AUTOMATIZACIÓN DEL FLUJO CI/CD INTEGRANDO HERRAMIENTAS DE ANÁLISIS DE SEGURIDAD MEDIANTE PIPELINES DEVSECOPS PARA LA PLATAFORMA SMART CAMPUS UIS.

NELSON ANDRÉS BARBOZA LANDINEZ
ISIDRO HERRERA RINCÓN

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS,
ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
BUCARAMANGA
2026

AUTOMATIZACIÓN DEL FLUJO CI/CD INTEGRANDO HERRAMIENTAS DE ANÁLISIS DE SEGURIDAD MEDIANTE PIPELINES DEVSECOPS PARA LA PLATAFORMA SMART CAMPUS UIS.

NELSON ANDRÉS BARBOZA LANDINEZ
ISIDRO HERRERA RINCÓN

Trabajo de grado para optar al título de Ingeniero de Sistemas

Director
Gabriel Rodrigo Pedraza Ferreira
Doctor en Ciencias de la Computación

Codirector
Jathinson Meneses Mendoza
Magíster en Gestión, Aplicación y Desarrollo de Software

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS,
ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
BUCARAMANGA
2026

Agradecimientos

Cada página de este trabajo es un reflejo y resultado de un conjunto de circunstancias, procesos y cambios. Es el fruto de un camino recorrido, un hito en un propósito y llamado que va más allá de lo que se puede contemplar en ese plano terrenal.

En primer lugar, quiero agradecer a Dios: ese Dios amoroso que envió a su hijo a morir en una cruz por nuestras iniquidades; ese Dios todopoderoso que se preocupa por esta humanidad que le da la espalda; ese Dios que permite que el sol salga para buenos y malos; ese padre y amigo espiritual el cual me acompañó en cada día. A pesar de mis errores, a pesar de no merecerlo, me permitió vivir este privilegio que pocas personas tienen, y quien uso este medio para que mi fe sea fortalecida, porque es necesario que el oro pase por el fuego para ser probado. Él fue el alfarero que ha hecho de mi lo que soy. Me encuentro infinitamente agradecido, porque incluso en esos momentos de silencio, sé que estuvo presente; sé que fue Él quien me protegió, me acompaño, me proveyó de lo necesario y me llevó por cada una de las etapas de este camino. Toda la gloria y honra sea para Él, porque se fortaleció en mi debilidad y fue quien me puso en este lugar el día de hoy. Sublime gracia inmerecida que me transformó con amor y misericordia.

También quiero agradecer a mi familia: a mi padre, a mi madre y a mi hermano, quienes fueron parte importante y trascendental de este proceso. Me apoyaron, me motivaron y no escatimaron en paciencia; Me soportaron en momentos de dificultad con sus palabras de amor. Este logro no es solo mío, es el resultado de todo lo que han sembrado en mí a lo largo de los años. Muchas gracias por cada sacrificio y por cada esfuerzo invertido en mi formación. Espero que este logro les honre.

Mi gratitud se extiende a la Universidad Industrial de Santander, bastión de la excelencia académica, la cual me brindó las herramientas necesarias para desarrollar las habilidades requeridas para el desarrollo de este trabajo. De igual manera a nuestro director y codirector, quienes con su conocimiento, dedicación y orientación oportuna nos guiaron a lo largo de este proceso. Su disposición para resolver dudas, sus observaciones certeras y su compromiso con nuestra formación fueron fundamentales para alcanzar esta meta.

Nelson Andrés Barboza Landinez

Agradecimientos

En primer lugar, agradezco a Dios por todo el amor que me ha brindado, y por ser quien me ha dado la fortaleza, sabiduría, constancia, gracia, dirección y guía a lo largo de este camino, quien me ha sostenido, y por el que he llegado hasta acá, permitiéndome más de lo que alguna vez imaginé.

Agradezco a mi familia, en especial a mis papás, Isidro y Myriam, y a mi hermana, Marialejandra, quienes han estado presentes, brindándome su amor, apoyo, tiempo, esfuerzo, dedicación y sacrificio en todo el trayecto de la universidad y de la vida. Son quienes me han motivado y han estado conmigo en todos los momentos, y por quienes están formadas las principales bases de lo que soy hoy. Por lo tanto, no es un logro individual, sino un logro en conjunto, el resultado de un trabajo en equipo hasta el día de hoy.

Quiero agradecer también a mi novia, Rosita, quien estuvo presente al final de esta etapa, y a lo largo de este proyecto, acompañándome, dándome también su amor y apoyo, y quien desde que se volvió una parte importante de mi vida ha sido de edificación y ayuda en todos los aspectos.

Expreso también mi agradecimiento a nuestro director, Gabriel Rodrigo Pedraza Ferreira, y a nuestro codirector, Jathinson Meneses Mendoza, por su guía, tiempo, disposición y valiosos aportes durante el desarrollo de este trabajo. Su acompañamiento y observaciones fueron fundamentales para orientar y fortalecer este proyecto.

Asimismo, agradezco a las demás personas que me acompañaron durante este proceso, especialmente a mi compañero y amigo Pablo, por su apoyo, disposición y acompañamiento en distintos momentos del desarrollo de este trabajo.

Y finalmente, agradezco a la Universidad Industrial de Santander y a la Escuela de Ingeniería de Sistemas e Informática, por la formación académica recibida y por brindarme las bases necesarias para alcanzar esta meta.

Isidro Herrera Rincón

TABLA DE CONTENIDO

Pág.

LISTA DE CUADROS

Pág.

LISTA DE FIGURAS

Pág.

LISTA DE ANEXOS

Pág.

RESUMEN

TÍTULO: AUTOMATIZACIÓN DEL FLUJO CI/CD INTEGRANDO HERRAMIENTAS DE ANÁLISIS DE SEGURIDAD MEDIANTE PIPELINES DEVSECOPS PARA LA PLATAFORMA SMART CAMPUS UIS. *

AUTORES: NELSON ANDRÉS BARBOZA LANDINEZ
ISIDRO HERRERA RINCÓN **

PALABRAS CLAVE: DEVSECOPS, CI/CD, SEGURIDAD TEMPRANA, AUTOMATIZACIÓN, SMART CAMPUS UIS.

DESCRIPCIÓN:

Este trabajo de grado en modalidad de investigación presenta el diseño e implementación de un flujo CI/CD con enfoque DevSecOps para la plataforma Smart Campus UIS, con el propósito de fortalecer la automatización, la trazabilidad y la incorporación temprana de controles de seguridad dentro del ciclo de vida del software. La propuesta surge a partir del análisis del estado actual de los procesos de desarrollo, construcción y despliegue de la plataforma, en los cuales se identificaron oportunidades de mejora relacionadas con validaciones manuales, y ausencia de controles sistemáticos de seguridad.

La solución se estructuró sobre un pipeline organizado en tres bloques funcionales: validación del código fuente y pruebas unitarias, control de calidad y seguridad temprana, y contenerización con despliegue controlado. Para su implementación se integraron Jenkins como orquestador principal, SonarQube Community Build para análisis estático de código (SAST), OWASP Dependency-Check para análisis de composición de software (SCA) y Trivy para el escaneo de imágenes de contenedor.

La validación del flujo se realizó principalmente sobre los microservicios admin_microservice y data_microservice, incluyendo compilación, pruebas unitarias, generación de artefactos, análisis de dependencias, construcción de imágenes, escaneo de seguridad y smoke tests posteriores al despliegue. Los resultados evidenciaron la viabilidad de automatizar etapas clave del proceso CI/CD e integrar controles de seguridad temprana, generando además evidencia técnica útil para la trazabilidad y priorización de hallazgos. En conclusión, el trabajo establece una base funcional y documentada para la adopción progresiva de prácticas DevSecOps en Smart Campus UIS.

---

* Trabajo de Grado
** Facultad de Ingenierías Fisicomecánicas. Escuela de Ingeniería de Sistemas e Informática. Director Gabriel Rodrigo Pedraza Ferreira, Doctor en Ciencias de la Computación. Codirector Jathinson Meneses Mendoza, Magíster en Gestión, Aplicación y Desarrollo de Software.

ABSTRACT

TITLE: AUTOMATION OF THE CI/CD FLOW BY INTEGRATING SECURITY ANALYSIS TOOLS THROUGH DEVSECOPS PIPELINES FOR THE SMART CAMPUS UIS PLATFORM. *

AUTHORS: NELSON ANDRÉS BARBOZA LANDINEZ
ISIDRO HERRERA RINCÓN **

KEY WORDS: DEVSECOPS, CI/CD, EARLY SECURITY, AUTOMATION, SMART CAMPUS UIS.

DESCRIPTION:

This bachelor thesis, developed under the research modality, presents the design and implementation of a CI/CD workflow with a DevSecOps approach for the Smart Campus UIS platform, aiming to strengthen automation, traceability, and the early incorporation of security controls throughout the software lifecycle. The proposal emerged from an analysis of the current development, build, and deployment processes of the platform, where opportunities for improvement were identified in relation to manual validations, and the lack of systematic security controls.

The solution was structured around a pipeline organized into three functional blocks: source code validation and unit testing, quality and early security control, and containerization with controlled deployment. Its implementation integrated Jenkins as the main orchestrator, SonarQube Community Build for Static Application Security Testing (SAST), OWASP Dependency-Check for Software Composition Analysis (SCA), and Trivy for container image scanning.

The workflow was mainly validated on the admin_microservice and data_microservice, including compilation, unit testing, artifact generation, dependency analysis, image building, security scanning, and post-deployment smoke tests. The results demonstrated the feasibility of automating key CI/CD stages and integrating early security controls, while also generating technical evidence useful for traceability and for prioritizing findings. In conclusion, this work establishes a functional and documented foundation for the progressive adoption of DevSecOps practices in Smart Campus UIS.

---

* Bachelor Thesis

** Faculty of Physicomechanical Engineering. School of Systems Engineering and Informatics. Director Gabriel Rodrigo Pedraza Ferreira, Doctor in Computer Science. Codirector Jathinson Meneses Mendoza, Master's Degree in Management, Application and Development of Software.

INTRODUCCIÓN

El desarrollo de software de productos y servicios actualmente requiere un énfasis en la seguridad desde etapas tempranas del ciclo de vida, lo que resulta esencial para la mitigación de riesgos y errores potenciales. [1] En este contexto, Smart Campus UIS es una plataforma donde convergen tecnologías de la información, dispositivos IoT y servicios administrativos para ofrecer soluciones integradas en entornos educativos. [2] Sin embargo, esta interdependencia incrementa la superficie de ataque y eleva los riesgos asociados a brechas de seguridad, configuraciones inseguras, o malas prácticas en el desarrollo de software [3].

El panorama del desarrollo de software se distingue por la creciente exigencia de ciclos de lanzamiento más cortos, lo que ha llevado a las organizaciones de TI a adoptar metodologías ágiles y la cultura DevOps para mejorar la agilidad, la escalabilidad y la velocidad de entrega [4]. Sin embargo, aunque este enfoque ayuda a acelerar los despliegues y a obtener retroalimentación más pronto, también trae consigo varios retos importantes, sobre todo cuando se trata de mantener la seguridad y cumplir con las regulaciones. Si no se implementan los controles adecuados, se abren puertas a vulnerabilidades [5]. Y esto no es un tema menor: En 2024, el costo promedio de una filtración de datos alcanzó los 4.88 millones de dólares, una cifra récord que deja claro lo que está en juego [6].

---

[1] ALENEZI, M. y ALMUAIRFI, S. Security Risks in the Software Development Lifecycle. En: International Journal of Recent Technology and Engineering (IJRTE). 2019, vol. 8, nro. 3, p. 7048-7055. Disponible en: https://www.ijrte.org/portfolio-item/C5374098319

[2] JIMÉNEZ, H.; CÁRCAMO, E. y PEDRAZA, G. Extensible software platform for smart campus based on microservices. En: RISTI - Revista Ibérica de Sistemas e Tecnologias de Informação. 2020, nro. E38, p. 270-282. Disponible en: www.scopus.com

[3] ANDRADE, R., et al. Factors of Risk Analysis for IoT Systems. En: Risks. 2022, vol. 10, nro. 8, art. 162. Disponible en: https://www.mdpi.com/2227-9091/10/8/162

[4] KUPPUSAMY VELLAMADAM PALAVESAM, et al. Building Automated Security Pipeline for Containerized Microservices. En: Journal of Advances in Mathematics and Computer Science. 2025, vol. 40, nro. 2, p. 53-66. DOI: https://doi.org/10.9734/jamcs/2025/v40i21969

[5] PRATES, L. y PEREIRA, R. DevSecOps practices and tools. En: International Journal of Information Security. 2025, vol. 24, art. 11. Disponible en: https://link.springer.com/article/10.1007/s10207-024-00914-z

[6] IBM. The cost of a data breach 2024: Financial industry insights. En: IBM Think Insights [sitio web]. 2024. [Consultado: 6 de junio de 2026]. Disponible en: https://www.ibm.com/think/insights/cost-of-a-data-breach-2024-financial-industry

En este punto donde la integración de la calidad y seguridad se vuelve crítica, surge DevSecOps como una evolución natural de DevOps, que integra prácticas de seguridad desde la concepción misma del software. Esta metodología se apoya en el principio de "Shift Left Security" (Desplazar la seguridad a la izquierda), buscando incorporar controles de forma proactiva y automatizada al inicio del ciclo de vida del desarrollo [7]. Como aporte específico, este trabajo se centra en la implementación de DevSecOps dentro de una arquitectura de microservicios particular, representada en la plataforma Smart Campus UIS, lo que permite establecer una referencia aplicable a entornos similares en el ámbito académico.

Partiendo de lo anterior, se propone mejorar la postura de seguridad en Smart Campus UIS mediante la implementación de pipelines DevSecOps que integren, tanto en los componentes administrativo como en el de datos, herramientas de análisis estático del código fuente, así como mecanismos de escaneo para imágenes Docker. Para ello, se plantea un enfoque basado en integración y despliegue continuos, apoyado en tecnologías de código abierto que permita la fácil adopción y adaptabilidad al contexto institucional. De esta manera se espera contribuir a la construcción de un entorno automatizado y reproducible, que no solo mejore la eficiencia y calidad del desarrollo, si no que reduzca riesgos de seguridad, aportando además una referencia metodológica para futuras iniciativas académicas en el área.

[7] PRATES. Op. cit.

1. PLANTEAMIENTO Y JUSTIFICACIÓN DEL PROBLEMA

En el contexto del desarrollo de software contemporáneo, la integración temprana de la seguridad en los flujos DevOps ha ganado relevancia, consolidando a DevSecOps como una tendencia clave [8]. Diversas experiencias en la industria han mostrado resultados positivos al adoptar este enfoque: por ejemplo, la implementación de prácticas DevSecOps en pipelines CI/CD ha logrado reducir hasta en un 50% las vulnerabilidades críticas en entornos productivos [9], evidenciando beneficios superiores frente a metodologías tradicionales. Esta tendencia global hacia “Security as Code” surge en respuesta a un entorno donde los ciclos de lanzamiento son cada vez más frecuentes y acelerados, impulsados por exigencias de entrega continua y flexibilidad a cambios. Frente a ello, la combinación de la cultura DevOps con la automatización de CI/CD y la incorporación sistemática de controles de seguridad se presenta como una aproximación sólida para mejorar la calidad del software sin frenar la velocidad de entrega.[10]

No obstante, estos avances traen aparejados retos importantes para la industria tecnológica [11]:

- Ausencia de controles adecuados en las distintas fases del ciclo de desarrollo.
- Vulnerabilidades de seguridad derivadas de malas prácticas de desarrollo.
- Riesgos de brechas de seguridad que comprometen la confidencialidad, integridad y disponibilidad de los datos.

[8] ABIONA, O., et al. The emergence and importance of DevSecOps: Integrating and reviewing security practices within the DevOps pipeline. En: World Journal of Advanced Engineering Technology and Sciences. 2024. DOI: https://doi.org/10.30574/wjaets.2024.11.2.0093.
[9] CORREA-VALOYES, J. Implementación de Herramientas de Seguridad en Pipelines CI/CD con Azure DevOps en la Empresa Periferia IT Group [en línea]. Universidad de Santander, 2025. [Consultado: 6 de junio de 2026]. Disponible en: https://repositorio.udes.edu.co/handle/001/12347
[10] KHOMH, F., et al. Understanding the impact of rapid releases on software quality. En: Empirical Software Engineering. 2015, vol. 20, p. 336-373. Disponible en: https://link.springer.com/article/10.1007/s10664-014-9308-x
[11] KHAN, R., et al. Systematic Literature Review on Security Risks and its Practices in Secure Software Development. En: IEEE Access. 2022, vol. 10, p. 5456-5481. Disponible en: https://ieeexplore.ieee.org/document/9669954

En el caso particular de Smart Campus UIS, plataforma IoT académica basada en microservicios contenerizados, estos desafíos se acentúan. La evolución del Smart Campus ha involucrado múltiples equipos, tecnologías y repositorios los cuales podrían llegar a procesar un gran volumen de información [12], generando una heterogeneidad tecnológica y organizativa que deriva en configuraciones inconsistentes, acumulación de deuda técnica y potenciales brechas de seguridad entre servicios. Esto impacta negativamente la mantenibilidad, confiabilidad y seguridad del ecosistema digital institucional.

Ante este panorama, se evidencia la necesidad de procesos que automaticen las pruebas de seguridad y la detección de errores, que escaneen vulnerabilidades en las imágenes de Docker a desplegar, que mejoren la gestión de dependencias y que proporcionen retroalimentación inmediata al equipo de desarrollo, fomentando una cultura de seguridad desde etapas tempranas.

En la industria del desarrollo de software se han adoptado diversas prácticas orientadas a fortalecer la seguridad durante el ciclo de vida de los sistemas. Entre ellas se destaca el modelo Secure Software Development Lifecycle (SSDLC), el cual incorpora medidas de seguridad en distintas fases del proceso de desarrollo. Sin embargo, dichas actividades suelen aplicarse de forma selectiva, discrecional y con escaso nivel de automatización [13]. En contraste, la metodología DevSecOps se presenta como una alternativa más reciente que se enfoca en la automatización de los procesos de integración y entregas continuas, además de incorporar de forma sistemática controles de seguridad. Este enfoque promueve la filosofía de (shift-left security), integrando análisis estáticos de código, pruebas dinámicas y revisión de dependencias en cada commit, lo que contribuye a reducir significativamente el costo y el tiempo de corrección de fallos. [14] Asimismo, DevSecOps automatiza pruebas de seguridad y despliegues

[12] JIMÉNEZ. Op. cit.
[13] RINDELL, K., et al. Security in agile software development: A practitioner survey. En: Information and Software Technology. 2021, vol. 131, art. 106488. DOI: https://doi.org/10.1016/j.infsof.2020.106488
[14] MARANDI, M.; BERTIA, A. y SILAS, S. Implementing and Automating Security Scanning to a DevSecOps CI/CD Pipeline. En: 2023 World Conference on Communication & Computing (WCONF). 2023, p. 1-6. Disponible en: https://ieeexplore.ieee.org/document/10235015

dentro de los pipelines de CI/CD, permitiendo detectar errores y vulnerabilidades en imágenes Docker y configuraciones antes de que lleguen a producción[15]. Adicionalmente, asegura la trazabilidad, el cumplimiento normativo y la automatización de los despliegues, lo cual permite agilizar los procesos y garantizar que la plataforma sea segura, escalable y confiable, alineándose con las mejores prácticas globales para aplicaciones críticas. [16]

A partir de la problemática y las consideraciones anteriores, se plantean las siguientes preguntas de investigación, cuyas respuestas guiarán el desarrollo del proyecto:

- ¿Cuál es el estado actual de los procesos de desarrollo, construcción y despliegue de los servicios de gestión y administración de Smart Campus UIS, y qué oportunidades de automatización se identifican para mejorar dichos procesos?
- ¿De qué manera se puede diseñar una arquitectura de pipelines CI/CD con enfoque automatizado para optimizar la calidad y seguridad en el entorno Smart Campus UIS?
- ¿Cómo integrar de forma continua y automatizada prácticas de seguridad como el SAST (análisis estático de código), SCA (análisis de composición de software) y escaneo de imágenes de contenedor dentro de los pipelines CI/CD de Smart Campus UIS?

[15] NIKOLOV, L. y ALEKSIEVA-PETROVA, A. Action Research on the DevSecOps Pipeline. En: 2023 International Scientific Conference on Computer Science (COMSCI). 2023, p. 1-6. Disponible en: https://ieeexplore.ieee.org/document/10315920

[16] FEIO, C., et al. An Empirical Study of DevSecOps Focused on Continuous Security Testing. En: 2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). 2024, p. 610-617. Disponible en: https://ieeexplore.ieee.org/document/10628738

2. OBJETIVOS

2.1 OBJETIVO GENERAL

Desarrollar un conjunto de pipelines que permita la automatización del flujo CI/CD con enfoque DevSecOps para la plataforma Smart Campus UIS, incorporando herramientas de análisis de seguridad a lo largo de todo el ciclo de vida del software, mediante prácticas de integración y despliegue continuos.

2.2 OBJETIVOS ESPECÍFICOS

- Analizar el estado actual de los procesos de desarrollo, construcción y despliegue de los servicios de gestión y administración de Smart Campus UIS, identificando oportunidades de automatización.
- Integrar de forma continua y automatizada prácticas de SAST (Análisis estático), SCA (Análisis de composición de software) y verificación de artefactos e imágenes.
- Diseñar una arquitectura DevSecOps, haciendo uso de herramientas conforme a las diferentes etapas del desarrollo y despliegue con el propósito de mejorar la calidad y seguridad en Smart Campus UIS.
- Documentar las configuraciones y procesos de los pipelines CI/CD con enfoque DevSecOps desarrollados en la plataforma Smart Campus UIS, con el fin de contribuir a su comprensión, replicabilidad y mantenimiento a largo plazo.

2.3 ALCANCE DEL PROYECTO

El presente trabajo se centra en el diseño e implementación de un flujo CI/CD con enfoque DevSecOps para la plataforma Smart Campus UIS, orientado a fortalecer la automatización, la trazabilidad y la incorporación temprana de controles de seguridad dentro del ciclo de vida del software. En este sentido, el alcance del proyecto abarca el análisis del estado actual de los procesos de desarrollo, construcción y despliegue de la plataforma, así como la definición de requerimientos, arquitectura, diseño técnico e

implementación de un pipeline alineado con los principios de integración continua y seguridad temprana.

La implementación desarrollada se enfocó principalmente en los microservicios de gestión y administración priorizados dentro del repositorio unificado del proyecto, particularmente en los componentes admin_microservice y data_microservice, sobre los cuales se aplicaron las etapas de compilación, pruebas unitarias, generación de artefactos, análisis estático del código fuente, análisis de dependencias, construcción de imágenes de contenedor, escaneo de seguridad y despliegue controlado en un entorno de prueba. De esta manera, el alcance cubre la automatización de validaciones funcionales iniciales y controles de seguridad sobre los componentes principales seleccionados de Smart Campus UIS.

En cuanto a la línea de seguridad integrada, el proyecto comprende la incorporación de prácticas SAST, SCA y escaneo de imágenes de contenedor dentro del flujo CI/CD, utilizando herramientas que permiten generar evidencia técnica y criterios de continuidad dentro del pipeline. Así, el alcance incluye la validación del código fuente, la revisión de dependencias externas y la evaluación de vulnerabilidades en los artefactos contenerizados antes de su despliegue en el entorno definido para pruebas.

Por otra parte, la validación posterior al despliegue se limitó a verificaciones operativas básicas mediante smoke tests, orientadas a comprobar la disponibilidad inicial de los servicios y su capacidad de respuesta en el entorno de prueba. En consecuencia, el alcance de este trabajo no abarca en esta etapa pruebas funcionales exhaustivas, pruebas de regresión completas, evaluaciones profundas de integración entre todos los componentes del ecosistema, ni la incorporación de análisis dinámico de seguridad (DAST) dentro de la implementación final del pipeline. Tampoco se plantea como objetivo la cobertura total de todos los servicios institucionales ni un despliegue productivo sobre la totalidad de la plataforma Smart Campus UIS.

En conjunto, el alcance del proyecto se orienta a establecer una base funcional y documentada de adopción DevSecOps para Smart Campus UIS, demostrando la

viabilidad de automatizar etapas clave del flujo CI/CD e integrar controles de seguridad temprana sobre un subconjunto representativo de la arquitectura, con potencial de ampliación en futuras iteraciones.

3. MARCO DE REFERENCIA

Para contextualizar adecuadamente el desarrollo del presente proyecto, el marco de referencia se divide en los siguientes apartados: antecedentes, que recoge trabajos previos relevantes de los últimos años y una serie de fundamentos teóricos, que cubren los conceptos clave, variables y herramientas relacionadas con el tema de estudio. En conjunto, estos apartados permiten comprender cómo los principios teóricos se articulan con el desarrollo investigativo y con las condiciones reales que motivan el proyecto, garantizando la coherencia entre el enfoque conceptual y el desarrollo práctico.

3.1 ANTECEDENTES DEL TEMA

Al ser un enfoque relativamente moderno, existen ya trabajos recientes que sientan precedentes sobre la implementación de DevOps/DevSecOps en contextos similares. A continuación, se destacan antecedentes (de los últimos cinco años) relevantes, cuyos hallazgos y experiencias nutren el presente proyecto:

- DevSecOps en pipelines CI/CD empresariales (UDES, 2025): En un trabajo de grado de la Universidad de Santander (UDES) el cual presentó la implementación de un marco DevSecOps en los pipelines CI/CD de una empresa de software. Dicho proyecto integró herramientas de seguridad como SonarQube (SAST), Mend Bolt (SCA) y OWASP ZAP (DAST) directamente en la tubería de despliegue automatizado, logrando mitigar vulnerabilidades críticas durante el ciclo de vida del desarrollo. Los resultados reportados incluyen una reducción del 50% en vulnerabilidades críticas en producción y mejoras en el cumplimiento normativo (ISO 27001, leyes de protección de datos) gracias a la trazabilidad y reportes automáticos generados por el pipeline [17]. Esta experiencia evidencia el impacto positivo de la automatización de controles de seguridad en entornos empresariales, aportando un referente práctico sobre la integración efectiva de herramientas DevSecOps que puede adaptarse al contexto del Smart Campus UIS.

[17] CORREA-VALOYES. Op. cit.

- DevSecOps con herramientas open-source en microservicios: En la Universidad Técnica del Norte (Ecuador), un proyecto de 2023 se orientó a implementar un enfoque DevSecOps con herramientas de código abierto en una arquitectura de microservicios, con el objetivo de la detección temprana de vulnerabilidades en el ciclo de desarrollo de software de una institución financiera core. Dicho trabajo integró scans automáticos de seguridad en cada etapa del desarrollo, evidenciando la viabilidad de adoptar DevSecOps incluso con herramientas libremente disponibles y aportando una guía práctica para entornos académicos y empresariales similares [18]. Los hallazgos resultan especialmente útiles para el presente estudio, al mostrar cómo herramientas open-source pueden adaptarse eficazmente a entornos complejos, aportando una base comparativa para la selección tecnológica del proyecto.

3.2 DEVSECOPS: CONCEPTOS Y FUNDAMENTOS TEÓRICOS.

DevSecOps se fundamenta en un enfoque ágil y automatizado que permite gestionar de manera continua el desarrollo, despliegue y operación del software. Para ello, se apoya en flujos de trabajo denominados pipelines de Integración Continua/Entrega Continua (CI/CD), los cuales orquestan el recorrido del código fuente desde su construcción inicial hasta su ejecución en producción. Estos pipelines incorporan herramientas automáticas en cada fase del ciclo de vida (compilación, pruebas, empaquetado, despliegue), lo que permite validar de forma continua la calidad y seguridad del software antes y durante su puesta en marcha [19]. DevSecOps representa un enfoque integral en el cual la seguridad se incorpora de forma temprana dentro del ciclo de vida del desarrollo de software, bajo el principio de “Shift Left Security”. Este modelo promueve la automatización y la prevención proactiva de vulnerabilidades desde las primeras etapas del proceso. A

[18] POZO RUIZ, D. M. y FIGUEROA ROSERO, J. A. Devsecops con herramientas Opensource orientados a microservicios para la detección de vulnerabilidades en el ciclo de desarrollo de software de CORE bancario para la banca personas en los Bancos Privados de la ciudad de Quito del Ecuador [en línea]. Tesis de maestría. Universidad Técnica del Norte, 2025. [Consultado: 6 de junio de 2026]. Disponible en: https://repositorio.utn.edu.ec/handle/123456789/17325

[19] CHANDRAMOULI, R. Implementation of DevSecOps for a Microservices-Based Application with Service Mesh [en línea]. Gaithersburg: National Institute of Standards and Technology, 2022. NIST Special Publication 800-204C. [Consultado: 6 de junio de 2026]. Disponible en: https://csrc.nist.gov/pubs/sp/800/204/c/final

diferencia de los enfoques tradicionales donde la seguridad se agregaba tardíamente o de manera aislada [20].

3.2.1 Integración continua y entrega continua (ci/cd). La Integración Continua (Continuous Integration) y la Entrega Continua (Continuous Delivery/Deployment) son prácticas fundamentales en DevOps que DevSecOps hereda y refuerza. Integración Continua (CI) implica integrar y compilar el código de todos los desarrolladores de forma frecuente en un repositorio común, ejecutando automáticamente pruebas unitarias y análisis estáticos con cada cambio. Esto permite detectar de inmediato errores de integración o defectos introducidos en el código fuente. Por su parte, la Entrega Continua (CD) se refiere a automatizar el despliegue de las aplicaciones en los distintos entornos (pruebas, producción) una vez que han pasado las etapas de build y testing, asegurando que siempre exista una versión candidata a ser liberada [21]. Un pipeline CI/CD combina estas prácticas para orquestar todo el flujo: cada commit desencadena un proceso automático que compila el código, ejecuta pruebas, analiza la calidad, empaqueta el artefacto y, si todo es satisfactorio, deja listo el despliegue de la nueva versión [22].

Las ventajas de CI/CD son evidentes en cuanto a agilidad y repetibilidad: se reduce el tiempo entre el desarrollo y la entrega, disminuyendo errores humanos al estar todo automatizado. No obstante, este mismo ritmo acelerado exige incluir controles de seguridad en el pipeline de integración. Aquí es donde DevSecOps añade valor, integrando pasos de verificación de seguridad en la CI/CD (por ejemplo, escaneos de vulnerabilidades o revisión de código automatizada) para que la velocidad no comprometa la seguridad. [23]

[20] GBENLE, P., et al. A DevSecOps-Centered Conceptual Model for Continuous Integration and Secure Deployment in Software Development Lifecycles. [s. l.]: [s. n.], s. f.
[21] KUMAR, N. Integrating Dynamic Security Testing Tools into CI/CD Pipelines: A Continuous Security Testing Case Study. En: International Journal of Science and Research (IJSR). 2021. DOI: https://doi.org/10.21275/sr24615152732
[22] NANDGAONKAR, S. y KHATAVKAR, V. CI-CD Pipeline For Content Releases. En: 2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT). 2022, p. 1-4. DOI: https://doi.org/10.1109/gcat55367.2022.9972129
[23] MARANDI. Op. cit.

3.2.2 Etapas esenciales de un pipeline ci/cd. Un pipeline CI/CD típico consta de varias fases o componentes por los que transita el software desde el código fuente hasta la puesta en producción, como se muestra en el cuadro 1:

Cuadro 1. Etapas de un Pipeline.

Etapa	Descripción
Configuración del código fuente	Es donde se establece un repositorio donde se almacena el código fuente del sistema, permitiendo la gestión de versiones, control de cambios y la configuración de esquemas del proyecto.
Configuración de compilaciones y ejecución	Se selecciona la herramienta de construcción adecuada para automatizar el proceso de compilación del software. Algunas opciones incluyen Ant, Maven y Gradle para Java; Make para C/C++; Grunt para JavaScript; y Rake para Ruby. Estas herramientas facilitan la integración continua en el desarrollo del sistema.
Construcción del software	Las herramientas de compilación detectan los cambios en el repositorio y generan versiones del software ejecutable. Durante este proceso se aplican pruebas automáticas, tales como: - Pruebas unitarias: verifican el correcto funcionamiento de componentes individuales del sistema. - Análisis estático del código: permite identificar errores, vulnerabilidades y verificar la conformidad con las guías de programación institucionales.
Etapa de staging	Se implementan entornos de prueba que replican el entorno de producción del sistema. En esta fase se ejecutan pruebas que aseguran la estabilidad e integración del sistema, tales como: - Pruebas de rendimiento: analizan la capacidad, estabilidad y escalabilidad bajo diferentes cargas.

	- Pruebas de cumplimiento: garantizan la adherencia a estándares de desarrollo.
Entorno de producción	Una vez validadas todas las pruebas, se despliega la versión final del proyecto en producción. Desplegando el nuevo código únicamente en una región específica antes del despliegue total, para asegurar la estabilidad y continuidad del servicio.

Fuente: Elaboración propia a partir de BASHIRU y OLUFEMI. [24]

En este contexto, el pipeline no se limita únicamente a una función de automatización técnica, sino que también actúa como una representación estructurada del ciclo de vida completo del desarrollo. Mediante la aplicación de herramientas como diagramas BPMN, tecnologías de contenedores y entornos virtualizados, se logra modelar y proteger los distintos flujos de trabajo del proyecto. Por medio de esta metodología, es viable implementar controles de seguridad en cada capa del sistema. [25]

3.2.3 Conceptos fundamentales de ciberseguridad. Todo pipeline DevSecOps debe sustentarse en principios sólidos de ciberseguridad para orientar sus controles. Los pilares fundamentales de la seguridad de la información se resumen en la llamada tríada CIA: Confidencialidad, Integridad y Disponibilidad. La confidencialidad implica proteger los datos contra accesos no autorizados; la integridad se refiere a asegurar que la información no sea alterada indebidamente, manteniendo su exactitud y procedencia legítima; y la disponibilidad procura que los sistemas y datos estén accesibles cuando se necesiten, tolerando fallos y ataques de denegación de servicio [26].

[24] BASHIRU, O. y OLUFEMI, O. An Enhanced CICD Pipeline: A DevSecOps Approach. En: International Journal of Computer Applications. 2023. DOI: https://doi.org/10.5120/ijca2023922594.
[25] BASINYA, E. A. y MALYSHEV, E. A. The Creation and Integration of the Technological Workflow for Software and Hardware-Software Development. En: 2023 IEEE XVI International Scientific and Technical Conference Actual Problems of Electronic Instrument Engineering (APEIE). 2023, p. 960-966. Disponible en: https://ieeexplore.ieee.org/document/10347739
[26] OWASP. Fundamentos de seguridad. En: OWASP Developer Guide [sitio web]. s. f. [Consultado: 6 de junio de 2026]. Disponible en: https://devguide.owasp.org/es/02-foundations/01-security-fundamentals

Estos tres principios guían la definición de requisitos de seguridad en cualquier sistema: por ejemplo, en Smart Campus UIS, garantizar la confidencialidad de datos sensibles de la comunidad universitaria, la integridad de las transacciones IoT que allí ocurren y la disponibilidad continua de los servicios educativos.

En el ámbito de desarrollo de software, además, existen referencias importantes como el estándar OWASP Top 10, que lista las principales vulnerabilidades en aplicaciones web, o los modelos de madurez de seguridad de software que ayudan a estructurar prácticas de seguridad a nivel organizacional [27]. Para este proyecto, cobran relevancia conceptos como la gestión de vulnerabilidades (identificación, evaluación y remediación de fallos de seguridad en el código o en componentes de terceros) y la seguridad en contenedores (asegurando imágenes Docker, credenciales y configuraciones).

3.2.4 Sast (análisis estático de seguridad de aplicaciones). El Análisis Estático de Código (Static Application Security Testing, SAST) es una técnica de prueba de seguridad que examina el código fuente de la aplicación sin ejecutar el programa, con el objetivo de identificar vulnerabilidades de seguridad, errores de programación y puntos débiles potenciales. Las herramientas SAST analizan el flujo de datos y el flujo de control del software, así como patrones conocidos de codificación insegura [28]. Una característica clave de SAST es que se aplica típicamente en etapas tempranas del desarrollo: los análisis pueden integrarse en el entorno de desarrollo del programador o ejecutarse automáticamente en cada commit dentro del pipeline CI [29].

Al detectar vulnerabilidades durante la construcción del código, se evita incurrir en costos elevados de corrección en etapas posteriores o en producción. Esto concuerda con el enfoque DevSecOps: los desarrolladores reciben retroalimentación inmediata de la

[27] BACH-NUTMAN, M. Understanding The Top 10 OWASP Vulnerabilities. En: ArXiv. 2020, abs/2012.09960. DOI: https://doi.org/10.48550/arXiv.2012.09960
[28] ESPOSITO, M.; FALASCHI, V. y FALESSI, D. An Extensive Comparison of Static Application Security Testing Tools. En: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering. 2024. DOI: https://doi.org/10.1145/3661167.3661199
[29] CROFT, R., et al. An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing. En: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 2021. DOI: https://doi.org/10.1145/3475716.3475781

herramienta SAST y pueden corregir el defecto antes de que avance en el pipeline. Además, las herramientas de análisis estático suelen generar reportes detallados indicando la ubicación exacta del problema en el código y recomendaciones para solucionarlo, lo cual acelera el proceso de remediación. [30]

En el contexto de Smart Campus UIS, la adopción de SAST permitiría fortalecer la calidad del código de los microservicios desde su origen. Cada vez que un desarrollador integre un cambio, el pipeline podría ejecutar un análisis estático (p. ej., con SonarQube) para verificar estándares de codificación y detectar vulnerabilidades conocidas o malas prácticas [31]. Esto crea un punto de control automático que eleva la barra de seguridad sin ralentizar significativamente el desarrollo, ya que todo ocurre de forma transparente en segundo plano. En suma, SAST es un componente esencial del pilar "DevSec" de DevSecOps, puesto que introduce la seguridad dentro del código antes de que éste se ejecute o despliegue.

3.2.5 Sca (análisis de composición de software). El Análisis de Composición de Software (SCA, Software Composition Analysis) es el proceso enfocado en identificar y gestionar los componentes de código abierto y de terceros que una aplicación utiliza, con el fin de controlar los riesgos asociados a ellos [32]. En la actualidad, la mayoría de las aplicaciones modernas dependen de múltiples librerías, frameworks y paquetes de terceros (open source) que agilizan el desarrollo. Sin embargo, estos componentes externos pueden introducir vulnerabilidades conocidas, código malicioso o problemas de licenciamiento en la aplicación si no se supervisan adecuadamente [33]. Por ejemplo, si Smart Campus UIS utiliza una versión desactualizada de una librería web con fallos de seguridad reportados, el SCA la detectaría y alertaría de la necesidad de actualizar a una versión segura. Asimismo, el análisis de composición permite administrar licencias de

[30] PISKACHEV, G.; BECKER, M. y BODDEN, E. Can the configuration of static analyses make resolving security vulnerabilities more effective? A user study. En: Empirical Software Engineering. 2023, vol. 28, p. 1-28. DOI: https://doi.org/10.1007/s10664-023-10354-3
[31] ESPOSITO. Op. cit.
[32] NOCERA, S., et al. Software Composition Analysis and Supply Chain Security in Apache Projects: an Empirical Study. En: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR). 2025, p. 103-115. DOI: https://doi.org/10.1109/msr66628.2025.00027
[33] PRANA, G., et al. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. En: Empirical Software Engineering. 2021, vol. 26. DOI: https://doi.org/10.1007/s10664-021-09959-3

software, identificando si alguna dependencia tiene licencias restrictivas o incompatibles con la política institucional.

3.2.6 Análisis dinámico (dynamic application security testing – dast). El análisis dinámico constituye una de las prácticas en la verificación de la seguridad del software, ya que permite evaluar el comportamiento de una aplicación durante su ejecución. A diferencia del análisis estático, que examina el código fuente sin ejecutarlo, el análisis dinámico se centra en la detección de vulnerabilidades en tiempo de ejecución, tales como fallos de validación de entradas, errores de manejo de sesiones o configuraciones inseguras [34]. Esta técnica resulta especialmente relevante en contextos donde la automatización de pruebas forma parte de los pipelines de DevSecOps, puesto que posibilita la identificación temprana de riesgos en entornos que simulan condiciones reales de producción. Sin embargo, en metodologías más tradicionales como el Secure Software Development Lifecycle (SSDLC), estas evaluaciones suelen llevarse a cabo de forma manual o mediante herramientas independientes, lo que limita su frecuencia y capacidad de respuesta ante cambios en el código [35].

3.3 SMART CAMPUS UIS

Un Smart Campus puede considerarse una extensión del concepto de Ciudad Inteligente, ya que integra tecnologías digitales y una infraestructura física que permiten la captura, el procesamiento y el monitoreo en tiempo real de datos [36]. Dado el amplio conjunto de servicios y la complejidad operativa que caracterizan a estas plataformas, resulta indispensable disponer de infraestructuras tecnológicas robustas y escalables que garanticen el cumplimiento de funcionalidades y metas operativas. En este marco, se han planteado arquitecturas basadas en microservicios [37], orientadas al desarrollo de

[34] AGGARWAL, A. y JALOTE, P. Integrating Static and Dynamic Analysis for Detecting Vulnerabilities. En: 30th Annual International Computer Software and Applications Conference (COMPSAC'06). 2006, vol. 1, p. 343-350. DOI: https://doi.org/10.1109/compsac.2006.55.
[35] RINDELL. Op. cit.
[36] BELLINI, P.; NESI, P. y PANTALEO, G. IoT-Enabled Smart Cities: A Review of Concepts, Frameworks and Key Technologies. En: Applied Sciences. 2022. DOI: https://doi.org/10.3390/app12031607.
[37] JIMÉNEZ. Op. cit.

plataformas de software interoperables y de alta usabilidad para los diferentes actores del entorno universitario.

3.4 MARCO DE HERRAMIENTAS PARA LA INTEGRACIÓN CI/CD CON ENFOQUE DEVSECOPS

La implementación de DevSecOps en el flujo CI/CD implica el uso de distintas herramientas en cada etapa del pipeline, entre ellas varias de seguridad, las cuales se usarán desde la integración continua hasta el despliegue.

A continuación, se describen las herramientas que resultan las posibilidades más adecuadas (Principalmente opensource) para cada parte del pipeline DevSecOps.

3.4.1 Herramientas de orquestación de ci/cd (integración continua y despliegue). La orquestación de CI/CD consiste en automatizar y coordinar todas las etapas del ciclo de vida del software, desde que se integra el código hasta que se despliega en producción.

Cuadro 2. Herramientas de Orquestación CI/CD

Herramienta	Descripción y rol
Jenkins	Es una herramienta de automatización de código abierto ampliamente adoptado para la implementación de pipelines CI/CD, contando con una gran variedad de plugins que facilitan y optimizan los procesos de automatización y pruebas. Gracias a su flexibilidad y solidez, se ha convertido en una de las herramientas más utilizadas para crear pipelines de integración y entrega continua que sean escalables y confiables.[38]
GitHub	Reconocida como una plataforma de CI/CD, su rol fundamental es

[38] AMGOTHU, S. An End-to-End CI/CD Pipeline Solution Using Jenkins and Kubernetes. En: International Journal of Science and Research (IJSR). 2024. Disponible en: https://www.ijsr.net/archive/v13i8/SR24826231120.pdf

Actions	actuar como un orquestador que permite definir flujos de trabajo automatizados (workflows) directamente en el repositorio de código fuente. Utilizado en combinación con el control de versiones de Git y facilitando la integración continua al permitir que cada commit desencadene un conjunto de acciones predefinidas, automatizando y coordinando las etapas del ciclo de vida del software.[39]

Fuente: Elaboración propia a partir de AMGOTHU; MANOLOV, GOTSEVA y HINOV.

3.4.2 Herramientas de construcción y pruebas. La automatización de la compilación y las pruebas es fundamental en un pipeline de Integración continua (CI). Para las aplicaciones, la selección de herramientas de construcción (Como Maven y Gradle) es crucial para garantizar la calidad y la eficiencia del ciclo de vida del software. [40]

Cuadro 3. Herramientas de construcción y pruebas.

Herramienta	Descripción y rol
Maven	Es una herramienta muy utilizada para automatizar la construcción de proyectos en Java. En el flujo del desarrollo, se encarga de automatizar la compilación, ejecución de pruebas unitarias y empaquetado de la aplicación. Además, gestiona automáticamente las dependencias, descargando las librerías necesarias desde repositorios como Maven Central. Su funcionamiento se basa en un archivo llamado POM (Product Object Model), donde se definen los datos del proyecto, sus dependencias y el proceso de construcción. Gracias a esto, Maven facilita la integración continua dentro del pipeline CI/CD, manteniendo

[39] MANOLOV, V.; GOTSEVA, D. y HINOV, N. Practical Comparison Between the CI/CD Platforms Azure DevOps and GitHub. En: Future Internet. 2025, vol. 17, art. 153. Disponible en: https://www.mdpi.com/1999-5903/17/4/153

[40] JOHNSON, O., et al. Developing advanced CI/CD pipeline models for Java and Python applications: A blueprint for accelerated release cycles. En: Computer Science & IT Research Journal. 2024. Disponible en: https://fepbl.com/index.php/csitrj/article/view/1758

	las versiones de las dependencias bajo control. [41]
Gradle	Es una herramienta de automatización muy utilizada en proyectos Java, especialmente dentro de pipelines CI/CD. Permitiendo configurar y ejecutar compilaciones de manera eficiente, facilitando el proceso de construcción, pruebas y despliegue de aplicaciones. Además, por ser de código abierto, se usa ampliamente en la creación de frameworks de automatización de pruebas empresariales.[42]

Fuente: Elaboración propia a partir de SOTO-VALERO, et al.; JOHNSON, et al. y DAS.

3.4.3 Herramientas de análisis estático de código (SAST). El análisis estático de código (SAST) es esencial en DevSecOps para detectar vulnerabilidades de seguridad en etapas tempranas del desarrollo. [43] Herramientas como las que se van a mencionar a continuación son ampliamente utilizadas para este fin y presentan diferencias entre sí en aspectos como cobertura, precisión y facilidad de integración.

Cuadro 4. Herramientas de análisis estático de código (SAST).

Herramienta	Descripción y rol
SonarQube	Es una herramienta de código abierto muy usada para evaluar la calidad y seguridad del código. En su versión gratuita soporta Java y varios lenguajes más, detectando errores, malas prácticas y posibles vulnerabilidades. Permite definir “Quality Gates”, es decir, reglas mínimas de calidad y seguridad que deben cumplirse. Si el análisis encuentra fallas críticas (Tipo inyecciones SQL, XSS o uso inseguro de APIs) el pipeline puede marcarse como fallido.

[41] SOTO-VALERO, C., et al. A comprehensive study of bloated dependencies in the Maven ecosystem. En: Empirical Software Engineering. 2020, vol. 26. Disponible en: https://link.springer.com/content/pdf/10.1007/s10664-020-09914-8.pdf
[42] DAS, K. Create an Enterprise-Level Test Automation Framework with Appium: Using Spring-Boot, Gradle, Junit, ALM Integration, and Custom Reports with TDD and BDD Support. Berkeley: Apress, 2022. Disponible en: https://link.springer.com/book/10.1007/978-1-4842-8197-0
[43] PRATES. Op. cit.

	Integrado en herramientas como Jenkins o GitHub Actions, SonarQube ayuda a detectar problemas de seguridad desde etapas tempranas del desarrollo (shift-left security) . [44] [45] [46]
SpotBugs + FindSecBugs	SpotBugs y FindSecBugs forman un conjunto de herramientas de código abierto muy usado para analizar aplicaciones Java. SpotBugs se encarga de revisar el bytecode compilado para detectar errores y vulnerabilidades, mientras que FindSecBugs amplía sus capacidades al enfocarse específicamente en problemas de seguridad. Esta herramienta juega un papel importante al facilitar pruebas más eficientes y detectar errores desde etapas tempranas del desarrollo (shift-left security). [47]

Fuente: Elaboración propia a partir de PRATES y PEREIRA; TAHAEI, et al. y ESPOSITO, FALASCHI y FALESSI.

3.4.4 Herramientas de analisis de composicion del software (SCA). SCA es una actividad que resulta fundamental en el pipeline CI/CD del proyecto, cuyo propósito principal es identificar y gestionar los componentes de código abierto y de terceros que utiliza la aplicación, con el fin de controlar los riesgos asociados. [48]

Para implementar SCA y el escaneo de artefactos en el flujo CI/CD es necesaria la elección de herramientas para este fin, tales como algunas de las que se van a mencionar a continuación.

Cuadro 5. Herramientas de análisis de composición del software (SCA).

Herramienta	Descripción y rol

[44] TAHAEI, M., et al. Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them. En: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. 2021. DOI: https://doi.org/10.1145/3411764.3445616
[45] ESPOSITO. Op. cit.
[46] PRATES. Op. cit.
[47] TAHAEI. Op. cit.
[48] PRANA. Op. cit.

OWASP Dependency-Check	Es una herramienta de código abierto muy reconocida dentro de la comunidad OWASP, utilizada para detectar vulnerabilidades en librerías y dependencias de un proyecto. Su funcionamiento se basa en analizar los metadatos de cada componente (como nombre y versión) y compararlos con bases de datos de vulnerabilidades conocidas (CVE). Aunque puede generar falsos positivos (ya que depende principalmente de información descriptiva) sigue siendo una pieza clave dentro de un pipeline, ayudando a identificar dependencias inseguras o versiones modificadas de librerías que podrían representar un riesgo. [49]
Dependabot	Es una herramienta integrada en GitHub que automatiza la actualización de dependencias para mantener el código al día y mejorar la seguridad del proyecto. Ofrece dos funciones principales: Una que actualiza versiones de dependencias según un archivo, y otra que detecta vulnerabilidades en librería usadas, generando pull requests automáticos con las correcciones necesarias. [50]
Trivy (Modo SCA)	Es una herramienta de código abierto diseñada para analizar vulnerabilidades en contenedores o imágenes Docker. Su función principal es detectar paquetes o configuraciones inseguras dentro de las imágenes, ayudando a reducir la deuda de seguridad al identificar riesgos antes de que lleguen a producción. Al integrarse en el pipeline CI/CD, Trivy permite realizar análisis

[49] PONTA, S.; PLATE, H. y SABETTA, A. Detection, assessment and mitigation of vulnerabilities in open source dependencies. En: Empirical Software Engineering. 2020, vol. 25, p. 3175-3215. DOI: https://doi.org/10.1007/s10664-020-09830-x

[50] HE, R., et al. Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. En: IEEE Transactions on Software Engineering. 2022, vol. 49, p. 4004-4022. DOI: https://doi.org/10.1109/tse.2023.3278129

	automáticos de las imágenes durante el proceso de construcción, garantizando que las aplicaciones contenerizadas sean más seguras. [51]

Fuente: elaboración propia a partir de PRANA, et al.; PONTA, PLATE y SABETTA; HE, et al. y ALI, MOHAMED y MAGDY.

3.4.5 Herramientas de análisis dinámico de aplicaciones (DAST). A diferencia del análisis estático, que revisa el código fuente, el DAST evalúa cómo se comporta la aplicación mientras se ejecuta. Esto permite detectar vulnerabilidades en tiempo real e identificar posibles riesgos en entornos que simulan condiciones reales de producción, lo que contribuye a una seguridad más sólida desde las primeras etapas del desarrollo.[52]

Para implementar DAST en el flujo CI/CD es necesaria la elección de herramientas para este fin, tal la que se va a mencionar a continuación.

Cuadro 6. Herramientas de análisis dinámico de aplicaciones (DAST).

Herramienta	Descripción y rol
OWASP ZAP (Zed Attack Proxy)	Es una herramienta de código abierto recomendada por la fundación OWASP para realizar pruebas de seguridad en aplicaciones web (WAST). Su objetivo es detectar vulnerabilidades comunes, como inyecciones SQL o fallos de autenticación. Integrado en un pipeline CI/CD, permite realizar pruebas de seguridad continuas, ejecutándose como un servicio externo que analiza cada nueva versión de la aplicación. Además, clasifica los riesgos detectados por nivel de seriedad (informativo, bajo, medio o alto) y genera reportes que pueden usarse para decidir si un build cumple con

[51] HE, R., et al. Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. En: IEEE Transactions on Software Engineering. 2022, vol. 49, p. 4004-4022. DOI: https://doi.org/10.1109/tse.2023.3278129
[52] RINDELL. Op. cit.

	los estándares de seguridad. [53] [54] [55]

Fuente: elaboración propia a partir de RINDELL, et al.; RAHMAN, et al. y RANGNAU, et al.

[53] علي. Op. cit.
[54] RAHMAN, A., et al. Analisis Implementasi Nucklei Vulnerability dan OWASP-ZAP Scanner untuk Deteksi Kerentanan Keamanan (Secure System) pada Platform Web Based. En: Jurnal Komputer Terapan. 2025. DOI: https://doi.org/10.35143/jkt.v11i1.6430
[55] RANGNAU, T., et al. Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. En: 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC). 2020, p. 145-154. DOI: https://doi.org/10.1109/edoc49727.2020.00026

4. METODOLOGÍA

Para el desarrollo del presente trabajo se adoptó un enfoque metodológico estructurado en fases consecutivas, correspondiente al modelo de cascada, también conocido como “waterfall”. Este enfoque permitió organizar el proyecto de manera secuencial, articulando cada fase con los objetivos específicos del trabajo de grado y con los productos obtenidos en cada etapa. A través de esta estructura fue posible avanzar desde el análisis del estado actual de la plataforma hasta el diseño, implementación, validación y documentación de un flujo CI/CD con enfoque DevSecOps para Smart Campus UIS. [56] [57]

4.1 FASE 1: AMBIENTACIÓN CONCEPTUAL Y BASES DEL ENTORNO.

En esta fase se establecieron los fundamentos conceptuales, tecnológicos y organizativos del proyecto. Se configuró el entorno de trabajo, se estructuró el repositorio base y se prepararon las herramientas necesarias para iniciar el desarrollo del pipeline DevSecOps, garantizando una base adecuada para las siguientes fases del ciclo de vida del software.

HITO

Entorno inicial configurado: Esta fase se consideró completada una vez se dispuso de un repositorio estructurado, un servidor básico de orquestación CI/CD activo y un pipeline inicial funcional orientado a la compilación automatizada.

ACTIVIDADES

A1.1 Configuración y definición del repositorio de código fuente en GitHub con control de la versión analizada mediante rama y commit.

A1.2 Instalación y configuración inicial del orquestador del pipeline.

[56] SARAVANOS, A. y CURINGA, M. Simulating the Software Development Lifecycle: The Waterfall Model. En: ArXiv. 2023, abs/2308.03940. DOI: https://doi.org/10.48550/arxiv.2308.03940

[57] WISIDAGAMA, N. y MARIKKAR, F. Waterfall Model over PCD.UCT Model Review. En: Automation of Technological and Business Processes. 2024. DOI: https://doi.org/10.15673/atbp.v16i3.2927

A1.3 Definición del flujo y definición del pipeline con etapas básicas de build y test.

PRODUCTOS OBTENIDOS

P1.1 – Repositorio estructurado del código fuente.

P1.2 – Servidor de orquestación CI/CD configurado.

P1.3 – Pipeline básico funcional con compilación automatizada.

4.2 FASE 2: DEFINICIÓN DE LINEAMIENTOS DE SEGURIDAD Y CALIDAD DEL SOFTWARE

Esta fase tuvo como propósito identificar los requerimientos funcionales y de seguridad del pipeline, así como establecer los criterios mínimos de calidad y los mecanismos de análisis que orientarían la solución propuesta. En esta etapa también se definió la línea base tecnológica para la integración de controles de seguridad dentro del flujo CI/CD.

HITO

Lineamientos de seguridad definidos: Al finalizar esta fase quedaron documentados los requerimientos funcionales y de seguridad del pipeline, junto con la selección base de las herramientas SAST y SCA adoptadas en el proyecto.

ACTIVIDADES

A2.1 – Identificación de requerimientos funcionales y de seguridad del proyecto.

A2.2 – Definición de las herramientas SAST y SCA para el análisis estático y de dependencias.

A2.3 – Documentación de los requerimientos técnicos y de seguridad del pipeline.

PRODUCTOS OBTENIDOS

P2.1 – Documentación de requerimientos funcionales y de seguridad.

P2.2 – Definición base de herramientas SAST/SCA establecida.

4.3 FASE 3: DISEÑO SHIFT LEFT SECURITY.

En esta fase se definió la arquitectura técnica del pipeline DevSecOps y se diseñaron los flujos de integración, validación, análisis de seguridad, contenerización y despliegue. También se establecieron las relaciones entre los componentes y las herramientas seleccionadas, buscando garantizar trazabilidad, control y coherencia entre el diagnóstico inicial y la solución planteada.

HITO

Diseño arquitectural del pipeline: Esta fase culminó con la elaboración del diagrama de arquitectura del pipeline DevSecOps, el diseño técnico detallado y la definición de las etapas principales del flujo automatizado.

ACTIVIDADES

A3.1 – Diseño del flujo integral del pipeline con etapas de build, test, análisis y despliegue.
A3.2 – Definición de la arquitectura de contenerización y despliegue (Docker).
A3.3 – Diseño de los mecanismos de escaneo de imágenes y dependencias.
A3.4 – Elaboración de la documentación técnica del diseño arquitectural.

PRODUCTOS OBTENIDOS

P3.1 – Diagrama de arquitectura del pipeline DevSecOps.
P3.2 – Documento de diseño técnico detallado.
P3.3 – Pipeline definido con etapas de validación y seguridad.

4.4 FASE 4: CONFIGURACIÓN, INTEGRACIÓN Y CONTENERIZACIÓN SEGURA.

En esta fase se implementó la solución planteada en el diseño arquitectural. Se construyeron y automatizaron las etapas del pipeline, integrando herramientas para la compilación, ejecución de pruebas unitarias, análisis estático del código, análisis de dependencias, construcción de imágenes y escaneo de seguridad sobre contenedores. Así mismo, se documentó la configuración técnica necesaria para soportar el flujo implementado.

HITO

Pipeline implementado con seguridad: Al finalizar esta fase el pipeline quedó operativo, integrando mecanismos de validación funcional inicial, análisis SAST, análisis SCA, construcción de imágenes y escaneo de seguridad sobre artefactos contenerizados.

ACTIVIDADES

A4.1 – Integración de herramientas de análisis estático de código (SAST) como SonarQube en el pipeline.

A4.2 – Escaneo de dependencias para detectar librerías vulnerables.

A4.3 – Automatización de validaciones del código y generación de artefactos de construcción para los componentes priorizados del sistema.

A4.4 – Automatización de la construcción de imágenes Docker para los componentes priorizados del proyecto.

A4.5 – Integración del análisis de imágenes para detectar vulnerabilidades en los contenedores generados.

A4.6 – Documentación detallada del flujo de construcción y validación de contenedores.

PRODUCTOS OBTENIDOS

P4.1 – Pipeline funcional implementado.

P4.2 – Imágenes Docker analizadas y validadas dentro del flujo definido.

P4.3 – Artefactos de despliegue del sistema.

4.5 FASE 5: VALIDACIÓN Y CIERRE DEL PIPELINE DEVSECOPS.

En la fase final se validó el comportamiento general del pipeline una vez integradas sus etapas principales. Para ello, se ejecutó el despliegue del sistema en un entorno de prueba y se realizaron verificaciones operativas básicas orientadas a comprobar la disponibilidad inicial de los servicios desplegados. Esta fase también incluyó la consolidación de la documentación final del proyecto y la integración de los resultados obtenidos en el documento de grado.

HITO

Validación y cierre: Esta fase culminó con el despliegue del sistema en un entorno de prueba, la ejecución de verificaciones operativas básicas posteriores al despliegue y la consolidación del documento final del proyecto.

ACTIVIDADES

A5.1 – Despliegue automatizado del contenedor en un entorno de pruebas.

A5.2 – Ejecución de verificaciones operativas básicas posteriores al despliegue, mediante smoke tests y validación de disponibilidad de servicios.

A5.3 – Consolidación de resultados, evidencias y reportes generados por el pipeline.

A5.4 – Documentación final del pipeline completo con sus fases de CI/CD y controles de seguridad aplicados.

A5.5 – Integración y estructuración de toda la documentación en un único documento final consolidado.

A5.6 – Revisión, ajustes y complementos necesarios para la versión definitiva del proyecto de grado.

PRODUCTOS OBTENIDOS

P5.1 – Entorno de prueba validado y operativo.

P5.2 – Reporte de resultados de validación y pruebas.

P5.3 – Versión del sistema verificada.

5. FASE 1: AMBIENTACIÓN CONCEPTUAL Y BASES DEL ENTORNO

El presente análisis constituye un antecedente técnico que constituyó las bases para el diseño y construcción de una arquitectura con enfoque DevSecOps aplicada a la plataforma Smart Campus UIS. Su propósito no se limita a describir la arquitectura existente; si no que busca ofrecer una exploración sistemática que permita examinar de manera integral la forma en la que el software es concebido, construido y puesto en operación.

Este diagnóstico permite comprender no solo la estructura funcional de la plataforma, sino que además establece el fundamento que define la estrategia de automatización e integración de controles de seguridad y busca identificar brechas técnicas, operativas y de seguridad que puedan mitigarse mediante prácticas de integración continua, despliegue automatizado y validaciones sistemáticas, apuntando así a la evolución controlada y segura de la plataforma.

5.1 CARACTERIZACIÓN ARQUITECTÓNICA ACTUAL

La plataforma Smart Campus UIS se encuentra estructurada bajo un modelo de servicios desacoplados que interactúan mediante mecanismos de mensajería asíncrona, particularmente a través de protocolos utilizados en entornos IoT. El sistema distingue dos componentes “core” claramente definidos: un componente encargado del procesamiento de datos provenientes de dispositivos IoT y un componente orientado a la gestión administrativa. Estos componentes se apoyan en servicios tales como brokers de mensajería, bases de datos, herramientas de monitoreo y visualización.

Desde el punto de vista estructural la arquitectura adopta principios asociados al paradigma de microservicios, además que los componentes se encuentran contenerizados. Los servicios identificados incluyen el servicio de procesamiento de datos IoT, el servicio de administración, un API Gateway, brokers MQTT y AMQP, una base de datos administrativa, un motor de almacenamiento de series temporales (InfluxDB), una herramienta de visualización (Grafana) y un simulador de dispositivos IoT

utilizado para pruebas y validación manual. La orquestación de estos servicios se realiza desde un repositorio central denominado “workshop”, el cual agrupa la configuración necesaria para levantar el entorno completo mediante Docker Compose. Adicionalmente, algunos de estos componentes se encuentran bajo un aislamiento lógico en un repositorio independiente asignado dentro del ecosistema.

5.2 ANÁLISIS DEL PROCESO DE DESARROLLO

El desarrollo para la plataforma Smart Campus UIS se encuentra distribuido en múltiples repositorios independientes, cada uno correspondiente a servicios específicos. Esta organización favorece la modularidad y el aislamiento lógico; no obstante, no se evidencia la existencia de una estrategia para la gestión de políticas de revisión de código ni mecanismos automatizados de validación.

En contraste, un enfoque centralizado, en el que todos los servicios se gestionan desde un repositorio unificado, similar como se encuentra actualmente de manera parcial el repositorio workshop, evidencia la posibilidad de ejecutar de manera sistemática un flujo de integración y despliegue continuo CI/CD, facilitando así automatización de pruebas unitarias, análisis estático de código y escaneo de dependencias en momentos específicos del ciclo de vida del software. La implementación de un repositorio centralizado con un pipeline CI/CD no solo permitiría la integración temprana de la seguridad (Shift Left Security), sino también mejora la consistencia en los despliegues, la trazabilidad completa de los cambios y visibilidad de métricas de calidad, estableciendo una base sólida para un modelo DevSecOps en la plataforma Smart Campus UIS.

5.3 ANÁLISIS DEL PROCESO DE CONSTRUCCIÓN

El proceso de construcción de artefactos presenta diferencias estructurales entre los microservicios administrativos y de datos, lo cual abre la posibilidad de fortalecer la estandarización en la estrategia de construcción y control de la cadena de suministro.

En el caso del microservicio administrativo, el Dockerfile instala (OPENJDK17) y copia directamente un archivo .jar previamente generado. Este enfoque presupone que el artefacto JAR ha sido construido con anterioridad en un entorno externo, por lo que la construcción de dicho archivo queda desacoplada del proceso de generación de la imagen. Dicha práctica introduce una dependencia implícita en el entorno local del desarrollador y en procesos manuales no auditados. Por otra parte, el microservicio de datos adopta un enfoque de ejecución directo desde un archivo Dockerfile que clona el repositorio de desarrollo durante el proceso de construcción de la imagen, en este escenario el código fuente incorporado depende del estado remoto del repositorio en el momento de la ejecución del build, y no de un proceso validado dentro de un flujo CI/CD.

El entorno actual refleja oportunidades de fortalecimiento en términos de reproducibilidad y seguridad a lo largo del ciclo de vida de la plataforma. La transición hacia un repositorio centralizado con ejecución de un flujo CI/CD permitiría eliminar la dependencia de entornos locales, asegurar la correspondencia entre código validado y artefacto desplegado, e incorporar controles automatizados que fortalezcan la integridad del proceso de construcción.

5.4 ANÁLISIS DEL PROCESO DE DESPLIEGUE

El despliegue de la plataforma se ejecuta mediante la invocación manual de Docker Compose, utilizando un archivo de orquestación que centraliza la definición de los servicios que componen el ecosistema Smart Campus UIS. Este enfoque permite levantar de manera unificada los distintos componentes tales como bases de datos, broker de mensajería, gateway, frontend entre otros. Facilitando su integración funcional. Posteriormente se realizan configuraciones complementarias tales como la conexión de herramientas de visualización con las bases de datos, registros de usuarios y configuración de dispositivos simulados. Dichas actividades se realizan por medio de intervención directa del operador o desarrollador encargado.

Si bien este esquema privilegia la flexibilidad y el control directo sobre la infraestructura, algunas configuraciones manuales o ausencia de scripts automatizados podrían limitar la

posibilidad de estandarizar completamente el proceso, por lo cual el modelo actual puede caracterizarse por un proceso de despliegue centralizado y funcional, apoyado en Docker Compose como herramienta de orquestación, pero con cierto grado de intervención manual. Desde una perspectiva orientada a la evolución hacia DevSecOps, se identifica la oportunidad de fortalecimiento tanto en automatización como en control sistemático de calidad y seguridad.

5.5 SÍNTESIS DEL ANÁLISIS

El análisis realizado evidencia que la plataforma Smart Campus UIS dispone de una arquitectura funcionalmente adecuada, basada en microservicios desacoplados y contenerizados, lo que constituye precedente técnico favorable para su evolución. No obstante, los procesos de desarrollo, construcción y despliegue presentan limitaciones estructurales en términos de automatización, trazabilidad y control sistemático de seguridad.

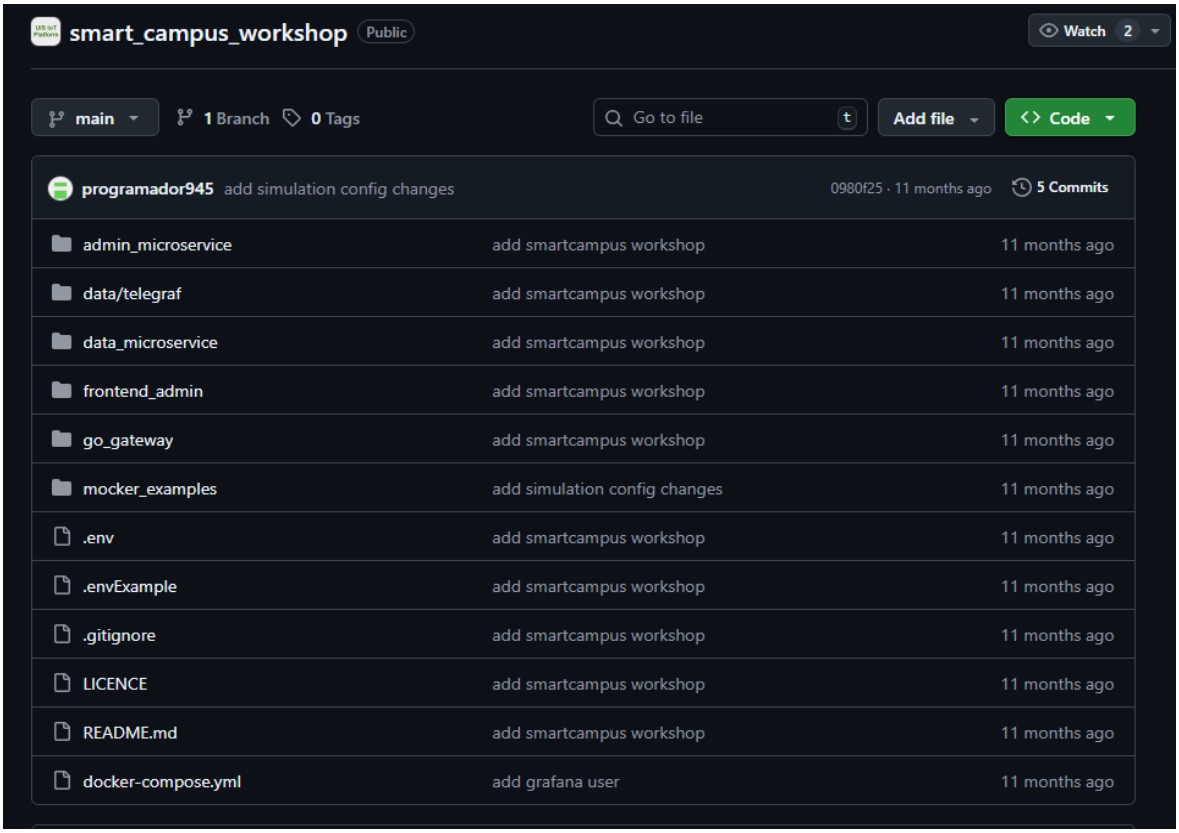
En conjunto, estas condiciones reflejan oportunidades claras de fortalecimiento en el ciclo de vida del software, particularmente en procesos de validación, automatización y seguridad. En este contexto, la incorporación de un servidor orientado a la orquestación de pipelines de integración y despliegue continuo como Jenkins se perfila como una alternativa viable para centralizar la construcción de artefactos, integrar controles de calidad y seguridad, con el fin de mejorar la trazabilidad entre cambios de código y despliegues. Entre sus principales ventajas Jenkins posee una arquitectura basada en plugins que permiten definir pipelines con capacidad de integrarse con diferentes herramientas de análisis estático, escaneo de dependencias, pruebas automatizadas y gestión de contenedores. Estas características lo convierten en un entorno adecuado para estructurar un flujo CI/CD alineado con principios DevSecOps, sin requerir modificaciones estructurales en la arquitectura funcional existente.

Esta síntesis confirma que la evolución hacia un modelo DevSecOps no implica una transformación radical de la arquitectura de servicios actual, sino la consolidación de un modelo operativo más controlado, automatizado y alineado con buenas prácticas en el ciclo de vida del software.

5.6 REPOSITORIO ESTRUCTURADO DEL CÓDIGO FUENTE - P1.1

Para la gestión y organización del código fuente del proyecto se utilizó un repositorio estructurado basado en el repositorio Workshop usado actualmente para generar despliegues, el cual fue tomado como punto de partida debido a que provee una arquitectura inicial adecuada para iniciar con el proceso de automatización del flujo CI/CD.

Figura 1. Estructura base del repositorio smart_campus_workshop utilizado como punto de partida para el proyecto.



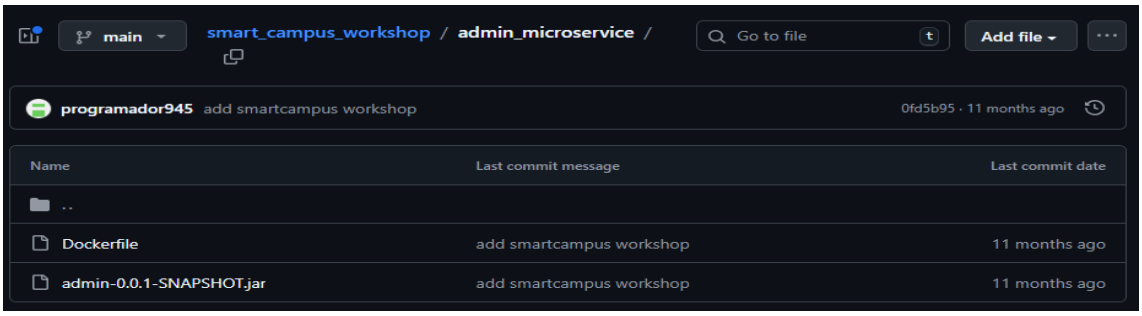
A partir de esta base, se realizó la adaptación y extensión de la estructura del repositorio con el fin de integrar los componentes necesarios para el desarrollo y evaluación del proyecto. Específicamente, se incorporó el código fuente correspondiente a los microservicios dentro de las carpetas **admin_microservice** y **data_microservice**,

permitiendo así centralizar la lógica de aplicación de cada servicio dentro de la arquitectura del repositorio.

Esta integración permitió disponer del código fuente necesario para la ejecución de pruebas contempladas dentro del enfoque DevSecOps propuesto en el proyecto, debido a que anteriormente al hacer el despliegue usando directamente una imagen previamente generada no era posible obtener exhaustividad y profundidad en las pruebas.

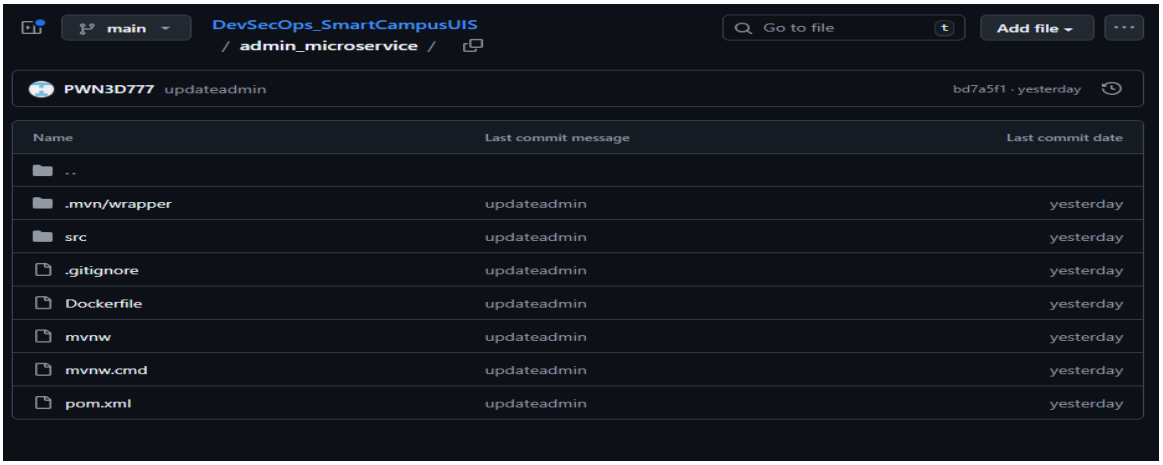
Antes:

Figura 2. ANTES - Comparación de la estructura del directorio admin_microservice antes y después de la integración del código fuente.



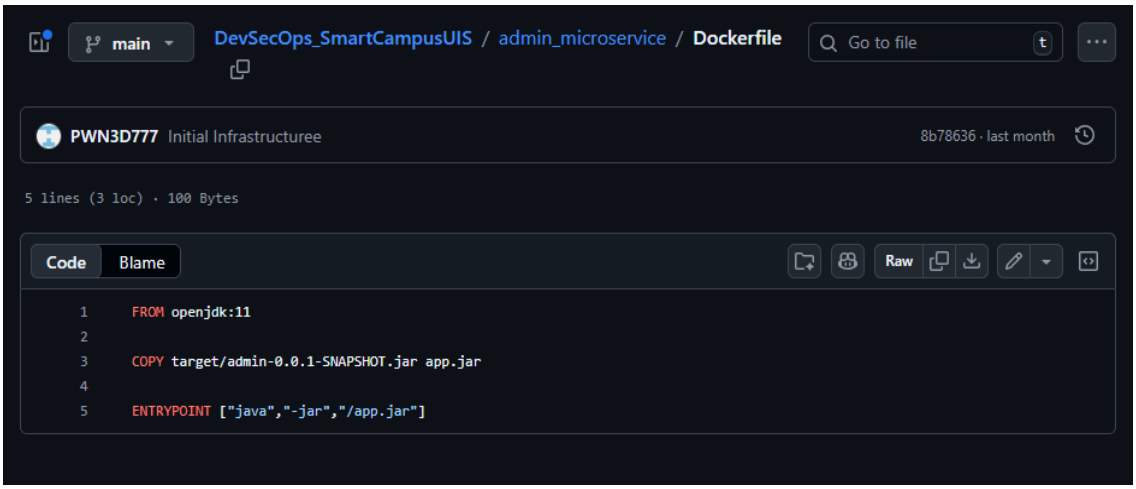
Después:

Figura 3. DESPUÉS - Comparación de la estructura del directorio admin_microservice antes y después de la integración del código fuente.



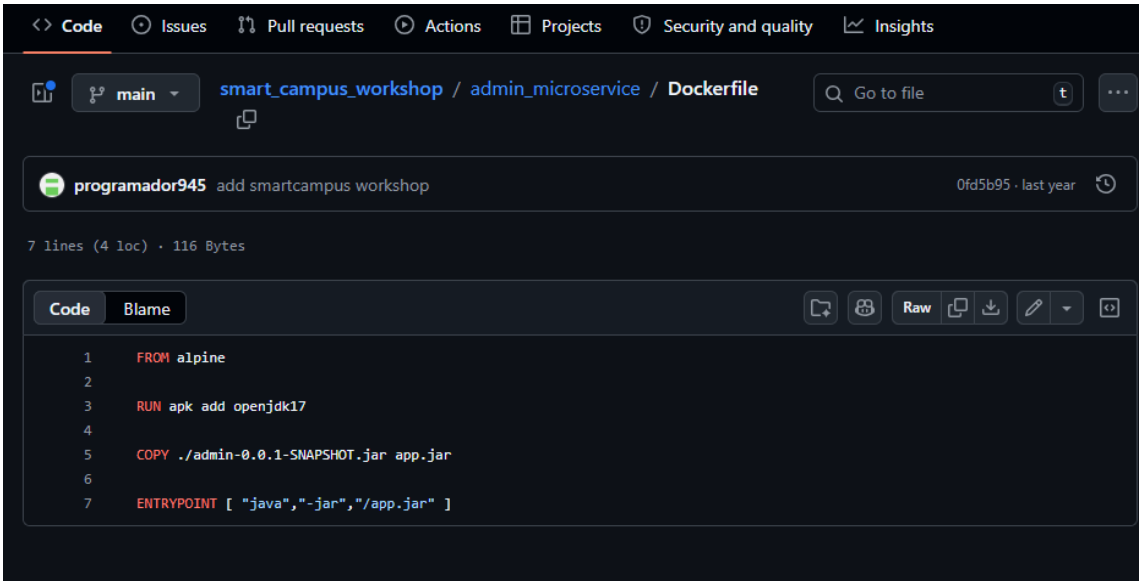
Antes:

Figura 4. ANTES - Comparación del Dockerfile del componente admin_microservice antes y después de su adaptación al flujo CI/CD.



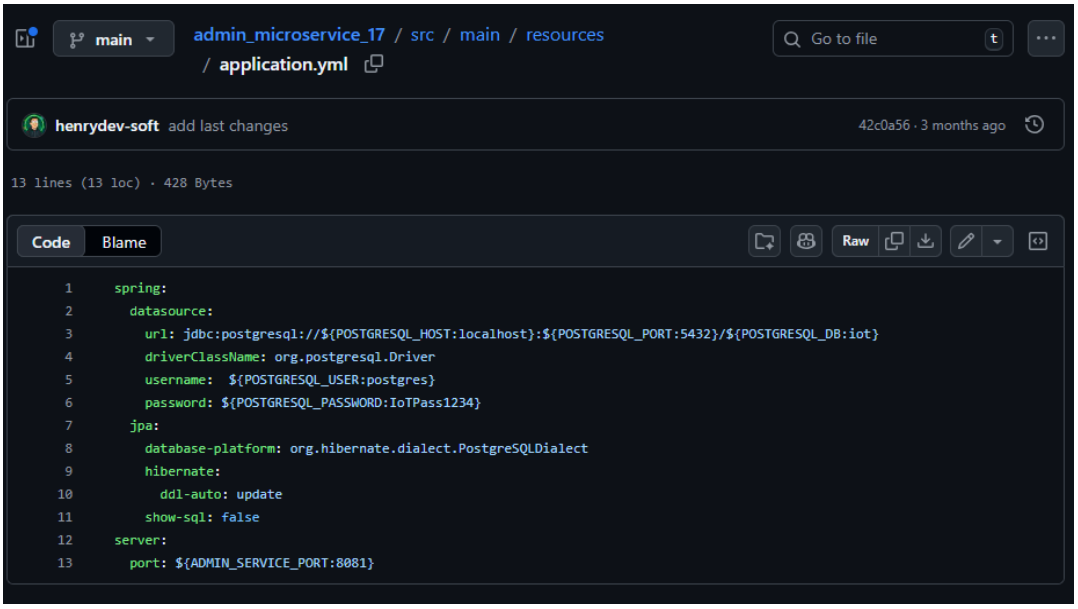
Después:

Figura 5. DESPUÉS - Comparación del Dockerfile del componente admin_microservice antes y después de su adaptación al flujo CI/CD.



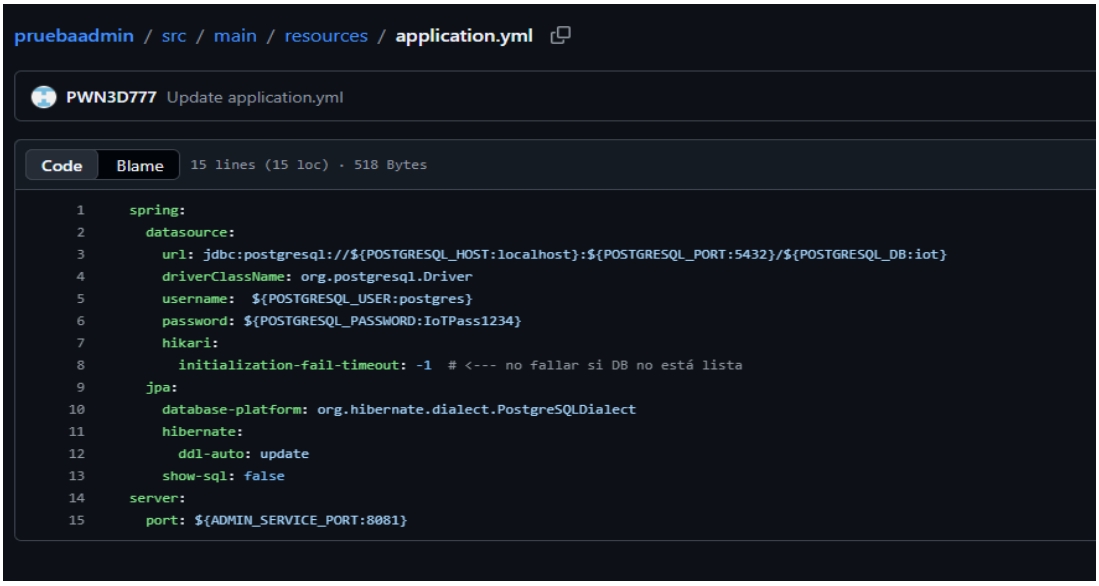
Antes:

Figura 6. ANTES - Comparación del archivo application.yml del componente admin_microservice antes y después de su ajuste para pruebas y despliegue.



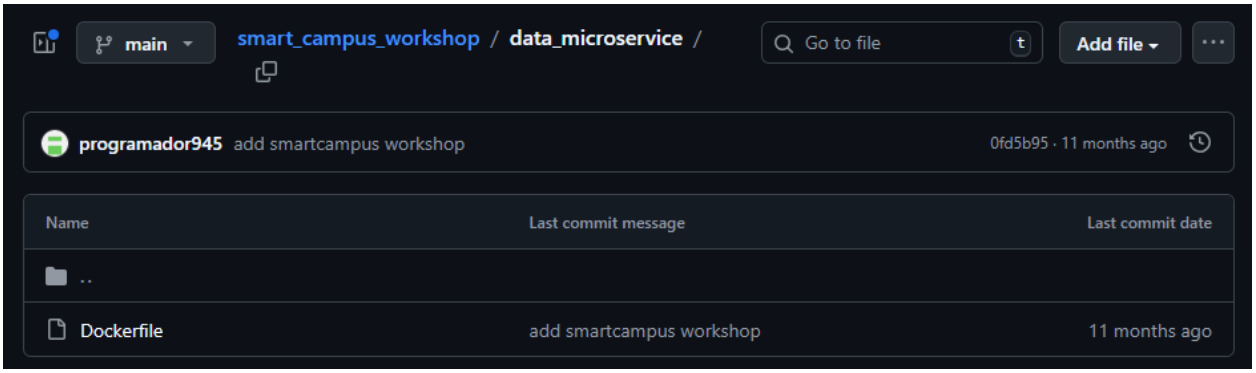
Después:

Figura 7. DESPUÉS - Comparación del archivo application.yml del componente admin_microservice antes y después de su ajuste para pruebas y despliegue.



Antes:

Figura 8. ANTES - Comparación de la estructura del directorio data_microservice antes y después de la integración del código fuente.

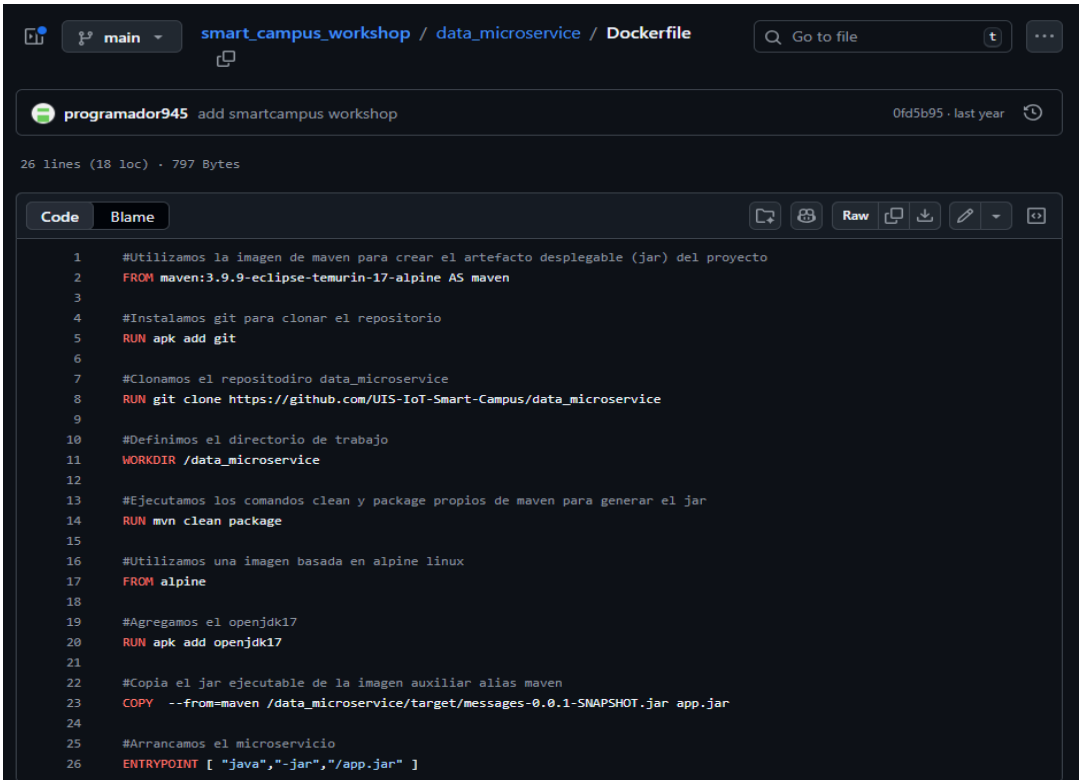


Después:

Figura 9. DESPUÉS - Comparación de la estructura del directorio data_microservice antes y después de la integración del código fuente.

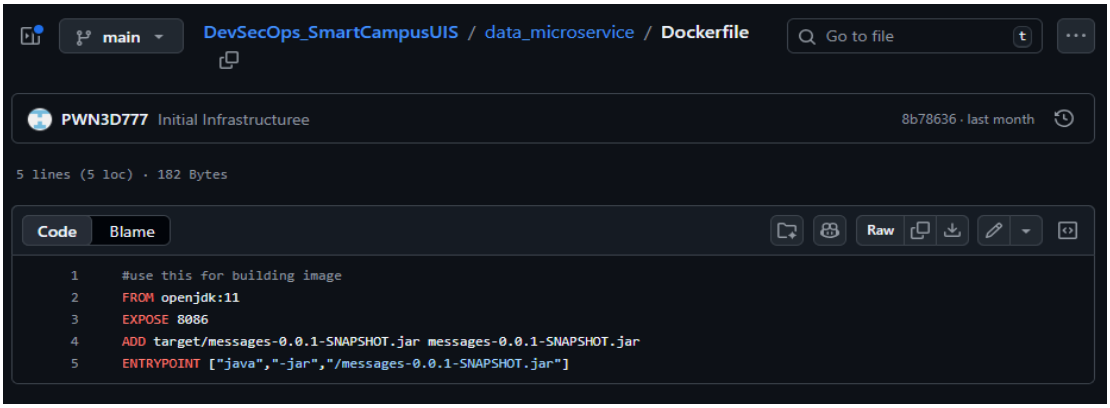
Antes:

Figura 10. ANTES - Comparación del Dockerfile del componente data_microservice antes y después de su adaptación al flujo CI/CD.



Después:

Figura 11. DESPUÉS - Comparación del Dockerfile del componente data_microservice antes y después de su adaptación al flujo CI/CD.

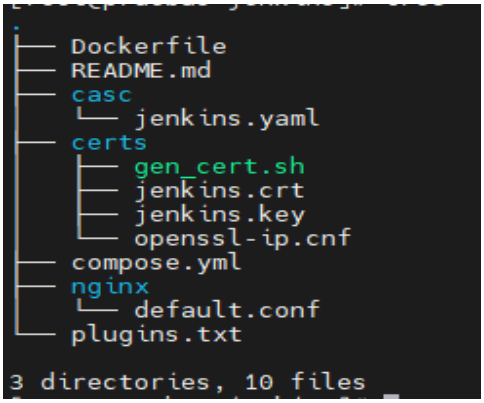


5.7 SERVIDOR DE ORQUESTACIÓN CI/CD CONFIGURADO - P1.2

Con el fin de soportar la automatización de los procesos de integración continua y despliegue continúo definidos para el proyecto, se implementó un servidor de orquestación CI/CD basado en Jenkins. Su despliegue se realizó mediante una arquitectura contenerizada administrada con Docker Compose, lo que permitió definir de forma declarativa los servicios requeridos, facilitar la administración del entorno y mejorar la portabilidad.

La configuración adoptada se basó en dos servicios principales: el controlador Jenkins, encargado de la gestión y ejecución de los pipelines, y un servidor Nginx configurado como proxy inverso para atender el acceso a la interfaz web. Esta separación permite desacoplar la lógica de orquestación del mecanismo de publicación del servicio, evitando la exposición directa del puerto interno de Jenkins y estableciendo una base de despliegue más organizada y controlada. En la Figura 12 se presenta la estructura general del directorio utilizado para el montaje.

Figura 12. Estructura del directorio utilizada para el montaje del servidor Jenkins con Nginx y configuración declarativa.

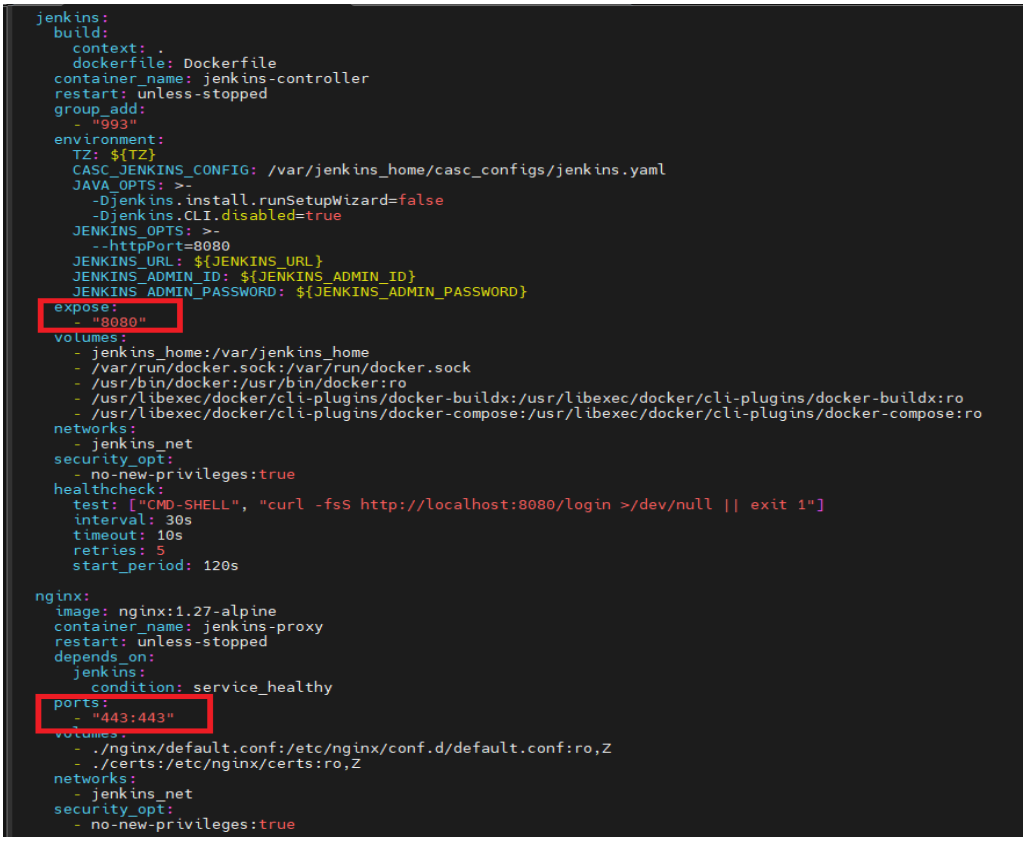


En una primera etapa, se definió la organización principal del despliegue a través de un directorio que reúne los archivos principales de configuración, entre ellos el Dockerfile de Jenkins, el archivo compose.yml, la configuración del proxy inverso Nginx, el archivo declarativo de Jenkins y los certificados requeridos para el acceso HTTPS. Esta

organización facilitó la separación de responsabilidades entre componentes y permitió mantener la configuración del servidor de forma centralizada y trazable.

Posteriormente, en el archivo compose.yml se declararon los dos servicios que componen la solución. El primero corresponde al servicio Jenkins, construido a partir de una imagen personalizada, y el segundo al servicio Nginx, utilizado para intermediar el acceso externo hacia la interfaz web del controlador. Como se observa en la Figura 13, Jenkins se configuró para operar internamente sobre el puerto 8080 dentro de la red del despliegue, mientras que el acceso publicado hacia los usuarios se canalizó a través del proxy inverso. Esta disposición permitió conservar una arquitectura más próxima a un entorno controlado, en el que el acceso a la interfaz del servidor se administra desde una capa intermedia.

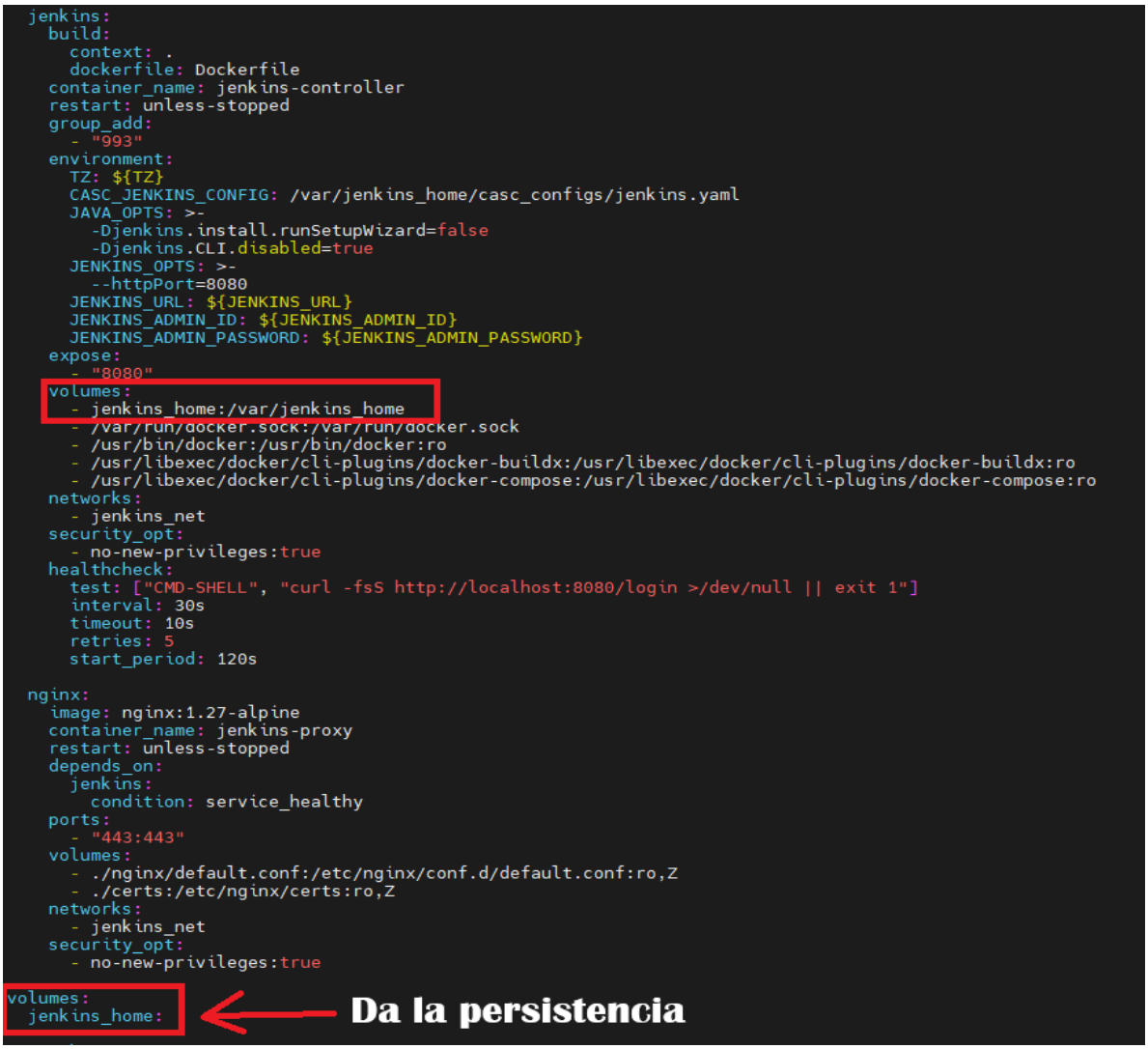
Figura 13. Fragmento del archivo compose.yml con la definición de los servicios Jenkins y Nginx.



```
jenkins:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: jenkins-controller
  restart: unless-stopped
  group_add:
    - "993"
  environment:
    TZ: ${TZ}
    CASC_JENKINS_CONFIG: /var/jenkins_home/casc_configs/jenkins.yaml
    JAVA_OPTS: >-
      -Djenkins.install.runSetupWizard=false
      -Djenkins.CLI.disabled=true
    JENKINS_OPTS: >-
      --httpPort=8080
    JENKINS_URL: ${JENKINS_URL}
    JENKINS_ADMIN_ID: ${JENKINS_ADMIN_ID}
    JENKINS_ADMIN_PASSWORD: ${JENKINS_ADMIN_PASSWORD}
  expose:
    - "8080"
  volumes:
    - jenkins_home:/var/jenkins_home
    - /var/run/docker.sock:/var/run/docker.sock
    - /usr/bin/docker:/usr/bin/docker:ro
    - /usr/libexec/docker/cli-plugins/docker-buildx:/usr/libexec/docker/cli-plugins/docker-buildx:ro
    - /usr/libexec/docker/cli-plugins/docker-compose:/usr/libexec/docker/cli-plugins/docker-compose:ro
  networks:
    - jenkins_net
  security_opt:
    - no-new-privileges:true
  healthcheck:
    test: ["CMD-SHELL", "curl -fsS http://localhost:8080/login >/dev/null || exit 1"]
    interval: 30s
    timeout: 10s
    retries: 5
    start_period: 120s

nginx:
  image: nginx:1.27-alpine
  container_name: jenkins-proxy
  restart: unless-stopped
  depends_on:
    jenkins:
      condition: service_healthy
  ports:
    - "443:443"
  volumes:
    - ./nginx/default.conf:/etc/nginx/conf.d/default.conf:ro,Z
    - ./certs:/etc/nginx/certs:ro,Z
  networks:
    - jenkins_net
  security_opt:
    - no-new-privileges:true
```

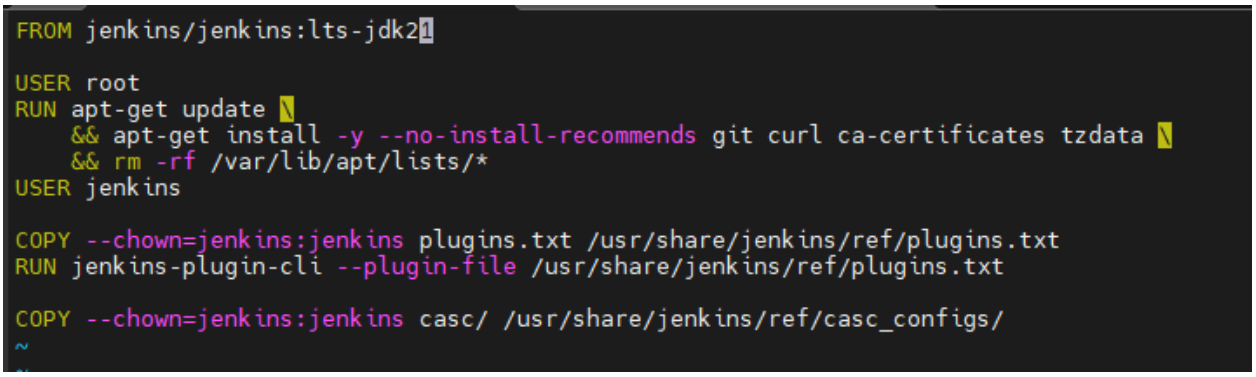
Adicionalmente, para garantizar la persistencia de la información operativa del servidor, se definió un volumen nombrado en la sección global del archivo Compose y posteriormente se asoció al directorio “/var/jenkins_home” del servicio Jenkins. En este directorio la plataforma almacena su configuración interna, usuarios, credenciales, jobs, plugins instalados, historial de ejecuciones y demás archivos requeridos para su funcionamiento. Por ello, la utilización del volumen persistente permitió conservar el estado del servidor incluso en escenarios de reinicio o recreación del contenedor. El fragmento correspondiente a esta configuración se muestra en la Figura 14.

Figura 14. Configuración del volumen persistente asociado a /var/jenkins_home en el despliegue de Jenkins.



Como parte de la preparación del entorno, también se construyó una imagen personalizada de Jenkins a partir de la imagen oficial, la cual se muestra en la figura Figura 15. Esta personalización permitió incorporar de forma inicial algunos componentes necesarios para la ejecución de pipelines, la integración con repositorios Git y la administración base del servidor. Entre estos elementos se incluyeron complementos orientados a la definición de pipelines, la conexión con plataformas de control de versiones y la gestión declarativa de la configuración. Y así, el despliegue no depende exclusivamente de ajustes manuales posteriores a la instalación, sino que partió de una base reproducible y alineada con los requerimientos técnicos del proyecto.

Figura 15. Dockerfile de la imagen personalizada de Jenkins con componentes base para la ejecución del pipeline.



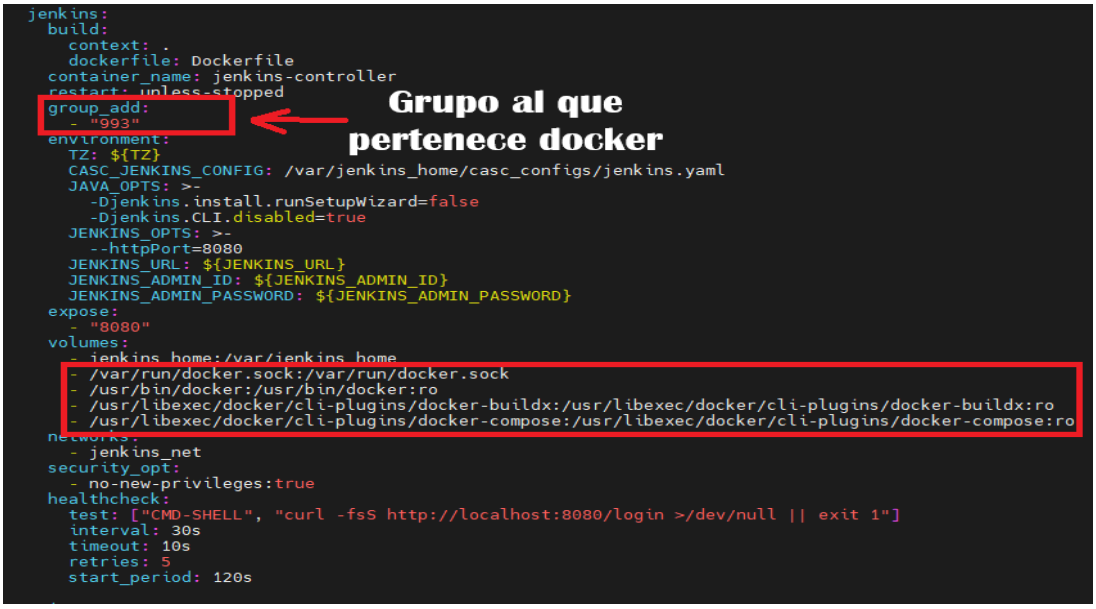
En cuanto a la publicación del servicio, Nginx se configuró como proxy inverso para recibir las conexiones HTTPS y reenviarlas al controlador Jenkins dentro de la red interna del despliegue. Esta decisión permitió centralizar la gestión del acceso web y separar la capa de publicación de la capa de orquestación. Así mismo, el uso de certificados para el servicio web proporciona un mecanismo de cifrado para la comunicación con la interfaz de administración. El fragmento principal de la configuración del proxy puede observarse en la Figura 16.

Figura 16. Configuración del proxy inverso Nginx para la publicación segura del servicio Jenkins.

```
upstream jenkins_upstream {
    keepalive 32;
    server jenkins:8080;
}

map $http_upgrade $connection_upgrade {
    default upgrade;
    '' close;
}

server {
    listen 443 ssl http2;
    server_name 192.168.1.142;

    ssl_certificate     /etc/nginx/certs/jenkins.crt;
    ssl_certificate_key /etc/nginx/certs/jenkins.key;

    ssl_session_timeout 1d;
    ssl_session_cache shared:SSL:10m;
    ssl_protocols TLSv1.2 TLSv1.3;

    ignore_invalid_headers off;

    location / {
        allow 192.168.1.0/24;
        allow 10.8.0.0/24;
        deny all;
```

Debe mencionarse, además, que durante esta fase del proyecto el controlador Jenkins conservó acceso al motor Docker del host con el fin de soportar la ejecución de tareas requeridas por los pipelines como se muestra en la Figura 17. Aunque esta decisión facilitó la integración inicial y la operación del entorno, también constituye un aspecto relevante desde el punto de vista del endurecimiento de la infraestructura, por lo que se documentó como una consideración técnica del montaje adoptado.

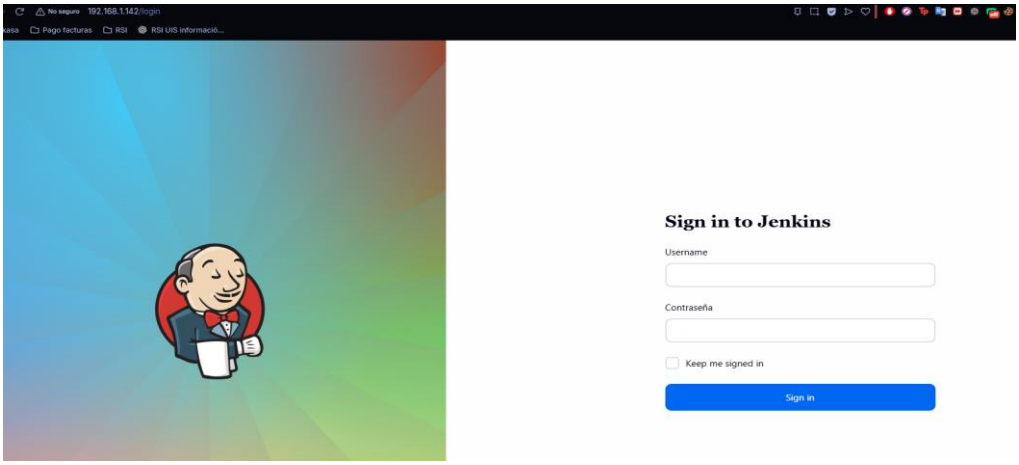
Figura 17. Acceso del controlador Jenkins al socket de Docker del host como consideración técnica del montaje.




```
jenkins:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: jenkins-controller
  restart: unless-stopped
  group_add:
    - "993"
  environment:
    TZ: ${TZ}
    CASC_JENKINS_CONFIG: /var/jenkins_home/casc_configs/jenkins.yaml
    JAVA_OPTS: >-
      -Djenkins.install.runSetupWizard=false
      -Djenkins.CLI.disabled=true
    JENKINS_OPTS: >-
      --httpPort=8080
    JENKINS_URL: ${JENKINS_URL}
    JENKINS_ADMIN_ID: ${JENKINS_ADMIN_ID}
    JENKINS_ADMIN_PASSWORD: ${JENKINS_ADMIN_PASSWORD}
  expose:
    - "8080"
  volumes:
    - jenkins_home:/var/jenkins_home
    - /var/run/docker.sock:/var/run/docker.sock
    - /usr/bin/docker:/usr/bin/docker:ro
    - /usr/libexec/docker/cli-plugins/docker-buildx:/usr/libexec/docker/cli-plugins/docker-buildx:ro
    - /usr/libexec/docker/cli-plugins/docker-compose:/usr/libexec/docker/cli-plugins/docker-compose:ro
  networks:
    - jenkins_net
  security_opt:
    - no-new-privileges:true
  healthcheck:
    test: ["CMD-SHELL", "curl -fsS http://localhost:8080/login >/dev/null || exit 1"]
    interval: 30s
    timeout: 10s
    retries: 5
    start_period: 120s
```

En conjunto, esta configuración permite disponer de un servidor Jenkins funcional, persistente y organizado en componentes diferenciados, proporcionando una base adecuada para la implementación de los pipelines CI/CD definidos en el proyecto y para la posterior incorporación de mecanismos adicionales de seguridad y control operacional. En la Figura 18 se muestra Jenkins ya en funcionamiento.

Figura 18. Interfaz web de Jenkins en funcionamiento tras la configuración inicial del entorno.



5.8 PIPELINE BÁSICO FUNCIONAL CON COMPILACIÓN AUTOMATIZADA - P1.3

Como parte de la implementación del proceso de integración continua, se configuró un pipeline básico funcional en el servicio de orquestación Jenkins. Este pipeline inicial se caracteriza por ejecutar una validación en la integración del código fuente almacenado en el repositorio del proyecto. Este proceso constituye la base sobre la cual se pueden integrar posteriormente etapas adicionales como pruebas automatizadas, análisis de seguridad y despliegue continuo.

La primera etapa del pipeline consiste en la recuperación del código fuente desde el repositorio remoto. Para ello se utilizó el mecanismo de checkout, que permite clonar el repositorio y obtener la versión más reciente del proyecto.

Posteriormente se ejecuta una fase de validación en la cual se ejecutan comandos que permiten verificar que el código recuperado del repositorio, específicamente el de data_microservice y admin_microservice se encuentra disponible y puede ser procesado correctamente. Para poder realizar esto es necesario inicialmente instalar los plugins necesarios de compilación básica tanto para Maven como para java.

Figura 19. Creación de una nueva tarea tipo Pipeline en Jenkins.

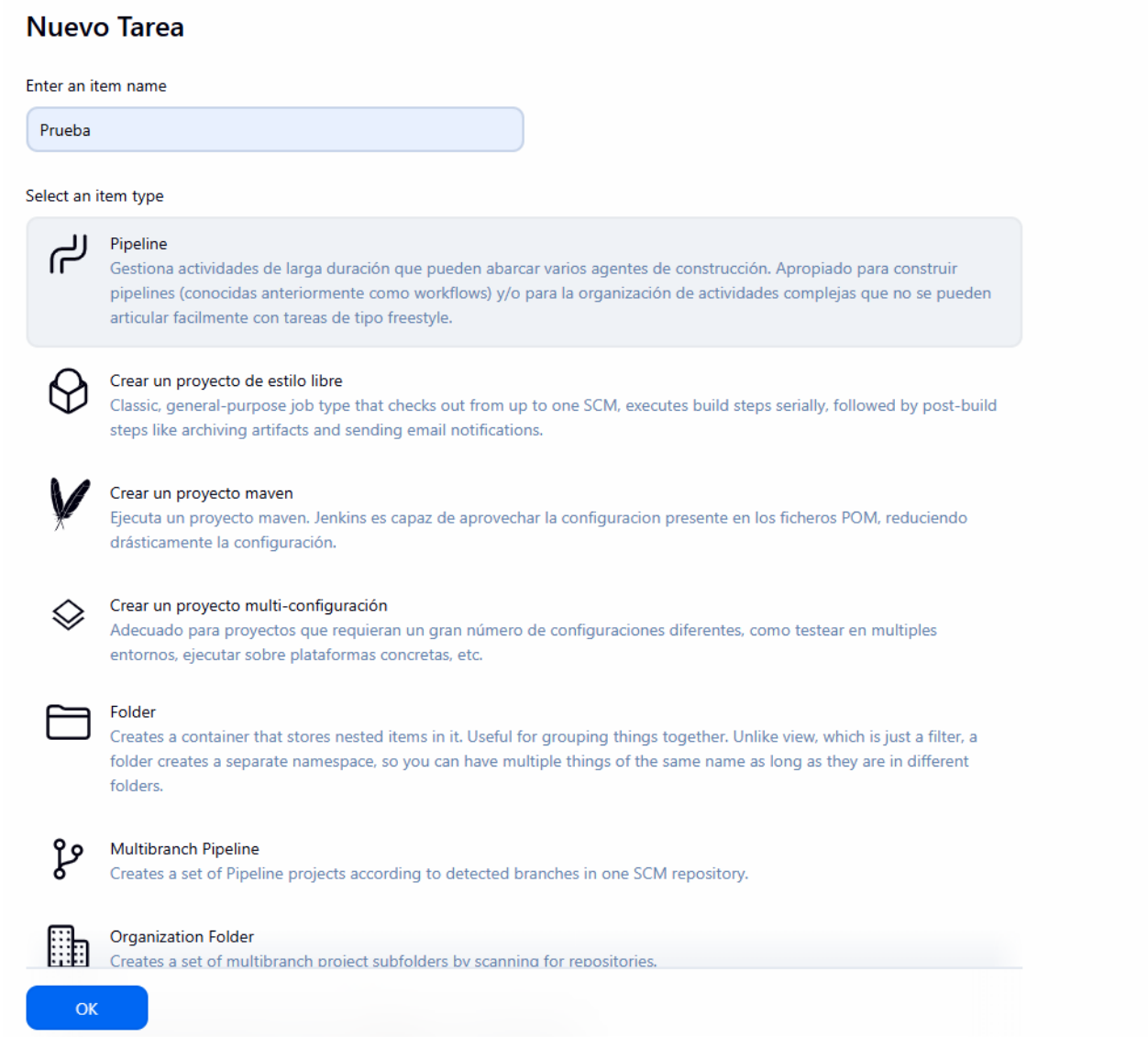


Figura 20. PASO 1 - Acceso al gestor de plugins de Jenkins para la instalación de componentes requeridos.

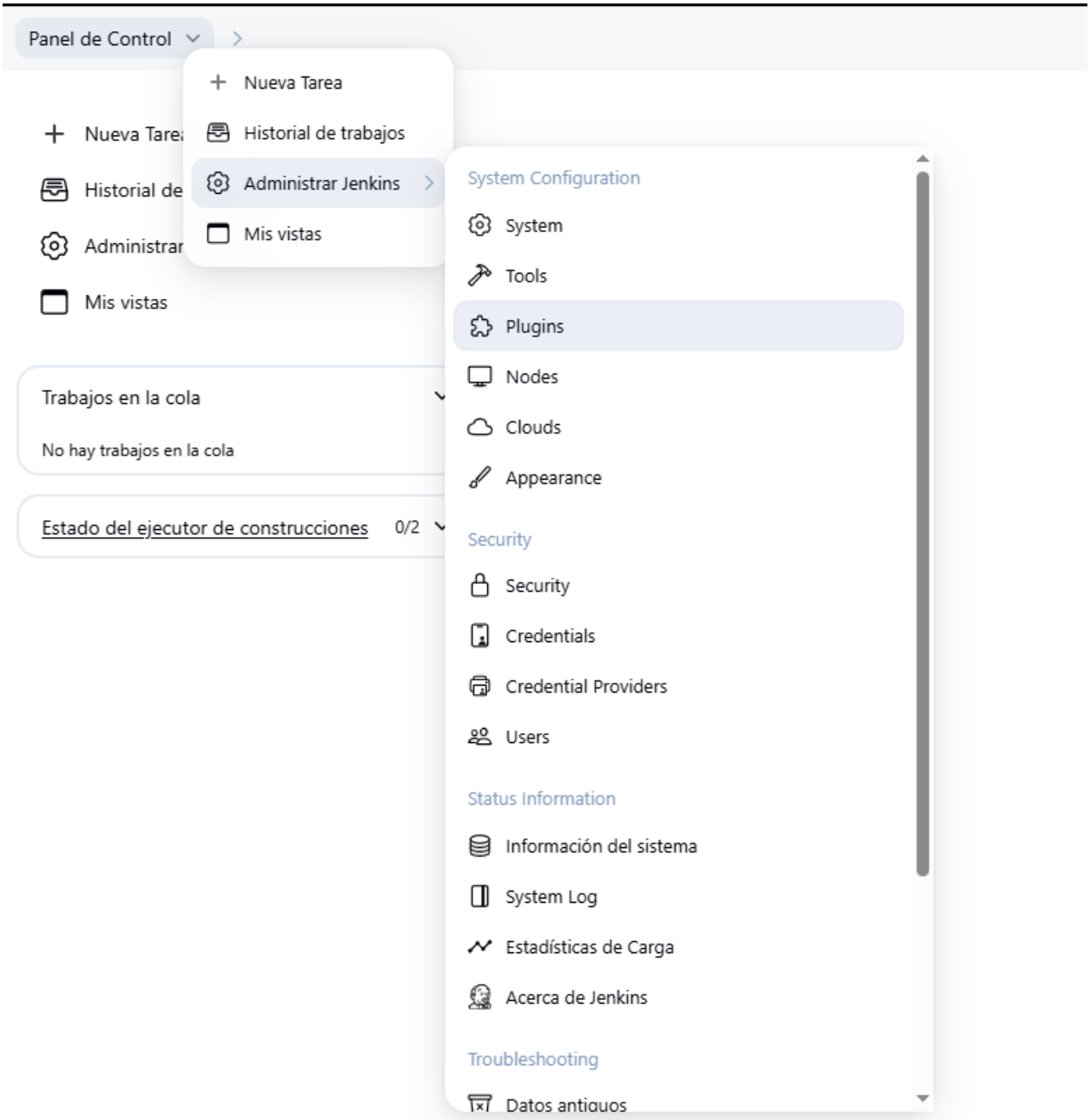


Figura 21. PASO 2 - Acceso al gestor de plugins de Jenkins para la instalación de componentes requeridos.

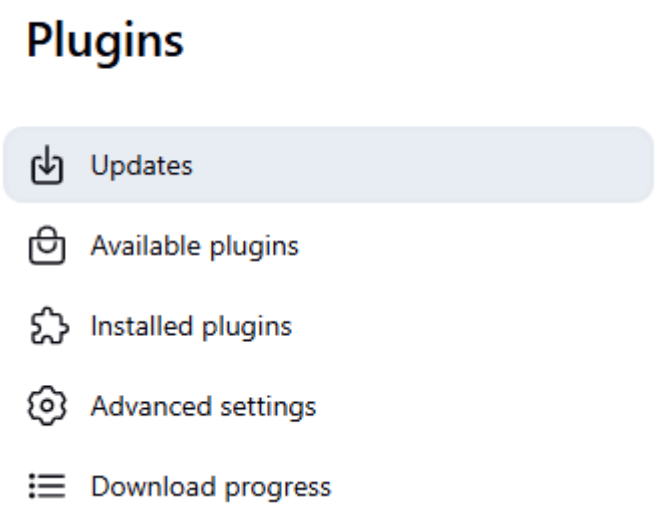


Figura 22. JDK - Instalación de plugins necesarios para la compilación y ejecución del pipeline.

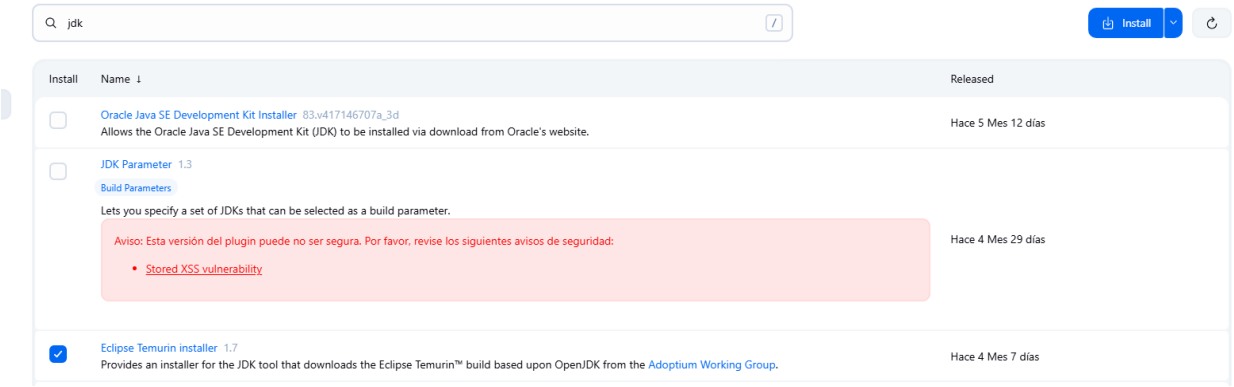


Figura 23. SONAR - Instalación de plugins necesarios para la compilación y ejecución del pipeline.

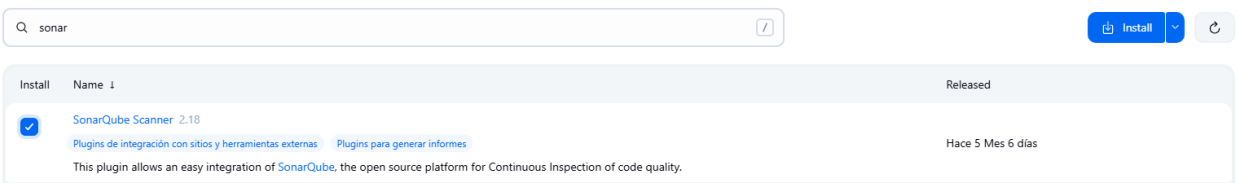


Figura 24. PLUGINS INSTALADOS - Instalación de plugins necesarios para la compilación y ejecución del pipeline.

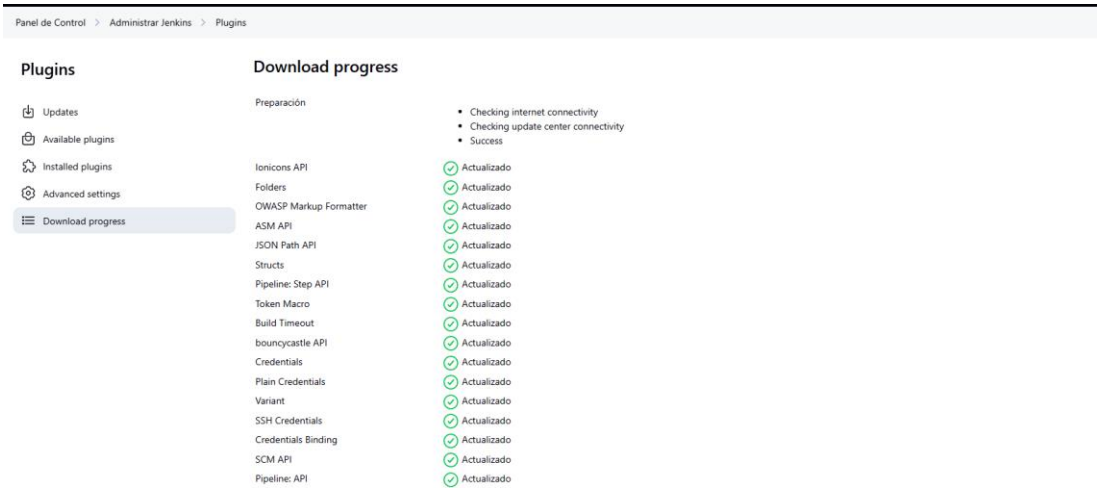


Figura 25. MAVEN - Configuración global de herramientas de construcción en Jenkins.

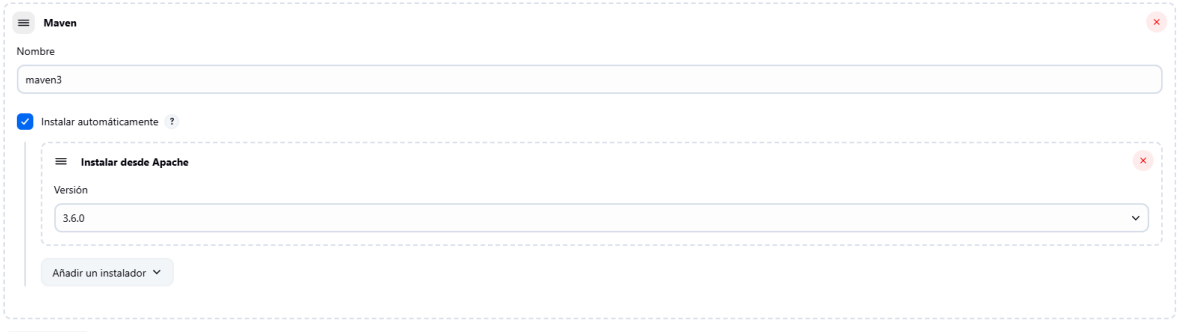


Figura 26. JDK - Configuración global de herramientas de construcción en Jenkins.

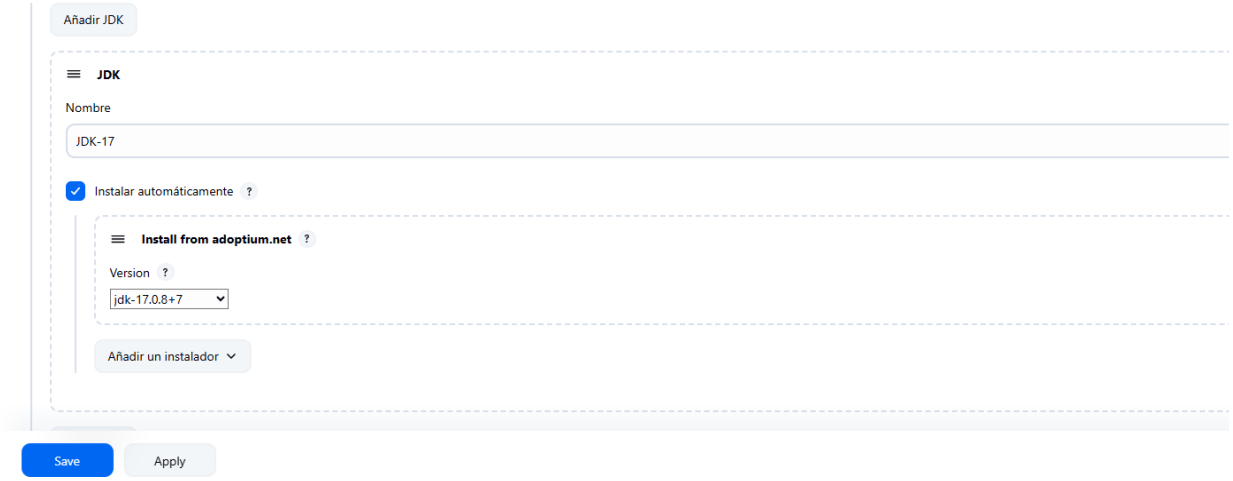


Figura 27. Script del pipeline básico funcional con etapas de checkout y compilación.

Pipeline script

Script ?

try sample Pipeline...

```
pipeline {
    agent any
    tools {
        jdk 'JDK-17'
        maven 'maven3'
    }
    stages {
        stage('Git checkout') {
            steps {
                git branch: 'main', url: 'https://github.com/PWN3D777/DevSecOps_SmartCampusUIS/'
            }
        }
        stage('Code compile' ){
            steps {
                dir('admin_microservice') {
                    sh "mvn clean compile"
                }
                dir('data_microservice') {
                    sh "mvn clean compile"
                }
            }
        }
    }
}
```

Figura 28. PARTE 1 - Ejecución exitosa del pipeline básico funcional en Jenkins.

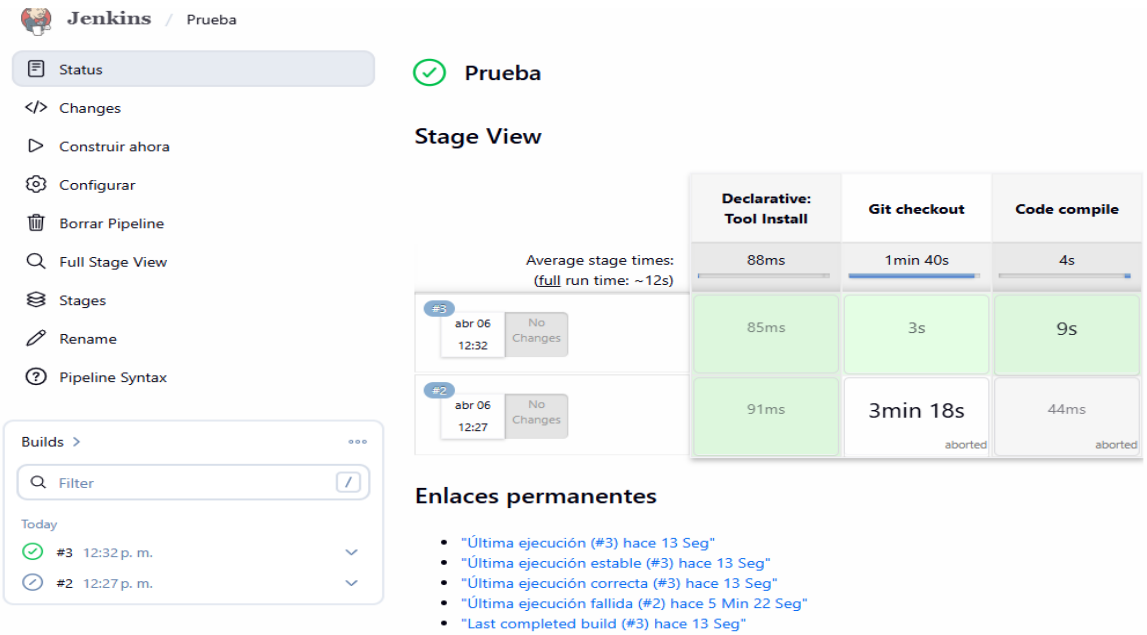
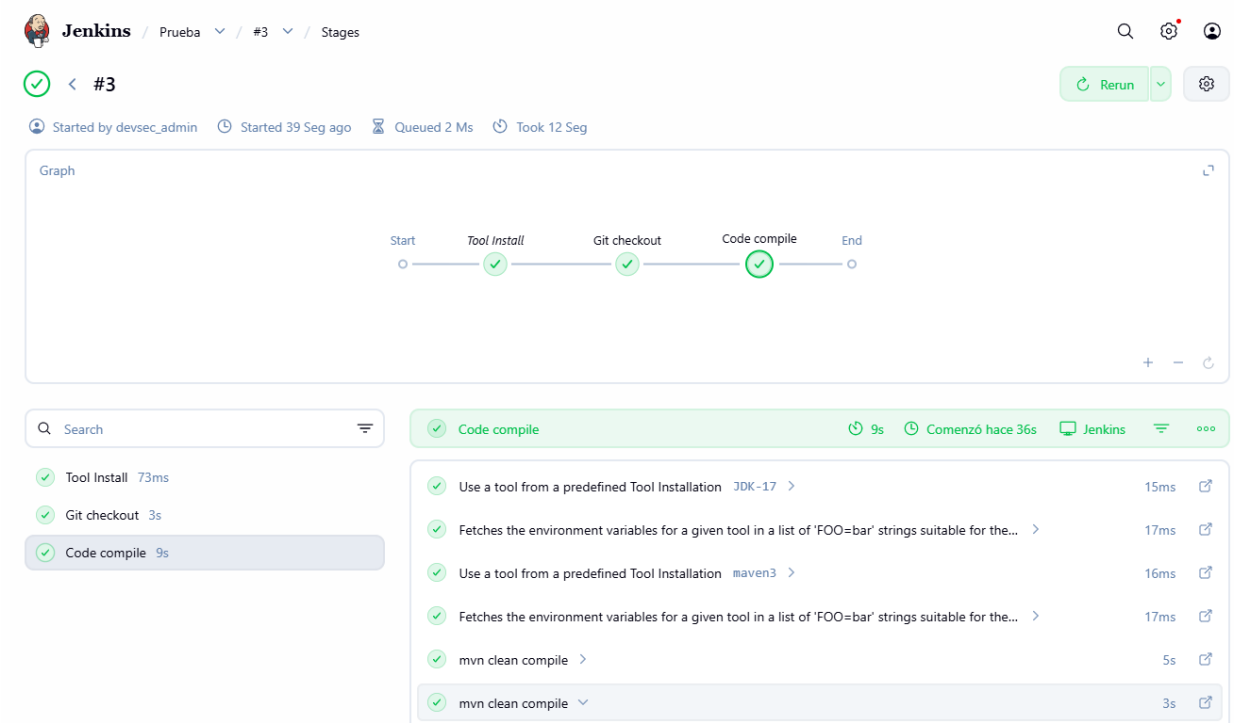


Figura 29. PARTE 2 - Ejecución exitosa del pipeline básico funcional en Jenkins.



6. FASES 2 Y 3: LINEAMIENTOS DE SEGURIDAD Y DISEÑO SHIFT LEFT SECURITY

6.1 DOCUMENTACIÓN DE REQUERIMIENTOS FUNCIONALES Y DE SEGURIDAD - P2.1

Con el propósito de orientar el diseño e implementación del flujo CI/CD con enfoque DevSecOps para la plataforma Smart Campus UIS, en esta parte se establecen los requerimientos funcionales y de seguridad que deben satisfacer la solución propuesta. Definir estos requerimientos hace parte de la base fundamental para la selección de herramientas, la estructuración de la arquitectura del pipeline y la posterior validación de su funcionamiento dentro del entorno del proyecto.

La identificación de requerimientos permite delimitar el alcance técnico del pipeline, concretar sus capacidades mínimas a esperar y definir los controles necesarios para incorporar prácticas de seguridad desde etapas tempranas del ciclo de vida del software. En este sentido, no se hacen únicamente como condiciones operativas del sistema, sino también como lineamientos que materializan el enfoque DevSecOps planteado en el proyecto.

Los requerimientos funcionales describen las capacidades que debe ofrecer el pipeline para cubrir y soportar la automatización del proceso de integración, validación y construcción del software. En este contexto, el sistema debe permitir la ejecución automatizada de las etapas definidas dentro del flujo, especialmente aquellas relacionadas con compilación, validaciones iniciales, análisis de seguridad y generación de artefactos. Así mismo, el pipeline debe trabajar sobre una versión identificable del proyecto, de manera que cada ejecución pueda asociarse con el estado correspondiente del software analizado (Por ejemplo, el número de commit, la rama y el número de build de jenkins).

De igual forma, el flujo automatizado debe incorporar etapas destinadas al análisis del software desde diferentes perspectivas. Esto implica la ejecución del análisis estático del

código fuente con el fin de detectar defectos, debilidades o problemas de mantenibilidad, así como el análisis de dependencias para identificar componentes con vulnerabilidades conocidas. Adicionalmente, cuando se requiera, el pipeline debe permitir la generación de artefactos o imágenes de contenedor y su correspondiente evaluación de seguridad antes de ser utilizados en etapas posteriores del proceso. En conjunto, estas capacidades buscan consolidar una cadena de validación continua que integre calidad, seguridad y automatización.

Otro requerimiento funcional relevante corresponde a la generación de evidencias del proceso. El pipeline debe producir registros, reportes y resultados consultables, que permitan conocer el estado de cada ejecución, los hallazgos detectados por las herramientas integradas y la relación entre dichos resultados y la versión específica del software analizado. Esta capacidad resulta esencial tanto para la trazabilidad técnica como para la documentación del proyecto, dado que facilita la verificación de los controles aplicados y la posterior interpretación de los resultados obtenidos.

En cuanto a los requerimientos de seguridad, se establece que toda ejecución del pipeline debe someterse a validaciones automáticas antes de considerarse apta para las siguientes etapas donde se hace la construcción, el empaquetado y el despliegue. Bajo esta lógica, la seguridad se integra como un criterio que se encuentra en todo el flujo y no como una actividad aislada al final del proceso. Esto responde al principio de Shift Left Security, según el cual la detección temprana de fallos y vulnerabilidades permite reducir el costo de corrección, fortalecer la calidad del software y mejorar el control, orden, trazabilidad y reglas del ciclo de desarrollo.

En concordancia con lo anterior, el pipeline debe incorporar mecanismos que permitan identificar vulnerabilidades en tres niveles principales: código fuente, dependencias externas y artefactos o imágenes generadas durante la construcción. Del mismo modo, debe establecer criterios mínimos de aceptación frente a los hallazgos detectados, de tal manera que sea posible determinar cuándo una ejecución cumple las condiciones necesarias para continuar y cuándo debe considerarse fallida. Estos criterios constituyen

un elemento central del control de calidad y seguridad, ya que permiten traducir los resultados de las herramientas en decisiones del pipeline.

También se considera necesario que las herramientas empleadas para la detección de vulnerabilidades operen con información actualizada, particularmente en los casos en que dependan de bases de datos de vulnerabilidades o listas de referencias externas. Esto garantiza una evaluación más confiable del software analizado y evita que el pipeline genere resultados incompletos o desactualizados. De manera adicional, los registros y reportes producidos no deben exponer credenciales, secretos u otra información sensible del entorno, preservando así la seguridad operativa de la infraestructura utilizada.

Tomando en cuenta lo anterior, los requerimientos definidos en esta fase se convierten en un aspecto importante que afecta directamente las etapas posteriores del proyecto.
En particular, constituyen la base para la definición de las herramientas SAST y SCA, el diseño de la arquitectura técnica del pipeline DevSecOps y la elaboración del documento de diseño detallado. Por tanto, su formulación no solo cumple una función descriptiva, sino que establece el marco de referencia técnico sobre el cual se soportará la implementación de la solución propuesta para Smart Campus UIS.

6.1.1 Requerimientos funcionales (RF). Los requerimientos funcionales definidos para el pipeline son los siguientes:

RF-01. El pipeline debe permitir la ejecución automatizada de las etapas de compilación definidas para el proyecto, empleando la herramienta de construcción correspondiente.
RF-02. El sistema debe trabajar sobre una versión identificable del software, de manera que cada ejecución pueda asociarse con el estado correspondiente del proyecto analizado.
RF-03. El pipeline debe ejecutar validaciones iniciales relacionadas al proceso de construcción, con el fin de verificar la consistencia básica del proyecto antes de las etapas de análisis de seguridad.
RF-04. El sistema debe integrar análisis estático del código fuente para identificar defectos, vulnerabilidades potenciales y problemas de calidad del software.

RF-05. El pipeline debe incorporar análisis de composición de software sobre dependencias y librerías utilizadas por el proyecto, con el fin de detectar componentes con vulnerabilidades conocidas.
RF-06. El sistema debe permitir la construcción de artefactos o imágenes generadas, cuando la naturaleza del servicio o del flujo definido así lo requiera.
RF-07. El pipeline debe ejecutar validaciones de seguridad sobre los artefactos o imágenes generadas, con el fin de identificar vulnerabilidades antes de su uso en las siguientes etapas.
RF-08. El sistema debe generar evidencias de ejecución, incluyendo logs, resultados y reportes de las herramientas integradas en el flujo.
RF-09. El pipeline debe permitir establecer condiciones de continuidad, aprobación o fallo con base en los resultados obtenidos durante las validaciones efectuadas en las etapas.
RF-10. El sistema debe mantener trazabilidad entre la ejecución realizada, los resultados obtenidos y la versión de software analizada.

6.1.2. Requerimientos de seguridad (RS). Los requerimientos de seguridad definidos para el pipeline son los siguientes:

RS-01. Toda ejecución del pipeline debe incluir validaciones automáticas de seguridad como parte del flujo de integración y construcción definido.
RS-02. El sistema debe permitir la detección temprana de vulnerabilidades en el código fuente, en las dependencias y en los artefactos o imágenes generados.
RS-03. Los artefactos e imágenes generados por el pipeline deben conservar integridad y trazabilidad respecto a la ejecución que los produjo.
RS-04. El sistema debe conservar registros y resultados que permitan auditar los hallazgos de seguridad detectados durante la ejecución del pipeline.
RS-05. El pipeline debe definir umbrales o criterios mínimos de aceptación frente a hallazgos de seguridad, de manera que sea posible determinar cuándo una ejecución debe considerarse satisfactoria o fallida.
RS-06. Las herramientas de análisis de seguridad utilizadas en el pipeline deben operar con información actualizada, especialmente cuando dependan de bases de datos de vulnerabilidades.

RS-07. Los logs, reportes y salidas generadas por el pipeline no deben exponer información sensible del entorno, como credenciales, secretos o configuraciones críticas.
RS-08. La solución debe facilitar la revisión posterior de resultados, configuraciones y decisiones tomadas durante la ejecución del pipeline, con fines de control técnico y auditoría.

6.2 DEFINICIÓN BASE DE HERRAMIENTAS SAST/SCA Y DE ESCANEO DE IMÁGENES ESTABLECIDAS - P2.2

Con la finalidad de dar cumplimiento a los lineamientos de calidad y seguridad definidos para el proyecto, en esta fase se establece la línea base de herramientas que hará los procesos de análisis estático de código fuente y análisis de composición de software dentro del pipeline DevSecOps propuesto para Smart Campus UIS. Esta definición constituye algo fundamental para las fases posteriores de diseño e implementación, en la medida en que permite delimitar los componentes tecnológicos que serán integrados al flujo de construcción, validación y control de seguridad.

Para el análisis estático del código fuente se establece SonarQube Community Build como herramienta base de SAST. De acuerdo con su documentación oficial, esta solución permite realizar revisión y análisis automatizado del código, integrándose con el pipeline CI/CD, y en su alcance base permite analizar la rama principal del proyecto.

La selección de SonarQube Community Build resulta adecuada para el presente trabajo porque permite incorporar controles de calidad y seguridad directamente sobre el código fuente, generando hallazgos utilizables dentro del pipeline. Así mismo, dispone de quality gates, cuyo resultado puede utilizarse como criterio para determinar si una ejecución debe continuar o considerarse fallida. En integración con Jenkins, la documentación oficial indica que es posible detener el pipeline mientras se espera el resultado del quality gate, siempre que se configure el webhook correspondiente entre SonarQube y Jenkins.

Para el análisis de composición de software se establece OWASP dependency check como herramienta base de SCA. Según su propósito dentro del pipeline, esta herramienta

permite identificar vulnerabilidades conocidas en dependencias, librerías y componentes de terceros presentes en el proyecto. En el caso de aplicaciones Java basadas en Maven, su utilización resulta especialmente coherente, dado que puede analizar archivos como pom.xml y otros artefactos asociados al ecosistema de construcción, permitiendo detectar riesgos derivados del uso de bibliotecas externas.

Por otra parte, para el escaneo de imágenes de contenedor se establece Trivy como herramienta base de análisis de imágenes. Su función principal será identificar vulnerabilidades conocidas presentes en los paquetes, bibliotecas y componentes incluidos dentro de las imágenes Docker generadas durante el proceso de construcción. Su incorporación resulta pertinente dentro del pipeline DevSecOps, ya que permite evaluar la seguridad del artefacto desplegable antes de su publicación o uso en entornos posteriores, fortaleciendo así el control de seguridad sobre la cadena de construcción y despliegue.

A partir de lo mencionado anteriormente, la línea base tecnológica de esta fase queda establecida de la siguiente manera: SonarQube Community Build será utilizado como herramienta de análisis estático de código fuente, OWASP Dependency-Check como herramienta de análisis de composición de software y Trivy como herramienta de escaneo de imágenes de contenedor. Esta distribución permite mantener una separación funcional clara entre el análisis del código, el análisis de dependencias y la validación de artefactos contenerizados, facilitando su integración progresiva dentro del pipeline DevSecOps.

6.2.1 Herramienta sast establecida. Se establece SonarQube Community Build como herramienta base para la ejecución de análisis estático de código fuente dentro del pipeline DevSecOps del proyecto. Su función principal será identificar problemas de calidad, confiabilidad y seguridad sobre el código analizado, aportando resultados que puedan ser utilizados como criterio de control dentro del flujo CI/CD.

6.2.2 Herramienta sca establecida. Se establece OWASP Dependency-Check como herramienta base para la ejecución de análisis de composición de software dentro del pipeline. Su función principal será identificar vulnerabilidades conocidas en dependencias, librerías y componentes de terceros presentes en el proyecto. Para ello, se contempla inicialmente el análisis sobre archivos de definición de dependencias, particularmente pom.xml en aplicaciones Java basadas en Maven.

6.2.3 Herramienta para el escaneo de imágenes establecida. Se establece Trivy como herramienta base para el escaneo de imágenes de contenedor dentro del pipeline DevSecOps del proyecto. Su función principal será identificar vulnerabilidades conocidas presentes en paquetes, bibliotecas y componentes incluidos en las imágenes Docker generadas durante el proceso de construcción. Su incorporación resulta de importancia, ya que permite evaluar la seguridad del artefacto desplegable antes de su publicación o uso en etapas posteriores, complementando así el análisis realizado previamente sobre el código fuente y las dependencias externas.

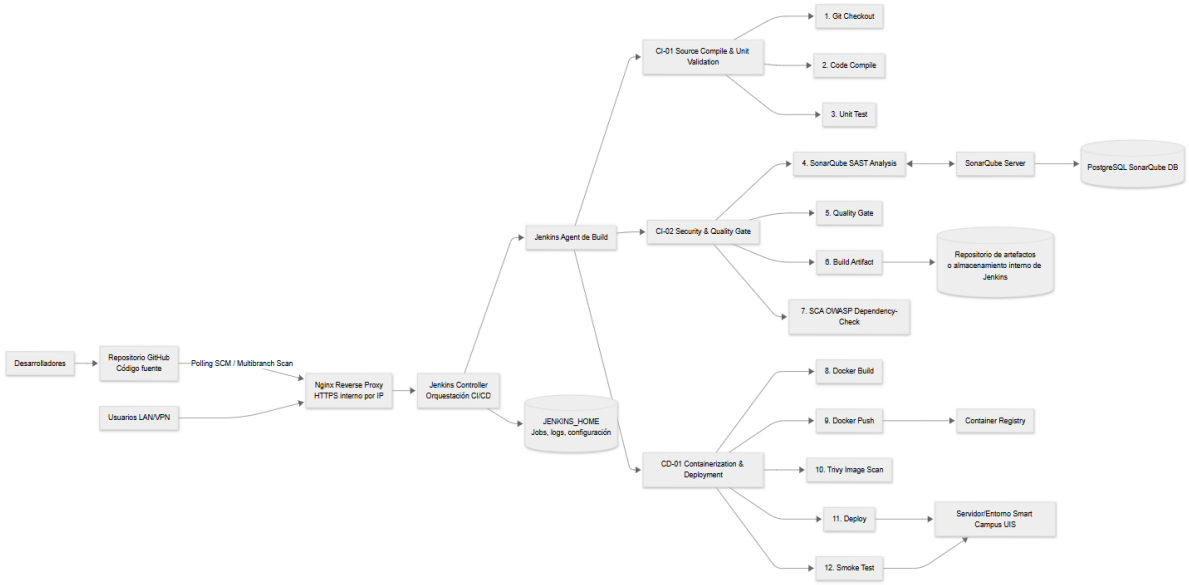
6.2.4 Criterio de adopción de la línea base tecnológica. La adopción de SonarQube Community Build, OWASP Dependency-Check y Trivy como herramientas base responde principalmente a su capacidad de integrarse con flujos CI/CD, a la posibilidad de convertir sus resultados en condiciones de aceptación o rechazo dentro del pipeline y a su utilidad para generar evidencia técnica del proceso. En el caso de SonarQube Community Build, esto se materializa mediante el uso de quality gates integrables con Jenkins; en el caso de OWASP Dependency-Check, mediante la identificación estructurada de vulnerabilidades en dependencias; y en el caso de Trivy, mediante el escaneo de imágenes de contenedor y la generación de reportes reutilizables para análisis posterior y documentación.

6.3 DIAGRAMA DE ARQUITECTURA DEL PIPELINE DEVSECOPS - P3.1

La arquitectura del pipeline DevSecOps propuesto para la plataforma Smart Campus UIS se representa mediante dos diagramas complementarios que permiten visualizar la

estructura lógica del flujo como la relación entre los principales componentes involucrados.

Figura 30. Arquitectura general del pipeline DevSecOps propuesto para Smart Campus UIS.

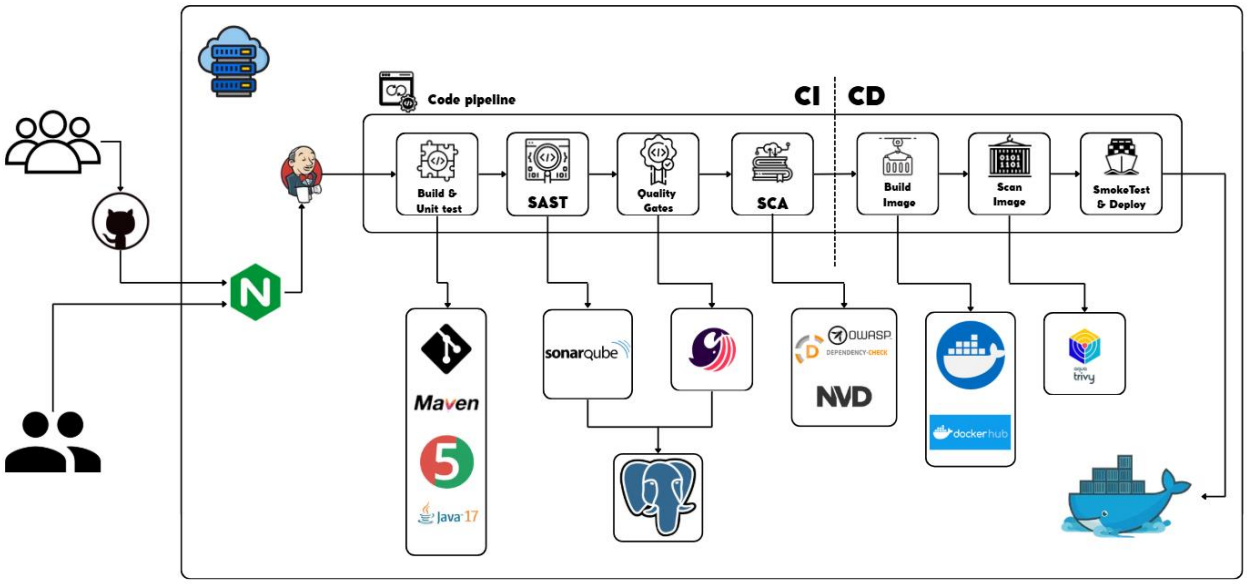


En el primer diagrama se presenta la secuencia general del proceso, iniciando desde la interacción del desarrollador con el repositorio centralizado del proyecto y su posterior integración con el servidor de orquestación Jenkins. A partir de esta base, el flujo se organiza en tres bloques funcionales: CI-01 (Source Compile & Unit Validation), CI-02 (Security & Quality Gate) y CD-01 (Containerization & Deployment). Esta separación permite distinguir de manera clara la validación inicial del código fuente, la incorporación de controles de calidad y seguridad temprana, y la generación, análisis y despliegue progresivo de artefactos contenerizados.

Dentro de esta arquitectura, el bloque CI-01 agrupa las etapas relacionadas con la obtención del código fuente, la compilación y la ejecución de pruebas unitarias. Posteriormente, el bloque CI-02 incorpora mecanismos de análisis estático del código, evaluación mediante *quality gates* y análisis de dependencias externas, permitiendo que

el software solo continúe su recorrido si cumple con los criterios mínimos definidos. Finalmente, el bloque CD-01 contempla la construcción de imágenes Docker, su escaneo de seguridad, la publicación en un registro de contenedores y el despliegue controlado del sistema, seguido de verificaciones operativas básicas.

Figura 31. Relación entre componentes y herramientas de la arquitectura del pipeline DevSecOps propuesto para Smart Campus UIS.



El segundo diagrama complementa esta representación al mostrar la relación entre el flujo del pipeline y las herramientas que soportan cada etapa. En esta vista se destacan Jenkins como núcleo de orquestación, Nginx como proxy inverso para la publicación del servicio, Maven como herramienta de construcción, SonarQube Community Build para el análisis SAST y los quality gates, OWASP Dependency-Check y la NVD para el análisis SCA, Docker para la contenerización y Trivy para el escaneo de imágenes. En conjunto, ambos diagramas sintetizan la arquitectura funcional y tecnológica del pipeline, proporcionando una visión integral del diseño planteado para Smart Campus UIS.

6.4 DISEÑO TÉCNICO DETALLADO - P3.2

Con base en el diagrama de arquitectura del pipeline DevSecOps presentado en la sección anterior, se define el diseño técnico detallado de la solución propuesta para la plataforma Smart Campus UIS. Este diseño tiene como fin describir de manera estructurada los componentes que conforman el flujo CI/CD, sus responsabilidades, las relaciones existentes entre ellos y la forma en que se articulan para soportar la automatización del proceso de construcción, validación, análisis de seguridad, contenerización y despliegue del software.

El diseño técnico propuesto se fundamenta en una arquitectura de orquestación centralizada, en la cual Jenkins constituye el núcleo del pipeline y coordina la ejecución secuencial y controlada de las distintas etapas del proceso. Desde este punto se integran tanto las herramientas de construcción y pruebas como los mecanismos de análisis de seguridad y los componentes requeridos para la generación y validación de artefactos contenerizados. Esta estructura permite consolidar en un único flujo automatizado actividades que anteriormente podían ejecutarse de forma separada o manual, fortaleciendo la trazabilidad, la repetibilidad y el control técnico del ciclo de vida del software.

A nivel de repositorio, el diseño parte de una estructura unificada que incorpora dentro de un mismo entorno de trabajo el código fuente de los microservicios principales de la plataforma, especialmente los componentes admin_microservice y data_microservice, junto con los archivos de soporte requeridos para su construcción, contenerización y despliegue.

Desde la perspectiva de la infraestructura, el servidor de orquestación CI/CD se encuentra desplegado mediante contenedores, haciendo uso de Docker Compose para definir su estructura base. En este entorno, Jenkins se encarga de ejecutar los pipelines, mientras que Nginx cumple la función de proxy inverso para dirigir el acceso hacia la interfaz web del servidor. Esta separación entre la capa de publicación y la capa de orquestación mejora la organización del montaje y proporciona una base más controlada para la administración del servicio. Así mismo, el uso de volúmenes persistentes permite conservar configuraciones, credenciales, historial de ejecuciones y demás datos

operativos del servidor, garantizando continuidad ante reinicios o recreación de contenedores.

De acuerdo con la arquitectura representada, el flujo técnico se organiza en tres bloques funcionales. El primero corresponde a CI-01 (Source Compile & Unit Validation), orientado a la recuperación del código fuente, la compilación de los microservicios, la ejecución de pruebas unitarias y la generación de artefactos iniciales. Este bloque constituye la primera capa de validación del pipeline, ya que verifica que el software pueda construirse correctamente y que sus componentes principales superen una validación funcional mínima antes de avanzar a controles posteriores.

El segundo bloque corresponde a CI-02 (Security & Quality Gate), donde se integran los mecanismos de control de calidad y seguridad temprana. En esta etapa se incorpora SonarQube Community Build como herramienta de análisis estático del código fuente, permitiendo identificar problemas de calidad, mantenibilidad y seguridad sobre el software analizado. De forma complementaria, se emplea OWASP Dependency-Check para el análisis de composición de software, con el fin de detectar vulnerabilidades conocidas en dependencias y componentes de terceros. Los resultados generados en este bloque se convierten en criterios de decisión dentro del pipeline por medio de quality gates y umbrales mínimos de aceptación, de tal forma que una ejecución puede continuar o detenerse según el nivel de cumplimiento alcanzado.

El tercer bloque corresponde a CD-01 (Containerization & Deployment), en el cual se realiza la construcción de imágenes Docker a partir de los artefactos previamente validados. Posteriormente, dichas imágenes son sometidas a escaneo de seguridad mediante Trivy, lo que permite identificar vulnerabilidades presentes en paquetes, bibliotecas y componentes incluidos dentro del contenedor final. Si los resultados son satisfactorios, las imágenes se publican en un registro de contenedores y se procede al despliegue progresivo del sistema en un entorno de prueba, seguido de verificaciones operativas básicas como smoke tests.

Una característica central del diseño técnico es la trazabilidad del proceso. Cada ejecución del pipeline debe poder asociarse con la versión del repositorio utilizada, los resultados obtenidos durante las validaciones, los artefactos generados y las imágenes construidas en cada fase. Para ello, el flujo contempla la conservación de logs, reportes técnicos, resultados de análisis y artefactos intermedios, de manera que puedan ser consultados posteriormente con fines de auditoría, validación académica o revisión técnica. Esta trazabilidad resulta especialmente relevante en un contexto DevSecOps, dado que permite relacionar hallazgos de seguridad con versiones específicas del software y documentar el comportamiento del pipeline.

Además, el diseño técnico contempla que el despliegue no se ejecute de manera monolítica, sino progresiva y controlada. Esto implica distinguir entre componentes de infraestructura, microservicios principales y servicios complementarios, respetando dependencias funcionales entre ellos y permitiendo validaciones posteriores al despliegue. Esta lógica favorece un arranque más estable del ecosistema Smart Campus UIS y permite detectar de manera temprana problemas asociados a la inicialización de servicios o a la interacción entre componentes.

En conjunto, el diseño técnico detallado define una solución en la que la automatización del pipeline no se limita a compilar y desplegar software, sino que incorpora validación funcional, control de calidad, análisis de seguridad, trazabilidad de artefactos y verificación operativa. Con ello, se consolida una base técnica coherente con los requerimientos funcionales y de seguridad establecidos previamente, y se proporciona el marco estructural necesario para la posterior implementación concreta del pipeline DevSecOps en la plataforma Smart Campus UIS.

6.5 PIPELINE DEFINIDO CON ETAPAS DE VALIDACIÓN Y SEGURIDAD - P3.3

Con el fin de materializar la arquitectura y el diseño técnico definidos en las secciones anteriores, se estructuró el pipeline incorporando diferentes etapas orientadas a la validación funcional, el análisis de seguridad y el despliegue progresivo del sistema. Es importante destacar que, el alcance de esta sección, únicamente se centra en la

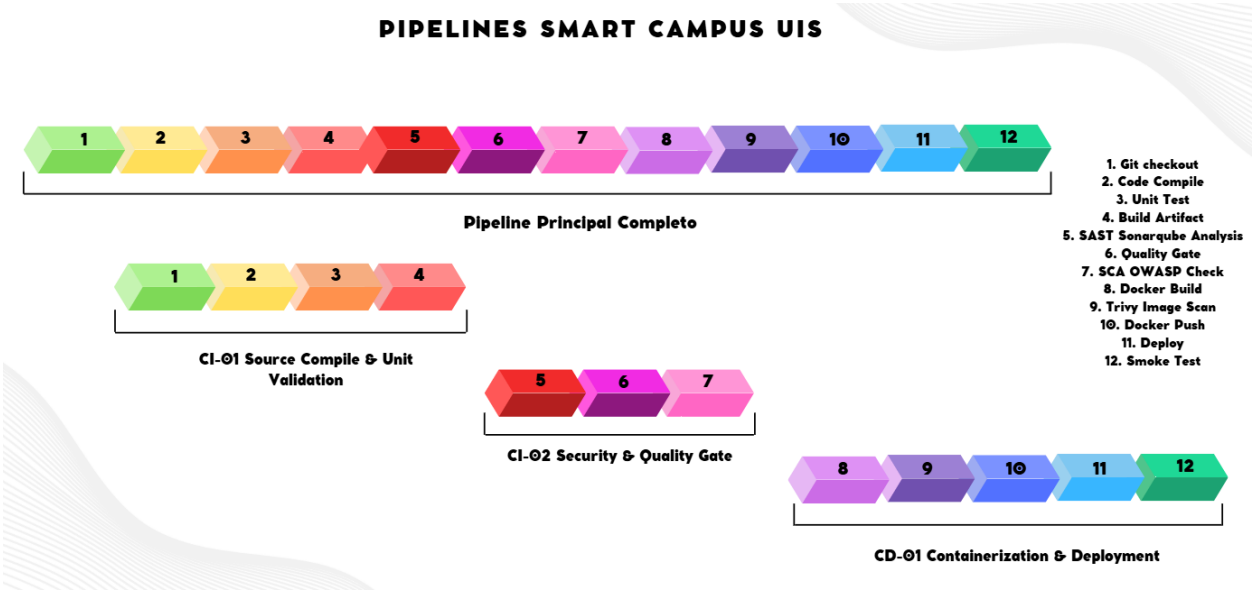
definición de las etapas. La implementación concreta de las herramientas y configuraciones requeridas para su ejecución será abordada en las secciones posteriores del proyecto.

Se estableció una estructura donde se contemplan doce fases o stages principales, que permiten compilar, evaluar y desplegar progresivamente la infraestructura Smart Campus UIS. Con el fin de mejorar la organización, mantenibilidad, granularidad y eficiencia del proceso el pipeline ha sido estructurado en tres subcomponentes principales:

1. CI-01 (Source Compile & Unit Validation)
2. CI-02 (Security & Quality Gate)
3. CD-01 (Containerization & Deployment).

La Figura 32 presenta la estructuración general del pipeline definido para Smart Campus UIS, mostrando la relación entre las doce etapas del flujo completo y su agrupación en los tres bloques funcionales CI-01, CI-02 y CD-01.

Figura 32. Estructuración general del pipeline DevSecOps propuesto para Smart Campus UIS.



Esta segmentación responde a principios fundamentales de la ingeniería de software, particularmente al principio de separación de responsabilidades, el cual establece que cada componente del sistema debe encargarse de una función específica y bien definida [58].

La división en tres bloques permite, en primer lugar, aislar la validación funcional del código CI-01, asegurando que el software cumple con los requisitos básicos de compilación y ejecución antes de ser sometido a análisis más complejos. En segundo lugar, el bloque CI-02 introduce controles de seguridad y calidad actuando como un punto de decisión crítico mediante el uso de mecanismos como lo son los quality gates, los cuales determinan si el software cumple con los estándares definidos para continuar en el proceso. Finalmente, el bloque CD-01 se encarga de la construcción del contenedor, su análisis de seguridad y su despliegue, garantizando que únicamente artefactos validados y seguros sean liberados a los entornos de ejecución.

Desde una perspectiva de eficiencia, esta segmentación permite alinear el pipeline al enfoque fail fast, ampliamente utilizado en prácticas DevOps [59], el cual busca detectar errores en las etapas más tempranas posibles para reducir costos y tiempos de corrección.

Adicionalmente, la inclusión del bloque CI-02 refleja la adopción del principio de “Shift Left Security”, en el cual las pruebas de seguridad se integran en fases tempranas del ciclo de vida del desarrollo, en lugar de ser realizadas únicamente al final [60]. Esto permite identificar vulnerabilidades de forma anticipada, reduciendo significativamente el riesgo de exposición en entornos productivos.

[58] NEELAN, A. The Perennial Importance of SOLID Principles in Software Design. En: Journal of Software Engineering and Simulation. 2024. DOI: https://doi.org/10.35629/3795-10035468
[59] WIEDEMANN, A., et al. The DevOps Phenomenon. En: Queue. 2019, vol. 17, p. 93-112. DOI: https://doi.org/10.1145/3329781.3338532
[60] MALIK, G. y PRASHASTI, P. Shift Left Security. En: The Eastasouth Journal of Information System and Computer Science. 2025. DOI: https://doi.org/10.58812/esiscs.v2i03.528

En conjunto, la arquitectura propuesta no solo mejora la calidad del software entregado, sino que también incrementa la trazabilidad, la automatización y la capacidad de escalamiento en el proceso de desarrollo.

6.5.1 Git Checkout. El pipeline inicia con la fase de git checkout, en la cual se obtiene el código fuente desde el repositorio de control de versiones. Esta etapa garantiza que el proceso se ejecute sobre una versión específica y controlada del software, permitiendo mantener trazabilidad sobre los cambios incorporados y asociar cada ejecución con un estado identificable del proyecto.

6.5.2 Code Compile. En esta fase se transforma el código fuente en un formato ejecutable mediante herramientas de construcción como Maven. De esta manera se permite identificar errores de sintaxis, problemas de configuración o dependencias faltantes que puedan impedir la construcción correcta del sistema.

6.5.3 Unit Test. Una vez compilado el código, se procede a la ejecución de pruebas unitarias. Estas pruebas validan el comportamiento de los componentes individuales del sistema, asegurando que cada unidad funcional opere conforme a lo esperado. Su incorporación en esta etapa permite detectar fallos lógicos en fases tempranas del desarrollo y aporta una primera capa de validación funcional dentro del pipeline.

6.5.4 Build Artifact. Superadas las pruebas unitarias, se genera el artefacto de construcción, el cual constituye la versión empaquetada de la aplicación. Este artefacto, representa la unidad que será utilizada para las siguientes etapas del pipeline, tanto para procesos de análisis como para la construcción de imágenes de contenedor.

6.5.5 Static Application Security Testing (SAST). En esta etapa se realiza el análisis estático de seguridad del código fuente con el fin de identificar defectos, vulnerabilidades potenciales y problemas relacionados con prácticas inseguras de desarrollo. Este análisis permite detectar hallazgos sin necesidad de ejecutar la aplicación, ayudando a fortalecer la postura de seguridad del software desde fases tempranas del flujo.

6.5.6 Quality Gates. Los Quality Gates funcionan como puntos de control dentro del flujo del pipeline, donde se evalúan los resultados obtenidos en las fases previas para determinar si el proyecto cumple con los criterios mínimos de calidad definidos. En caso de que el software no cumpla con los umbrales establecidos, el pipeline puede detener su ejecución, evitando que código con fallos o vulnerabilidades continúe avanzando hacia las siguientes fases.

6.5.7 Software Composition Analysis (SCA) Owasp Check. En esta etapa se analizan las dependencias externas utilizadas por el proyecto, especialmente aquellas provenientes de bibliotecas y componentes de terceros. Se busca identificar posibles vulnerabilidades conocidas presentes en estas dependencias. Resulta fundamental debido a que Smart Campus incorpora componentes de terceros que pueden contener vulnerabilidades reportadas en bases de datos públicas de seguridad.

6.5.8 Docker Build. Esta etapa permite la generación de un artefacto desplegable mediante la construcción de imágenes de contenedor utilizando Docker. A través de este proceso se encapsulan el software y los elementos necesarios para su ejecución, produciendo una imagen consistente y reutilizable para etapas posteriores de validación y despliegue.

6.5.9 Trivy Image Scan. Después de construir la imagen del contenedor, el pipeline contempla una fase de análisis de seguridad enfocada en el escaneo de la imagen generada. El objetivo de esta etapa es identificar vulnerabilidades presentes en paquetes, bibliotecas o componentes incluidos dentro de la imagen Docker, permitiendo validar la seguridad del contenedor antes de su publicación o despliegue.

6.5.10 Docker Push. Una vez validada la imagen, esta se publica en un repositorio o registro de contenedores. Esta etapa permite almacenar, versionar y distribuir el artefacto generado, facilitando su recuperación posterior para procesos de despliegue o validación en otros entornos.

6.5.11 Deploy. Una vez validada la seguridad de la imagen, se procede al despliegue de la aplicación.

6.5.12 Smoke Test. Finalmente, se ejecuta una fase de Smoke Test, cuyo propósito es comprobar que la aplicación desplegada inicia correctamente y que sus componentes esenciales responden de manera esperada. Esta validación constituye una verificación operativa básica después del despliegue, permitiendo detectar de forma temprana fallos críticos en la disponibilidad inicial del sistema.

En conjunto, en este capítulo quedaron definidos los lineamientos que orientan la solución propuesta para Smart Campus UIS, abarcando los requerimientos funcionales y de seguridad, la línea base tecnológica seleccionada, la arquitectura general del pipeline y el diseño técnico de sus etapas principales. Esta definición permitió establecer una relación coherente entre las necesidades identificadas en el diagnóstico previo y los mecanismos concretos de automatización, validación y control de seguridad incorporados en el flujo CI/CD. De esta manera, el capítulo consolida la base conceptual y técnica sobre la cual se sustenta la implementación posterior del pipeline DevSecOps, garantizando que su construcción responda a criterios de trazabilidad, validación progresiva y seguridad temprana.

7. FASE 4: CONFIGURACIÓN, INTEGRACIÓN Y CONTENERIZACIÓN SEGURA

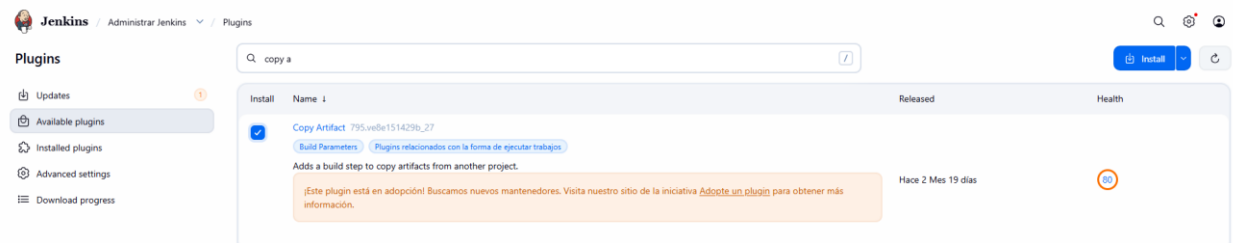
En este capítulo se muestra la implementación técnica del pipeline DevSecOps propuesto para la plataforma Smart Campus UIS, materializando la arquitectura y el diseño definidos en las secciones anteriores. Su propósito es describir cómo se integraron las herramientas, configuraciones y mecanismos de automatización necesarios para validar el código fuente, incorporar controles de seguridad, construir artefactos desplegables y ejecutar el despliegue controlado del sistema.

La implementación se organiza de acuerdo con los tres bloques funcionales definidos previamente en el pipeline: CI-01 (Source Compile & Unit Validation), CI-02 (Security & Quality Gate) y CD-01 (Containerization & Deployment). Esta organización permite mantener una relación directa entre el diseño lógico del flujo y su materialización técnica, facilitando la comprensión del proceso, la trazabilidad de sus componentes y la documentación progresiva de cada fase de automatización.

Uno de los aspectos centrales de esta implementación es el desacoplamiento entre los distintos bloques del pipeline. Para soportar esa lógica, se usan mecanismos de persistencia y transferencia de artefactos, particularmente mediante las funcionalidades archiveArtifacts y copyArtifacts de Jenkins. Usando este enfoque, los artefactos generados en una fase pueden conservarse y ser reutilizados por etapas posteriores sin necesidad de reconstrucción, fortaleciendo la consistencia entre ejecuciones y la trazabilidad de los entregables generados.

En este contexto, debe señalarse que copyArtifacts corresponde a una funcionalidad soportada mediante el plugin Copy Artifact de Jenkins, el cual debe ser instalado para permitir la transferencia de artefactos entre distintos pipelines sin requerir una nueva construcción. Su incorporación resulta especialmente útil en una arquitectura desacoplada como la implementada en este proyecto, donde cada bloque del flujo puede consumir artefactos producidos y validados previamente por otro bloque.

Figura 33. Instalación plugin Copy Artifact en Jenkins



A partir de esta base, las subsecciones siguientes describen la implementación de cada bloque del pipeline: primero la validación funcional y generación de artefactos, luego la integración de controles de calidad y seguridad temprana, y finalmente la construcción, validación y despliegue de imágenes de contenedor. Con esto, el capítulo no se limita solo a exponer configuraciones aisladas, si no que documenta la forma en que las configuraciones funcionan en conjunto para dar vida al flujo DevSecOps implementado en el proyecto.

7.1 PIPELINE FUNCIONAL IMPLEMENTADO - P4.1

El componente CI-01 de pipeline DevSecOps tiene como propósito garantizar la correcta compilación, validación funcional y generación de artefactos del sistema antes de su análisis de seguridad y posterior despliegue.

7.1.1 Organización de proyectos, ejecución de pruebas unitarias y generación de artefactos. La arquitectura del sistema está compuesta por múltiples microservicios, específicamente admin_microservice y data_microservice, organizados en directorios independientes dentro del repositorio. Esta separación permite aplicar principios de modularidad y facilita la ejecución de procesos de integración continua de forma aislada por componente.

Es importante destacar que las pruebas automatizadas dentro del pipeline se enfocan exclusivamente en estos dos microservicios, debido a que constituyen la capa de lógica de negocio del sistema. En ellos se implementan reglas funcionales, el procesamiento de datos y la interacción con los diferentes componentes de almacenamiento y mensajería.

Por el contrario, los demás servicios presentes en la arquitectura, como bases de datos, brokers de mensajería, herramientas de monitoreo y visualización, corresponden a componentes de soporte o infraestructura, los cuales no contienen lógica de negocio propia ni código fuente que requieren validación mediante pruebas unitarias.

Durante la ejecución del pipeline, se utiliza la instrucción dir() para acceder a cada uno de los directorios anteriormente mencionados y ejecutar los comandos correspondientes.

En primer lugar, se realiza la compilación del código mediante el comando **mvn clean compile**, el cual permite validar la integridad estructural del proyecto, identificar errores de sintaxis y asegurar que todas las dependencias estén correctamente configuradas.
Posteriormente, se ejecutan las pruebas unitarias utilizando el comando **mvn test**. Estas pruebas tienen como objetivo validar el comportamiento funcional de las unidades de código de manera independiente, permitiendo detectar errores lógicos en etapas tempranas, del ciclo de desarrollo. La ejecución de pruebas unitarias en cada microservicio de forma separada garantiza una mayor precisión en la identificación de fallos.

Adicionalmente, se lleva a cabo la generación de los artefactos mediante el comando **mvn clean install**, el cual compila, prueba y empaqueta la aplicación en un formato distribuible. Este artefacto representa la unidad funcional que será utilizada en las siguientes etapas del pipeline, particularmente en los procesos de seguridad y despliegue.

Figura 34. Parametrización inicial del pipeline en Jenkins mediante variables de repositorio.

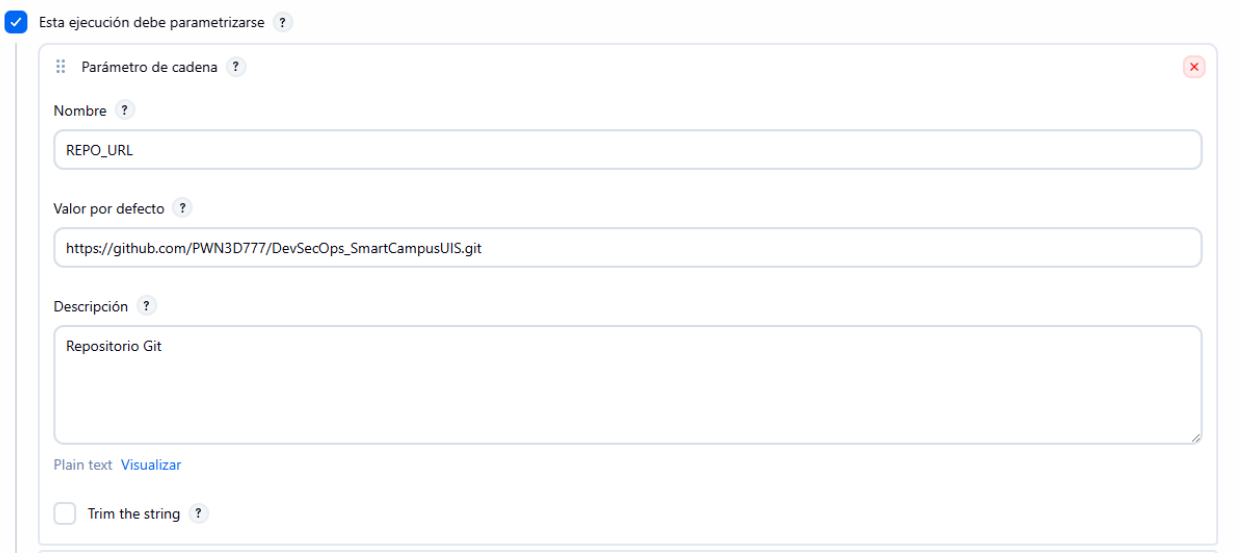


Figura 35. Parámetros de rama utilizados para controlar la versión analizada en el pipeline.

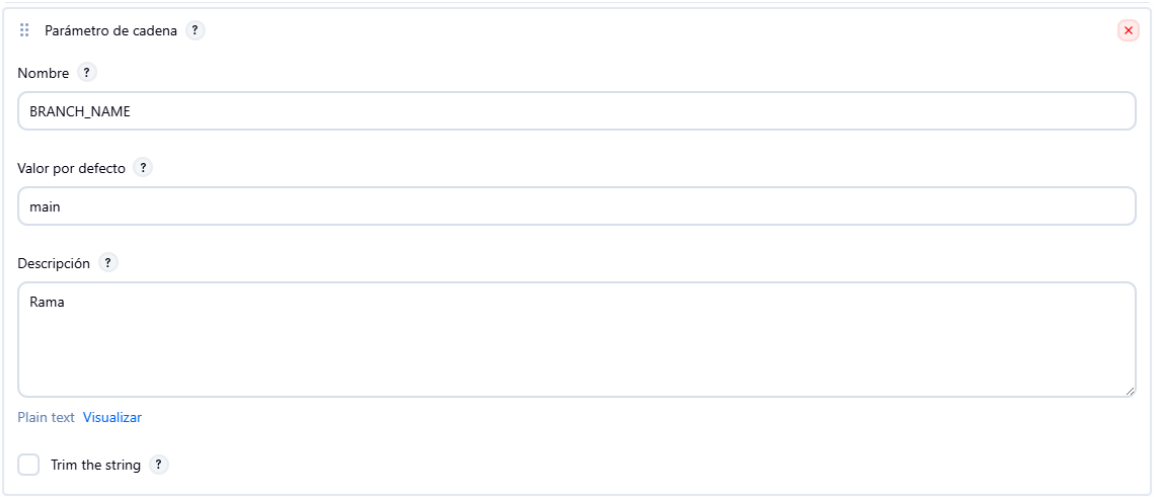


Figura 36. Parámetros de commit utilizados para controlar la versión analizada en el pipeline.

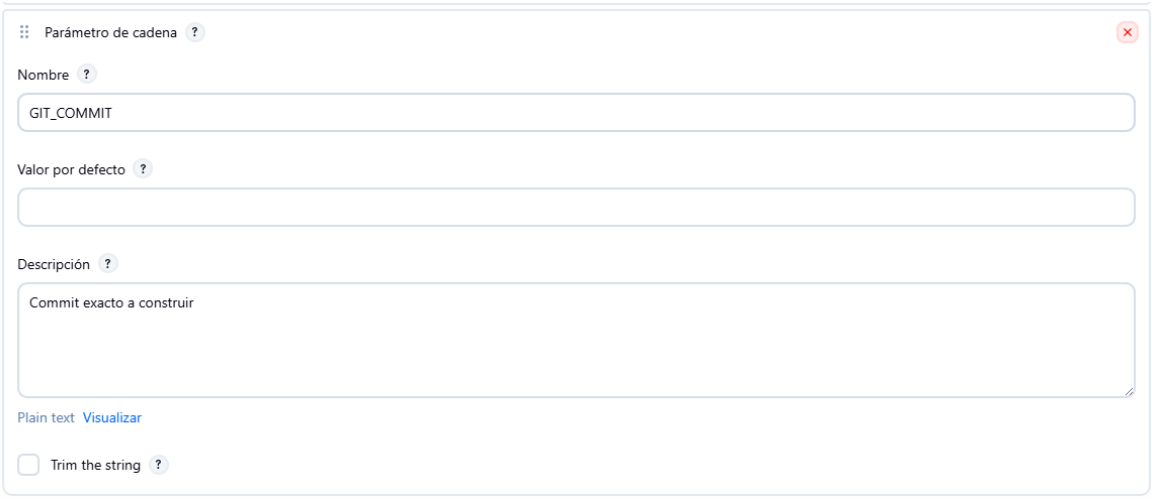


Figura 37. Definición de la etapa de recuperación del código fuente (Git checkout) dentro del pipeline.

```
pipeline {
    agent any

    tools {
        jdk 'JDK-17'
        maven 'maven3'
    }

    parameters {
        string(name: 'REPO_URL', defaultValue: 'https://github.com/PWN3D777/DevSecOps_SmartCampusUIS.git', description: 'Repositorio Git')
        string(name: 'BRANCH_NAME', defaultValue: 'main', description: 'Rama')
        string(name: 'GIT_COMMIT', defaultValue: '', description: 'Commit exacto a construir')
    }

    stages {
        stage('Git checkout') {
            steps {
                git branch: params.BRANCH_NAME, url: params.REPO_URL
                script {
                    if (params.GIT_COMMIT?.trim()) {
                        sh "git checkout ${params.GIT_COMMIT}"
                    }
                }
                sh 'git rev-parse HEAD > .git_commit'
            }
        }
```

Figura 38. Script del pipeline para las etapas de compilación, pruebas unitarias y generación de artefactos en los microservicios priorizados.

```
27
28          stage('Code compile') {
29              steps {
30                  dir('admin_microservice') {
31                      sh 'mvn -B clean compile'
32                  }
33                  dir('data_microservice') {
34                      sh 'mvn -B clean compile'
35                  }
36              }
37          }
38
39          stage('Unit Test') {
40              steps {
41                  dir('admin_microservice') {
42                      sh 'mvn -B test'
43                  }
44                  dir('data_microservice') {
45                      sh 'mvn -B test'
46                  }
47              }
48          }
49
50          stage('Build artifact') {
51              steps {
52                  dir('admin_microservice') {
53                      sh 'mvn -B -DskipTests package'
54                  }
55                  dir('data_microservice') {
56                      sh 'mvn -B -DskipTests package'
57                  }
58              }
```

Con el fin de garantizar la trazabilidad y análisis de los resultados de las pruebas unitarias, se integró el uso del plugin JUnit de Jenkins encargado de procesar los reportes generados. Durante la ejecución de pruebas unitarias mediante Maven, se generan automáticamente reportes en formato XML. Estos reportes contienen información detallada sobre el estado de las pruebas incluyendo casos exitosos, fallidos y errores de ejecución. Para procesar esta información dentro del pipeline, se utiliza la directiva junit en la sección post, la cual permite a Jenkins interpretar los resultados generados y presentarlos de manera estructurada en la interfaz del sistema. Esto facilita la visualización de métricas como números de pruebas ejecutadas, tasa de éxito y fallos específicos.

La integración de JUnit no solo mejora la visibilidad del estado del software, sino que también permite establecer criterios de calidad medibles, los cuales pueden ser utilizados en etapas posteriores. De esta manera, se fortalece el control de calidad dentro del proceso de integración continua.

Figura 39. Instalación del plugin junit en Jenkins.

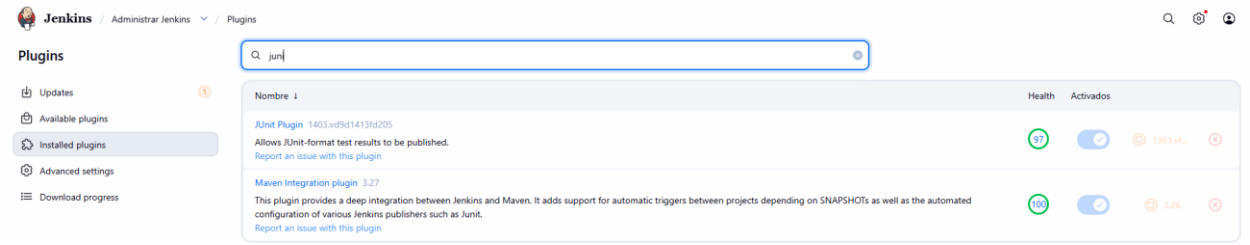


Figura 40. Integración del plugin JUnit y configuración de acciones posteriores para el procesamiento de reportes y archivado de artefactos.

```
61
62      post {
63          always {
64              junit '**/target/surefire-reports/*.xml'
65              archiveArtifacts artifacts: '''admin_microservice/target/*.jar,
66              data_microservice/target/*.jar, .git_commit, **/Dockerfile''', fingerprint: true
67          }
68      }
69  }
```

Figura 41. Tendencia de los resultados de pruebas unitarias registradas por Jenkins.

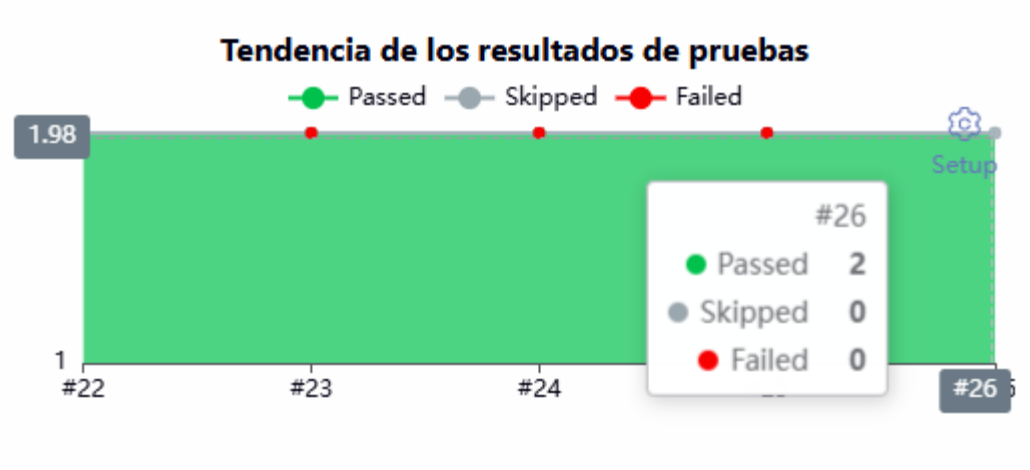


Figura 42. PARTE 1 - Ejecución exitosa del bloque CI-01 con generación de artefactos y visualización de etapas en Jenkins.

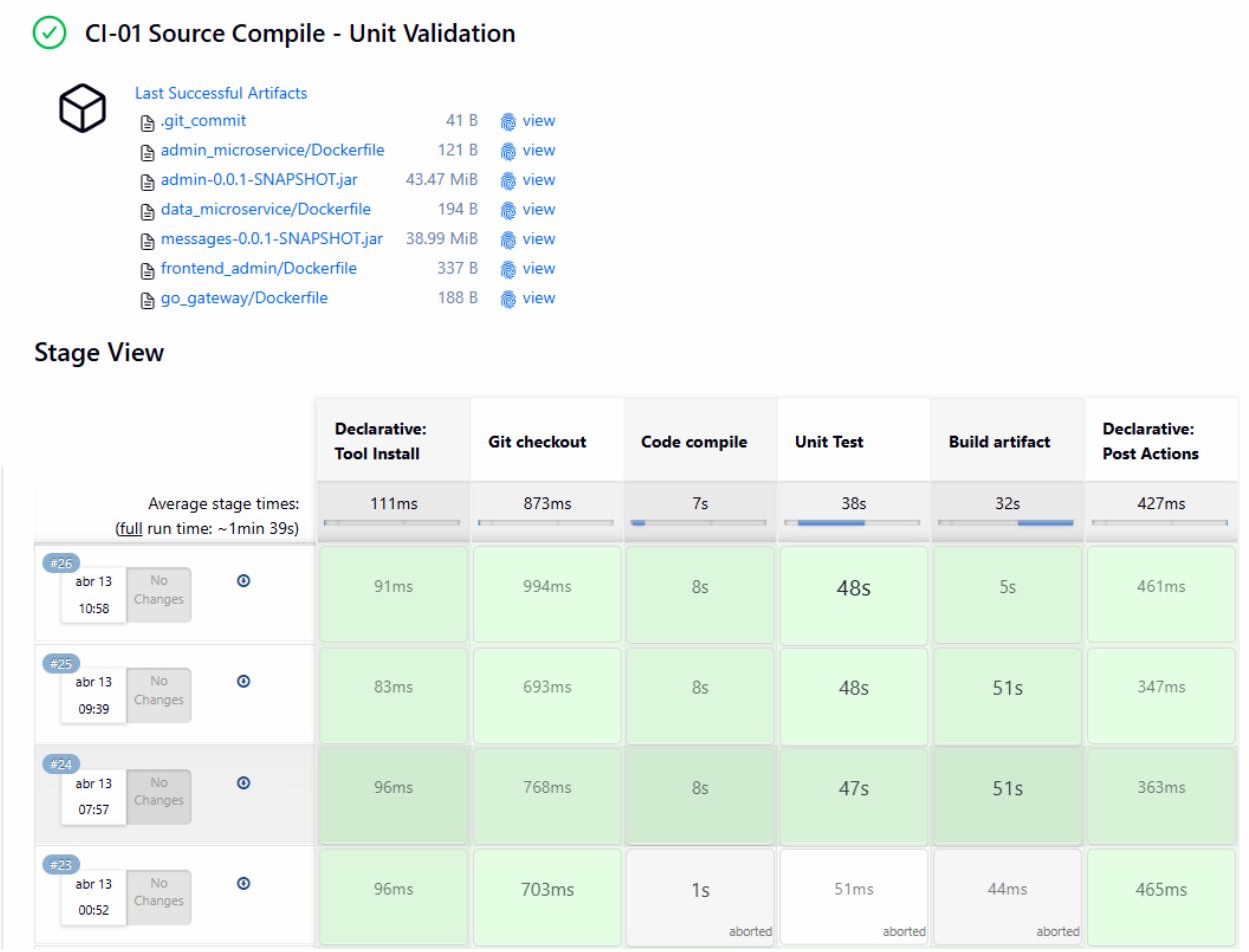
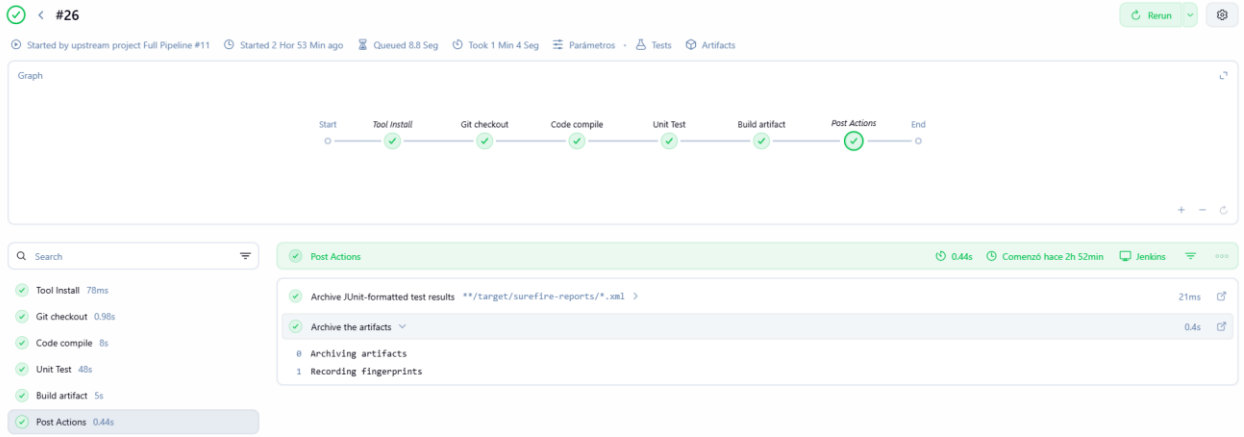


Figura 43. PARTE 2 - Ejecución exitosa del bloque CI-01 con generación de artefactos y visualización de etapas en Jenkins.



7.2 CONFIGURACIÓN E INTEGRACIÓN DE AUTOMATIZACIÓN PARA EL ANÁLISIS DE SEGURIDAD DEL CÓDIGO FUENTE (SAST), QUALITY GATE E INTEGRACIÓN DEL ANÁLISIS DE DEPENDENCIAS (SCA)

El bloque CI-02 del pipeline DevSecOps tiene como propósito incorporar controles de calidad y seguridad temprana sobre el software validado previamente en la fase de compilación y pruebas unitarias. En esta etapa se integran mecanismos orientados al análisis estático del código fuente, la evaluación de criterios mínimos de calidad mediante quality gates y el análisis de dependencias externas, con el fin de impedir que versiones no conformes del software avancen hacia fases posteriores de contenerización y despliegue.

La implementación de este bloque se apoya en la línea base tecnológica establecida previamente para el proyecto, en la cual SonarQube Community Build fue definido como herramienta SAST y OWASP Dependency-Check como herramienta SCA. Esta separación funcional permite evaluar, por un lado, la calidad y seguridad del código desarrollado internamente y, por otro, los riesgos asociados al uso de bibliotecas y componentes de terceros. Con ello, el pipeline incorpora una segunda capa de validación centrada en la seguridad del software, coherente con el principio de Shift Left Security adoptado en el trabajo.

7.2.1 Implementación de automatización sast y quality gate. Partiendo de la base planteada en secciones anteriores resulta de gran importancia configurar el servicio de análisis estático de código. El uso de la herramienta se debe fundamentar en la necesidad de incorporar controles automáticos de calidad y seguridad del código permitiendo identificar vulnerabilidades, errores de programación y malas prácticas antes de que el software sea desplegado en producción. De esta manera, es como SonarQube fortalece la integración de prácticas de seguridad dentro del ciclo de desarrollo, alineándose con los principios DevSecOps.

La integración de la herramienta se realiza por medio la configuración del servicio definido mediante Docker Compose, donde se especifica la imagen del servicio, la conexión con

una base de datos PostgreSQL, los puertos expuestos y los volúmenes utilizados para la persistencia de datos. Adicionalmente en este archivo se configura la base de datos utilizada, donde se crea un usuario específico dentro del servidor PostgreSQL y se genera la base de datos que será utilizada por el sistema de análisis.

A grandes rasgos en el proceso se ejecutan comandos SQL para cumplir con las siguientes funciones:

1. Creación de la base de datos asociada.
2. Creación del usuario del sistema de análisis.
3. Asignación de privilegios de acceso al usuario creado.
4. Establecimiento de una contraseña segura para el acceso a la base de datos.

Figura 44. Configuración del servicio SonarQube dentro del entorno contenerizado.

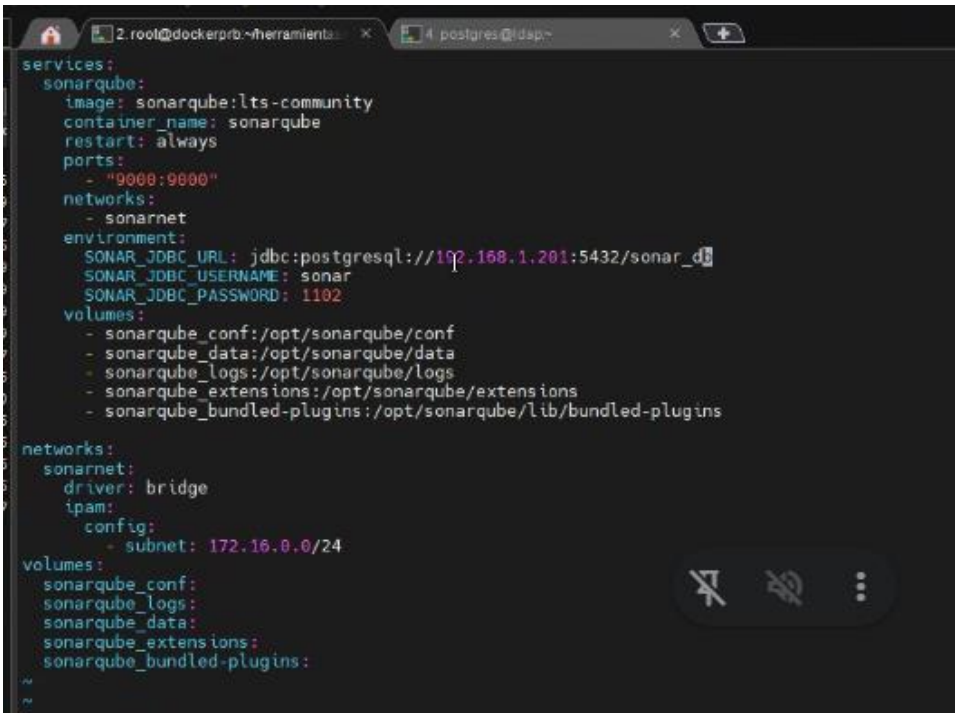


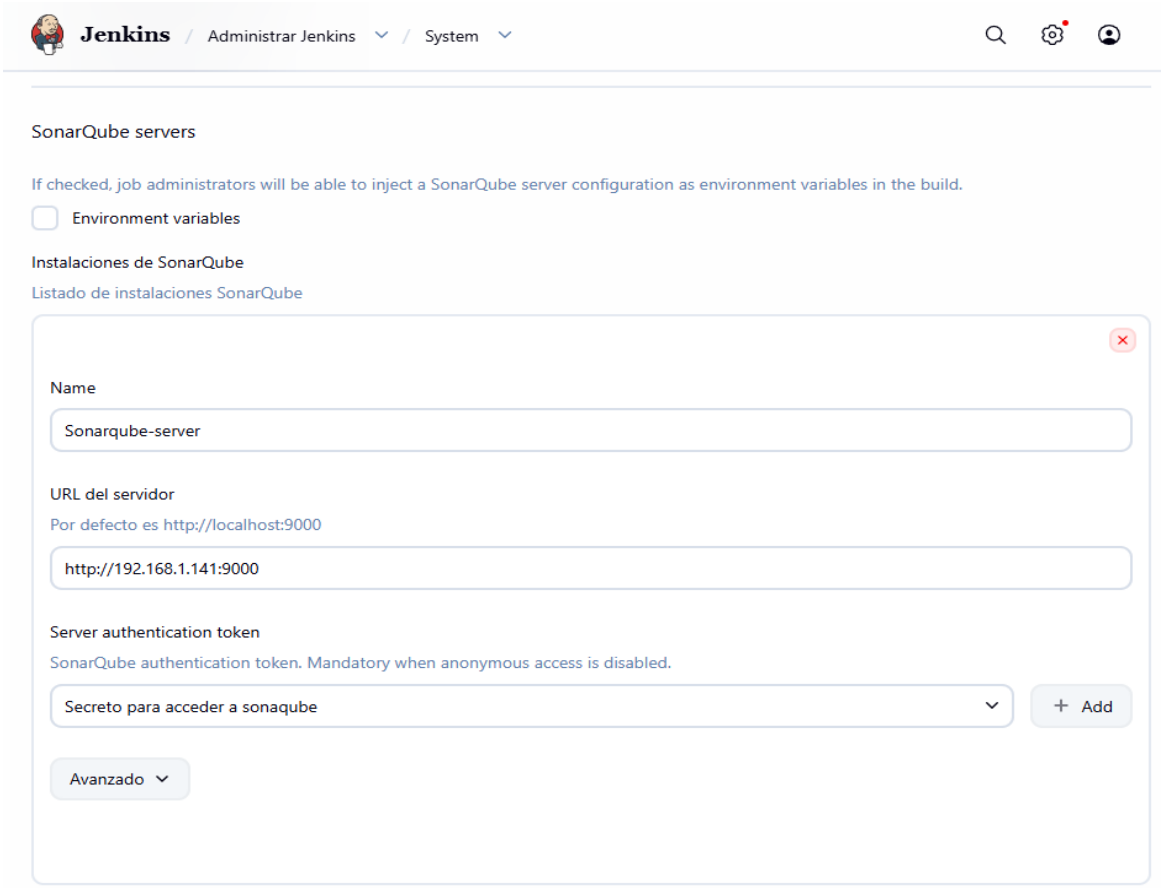
Figura 45. Creación de la base de datos de SonarQube en Postgresql.

Posteriormente dentro de Jenkins se registra el servidor de SonarQube especificando la URL del servicio y el mecanismo de autenticación mediante token. Esta integración permite que, durante la ejecución del pipeline, se puedan generar reportes sobre problemas de seguridad y métricas de calidad del software que apoyan el proceso de análisis durante el proceso de desarrollo.

Posteriormente se configura la herramienta SonarQube Scanner dentro del apartado de herramientas de Jenkins siguiendo el siguiente path:
Manage Jenkins > Tools > Busca SonarQube Scanner > Agrega SonarQube scanner > Selecciona un nombre y una versión.

Dentro del pipeline se invoca el escáner de SonarQube especificando el nombre del proyecto, el directorio base de análisis y la clave identificadora del proyecto por medio de la configuración de variables de entorno.

Figura 46. Generación del token de autenticación en SonarQube para su integración con Jenkins.



Para la autenticación mediante token es necesario navegar por medio de SonarQube server > Administration > Security > Users > Click en el icono debajo de Token y luego llenar los campos de Nombre y Fecha de expiración y Click en generar. Se copia el Token > De vuelta a Jenkins > Manage Jenkins > Credentials > click on Global > Add credentials > selecciona el tipo 'Secret Text' > Se pega el token en la sección Secret > Se agrega una descripción opcional.

Figura 47. Registro de credenciales de SonarQube en Jenkins mediante Secret Text.

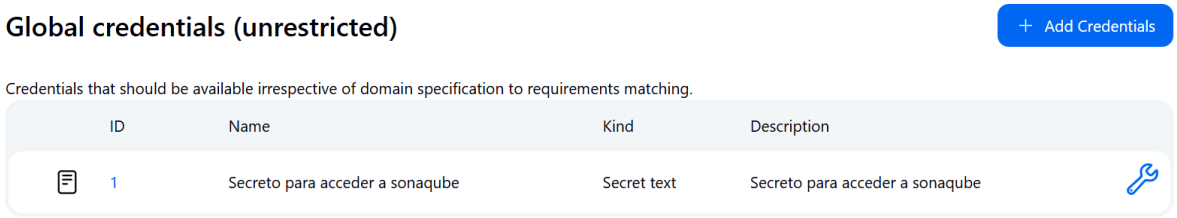
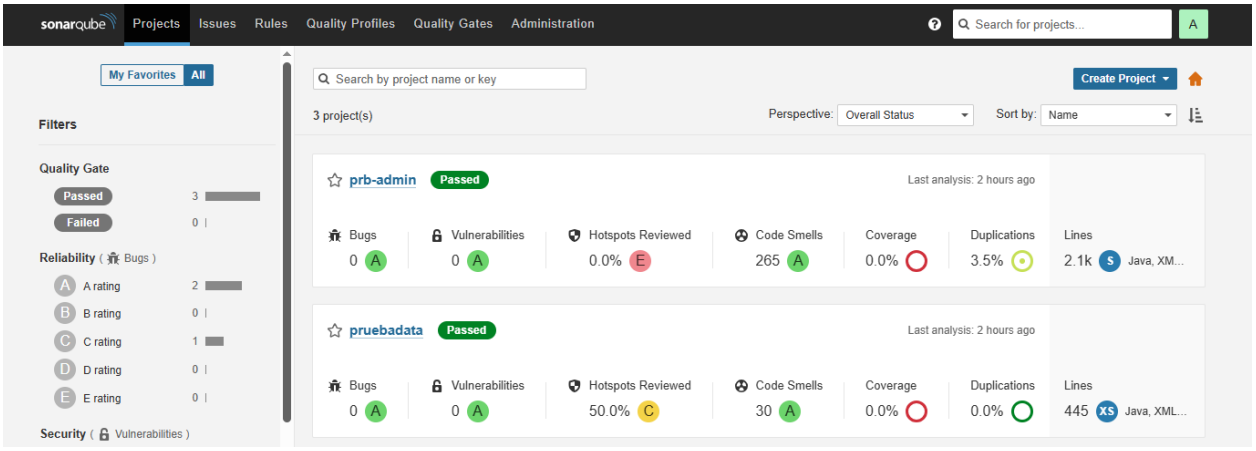


Figura 48. Invocación del análisis SAST dentro del pipeline mediante SonarScanner.

```
pipeline {
    agent any
    environment {
    SCANNER_HOME = tool 'sonar-scanner'
    NVD_API_KEY = credentials('nvd-api-key')
    }
    stages {
        stage('Get Artifacts from Pipeline 1') {
            steps {
                copyArtifacts(
                    projectName: 'CI-01 Source Compile - Unit Validation',
                    selector: [$class: 'StatusBuildSelector', stable: false]
                )
            }
        }
        stage('Sonarqube Analysis') {
            steps {
                withSonarQubeEnv('Sonarqube-server') {
                     dir('admin_microservice') {
                        sh ''' $SCANNER_HOME/bin/sonar-scanner -Dsonar.projectName=prb-admin \
                        -Dsonar.java.binaries=. \
                        -Dsonar.projectKey=prb-admin '''
                    }

                    dir('data_microservice') {
                        sh ''' $SCANNER_HOME/bin/sonar-scanner -Dsonar.projectName=pruebadata \
                        -Dsonar.java.binaries=. \
                        -Dsonar.projectKey=pruebadata '''
                    }
                }
            }
        }
```

Como resultado de este proceso, SonarQube genera métricas relacionadas con la calidad del código, identificando defectos, vulnerabilidades, code smells y métricas de mantenibilidad. En el panel de resultados del proyecto se observa que el Quality Gate fue superado correctamente, indicando que el código cumple con las condiciones mínimas de calidad definidas en la plataforma.

Figura 49. Resultado del análisis SAST.



Dentro de la arquitectura del pipeline DevSecOps, el Quality Gate se establece como un mecanismo fundamental de control que permite evaluar de manera automática si el software cumple con los criterios mínimos de calidad y seguridad antes de avanzar a las siguientes etapas del proceso. Este componente se ubica en el bloque CI-02 y actúa como un punto de decisión crítico dentro del flujo de integración continua.

El Quality Gate se implementa mediante la integración de herramientas de análisis estático, como SonarQube, las cuales permiten evaluar el código fuente en función de un conjunto de reglas previamente definidas. Estas reglas incluyen métricas relacionadas con la calidad del código, tales como la cantidad de errores, vulnerabilidades, code smells y nivel de deuda técnica.

Durante la ejecución del pipeline, una vez finalizado el análisis estático (SAST), los resultados obtenidos son enviados a SonarQube, donde se comparan con los umbrales establecidos en el Quality Gate. Estos umbrales pueden configurarse según las necesidades del proyecto; por ejemplo, se puede exigir la ausencia de vulnerabilidades críticas o una cobertura mínima de pruebas superior a un determinado porcentaje.

El resultado de esta evaluación es binario: aprobado o rechazado. En caso de que el código no cumpla con los criterios definidos, el Quality Gate bloquea automáticamente la ejecución del pipeline, impidiendo que el artefacto avance hacia las etapas de

empaquetado, escaneo de contenedores o despliegue. Este comportamiento garantiza que únicamente versiones del software que cumplen con los estándares establecidos puedan ser promovidas a entornos posteriores.

Figura 50. Creación del Quality Gate SmartCampus en SonarQube.

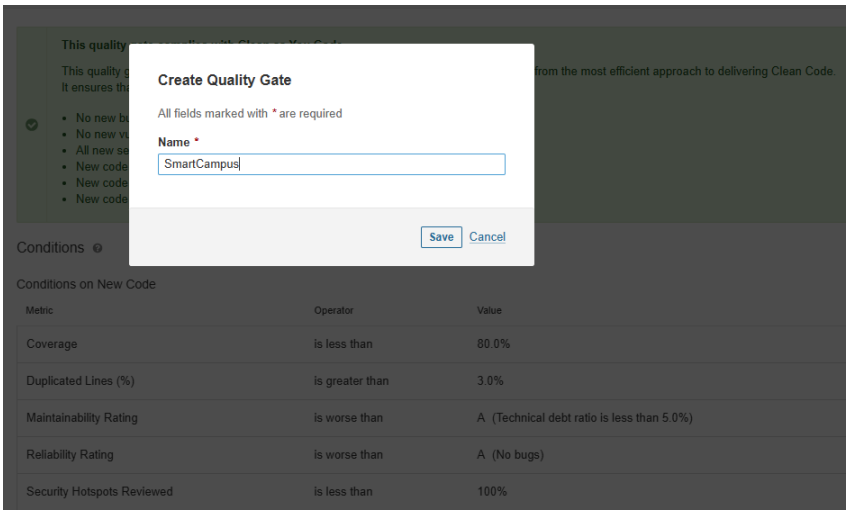


Figura 51. Configuración de condiciones del Quality Gate SmartCampus para código nuevo y código global.

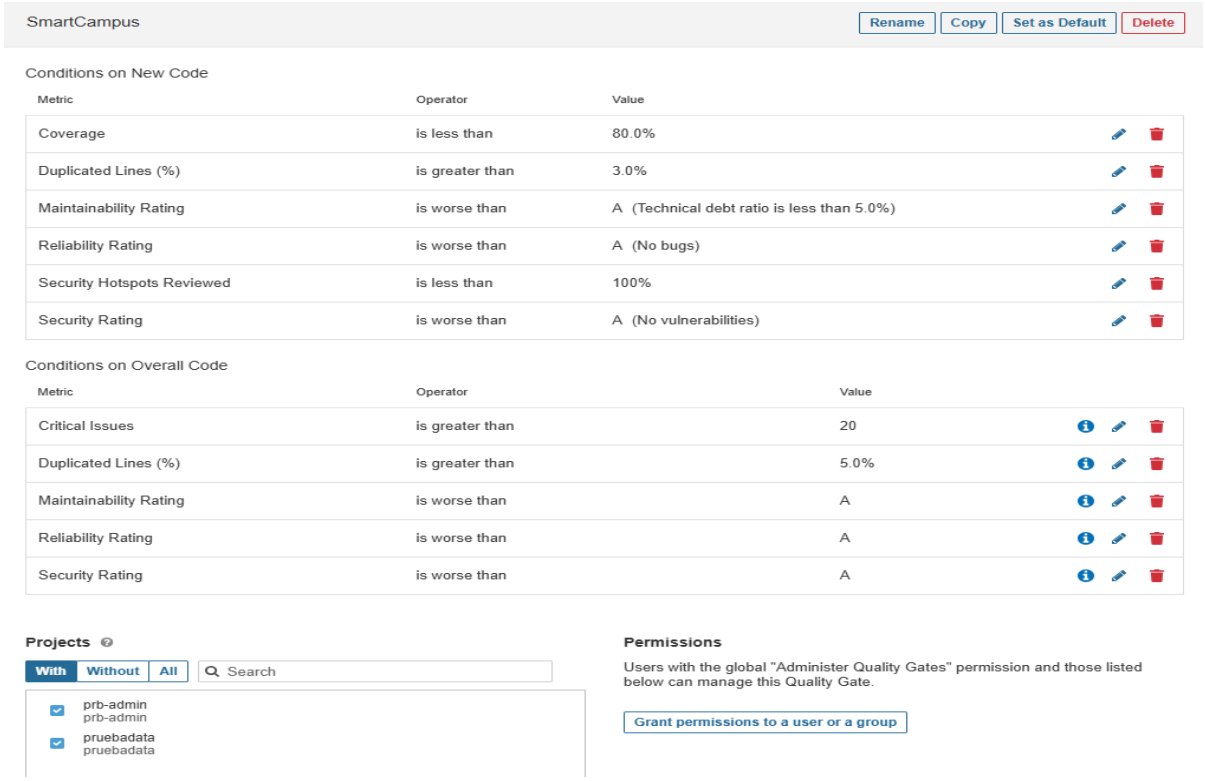


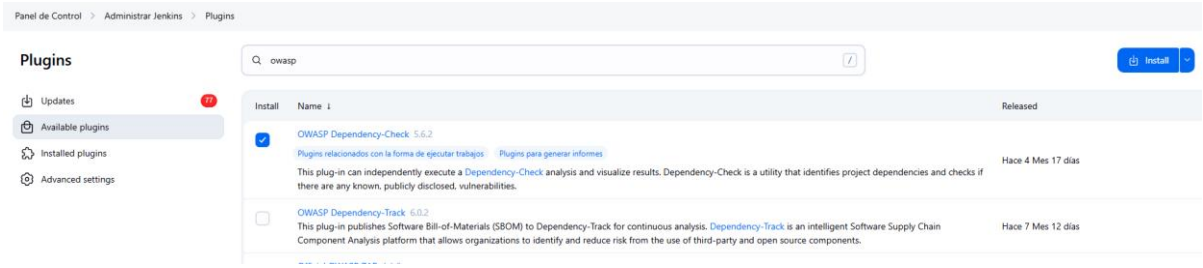
Figura 52. Integración de la etapa Quality Gate en el pipeline de Jenkins mediante waitForQualityGate.

```
34
35          stage('Quality Gate') {
36              steps {
37                  waitForQualityGate abortPipeline: true
38              }
39          }
```

7.2.2 Integración de software composition analysis. Se integra el análisis de dependencias externas mediante el uso de OWASP Dependency-Check, una herramienta desarrollada por OWASP que permite identificar vulnerabilidades conocidas presentes en las bibliotecas y componentes de terceros utilizados en el proyecto. La incorporación de este mecanismo responde a la necesidad de controlar los riesgos asociados al uso de dependencias externas, las cuales pueden introducir vulnerabilidades en el software aun cuando el código desarrollado internamente sea seguro.

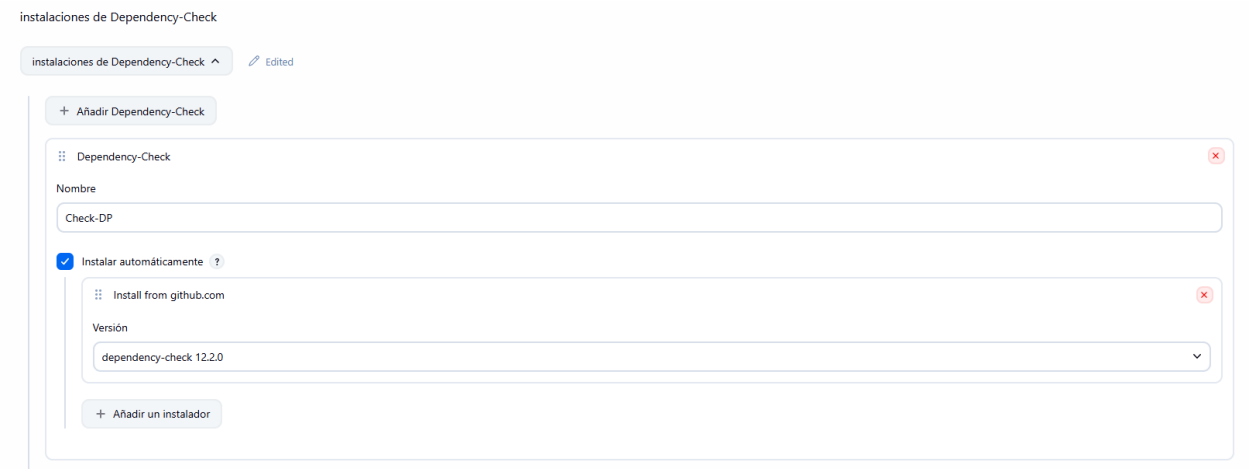
En la captura correspondiente se observa la instalación del plugin de OWASP Dependency-Check dentro del gestor de plugins de Jenkins. Este plugin permite integrar la herramienta de análisis de composición de software directamente en el pipeline de integración continua, facilitando la ejecución automática de escaneos de seguridad sobre las dependencias de los componentes.

Figura 53. Instalación del plugin OWASP Dependency-Check en Jenkins.



Una vez instalado el plugin, se procede a la configuración de la herramienta dentro del apartado de herramientas globales de Jenkins. En esta etapa se define el nombre de la instalación y se habilita la descarga automática de la versión correspondiente de la herramienta.

Figura 54. Configuración global de OWASP Dependency-Check en Jenkins.



La utilización de una API Key de la NVD optimiza el proceso de análisis, evitando limitaciones en el número de consultas y mejorando la precisión de los resultados. Esta clave se gestiona de forma segura dentro de Jenkins mediante el uso de credenciales, lo que permite proteger información sensible y evitar su exposición en el código del pipeline.

Figura 55. Gestión segura de la API Key de la NVD dentro de Jenkins.

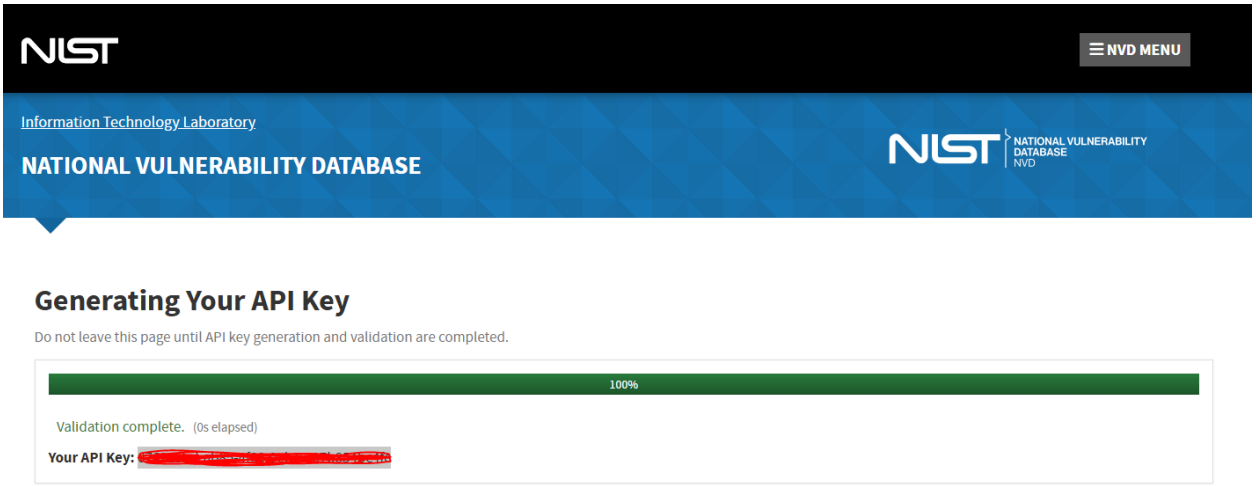


Figura 56. Vista general del almacén de credenciales configurado en Jenkins para los servicios integrados del pipeline.

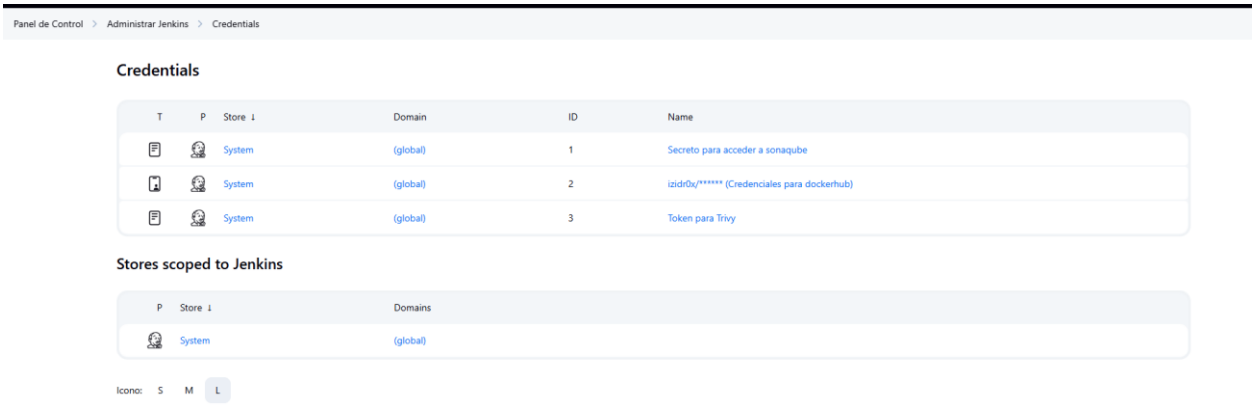


Figura 57. Credenciales globales de Jenkins.

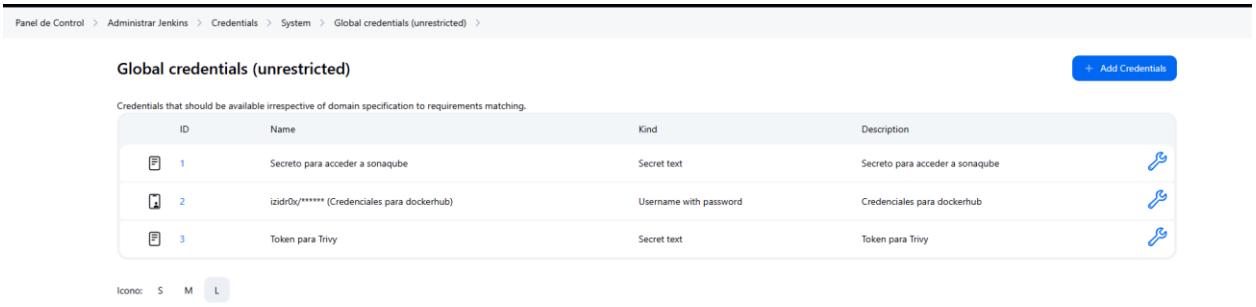
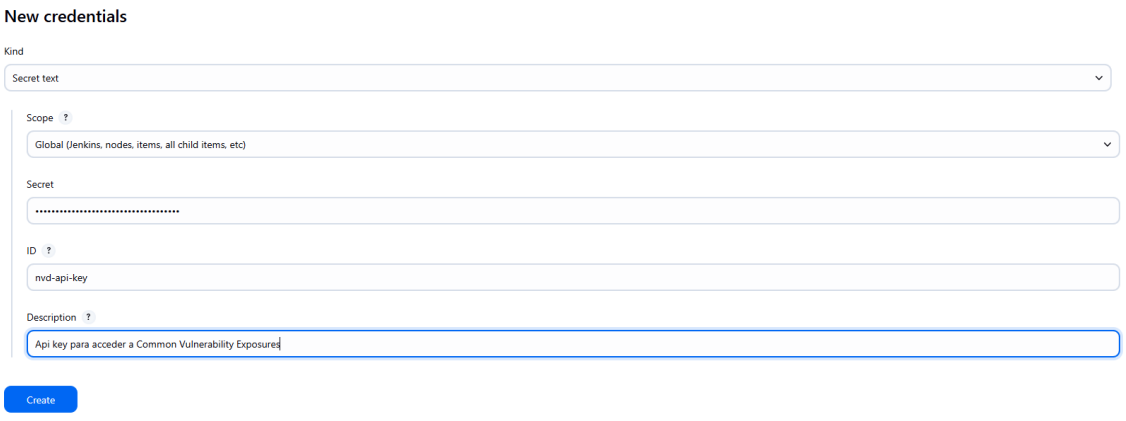


Figura 58. Creación de la credencial nvd-api-key en Jenkins para el acceso a la NVD.



Dentro del pipeline se agregan los comandos necesarios para la ejecución del escaneo de dependencias utilizando la herramienta previamente configurada.

Figura 59. Definición de la variable de entorno NVD_API_KEY en el pipeline mediante credenciales de Jenkins.

```
1
2  pipeline {
3      agent any
4      environment {
5      SCANNER_HOME = tool 'sonar-scanner'
6      NVD_API_KEY = credentials('nvd-api-key')
7      }
```

Figura 60. Integración de la etapa OWASP CHECK en el pipeline con uso de la API Key de la NVD y publicación del reporte XML.

```
40         stage('OWASP CHECK'){
41             steps{
42
43                 dependencyCheck additionalArguments: """
44                 --scan ./
45                 --format XML
46                 --nvdApiKey ${NVD_API_KEY}
47                 --nvdApiDelay 6000
48                 --out .
49                 """, odcInstallation: 'Check-DP'
50
51                 dependencyCheckPublisher pattern: 'dependency-check-report.xml', stopBuild: false
52             }
53         }
54     }
55     post {
56         always {
57             archiveArtifacts artifacts: '''admin_microservice/target/*.jar, data_microservice/target/*.jar,
58             .git_commit, **/Dockerfile''', fingerprint: true
59         }
60     }
61 }
62
```

Como resultado de este proceso se genera un reporte de análisis en formato XML que contiene el detalle de las vulnerabilidades detectadas, incluyendo su identificación, severidad y descripción. Dicho reporte puede ser posteriormente procesado para apoyar la toma de decisiones de desarrollo y despliegue.

Figura 61. Tendencia histórica de vulnerabilidades detectadas por OWASP Dependency-Check en ejecuciones del pipeline.

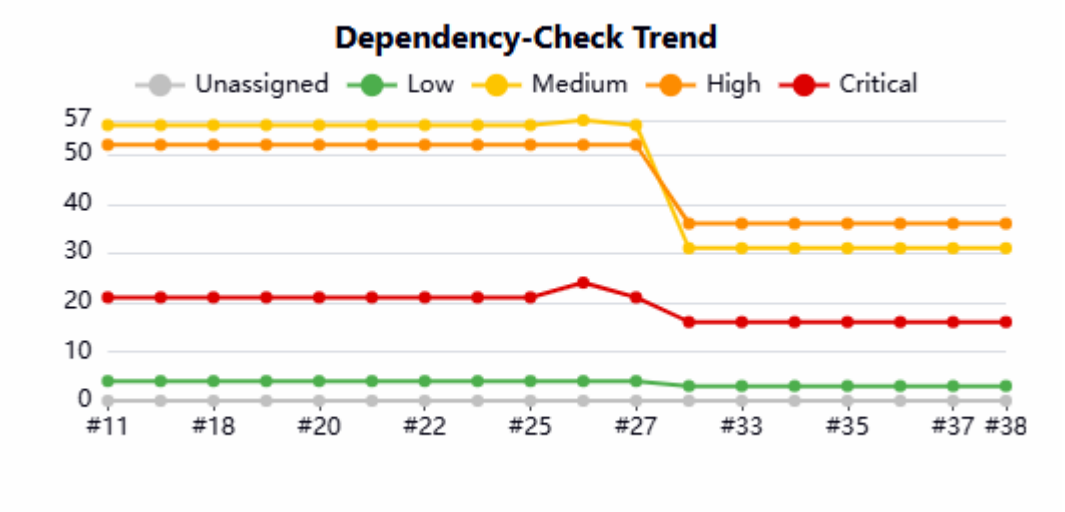


Figura 62. PARTE 1 - Ejecución satisfactoria del bloque CI-02 Security - Quality Gate con artefactos y etapas completadas en Jenkins.

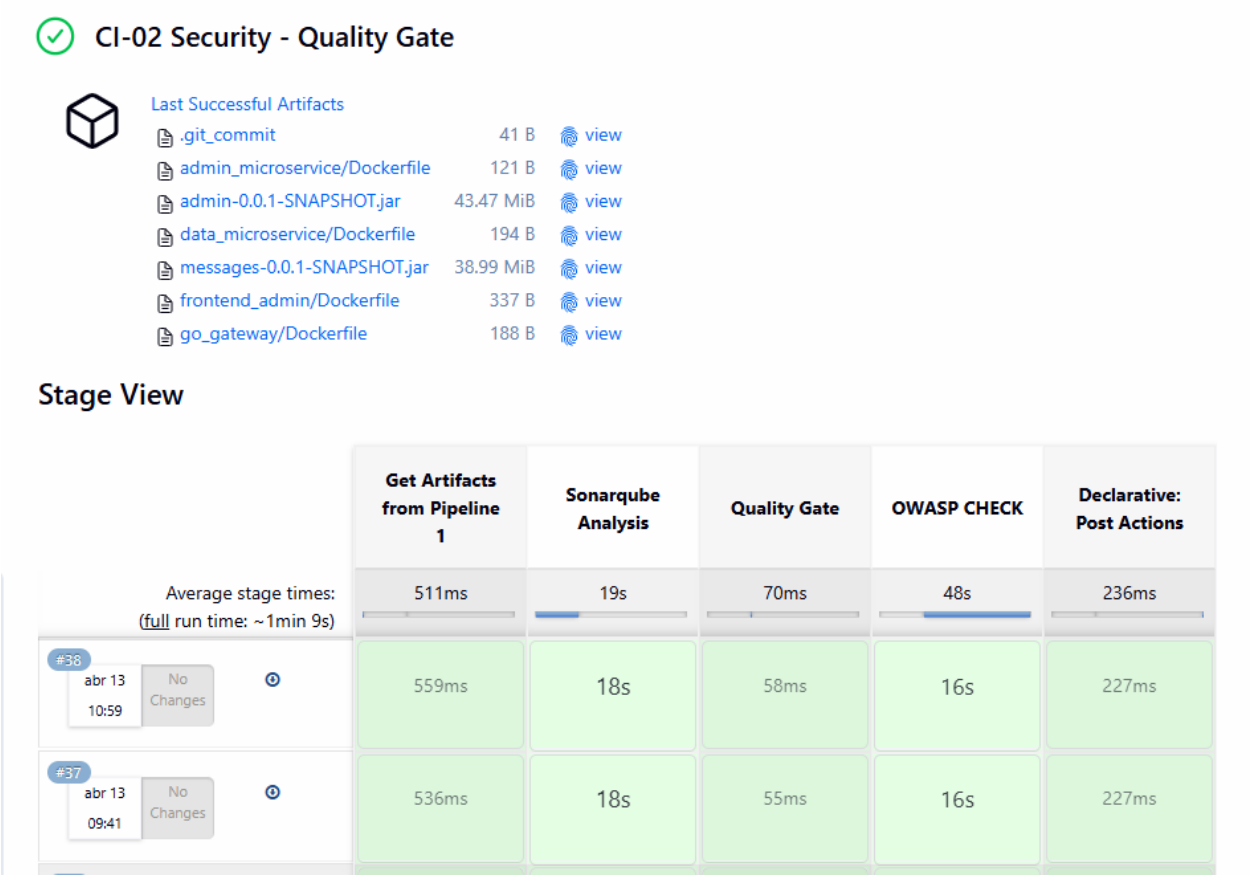
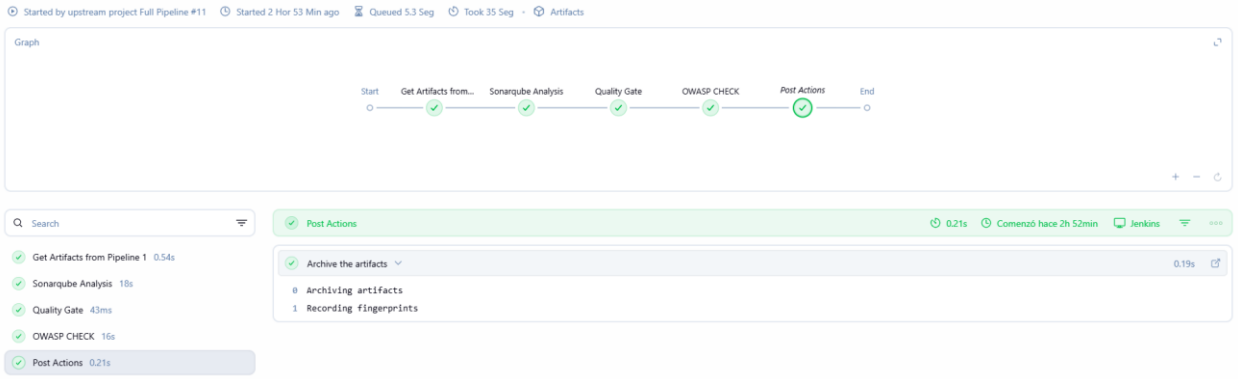


Figura 63. PARTE 2 - Ejecución satisfactoria del bloque CI-02 Security - Quality Gate con artefactos y etapas completadas en Jenkins.



7.3 ARTEFACTOS DE DESPLIEGUE DEL SISTEMA - P4.3

7.3.1 Configuración de herramientas de contenerización. Con el propósito de habilitar la construcción y despliegue de contenedores dentro del pipeline, se realiza la instalación de los plugins de integración con Docker en el servidor de Jenkins. Esta integración permite que el sistema de automatización interactúe directamente con el motor de contenedores, facilitando la ejecución de procesos de construcción, gestión y despliegue de imágenes dentro del flujo automatizado de desarrollo.

Se procede con la instalación de los plugins Docker, Docker Pipeline, Docker API y docker-build-step dentro del gestor de plugins de Jenkins. Estos componentes amplían las capacidades del servidor permitiendo ejecutar comandos y operaciones relacionadas con Docker directamente desde las distintas etapas definidas.

Figura 64. Instalación de plugins de integración con Docker en Jenkins.

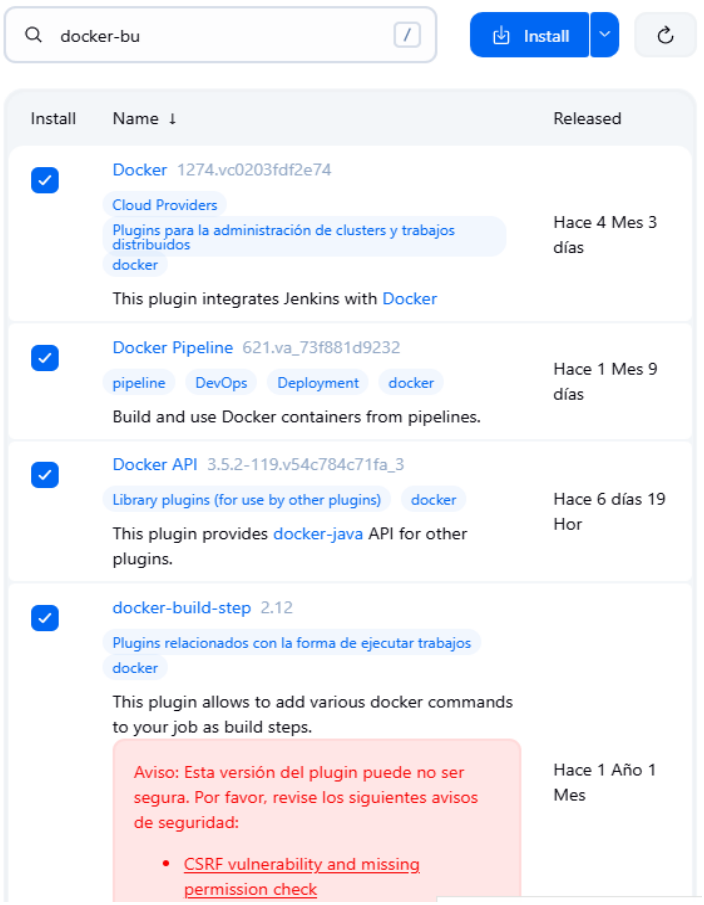
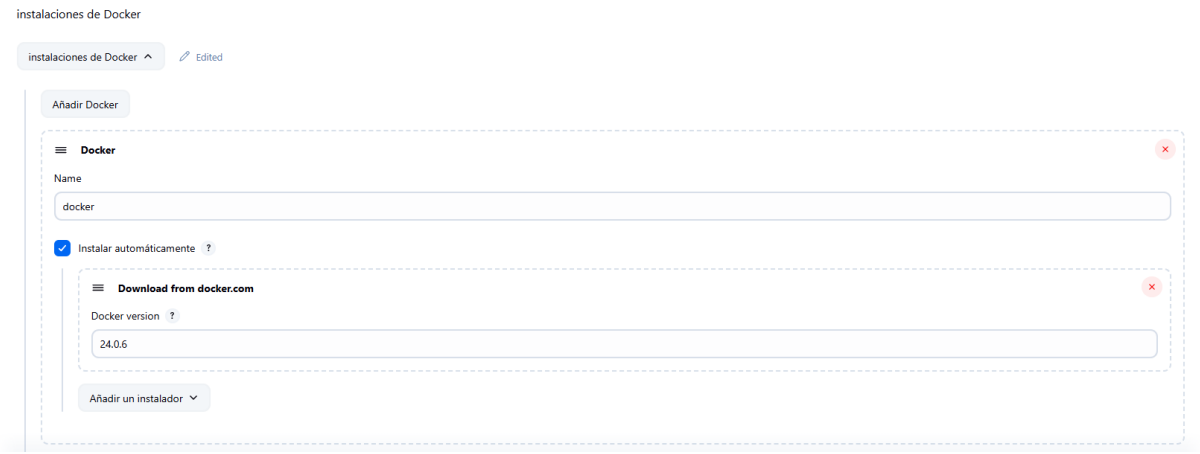


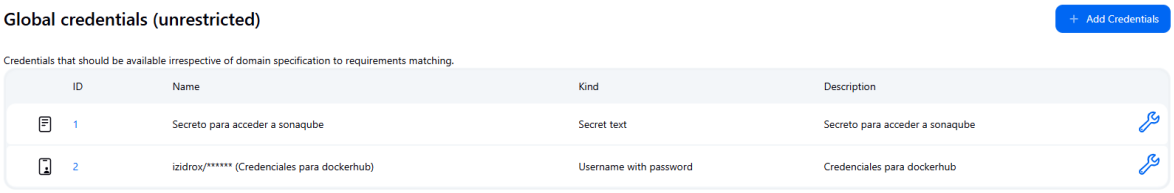
Figura 65. Configuración de la instalación de Docker en Jenkins.



Adicionalmente, se configuran las credenciales necesarias para establecer autenticación con el registro de contenedores utilizado para almacenar las imágenes generadas

durante el proceso de construcción. La gestión centralizada de credenciales dentro de Jenkins permite mantener un control adecuado sobre el acceso a los repositorios de imágenes.

Figura 66. Credenciales globales configuradas en Jenkins para autenticación con Docker Hub.



7.3.2 Construcción de la imagen del sistema. Una vez completadas las etapas de análisis de seguridad del código y de las dependencias del proyecto, el pipeline procede a la construcción de la imagen contenerizada de la aplicación. Para ello se define un archivo Dockerfile, en el cual se especifican los parámetros necesarios para generar el entorno de ejecución de la aplicación dentro de un contenedor.

En este archivo se establece la imagen base utilizada para la aplicación, el puerto expuesto por el servicio, el artefacto generado durante la fase de compilación del proyecto y el comando encargado de iniciar la ejecución del contenedor. Esta configuración permite encapsular todos los elementos necesarios para ejecutar la aplicación de manera consistente en diferentes entornos.

Figura 67. Definición de la herramienta Docker y de las variables de entorno utilizadas para la construcción de imágenes en el pipeline.

```
pipeline {
    agent any

    tools {
        dockerTool 'docker'
    }

    environment {
        DOCKER_NAMESPACE = 'izidr0x'
        IMAGE_TAG = "${env.BUILD_NUMBER}"
    }

    stages {
        stage('Git checkout') {
            steps {
                git branch: 'main', url: 'https://github.com/PWN3D777/DevSecOps_SmartCampusUIS.git'
            }
        }
        stage('Get Artifacts from CI-01 (Prueba1)') {
            steps {
                copyArtifacts(
                    projectName: 'CI-02 Security - Quality Gate',
                    selector: [$class: 'StatusBuildSelector', stable: false],
                    filter: 'admin_microservice/target/*.jar, data_microservice/target/*.jar'
                )
            }
        }

```

Figura 68. Implementación de la etapa Docker build para la construcción y etiquetado de imágenes de los microservicios.

```
        stage('Docker build') {
            steps {
                sh """
                    docker build -t ${DOCKER_NAMESPACE}/adminprb:${IMAGE_TAG} ./admin_microservice
                    docker build -t ${DOCKER_NAMESPACE}/data:${IMAGE_TAG} ./data_microservice

                    docker tag ${DOCKER_NAMESPACE}/adminprb:${IMAGE_TAG} ${DOCKER_NAMESPACE}/adminprb:latest
                    docker tag ${DOCKER_NAMESPACE}/data:${IMAGE_TAG} ${DOCKER_NAMESPACE}/data:latest
                """
            }
        }

```

7.3.3 Imágenes Docker analizadas y validadas dentro del flujo definido - P4.2. Como parte del bloque CD-01 se implementa la etapa de análisis de seguridad de imágenes de contenedor mediante la herramienta Trivy. Esta fase tiene como objetivo identificar vulnerabilidades presentes en las imágenes Docker generadas previamente, antes de su despliegue en el entorno de ejecución, fortaleciendo así la postura de seguridad del sistema.

La ejecución del escaneo se realiza a través de contenedores efímeros de Trivy, lo que permite evitar la instalación directa de la herramienta en el entorno del agente de integración continua. Para ello, se utiliza el comando docker run con la opción --rm, garantizando que el contenedor se elimine automáticamente una vez finalizado el análisis, lo que contribuye a mantener un entorno limpio y reproducible.
Inicialmente, se crea un directorio denominado trivy-reports, el cual se utiliza para almacenar los resultados generados por los escaneos. Este directorio se monta como volumen dentro del contenedor de Trivy mediante la opción -v, permitiendo persistir los reportes fuera del contenedor. Adicionalmente, se monta el socket de Docker (/var/run/docker.sock), lo que permite a Trivy acceder a las imágenes locales construidas durante el pipeline.

En cuanto al formato de salida, se emplea la opción -f table, la cual genera reportes en formato tabular, facilitando su lectura e interpretación por parte de los equipos técnicos. Estos reportes son almacenados en archivos de texto dentro del directorio previamente creado, diferenciando cada microservicio mediante nombres específicos como admin-trivy.txt y data-trivy.txt.

Figura 69. Implementación de la etapa Trivy scan para el análisis de seguridad de imágenes y el archivado de reportes generados.

```
41        stage('Trivy scan') {
42            steps {
43                sh '''
44                    mkdir -p trivy-reports
45
46                    docker run --rm \
47                      -v /var/run/docker.sock:/var/run/docker.sock \
48                      -v "$PWD/trivy-reports:/reports" \
49                      aquasec/trivy:latest image \
50                      --severity HIGH,CRITICAL \
51                      --exit-code 0 \
52                      --format table \
53                      --no-progress \
54                      ${DOCKER_NAMESPACE}/adminprb:${IMAGE_TAG} \
55                      > trivy-reports/admin-trivy.txt 2>&1 || true
56
57                    docker run --rm \
58                      -v /var/run/docker.sock:/var/run/docker.sock \
59                      -v "$PWD/trivy-reports:/reports" \
60                      aquasec/trivy:latest image \
61                      --severity HIGH,CRITICAL \
62                      --exit-code 0 \
63                      --format table \
64                      --no-progress \
65                      ${DOCKER_NAMESPACE}/data:${IMAGE_TAG} \
66                      > trivy-reports/data-trivy.txt 2>&1 || true
67                '''
68            }
69            post {
70                always {
71                    archiveArtifacts artifacts: 'trivy-reports/*.txt', fingerprint: true, allowEmptyArchive: true
72                }
73            }
74        }
```

Una vez finalizado el escaneo de la imagen, el pipeline realiza el proceso de autenticación contra el registro de contenedores utilizando las credenciales previamente configuradas en Jenkins. En esta etapa se ejecuta el proceso de autenticación mediante el comando docker login, el cual permite establecer una sesión segura con el repositorio remoto.

Posteriormente, la imagen generada es publicada en el registro de contenedores mediante la ejecución del comando docker push. Este procedimiento permite almacenar la imagen construida en un repositorio centralizado, desde donde podrá ser recuperada posteriormente para su despliegue en distintos entornos de ejecución.

Figura 70. Credenciales registradas en Jenkins para la autenticación con Docker Hub durante la publicación de imágenes.

Figura 71. Implementación de la etapa Docker push para la autenticación y publicación de imágenes en el registro de contenedores.

```
73        stage('Docker push') {
74            steps {
75                withCredentials([usernamePassword(credentialsId: '2', usernameVariable: 'DOCKER_USER', passwordVariable: 'DOCKER_PASS')]) {
76                    sh """
77                        echo \$DOCKER_PASS | docker login -u \$DOCKER_USER --password-stdin
78
79                        docker push ${DOCKER_NAMESPACE}/adminprb:${IMAGE_TAG}
80                        docker push ${DOCKER_NAMESPACE}/data:${IMAGE_TAG}
81                        docker push ${DOCKER_NAMESPACE}/adminprb:latest
82                        docker push ${DOCKER_NAMESPACE}/data:latest
83                    """
84                }
85            }
86        }
```

El despliegue de la aplicación se realiza de manera progresiva utilizando Docker Compose, con el objetivo de garantizar un arranque ordenado y controlado de los diferentes componentes del sistema. En una primera fase, se despliegan los servicios de infraestructura, incluyendo bases de datos y componentes de mensajería, tales como db, mongo, emqx e influxdb. Estos servicios constituyen la base sobre la cual operan los microservicios principales, por lo que su disponibilidad es crítica.

Posteriormente, se introduce un tiempo de espera controlado que permite la correcta inicialización de estas dependencias. Una vez completada esta fase, se despliegan los servicios principales del sistema, correspondientes a los microservicios admin y data, seguidos de una nueva pausa para asegurar su disponibilidad.

Finalmente, se realiza el despliegue de los servicios complementarios, como gateway, frontend y grafana, los cuales corresponden a la capa de acceso y monitoreo del sistema.

Este enfoque escalonado permite evitar problemas de dependencia entre servicios y reduce la probabilidad de fallos durante el arranque.

Figura 72. Implementación de las etapas de despliegue progresivo de dependencias, microservicios principales y servicios complementarios mediante Docker Compose.

```
88          stage('Deploy dependencies') {
89              steps {
90                  sh 'docker compose up -d db mongo emqx influxdb'
91              }
92          }
93
94          stage('Wait dependencies') {
95              steps {
96                  sh 'sleep 20'
97              }
98          }
99
100         stage('Deploy admin and data') {
101             steps {
102                 sh 'docker compose up -d admin data'
103             }
104         }
105
106         stage('Wait core services') {
107             steps {
108                 sh 'sleep 20'
109             }
110         }
111
112         stage('Deploy remaining services') {
113             steps {
114                 sh 'docker compose up -d gateway frontend grafana'
115             }
116         }
117
```

7.3.4 Smoke Test. Una vez desplegados todos los servicios, se ejecuta una fase de Smoke Test con el propósito de verificar que el sistema se encuentra operativo. Esta validación se realiza en dos niveles: infraestructura y aplicación.

En primer lugar, se comprueba que todos los contenedores se encuentren en estado de ejecución (running) mediante comandos de Docker Compose. Esta verificación permite detectar fallos en el despliegue de servicios específicos, deteniendo el pipeline en caso de que alguno no se encuentre activo.

En segundo lugar, se realizan pruebas de conectividad HTTP hacia los servicios expuestos, como data, gateway y frontend. Para ello, se envían solicitudes utilizando curl y se evalúa el código de respuesta HTTP. Si un servicio no responde (código 000), se considera como un fallo crítico y se interrumpe la ejecución del pipeline.

Este enfoque permite validar de forma rápida y automatizada que los componentes esenciales del sistema están disponibles y funcionando correctamente, alineándose con el principio de “fail fast” y fortaleciendo la confiabilidad del proceso de despliegue continuo.

Figura 73. Implementación de la etapa Smoke Test para la verificación del estado de contenedores y la conectividad HTTP de los servicios desplegados.

```
        stage('Smoke Test') {
            steps {
                sh '''
                    set -e

                    HOST_IP="192.168.1.141"

                    echo "=== Estado de servicios ==="
                    docker compose ps

                    for svc in db mongo emqx influxdb admin data gateway frontend grafana; do
                      docker compose ps --services --filter status=running | grep -x "$svc" >/dev/null || {
                        echo "FAIL: $svc no está corriendo"
                        exit 1
                      }
                    done

                    check_http() {
                      NAME="$1"
                      URL="$2"
                      CODE=$(curl -s -o /dev/null -w "%{http_code}" "$URL" || true)

                      if [ "$CODE" = "000" ]; then
                        echo "FAIL: $NAME no responde en $URL"
                        exit 1
                      else
                        echo "PASS: $NAME responde con HTTP $CODE en $URL"
                      fi
                    }

                    check_http "data" "http://${HOST_IP}:8082/"
                    check_http "gateway" "http://${HOST_IP}:8080/"
                    check_http "frontend" "http://${HOST_IP}:4000/"
                '''
            }
        }
```

Figura 74. PARTE 1 - Ejecución del bloque CD-01 Containerization - Deployment con resultado del Smoke Test y comportamiento de las etapas de despliegue.

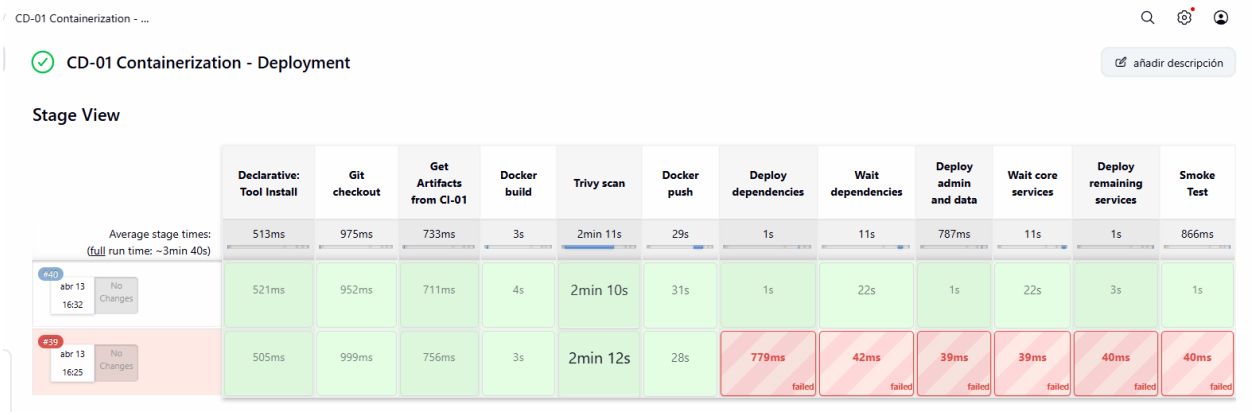
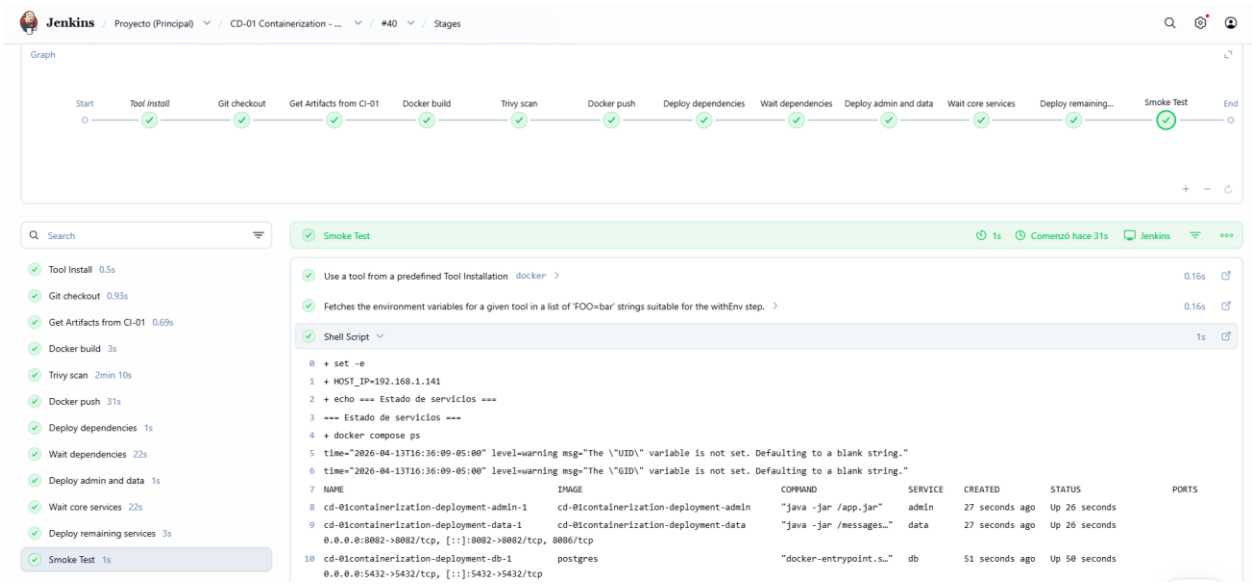


Figura 75. PARTE 2 - Ejecución del bloque CD-01 Containerization - Deployment con resultado del Smoke Test y comportamiento de las etapas de despliegue.



La ejecución exitosa de todas estas etapas confirma que el pipeline se encuentra completamente funcional, permitiendo que tanto el componente admin como el de data sean validados y analizados desde el punto de vista de seguridad, además toda la arquitectura es desplegada garantizando así la disponibilidad del servicio completo. De esta manera se garantiza un entorno de pruebas validado y operativo en el cual se

establecen controles de seguridad sobre los microservicios de gestión y administración en Smart Campus UIS.

Figura 76. PARTE 1 - Vista consolidada del Full Pipeline por bloques CI-01, CI-02 y CD-01 con tiempos promedio de ejecución.

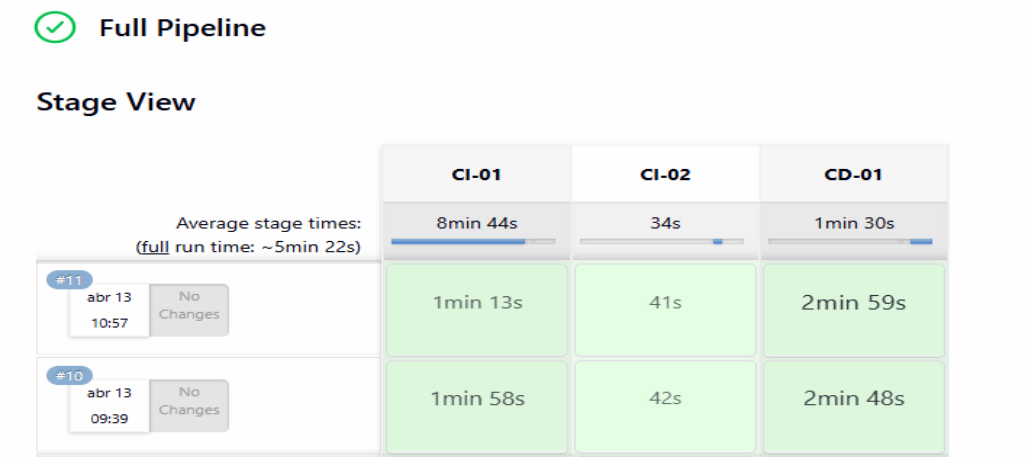


Figura 77. PARTE 2 - Vista consolidada del Full Pipeline por bloques CI-01, CI-02 y CD-01 con tiempos promedio de ejecución.

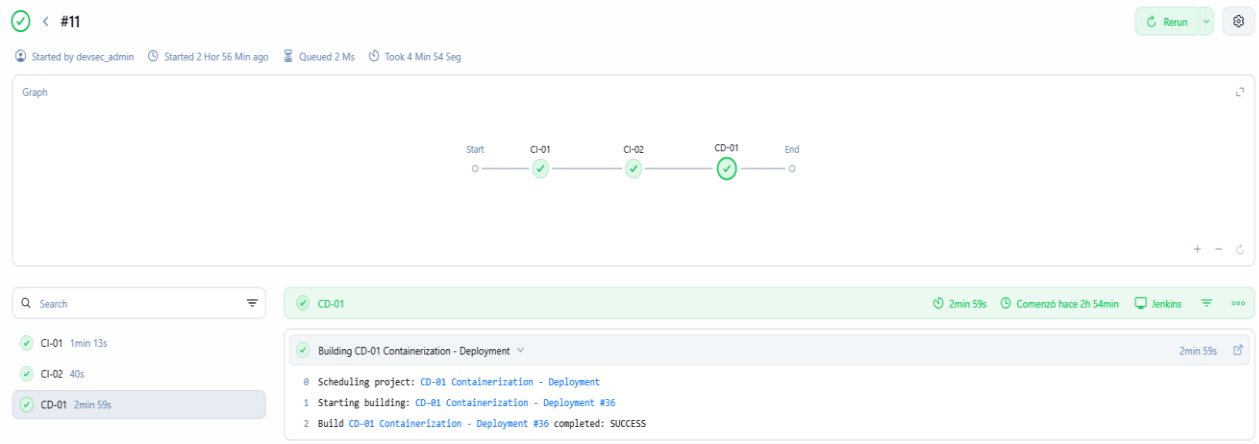
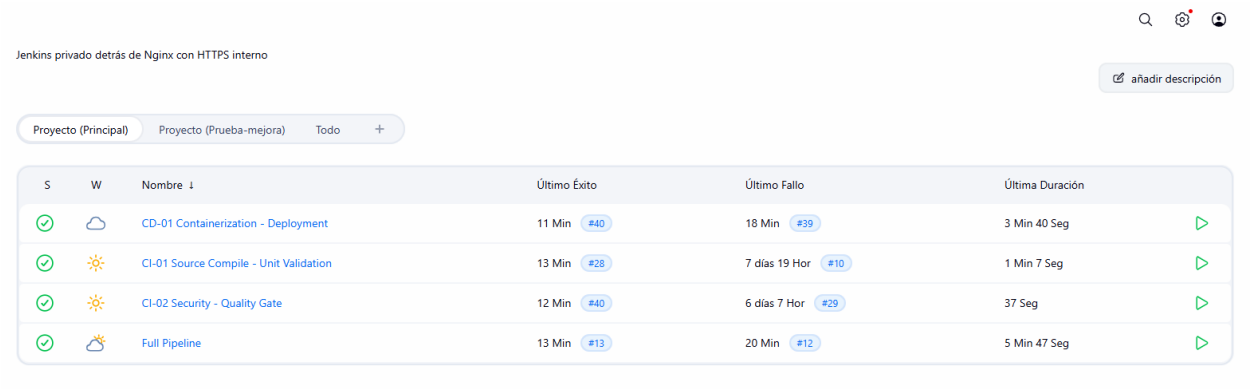


Figura 78. Panel general de ejecución de pipelines configurados para Smart Campus UIS en Jenkins.



A partir de la implementación descrita en este capítulo, se logró materializar técnicamente la arquitectura DevSecOps propuesta para Smart Campus UIS, integrando de forma operativa los mecanismos de compilación, pruebas unitarias, análisis SAST, análisis SCA, construcción de imágenes, escaneo de seguridad y despliegue controlado en entorno de prueba. La configuración realizada demuestra que el flujo definido no se mantuvo únicamente en el nivel de diseño, sino que fue llevado a una solución funcional soportada por herramientas concretas y articuladas dentro de un mismo proceso automatizado. En este sentido, el capítulo evidencia que la propuesta desarrollada permite fortalecer la trazabilidad, la repetibilidad y el control técnico del ciclo de vida del software, constituyendo una base efectiva para su validación mediante los resultados presentados en el capítulo siguiente.

8. FASE 5: VALIDACIÓN Y CIERRE DEL PIPELINE DEVSECOPS

En este capítulo se presentan y analizan los resultados obtenidos durante la ejecución del pipeline DevSecOps implementado para la plataforma Smart Campus UIS. El análisis se organiza de acuerdo con los principales mecanismos de validación incorporados en el flujo: pruebas unitarias, análisis estático del código fuente, análisis de dependencias, escaneo de imágenes de contenedor y pruebas básicas posteriores al despliegue.

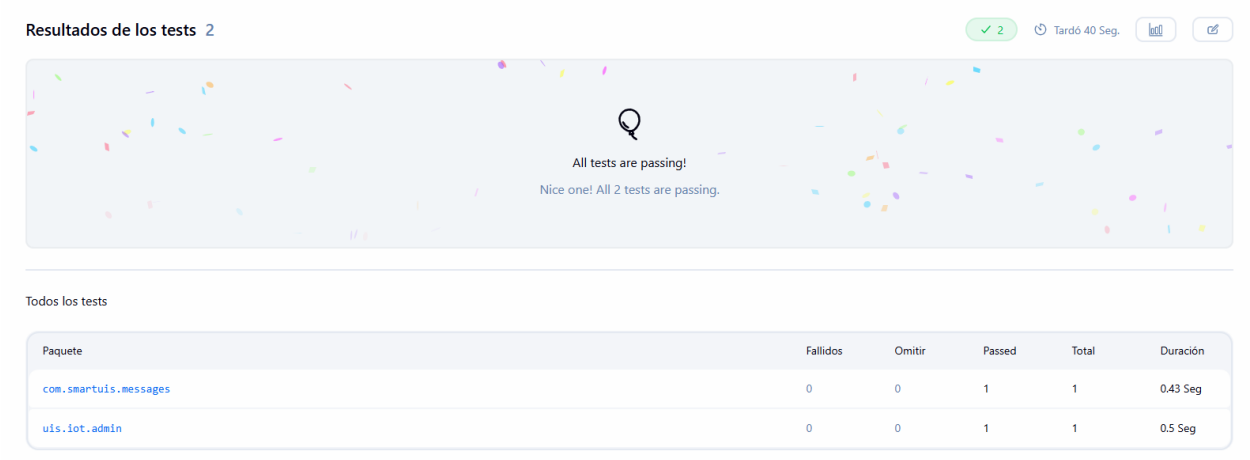
El propósito de este capítulo no es únicamente evidenciar la ejecución técnica de las herramientas integradas, sino también valorar en qué medida los resultados obtenidos permiten responder a los objetivos del proyecto, particularmente en lo relacionado con la automatización del flujo CI/CD, la incorporación temprana de controles de seguridad y la trazabilidad del proceso de construcción y despliegue del software.

Los resultados se interpretan a partir de la evidencia generada por el propio pipeline, incluyendo reportes, logs, métricas y estados de ejecución. De esta manera, el capítulo permite establecer una relación directa entre el diseño propuesto, la implementación realizada y el comportamiento observado durante la validación del flujo DevSecOps en el entorno de pruebas definido para el proyecto.

8.1 ENTORNO DE PRUEBA VALIDADO Y OPERATIVO CON REPORTE DE RESULTADOS DE VALIDACIÓN. P5.1 - P5.2

8.1.1 Tests unitarios. Como parte de la validación funcional inicial del pipeline DevSecOps, se ejecutaron pruebas automatizadas sobre los dos microservicios principales considerados dentro del alcance del proyecto: el microservicio de datos, identificado en los resultados como com.smartuis.messages, y el microservicio administrativo, identificado como uis.iot.admin. Los resultados obtenidos evidencian que ambos componentes superaron satisfactoriamente la fase de pruebas, registrando en cada caso 1 prueba ejecutada, 1 prueba aprobada, 0 fallidas y 0 omitidas.

Figura 79. Resultado global de las pruebas unitarias ejecutadas sobre los microservicios priorizados del proyecto.



En el caso del microservicio de datos, la prueba reportada corresponde a la clase MessagesApplicationTests, en la cual se ejecutó satisfactoriamente el caso contextLoads, con un tiempo aproximado de 0.43 segundos. De manera similar, para el microservicio administrativo se registró una ejecución exitosa con una duración aproximada de 0.5 segundos. Estos resultados permiten verificar que, al menos en el nivel de validación implementado en esta etapa, ambos servicios pueden cargar correctamente su contexto de aplicación y completar su inicialización básica sin errores.

Figura 80. Resultado de las pruebas unitarias del microservicio de datos com.smartuis.messages.

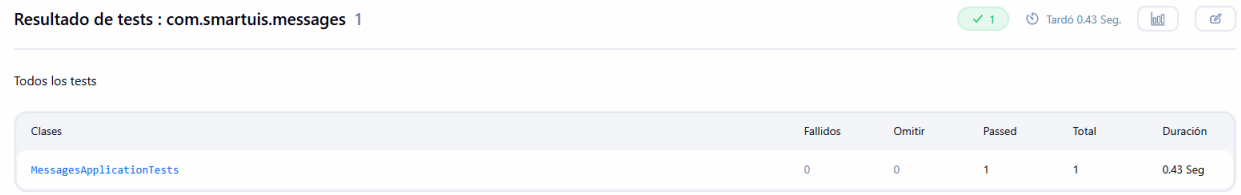
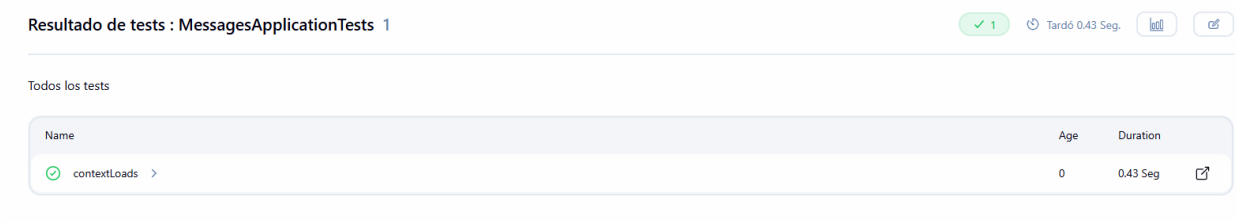


Figura 81. Resultado del caso de prueba contextLoads en MessagesApplicationTests.



Adicionalmente, el historial de ejecuciones mostrado para ambos microservicios permite observar un comportamiento estable en builds consecutivos, sin presencia de fallos ni pruebas omitidas en los registros visualizados. En el caso de com.smartuis.messages, los tiempos de ejecución se mantuvieron en un rango cercano entre 350 ms y 435 ms, mientras que para uis.iot.admin los tiempos observados se ubicaron aproximadamente entre 444 ms y 519 ms. Esto evidencia consistencia en la respuesta de las pruebas a lo largo de distintas ejecuciones del pipeline y refuerza la estabilidad básica del proceso de validación automatizada.

Figura 82. Tendencia histórica de duración y estado de las pruebas unitarias del microservicio com.smartuis.messages.

Figura 83. Tendencia histórica de duración y estado de las pruebas unitarias del microservicio uis.iot.admin.

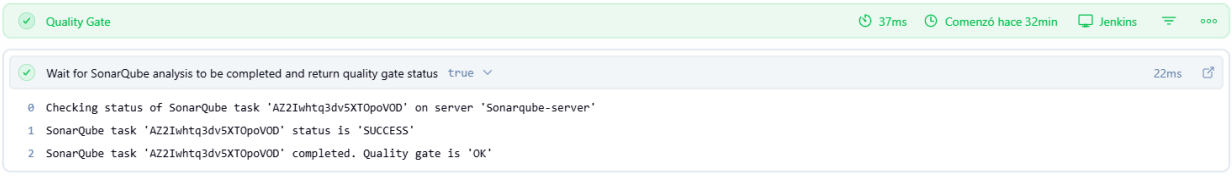
Desde la perspectiva del proceso de desarrollo, estos resultados aportan beneficios relevantes para el equipo encargado del software. En primer lugar, proporcionan una validación temprana sobre el estado básico de los microservicios después de cada cambio hecho en el repositorio, reduciendo la probabilidad de que errores de inicialización o configuraciones incorrectas avancen hacia etapas posteriores del pipeline. En segundo lugar, permiten ofrecer retroalimentación rápida a los desarrolladores, quienes pueden identificar de manera inmediata si una modificación afecta la capacidad de arranque del servicio. Esto fortalece la confianza en la integración continua, mejora la trazabilidad de los cambios y disminuye el tiempo requerido para detectar fallos básicos dentro del ciclo de desarrollo.

Sin embargo, también debe señalarse que el conjunto de pruebas ejecutado en esta fase es aún reducido, por lo que su aporte se orienta principalmente a validar la carga del contexto y la estabilidad básica de los microservicios, más que a cubrir de forma exhaustiva la lógica funcional interna del sistema. Aun así, dentro del alcance definido

para el proyecto, los resultados obtenidos demuestran que el pipeline logra integrar exitosamente una validación automatizada inicial, aportando evidencia de funcionamiento, estabilidad y trazabilidad sobre los componentes analizados.

8.1.2 Sast (Static Application Security Testing). Como parte del bloque de validación de seguridad y calidad del pipeline, se ejecutó el análisis estático del código fuente mediante SonarQube, integrando sus resultados al proceso automatizado de Jenkins. Dentro del diseño del proyecto, esta etapa cumple la función de identificar defectos, vulnerabilidades potenciales, code smells y métricas de mantenibilidad, mientras que el Quality Gate actúa como mecanismo de decisión para determinar si el software puede continuar hacia etapas posteriores del flujo. En la evidencia obtenida desde Jenkins se observa que la tarea de análisis finalizó con estado SUCCESS y que el Quality Gate fue reportado como OK, indicando que, bajo los criterios configurados para el proyecto, los componentes analizados cumplieron con las condiciones mínimas para avanzar en el pipeline.

Figura 84. Resultado satisfactorio de la etapa Quality Gate en Jenkins tras el análisis de SonarQube.



En el caso del componente administrativo, identificado en SonarQube como prb-admin, el tablero general reportó 0 bugs, 0 vulnerabilities, 6 security hotspots, 265 code smells, una deuda técnica estimada de 1 día y 7 horas, una duplicación de 3.5 % sobre aproximadamente 2.1 mil líneas y 11 bloques duplicados. Adicionalmente, el panel mostró calificaciones A en confiabilidad, seguridad y mantenibilidad, mientras que el apartado de Security Review se ubicó en E, debido a que los security hotspots permanecían sin revisión. Este resultado sugiere que, aunque no se detectaron

vulnerabilidades confirmadas ni errores clasificados como bugs bajo las reglas aplicadas, sí existen oportunidades importantes de mejora relacionadas con mantenibilidad, convenciones de codificación y revisión de configuraciones sensibles. Entre los hallazgos mostrados se observan, por ejemplo, advertencias asociadas al uso de una imagen base que ejecuta el contenedor con usuario root y a la presencia de características de depuración que deberían desactivarse en entornos productivos.

Figura 85. Panel general de resultados SAST del componente administrativo prb-admin en SonarQube.

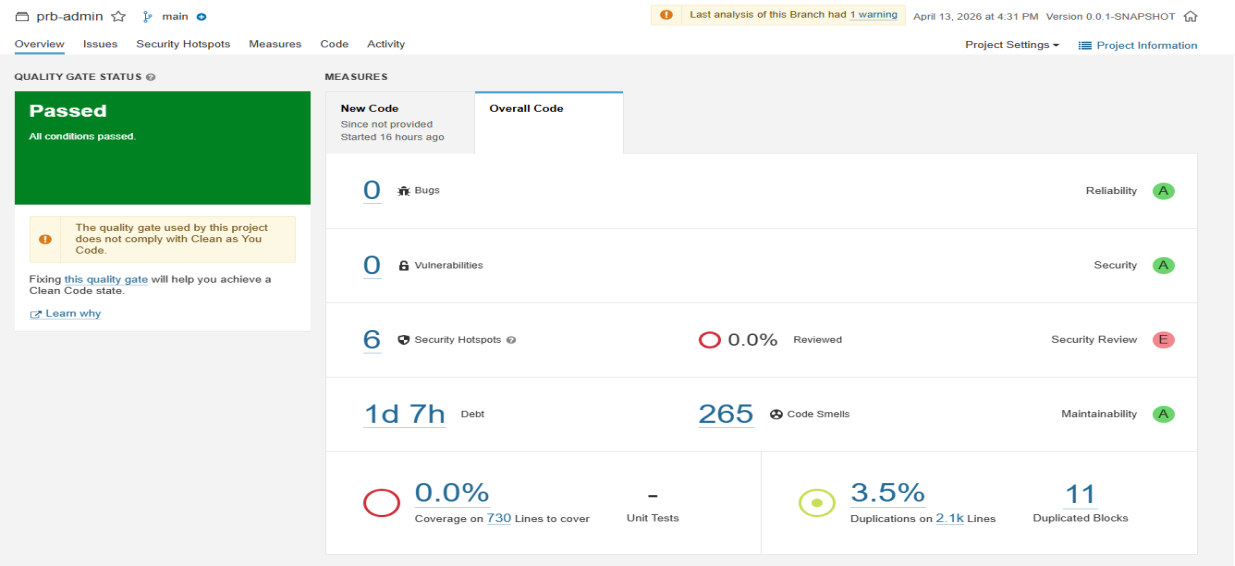


Figura 86. Detalle de hallazgos de mantenibilidad y code smells del componente administrativo prb-admin.

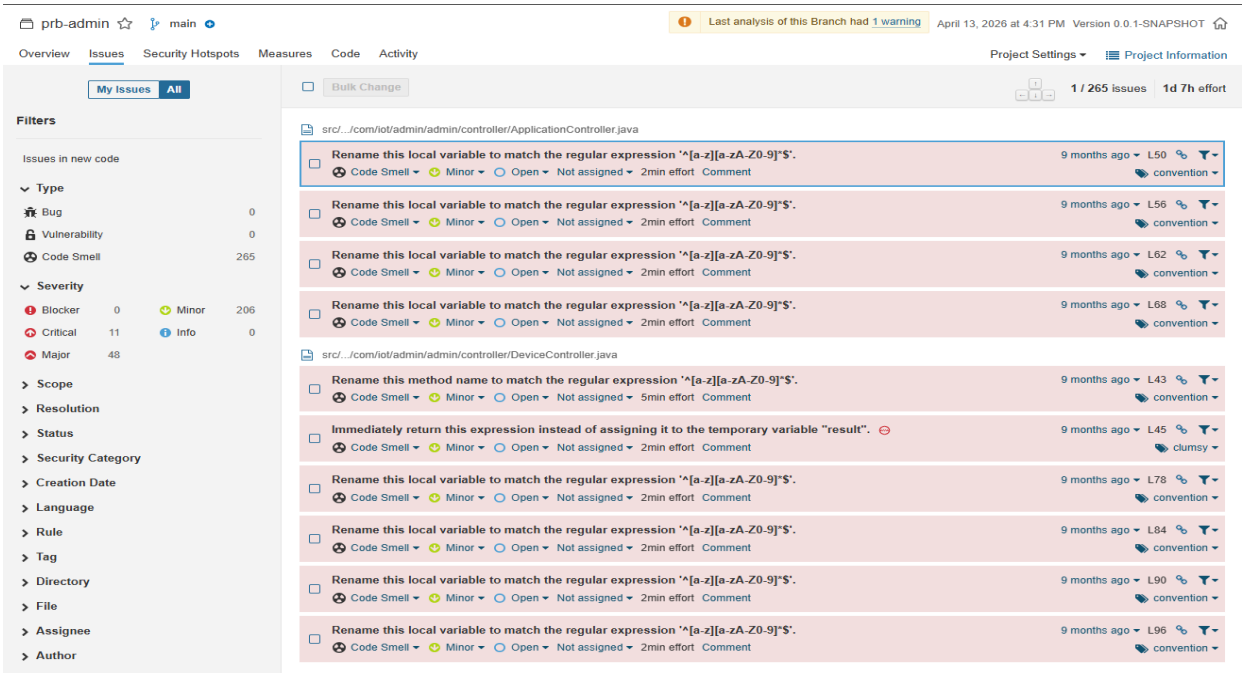
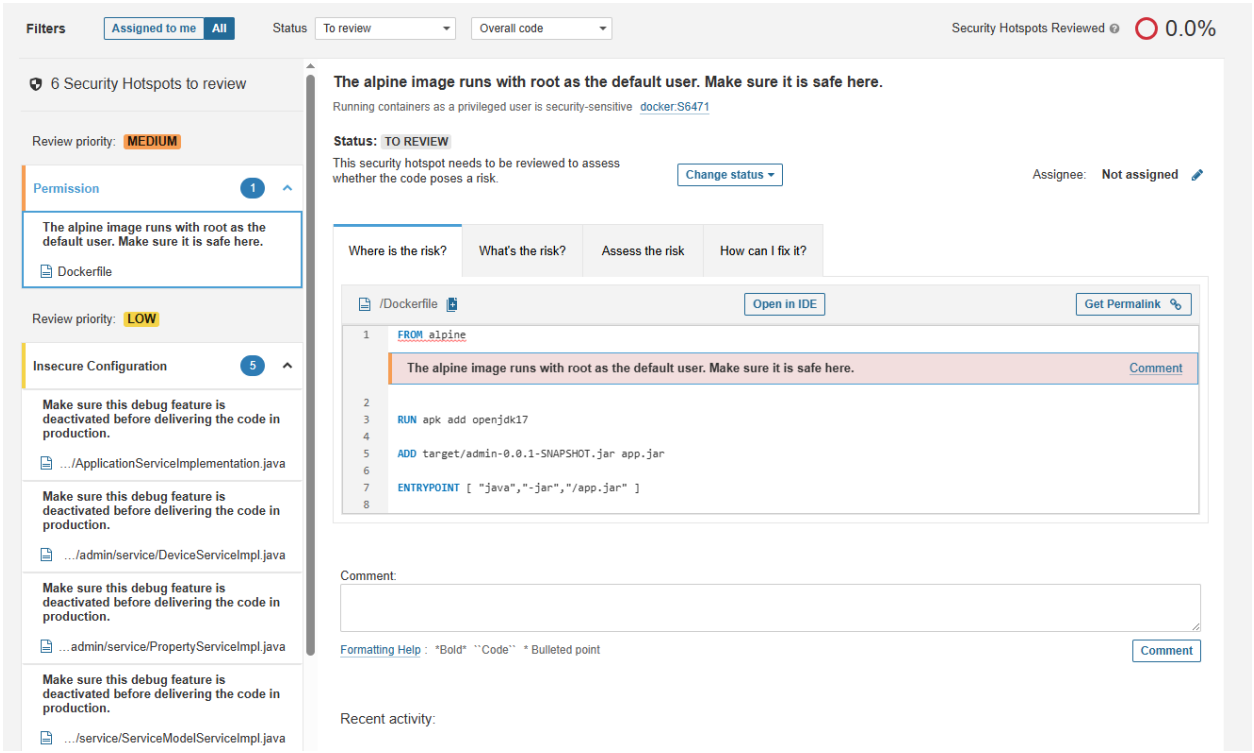


Figura 87. Detalle de un security hotspot asociado al componente administrativo prb-admin en SonarQube.



Para el componente de datos, identificado como pruebadata, los resultados reflejaron un estado más favorable en términos comparativos. El tablero reportó 0 bugs, 0 vulnerabilities, 1 security hotspot, 30 code smells, una deuda técnica estimada de 3 horas y 8 minutos, 0 % de duplicación y 1 prueba unitaria registrada. De igual forma, el panel mostró calificaciones A en confiabilidad, seguridad y mantenibilidad, mientras que el apartado de Security Review apareció en C, acompañado de un porcentaje de revisión del 50 %. En el listado de hallazgos se identifican observaciones relacionadas con convenciones de nombres, fragmentos de código comentado que deberían eliminarse, uso de System.out o System.err en lugar de mecanismos formales de registro y recomendaciones para declarar constantes de forma más controlada. También se evidenció un *security hotspot* asociado a la permanencia de funciones de depuración activas en código que podría llegar a producción. En comparación con el componente administrativo, este microservicio presenta una menor cantidad de observaciones y una deuda técnica considerablemente más baja, lo que sugiere un mejor estado relativo de mantenibilidad.

Figura 88. Panel general de resultados SAST del componente de datos pruebadata en SonarQube.

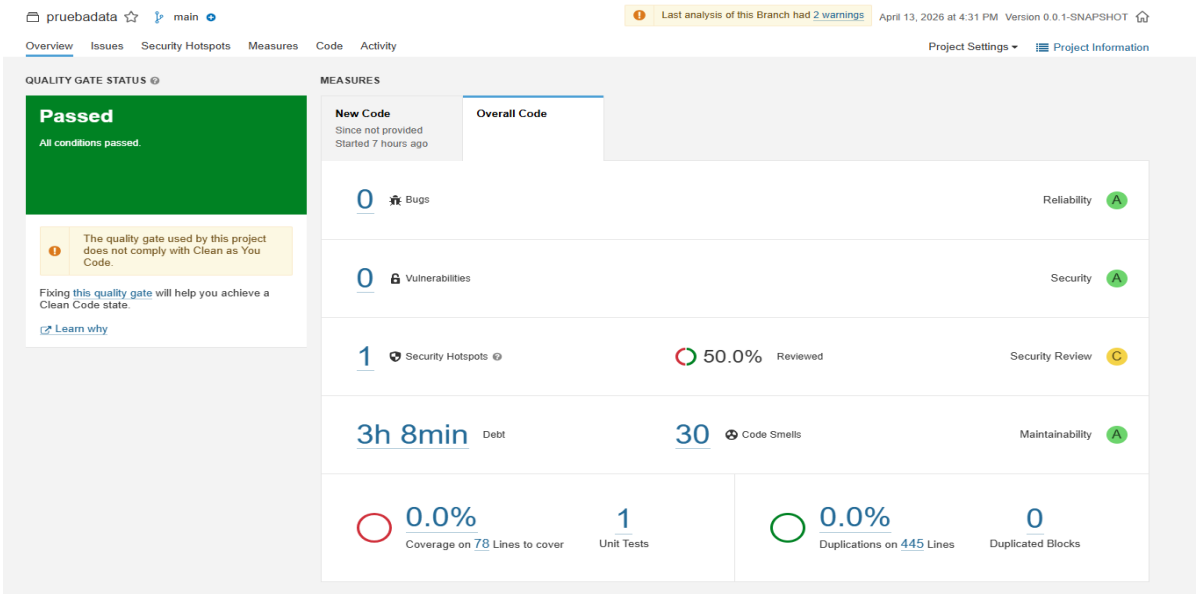


Figura 89. Detalle de hallazgos de mantenibilidad y code smells del componente de datos pruebadata.

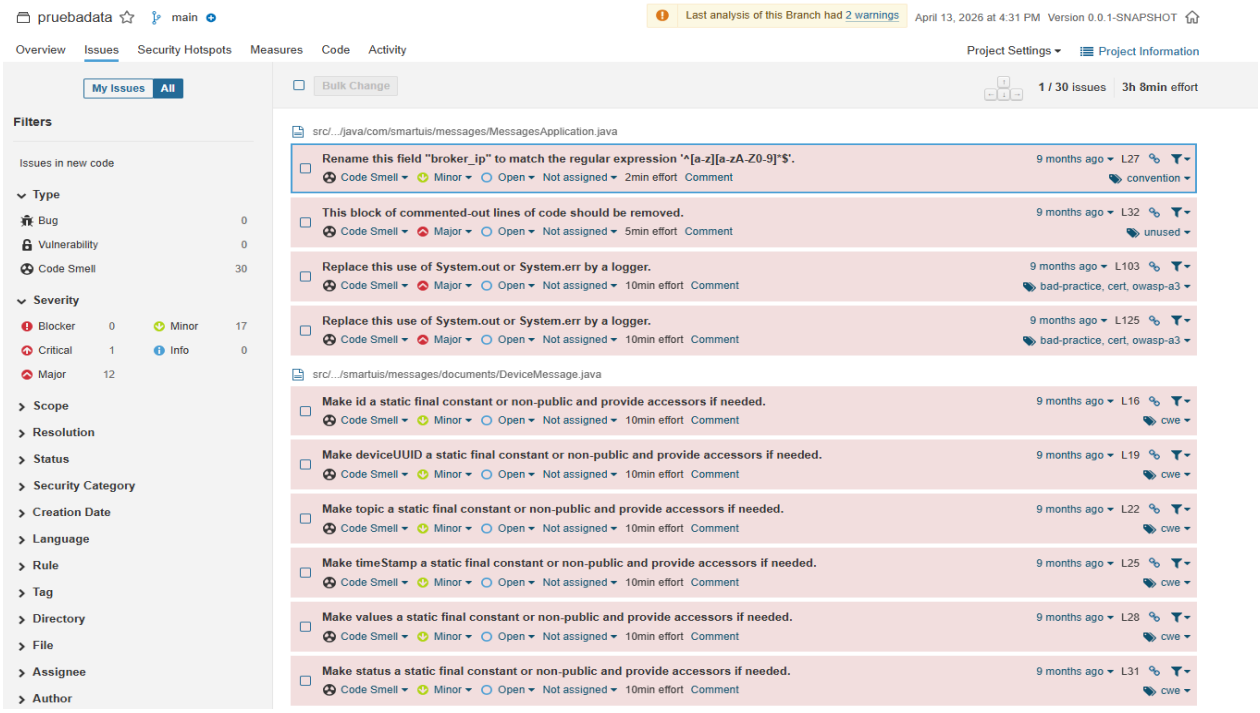
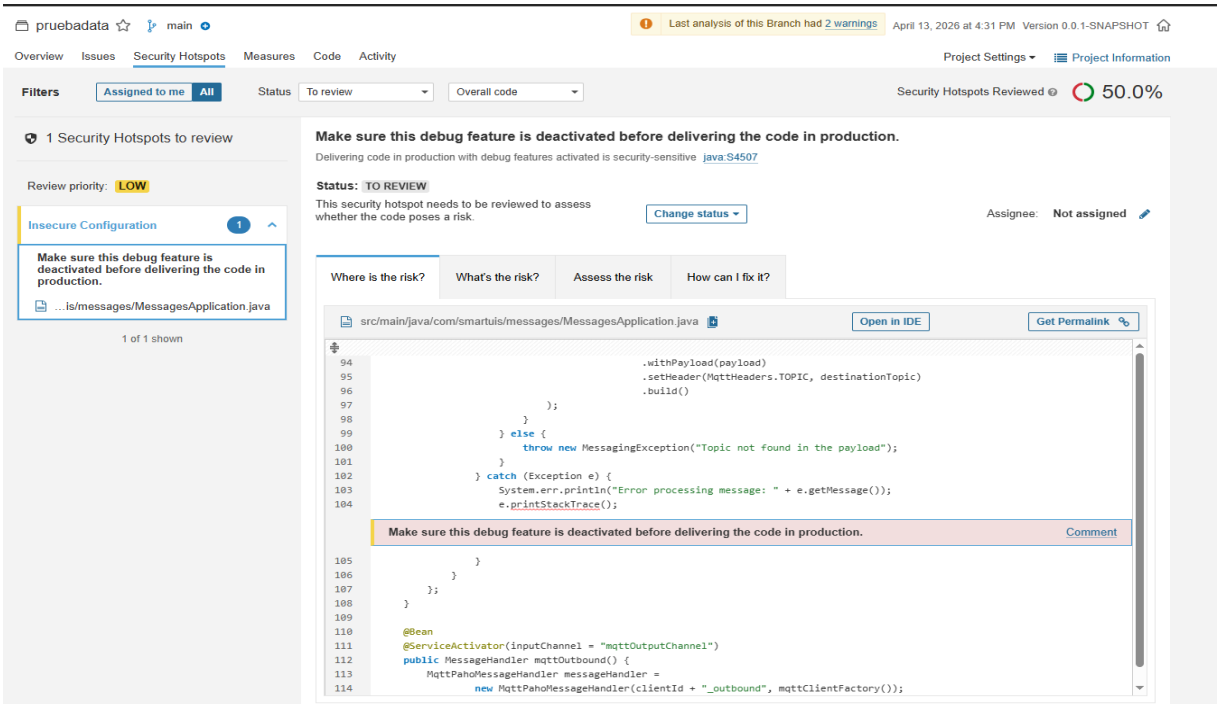


Figura 90. Detalle del security hotspot identificado en el componente de datos pruebadata.



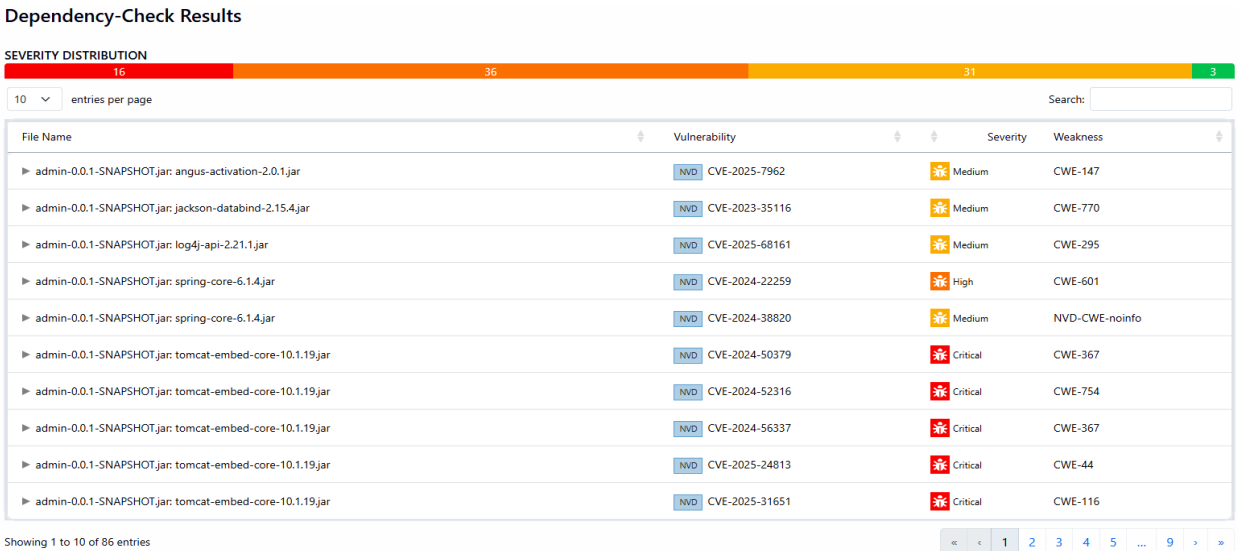
Desde una perspectiva de interpretación, estos resultados muestran que el análisis SAST no debe entenderse únicamente como una búsqueda de vulnerabilidades confirmadas, sino como un mecanismo más amplio de control de calidad y seguridad temprana. El hecho de que ambos componentes hayan superado el Quality Gate no implica que el código se encuentre completamente libre de aspectos a corregir, sino que, según la configuración aplicada, cumple con el umbral mínimo establecido para continuar dentro del pipeline. De hecho, la propia interfaz de SonarQube advierte que el proyecto no se ajusta completamente al enfoque Clean as You Code, lo que refuerza la necesidad de revisar y remediar progresivamente los hallazgos reportados.

Para el equipo de desarrollo, este tipo de resultados aporta un valor importante en el ciclo de trabajo. En primer lugar, ofrece retroalimentación temprana e inmediata sobre el estado del código después de cada integración, permitiendo detectar problemas de mantenibilidad, configuraciones inseguras y prácticas de codificación mejorables antes de que el software avance a fases de empaquetado o despliegue. En segundo lugar, facilita la priorización técnica, ya que permite identificar cuál de los componentes requiere mayor atención correctiva; en este caso, el componente administrativo presenta una carga significativamente mayor de code smells, security hotspots y deuda técnica que el componente de datos. Finalmente, fortalece la trazabilidad y la toma de decisiones, dado que los desarrolladores y responsables del pipeline pueden relacionar cada análisis con una ejecución concreta, revisar evidencia puntual de los hallazgos y usar el Quality Gate como un criterio automático de continuidad dentro del flujo DevSecOps.

8.1.3 Sca (Software Composition Analysis). Como parte del bloque de validación de seguridad del pipeline, se ejecutó el análisis de composición de software (SCA) mediante OWASP Dependency-Check, con el propósito de identificar vulnerabilidades conocidas en bibliotecas y dependencias de terceros utilizadas por el proyecto. Esta etapa resulta especialmente importante dentro del enfoque DevSecOps, ya que permite detectar riesgos que no provienen directamente del código desarrollado por el equipo, sino de componentes externos incorporados durante la construcción del software.

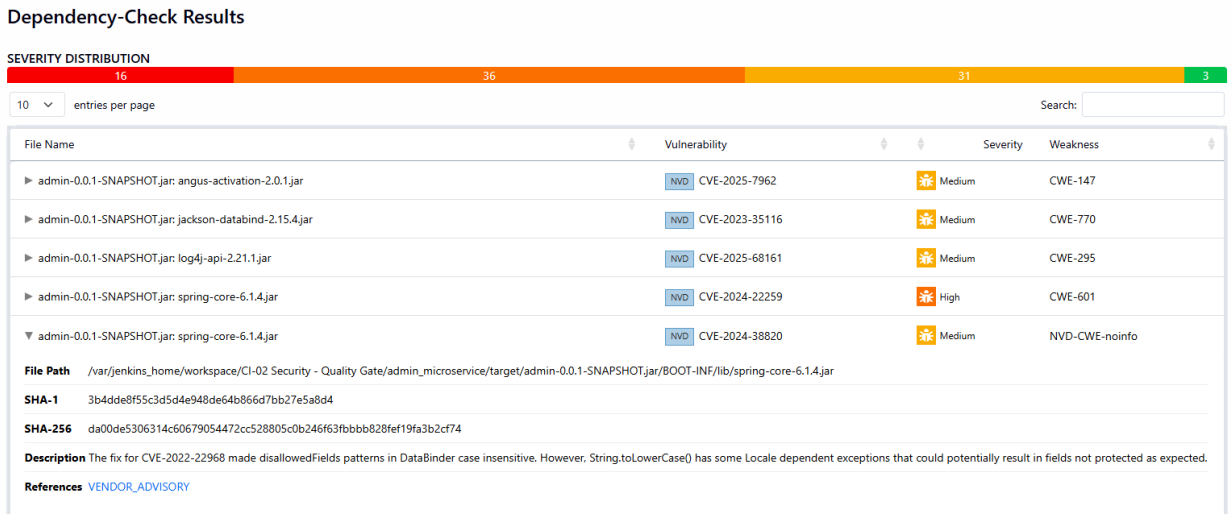
De acuerdo con la evidencia obtenida, el análisis realizado sobre los artefactos admin-0.0.1-SNAPSHOT.jar y messages-0.0.1-SNAPSHOT.jar reportaron un total de 86 vulnerabilidades asociadas a dependencias incluidas en el componente evaluado. La distribución por severidad mostró 16 vulnerabilidades críticas, 36 altas, 31 medias y 3 bajas, lo que evidencia una concentración importante de hallazgos en niveles de riesgo elevados. Entre las dependencias reportadas se observan componentes como spring-core, tomcat-embed-core, jackson-databind, log4j-api y angus-activation, lo que permite identificar con mayor precisión los puntos del ecosistema de librerías que requieren atención prioritaria.

Figura 91. Resultado general del análisis SCA de los artefactos admin mediante OWASP Dependency-Check.



Adicionalmente, el reporte detallado evidencia que la herramienta no solo identifica la dependencia afectada y la vulnerabilidad asociada, sino que también proporciona información complementaria útil para su análisis, como la severidad, la debilidad relacionada (CWE), la ruta del archivo dentro del artefacto, valores de integridad como SHA-1 y SHA-256, y referencias técnicas para ampliar la revisión del hallazgo. Un ejemplo visible en la evidencia corresponde a la dependencia spring-core-6.1.4.jar, vinculada al CVE-2024-38820, clasificado con severidad media, lo que demuestra la capacidad de la herramienta para ofrecer trazabilidad técnica sobre cada vulnerabilidad detectada.

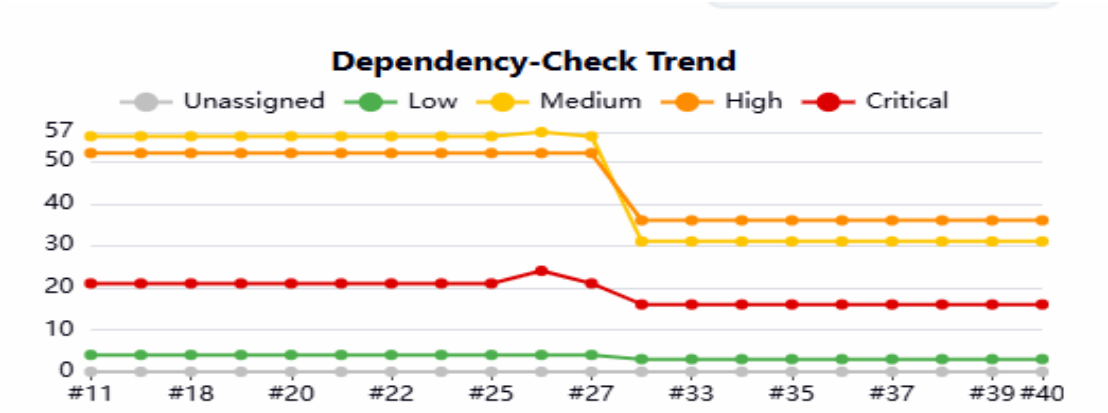
Figura 92. Detalle técnico de una vulnerabilidad identificada en dependencias del artefacto analizado por OWASP Dependency-Check.



El comportamiento histórico mostrado en la gráfica de tendencia permite observar que, en ejecuciones previas del pipeline, el número de vulnerabilidades se mantenía en valores más altos, y que a partir de builds posteriores se presentó una reducción visible en varias categorías de severidad. Aunque la cantidad de hallazgos sigue siendo considerable, esta tendencia sugiere que el proceso de revisión y ajuste de dependencias produjo una mejora parcial en el estado de seguridad del componente analizado. En otras palabras, el análisis SCA no solo permite detectar vulnerabilidades en un punto específico

del tiempo, sino también hacer seguimiento a la evolución del riesgo asociado a las dependencias a lo largo de distintas ejecuciones del pipeline.

Figura 93. Tendencia histórica de vulnerabilidades detectadas por OWASP Dependency-Check en ejecuciones del pipeline.

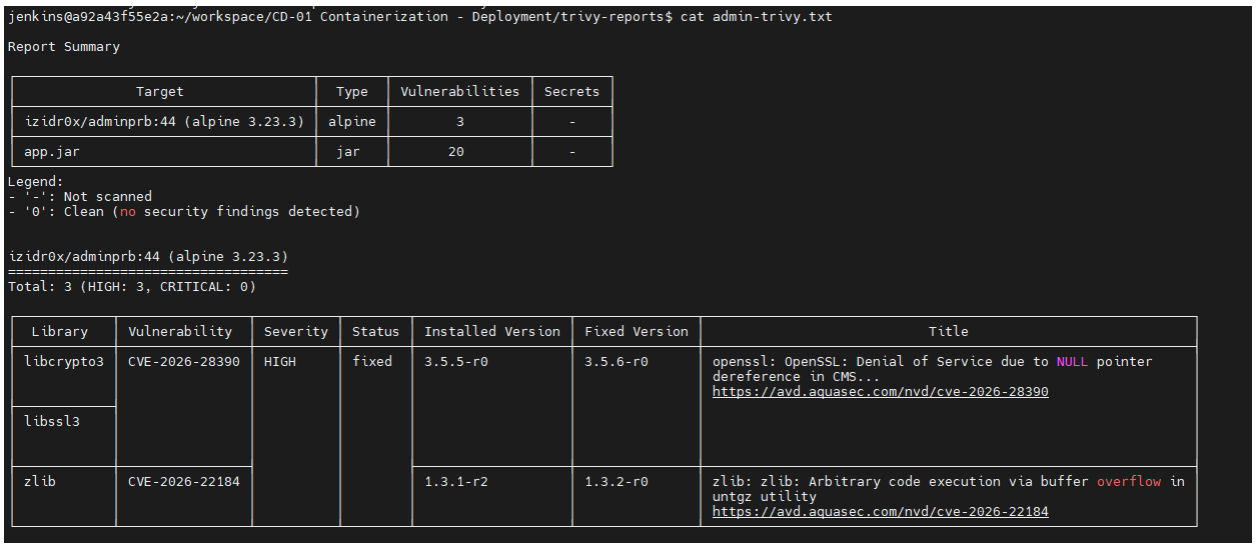


No obstante, los resultados también muestran que la superación de etapas previas del pipeline no implica necesariamente una condición de seguridad completa sobre los componentes de terceros utilizados por la aplicación. Por el contrario, el análisis SCA evidencia que la seguridad del software depende en buena medida del estado de sus dependencias, por lo que esta fase se convierte en un control complementario indispensable frente al análisis estático del código fuente. Dentro del alcance del proyecto, esta evidencia confirma que el pipeline implementado logra incorporar de manera efectiva una validación automatizada sobre la composición del software, generando información útil para la identificación, priorización y seguimiento de vulnerabilidades en dependencias externas.

8.1.4 Escaneo de imágenes. Como parte del bloque de contenerización y despliegue del pipeline DevSecOps, se ejecutó el escaneo de seguridad de imágenes mediante Trivy, con el propósito de identificar inicialmente vulnerabilidades críticas y altas presentes tanto en la imagen base como en los componentes empaquetados dentro del artefacto desplegable. Esta etapa complementa los controles previos de análisis estático y de dependencias, ya que permite evaluar la seguridad del contenedor resultante antes de su publicación y posterior despliegue en el entorno de prueba.

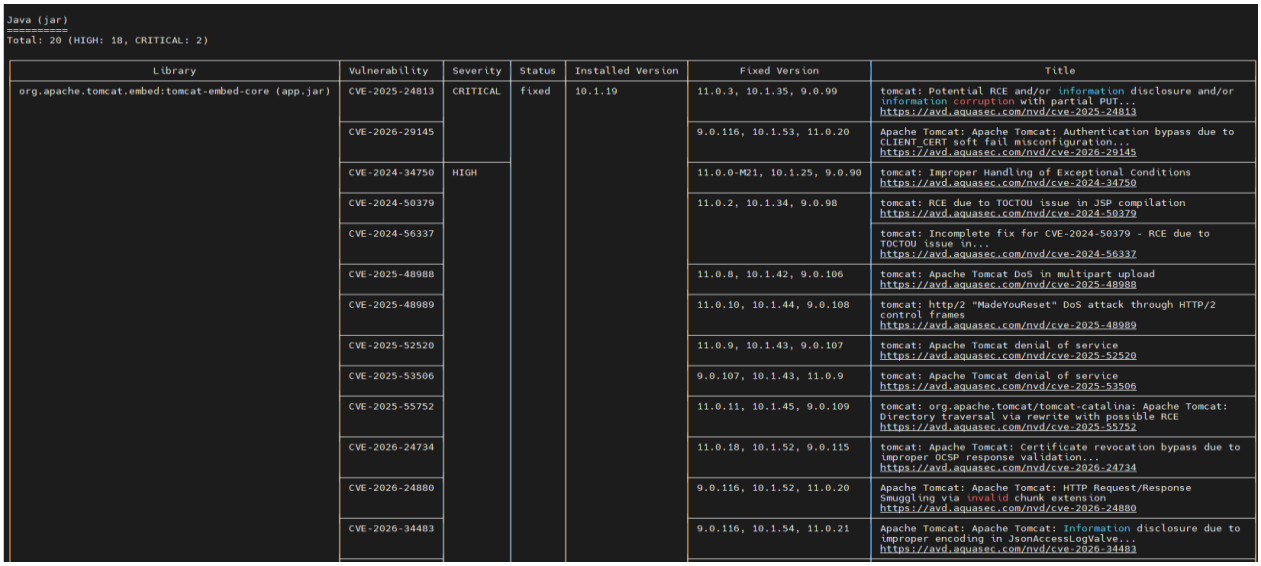
En el caso del componente administrativo, el reporte generado por Trivy mostró dos niveles de análisis dentro de la imagen evaluada. Por una parte, la imagen base Alpine 3.23.3 reportó 3 vulnerabilidades, todas clasificadas con severidad alta, mientras que el archivo app.jar incorporado en la imagen presentó 20 vulnerabilidades, distribuidas en 18 altas y 2 críticas. Esto evidencia que el mayor volumen de hallazgos no se concentra en la base del contenedor, sino en las bibliotecas empaquetadas dentro de la aplicación. Entre los componentes afectados se observa una presencia importante de vulnerabilidades asociadas a tomcat-embed-core, además de hallazgos en bibliotecas del sistema como libcrypto3, libssl3 y zlib.

Figura 94. Reporte resumido de Trivy para la imagen izidr0x/adminprb:44.



```
jenkins@a92a43f55e2a:~/workspace/CD-01 Containerization - Deployment/trivy-reports$ cat admin-trivy.txt

Report Summary

┌──────────────────────────────────┬────────┬─────────────────┬─────────┐
│              Target              │  Type  │ Vulnerabilities │ Secrets │
├──────────────────────────────────┼────────┼─────────────────┼─────────┤
│ izidr0x/adminprb:44 (alpine 3.23.3) │ alpine │        3        │    -    │
├──────────────────────────────────┼────────┼─────────────────┼─────────┤
│ app.jar                          │  jar   │       20        │    -    │
└──────────────────────────────────┴────────┴─────────────────┴─────────┘
Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)


izidr0x/adminprb:44 (alpine 3.23.3)
==================================
Total: 3 (HIGH: 3, CRITICAL: 0)

┌─────────────┬────────────────┬──────────┬────────┬───────────────────┬───────────────┬──────────────────────────────────────────────────────────────┐
│   Library   │ Vulnerability  │ Severity │ Status │ Installed Version │ Fixed Version │                            Title                             │
├─────────────┼────────────────┼──────────┼────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ libcrypto3  │ CVE-2026-28390 │ HIGH     │ fixed  │ 3.5.5-r0          │ 3.5.6-r0      │ openssl: OpenSSL: Denial of Service due to NULL pointer      │
│             │                │          │        │                   │               │ dereference in CMS...                                        │
│             │                │          │        │                   │               │ https://avd.aquasec.com/nvd/cve-2026-28390                   │
├─────────────┤                │          │        │                   │               │                                                              │
│ libssl3     │                │          │        │                   │               │                                                              │
├─────────────┼────────────────┼──────────┼────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ zlib        │ CVE-2026-22184 │          │        │ 1.3.1-r2          │ 1.3.2-r0      │ zlib: zlib: Arbitrary code execution via buffer overflow in  │
│             │                │          │        │                   │               │ untgz utility                                                │
│             │                │          │        │                   │               │ https://avd.aquasec.com/nvd/cve-2026-22184                   │
└─────────────┴────────────────┴──────────┴────────┴───────────────────┴───────────────┴──────────────────────────────────────────────────────────────┘
```

Figura 95. Vulnerabilidades detectadas por Trivy en el componente Java de la imagen administrativa.

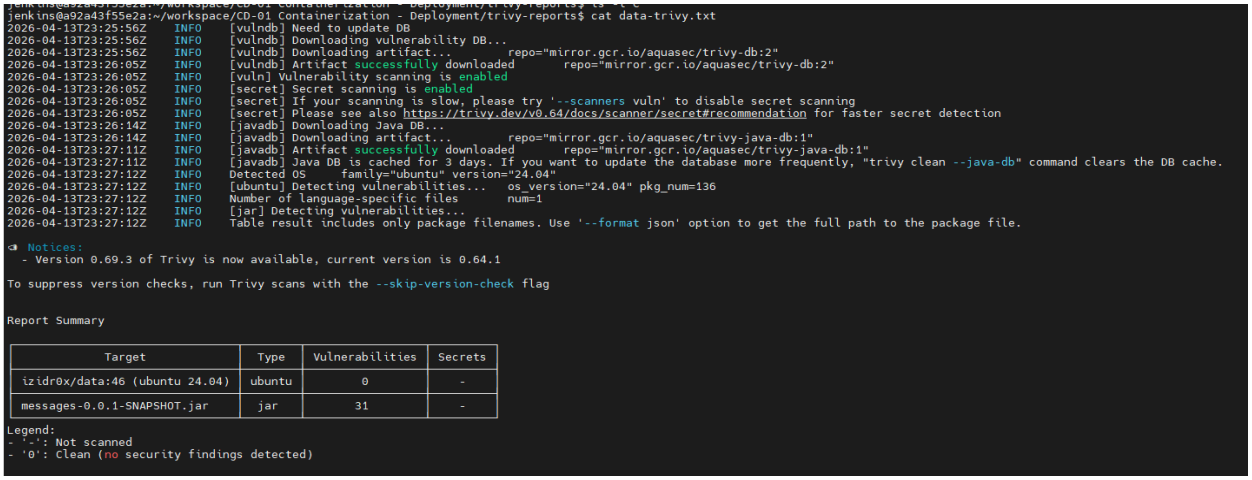


Java (jar)
==========
Total: 20 (HIGH: 18, CRITICAL: 2)

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
org.apache.tomcat.embed:tomcat-embed-core (app.jar)	CVE-2025-24813	CRITICAL	fixed	10.1.19	11.0.3, 10.1.35, 9.0.99	tomcat: Potential RCE and/or information disclosure and/or information corruption with partial PUT... https://avd.aquasec.com/nvd/cve-2025-24813
	CVE-2026-29145				9.0.116, 10.1.53, 11.0.20	Apache Tomcat: Apache Tomcat: Authentication bypass due to CLIENT_CERT soft fail misconfiguration... https://avd.aquasec.com/nvd/cve-2026-29145
	CVE-2024-34750	HIGH			11.0.0-M21, 10.1.25, 9.0.90	tomcat: Improper Handling of Exceptional Conditions https://avd.aquasec.com/nvd/cve-2024-34750
	CVE-2024-50379				11.0.2, 10.1.34, 9.0.98	tomcat: RCE due to TOCTOU issue in JSP compilation https://avd.aquasec.com/nvd/cve-2024-50379
	CVE-2024-56337					tomcat: Incomplete fix for CVE-2024-50379 - RCE due to TOCTOU issue in... https://avd.aquasec.com/nvd/cve-2024-56337
	CVE-2025-48988				11.0.8, 10.1.42, 9.0.106	tomcat: Apache Tomcat DoS in multipart upload https://avd.aquasec.com/nvd/cve-2025-48988
	CVE-2025-48989				11.0.10, 10.1.44, 9.0.108	tomcat: http/2 "MadeYouReset" DoS attack through HTTP/2 control frames https://avd.aquasec.com/nvd/cve-2025-48989
	CVE-2025-52520				11.0.9, 10.1.43, 9.0.107	tomcat: Apache Tomcat denial of service https://avd.aquasec.com/nvd/cve-2025-52520
	CVE-2025-53506				9.0.107, 10.1.43, 11.0.9	tomcat: Apache Tomcat denial of service https://avd.aquasec.com/nvd/cve-2025-53506
	CVE-2025-55752				11.0.11, 10.1.45, 9.0.109	tomcat: org.apache.tomcat/tomcat-catalina: Apache Tomcat: Directory traversal via rewrite with possible RCE https://avd.aquasec.com/nvd/cve-2025-55752
	CVE-2026-24734				11.0.18, 10.1.52, 9.0.115	tomcat: Apache Tomcat: Certificate revocation bypass due to improper OCSP response validation... https://avd.aquasec.com/nvd/cve-2026-24734
	CVE-2026-24880				9.0.116, 10.1.52, 11.0.20	Apache Tomcat: Apache Tomcat: HTTP Request/Response Smuggling via invalid chunk extension https://avd.aquasec.com/nvd/cve-2026-24880
	CVE-2026-34483				9.0.116, 10.1.54, 11.0.21	Apache Tomcat: Apache Tomcat: Information disclosure due to improper encoding in JsonAccessLogValve... https://avd.aquasec.com/nvd/cve-2026-34483

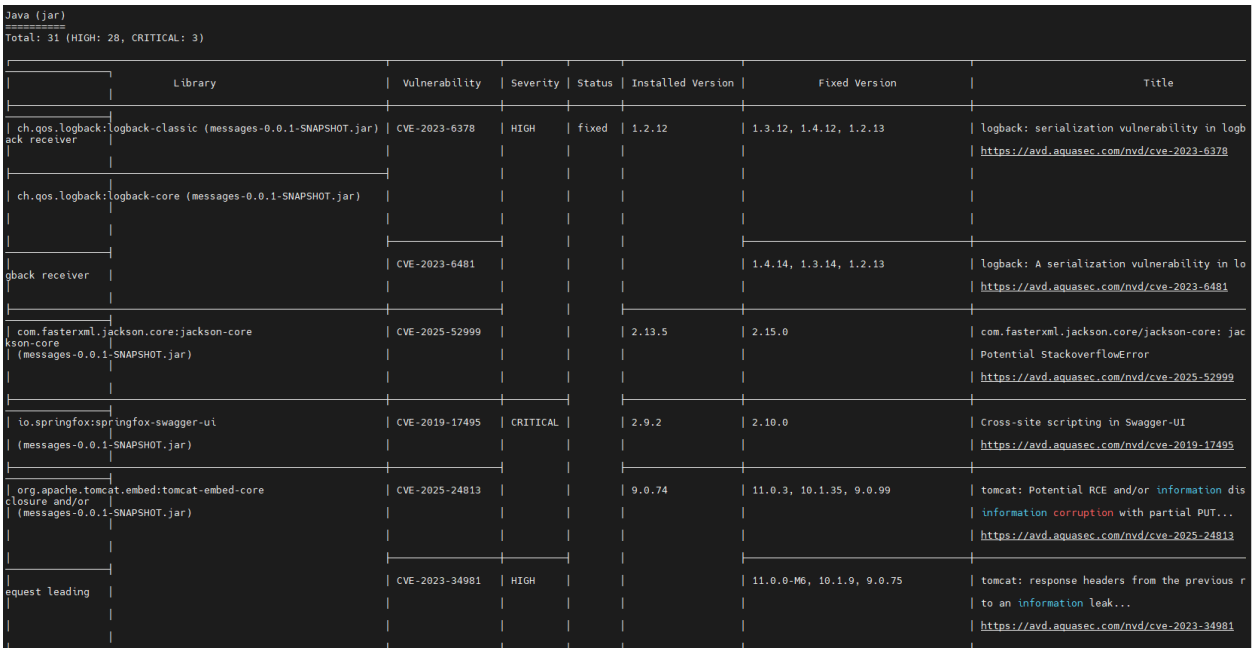
Para el componente de datos, el resultado fue aún más representativo. En este caso, la imagen base Ubuntu 24.04 no reportó vulnerabilidades en el resumen mostrado, mientras que el archivo messages-0.0.1-SNAPSHOT.jar presentó un total de 31 vulnerabilidades, de las cuales 28 fueron clasificadas como altas y 3 como críticas. Entre los hallazgos visibles se identifican componentes como logback-classic, logback-core, jackson-core, springfox-swagger-ui y tomcat-embed-core, lo que confirma nuevamente que la mayor exposición de riesgo se encuentra en las dependencias incluidas dentro del artefacto Java y no necesariamente en la imagen base utilizada para el contenedor.

Figura 96. Reporte resumido de Trivy para la imagen izidr0x/data:46.



```
jenkins@a92a43f55e2a:~/workspace/CD-01 Containerization - Deployment/trivy-reports$ cat data-trivy.txt
2026-04-13T23:25:56Z    INFO    [vulndb] Need to update DB
2026-04-13T23:25:56Z    INFO    [vulndb] Downloading vulnerability DB...
2026-04-13T23:25:56Z    INFO    [vulndb] Downloading artifact...        repo="mirror.gcr.io/aquasec/trivy-db:2"
2026-04-13T23:26:05Z    INFO    [vulndb] Artifact successfully downloaded       repo="mirror.gcr.io/aquasec/trivy-db:2"
2026-04-13T23:26:05Z    INFO    [vuln] Vulnerability scanning is enabled
2026-04-13T23:26:05Z    INFO    [secret] Secret scanning is enabled
2026-04-13T23:26:05Z    INFO    [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2026-04-13T23:26:05Z    INFO    [secret] Please see also https://trivy.dev/v0.64/docs/scanner/secret#recommendation for faster secret detection
2026-04-13T23:26:14Z    INFO    [javadb] Downloading Java DB...
2026-04-13T23:26:14Z    INFO    [javadb] Downloading artifact...        repo="mirror.gcr.io/aquasec/trivy-java-db:1"
2026-04-13T23:27:11Z    INFO    [javadb] Artifact successfully downloaded       repo="mirror.gcr.io/aquasec/trivy-java-db:1"
2026-04-13T23:27:11Z    INFO    [javadb] Java DB is cached for 3 days. If you want to update the database more frequently, "trivy clean --java-db" command clears the DB cache.
2026-04-13T23:27:12Z    INFO    Detected OS     family="ubuntu" version="24.04"
2026-04-13T23:27:12Z    INFO    [ubuntu] Detecting vulnerabilities...   os_version="24.04" pkg_num=136
2026-04-13T23:27:12Z    INFO    Number of language-specific files       num=1
2026-04-13T23:27:12Z    INFO    [jar] Detecting vulnerabilities...
2026-04-13T23:27:12Z    INFO    Table result includes only package filenames. Use '--format json' option to get the full path to the package file.

Notices:
  - Version 0.69.3 of Trivy is now available, current version is 0.64.1

To suppress version checks, run Trivy scans with the --skip-version-check flag


Report Summary

┌──────────────────────────────┬────────┬─────────────────┬─────────┐
│            Target            │  Type  │ Vulnerabilities │ Secrets │
├──────────────────────────────┼────────┼─────────────────┼─────────┤
│ izidr0x/data:46 (ubuntu 24.04) │ ubuntu │        0        │    -    │
├──────────────────────────────┼────────┼─────────────────┼─────────┤
│ messages-0.0.1-SNAPSHOT.jar  │  jar   │       31        │    -    │
└──────────────────────────────┴────────┴─────────────────┴─────────┘
Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)
```

Figura 97. Vulnerabilidades detectadas por Trivy en el componente Java de la imagen de datos.



```
Java (jar)
==========
Total: 31 (HIGH: 28, CRITICAL: 3)

┌──────────────────────────────────────────────────┬──────────────────┬──────────┬────────┬───────────────────┬──────────────────────────────┬──────────────────────────────────────────────────┐
│                     Library                      │  Vulnerability   │ Severity │ Status │ Installed Version │        Fixed Version         │                      Title                       │
├──────────────────────────────────────────────────┼──────────────────┼──────────┼────────┼───────────────────┼──────────────────────────────┼──────────────────────────────────────────────────┤
│ ch.qos.logback:logback-classic (messages-0.0.1-SNAPSHOT.jar) │ CVE-2023-6378    │ HIGH     │ fixed  │ 1.2.12            │ 1.3.12, 1.4.12, 1.2.13       │ logback: serialization vulnerability in logback receiver │
│                                                  │                  │          │        │                   │                              │ https://avd.aquasec.com/nvd/cve-2023-6378        │
├──────────────────────────────────────────────────┤                  │          │        │                   │                              │                                                  │
│ ch.qos.logback:logback-core (messages-0.0.1-SNAPSHOT.jar)    │                  │          │        │                   │                              │                                                  │
│                                                  │                  │          │        │                   │                              │                                                  │
│                                                  ├──────────────────┤          │        │                   │                              │                                                  │
│                                                  │ CVE-2023-6481    │          │        │                   │ 1.4.14, 1.3.14, 1.2.13       │ logback: A serialization vulnerability in logback receiver │
│                                                  │                  │          │        │                   │                              │ https://avd.aquasec.com/nvd/cve-2023-6481        │
├──────────────────────────────────────────────────┼──────────────────┼──────────┼────────┼───────────────────┼──────────────────────────────┼──────────────────────────────────────────────────┤
│ com.fasterxml.jackson.core:jackson-core          │ CVE-2025-52999   │          │        │ 2.13.5            │ 2.15.0                       │ com.fasterxml.jackson.core/jackson-core: jackson-core │
│ (messages-0.0.1-SNAPSHOT.jar)                    │                  │          │        │                   │                              │ Potential StackoverflowError                     │
│                                                  │                  │          │        │                   │                              │ https://avd.aquasec.com/nvd/cve-2025-52999       │
├──────────────────────────────────────────────────┼──────────────────┼──────────┼────────┼───────────────────┼──────────────────────────────┼──────────────────────────────────────────────────┤
│ io.springfox:springfox-swagger-ui                │ CVE-2019-17495   │ CRITICAL │        │ 2.9.2             │ 2.10.0                       │ Cross-site scripting in Swagger-UI               │
│ (messages-0.0.1-SNAPSHOT.jar)                    │                  │          │        │                   │                              │ https://avd.aquasec.com/nvd/cve-2019-17495       │
├──────────────────────────────────────────────────┼──────────────────┼──────────┼────────┼───────────────────┼──────────────────────────────┼──────────────────────────────────────────────────┤
│ org.apache.tomcat.embed:tomcat-embed-core        │ CVE-2025-24813   │          │        │ 9.0.74            │ 11.0.3, 10.1.35, 9.0.99      │ tomcat: Potential RCE and/or information disclosure and/or │
│ (messages-0.0.1-SNAPSHOT.jar)                    │                  │          │        │                   │                              │ information corruption with partial PUT...       │
│                                                  │                  │          │        │                   │                              │ https://avd.aquasec.com/nvd/cve-2025-24813       │
│                                                  ├──────────────────┼──────────┼────────┼───────────────────┼──────────────────────────────┼──────────────────────────────────────────────────┤
│                                                  │ CVE-2023-34981   │ HIGH     │        │ 11.0.0-M6, 10.1.9, 9.0.75    │ tomcat: response headers from the previous request leading │
│                                                  │                  │          │        │                   │                              │ to an information leak...                        │
│                                                  │                  │          │        │                   │                              │ https://avd.aquasec.com/nvd/cve-2023-34981       │
└──────────────────────────────────────────────────┴──────────────────┴──────────┴────────┴───────────────────┴──────────────────────────────┴──────────────────────────────────────────────────┘
```

Estos resultados permiten interpretar que el escaneo de imágenes no solo cumple la función de revisar el sistema operativo base del contenedor, sino también la composición interna del artefacto desplegado. Desde esta perspectiva, Trivy aporta una visión complementaria a la obtenida mediante SCA, ya que evidencia cómo las vulnerabilidades

de dependencias terminan materializándose dentro de la imagen final que será promovida a etapas posteriores del pipeline. Esto resulta especialmente relevante en un entorno DevSecOps, donde no basta con validar el código fuente o las dependencias de manera aislada, sino que también es necesario examinar el artefacto empaquetado tal como será ejecutado.

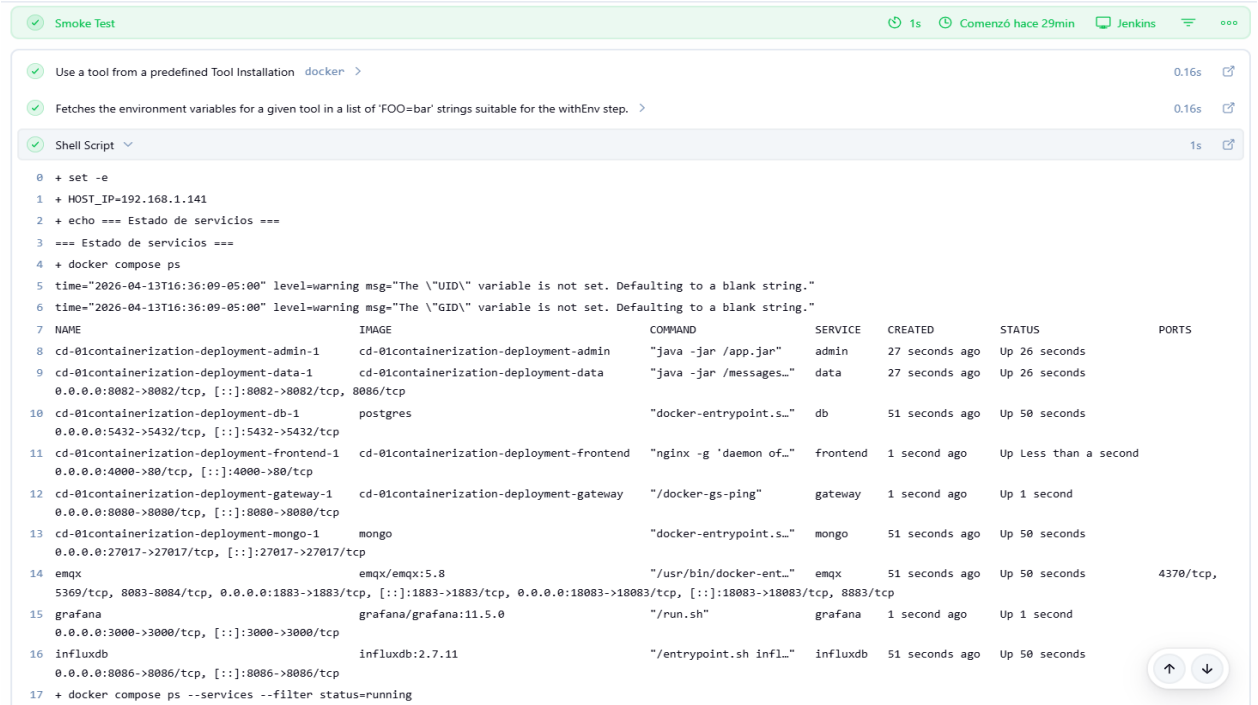
Desde el punto de vista del equipo de desarrollo, esta etapa ofrece beneficios importantes. En primer lugar, permite identificar de forma temprana si una imagen candidata a despliegue contiene componentes con vulnerabilidades severas, evitando que el proceso avance sin una revisión mínima de seguridad. En segundo lugar, brinda información concreta para priorizar acciones correctivas, como la actualización de bibliotecas embebidas, el cambio de versiones de dependencias o la revisión de la imagen base seleccionada. Finalmente, fortalece la trazabilidad del pipeline, ya que cada escaneo queda asociado a una imagen específica y a una ejecución concreta, permitiendo relacionar los hallazgos con el artefacto que efectivamente se construyó y evaluó.

Sin embargo, los resultados también muestran que la superación de etapas previas del pipeline no garantiza por sí sola que la imagen final se encuentre en un estado óptimo de seguridad. En ambos componentes persisten vulnerabilidades de severidad alta y crítica, especialmente concentradas en los archivos JAR incluidos dentro de las imágenes. Por tanto, esta fase se consolida como un control indispensable dentro del flujo DevSecOps implementado, ya que permite extender la validación de seguridad hasta el nivel del artefacto desplegable y aporta evidencia útil para la revisión y mejora progresiva del proceso de construcción y despliegue. En conjunto, los resultados obtenidos demuestran que el pipeline logra incorporar de forma efectiva una validación automatizada sobre las imágenes generadas para los microservicios principales del proyecto, reforzando la seguridad antes del despliegue en el respectivo entorno.

8.1.5 Smoke test. Como parte de la validación final del flujo de despliegue, se ejecutó una fase de Smoke Test con el propósito de comprobar que los servicios desplegados por el pipeline alcanzaran un estado operativo básico en el entorno de prueba. Esta etapa se planteó como una verificación rápida posterior al despliegue, orientada a confirmar tanto la disponibilidad de los contenedores como la capacidad de respuesta inicial de los servicios desplegados por la arquitectura.

En un primer nivel, la validación se realizó sobre el estado de los contenedores mediante el comando “docker compose ps”. Los resultados obtenidos evidenciaron que los servicios esperados dentro del entorno desplegado se encontraban en estado Up, incluyendo los componentes admin, data, db, frontend, gateway, mongo, emqx, grafana e influxdb. Este resultado confirma que la secuencia de despliegue ejecutada por el pipeline permitió levantar correctamente tanto la infraestructura base como los microservicios principales y los servicios complementarios del ecosistema Smart Campus UIS.

Figura 98. Estado operativo de los contenedores desplegados en el entorno de prueba.



En un segundo nivel, el script de validación verificó de manera individual la presencia de los servicios en ejecución utilizando docker compose ps --services --filter status=running, junto con búsquedas específicas para cada componente. Esta comprobación permitió asegurar que los contenedores esperados no solo habían sido creados, sino que efectivamente permanecían activos al momento de la validación. De esta manera, el smoke test aportó una confirmación adicional sobre la estabilidad inicial del despliegue y sobre la correcta disponibilidad de los servicios críticos dentro del entorno de pruebas.

Figura 99. Comprobación de servicios activos dentro del entorno desplegado.

Shell Script

```
+ docker compose ps --services --filter status=running
+ grep -x db
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"UID\" variable is not set. Defaulting to a blank string."
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"GID\" variable is not set. Defaulting to a blank string."
+ docker compose ps --services --filter status=running
+ grep -x mongo
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"UID\" variable is not set. Defaulting to a blank string."
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"GID\" variable is not set. Defaulting to a blank string."
+ docker compose ps --services --filter status=running
+ grep -x emqx
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"UID\" variable is not set. Defaulting to a blank string."
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"GID\" variable is not set. Defaulting to a blank string."
+ docker compose ps --services --filter status=running
+ grep -x influxdb
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"UID\" variable is not set. Defaulting to a blank string."
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"GID\" variable is not set. Defaulting to a blank string."
+ docker compose ps --services --filter status=running
+ grep -x admin
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"UID\" variable is not set. Defaulting to a blank string."
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"GID\" variable is not set. Defaulting to a blank string."
+ docker compose ps --services --filter status=running
+ grep -x data
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"UID\" variable is not set. Defaulting to a blank string."
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"GID\" variable is not set. Defaulting to a blank string."
+ docker compose ps --services --filter status=running
+ grep -x gateway
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"UID\" variable is not set. Defaulting to a blank string."
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"GID\" variable is not set. Defaulting to a blank string."
+ docker compose ps --services --filter status=running
```

Posteriormente, se realizaron verificaciones de conectividad HTTP sobre algunos de los servicios expuestos. En particular, el microservicio data respondió con código HTTP 404 en la ruta consultada, el servicio gateway también respondió con HTTP 404, y el componente frontend respondió con HTTP 200. Dentro del criterio definido para esta fase, estas respuestas fueron consideradas satisfactorias, ya que el objetivo principal del smoke test no consistía en validar la lógica funcional completa de los endpoints, sino en comprobar que los servicios estuvieran accesibles y fueran capaces de responder a solicitudes HTTP. En este contexto, el código 404 no se interpretó como una caída del servicio, sino como evidencia de que la aplicación se encontraba activa, aunque la ruta consultada no correspondiera a un recurso disponible. El criterio de fallo crítico se reservó para escenarios sin respuesta, representados por códigos como 000, que indicarían indisponibilidad efectiva del servicio.

Figura 100. Resultado de las solicitudes HTTP ejecutadas como parte del smoke test.

Shell Script

```
+ grep -x frontend
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"UID\" variable is not set. Defaulting to a blank string."
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"GID\" variable is not set. Defaulting to a blank string."
+ docker compose ps --services --filter status=running
+ grep -x grafana
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"UID\" variable is not set. Defaulting to a blank string."
time="2026-04-13T16:36:09-05:00" level=warning msg="The \"GID\" variable is not set. Defaulting to a blank string."
+ check_http data http://192.168.1.141:8082/
+ NAME=data
+ URL=http://192.168.1.141:8082/
+ curl -s -o /dev/null -w %{http_code} http://192.168.1.141:8082/
+ CODE=404
+ [ 404 = 000 ]
+ echo PASS: data responde con HTTP 404 en http://192.168.1.141:8082/
PASS: data responde con HTTP 404 en http://192.168.1.141:8082/
+ check_http gateway http://192.168.1.141:8080/
+ NAME=gateway
+ URL=http://192.168.1.141:8080/
+ curl -s -o /dev/null -w %{http_code} http://192.168.1.141:8080/
+ CODE=404
+ [ 404 = 000 ]
+ echo PASS: gateway responde con HTTP 404 en http://192.168.1.141:8080/
PASS: gateway responde con HTTP 404 en http://192.168.1.141:8080/
+ check_http frontend http://192.168.1.141:4000/
+ NAME=frontend
+ URL=http://192.168.1.141:4000/
+ curl -s -o /dev/null -w %{http_code} http://192.168.1.141:4000/
+ CODE=200
+ [ 200 = 000 ]
+ echo PASS: frontend responde con HTTP 200 en http://192.168.1.141:4000/
PASS: frontend responde con HTTP 200 en http://192.168.1.141:4000/
```

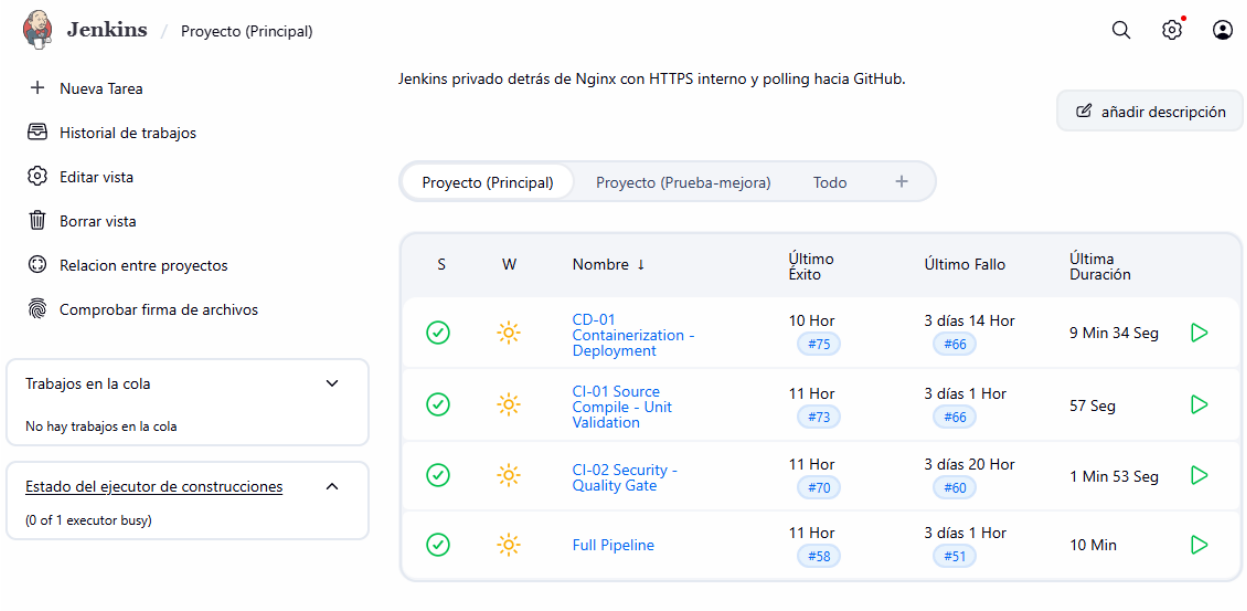
Desde la perspectiva del equipo de desarrollo, esta etapa aporta un beneficio importante al ofrecer una validación rápida del resultado del despliegue antes de considerar que la ejecución del pipeline ha concluido satisfactoriamente. Esto permite detectar de manera temprana problemas evidentes de inicialización, fallos en dependencias entre servicios o errores de publicación de puertos, reduciendo el riesgo de dar por exitoso un despliegue que en realidad no se encuentra operativo. Además, proporciona una evidencia clara y trazable del estado básico del sistema inmediatamente después de la ejecución automatizada.

No obstante, también debe señalarse que esta validación corresponde a una prueba básica de disponibilidad y arranque, por lo que no sustituye pruebas funcionales más profundas ni evaluaciones de integración completas. Aun así, dentro del alcance definido para el proyecto, los resultados obtenidos demuestran que el pipeline DevSecOps implementado logra desplegar la arquitectura en un entorno de prueba y validar de forma automatizada su disponibilidad inicial, lo que fortalece la confiabilidad del proceso de construcción y despliegue continuo.

8.2 VERSIÓN DEL SISTEMA VERIFICADA - P5.3

En conjunto, los resultados obtenidos permiten afirmar que el pipeline DevSecOps implementado para Smart Campus UIS cumple con el propósito de automatizar etapas clave del flujo CI/CD e incorporar controles de seguridad temprana sobre el software analizado. Las evidencias generadas muestran que el proceso logró ejecutar validaciones funcionales iniciales, análisis estático del código, revisión de dependencias, escaneo de imágenes y verificaciones operativas básicas posteriores al despliegue, proporcionando información útil para la trazabilidad de cada ejecución y para la identificación temprana de hallazgos técnicos y de seguridad. Aunque los resultados también evidencian la persistencia de oportunidades de mejora, especialmente en cobertura de pruebas y remediación de vulnerabilidades en dependencias e imágenes, el comportamiento general del flujo demuestra que la propuesta es viable y aporta una base concreta para evolucionar hacia un modelo de desarrollo más automatizado, controlado y alineado con principios DevSecOps dentro del contexto institucional de Smart Campus UIS.

Figura 101. Versión del sistema verificada.



9. CONCLUSIONES Y TRABAJO FUTURO

9.1 CONCLUSIONES

El desarrollo del presente trabajo permitió cumplir el objetivo general planteado, al diseñar e implementar un flujo CI/CD con enfoque DevSecOps para la plataforma Smart Campus UIS, incorporando controles automatizados de validación funcional, análisis de seguridad del código fuente, análisis de dependencias, escaneo de imágenes de contenedor y verificación operativa básica posterior al despliegue. En este sentido, el proyecto consolidó una base técnica que integra automatización, trazabilidad y seguridad temprana dentro del ciclo de vida del software.

En relación con el primer objetivo específico, el análisis del estado actual de Smart Campus UIS permitió identificar que la plataforma ya contaba con una arquitectura funcional basada en microservicios contenerizados, pero que sus procesos de desarrollo, construcción y despliegue presentaban limitaciones importantes en términos de automatización, trazabilidad y control sistemático de seguridad. Se evidenciaron dependencias de procesos manuales, desacoplamiento entre código validado y artefactos desplegados, así como ausencia de mecanismos automáticos de validación continua. A partir de este diagnóstico, se justificó técnicamente la adopción de una arquitectura de pipeline centralizada y orientada a principios DevSecOps.

Respecto al segundo objetivo específico, se logró integrar de manera continua y automatizada prácticas de análisis estático de seguridad, análisis de composición de software y verificación de artefactos e imágenes. La línea base tecnológica quedó conformada por SonarQube Community Build como herramienta SAST, OWASP Dependency-Check como herramienta SCA y Trivy como mecanismo de escaneo de imágenes. Esta integración permitió establecer validaciones en diferentes niveles del pipeline: sobre el código fuente, sobre las dependencias externas y sobre el artefacto contenerizado final, fortaleciendo el enfoque de seguridad temprana propuesto en el trabajo.

En cuanto al tercer objetivo específico, se diseñó una arquitectura DevSecOps estructurada en bloques funcionales que separan la validación inicial del código, el control de calidad y seguridad temprana, y la contenerización con despliegue controlado. Esta organización permitió establecer un flujo más claro, mantenible y trazable, soportado por Jenkins como orquestador principal, complementado con herramientas de construcción, análisis y despliegue acordes con las necesidades del proyecto. Como resultado, se definió una solución coherente con el principio de separación de responsabilidades y con el enfoque Shift Left Security adoptado para Smart Campus UIS.

Frente al cuarto objetivo específico, el trabajo también logró documentar las configuraciones, decisiones técnicas y procesos implementados en el pipeline. Esto se evidencia en la descripción detallada del entorno de orquestación, la estructuración del repositorio, la definición de requerimientos funcionales y de seguridad, la selección de herramientas, la implementación de los stages y el análisis de resultados obtenidos. En consecuencia, el documento no solo presenta una solución funcional, sino que también deja una base de referencia útil para su comprensión, replicabilidad y mantenimiento posterior dentro del contexto institucional.

Los resultados obtenidos durante la validación del pipeline permiten concluir que la automatización incorporada aporta beneficios concretos al proceso de desarrollo. Las pruebas unitarias ejecutadas sobre los microservicios principales mostraron ejecuciones satisfactorias y estables en los builds observados, lo que proporciona una validación funcional inicial y retroalimentación temprana para el equipo de desarrollo. De igual forma, el análisis SAST permitió identificar hallazgos relacionados con mantenibilidad, seguridad y calidad del código, mientras que el uso de quality gates introdujo un criterio automático de continuidad dentro del flujo CI/CD.

Por otra parte, los resultados del análisis SCA y del escaneo de imágenes evidencian que la implementación del pipeline no debe interpretarse como una garantía de que el software haya quedado libre de vulnerabilidades, sino como un mecanismo efectivo para detectarlas tempranamente y generar evidencia útil para su priorización y tratamiento. En particular, se identificaron vulnerabilidades de severidad alta y crítica tanto en

dependencias como en los artefactos empaquetados dentro de las imágenes de contenedor, lo que confirma la necesidad de mantener controles continuos sobre la cadena de suministro del software y sobre los componentes de terceros utilizados por la plataforma.

Finalmente, el smoke test permitió comprobar que el pipeline no solo compila, analiza y construye artefactos, sino que también es capaz de desplegar la arquitectura en un entorno de prueba y validar de manera automatizada su disponibilidad inicial. Aunque esta validación es básica y no sustituye pruebas funcionales exhaustivas ni evaluaciones de integración más profundas, sí constituye una evidencia importante de operatividad del flujo implementado. En conjunto, el trabajo demuestra que la incorporación de un pipeline CI/CD con enfoque DevSecOps en Smart Campus UIS es viable, útil y pertinente, y que representa un avance significativo hacia un modelo de desarrollo más controlado, reproducible y orientado a la seguridad desde etapas tempranas.

9.2 TRABAJO FUTURO

Como trabajo futuro, se propone ampliar el alcance de la validación automatizada incorporada en el pipeline, especialmente lo relacionado con la parte de pruebas. Si bien la solución implementada permitió integrar pruebas unitarias y una verificación operativa básica posterior al despliegue, resulta conveniente avanzar hacia pruebas de integración más profundas, validaciones funcionales más representativas y mecanismos de regresión automatizada que permitan fortalecer la confiabilidad del software desde una perspectiva más amplia. Esta evolución permitiría que el pipeline no solo valide la construcción y el arranque de los servicios, si no también comportamientos funcionales más cercanos al uso real de la plataforma.

También resulta pertinente incorporar mecanismos adicionales de seguridad que complementen el alcance actual del trabajo. En particular, una extensión natural del pipeline sería la integración de pruebas dinámicas de seguridad (DAST), con el fin de evaluar el comportamiento de los servicios desplegados en ejecución y detectar vulnerabilidades que no pueden identificarse únicamente mediante análisis estático,

revisión de dependencias o escaneo de imágenes. Esta línea permitiría avanzar hacia una validación más completa del software dentro del enfoque DevSecOps.

De forma complementaria, se propone fortalecer la gestión de vulnerabilidades detectadas en dependencias e imágenes de contenedor. Los resultados obtenidos muestran que aún persisten hallazgos de severidad alta y crítica en componentes externos y artefactos empaquetados, por lo que futuras iteraciones del pipeline podrían incorporar políticas más estrictas de actualización de dependencias, umbrales de rechazo más maduros, definir reglas para corregir primero las vulnerabilidades más graves y configurar el pipeline para bloquear la publicación o el despliegue de artefactos e imágenes que superen los niveles de riesgo permitidos.

Otra línea de mejora se relaciona con el endurecimiento de la infraestructura que soporta el pipeline. Durante la implementación se documentó como consideración técnica el acceso del controlador Jenkins al motor Docker del host para soportar tareas requeridas por los pipelines. En trabajos posteriores sería conveniente evaluar alternativas más seguras de ejecución, aislamiento y administración del entorno de CI/CD, con el fin de reducir la superficie de exposición de la infraestructura y fortalecer la seguridad operativa del orquestador.

Así mismo, se considera de gran valor extender el alcance del pipeline hacia otros componentes del ecosistema Smart Campus UIS que actualmente no fueron priorizados dentro de la validación automatizada. Esto permitiría evolucionar desde una implementación focalizada en los microservicios principales hacia una estrategia más integral de automatización y seguridad sobre una mayor porción de la arquitectura institucional, conservando el enfoque progresivo y controlado planteado en este trabajo.

Finalmente, como proyección a nivel académico e institucional, el trabajo podría complementarse con la incorporación de mecanismos más avanzados de trazabilidad y seguridad de la cadena de suministro, tales como generación de SBOM (Software Bill of Materials), firma y verificación de artefactos, validación de secretos, políticas de despliegue más estrictas, estrategias de rollback automatizado y mayor integración con

monitoreo y observabilidad posterior al despliegue. Estas extensiones permitirían llevar la solución propuesta hacia un nivel de madurez superior, consolidando a Smart Campus UIS como un caso de referencia en adopción progresiva de prácticas DevSecOps dentro de entornos universitarios.

BIBLIOGRAFÍA

ABIONA, O., et al. The emergence and importance of DevSecOps: Integrating and reviewing security practices within the DevOps pipeline. En: World Journal of Advanced Engineering Technology and Sciences. 2024. DOI: https://doi.org/10.30574/wjaets.2024.11.2.0093.

AGGARWAL, A. y JALOTE, P. Integrating Static and Dynamic Analysis for Detecting Vulnerabilities. En: 30th Annual International Computer Software and Applications Conference (COMPSAC'06). 2006, vol. 1, p. 343-350. DOI: https://doi.org/10.1109/compsac.2006.55.

ALENEZI, M. y ALMUAIRFI, S. Security Risks in the Software Development Lifecycle. En: International Journal of Recent Technology and Engineering (IJRTE). 2019, vol. 8, nro. 3, p. 7048-7055. Disponible en: https://www.ijrte.org/portfolio-item/C5374098319

AMGOTHU, S. An End-to-End CI/CD Pipeline Solution Using Jenkins and Kubernetes. En: International Journal of Science and Research (IJSR). 2024. Disponible en: https://www.ijsr.net/archive/v13i8/SR24826231120.pdf

ANDRADE, R., et al. Factors of Risk Analysis for IoT Systems. En: Risks. 2022, vol. 10, nro. 8, art. 162. Disponible en: https://www.mdpi.com/2227-9091/10/8/162

APARO, C., et al. An Analysis System to Test Security of Software on Continuous Integration-Continuous Delivery Pipeline. En: 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). 2023, p. 58-67. Disponible en: https://ieeexplore.ieee.org/document/10190695

BACH-NUTMAN, M. Understanding The Top 10 OWASP Vulnerabilities. En: ArXiv. 2020, abs/2012.09960. DOI: https://doi.org/10.48550/arXiv.2012.09960

BASHIRU, O. y OLUFEMI, O. An Enhanced CICD Pipeline: A DevSecOps Approach. En: International Journal of Computer Applications. 2023. DOI: https://doi.org/10.5120/ijca2023922594.

BASINYA, E. A. y MALYSHEV, E. A. The Creation and Integration of the Technological Workflow for Software and Hardware-Software Development. En: 2023 IEEE XVI International Scientific and Technical Conference Actual Problems of Electronic Instrument Engineering (APEIE). 2023, p. 960-966. Disponible en: https://ieeexplore.ieee.org/document/10347739

BELLINI, P.; NESI, P. y PANTALEO, G. IoT-Enabled Smart Cities: A Review of Concepts, Frameworks and Key Technologies. En: Applied Sciences. 2022. DOI: https://doi.org/10.3390/app12031607.

CHANDRAMOULI, R. Implementation of DevSecOps for a Microservices-Based Application with Service Mesh [en línea]. Gaithersburg: National Institute of Standards and Technology, 2022. NIST Special Publication 800-204C. [Consultado: 6 de junio de 2026]. Disponible en: https://csrc.nist.gov/pubs/sp/800/204/c/final

CORREA-VALOYES, J. Implementación de Herramientas de Seguridad en Pipelines CI/CD con Azure DevOps en la Empresa Periferia IT Group [en línea]. Universidad de Santander, 2025. [Consultado: 6 de junio de 2026]. Disponible en: https://repositorio.udes.edu.co/handle/001/12347

CROFT, R., et al. An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing. En: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 2021. DOI: https://doi.org/10.1145/3475716.3475781

DARIENKO, D. y KOHUT, N. Docker Container Image Scanning Methods. En: Computer Systems and Network. 2024. DOI: https://doi.org/10.23939/csn2024.02.034

DAS, K. Create an Enterprise-Level Test Automation Framework with Appium: Using Spring-Boot, Gradle, Junit, ALM Integration, and Custom Reports with TDD and BDD Support. Berkeley: Apress, 2022. Disponible en: https://link.springer.com/book/10.1007/978-1-4842-8197-0

ESPOSITO, M.; FALASCHI, V. y FALESSI, D. An Extensive Comparison of Static Application Security Testing Tools. En: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering. 2024. DOI: https://doi.org/10.1145/3661167.3661199

FEIO, C., et al. An Empirical Study of DevSecOps Focused on Continuous Security Testing. En: 2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). 2024, p. 610-617. Disponible en: https://ieeexplore.ieee.org/document/10628738

GBENLE, P., et al. A DevSecOps-Centered Conceptual Model for Continuous Integration and Secure Deployment in Software Development Lifecycles. [s. l.]: [s. n.], s. f.

HE, R., et al. Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot. En: IEEE Transactions on Software Engineering. 2022, vol. 49, p. 4004-4022. DOI: https://doi.org/10.1109/tse.2023.3278129

HYUN, G., et al. The Impact of an Automation System Built with Jenkins on the Efficiency of Container-Based System Deployment. En: Sensors. 2024, vol. 24. Disponible en: https://www.mdpi.com/1424-8220/24/18/6002

IBM. The cost of a data breach 2024: Financial industry insights. En: IBM Think Insights [sitio web]. 2024. [Consultado: 6 de junio de 2026]. Disponible en: https://www.ibm.com/think/insights/cost-of-a-data-breach-2024-financial-industry

JIMÉNEZ, H.; CÁRCAMO, E. y PEDRAZA, G. Extensible software platform for smart campus based on microservices. En: RISTI - Revista Ibérica de Sistemas e Tecnologias de Informação. 2020, nro. E38, p. 270-282. Disponible en: www.scopus.com

JOHNSON, O., et al. Developing advanced CI/CD pipeline models for Java and Python applications: A blueprint for accelerated release cycles. En: Computer Science & IT Research Journal. 2024. Disponible en: https://fepbl.com/index.php/csitrj/article/view/1758

KHAN, R., et al. Systematic Literature Review on Security Risks and its Practices in Secure Software Development. En: IEEE Access. 2022, vol. 10, p. 5456-5481. Disponible en: https://ieeexplore.ieee.org/document/9669954

KHOMH, F., et al. Understanding the impact of rapid releases on software quality. En: Empirical Software Engineering. 2015, vol. 20, p. 336-373. Disponible en: https://link.springer.com/article/10.1007/s10664-014-9308-x

KUMAR, N. Integrating Dynamic Security Testing Tools into CI/CD Pipelines: A Continuous Security Testing Case Study. En: International Journal of Science and Research (IJSR). 2021. DOI: https://doi.org/10.21275/sr24615152732

KUPPUSAMY VELLAMADAM PALAVESAM, et al. Building Automated Security Pipeline for Containerized Microservices. En: Journal of Advances in Mathematics and Computer Science. 2025, vol. 40, nro. 2, p. 53-66. DOI: https://doi.org/10.9734/jamcs/2025/v40i21969

KUSUMADEWI, R. y ADRIAN, R. Performance Analysis of Devops Practice Implementation Of CI/CD Using Jenkins. En: MATICS: Jurnal Ilmu Komputer dan Teknologi Informasi. 2023. Disponible en: https://ejournal.uin-malang.ac.id/index.php/saintek/article/view/17091

LIN, J., et al. Vulnerability management in Linux distributions. En: Empirical Software Engineering. 2023, vol. 28, p. 1-34. DOI: https://doi.org/10.1007/s10664-022-10267-7

MALIK, G. y PRASHASTI, P. Shift Left Security. En: The Eastasouth Journal of Information System and Computer Science. 2025. DOI: https://doi.org/10.58812/esiscs.v2i03.528

MANOLOV, V.; GOTSEVA, D. y HINOV, N. Practical Comparison Between the CI/CD Platforms Azure DevOps and GitHub. En: Future Internet. 2025, vol. 17, art. 153. Disponible en: https://www.mdpi.com/1999-5903/17/4/153

MARANDI, M.; BERTIA, A. y SILAS, S. Implementing and Automating Security Scanning to a DevSecOps CI/CD Pipeline. En: 2023 World Conference on Communication & Computing (WCONF). 2023, p. 1-6. Disponible en: https://ieeexplore.ieee.org/document/10235015

MAXIMINI, D. The Scrum Culture: Introducing Agile Methods in Organizations. 2 ed. Cham: Springer International Publishing, 2018. Disponible en: https://link.springer.com/book/10.1007/978-3-319-73842-0

MICROSOFT. Calculadora de precios | Microsoft Azure [sitio web]. 2025. [Consultado: 6 de junio de 2026]. Disponible en: https://azure.microsoft.com/es-es/pricing/calculator

NANDGAONKAR, S. y KHATAVKAR, V. CI-CD Pipeline For Content Releases. En: 2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT). 2022, p. 1-4. DOI: https://doi.org/10.1109/gcat55367.2022.9972129

NEELAN, A. The Perennial Importance of SOLID Principles in Software Design. En: Journal of Software Engineering and Simulation. 2024. DOI: https://doi.org/10.35629/3795-10035468

NIKOLOV, L. y ALEKSIEVA-PETROVA, A. Action Research on the DevSecOps Pipeline. En: 2023 International Scientific Conference on Computer Science (COMSCI). 2023, p. 1-6. Disponible en: https://ieeexplore.ieee.org/document/10315920

NOCERA, S., et al. Software Composition Analysis and Supply Chain Security in Apache Projects: an Empirical Study. En: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR). 2025, p. 103-115. DOI: https://doi.org/10.1109/msr66628.2025.00027

OWASP. Fundamentos de seguridad. En: OWASP Developer Guide [sitio web]. s. f. [Consultado: 6 de junio de 2026]. Disponible en: https://devguide.owasp.org/es/02-foundations/01-security-fundamentals

PISKACHEV, G.; BECKER, M. y BODDEN, E. Can the configuration of static analyses make resolving security vulnerabilities more effective? A user study. En: Empirical Software Engineering. 2023, vol. 28, p. 1-28. DOI: https://doi.org/10.1007/s10664-023-10354-3

PONTA, S.; PLATE, H. y SABETTA, A. Detection, assessment and mitigation of vulnerabilities in open source dependencies. En: Empirical Software Engineering. 2020, vol. 25, p. 3175-3215. DOI: https://doi.org/10.1007/s10664-020-09830-x

POZO RUIZ, D. M. y FIGUEROA ROSERO, J. A. Devsecops con herramientas Opensource orientados a microservicios para la detección de vulnerabilidades en el ciclo de desarrollo de software de CORE bancario para la banca personas en los Bancos Privados de la ciudad de Quito del Ecuador [en línea]. Tesis de maestría. Universidad Técnica del Norte, 2025. [Consultado: 6 de junio de 2026]. Disponible en: https://repositorio.utn.edu.ec/handle/123456789/17325

PRANA, G., et al. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. En: Empirical Software Engineering. 2021, vol. 26. DOI: https://doi.org/10.1007/s10664-021-09959-3

PRATES, L. y PEREIRA, R. DevSecOps practices and tools. En: International Journal of Information Security. 2025, vol. 24, art. 11. Disponible en: https://link.springer.com/article/10.1007/s10207-024-00914-z

RADHIKA, B., et al. Consistency analysis and flow secure enforcement of SELinux policies. En: Computers & Security. 2020, vol. 94, art. 101816. DOI: https://doi.org/10.1016/j.cose.2020.101816

RAHMAN, A., et al. Analisis Implementasi Nucklei Vulnerability dan OWASP-ZAP Scanner untuk Deteksi Kerentanan Keamanan (Secure System) pada Platform Web Based. En: Jurnal Komputer Terapan. 2025. DOI: https://doi.org/10.35143/jkt.v11i1.6430

RANGNAU, T., et al. Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. En: 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC). 2020, p. 145-154. DOI: https://doi.org/10.1109/edoc49727.2020.00026

RINDELL, K., et al. Security in agile software development: A practitioner survey. En: Information and Software Technology. 2021, vol. 131, art. 106488. DOI: https://doi.org/10.1016/j.infsof.2020.106488

SARAVANOS, A. y CURINGA, M. Simulating the Software Development Lifecycle: The Waterfall Model. En: ArXiv. 2023, abs/2308.03940. DOI: https://doi.org/10.48550/arxiv.2308.03940

SOTO-VALERO, C., et al. A comprehensive study of bloated dependencies in the Maven ecosystem. En: Empirical Software Engineering. 2020, vol. 26. Disponible en: https://link.springer.com/content/pdf/10.1007/s10664-020-09914-8.pdf

TAHAEI, M., et al. Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them. En: Proceedings of the 2021 CHI

Conference on Human Factors in Computing Systems. 2021. DOI: https://doi.org/10.1145/3411764.3445616

WIEDEMANN, A., et al. The DevOps Phenomenon. En: Queue. 2019, vol. 17, p. 93-112. DOI: https://doi.org/10.1145/3329781.3338532

WISIDAGAMA, N. y MARIKKAR, F. Waterfall Model over PCD.UCT Model Review. En: Automation of Technological and Business Processes. 2024. DOI: https://doi.org/10.15673/atbp.v16i3.2927

علي, ن.; محمد, س. y مجدي, د. Comparative Evaluation of Software Composition Analysis Tools in Context of Technical Debt Reduction. En: المجلة الدولية للسياسات العامة في مصر. 2024. DOI: https://doi.org/10.21608/ijppe.2024.389252

ANEXOS

Anexo A. Repositorio de infraestructura DevSecOps.

Descripción:

Repositorio que contiene la configuración de la infraestructura implementada para el ambiente DevSecOps del proyecto, incluyendo los archivos relacionados con Jenkins, SonarQube, certificados, configuración de Nginx y documentación técnica de despliegue.

Contenido principal del repositorio:

Cuadro 7. Contenido del repositorio de infraestructura DevSecOps.

Elemento	Descripción
Jenkins	Configuración del servicio, Docker Compose, persistencia y documentación de operación.
Nginx / certificados	Configuración usada para publicar Jenkins mediante proxy inverso y TLS en el ambiente de pruebas.
SonarQube	Archivos de despliegue y configuración del servicio de análisis estático.
Pipelines	Documentación o definición de los bloques CI-01, CI-02 y CD-01.
README.md	Guía general de instalación, uso y mantenimiento del ambiente.

Fuente: Elaboración propia.

Enlace al repositorio:

Repositorio de infraestructura DevSecOps:
https://github.com/Izidr0x/infra-DevSecOps_SmartCampusUIS/tree/main

Anexo B. Repositorio del código analizado de Smart Campus UIS.

Descripción:

Repositorio correspondiente al código fuente de los microservicios analizados mediante los pipelines DevSecOps implementados en el proyecto. Este repositorio fue usado como base para ejecutar compilación, validación, análisis de calidad, análisis de seguridad y construcción de imágenes.

Contenido principal del repositorio:

Cuadro 8. Contenido del repositorio del código analizado.

Elemento	**Descripción**
Microservicio administrativo	Código fuente del servicio analizado dentro del flujo CI/CD.
Microservicio de datos	Código fuente del servicio analizado dentro del flujo CI/CD.
Archivos Maven	Configuración de construcción y dependencias del proyecto.
Código fuente Java	Componentes funcionales evaluados por el pipeline.
Dockerfile / configuración de contenedores	Archivos utilizados para construir las imágenes del proyecto, si aplica.

Fuente: Elaboración propia a partir del código fuente de la plataforma Smart Campus UIS.

Enlace al repositorio:

Repositorio del código analizado:
https://github.com/PWN3D777/DevSecOps_SmartCampusUIS/tree/main