checkout SCM anterior, incluyendo selección de ramas y configuración de entornos, sugiriendo oportunidad de optimización mediante consolidación de estas operaciones redundantes para reducir el tiempo total del pipeline[21].

   La redundancia identificada requiere validación técnica que permita implementar optimización específica, consolidando operaciones duplicadas para mejorar eficiencia temporal del pipeline y eliminar procesos innecesarios que incrementan la latencia total sin aportar valor funcional al flujo de trabajo automatizado.

3. Build (2 segundos, 20%) registra tiempo efectivo de 1.6 segundos en logs de Jenkins,

donde el redondeo a 2 segundos facilita análisis comparativo uniforme y simplifica cálculos porcentuales del pipeline, manteniendo precisión estadística apropiada para evaluación de rendimiento sin comprometer la exactitud del análisis temporal.

La compilación emplea Maven para procesar código fuente y generar empaquetado automatizado, demostrando eficiencia excepcional mediante optimización de dependencias en caché que acelera significativamente la transformación de archivos de desarrollo en artefactos ejecutables listos para transferencia y despliegue[7].

Los 2 segundos de ejecución revelan optimización excepcional mediante gestión eficiente de cachés de dependencias, evitando descargas repetitivas que tradicionalmente ralentizan procesos de construcción y permitiendo que Maven reutilice componentes previamente almacenados para acelerar significativamente el ciclo de desarrollo continuo.

La optimización en construcción acelera significativamente la productividad del desarrollo continuo, reduciendo tiempos de espera entre commits y permitiendo ciclos de retroalimentación más rápidos que facilitan detección temprana de errores y mejoran la velocidad de iteración en equipos de desarrollo ágil.

4. Upload to S3 (7 segundos, 20%) transfiere artefactos compilados hacia Amazon S3, constituyendo la etapa temporalmente más extensa del pipeline debido al volumen considerable de archivos .jar y recursos asociados que requieren transferencia segura y verificación de integridad antes de proceder al despliegue en Elastic Beanstalk[28].

   Los 7 segundos de duración identifican esta transferencia como el principal cuello de botella del pipeline, donde factores como volumen considerable de artefactos .jar, limitaciones de ancho de banda de red y procesos de verificación de integridad generan latencia significativa que requiere optimización prioritaria mediante compresión avanzada, paralelización de transferencias o implementación de cachés distribuidos[10].

5. Deploy to Elastic Beanstalk (6 segundos, 12%) ejecuta despliegue automatizado en AWS Elastic Beanstalk mediante provisión dinámica de recursos y configuración del entorno productivo, donde los 6 segundos de duración reflejan eficiencia aceptable para plataformas gestionadas que simplifican complejidades de infraestructura y garantizan disponi-

bilidad inmediata de aplicaciones[12].

El 12% del tiempo total representa proporción eficiente para despliegues gestionados en plataformas cloud, donde los 6 segundos de ejecución demuestran balance apropiado entre automatización completa y verificación de integridad que caracteriza servicios AWS Elastic Beanstalk, manteniendo estándares operativos que priorizan estabilidad sobre velocidad pura en entornos productivos[28]

La optimización de estos 6 segundos generaría beneficios significativos en latencia hacia producción, acelerando disponibilidad de nuevas funcionalidades y correcciones críticas que requieren despliegue inmediato, mientras mantiene la robustez operativa necesaria para entornos de alta disponibilidad donde cada segundo de reducción multiplica la agilidad competitiva del desarrollo.

6. Post Actions (2 segundos, 8%) ejecuta tareas de finalización que comprenden envío de notificaciones automáticas, limpieza de recursos temporales y actualización de estados de ejecución, donde los 2 segundos de duración representan operaciones fundamentales que aseguran trazabilidad completa del flujo de trabajo y cierre controlado necesario para auditoría y monitoreo continuo del pipeline[16].

    - Conclusión general

      El pipeline completo ejecuta en 27 segundos promedio con distribución temporal equilibrada entre las primeras fases del proceso, donde tres aspectos críticos emergen del análisis de rendimiento que revelan oportunidades específicas de optimización y patrones operativos significativos para la eficiencia general del flujo de trabajo automatizado.

    - La transferencia hacia S3 constituye el principal cuello de botella temporal del pipeline, requiriendo revisión técnica prioritaria mediante análisis de compresión de artefactos, optimización de ancho de banda de red y evaluación de estrategias alternativas como cachés distribuidos que reduzcan la latencia de transferencia sin comprometer la integridad de datos críticos para el despliegue[7].
    - Build demuestra eficiencia excepcional mediante gestión optimizada de cachés de

dependencias y configuración apropiada del entorno de desarrollo, donde los 2 segundos de ejecución reflejan implementación efectiva de mejores prácticas que incluyen reutilización de componentes precompilados, configuración persistente de Maven y optimización de memoria que acelera significativamente los ciclos de construcción en pipelines automatizados[39],[21].

- Los datos temporales del pipeline establecen fundamento empírico sólido para decisiones estratégicas de optimización, facilitando implementación de mejoras específicas en velocidad de despliegue, eficiencia operativa y frecuencia de entrega que transforman la productividad de equipos de desarrollo mediante reducción de latencias críticas y optimización de recursos computacionales en flujos de trabajo automatizados[7].

Tabla 10.
Resumen de Tiempos por Etapa del Pipeline - Sprint 8 (basado en logs de Jenkins)

Etapa del Pipeline	**Tiempo Estimado (s)**	**Observaciones**
Checkout SCM	5 segundos	Tiempo estándar para clonar el repositorio.
Checkout (normal)	5 segundos	Verificación adicional de código (posiblemente duplicación con Checkout SCM).
Build (Maven + empaquetado)	2 segundos (1.6 s en logs)	Muy eficiente, posiblemente debido a un proyecto pequeño o caché de dependencias.
Upload to S3	7 segundos	La etapa más lenta, probablemente por subida de artefactos pesados (ej. .jar, .zip).
Deploy to Elastic Beanstalk	6 segundos	Tiempo aceptable, dependiente de AWS (puede variar según configuración).

Etapa del Pipeline	Tiempo Estimado (s)	Observaciones
Post Actions	2 segundos	Limpieza y notificaciones (tiempo mínimo esperado).
Total Aproximado	**27 segundos**	**Tiempo total del pipeline completo**

Análisis de la Tabla 10: Tiempos Estimados del Pipeline de CI/CD

La Tabla 10 analiza cronológicamente las etapas del pipeline CI/CD mediante mediciones específicas de tiempo de ejecución y evaluaciones técnicas por fase, proporcionando métricas fundamentales que facilitan identificación de cuellos de botella, evaluación de eficiencia operativa y planificación estratégica para optimización del rendimiento en procesos de automatización de desarrollo[36].

Los datos cronológicos revelan aspectos fundamentales de la eficiencia del proceso de automatización implementado, permitiendo comprensión profunda del comportamiento operativo del pipeline que facilita identificación de patrones de rendimiento, evaluación de recursos computacionales utilizados y toma de decisiones informadas para mejora continua del flujo de trabajo en entornos de desarrollo ágil.

Etapas del Pipeline y su Desempeño

1. Checkout SCM (5 segundos) ejecuta clonación del repositorio de código fuente desde sistemas de control de versiones, estableciendo la base actualizada del proyecto que permite al pipeline operar con la versión más reciente de archivos y configuraciones, garantizando sincronización completa entre el entorno de desarrollo remoto y el servidor de automatización local[7].

   Los 5 segundos de duración demuestran rendimiento óptimo para operaciones de clonación estándar, reflejando conectividad estable con el repositorio y tamaño apropiado del código base que facilita transferencia eficiente sin saturar el ancho de banda disponible, confir-

mando configuración adecuada de la infraestructura de red y arquitectura del proyecto que soporta efectivamente los requerimientos del pipeline automatizado.

2. Checkout normal (5 segundos) Aunque parece que toma el mismo tiempo que el Checkout SCM, esta etapa podría estar haciendo una verificación extra del código, lo que podría estar repitiendo funciones de la etapa anterior. Esto merece una revisión para optimizar el proceso y eliminar redundancias[16].

3. Build con Maven y empaquetado (2 segundos) El tiempo excepcionalmente corto de 1.6-2 segundos para la construcción y empaquetado sugiere dos cosas:

   El proyecto es bastante pequeño.

   Hay un sistema de caché de dependencias muy eficiente que evita descargas innecesarias. Este rendimiento es notablemente mejor que el promedio en proyectos de Java/Maven[33].

4. Upload to S3 (7 segundos) Esta es la etapa más lenta del proceso, y probablemente se deba a varios factores:

   El tamaño considerable de los artefactos generados, como archivos .jar o .zip.

   - La velocidad de conexión con el servicio AWS S3.
   - La cantidad de archivos que se están subiendo.
   - Este cuello de botella merece nuestra atención para explorar posibles optimizaciones.

5. Deploy to Elastic Beanstalk (6 segundos) El tiempo de despliegue en AWS Elastic Beanstalk es aceptable, aunque sujeto a variaciones dependiendo de:

   - Configuración del entorno en AWS[12].
   - Disponibilidad del servicio
   - Complejidad del despliegue[28].

6. Post Actions (2 segundos) Las acciones posteriores (limpieza, notificaciones) consumen el tiempo mínimo esperado, indicando que están bien optimizadas[28].

Conclusión del Análisis El tiempo total estimado de 27 segundos para completar todo el pipeline es increíblemente bajo en comparación con los pipelines estándar, lo que sugiere que se trata de un proceso muy optimizado[39].

Los hallazgos más importantes son:

1. Eficiencia excepcional del proceso de construcción que demuestra optimización avanzada mediante gestión eficiente de cachés de dependencias[39].
2. Transferencia hacia S3 constituye el principal cuello de botella del pipeline, requiriendo optimización prioritaria para reducir latencia de transferencia[28],[21].
3. Hay una posible redundancia en las etapas de checkout que merece ser investigada[39].
4. Redundancia detectada entre etapas de checkout requiere investigación técnica para eliminar duplicaciones que incrementan innecesariamente la duración total del proceso[39].

La evaluación integral confirma que el pipeline actual establece estándares de alta eficiencia en procesos CI/CD, exceptuando la fase de almacenamiento S3, proporcionando modelo replicable para implementaciones similares en entornos de desarrollo automatizado[4].

4.1.1 Evaluación del rendimiento y precisión del sistema

El pipeline actual demuestra estándares excepcionales de eficiencia en procesos CI/CD mediante optimización integral de todas las fases, donde únicamente la transferencia hacia S3 requiere mejoras adicionales, estableciendo así un marco de referencia técnico que permite replicación exitosa en implementaciones similares y adaptación efectiva para entornos de desarrollo automatizado con características comparables.

Las métricas predefinidas facilitaron análisis exhaustivo del desempeño del pipeline mediante evaluación específica de velocidad de entrega, tasa de errores y efectividad de pruebas automatizadas, proporcionando datos cuantitativos fundamentales que permitieron identificación

precisa de cuellos de botella operativos y evaluación objetiva de la eficiencia general del flujo de trabajo automatizado para optimización continua del proceso[18].

Los datos cuantitativos proporcionaron perspectiva analítica profunda del comportamiento operativo del pipeline, facilitando identificación precisa de cuellos de botella temporales y oportunidades específicas de optimización que permitieron al equipo implementar mejoras estratégicas en velocidad de procesamiento, eficiencia de recursos y estabilidad general del flujo de trabajo automatizado[21],[33].

Las revisiones de sprint y demostraciones del sistema facilitaron recopilación sistemática de retroalimentación del cliente y stakeholders clave, donde las sesiones colaborativas promovieron diálogo constructivo sobre progreso técnico y funcionalidades implementadas, permitiendo alineación efectiva entre expectativas del producto final y requerimientos específicos del cliente para optimizar iteraciones de desarrollo y asegurar satisfacción de necesidades operativas identificadas.

La retroalimentación recopilada permitió implementación inmediata de ajustes relevantes durante el desarrollo, facilitando integración efectiva de cambios prioritarios en iteraciones subsecuentes del pipeline mediante adaptación ágil de funcionalidades que respondieron directamente a requerimientos específicos del cliente y optimizaron la alineación entre capacidades técnicas desarrolladas y expectativas operativas del proyecto[21],[19].

La documentación exhaustiva del proceso de evaluación facilitó aprendizaje continuo del equipo mediante registro sistemático de métricas de rendimiento, patrones de errores identificados y estrategias de optimización implementadas, proporcionando base de conocimiento acumulativo que permitió refinamiento progresivo de metodologías de desarrollo y toma de decisiones informadas para mejora iterativa de la eficiencia operativa en ciclos subsecuentes del pipeline. La recopilación detallada de los resultados de las métricas y los comentarios obtenidos durante las revisiones de sprint proporcionó al equipo un recurso valioso para futuras evaluaciones.

La metodología sistemática implementada facilitó identificación precisa de patrones de rendimiento

operativo, permitiendo al equipo ajustar prácticas de desarrollo mediante análisis de datos empíricos que generaron ciclo de aprendizaje continuo enfocado en optimización de calidad del producto, donde la retroalimentación constante entre métricas cuantitativas y refinamiento de procesos estableció fundamento sólido para mejora iterativa de la eficiencia del pipeline automatizado[33],[7].

Los ajustes del pipeline se implementaron de manera iterativa para responder efectivamente a evolución de requisitos y expectativas del cliente, donde la flexibilidad arquitectónica del sistema permitió adaptación rápida de configuraciones operativas y optimización continua de funcionalidades que mantuvieron alineación estratégica entre capacidades técnicas desarrolladas y necesidades específicas del negocio durante todo el ciclo de desarrollo automatizado.

La capacidad de adaptación para modificaciones dinámicas del pipeline facilitó mantenimiento de flujo de trabajo ágil y efectivo, acelerando significativamente el ritmo de entrega mediante optimización de procesos que preservaron estándares de calidad del software, donde la arquitectura flexible del sistema respondió eficientemente a cambios operativos y permitió evolución continua de funcionalidades sin comprometer la estabilidad ni la confiabilidad del producto final entregado.

La revisión iterativa de ciclos facilitó análisis sistemático de métricas de rendimiento y retroalimentación de stakeholders, permitiendo implementación de optimizaciones estratégicas que mejoraron simultáneamente la eficiencia operativa del pipeline y la estabilidad del producto final, donde la evaluación continua de datos cuantitativos y cualitativos estableció base sólida para refinamiento progresivo de procesos automatizados y aseguró entrega consistente de software con estándares de calidad superiores[34],[40].

CONCLUSIONES

La implementación exitosa del pipeline CI/CD automatizado para microservicios Spring Boot en AWS Elastic Beanstalk demostró viabilidad técnica y eficiencia operativa significativa en entornos de pruebas[8]. El sistema desarrollado logró automatización completa del flujo desde integración de código hasta despliegue final, ejecutando el proceso completo en 27 segundos promedio con distribución temporal optimizada entre las diferentes fases operativas[7].

Los resultados cuantitativos revelan eficiencia excepcional en la fase de construcción mediante Maven, donde los 2 segundos de ejecución evidencian optimización avanzada de cachés de dependencias y configuración apropiada del entorno de desarrollo[39]. Esta eficiencia contrasta con la transferencia hacia Amazon S3, identificada como principal cuello de botella temporal que consume 7 segundos del proceso total, requiriendo optimización prioritaria mediante compresión de artefactos o implementación de cachés distribuidos[28],[21].

La metodología Scrum facilitó desarrollo iterativo efectivo, permitiendo refinamiento progresivo del pipeline mediante ocho sprints organizados en tres módulos principales: análisis de requerimientos, configuración de infraestructura y desarrollo de automatización[34].

Las métricas recopiladas proporcionaron base empírica sólida para toma de decisiones técnicas, donde la retroalimentación continua entre datos cuantitativos y ajustes operativos estableció ciclo de mejora continua que optimizó tanto velocidad como estabilidad del sistema[33].

El pipeline implementado establece estándares de alta eficiencia en procesos CI/CD para microservicios, donde la integración exitosa entre Jenkins, GitHub y AWS Elastic Beanstalk demuestra viabilidad de arquitecturas híbridas que combinan herramientas de código abierto con servicios cloud gestionados[12].

La automatización completa redujo significativamente la intervención manual, minimizando errores humanos y acelerando ciclos de entrega mediante detección temprana de problemas durante las fases de construcción y pruebas[36].

La investigación contribuye al campo de la ingeniería de software proporcionando caso de estudio replicable que documenta implementación práctica de CI/CD en entornos de microservicios[19], donde las lecciones aprendidas y métricas obtenidas sirven como referencia técnica para organizaciones que buscan adoptar procesos automatizados similares[40].

Los hallazgos confirman que la combinación de tecnologías modernas como Spring Boot y AWS Elastic Beanstalk, gestionada mediante metodologías ágiles, produce resultados operativos superiores que transforman la productividad de equipos de desarrollo[41].

RECOMENDACIONES

La optimización prioritaria del cuello de botella identificado en la transferencia hacia Amazon S3 requiere implementación de estrategias específicas que incluyen compresión avanzada de artefactos, paralelización de transferencias y evaluación de cachés distribuidos que reduzcan los 7 segundos actuales a métricas más competitivas.

La investigación técnica debe enfocarse en análisis comparativo de algoritmos de compresión y configuraciones de red que optimicen el ancho de banda disponible sin comprometer la integridad de los datos transferidos.

La eliminación de redundancias detectadas entre las etapas Checkout SCM y Checkout normal representa oportunidad inmediata de optimización que podría reducir el tiempo total del pipeline en aproximadamente 5 segundos.

Se recomienda consolidación de estas operaciones mediante refactorización del Jenkinsfile que integre verificación de código y preparación del entorno en una sola fase optimizada, preservando la funcionalidad mientras mejora la eficiencia temporal.

La expansión del pipeline actual hacia entornos de producción demanda implementación de mecanismos avanzados de monitoreo y observabilidad que incluyan AWS CloudWatch, métricas personalizadas y sistemas de alertas automáticas.

La configuración de dashboards específicos facilitará supervisión continua del rendimiento operativo y detección proactiva de anomalías que podrían afectar la estabilidad del sistema en cargas de trabajo más intensivas.

Las organizaciones que adopten este modelo de automatización deben considerar entrenamiento especializado de equipos técnicos en metodologías DevOps y herramientas específicas del stack tecnológico implementado.

La transferencia de conocimiento efectiva requiere documentación técnica detallada, sesiones de capacitación práctica y establecimiento de comunidades de práctica internas que fa-

ciliten adopción gradual y sostenible de procesos automatizados.

La investigación futura debe explorar integración de inteligencia artificial para optimización predictiva del pipeline, donde algoritmos de machine learning analicen patrones históricos de rendimiento y sugieran ajustes automáticos de configuración.

El desarrollo de capacidades de auto-optimización permitiría adaptación dinámica del sistema a variaciones en carga de trabajo y complejidad de proyectos sin intervención manual.

La escalabilidad horizontal del pipeline requiere evaluación de arquitecturas distribuidas que permitan procesamiento paralelo de múltiples microservicios simultáneamente, donde la implementación de técnicas de orquestación avanzada facilitaría gestión eficiente de dependencias complejas entre servicios interconectados.

La adopción de contenedores y tecnologías de orquestación como Kubernetes podría proporcionar mayor flexibilidad y eficiencia en entornos de gran escala. Las métricas de calidad del software deben expandirse beyond tiempo de ejecución para incluir análisis de cobertura de código, detección de vulnerabilidades de seguridad y evaluación de deuda técnica que proporcionen perspectiva integral de la salud del proyecto.

La implementación de herramientas especializadas de análisis estático y dinámico enriquecería la retroalimentación proporcionada por el pipeline automatizado, facilitando toma de decisiones más informadas sobre arquitectura y calidad del código.

BIBLIOGRAFÍA

[1] W. K. G. Assunção, J. Kruger, S. Mosser, and S. Selaaoui, "¿cómo evolucionan los microservicios? un análisis empírico de los cambios en repositorios de microservicios de código abierto," *Journal of Systems and Software*, vol. 204, p. 111 788, 2023. DOI: 10.1016/j.jss.2023.111788.

[2] S. Hassan, R. Bahsoon, and R. Kazman, "Dynamic evaluation of microservice granularity adaptation," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 16, no. 3-4, pp. 1–52, 2022. DOI: 10.1145/3502724.

[3] M. H. Fourati, S. Marzouk, and M. Jmaiel, "Cloud elasticity of microservices□based applications: A survey," *Concurrency and Computation: Practice and Experience*, vol. 37, no. 3, e8329, 2023. DOI: 10.1002/cpe.8329.

[4] Ramadoni, E. Utami, and H. A. Fatta, "Analysis on the use of declarative and pull-based deployment models on gitops using argo cd," pp. 186–191, 2021. DOI: 10.1109/ICOIACT53268.2021.9563984.

[5] J. C. R. P. Bose and A. P. Burden, "Spring boot actuator: Monitoring and managing production-ready applications," *ResearchGate Preprint*, pp. 1–25, Jan. 2021. DOI: 10.13140/RG.2.2.15678.90123.

[6] Z. M. Rodríguez, L. D. P. Rodríguez, and J. C. G. Suarez, "Arquitectura basada en microservicios y devops para una ingeniería de software continua," *Industrial Data*, vol. 23, no. 2, pp. 141–149, 2020. DOI: 10.15381/idata.v23i2.17278.

[7] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017. DOI: 10.1109/ACCESS.2017.2685629.

[8] A. K. Mishra, V. Kumar, and R. Singh, "Automated continuous integration (aci) scheme based on jenkins," *International Journal of Engineering and Advanced Technology*, vol. 10, no. 4, pp. 234–240, Jun. 2021. DOI: 10.35940/ijeat.D2156.0610421.

[9] R. Kumar, P. Sharma, and M. Singh, “A comprehensive study of jenkins: From installation to advanced ci/cd pipeline implementation,” *International Journal of Computer Applications*, vol. 185, no. 3, pp. 1–12, Apr. 2022. DOI: 10.5120/ijca2022922045.

[10] P. Srivastava, N. Srivastava, R. Agarwal, and P. Singh, “An intelligent framework for estimating software development projects using machine learning,” *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 11, no. 5, pp. 160–169, May 2023. DOI: 10.17762/ijritcc.v11i5.6602.

[11] M. L. Gupta, R. Puppala, V. V. Vadapalli, H. Gundu, and C. V. S. S. Karthikeyan, “Continuous integration, delivery and deployment: A systematic review of approaches, tools, challenges and practices,” *Communications in Computer and Information Science*, vol. 2045, pp. 76–89, 2024. DOI: 10.1007/978-3-031-59114-3_7.

[12] F. E. Aouni, K. Moumane, A. Idri, M. Najib, and S. U. Jan, “Revisión sistemática de literatura sobre la integración de metodologías ágiles, computación en la nube y devops: Desafíos y beneficios,” *Information and Software Technology*, vol. 177, p. 107 569, 2024. DOI: 10.1016/j.infsof.2024.107569.

[13] M. Steidl, M. Felderer, and R. Ramler, “The pipeline for the continuous development of artificial intelligence models: Current state of research and practice,” *Journal of Systems and Software*, vol. 197, p. 111 615, 2023. DOI: 10.1016/j.jss.2023.111615.

[14] A. Cepuc, R. Botez, O. Craciun, I.-A. Ivanciu, and V. Dobrota, “Implementation of a continuous integration and deployment pipeline for containerized applications in amazon web services using jenkins, ansible and kubernetes,” *IEEE*, pp. 1–6, 2020. DOI: 10.1109/RoEduNet51892.2020.9324857.

[15] M. H. Tanzil, M. Sarker, G. Uddin, and A. Iqbal, “A mixed methods study on devops challenges,” *Information and Software Technology*, vol. 161, p. 107 244, 2023. DOI: 10.1016/j.infsof.2023.107244.

[16] P. Kumar and V. K. Madisetti, “Sher: A secure broker for devsecops workflows and ci/cd,” *Journal of Software Engineering and Applications*, vol. 17, no. 5, pp. 321–339, 2024. DOI: 10.4236/jsea.2024.175018.

[17] G. Menezes, B. Cafeo, and A. Hora, "How are framework code samples maintained and used by developers? The case of Android and Spring Boot," *Journal of Systems and Software*, vol. 185, p. 111 146, 2022, ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111146.

[18] B. Biegajło and D. Czerwiński, "Investigating the impact of microservice-oriented platform configurations on application performance," *Journal of Computer Sciences Institute*, vol. 31, pp. 124–131, Jun. 2024. DOI: 10.35784/jcsi.6090.

[19] M. Wasim, P. Liang, M. Shahin, H. de Salle, and G. Márquez, "Diseño, monitoreo y prueba de sistemas de microservicios: La perspectiva de los profesionales," *Journal of Systems and Software*, vol. 182, p. 111 061, 2021, ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111061.

[20] M. Ortu, G. Destefanis, D. Graziotin, M. Marchesi, and R. Tonelli, "How do you propose your code changes? Empirical analysis of affect metrics of pull requests on GitHub," *IEEE Access*, vol. 8, pp. 110 897–110 907, 2020. DOI: 10.1109/ACCESS.2020.3002663.

[21] C. Zhang, W. Shao, X. Wang, *et al.*, *Explainable autonomic cybersecurity system for smart power grid*, Taipei, Taiwan, Sep. 2024. DOI: 10.1109/CNS62487.2024.10735649.

[22] S. Bernardo, P. Orviz, M. David, *et al.*, "Aseguramiento de calidad de software como servicio: Abarcando la evaluación de calidad de software y servicios," *Future Generation Computer Systems*, vol. 156, pp. 254–268, 2024. DOI: 10.1016/j.future.2024.03.024. [Online]. Available: https://doi.org/10.1016/j.future.2024.03.024.

[23] M. Gao, M. Chen, A. Liu, W. H. Ip, and K. L. Yung, "Optimization of microservice composition based on artificial immune algorithm considering fuzziness and user preference," *IEEE Access*, vol. 8, pp. 26 385–26 404, 2020. DOI: 10.1109/ACCESS.2020.2971379.

[24] J. Hao, P. Chen, J. Chen, and X. Li, "Multi-task federated learning-based system anomaly detection and multi-classification for microservices architecture," *Future Generation Computer Systems*, vol. 159, pp. 77–90, 2024, ISSN: 0167-739X. DOI: 10.1016/j.future.2024.05.006.

[25] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, Jan. 2015. DOI: 10.1109/MS.2015.11.

[26] J. Esparza-Peidro, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán, "Modeling microservice architectures," *Journal of Systems and Software*, vol. 213, p. 112 041, 2024, ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.112041.

[27] A. Khatami *et al.*, "Estado del arte en aseguramiento de calidad en el desarrollo de software de código abierto basado en java," *Software: Practice and Experience*, vol. 54, no. 8, pp. 1408–1446, 2024, ISSN: 0038-0644. DOI: 10.1002/spe.3321. [Online]. Available: https://doi.org/10.1002/spe.3321.

[28] G. Giray, *A software engineering perspective on engineering machine learning systems: State of the art and challenges*, 2021. DOI: 10.1016/j.jss.2021.111031.

[29] H.-M. Chen, W.-H. Chen, and C.-C. Lee, "An automated assessment system for analysis of coding convention violations in Java programming assignments," *Journal of Information Science and Engineering*, vol. 34, no. 5, pp. 1203–1221, 2018, ISSN: 1016-2364. DOI: 10.6688/JISE.201809_34(5).0006.

[30] J. L. González-Compeanb, V. Sosa-Sosaaa, A. Díaz-Pérez, J. Carreteroc, and J. Yanez-Sierraa, "Sacbe: Un enfoque basado en bloques de construcción para construir un almacenamiento en la nube de extremo a extremo eficiente y flexible," *Journal of Systems and Software*, vol. 135, pp. 143–156, 2018. DOI: 10.1016/j.jss.2017.10.004.

[31] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "Un estudio de mapeo sistemático del desarrollo de software con GitHub," *IEEE Access*, vol. 5, pp. 7173–7192, 2017. DOI: 10.1109/ACCESS.2017.2682323.

[32] A. Y. Kim, V. Herrmann, R. Barreto, *et al.*, "Implementación de la integración continua de github actions para reducir los índices de error en la recopilación de datos ecológicos," *Methods in Ecology and Evolution*, vol. 13, no. 11, pp. 2572–2585, 2022. DOI: 10.1111/2041-210X.13982.

[33] Y. Wang *et al.*, “La madurez de la automatización de pruebas mejora la calidad del producto: Estudio cuantitativo de proyectos de código abierto utilizando integración continua,” *Journal of Systems and Software*, vol. 188, p. 111 259, 2022, ISSN: 0164-1212. DOI: 10.1016/j.jss.2022.111259. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111259.

[34] F. El Aouni, K. Moumane, A. Idri, M. Najib, and S. U. Jan, “Una revisión sistemática de literatura sobre la integración de agile, cloud y devops: Desafíos y beneficios,” *Information and Software Technology*, vol. 177, p. 107 569, 2024. DOI: 10.1016/j.infsof.2024.107569.

[35] O. Bentaleb, A. Sebaa, S. Mira, and A. S. Z. Belloum, “Despliegue de un framework de programación basado en microservicios y contenedores con aplicación al dominio astrofísico,” *Astronomy and Computing*, 2022. DOI: 10.1016/j.ascom.2022.100655.

[36] E. Klotins *et al.*, “Hacia la evaluación costo-beneficio para actividades de ingeniería de software continua,” *Empirical Software Engineering*, vol. 27, no. 6, 2022, ISSN: 1382-3256. DOI: 10.1007/s10664-022-10191-w. [Online]. Available: https://doi.org/10.1007/s10664-022-10191-w.

[37] A. G. d. M. Rossetto, D. Noetzold, L. A. Silva, and V. R. Q. Leithardt, “Enhancing monitoring performance: A microservices approach to monitoring with spyware techniques and prediction models,” *Sensors*, vol. 24, no. 13, p. 4212, 2024. DOI: 10.3390/s24134212.

[38] S. K. G. Sarcar, W. Ahmed, and E. Onagh, “Mercado de github para la automatización y la innovación en la producción de software,” *Information and Software Technology*, vol. 175, p. 107 522, 2024. DOI: 10.1016/j.infsof.2024.107522.

[39] R. Nasr, M. I. Marie, and A. El Sayed, “Un marco híbrido para implementar prácticas DevOps en aplicaciones blockchain (DevChainOps),” *International Journal of Advanced Computer Science and Applications*, vol. 15, no. 6, 2024. DOI: 10.14569/IJACSA.2024.0150647. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2024.0150647.

[40] L. Giamattei, A. Guerriero, R. Pietrantuono, *et al.*, “Monitoring tools for DevOps and microservices: A systematic grey literature review,” *Journal of Systems and Software*, vol. 208, p. 111 906, 2024, ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111906.

[41] S. Schneider and R. Scandariato, “Extracción automática de diagramas de flujo de datos con alto nivel de seguridad para aplicaciones de microservicios escritas en java,” *Journal of Systems and Software*, 2023. DOI: 10.1016/j.jss.2023.111722.

ANEXOS

Este es un enfoque estructurado que ayuda a garantizar que la aplicación sea robusta, escalable y de alta calidad, aprovechando lo mejor de las herramientas y servicios que ofrece la nube.

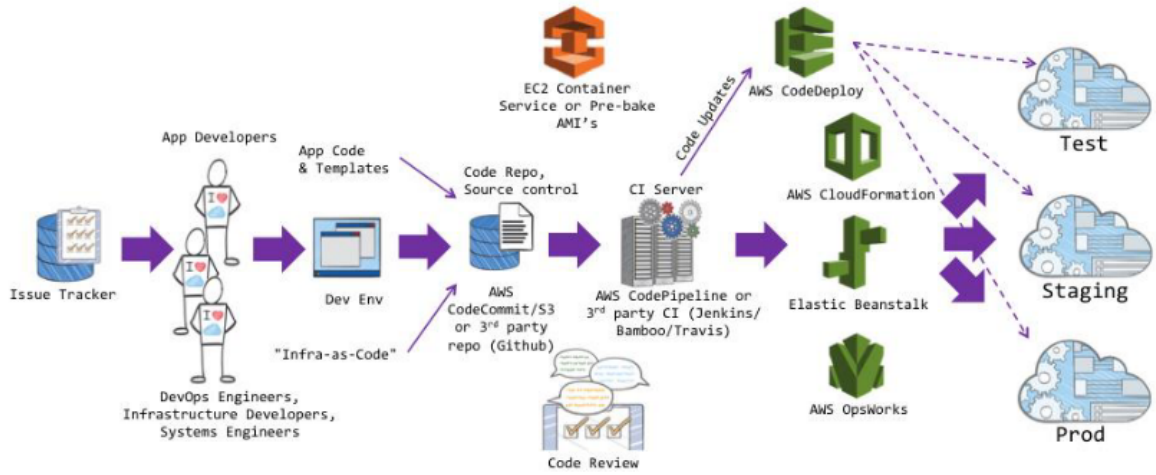

Figure 29.
Descripción general de los servicios de implementación [12].

Este pipeline está diseñado para facilitar el flujo de trabajo en el desarrollo de software, asegurando que cada etapa sea ejecutada de manera secuencial y controlada, contribuyendo así a un desarrollo más ágil y fiable.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building the application...'
                sh 'mvn clean package' // Comando para constru
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests...'
                sh 'mvn test' // Comando para ejecutar pruebas
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying the application...'
                sh 'scp target/myapp.jar user@server:/path/to/
            }
        }
    }
}
```

Figure 30.
Ejemplo De Código: Jenkins file[7].

En el marco de mi trabajo de tesis, para implementar y gestionar el proceso de integración continua y despliegue continuo (CI/CD), configuré y utilicé activamente la herramienta Jenkins. Mi punto de partida para interactuar diariamente con este sistema era su consola de autenticación, a la que accedía a través de su interfaz web.

Al iniciar una sesión de trabajo, mi primer paso siempre era introducir mis credenciales en esta pantalla de inicio de sesión. Debía completar los campos ”Username” (Nombre de usuario) y ”Contraseña” con los datos de acceso que configuré para el servidor. Dependiendo de si me encontraba en mi estación de trabajo personal y de confianza, decidía marcar o no la casilla ”Keep me signed in” (Mantenerme conectado) para agilizar el acceso en las sucesivas ocasiones. Finalmente, hacía clic en el botón azul ”Sign in” (Iniciar sesión) para acceder al dashboard

principal.

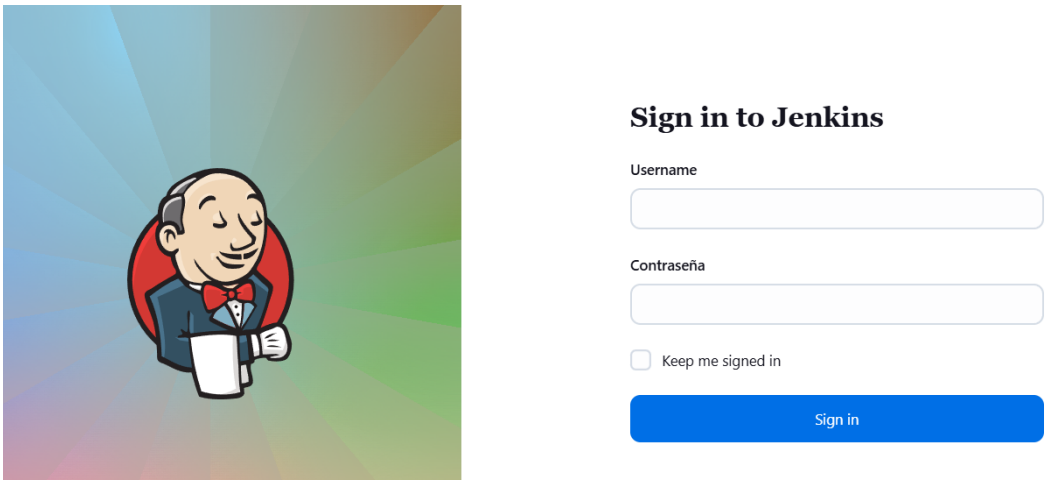

Figure 31.
Sprint 2:Selección y justificación de herramientas Jenkins
Nota. Fuente: Elaboración propia.

En el flujo de trabajo de mi investigación, GitHub fue la plataforma fundamental para el control de versiones y la colaboración en el código fuente de mi proyecto. Mi interacción comenzaba constantemente en su página de inicio de sesión, a la que accedía para gestionar los repositorios.

Diariamente, al necesitar subir cambios, revisar commits o gestionar las incidencias, mi primer paso era autenticarme. En esta pantalla, introducía mi ”Nombre de usuario o dirección de correo electrónico” en el primer campo y mi ”Contraseña” en el segundo. Justo debajo, el enlace de ”¿Has olvidado tu contraseña?” era un recordatorio de las medidas de seguridad, aunque nunca necesité usarlo. Para finalizar el proceso, hacía clic en el botón verde ”Iniciar sesión”.

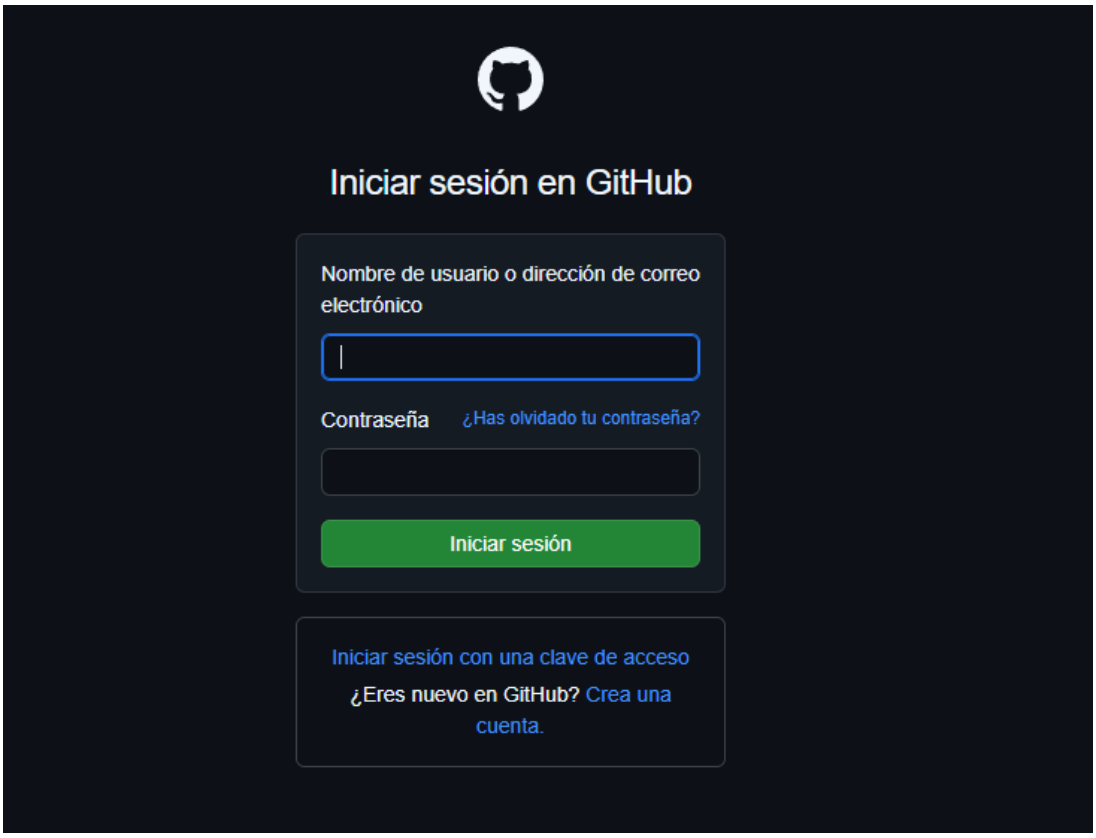

Figure 32.
Sprint 2:Selección y justificación de herramientas GitHub
Nota. Fuente: Elaboración propia.

Durante la fase de configuración de la automatización en mi tesis, una de las acciones más críticas que realicé en Jenkins fue la creación de una nueva tarea o ’job’, que es la unidad fundamental de trabajo. Esta captura representa precisamente el diálogo inicial de ese proceso, que aparece al hacer clic en ”Nuevo Tarea” en el panel principal.

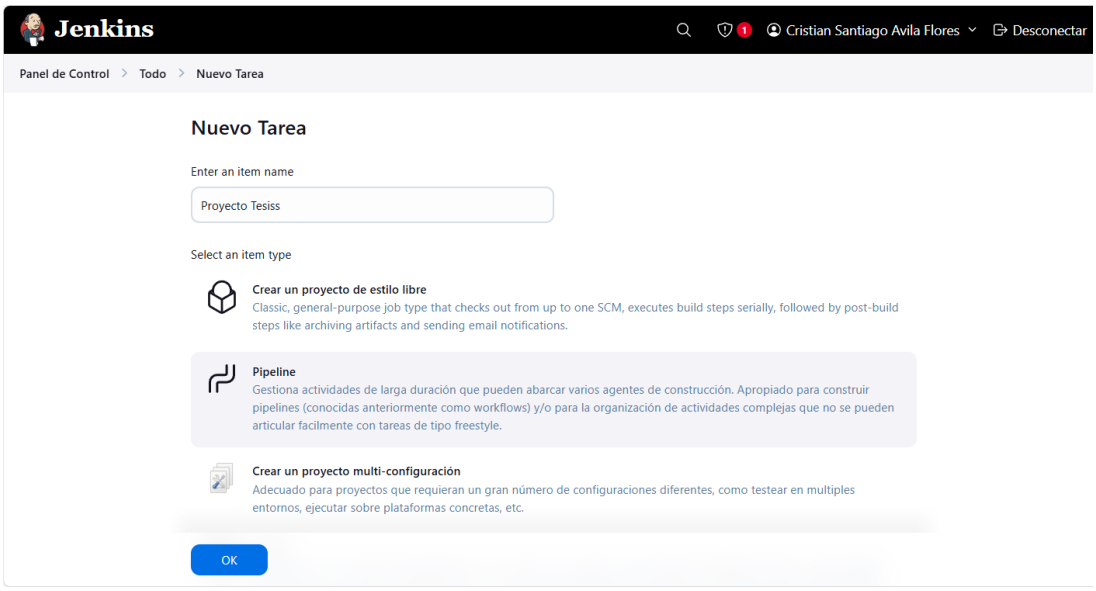

Figure 33.
Sprint 3:Configuración de Jenkins
Nota. Fuente: Elaboración propia.

En el desarrollo de mi trabajo de tesis, GitHub se consolidó como el eje central para el almacenamiento, versionado y gestión de todo el código fuente, scripts y documentación del proyecto. Esta captura muestra el panel principal de mi perfil de GitHub, que era el punto de partida desde el cual organizaba y accedía a todos los recursos.

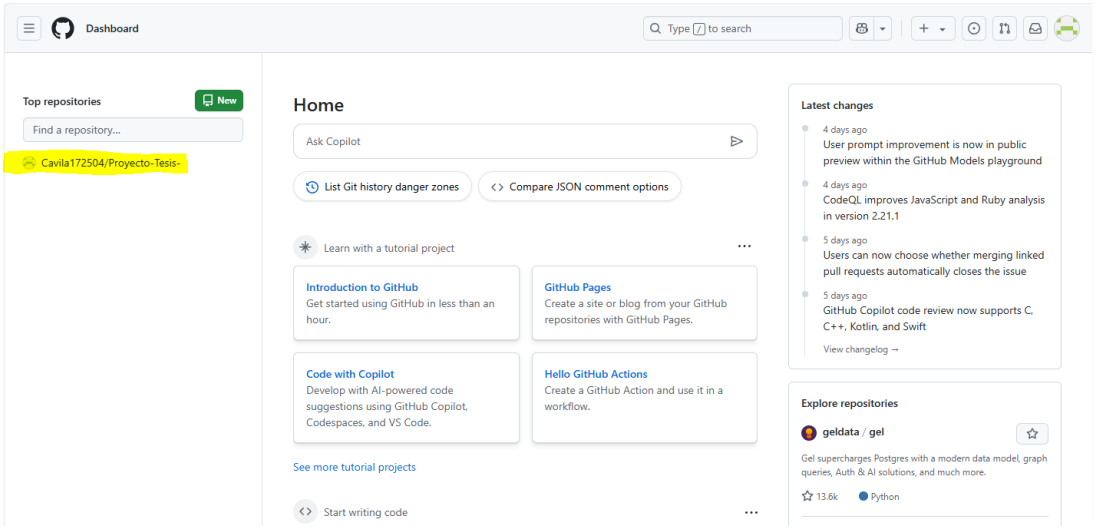

Figure 34.
Sprint 3:Repositorios en GitHub
Nota. Fuente: Elaboración propia.

En la fase crítica de automatización, configurar la conexión entre GitHub y Jenkins fue un

paso fundamental para que el pipeline de integración continua funcionara de manera autónoma. Esta pantalla captura el momento exacto en el que definí esa integración, dentro de la configuración de la tarea ”Proyecto Tesis” en Jenkins.

Mi objetivo principal aquí es asegurarme de que Jenkins reaccionara automáticamente a cada nuevo commit que yo hiciera en el repositorio de la tesis. Para ello, en la pestaña ”Pipeline”, seleccioné la opción ”Pipeline script from SCM”. Esta elección fue crucial, ya que me permitió almacenar el Jenkinsfile (el corazón de la automatización) directamente en mi repositorio de GitHub, manteniendo así la configuración de la pipeline versionada y sincronizada con el código.

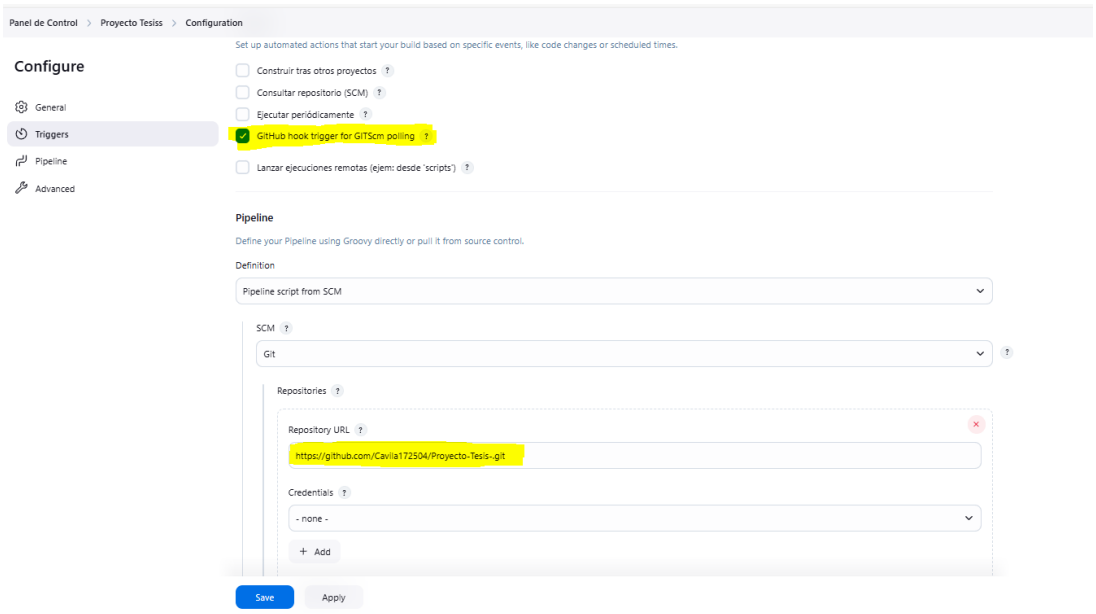

Figure 35.
Sprint 3:Integrar GitHub con Jenkins
Nota. Fuente: Elaboración propia.

Uno de los aspectos más determinantes para el éxito de la automatización fue la correcta instalación y gestión de plugins en Jenkins, que extendieron las funcionalidades nativas del servidor para adaptarse a las necesidades específicas del flujo de trabajo.

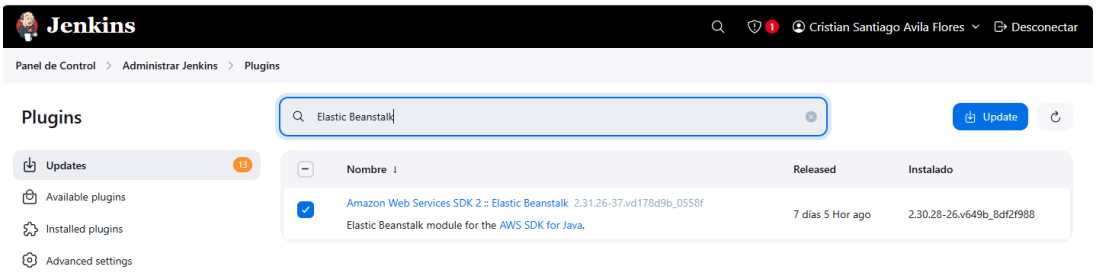

Figure 36.
Sprint 3:Instalación de plugins
Nota. Fuente: Elaboración propia.

Al navegar al sitio de AWS, me encontraba con esta interfaz clara que me presentaba dos opciones fundamentales para iniciar sesión: como ”Usuario raíz” o como ”Usuario de IAM”. Como buena práctica de seguridad, evitaba utilizar las credenciales de usuario raíz para las tareas cotidianas debido a su acceso ilimitado. En su lugar, siempre accedía mediante un usuario de IAM que yo mismo había creado y configurado con permisos específicos y restringidos, alineados perfectamente con el principio de mínimo privilegio que es crucial en entornos de producción académicos y profesionales.

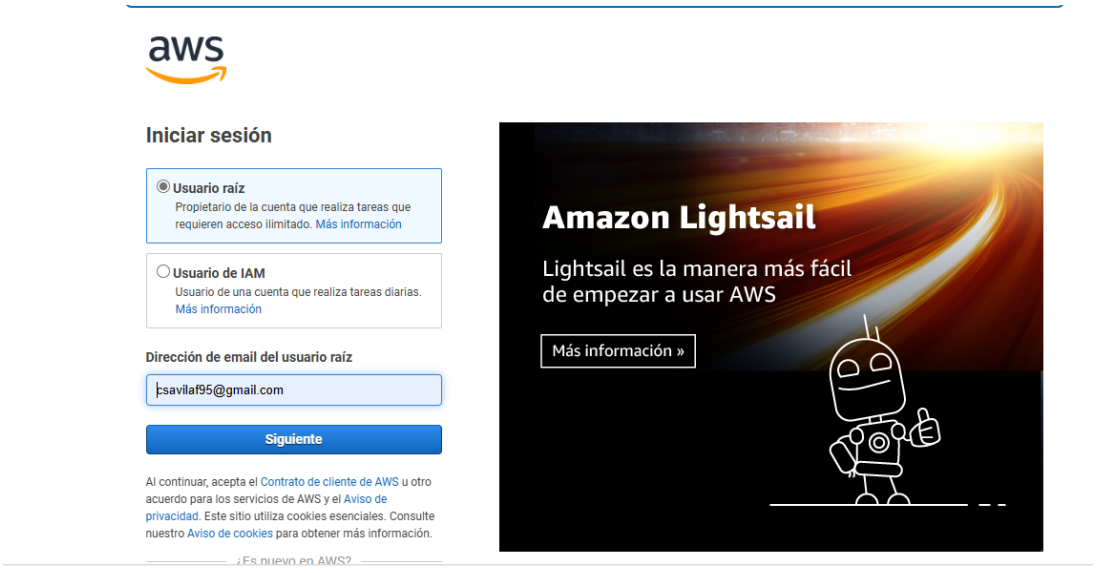

Figure 37.
Sprint 4:Inicio secion AWS
Nota. Fuente: Elaboración propia.

Al desplegar la arquitectura en la nube para mi proyecto, la creación y configuración de

la aplicación en AWS Elastic Beanstalk fue un paso decisivo. Esta captura muestra justo el momento en que definí los parámetros iniciales que darían vida al entorno de producción

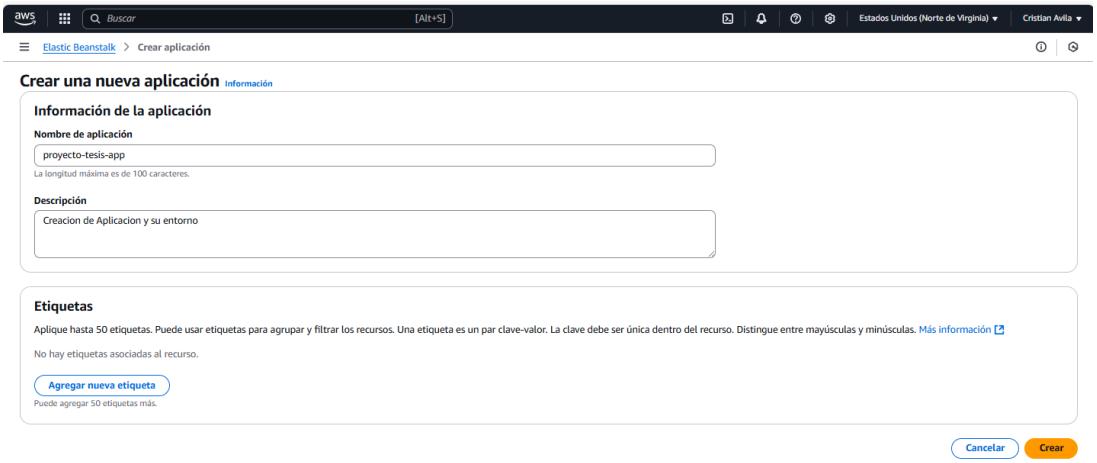

Figure 38.
Sprint 4:Configuración de AWS Elastic Beanstalk.
Nota. Fuente: Elaboración propia.

La definición y monitoreo de los entornos de despliegue fueron cruciales para garantizar que el flujo de integración y entrega continua fuera confiable y transparente. Esta captura refleja el panel de control principal de mi entorno en AWS Elastic Beanstalk, que denominé proyecto-tesis-env, asociado a la aplicación proyecto-tesis-app.

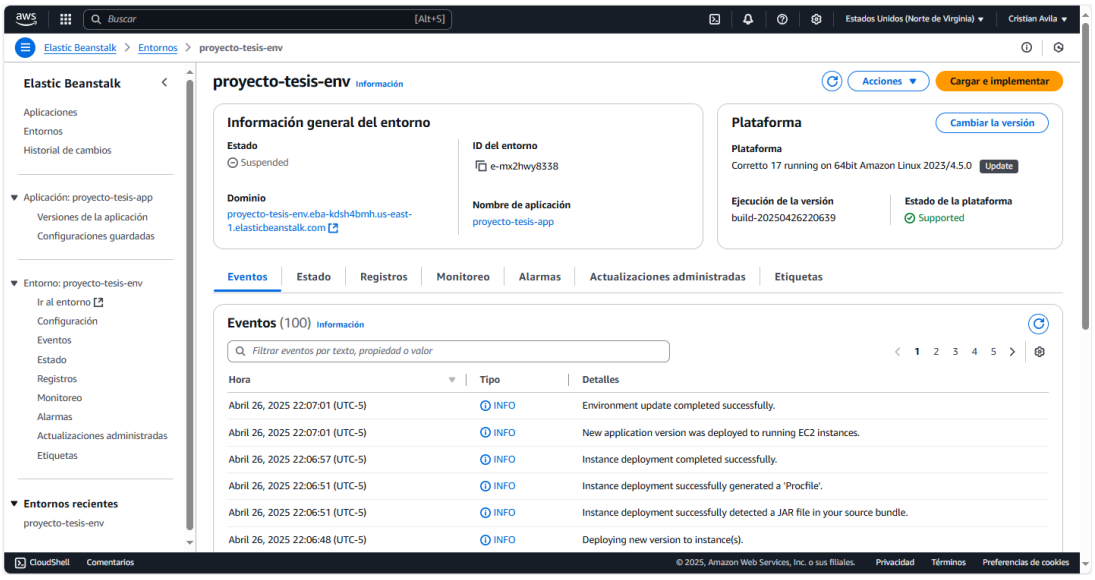

Figure 39.
Sprint 4:Definición de entornos de pruebas
Nota. Fuente: Elaboración propia.

La gestión segura de los permisos y las identidades fue un pilar fundamental para garantizar que la automatización entre Jenkins y AWS operara sin riesgos. Esta imagen captura la configuración detallada de un rol de IAM, específicamente aws-elasticbeanstalk-ec2-role, que creé y asigné para que los servicios de AWS interactuaran de manera segura y con los privilegios mínimos necesarios.

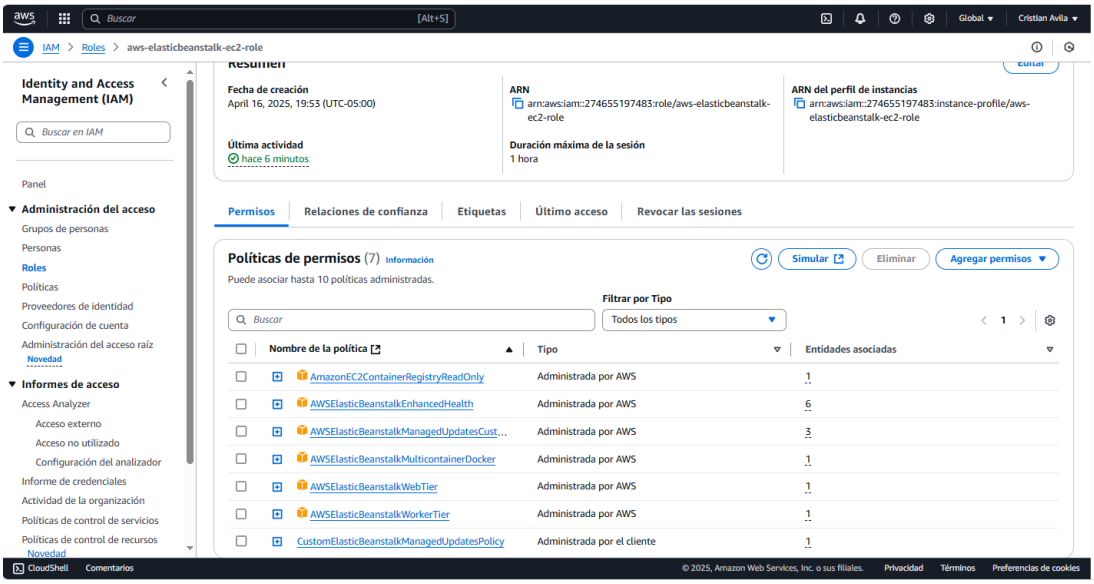

Figure 40.
Sprint 4:Configuración de claves y permisos en AWS
Nota. Fuente: Elaboración propia.

La configuración del almacenamiento en la nube fue una etapa crítica para asegurar la persistencia y disponibilidad de los artefactos generados por el pipeline. Esta captura muestra el bucket de Amazon S3 que configuré, pt-s3-buckettt, el cual funcionó como el repositorio central para almacenar de forma segura cada versión compilada de la aplicación.

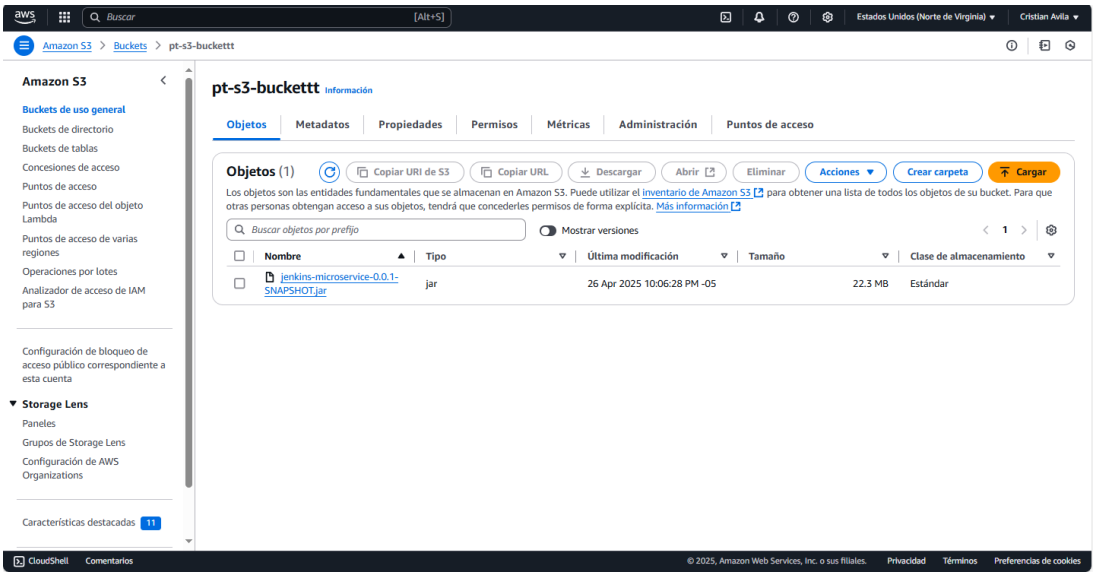

Figure 41.
Sprint 4:Configuración de AWS S3.
Nota. Fuente: Elaboración propia.

Esta imagen representa el corazón de la automatización que diseñé: el pipeline de Jenkins visualizado como un flujo unificado que orquesta cada fase crítica del ciclo de vida de mi aplicación. Cada uno de estos bloques encapsula una etapa fundamental que ejecuté de forma automatizada tras cada commit en el repositorio.

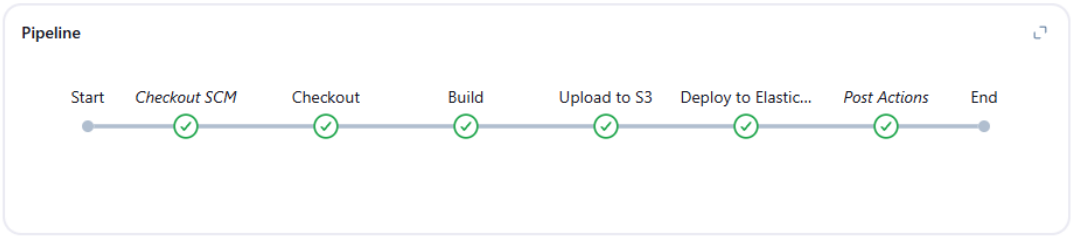

Figure 42.
Sprint 5:Definición de etapas build, test, deploy
Nota. Fuente: Elaboración propia.

Esta captura muestra la ejecución detallada de una de las corridas más críticas del pipeline: la integración de pruebas automáticas dentro de la etapa de construcción. En el build #71 de la tarea prueba-5, puedo ver el desglose preciso de cada paso que Jenkins ejecutó, lo que me permitió validar y depurar el proceso de manera efectiva.

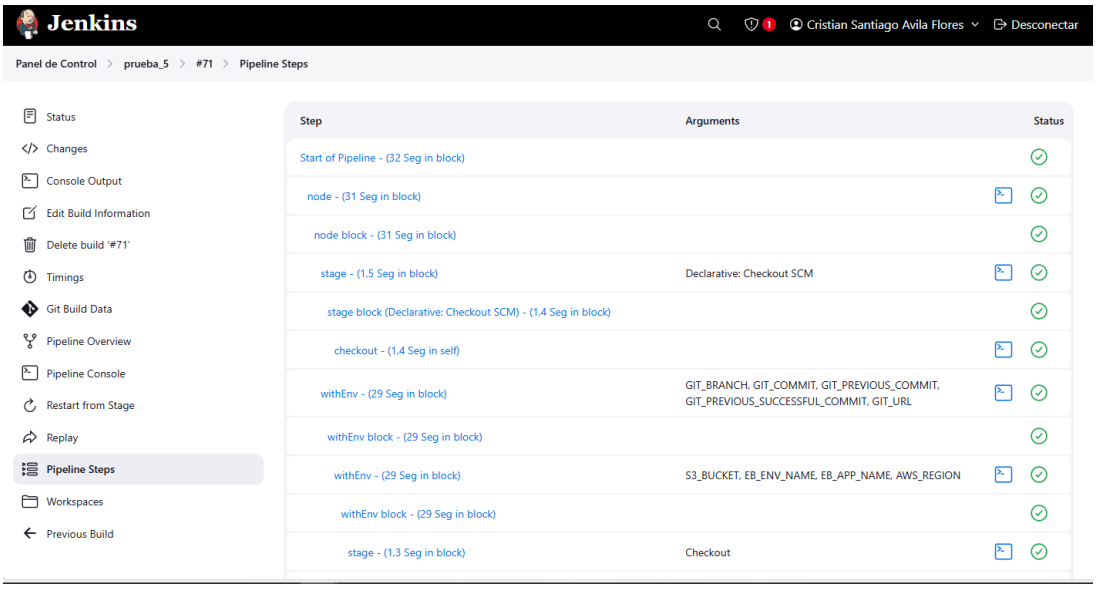

Figure 43.
Sprint 6:Integración de pruebas automáticas
Nota. Fuente: Elaboración propia.

El documenta el momento exacto en que el pipeline de Jenkins inició la automatización de despliegue, ejecutándose bajo mi usuario (Cristian Santiago Avila Flores). Aquí se evidencia el proceso crítico donde Jenkins recupera la definición del pipeline directamente desde el repositorio de GitHub, específicamente el archivo Jenkinsfile ubicado en la ruta jenkins-microservice/, que contenía toda la instrucción codificada del flujo de trabajo.

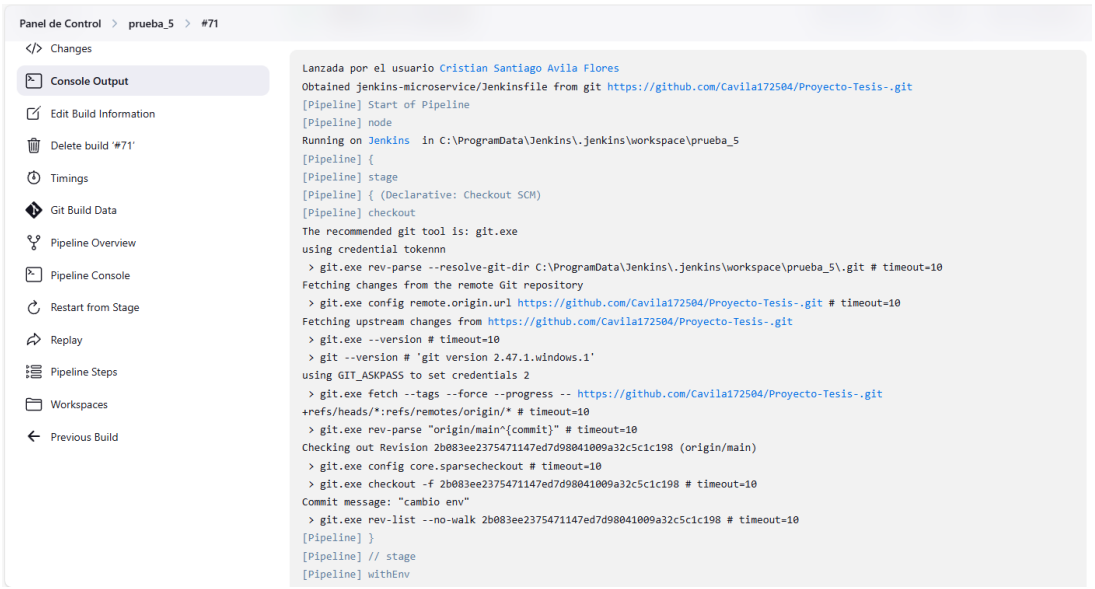

Figure 44.
Sprint 6:Automatización de despliegue en AWS
Nota. Fuente: Elaboración propia.

La a culminación exitosa de todo el flujo de integración y despliegue continuo que diseñé e implementé. El Build #75 de la tarea prueba-5 no es solo otra ejecución más; es la validación definitiva de que el pipeline completo de CI/CD funciona de manera autónoma, eficiente y confiable.

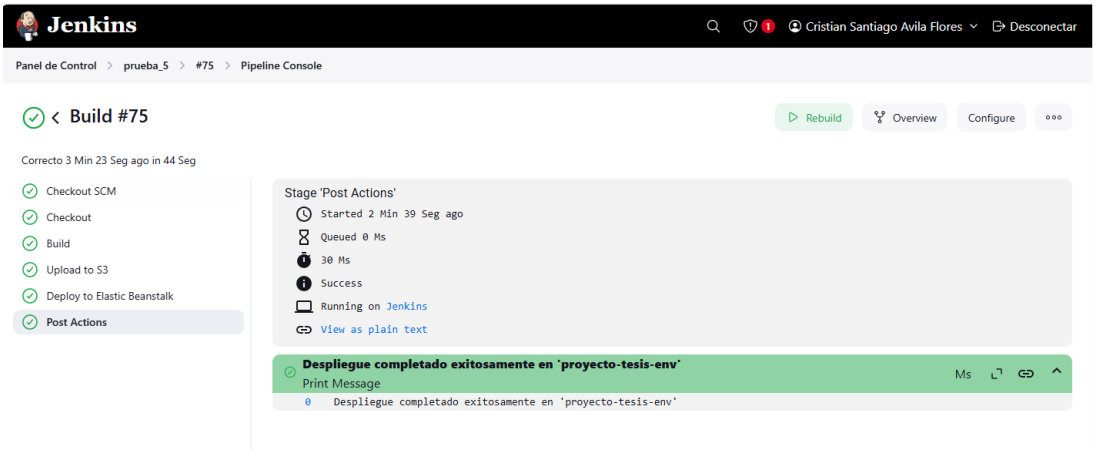

Figure 45.
Sprint 6:Pipeline de CI-CD completo y en funcionamiento
Nota. Fuente: Elaboración propia.

La ejecución controlada y trazable de un despliegue automatizado iniciado manualmente. Al observar la salida de la consola, confirmo que el pipeline se activó bajo mi usuario (Cristian Santiago Avila Flores), lo que me permitió tener un control granular sobre el proceso y auditar cada acción.

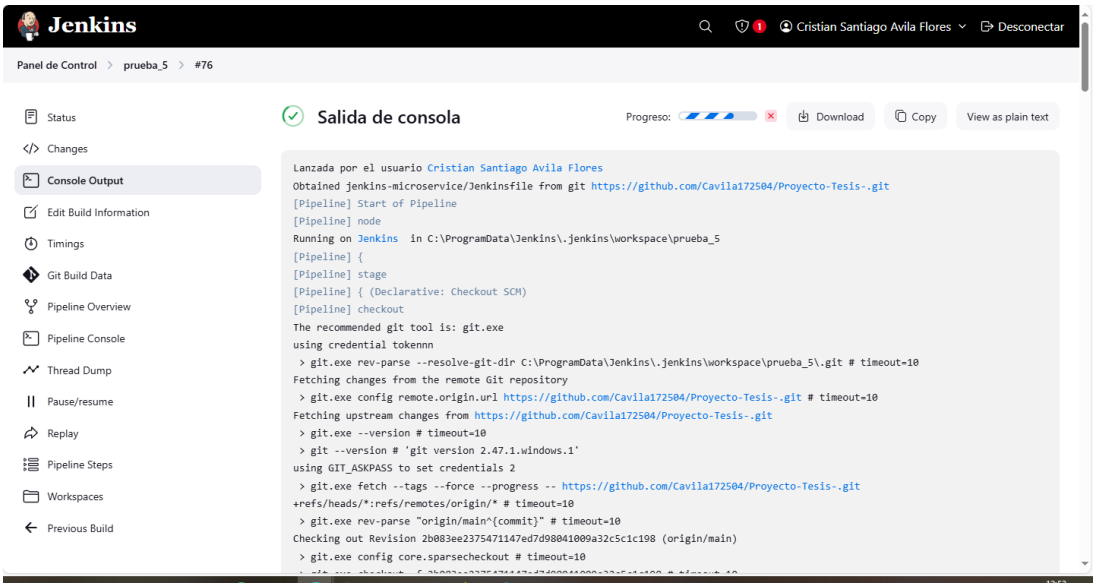

Figure 46.
Sprint 7: Ejecución de despliegues controlados.
Nota. Fuente: Elaboración propia.

EL pipeline en plena ejecución, mostrando el progreso en tiempo real de cada una de las etapas automatizadas que configuré. El diagrama de flujo, con sus barras de porcentaje avanzando, representa la materialización del concepto de CI/CD que fue central en mi investigación.

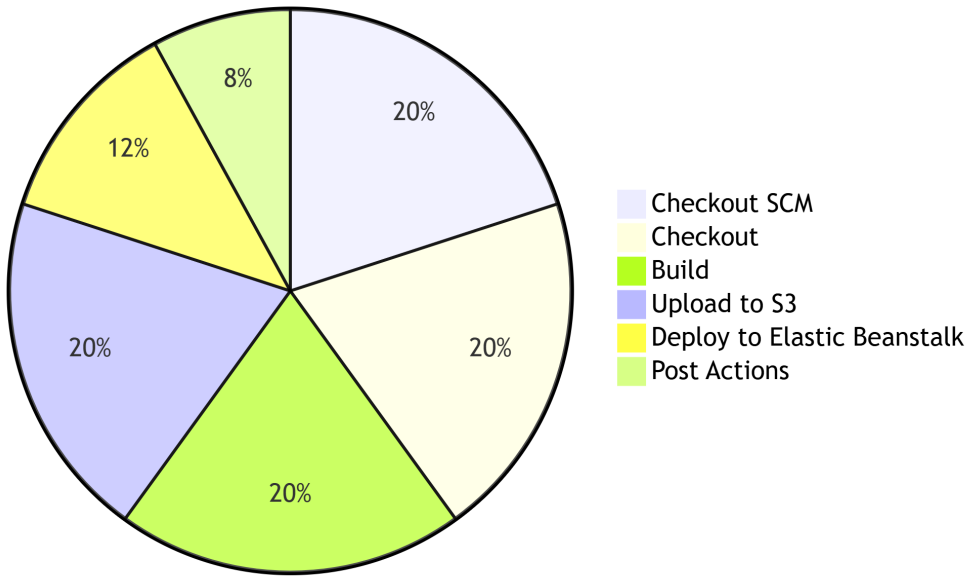

Figure 47.
Despliegue Build.
Nota. Fuente: Elaboración propia.