UNIVERSIDAD POLITÉCNICA SALESIANA ECUADOR

POSGRADOS

MAESTRÍA EN

SOFTWARE CON MENCIÓN EN DISEÑO DE ARQUITECTURA DE SISTEMAS

RPC-SO-34-NO.778-2021

OPCIÓN DE TITULACIÓN:

PROYECTO DE TITULACIÓN CON COMPONENTES DE INVESTIGACIÓN APLICADA Y/O DE DESARROLLO

TEMA:

DESARROLLO DE PROCESO ESTÁNDAR DE ASEGURAMIENTO DE CALIDAD MEDIANTE AUTOMATIZACIÓN DE PUNTOS DE CONTROL, EN LA CONSTRUCCIÓN DE SOFTWARE EN LA COOPERATIVA JARDÍN AZUAYO

AUTORES:

CHRISTIAN RIGOBERTO ESTRADA ONCE
SANTIAGO DE JESÚS PARRA REINOSO

DIRECTOR:

PABLO SEBASTIÁN CALDERÓN MALDONADO

CUENCA – ECUADOR
2026

Autores:

Christian Rigoberto Estrada Once
Ingeniero en Sistemas.
Candidato a Magíster en Software con Mención en Diseño de Arquitectura de Sistemas por la Universidad Politécnica Salesiana – Sede Cuenca.
cestrado@est.ups.edu.ec

Santiago De Jesús Parra Reinoso
Ingeniero en Sistemas.
Candidato a Magíster en Software con Mención en Diseño de Arquitectura de Sistemas por la Universidad Politécnica Salesiana – Sede Cuenca.
sparrar@est.ups.edu.ec

Dirigido por:

Pablo Sebastián Calderón Maldonado
Ingeniero en Sistemas.
Magister en Software con mención en diseño de Arquitectura de Sistemas.
pcalderonm@ups.edu.ec

Desarrollo de proceso estándar de aseguramiento de calidad mediante automatización de puntos de control, en la construcción de software en la Cooperativa Jardín Azuayo

DEDICATORIA

El presente trabajo de tesis está dedicado a mis hijos, Alexander y Emily. Ustedes son y serán mis pilares; quiero que sepan que toda meta, objetivo o sueño se alcanza con esfuerzo, disciplina y determinación. Que nadie les diga que no pueden confíen siempre en ustedes y en sus capacidades, sin temer al qué dirán ni a los obstáculos del día a día. Son mi orgullo más grande.

A mis padres, Alegría y Juan, por su amor incondicional, su guía y su ejemplo. Gracias por estar presentes en cada etapa, por su ánimo constante y por creer en mí. Ustedes son mi inspiración y mi fortaleza; deseo que este logro los haga sentirse orgullosos. A Dios, gracias por la vida y por permitirme tenerlos a mi lado.

Finalmente, a todas las personas que, con sus ideas, consejos, palabras de aliento y compañía, contribuyeron a la realización de este trabajo de titulación, mi gratitud sincera. Reconozco en ustedes un apoyo invaluable; Llenan de paciencia y confianza; Mi agradecimiento por su respaldo incondicional; Recibir su aliento hizo posible concluir este sueño.

Christian Rigoberto Estrada Once

AGRADECIMIENTO

Agradezco, ante todo, a Dios por la vida, la salud y la fortaleza concedidas a mi familia y a mí, que hicieron posible sostener este camino con esperanza y constancia. A mis padres e hijos, por su presencia atenta, su amor inquebrantable y sus palabras de aliento, que me impulsaron a concluir este logro.

A la Cooperativa de Ahorro y Crédito Jardín Azuayo, mi lugar de trabajo, gracias por brindarme la oportunidad de crecer profesionalmente y por confiar en mi desarrollo. A mis compañeros y amigos, por el trabajo compartido y la amistad; de manera especial, a Santiago Parra, por los aprendizajes obtenidos en el curso de esta maestría, que fortalecieron tanto lo personal como lo profesional.

Al Msc. Pablo Sebastián Calderón Maldonado, mi sincero reconocimiento por su orientación cercana y valiosa durante la elaboración de este trabajo de titulación; su guía y apoyo fueron determinantes para llevar este proyecto a buen término.

Christian Rigoberto Estrada Once

DEDICATORIA

A mi padre Reimundo, por su cariño y amor, y por ser un apoyo incondicional en cada paso de este camino. Sus enseñanzas me han dado la fortaleza para continuar adelante y no darme por vencido.

A mis hermanas y hermanos, por sus palabras de aliento y por comprender el tiempo y la dedicación que este proceso requería para alcanzar este objetivo.

A mi hermana en especial Lucila, quien ha sido como una madre para mí y una fuente constante de inspiración. Gracias a su apoyo incondicional he podido llegar hasta aquí.

A mi hermano Fabián, quien ya no está presente físicamente, pero sé que estaría feliz y orgulloso de verme cumplir este sueño. Su impulso, motivación y ayuda marcaron mi vida y mi crecimiento profesional.

A mi madre Esthleta, quien por circunstancias de la vida ya no está conmigo, pero cuyo amor, valores y ejemplo perduran en cada decisión que tomo. Su recuerdo ha sido mi guía y mi mayor fortaleza para no rendirme.

Finamente, a todos los que han creído en mí y me han inspirado a seguir adelante. Este trabajo es el reflejo de su confianza y de los valores que me inculcaron.

Santiago De Jesús Parra Reinoso

AGRADECIMIENTO

Es un honor para mí reconocer y expresar mi sincero agradecimiento a todas las personas que han sido parte fundamental en la culminación de este proyecto.

En primer lugar, agradezco a Dios, por concederme la salud, la fortaleza y la capacidad necesarias para alcanzar la meta que me propuse.

Mi agradecimiento se extiende a mi familia, cuya presencia constante, apoyo emocional y comprensión a lo largo de mi trayectoria académica me brindaron la fortaleza para continuar. A mi padre y a mi hermana, por su amor incondicional y por ser una fuente permanente de inspiración; y a todos aquellos que creyeron en mí y me motivaron a alcanzar mis objetivos.

Agradezco al Ing. Pablo, cuya orientación experta, rigurosidad académica y valiosas sugerencias fueron esenciales para el desarrollo de este trabajo.

A mi compañero de tesis Christian Estrada, gracias a su apoyo, esfuerzo y dedicación, elementos que fueron determinantes para culminar esta meta compartida.

Finalmente, expreso mi sincero agradecimiento a mis amigos y compañeros, quienes de una u otra manera contribuyeron a mi crecimiento personal, profesional y académico. Su acompañamiento y aliento durante este proceso han sido profundamente significativos para mí.

Santiago De Jesús Parra Reinoso

Tabla de Contenido

Índice de Figuras

Índice de Tablas

Desarrollo de Proceso Estándar de Aseguramiento de Calidad Mediante Automatización de Puntos de Control, En La Construcción de Software en la Cooperativa Jardín Azuayo.

AUTOR(ES):

CHRISTIAN RIGOBERTO ESTRADA ONCE
Y SANTIAGO DE JESÚS PARRA
REINOSO

Resumen

Esta investigación propone y valida un proceso estándar de aseguramiento de la calidad (QA) sustentado en la automatización de puntos de control a lo largo del ciclo de vida del software en la Cooperativa de Ahorro y Crédito Jardín Azuayo. Se adopta una metodología aplicada de alcance exploratorio–descriptivo, combinando técnica cuantitativa siendo estas pruebas piloto y métricas. La propuesta articula lineamientos técnicos, criterios de aceptación y umbrales integrados al pipeline CI/CD, incluyendo análisis estático, pruebas unitarias, de integración y validaciones de seguridad con el fin de reducir variabilidad, fuga de defectos y acelerar el tiempo de ciclo.

Los resultados muestran mejoras en indicadores como cobertura de pruebas, eficiencia de remoción de defectos (DRE), tiempo de ciclo y estabilidad de despliegues; además, se fortalece la trazabilidad entre requisitos, casos de prueba y entregables. Se concluye que la estandarización con automatización es viable y replicable en el contexto institucional y que su adopción progresiva acompañada de gestión del cambio y capacitación contribuye a sostener la calidad en el tiempo.

Palabras clave:

Aseguramiento de la calidad, Automatización, CI/CD, Puntos de control, Análisis estático, Pruebas automatizadas.

Abstract

This research proposes and validates a standard quality assurance (QA) process based on the automation of checkpoints throughout the software lifecycle at the Jardín Azuayo Savings and Credit Cooperative. An exploratory-descriptive methodology is adopted, combining quantitative techniques (pilot tests) and metrics. The proposal articulates technical guidelines, acceptance criteria, and thresholds integrated into the CI/CD pipeline, including static analysis, unit testing, integration testing, and security validations to reduce variability and defect leakage and accelerate cycle time.

The results show improvements in indicators such as test coverage, defect removal efficiency (DRE), cycle time, and deployment stability; in addition, traceability between requirements, test cases, and deliverables is strengthened. The conclusion is that standardization with automation is viable and replicable in the institutional context and that its progressive adoption, accompanied by change management and training, contributes to sustaining quality over time.

Palabras clave:
Iimperdiet, Phasellus vestibulum, Donec tristique, Morbi eros massa
Quality assurance, Automation, CI/CD, Control gates, Static analysis, Automated testing.

1 Introducción

En el ámbito del desarrollo de software, el aseguramiento de la calidad ha evolucionado desde prácticas empíricas hacia metodologías estandarizadas y basadas en métricas objetivas. La automatización de puntos de control dentro del ciclo de vida del software se ha convertido en un componente crítico para alcanzar niveles óptimos de eficiencia, confiabilidad y mantenimiento continuo.

En este capítulo se recogen diversos antecedentes investigativos, técnicos y normativos que respaldan el planteamiento de esta tesis. Se analizan trabajos previos enfocados en la automatización de pruebas, implementación de estándares de calidad y adopción de modelos de mejora en organizaciones públicas y privadas. Esta revisión no solo proporciona una base comparativa, sino que también identifica vacíos relevantes que la presente investigación busca cubrir.

1.1 Descripción del problema

En la actualidad, la Cooperativa de Ahorro y Crédito Jardín Azuayo implementa procesos de control de calidad en el desarrollo de sus aplicaciones web a través de planes de pruebas elaborados manualmente por el equipo de Investigación y Desarrollo, Calidad de aplicaciones y Arquitectura de Software. Este enfoque, si bien ha permitido mantener un nivel básico de validación funcional, presenta importantes limitaciones en términos de eficiencia, cobertura, trazabilidad y escalabilidad, especialmente ante el crecimiento continuo de sistemas y requerimientos (Fewster & Graham, 1999)., inclusive dejando validaciones importantes sin consideración.

Las pruebas manuales demandan tiempos prolongados en la mayoría de las veces más que el desarrollo de las aplicaciones web, están sujetas a errores humanos y dificultan la implementación de prácticas modernas como la entrega e integración continua (CI/CD) (Humble & Farley, 2010). Además, al no contar con mecanismos

automatizados para validar regresiones o verificar constantemente la calidad del código, se incrementa el riesgo de liberar versiones con defectos que afectan la experiencia del usuario o comprometen la integridad del sistema (Gerard, 2007).

1.2 Formulación del problema

¿Cómo validar que las funcionalidades preexistentes en el aplicativo no se vean comprometidas cuando se realizan nuevas incorporaciones o cambios en la aplicación web Historial Financiero evitando reprocesos y despliegues innecesarios en los servidores de aplicaciones de la Cooperativa Jardín Azuayo?

1.3 Justificación del problema

La calidad del software es un factor clave en el desarrollo de aplicaciones web, especialmente en organizaciones como la Cooperativa de Ahorro y Crédito Jardín Azuayo, donde los sistemas informáticos están directamente relacionados con la confiabilidad, eficiencia operativa y satisfacción del socio (Pressman & Maxim, 2020). En la actualidad, la cooperativa realiza sus procesos de verificación de calidad mediante planes de prueba manuales, los cuales, aunque funcionales, presentan limitaciones importantes en cuanto a escalabilidad, precisión y velocidad de validación (Fewster & Graham, Software test automation: Effective use of test execution tools, 1999).

Este proyecto se justifica en la necesidad de mejorar y modernizar el proceso de control de calidad en el desarrollo de software mediante la incorporación de herramientas y prácticas de automatización. La automatización de pruebas permite reducir significativamente el tiempo invertido en validaciones repetitivas, incrementar la cobertura de prueba, detectar errores de manera temprana y fomentar un ciclo de desarrollo más ágil y confiable (Gerard, 2007). Además, facilita la implementación de estrategias de integración y entrega continua (CI/CD), que hoy en día representan un estándar en el desarrollo de software profesional (Humble & Farley, 2010).

Al diseñar un conjunto de lineamientos técnicos y aplicarlos a través de un prototipo funcional en uno de los sistemas web de la cooperativa, se busca no solo validar la factibilidad técnica del enfoque, sino también demostrar su impacto positivo en la productividad del equipo y en la calidad final del software entregado. Este proyecto, por tanto, no solo aporta una solución técnica a un problema operativo, sino que también representa un paso estratégico hacia la modernización de las prácticas de desarrollo y aseguramiento de calidad dentro la institución.

1.4 Delimitación del problema

La investigación se enfoca en el área de desarrollo de software interno de la Cooperativa de Ahorro y Crédito Jardín Azuayo, específicamente en el análisis y mejora del proceso de aseguramiento de calidad mediante la automatización de puntos de control en el ciclo de vida del software.

El estudio se limita al sistema web Historial Financiero, desarrollado por la cooperativa, por ser una aplicación clave para validar mecanismos de control de calidad y reducir errores funcionales.

En cuanto al entorno tecnológico, se considera la adopción de herramientas compatibles con la infraestructura actual basada en Java y tecnologías web, priorizando soluciones open-source o comerciales de fácil integración, sin necesidad de alterar significativamente el sistema existente.

1.5 Objetivos

1.5.1 Objetivo general

Desarrollar un proceso estándar de aseguramiento de calidad, mediante la automatización de puntos de control en el ciclo de vida del software, para garantizar eficiencia y confiabilidad en la construcción de software en la Cooperativa de Ahorro y Crédito Jardín Azuayo.

1.5.2 Objetivos Específicos

a) Analizar el proceso actual de control de calidad en el desarrollo de aplicaciones web en la Cooperativa de Ahorro y Crédito Jardín Azuayo para identificar puntos de validación críticos que requieran ser automatizados.
b) Investigar, analizar y seleccionar herramientas apropiadas para la automatización de procesos de pruebas y validación de calidad, dentro de los entornos de desarrollo web en la Cooperativa de Ahorro y Crédito Jardín Azuayo.
c) Diseñar un conjunto de lineamientos estándar que definan buenas prácticas, herramientas, flujos de trabajo y criterios para la automatización de controles de calidad.
d) Desarrollar un prototipo que aplique los lineamientos de aseguramiento de calidad diseñados, en el sistema web Historial Financiero de la Cooperativa de Ahorro y Crédito Jardín Azuayo.
e) Documentar, medir y evaluar la efectividad en la automatización de los lineamientos implementados en términos de reducción de errores, mejora en tiempos de entrega continua y cumplimiento de estándares de calidad de software.

2 Marco teórico referencial

El presente capítulo tiene como finalidad establecer los fundamentos conceptuales y técnicos que sustentan la propuesta de desarrollo de un proceso estándar de aseguramiento de calidad mediante la automatización de puntos de control en el ciclo de vida del software. Para ello, se exponen teorías, modelos, normas y herramientas que respaldan la investigación, así como el análisis de buenas prácticas reconocidas en la industria del desarrollo de software.

En un entorno cada vez más exigente y competitivo, las organizaciones que desarrollan soluciones tecnológicas deben garantizar la calidad de sus productos mediante enfoques sistemáticos y estructurados. Particularmente en el sector financiero, donde los sistemas de información gestionan datos sensibles y operaciones críticas, el aseguramiento de la calidad no es solo un requisito técnico, sino también una necesidad estratégica.

El marco teórico aborda, en primer lugar, los principios del aseguramiento de la calidad del software, sus objetivos y su integración en el ciclo de vida del desarrollo. Posteriormente, se detalla la importancia de la automatización en los procesos de validación y verificación, así como los estándares internacionales aplicables, entre ellos ISO/IEC 25010. También se analizan herramientas especializadas para pruebas automatizadas y se exploran modelos de implementación como DevOps, el modelo V y CMMI (Capability Maturity Model Integration), que aportan una visión integral para estructurar procesos de calidad sostenibles y replicables.

Este capítulo establece, por tanto, los cimientos conceptuales que orientan el diseño e implementación del proceso propuesto en esta investigación, proporcionando un marco de referencia técnico coherente con las necesidades de calidad en el desarrollo de software de la Cooperativa Jardín Azuayo.

2.1 Fundamentos del Aseguramiento de la Calidad de Software

El Aseguramiento de la Calidad de Software (SQA, por sus siglas en inglés Software Quality Assurance) constituye un conjunto estructurado de actividades planificadas y sistemáticas que se incorporan en las distintas etapas del proceso de desarrollo de software, con el propósito de garantizar que tanto los productos como los procesos asociados cumplan los estándares de calidad establecidos. En esencia, este enfoque busca prevenir la aparición de errores y defectos en lugar de detectarlos una vez que se han manifestado, fortaleciendo la fiabilidad del producto final y optimizando los recursos técnicos y humanos del equipo de desarrollo (Pressman & Maxim, 2021).

De acuerdo con las definiciones propuestas por organismos internacionales especializados en ingeniería del software, el SQA representa un marco organizativo compuesto por políticas, procedimientos y recursos técnicos que aseguran el cumplimiento de normas, prácticas y directrices técnicas a lo largo de todo el ciclo de vida del producto. Dicho marco abarca desde la identificación de requisitos, pasando por las etapas de diseño, construcción y validación, hasta llegar al mantenimiento posterior a la puesta en producción, configurándose, así como un sistema integral de control y mejora continua (IEEE, 2014).

El aseguramiento de la calidad contempla múltiples dimensiones de evaluación, entre las que destacan:

- Confiabilidad, entendida como la capacidad del sistema para ejecutar sus funciones bajo condiciones específicas durante un periodo determinado.
- Usabilidad, que hace referencia a la facilidad con la que los usuarios aprenden y utilizan la aplicación para alcanzar sus objetivos.
- Mantenibilidad, relacionada con la facilidad para modificar, adaptar o corregir el software después de su entrega.

- Portabilidad, asociada con la posibilidad de trasladar el sistema a diferentes entornos tecnológicos sin afectar su funcionamiento (International Organization for Standardization, 2011).

Para alcanzar estos atributos, el SQA se apoya en modelos de calidad y estándares internacionales que proporcionan lineamientos y criterios de evaluación. Entre los más relevantes se encuentran:

- ISO/IEC 25010, que define un modelo de calidad del producto de software, estableciendo características y subcaracterísticas medibles.
- ISO/IEC 12207, que describe los procesos del ciclo de vida del software, promoviendo la estandarización de actividades.
- CMMI (Capability Maturity Model Integration), orientado a medir el grado de madurez de los procesos de desarrollo y a establecer estrategias para su mejora continua (CMMI Institute, 2018).

Conviene subrayar que el SQA no se limita exclusivamente a las pruebas de software. Abarca un espectro más amplio de actividades, como auditorías técnicas, revisiones formales, seguimiento de métricas, documentación de procesos y la implementación de controles automatizados que permiten detectar desviaciones en tiempo real. Esta combinación de acciones proporciona una visión integral del estado del producto y de la eficacia de los procedimientos aplicados (Sommerville, 2016).

En organizaciones que priorizan la mejora continua en la entrega de soluciones tecnológicas, el aseguramiento de la calidad adquiere un carácter estratégico. Su incorporación desde las etapas iniciales del desarrollo posibilita reducir el reproceso, incrementar la satisfacción del usuario final y fortalecer la estabilidad operativa del sistema. En el caso de la Cooperativa de Ahorro y Crédito Jardín Azuayo, la adopción de un proceso formal de SQA se convierte en un elemento clave para consolidar la eficiencia del desarrollo de software, mantener la consistencia de los resultados y asegurar la sostenibilidad de las soluciones implementadas dentro de su entorno tecnológico (Pressman & Maxim, 2021).

2.2 Automatización de Puntos de Control en el Ciclo de Vida del Software

La automatización de puntos de control constituye una estrategia fundamental para fortalecer el aseguramiento de la calidad del software. Esta práctica permite incorporar validaciones sistemáticas y repetibles dentro de las distintas fases del ciclo de vida del desarrollo, garantizando la detección temprana de defectos y el cumplimiento de estándares técnicos definidos por la organización (Pressman & Maxim, 2021).

En el contexto actual de desarrollo ágil y continuo, caracterizado por entregas frecuentes y la integración de equipos multidisciplinarios, los procesos manuales de verificación resultan insuficientes para sostener la calidad de manera consistente. Frente a ello, la automatización de controles posibilita establecer una cadena de validaciones permanente y trazable, en la que cada fase del ciclo de vida cuenta con puntos de control predefinidos que aseguran el cumplimiento de objetivos de calidad antes de avanzar a la siguiente etapa (IEEE Standards Association, 2021).

Estos puntos de control, también denominados quality gates, se implementan mediante herramientas de integración y entrega continua (CI/CD), donde se configuran etapas automatizadas de revisión, compilación, prueba y despliegue. Cada punto representa un criterio técnico verificable que determina si un artefacto cumple las condiciones necesarias para ser promovido a la siguiente fase; por ejemplo, el análisis estático del código, la detección de vulnerabilidades de seguridad o la evaluación del rendimiento. De esta manera, se garantiza que únicamente las versiones que cumplen los estándares avancen dentro del flujo de desarrollo (International Organization for Standardization & International Electrotechnical Commission, 2017).

El valor de esta estrategia radica en impulsar un aseguramiento de la calidad preventivo y proactivo. En efecto, la incorporación temprana de controles principio conocido como Shift-Left acorta los ciclos de retroalimentación, disminuye los

costos de corrección y previene defectos que, de otro modo, se evidenciarían en fases posteriores, con mayor impacto operativo y financiero (Pressman & Maxim, 2021).

Desde una perspectiva técnica, la automatización de puntos de control suele implementarse a través de herramientas como GitLab CI/CD, Jenkins o Azure DevOps, que permiten configurar pipelines de construcción y despliegue continuo. En estos flujos se integran linters (por ejemplo, ESLint o PMD), analizadores estáticos (por ejemplo, SonarQube) y frameworks de pruebas unitarias e integrales (como JUnit, NUnit, PyTest o Cypress). La combinación de estas herramientas genera un ecosistema automatizado que monitorea la calidad en tiempo real, produce reportes de cumplimiento y bloquea la liberación de componentes que no satisfacen los criterios definidos (Sommerville, 2016).

Además de su beneficio técnico, la automatización de puntos de control aporta ventajas de gestión y gobernanza. Por un lado, fomenta la trazabilidad de las decisiones al dejar registro de los resultados de cada validación; por otro, fortalece la transparencia del proceso, ya que toda acción queda documentada dentro del pipeline, facilitando la auditoría interna y externa. De igual modo, permite medir la efectividad de las mejoras introducidas, comparando métricas antes y después de la automatización (tiempo de ciclo, eficiencia en la remoción de defectos o estabilidad de los despliegues).

En entornos organizacionales como la Cooperativa de Ahorro y Crédito Jardín Azuayo, la adopción de esta práctica constituye un paso decisivo hacia la madurez del proceso de desarrollo de software. Al integrar controles automatizados en las fases de construcción, prueba e implementación, la cooperativa fortalece su capacidad para entregar productos confiables, consistentes y sostenibles, alineados con los principios de mejora continua y optimización de recursos.

2.3 Buenas Prácticas y Estándares de Calidad

El desarrollo de software orientado a la calidad requiere la adopción de buenas prácticas y estándares internacionales que permitan establecer un marco común para planificar, ejecutar y evaluar las actividades a lo largo del ciclo de vida del producto. Estos referentes promueven la uniformidad de procesos, incrementan la fiabilidad de los resultados y facilitan la comparación y mejora continua entre proyectos y organizaciones (Sommerville, 2016).

Entre los principales marcos normativos se encuentra la ISO/IEC 25010, que propone un modelo de calidad del producto y de su uso, estructurado en características medibles como la eficiencia del desempeño, la compatibilidad, la seguridad, la usabilidad, la mantenibilidad y la portabilidad. Este modelo ofrece una guía objetiva para evaluar los atributos internos y externos del software, lo cual permite a los equipos de desarrollo definir metas de calidad verificables desde las etapas iniciales del proyecto (International Organization for Standardization, 2011).

De manera complementaria, la ISO/IEC 12207 describe los procesos del ciclo de vida del software, estableciendo las actividades, entradas, salidas y responsabilidades asociadas a cada fase. Su aplicación fomenta la estandarización de tareas y la trazabilidad de resultados, garantizando que todas las etapas dentro del desarrollo del software se ejecuten conforme a prácticas reconocidas y reproducibles (International Organization for Standardization & International Electrotechnical Commission, 2017).

Por su parte, el modelo CMMI (Capability Maturity Model Integration) ofrece un marco para evaluar la madurez de los procesos organizacionales y promover una cultura de mejora continua. Este enfoque permite identificar fortalezas, debilidades y oportunidades de optimización en las prácticas de desarrollo, contribuyendo a la reducción de defectos y al control efectivo del riesgo (CMMI Institute, 2018).

Adicionalmente, las buenas prácticas de la industria del software, impulsadas por comunidades como IEEE (Institute of Electrical and Electronics Engineers) o ISTQB

(Software Testing Qualifications Board), complementan estos estándares mediante guías sobre gestión de pruebas, documentación técnica, métricas de calidad y control de versiones. Su adopción permite construir procesos más transparentes y auditables, ya que promueven la generación sistemática de evidencias que sustentan las decisiones de liberación.

En conjunto, estos marcos normativos y metodológicos proporcionan una base estructurada para la gestión integral de la calidad. Su aplicación coherente contribuye a minimizar la variabilidad del proceso, fortalecer la trazabilidad de los artefactos y garantizar la confiabilidad del producto final. Así, las organizaciones logran no solo cumplir con estándares técnicos, sino también generar una ventaja competitiva sostenida mediante la mejora continua y la madurez de sus procesos de desarrollo.

2.4 Procesos de Calidad en Entornos Financieros

En el ámbito financiero, la calidad del software adquiere un valor estratégico, ya que los sistemas deben garantizar la integridad de la información, la seguridad de las transacciones y la disponibilidad continua de los servicios. A diferencia de otros sectores, las aplicaciones financieras operan bajo altos niveles de exigencia regulatoria y operativa, lo que exige procesos de aseguramiento de la calidad más rigurosos, medibles y sostenibles (Basili, Caldiera, & Rombach, 2010).

La naturaleza crítica de las operaciones bancarias y cooperativas demanda que cada componente de software sea confiable y predecible. En este contexto, los procesos de calidad se diseñan para prevenir interrupciones, pérdidas de datos o fallos funcionales que puedan afectar directamente la estabilidad institucional o la confianza del usuario. Por tanto, la gestión de la calidad en entornos financieros no se limita al cumplimiento técnico, sino que constituye un mecanismo de gestión del riesgo tecnológico (International Organization for Standardization, 2011).

Estos sistemas requieren un enfoque integral que abarque tanto los aspectos funcionales precisión, consistencia y trazabilidad de las operaciones como los atributos no funcionales, entre ellos la seguridad, el rendimiento y la resiliencia. Para ello, las organizaciones financieras implementan políticas de control que permitan detectar vulnerabilidades, reducir la exposición a incidentes y mantener la continuidad del servicio incluso ante condiciones adversas (International Organization for Standardization & International Electrotechnical Commission, 2017).

En la práctica, las entidades del sector aplican pruebas automatizadas y controles de calidad continúa integrados a sus flujos de desarrollo, con el fin de validar el cumplimiento de normativas como PCI DSS e ISO/IEC 27001, que establecen lineamientos de seguridad de la información y protección de datos. Estas pruebas incluyen desde evaluaciones de integridad transaccional hasta análisis de rendimiento y pruebas de estrés, garantizando que el sistema mantenga un comportamiento estable y seguro en escenarios de alta demanda (PCI Security Standards Council, 2022).

Asimismo, la incorporación de puntos de control automatizados dentro de los pipelines de integración continua (CI) permite mantener una verificación constante del cumplimiento de los criterios de calidad definidos. Esta práctica asegura que cada versión del software sea desplegada únicamente cuando haya superado las pruebas y revisiones correspondientes, reduciendo el riesgo de liberar componentes con defectos o vulnerabilidades (CMMI Institute, 2018).

En el caso de la Cooperativa de Ahorro y Crédito Jardín Azuayo, la automatización del aseguramiento de la calidad representa una herramienta clave para fortalecer la confianza institucional, optimizar los recursos técnicos y minimizar errores humanos en procesos críticos como la gestión de cuentas, créditos y transacciones. De este modo, la calidad del software se convierte en un factor diferenciador, que no solo garantiza la eficiencia operativa, sino que también contribuye a la reputación, sostenibilidad tecnológica y cumplimiento normativo de la entidad (Kim, G., Humble, J., Debois, P., & Willis, J., 2016).

2.5 Herramientas para Automatización de la Calidad

El proceso de automatización de la calidad del software requiere un conjunto de herramientas complementarias que permitan ejecutar, medir y supervisar las actividades de aseguramiento en cada etapa del ciclo de vida del producto. Estas herramientas facilitan la integración de controles automáticos dentro de los flujos de desarrollo y despliegue, garantizando la detección oportuna de errores, el cumplimiento de estándares técnicos y la generación de evidencia trazable sobre el estado de calidad del sistema (Meszaros, 2007).

De manera general, las herramientas de automatización pueden clasificarse en cuatro categorías principales: automatización funcional, pruebas unitarias, análisis estático y orquestación de procesos CI/CD. Cada una cumple un rol específico dentro del ecosistema de desarrollo, pero su verdadero valor se manifiesta cuando actúan de forma coordinada y continua.

En primer lugar, la automatización funcional permite validar el comportamiento del software desde la perspectiva del usuario final. Frameworks como Selenium o Cypress simulan interacciones reales con la interfaz de la aplicación, verificando la correcta ejecución de los flujos de negocio críticos. Estas herramientas posibilitan la ejecución repetitiva de pruebas sobre distintos navegadores y entornos, asegurando la consistencia funcional ante cambios de código o actualizaciones tecnológicas (International Software Testing Qualifications Board, 2020).

En segundo lugar, las pruebas unitarias constituyen la base del control de calidad a nivel de código. Frameworks como JUnit, NUnit o PyTest permiten evaluar de manera aislada cada componente del sistema, verificando su comportamiento frente a diferentes entradas y condiciones. Este enfoque reduce la aparición de defectos en fases posteriores y fortalece la estabilidad del proceso de integración (Pressman & Maxim, 2021).

En tercer lugar, el análisis estático de código es una práctica orientada a evaluar la calidad estructural del software sin necesidad de ejecutarlo. Herramientas como SonarQube, ESLint o PMD detectan vulnerabilidades, incumplimiento de convenciones de estilo y posibles errores lógicos, generando reportes cuantificables de mantenibilidad, seguridad y complejidad del código. Su integración dentro del pipeline de CI/CD permite establecer umbrales mínimos de calidad que deben cumplirse antes de la liberación de cualquier versión (International Organization for Standardization, 2011) (IEEE Standards Association, 2021).

En cuarto lugar, las plataformas de integración y entrega continua (CI/CD) como GitLab CI/CD, Jenkins o Azure DevOps desempeñan un papel clave al automatizar la compilación, ejecución de pruebas y despliegue de versiones. Estos sistemas permiten definir pipelines configurables, donde cada etapa ejecuta tareas específicas de validación, generando reportes automáticos y registros de auditoría que aseguran la trazabilidad completa del proceso (CMMI Institute, 2018).

Finalmente, la integración conjunta de estas herramientas genera un ecosistema de aseguramiento de calidad automatizado, que no solo mejora la eficiencia operativa, sino que también promueve la trazabilidad, la reproducibilidad de los resultados y la madurez organizacional del proceso de desarrollo. Este entorno técnico se convierte en la base sobre la cual se construye un modelo de calidad sostenible, alineado con los principios de mejora continua y excelencia operativa.

3 Desarrollo del proyecto

3.1 Introducción

Este capítulo describió el desarrollo del proceso de aseguramiento de la calidad de software implementado en la Cooperativa de Ahorro y Crédito Jardín Azuayo, centrándose en la selección, evaluación e integración de herramientas destinadas a la automatización de pruebas y al análisis estático del código. La intervención se concibió como una proyección práctica de los fundamentos teóricos expuestos en el marco conceptual y se aplicó sobre el entorno real de desarrollo de la institución.

En este marco, se precisaron los criterios técnicos y funcionales que orientaron la elección de las soluciones adoptadas, y se documentaron las etapas de instalación, configuración e incorporación de dichas herramientas al flujo de trabajo del equipo. Del mismo modo, se establecieron mecanismos de validación para comprobar el alineamiento con los estándares de calidad internos, asegurando coherencia entre las prácticas de aseguramiento y los objetivos estratégicos del área tecnológica.

Finalmente, se fundamentó la elección de Selenium como herramienta principal para la automatización de pruebas funcionales, dado que acreditó compatibilidad con la infraestructura vigente, ofreció flexibilidad frente a los requisitos funcionales y no funcionales del proyecto, y demostró una integración efectiva con los sistemas de integración continua ya desplegados. Esta decisión marcó un avance en la madurez del proceso de desarrollo, al incorporar capacidades de automatización que fortalecieron la eficiencia, la trazabilidad y la fiabilidad del software institucional.

3.2 Herramientas para Automatización de Pruebas

Con el propósito de asegurar eficiencia, precisión y confiabilidad en la ejecución de pruebas, se llevó a cabo una comparación técnica entre tres herramientas de

referencia para aplicaciones web: Selenium, Cypress y Playwright. Cada solución se analizó a partir de criterios técnicos y operativos, considerando la compatibilidad con los lenguajes utilizados en los sistemas institucionales, las tecnologías backend y frontend implementadas por la Cooperativa de Ahorro y Crédito Jardín Azuayo, así como la facilidad de integración en entornos Docker y su articulación con pipelines de integración continua (CI).

De forma complementaria, se valoraron la facilidad de adopción, el soporte de la comunidad, la documentación disponible y la capacidad de cada herramienta para responder a los requisitos funcionales y no funcionales definidos por la organización. Este proceso comparativo permitió identificar fortalezas y limitaciones, así como delimitar los escenarios de uso más convenientes para cada tecnología en el contexto operativo de la cooperativa, sentando las bases para seleccionar la alternativa más adecuada para el modelo de automatización de pruebas.

3.2.1 Cypress

Cypress se caracterizó por ser una herramienta moderna de automatización orientada a pruebas end-to-end (E2E) para aplicaciones web, diseñada para facilitar la ejecución, la depuración y el mantenimiento de escenarios funcionales. Operó directamente sobre el navegador, lo que permitió observar interacciones en tiempo real y obtener retroalimentación inmediata del comportamiento del sistema en cada corrida (Cypress.io, s.f.).

Su arquitectura se sustentó en JavaScript con integración nativa en Node.js, rasgo que favoreció su adopción en proyectos frontend construidos con React, Angular o Vue.js. Esta compatibilidad convirtió a Cypress en una alternativa natural para entornos centrados en tecnologías JavaScript, al habilitar una transición fluida entre desarrollo, pruebas e integración continua (Rauschmayer, A., 2019).

Entre los beneficios observados destacaron la configuración sencilla, la rapidez de ejecución y la interfaz gráfica que facilitó la visualización de resultados y el diagnóstico de fallos. Asimismo, su comunidad activa y la documentación

actualizada agilitaron la resolución de incidencias y la extensión de funcionalidades (Cypress.io, s.f.).

Con todo, se identificó como limitación su enfoque limitado al ecosistema JavaScript, aspecto que condicionó su uso en arquitecturas mixtas con backends no JavaScript. En el caso de la Cooperativa de Ahorro y Crédito Jardín Azuayo con backend en Java y frontend en React, Cypress ofreció plena compatibilidad a nivel de interfaz, pero exigió adecuaciones adicionales para cubrir lógica de servidor o pruebas de integración de mayor complejidad.

Pese a ello, Cypress se consolidó como una solución eficiente y eficaz para pruebas funcionales web en contextos dominados por JavaScript. Su integración en pipelines de CI permitió ejecutar casos automáticamente ante cada commit, reforzando la agilidad y la confiabilidad en la entrega de software.

3.2.2 Playwright

Playwright se evaluó como un framework de automatización para pruebas web que destacó por su soporte multiplataforma y su capacidad de orquestar múltiples navegadores en paralelo. Desarrollado por Microsoft, ofreció una API unificada que permitió ejecutar pruebas de forma coherente sobre Chromium, Firefox y WebKit, rasgo que lo posicionó como una alternativa versátil para escenarios con exigencias cross-browser (Playwright.dev, s.f.).

Una de sus fortalezas principales fue la espera automática (auto-waiting): los comandos sincronizaron de manera interna la interacción hasta que los elementos estuvieron listos, reduciendo la dependencia de timeouts explícitos y la fragilidad asociada a interfaces con tiempos de carga variables (Playwright.dev, s.f.). Gracias a esta capacidad, los guiones de prueba resultaron más robustos frente a cambios temporales y contenido dinámico.

En términos de adopción tecnológica, Playwright brindó soporte para JavaScript, TypeScript, Python, C# y Java, lo que facilitó su integración en stacks heterogéneos (Playwright.dev, s.f.). En el contexto de la Cooperativa de Ahorro y Crédito Jardín

Azuayo, esta amplitud permitió alinear la herramienta con un frontend en React y servicios backend en Java, evitando fragmentar el ecosistema de testing.

Asimismo, el framework integró capacidades avanzadas de trazado de ejecución (trace), captura de video y tomas de pantalla automáticas, útiles para trazabilidad y diagnóstico; además, habilitó la ejecución en modo headless, lo que favoreció su uso en entornos de integración continua (CI) sin interfaz gráfica (HeadSpin, 2024). Estas prestaciones facilitaron su incorporación a pipelines CI/CD, ejecutando pruebas en cada build sin intervención manual.

Con todo, se observaron retos vinculados a la curva de aprendizaje en escenarios heterogéneos y a la configuración minuciosa del entorno en casos complejos. Para la Cooperativa de Ahorro y Crédito Jardín Azuayo, si bien la compatibilidad técnica lo hizo viable, la interacción con servicios backend (Java y microservicios) pudo requerir adaptaciones y entornos simulados cuidadosamente definidos para alcanzar una cobertura adecuada.

En síntesis, Playwright se perfiló como una opción sólida y flexible para la automatización de pruebas web: soporte cross-browser, auto-waiting, multilenguaje y herramientas de trazabilidad respaldaron su idoneidad en sistemas web modernos con ambientes variados y demandas de calidad integradas.

3.2.3 Selenium

Selenium se consideró una de las herramientas más consolidadas para la automatización de pruebas funcionales en aplicaciones web. Su arquitectura multicomponente WebDriver, IDE y Grid permitió ejecutar pruebas en distintos navegadores, sistemas operativos y entornos distribuidos, lo que explicó su amplia adopción como estándar dentro de los procesos de aseguramiento de la calidad de software (Sommerville, 2016).

A diferencia de otras alternativas, Selenium ofreció compatibilidad con múltiples lenguajes de programación (Java, Python, C#, JavaScript, Ruby), rasgo que facilitó su incorporación en proyectos con arquitecturas heterogéneas. Asimismo, soportó

la ejecución sobre navegadores como Chrome, Firefox, Edge y Safari, reforzando su utilidad en escenarios empresariales donde la cobertura multiplataforma resultó crítica (Pressman & Maxim, 2021).

Otra fortaleza relevante radicó en su integración nativa con herramientas de orquestación y control de versiones GitLab CI/CD, Jenkins y Docker, lo que habilitó la ejecución de pruebas automatizadas dentro de pipelines de integración continua. De este modo, se aseguró una validación constante a medida que se generaron nuevas versiones del código, promoviendo un enfoque de entrega continua y un control de calidad preventivo (International Software Testing Qualifications Board, 2020).

En paralelo, Selenium contó con una comunidad amplia, bibliotecas de soporte y documentación actualizada, factores que simplificaron su mantenimiento y favorecieron la adaptación a diversos entornos tecnológicos. Estas características respaldaron su uso como solución confiable y escalable en organizaciones que buscaron estabilidad y eficiencia en sus procesos de desarrollo.

La selección tecnológica se ajustó a la arquitectura, los lenguajes y la madurez del proceso. En la Cooperativa de Ahorro y Crédito Jardín Azuayo, se dispuso de Java en backend y Node.js/React en frontend, con componentes de interfaz como DataTables, pop-ups, menús laterales, paneles, acordeones, botones y campos de texto. A la vez, GitLab operó como repositorio central y orquestador de pipelines de Integración Continua (CI), mientras que Docker se utilizó para la gestión estandarizada de entornos. Bajo este marco, Selenium se consolidó como la alternativa más pertinente por su compatibilidad multiplataforma, su integración con flujos CI/CD y su ecosistema activo de comunidad y documentación; además, su licenciamiento de código abierto facilitó la adopción institucional y favoreció la escalabilidad del modelo propuesto.

Por contraste, herramientas más recientes como Cypress o Playwright mostraron ventajas en proyectos construidos exclusivamente sobre JavaScript/TypeScript, especialmente por su rapidez y capacidades de paralelismo. Sin embargo, en

ecosistemas mixtos como el de la Cooperativa de Ahorro y Crédito Jardín Azuayo, Selenium ofreció un equilibrio más favorable entre costo operativo, curva de aprendizaje y robustez técnica, lo que reforzó su idoneidad para el contexto analizado.

En síntesis, Selenium se presentó como una solución madura, estable y escalable, capaz de integrarse eficazmente en entornos tecnológicos diversos y de responder a las necesidades de control de calidad dentro de un proceso de desarrollo ágil y sostenible.

Criterio	Selenium	Cypress	Playwright
Lenguajes soportados	Multilenguaje: Java, Python, C#, JavaScript, Kotlin, Ruby, entre otros. Se acopla a las tecnologías usadas en la Cooperativa Jardín Azuayo (Java en backend y Node.js/React en frontend).	Solo soporta JavaScript y TypeScript, lo que limita su adopción en entornos mixtos como el de la organización.	Principalmente JavaScript, TypeScript, Python, Java y C#. Aunque es más flexible que Cypress, su comunidad multilenguaje es más reciente.
Compatibilidad con frameworks frontend	Excelente integración con aplicaciones React, soportando componentes dinámicos: datatable, popups, accordions, menús laterales, paneles, botones y textbox.	Muy buena compatibilidad con React y Angular, pero su enfoque está limitado al ecosistema JS.	Alta compatibilidad con React y Angular, también con Vue; aunque más reciente en validación de componentes complejos.
Aplicaciones soportadas	Aplicaciones web dinámicas y responsivas, con ejecución en múltiples navegadores y dispositivos (móviles y escritorio).	Orientado principalmente a Chrome y navegadores basados en Chromium; soporte limitado en dispositivos móviles reales.	Soporta navegadores modernos (Chromium, WebKit, Firefox) y simulación de entornos móviles.

Curva de aprendizaje	Baja, amplia documentación, ejemplos y comunidad. Equipos de desarrollo/QA pueden adaptarse rápidamente.	Baja-media. Fácil de usar para JS developers, pero difícil para equipos que trabajan con otros lenguajes.	Media. API clara, pero requiere mayor familiaridad con JS/TS y conceptos de concurrencia.
Comunidad y soporte	Amplia, con más de una década de desarrollo y múltiples foros, librerías y extensiones.	Comunidad creciente, pero mucho más limitada que Selenium.	Comunidad en crecimiento, impulsada por Microsoft, pero aún menor que Selenium.
Costo	Gratis open source.	Gratis open source.	Gratis open source.
Integración con CI/CD	Integración robusta con GitLab CI/CD, Jenkins, Bamboo, Azure DevOps, entre otros.	Integración con GitHub Actions, GitLab CI y Jenkins, aunque con limitaciones en ejecución distribuida.	Buena integración con pipelines CI/CD, con soporte oficial para GitHub Actions y Azure Pipelines.
Escalabilidad	Altamente escalable con Selenium Grid y contenedores Docker para ejecutar pruebas en paralelo.	Escalabilidad limitada, no nativa; requiere configuraciones adicionales.	Buena escalabilidad con soporte paralelo nativo, pero aún menos probado en entornos empresariales grandes.
Idoneidad en la Cooperativa Jardín Azuayo	Óptima compatibilidad con tecnologías actuales (Java, Node.js, React), bajo costo, curva de aprendizaje reducida, y facilidad de integración con Docker y GitLab ya utilizados en la organización.	Parcial, útil si el ecosistema fuera 100% JavaScript/TypeScript, pero no se ajusta a la mezcla tecnológica de la organización.	Baja, ofrece flexibilidad, pero la curva de aprendizaje y menor adopción en el mercado y arquitectura de aplicaciones dentro de Jardín Azuayo, lo hacen menos conveniente actualmente.

Tabla 1. Comparación entre Cypress, Playwright y Selenium

3.3 Herramienta para Análisis de Código Estático

En la ingeniería de software contemporánea, uno de los retos centrales consistió en sostener la calidad y la seguridad del producto sin comprometer la velocidad de entrega. En ese marco, el enfoque Shift-Left aplicado en la cultura DevOps se adoptó como estrategia clave para reforzar el aseguramiento de la calidad: se trasladaron actividades críticas pruebas unitarias, de integración, de desempeño y de seguridad a etapas tempranas del ciclo, con el fin de detectar defectos y vulnerabilidades antes de que escalaran en costo y retrabajo (Red Hat, 2023).

La aplicación del enfoque se apoyó en la automatización de pruebas dentro de procesos de integración continua (CI) y entrega continua (CD), lo que proveyó retroalimentación inmediata a los desarrolladores. Mediante herramientas de continuous testing (pruebas continuas), los equipos ejecutaron de manera sistemática casos unitarios, funcionales, de regresión y de seguridad ante cada cambio, aumentando la confiabilidad y reduciendo riesgos de fallos en producción; con ello, las pruebas pasaron a ser un componente permanente del pipeline y no una verificación tardía (Roadmap.sh, s.f.).

En consecuencia, la adopción del Shift-Left dentro de DevOps permitió acelerar entregas de software sin sacrificar calidad, articulando pruebas tempranas con CI/CD para contener costos, tiempo y elevar la estabilidad y seguridad del producto alineados con los requerimientos del negocio.

Bajo este enfoque, la incorporación de ESLint como herramienta de análisis estático se definió como un mecanismo esencial para fortalecer el control de calidad desde el inicio del desarrollo. Allí donde tradicionalmente la detección de errores sintácticos, inconsistencias de estilo o vulnerabilidades menores ocurría en etapas posteriores, la integración de ESLint en el flujo de trabajo posibilitó identificar tales defectos en tiempo real, directamente en el entorno de desarrollo (OpenJS Foundation, 2025).

En la práctica, ESLint actuó como control preventivo que veló por la conformidad con estándares institucionales, señaló malas prácticas e inconsistencias y anticipó errores potenciales antes de que el software progresara a fases superiores del pipeline. De este modo, garantizó una validación constante del estilo y de las convenciones, promovió uniformidad entre desarrolladores y contribuyó a la mantenibilidad. Además, al detectar variables no usadas, declaraciones redundantes y posibles fallos lógicos, redujo defectos susceptibles de manifestarse en producción (Wang, Ali, & Petersen, 2020).

Otra ventaja determinante radicó en su integración continua: ESLint se ejecutó automáticamente durante la escritura del código y dentro de las etapas de CI/CD, validando cada commit antes de integrarse a la rama principal. Este comportamiento aseguró umbrales de calidad predefinidos, eliminó discrepancias entre equipos y sostuvo un estilo homogéneo de programación.

Asimismo, ESLint ofreció flexibilidad y productividad mediante reglas personalizables y amplia compatibilidad con editores en particular, Visual Studio Code, lo que facilitó la detección inmediata y la corrección in situ. Esta estandarización mejoró la colaboración, disminuyó la deuda técnica y elevó la calidad global del producto.

En conjunto, estos elementos convirtieron a ESLint en un pilar operativo del Shift-Left dentro del pipeline enfocado en DevOps: se configuró un ecosistema de validaciones preventivas donde la calidad dejó de ser un control final y pasó a un proceso continuo y automatizado. En síntesis, ESLint resultó efectivo para garantizar que los equipos no solo generaran código funcional, sino también consistente, seguro y alineado con las normas de calidad institucional. Por esta razón, se implementó como herramienta principal de análisis estático en el entorno Visual Studio Code de la Cooperativa de Ahorro y Crédito Jardín Azuayo, con especial énfasis en proyectos basados en React.

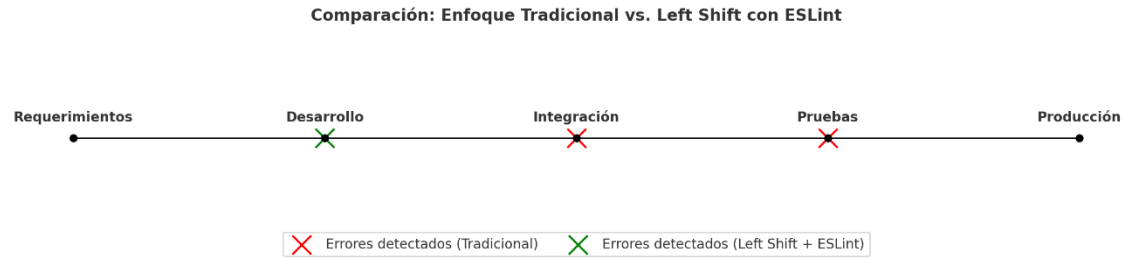


Figura. 1 Left Shift con ESLint

La instalación y configuración de ESLint en el entorno de desarrollo constituyeron un hito clave para asegurar que el análisis estático del código se ejecutara de forma automática y consistente a lo largo de todo el ciclo de vida del proyecto. Su integración con Visual Studio Code (VS Code) permitió que las validaciones de estilo, sintaxis y calidad ocurrieran en tiempo real, impulsando una cultura de prevención de errores desde la misma escritura del código fuente.

Previo a la instalación, se verificaron los requisitos básicos:

1. Node.js y su gestor npm, dado que ESLint se distribuyó como paquete del ecosistema Node; y
2. Visual Studio Code, definido como el IDE principal donde se ejecutaría el análisis.

Con estas condiciones cumplidas, el proceso se inició desde el Marketplace de VS Code, buscando e instalando la extensión oficial ESLint mantenida por la OpenJS Foundation. Dicha extensión habilitó la detección inmediata de errores y advertencias directamente en el editor, mostrando señales visuales que facilitaron la corrección temprana de inconsistencias y el alineamiento con las reglas establecidas (OpenJS Foundation, 2025).

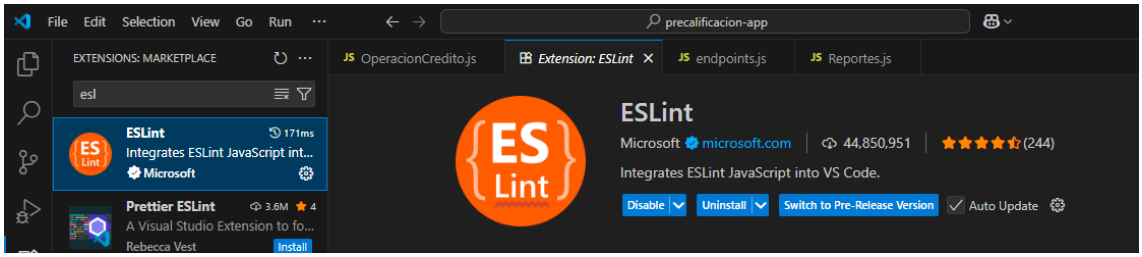


Figura. 2 Instalación ESLint en VS Code

Posteriormente, se incorporó ESLint como dependencia de desarrollo ejecutando el comando *npm install eslint --save-dev* en la raíz del proyecto. A continuación, se generó una configuración base mediante *npx eslint --init*, con lo cual se creó el archivo de definición (eslint.config.js o .eslintrc.json) donde se establecieron las reglas, los entornos y las extensiones a emplear.

Una vez disponible el archivo eslint.config.js, se procedió a personalizar su contenido conforme a las necesidades específicas del frontend de la Cooperativa de Ahorro y Crédito Jardín Azuayo. Esta adaptación tuvo por finalidad ajustar reglas y parámetros de análisis estático a las particularidades tecnológicas institucionales, asegurando la alineación del código con las buenas prácticas definidas. En esa línea, se implementó la configuración definitiva en el archivo, construida según los requerimientos técnicos y operativos vigentes para los proyectos web de la cooperativa.

```
import pluginPrettier from "eslint-plugin-prettier";
import pluginJs from "@eslint/js";
import pluginReact from "eslint-plugin-react";
import pluginReactHooks from "eslint-plugin-react-hooks";
import globals from "globals";

export default [
  {
    files: ["**/*.{js,jsx}"],
    languageOptions: {
      ecmaVersion: 2022,
      sourceType: "module",
      globals: {
        ...globals.browser,
      },
      parserOptions: {
        ecmaFeatures: {
          jsx: true,
        },
      },
    },
    plugins: {
      react: pluginReact,
      "react-hooks": pluginReactHooks,
      prettier: pluginPrettier,
    },
    rules: {
      ...pluginJs.configs.recommended.rules,
      ...pluginReact.configs.flat.recommended.rules,
      "prettier/prettier": "error",
      "react-hooks/rules-of-hooks": "error",
      "react-hooks/exhaustive-deps": "warn",

      "no-unused-vars": [
        "error",
        {
          vars: "all",
          args: "after-used",
          ignoreRestSiblings: true,
          caughtErrors: "none",
        },
      ],

      "react/jsx-uses-react": "off",
      "react/react-in-jsx-scope": "off",

      "no-console": "warn",
      "no-debugger": "error",
      eqeqeq: ["error", "always"],
      curly: ["error", "all"],
    },
    settings: {
      react: {
        version: "detect",
      },
    },
  },
];
```

Figura. 3 Configuración archivo eslint.config.js

Con el propósito de fortalecer el aseguramiento de la calidad en el entorno de desarrollo de la Cooperativa de Ahorro y Crédito Jardín Azuayo, se implementó una configuración personalizada del archivo eslint.config.js (véase Figura. 3). Este archivo definió las reglas, los parámetros y los entornos necesarios para ejecutar el análisis estático en proyectos basados en React y JavaScript. A continuación, se presenta la explicación técnica de su contenido y la justificación de cada componente.

1. **Ámbito de aplicación:**

   La configuración estableció explícitamente el alcance del análisis sobre todos los archivos JavaScript y JSX del proyecto (files: ["**/*.{js,jsx}"]). Con ello, cualquier componente de interfaz, módulo utilitario o script dentro de la jerarquía del repositorio quedó sujeto a las mismas políticas de calidad. Esta decisión evitó zonas fuera del linting (análisis estático) y garantizó criterios de validación coherentes y homogéneos en todo el frontend institucional.

2. **Opciones de lenguaje y parser:**

   En languageOptions, el proyecto adoptó:

   - ECMAScript 2022 (ecmaVersion: 2022) y módulos ES (*sourceType: "module"*), habilitando sintaxis moderna y patrones de importación acordes con la herramienta de empaquetado.
   - La activación de JSX (*parserOptions.ecmaFeatures.jsx: true*) que permitió analizar correctamente componentes React y su combinación entre marcado y lógica.
   - La declaración de variables globales del navegador (*globals: { ...globals.browser }*) anticipando entidades como window y document, para evitar falsos positivos al interactuar con APIs del DOM.

En conjunto, estas opciones alinearon el análisis de código con la realidad tecnológica del frontend institucional, reduciendo fricciones y advertencias innecesarias.

3. **Ecosistema de plugins:**

   El bloque plugins integró extensiones especializadas que ampliaron el alcance del análisis.

   - *react: pluginReact*: reglas específicas para componentes y convenciones de JSX.
   - *"react-hooks": pluginReactHooks*: refuerzo el uso correcto y la gestión de dependencias en hooks como useEffect o useMemo.
   - *prettier: pluginPrettier*: incorporó el formateador automático de código en el flujo de linting (análisis de código), asegurando uniformidad visual y evitando conflictos entre estilo y validación.

Combinando estos módulos con el paquete base @eslint/js, la configuración cubrió errores semánticos y aseguró consistencia sintáctica /estética, configurando un sistema integral de control de calidad.

4. **Base de reglas y herencia recomendada:**

   La política se apoyó en perfiles recomendados y se reforzó con ajustes propios.

   - pluginJs.configs.recommended.rules aplicó el núcleo de reglas sugeridas por ESLint para errores frecuentes en JavaScript.
   - pluginReact.configs.flat.recommended.rules extendió la cobertura hacia buenas prácticas de React, bajo el esquema Flat Config.

   De este modo, el proyecto adoptó un modelo híbrido que combinó lineamientos estándar reconocidos por la comunidad con políticas personalizadas ajustadas al contexto de desarrollo institucional.

5. **Políticas clave de validación (rules):**

   El bloque rules constituyó el núcleo del archivo y fijó criterios estrictos y preventivos, alineados con la filosofía Shift-Left. Entre los más relevantes destacaron:

   - *"prettier/prettier": "error"*, que elevó el formateo correcto a condición obligatoria para asegurar consistencia y claridad.
   - *"react-hooks/rules-of-hooks": "error"* y *"react-hooks/exhaustive-deps": "warn"*, garantizaron el uso adecuado de hooks y la gestión correcta de dependencias.
   - *"no-unused-vars": "error*", evitó variables sin uso y redujo deuda técnica.
   - *"no-console": "warn"* y *"no-debugger": "error"*, limitaron trazas de depuración en entornos productivos.
   - *eqeqeq: ["error", "always"]* y *curly: ["error", "all"]*, impusieron comparaciones estrictas y uso obligatorio de llaves, reforzando seguridad y legibilidad del código.

   De forma conjunta, estas políticas previnieron errores tempranos, uniformaron la escritura del código y fortalecieron la mantenibilidad.

6. **Compatibilidad con React ≥17:**

   La desactivación de *react/jsx-uses-react* y *react/react-in-jsx-scope* respondió a los cambios introducidos desde React 17, que eliminaron la necesidad de importar explícitamente la librería en cada archivo. Este ajuste evitó advertencias innecesarias y mantuvo el análisis alineado con la evolución del framework.

7. **Detección automática de la versión de React:**

   La instrucción *settings.react.version: "detect"* permitió que ESLint identificara automáticamente la versión instalada de React. Esta

sincronización previno conflictos y redujo ajustes manuales ante actualizaciones, asegurando que el análisis permaneciera acorde con la versión vigente del framework y, por tanto, con el comportamiento real de la aplicación.

Con este diseño, la Flat Config de ESLint operó como un contrato de calidad: definió el ámbito (archivos, lenguaje y navegador), especializó la verificación (plugins), partió de estándares reconocidos (perfiles recomendados) y aplicó políticas estrictas donde el impacto fue mayor (hooks, estilo, comparaciones, depuración y claridad estructural). Todo ello quedó alineado con el contexto React/JavaScript de la Cooperativa de Ahorro y Crédito Jardín Azuayo y con la estrategia de integración en VS Code y CI/CD, materializando un control temprano, automático y trazable.

En síntesis, la implementación de eslint.config.js representó un avance estratégico en el control de calidad del software institucional. La configuración consolidó un proceso de verificación temprana, automatizada y continua del código fuente, coherente con Shift-Left. Su estructura modular y flexible aseguró compatibilidad con React y JavaScript modernos y promovió una cultura de estandarización y mantenimiento sostenible. Gracias a ello, el equipo de desarrollo de software pudo detectar y corregir errores desde fases iniciales, mantener un estilo uniforme y garantizar que cada entrega cumpliera los umbrales de calidad definidos. En conjunto, la solución optimizó flujos de trabajo, redujo reproceso y contribuyó a la madurez técnica del proceso de desarrollo, consolidando una infraestructura de calidad robusta y escalable en la Cooperativa de Ahorro y Crédito Jardín Azuayo.

3.4 Implementación de Selenium modo localhost

En el marco del proyecto de automatización para el aseguramiento de la calidad del software en la Cooperativa de Ahorro y Crédito Jardín Azuayo, se habilitó la ejecución de pruebas funcionales con Selenium en entorno local (localhost). Esta modalidad permitió validar el comportamiento de las aplicaciones web

directamente sobre el ambiente de desarrollo, sin requerir servidores remotos ni infraestructura adicional de despliegue. Con ello, se aseguró una retroalimentación temprana y continua, en coherencia con el principio Shift-Left, que favorece la detección y corrección de defectos desde las primeras fases del ciclo de vida del software (Pressman & Maxim, 2021).

El uso de Selenium en modo local proporcionó un entorno controlado y reproducible, idóneo para la etapa de prototipado, en la que se verificó la estabilidad de los casos de prueba automatizados antes de incorporarlos al pipeline de Integración Continua (CI). Así mismo, esta estrategia facilitó el diagnóstico de errores, la validación de flujos de usuario y la depuración en tiempo real de los scripts, aspectos clave para respaldar la confiabilidad del conjunto de pruebas y elevar la calidad del producto software (Sommerville, 2016).

3.4.1 Configuraciones previas

Antes de ejecutar las pruebas automatizadas, se llevó a cabo una verificación integral del entorno técnico requerido para la operación de Selenium en modo local. El objetivo fue confirmar que cada componente del ecosistema de desarrollo estuviera instalado, configurado y actualizado, de modo que se garantizara la compatibilidad entre herramientas y la estabilidad del entorno de pruebas.

Verificación de la versión de Google Chrome: en primera instancia, se comprobó la versión exacta del navegador instalada en el equipo de desarrollo. Para ello, se accedió al menú principal del navegador, se ingresó en Configuración y, posteriormente, se consultó el apartado Acerca de Chrome, donde se visualizó la versión vigente. Este procedimiento permitió corroborar la compatibilidad entre el navegador y el ChromeDriver utilizado por Selenium, previniendo fallos por desalineación de versiones. Asimismo, se mantuvo el navegador actualizado a fin de asegurar la correcta ejecución de los scripts y la detección confiable de elementos dinámicos en la interfaz web.

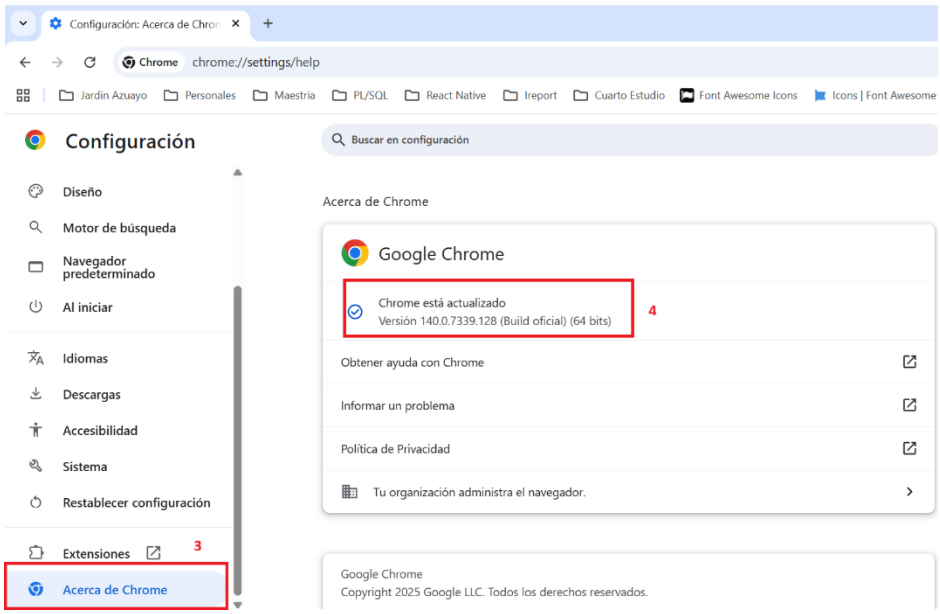


Figura. 4 Versión de Chrome

Verificación e instalación de Java Development Kit (JDK): e l JDK, imprescindible para la ejecución de Selenium, se validó mediante el comando *java -version* en la consola del sistema, con el fin de corroborar su presencia y la versión disponible. Para el proyecto se adoptó la versión 17 o superior, por asegurar compatibilidad con dependencias modernas de Selenium. En los equipos donde no se registró una instalación previa, se procedió a instalar la distribución Amazon Corretto, cuya estabilidad y soporte multiplataforma garantizaron un funcionamiento confiable del entorno.

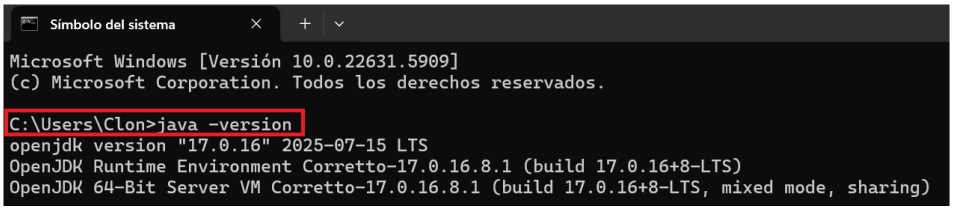


Figura. 5 Verificación versión JDK

Verificación e instalación de Apache Maven: Apache Maven funciona como gestor de dependencias y sistema de construcción para proyectos Java, Maven se verificó ejecutando el comando *mvn -v* en la consola de Windows, con lo cual se confirmó su presencia y la versión instalada. En los casos en que la herramienta no estuvo disponible, se realizó la descarga desde el sitio oficial; posteriormente, se

descomprimió el paquete y se ubicó en una ruta estándar (por ejemplo, C:\maven), para luego configurar la variable de entorno correspondiente y habilitar el uso global del comando mvn. Con este procedimiento, el entorno de desarrollo quedó preparado para compilar, ejecutar y gestionar de manera automatizada y estandarizada las dependencias requeridas por Selenium y por el proyecto en su conjunto.

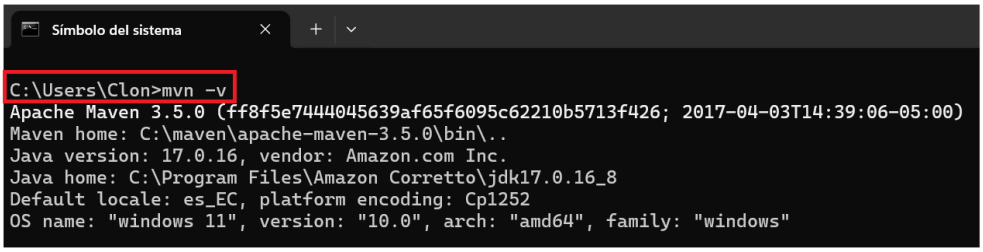


Figura. 6 Verificación versión Apache Maven

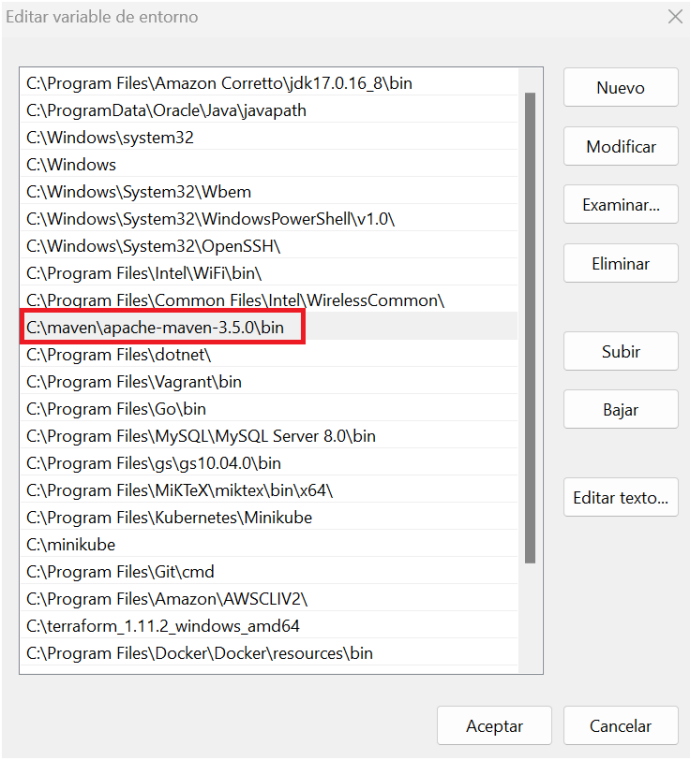


Figura. 7 Configuración variable entorno Apache Maven

Verificación e instalación de Node.js: Node.js opera como herramienta complementaria para la gestión de dependencias y la ejecución de scripts en el ecosistema web, Node.js se comprobó ejecutando el comando *node -v* en la consola, con lo cual se confirmó su presencia y la versión instalada. Cuando no se

dispuso de Node.js, se procedió a descargarlo desde el sitio oficial e instalarlo mediante su asistente, el cual configuró automáticamente las rutas en las variables de entorno (Path). Mantener Node.js actualizado habilitó la ejecución de utilidades asociadas al flujo de pruebas con Selenium y facilitó futuras automatizaciones orientadas a frontends basados en React o Angular.

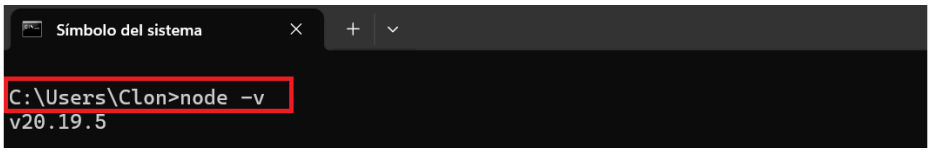


Figura. 8 Verificación versión Node.js

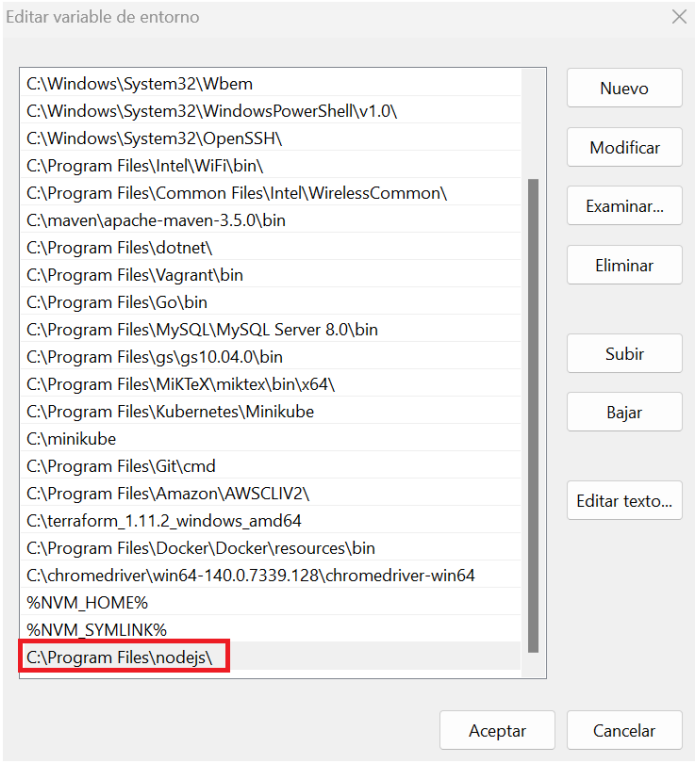


Figura. 9 Configuración variable entorno Node.js

Concluidas las validaciones e instalaciones de cada componente, el entorno de desarrollo quedó habilitado para la ejecución de pruebas automatizadas con Selenium. Este proceso previo aseguró la compatibilidad entre herramientas, minimizó fallos derivados de configuraciones incorrectas y estableció un marco de trabajo estable y reproducible para las etapas subsiguientes del proyecto. En particular, la correcta instalación de Java, Maven, Node.js y Google Chrome

constituyó el cimiento para una ejecución fluida y confiable de los casos de prueba. En consecuencia, la preparación del entorno no sólo se configuró como una tarea técnica inicial, sino como un factor determinante para la eficiencia y precisión del aseguramiento de la calidad en el desarrollo de software de la Cooperativa de Ahorro y Crédito Jardín Azuayo (Pressman & Maxim, 2021).

3.4.2 Creación de prototipo Seleniun Localhost

Verificado y configurado el entorno técnico para la automatización, se construyó un prototipo funcional de Selenium en modo localhost. El objetivo fue comprobar la factibilidad técnica del modelo de automatización en un contexto controlado, validando la interacción entre los componentes de software y la infraestructura local de desarrollo. Adicionalmente, el prototipo sirvió como base de referencia sobre la cual se proyectó la construcción de casos de prueba automatizados destinados a su posterior integración en el pipeline de Integración Continua (CI).

Para la implementación se empleó el lenguaje Java, dada su compatibilidad nativa con las librerías de Selenium WebDriver y su amplia adopción en ámbitos empresariales. Como entorno de programación se seleccionó IntelliJ IDEA (versión 2022.2.2), por ofrecer un IDE robusto con soporte nativo para Maven y capacidades avanzadas de asistencia al desarrollo, lo que facilitó la creación, ejecución y depuración de las pruebas automatizadas.

En la configuración inicial se generó un proyecto Maven, eligiendo arquitectura Java con el arquetipo Quickstart. Esta estructura aportó una organización modular del código, con directorios claramente definidos para clases principales y tests, y permitió centralizar la gestión de dependencias. Con ello, se garantizó compatibilidad entre librerías, se automatizó la compilación y se homogeneizó la ejecución en distintos entornos (Sonatype, 2008).

Cabe destacar que el arquetipo Quickstart proporcionó una plantilla mínima pero estandarizada, alineada con las convenciones de Maven, que facilitó la ampliación progresiva del proyecto sin comprometer su mantenibilidad. Esta base simplificó la incorporación de nuevas dependencias, la integración de librerías externas y la

ejecución automatizada de pruebas; aspectos clave para un desarrollo ágil y escalable (Apache Maven Project, 2023).

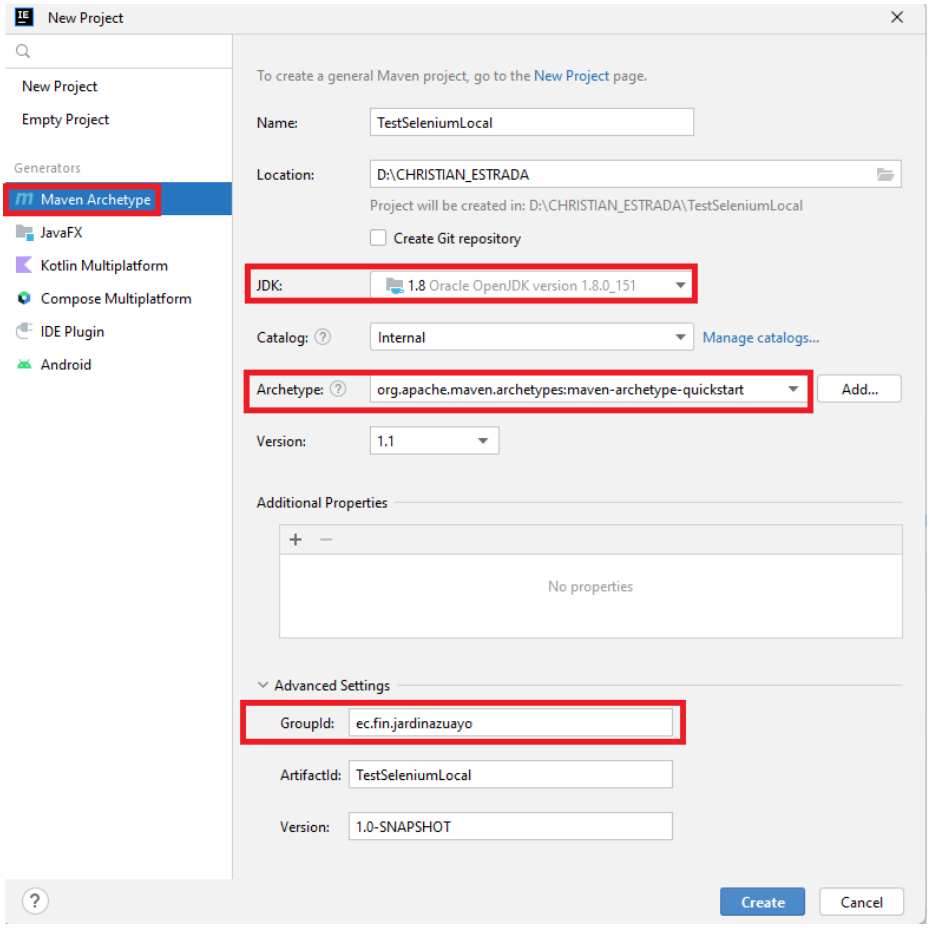


Figura. 10 Configuración proyecto prototipo

Una vez creado el proyecto, se incorporaron en el archivo pom.xml las dependencias esenciales para la ejecución del prototipo. Entre ellas destacaron Selenium WebDriver, JUnit y WebDriverManager, cada una con un rol específico dentro del flujo de automatización.

- Selenium WebDriver habilitó la ejecución de pruebas sobre navegadores reales, logrando validar la experiencia de usuario y verificar la funcionalidad completa de la aplicación web bajo condiciones controladas (Selenium Project, 2025).
- A su vez, JUnit proveyó una infraestructura sólida para estructurar y ejecutar pruebas unitarias e integradas, permitiendo comprobar el comportamiento de componentes antes de su integración definitiva (JUnit Project, 2025).

- Finalmente, WebDriverManager automatizó la descarga, configuración y actualización de controladores (por ejemplo, ChromeDriver o GeckoDriver), eliminando ajustes manuales y mejorando la portabilidad y el mantenimiento del sistema de pruebas (Bonigarcia, 2025).

La combinación de estas librerías permitió construir scripts robustos, reutilizables y portables, con un funcionamiento estable en entornos locales y con proyección a escenarios de ejecución remota. En conjunto, esta configuración sentó una base técnica consistente para un modelo de automatización eficiente y escalable, alineado con buenas prácticas de ingeniería de software y con las recomendaciones de madurez de procesos de desarrollo (CMMI Institute, 2018).

pom.xml (TestSeleniumLocal)

```
18
19  <dependencies>
20      <!-- Dependencia de Selenium 4.x compatible con Java 11 y Chrome 136 -->
21      <dependency>
22          <groupId>org.seleniumhq.selenium</groupId>
23          <artifactId>selenium-java</artifactId>
24          <version>4.25.0</version>
25      </dependency>
26
27      <!-- WebDriverManager más reciente -->
28      <dependency>
29          <groupId>io.github.bonigarcia</groupId>
30          <artifactId>webdrivermanager</artifactId>
31          <version>5.9.2</version>
32      </dependency>
33
34      <!-- JUnit 4 -->
35      <dependency>
36          <groupId>junit</groupId>
37          <artifactId>junit</artifactId>
38          <version>4.13.2</version>
39          <scope>test</scope>
40      </dependency>
```

Figura. 11 Referencias archivo pom.xml

La clase HistorialFinancieroTest.java se constituyó en el núcleo funcional del prototipo implementado con Selenium. Su propósito fue verificar el comportamiento del módulo Historial Financiero de la aplicación institucional mediante la ejecución automatizada de casos que replicaron las interacciones del usuario en el navegador. La clase se desarrolló en Java y se integró con las dependencias previamente declaradas en el pom.xml, lo que aseguró compatibilidad con las APIs de Selenium WebDriver y el framework de pruebas

JUnit. Con el fin de explicar de manera ordenada la lógica de configuración y los elementos que la componen, se presentó en primera instancia la configuración de HistorialFinancieroTest.java (véase Figura. 12) y, a continuación, se expuso el análisis técnico detallado de dicho archivo.

HistorialFinancieroTest.java

```java
package ec.fin.jardiazuayo;
import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.WebDriverWait;

Santiago-Parra88 *
public class HistorialFinancieroTest {
    Santiago-Parra88 *
    public static void main(String[] args) {
        WebDriver driver = null;

        try {
            WebDriverManager.chromedriver().setup();
            driver = new ChromeDriver();
            WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

            // Navegar a Google
            driver.get("https://                              ");

            // Imprimir información de la página
            System.out.println("Título de la página: " + driver.getTitle());
            System.out.println("URL actual: " + driver.getCurrentUrl());

            WebElement campo;

            // Login
            campo = driver.findElement(By.id("username"));
            campo.sendKeys( ...keysToSend: "usuario", Keys.TAB);
            campo = driver.findElement(By.id("password"));
            campo.sendKeys( ...keysToSend: "0", Keys.TAB);
            campo = driver.findElement(By.id("kc-login"));
            campo.click();
            Thread.sleep( millis: 4000);

            // Selección sucursal
            campo = driver.findElement(By.id("mainForm:j_idt86:combo_sucursal"));
            campo.click();
            campo = driver.findElement(By.id("mainForm:j_idt86:combo_sucursal_filter"));
            campo.sendKeys( ...keysToSend: "cuenca", Keys.ENTER);
            campo = driver.findElement(By.id("mainForm:j_idt91"));
            campo.click();
            Thread.sleep( millis: 1000);

            // Tiempo de espera para visualizar la pagina a realizar pruebas
            System.out.println("Esperando 10 segundos para que veas la página...");
            Thread.sleep( millis: 10000);

            System.out.println("¡Navegación completada exitosamente con Chrome for Testing!");

        } catch (Exception e) {
            System.err.println("Error durante la ejecución:");
            e.printStackTrace();
        } finally {
            if (driver != null) {
                System.out.println("Cerrando navegador...");
                driver.quit();
            }
            System.out.println("¡Aplicación terminada!");
        }
    }
}
```

Figura. 12 Configuración HistorialFinancieroTest.java

Al iniciar el análisis del código, la clase principal HistorialFinancieroTest.java comenzó con la importación de las dependencias necesarias, entre ellas:

- org.openqa.selenium.*
- org.openqa.selenium.chrome.ChromeDriver
- org.junit.*
- io.github.bonigarcia.wdm.WebDriverManager

Estas librerías proveyeron las capacidades esenciales para automatizar acciones en el navegador, gestionar el ciclo de vida de los controladores (drivers) y estructurar las pruebas unitarias bajo JUnit. Gracias a esta integración, la clase operó como un componente autónomo dentro del flujo de ejecución, manteniendo coherencia con la estructura modular del proyecto y las configuraciones declaradas en Maven.

En esta implementación se utilizó la librería WebDriverManager, que simplificó de forma significativa la gestión automática de controladores de navegadores al eliminar la necesidad de definir manualmente rutas de acceso o versiones específicas. Mediante la instrucción *WebDriverManager.chromedriver().setup()*, el sistema descargó y configuró automáticamente la versión de ChromeDriver compatible con el Google Chrome instalado en el entorno de desarrollo, evitando incompatibilidades. Esta automatización redujo la intervención manual, mejoró la portabilidad y aumentó la estabilidad del entorno de pruebas, permitiendo ejecutar casos de forma uniforme en distintos equipos sin configuraciones adicionales.

A continuación, con la sentencia *driver = new ChromeDriver();*, se creó una instancia del navegador en modo automatizado, estableciendo un canal de comunicación directa entre el script y el navegador real. Este objeto actuó como enlace principal entre el código y la interfaz del navegador, posibilitando operaciones como la apertura de páginas web, la interacción con elementos del DOM (Document Object Model), el envío de datos, la captura de errores y la validación de resultados visuales o textuales. Adicionalmente, la correcta instanciación del driver permitió controlar múltiples sesiones concurrentes, gestionar ventanas emergentes y ejecutar secuencias complejas que simularon con precisión el comportamiento del usuario final, garantizando un análisis funcional detallado del sistema bajo prueba.

Así mismo, se configuró el objeto *driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));*, con un tiempo máximo de espera de 10 segundos, destinado a sincronizar la ejecución de los comandos con los tiempos de carga de los elementos de la interfaz. Este mecanismo de espera implícita permitió que Selenium gestionara de forma automática las variaciones de latencia del sistema o la red, evitando errores de sincronización o fallos por elementos aún no disponibles. De este modo, el script aguardó hasta que cada elemento cumpliera condiciones como ser visible, estar habilitado o interactuable antes de continuar con la siguiente acción.

Finalmente, la instrucción *driver.get()* ordenó al navegador abrir la URL del sistema en desarrollo, correspondiente al portal institucional de aplicaciones de la Cooperativa de Ahorro y Crédito Jardín Azuayo. Este comando marcó el inicio formal del ciclo de pruebas automatizadas, permitiendo validar funcionalidades iniciales de acceso, autenticación y navegación general dentro del entorno web de la institución.

El bloque central del script ejecutó las acciones de prueba siguiendo el patrón Arrange-Act-Assert, que estructuró el flujo de verificación en tres fases bien definidas:

- Arrange (preparación): se definieron credenciales, selectores de elementos y valores esperados de validación.
- Act (ejecución): se automatizaron acciones del usuario, como ingreso de credenciales, interacción con botones, menús, campos de texto y navegación entre módulos.
- Assert (verificación): se compararon los resultados obtenidos con los valores esperados mediante aserciones (assertEquals, assertTrue), confirmando que el comportamiento requerido del sistema coincide con los requisitos funcionales.

Por ejemplo, el script localizó los campos de usuario y contraseña mediante *driver.findElement(By.id("username"))* y *driver.findElement(By.id("password"))*

(véase Figura. 12), respectivamente, simulando el ingreso de datos válidos para iniciar sesión. Luego, navegó hacia el módulo Historial Financiero, donde verificó la correcta visualización de registros mediante la búsqueda de elementos identificados por clases o etiquetas HTML. Esta metodología aseguró que los componentes del frontend respondieran adecuadamente ante entradas reales, reproduciendo con fidelidad la experiencia del usuario final y validando la interacción entre interfaz y lógica de negocio.

Al concluir la ejecución, se aplicaron aserciones de verificación para comprobar que los resultados mostrados en la interfaz coincidieran con los valores esperados definidos en los casos de prueba. Este proceso reforzó la confiabilidad del flujo de usuario, validó la consistencia de los datos y garantizó la integridad funcional del módulo evaluado.

Finalmente, la llamada del método *driver.quit()* (véase Figura. 12) cerró el navegador y liberó recursos del sistema, completando el ciclo de vida de la prueba automatizada. Este procedimiento aseguró que el entorno quedara limpio y disponible para ejecuciones posteriores, evitando conflictos entre sesiones y favoreciendo la reproducibilidad de resultados dentro del proceso continuo de aseguramiento de calidad.

En síntesis, la implementación del prototipo local con Selenium mediante la creación de la clase HistorialFinancieroTest.java evidenció la madurez del enfoque adoptado para el aseguramiento de la calidad en la Cooperativa de Ahorro y Crédito Jardín Azuayo. El desarrollo de la clase HistorialFinancieroTest.java no solo permitió validar la capacidad del framework para automatizar flujos funcionales críticos, sino que también demostró la integración efectiva entre Java, Maven y WebDriverManager en un entorno controlado. Este ejercicio consolidó un modelo reproducible de pruebas que garantizó consistencia, portabilidad y trazabilidad en las validaciones del sistema, sentando bases técnicas para extender la automatización hacia escenarios más complejos y procesos de integración continua, fortaleciendo así el ciclo de vida del software institucional.

Ahora bien, para lograr la correcta ejecución del prototipo y asegurar su funcionamiento estable en distintas estaciones de trabajo, resultó indispensable cumplir ciertos requisitos previos de configuración. Estos abarcaron la verificación de versiones de controladores, dependencias de entorno y parámetros de ejecución de Selenium, garantizando la sincronización entre navegador, drivers y librerías. En este sentido, la explicación subsiguiente detalló los ajustes necesarios para la ejecución local, incluyendo la descarga de ChromeDriver con el comando *npx @puppeteer/browsers install chromedriver@<<Version Chrome>>* y la configuración del entorno en el proyecto Maven, asegurando compatibilidad con el prototipo HistorialFinancieroTest.java y la infraestructura tecnológica de la Cooperativa de Ahorro y Crédito Jardín Azuayo.

El primer paso consistió en descargar la versión de ChromeDriver compatible con el navegador instalado. Para ello, se ejecutó en la terminal del proyecto el comando *npx @puppeteer/browsers install chromedriver@141.0.7390.66*. Una vez completada la descarga, se copió el ejecutable a una ruta estable (por ejemplo, C:\chromedriver) y se añadió dicha ruta a la variable de entorno PATH. Finalmente, se verificó la instalación ejecutando *chromedriver --version*, con lo que se confirmó que el sistema reconocía el controlador y se validó la versión instalada.

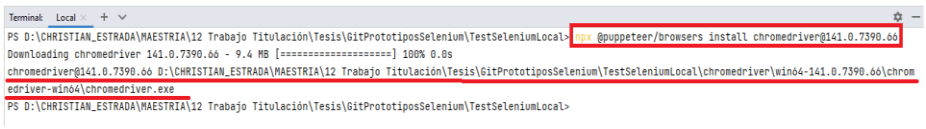


Figura. 13 Descarga y ubicación de Chromedriver

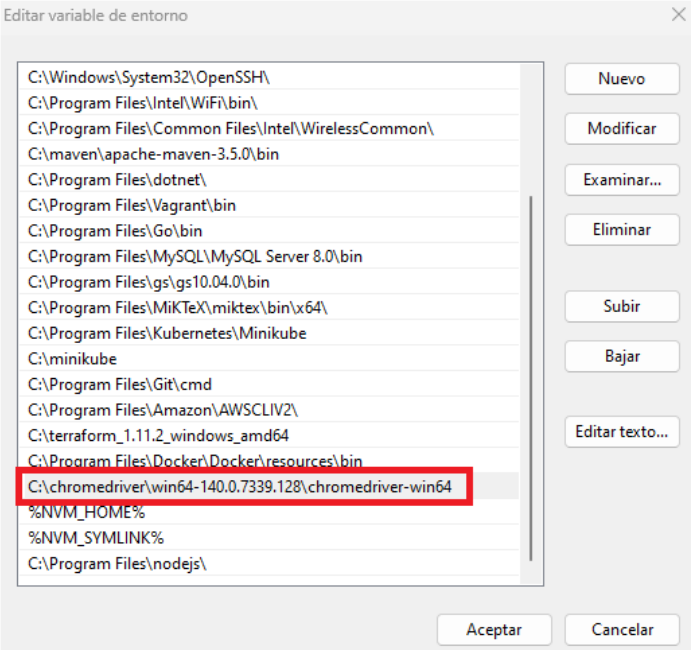


Figura. 14 Variable de entorno Chromedriver

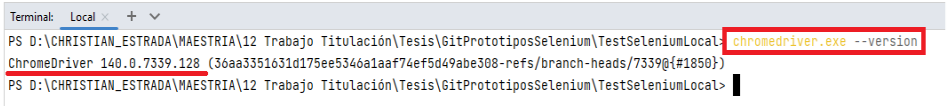


Figura. 15 Validación versión Chormedriver

A continuación, se definió el entorno de ejecución de Java (JDK) dentro del proyecto en IntelliJ IDEA. Para ello, se accedió a las propiedades del proyecto (clic derecho, Open Module Settings o presionando la tecla F4) y, en el panel de configuración, se ingresó a la sección SDKs para seleccionar la versión del JDK previamente instalada en el sistema. Esta asignación aseguró que el compilador y las librerías asociadas fueran reconocidas por el IDE, garantizando la correcta compilación y la ejecución de las pruebas automatizadas implementadas con Selenium.

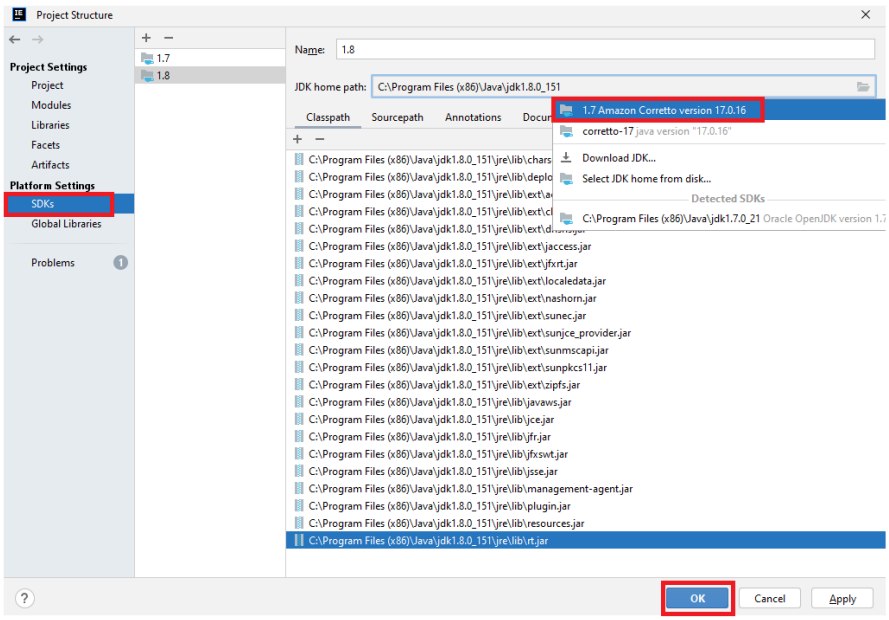


Figura. 16 Especificación JDK en IDE

Con la configuración completada, se ejecutó la clase HistorialFinancieroTest.java desde IntelliJ IDEA, mediante la opción Run *'HistorialFinancieroTest.main'* o el atajo Ctrl + Mayús + F10, iniciando el flujo automatizado de pruebas locales sobre el portal institucional.

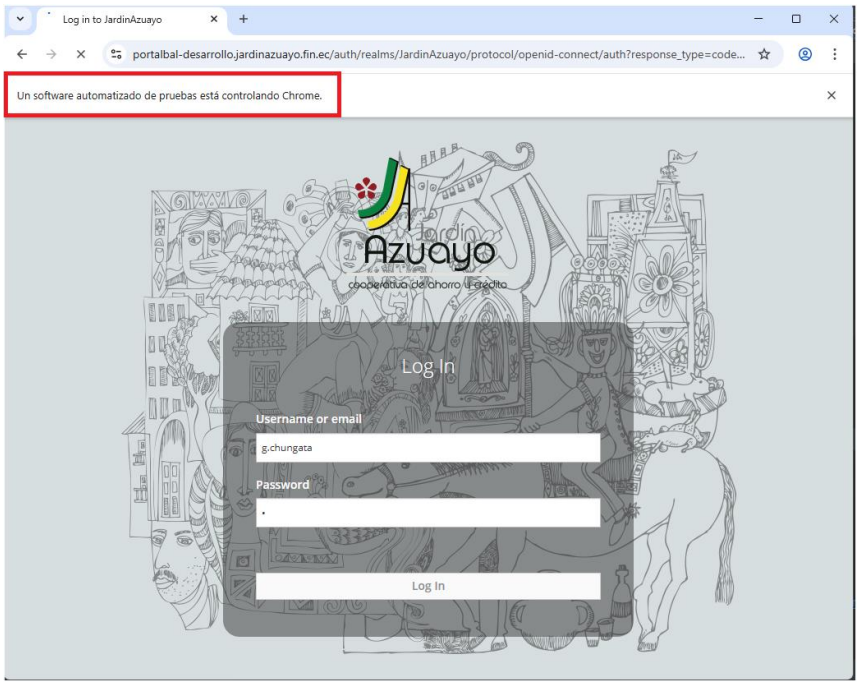


Figura. 17 Ejecución Selenium localhost

La verificación sistemática de componentes y dependencias redujo el riesgo de errores por desalineación de versiones y optimizó el tiempo de preparación del entorno, consolidando un modelo de ejecución estándar que sirvió de base para futuras integraciones en Docker y su incorporación en pipelines CI/CD de la Cooperativa de Ahorro y Crédito Jardín Azuayo.

Ejecutar el prototipo en localhost permitió validar la funcionalidad del modelo de automatización en condiciones controladas, asegurando la estabilidad del flujo de pruebas y la comunicación correcta entre Selenium y el navegador Chrome. Además, la estructura modular de la clase HistorialFinancieroTest.java, sustentada en JUnit y Maven, facilitó una ejecución ordenada, la generación automática de evidencias y la trazabilidad de resultados. De igual manera, la incorporación de WebDriverManager mejoró la portabilidad del sistema al eliminar configuraciones manuales y habilitar su ejecución en diferentes entornos de trabajo. En conjunto, esta fase afianzó la base técnica del modelo y evidenció la viabilidad del enfoque de automatización dentro del ciclo institucional de aseguramiento de la calidad.

3.5 Implementación de Selenium con Docker

La integración de Selenium con Docker constituyó un avance sustantivo en la automatización de pruebas, al habilitar la ejecución en entornos aislados, reproducibles y consistentes sin depender de configuraciones locales específicas. Esta estrategia respondió a la necesidad institucional de la Cooperativa de Ahorro y Crédito Jardín Azuayo de uniformar las validaciones funcionales y estabilizar los entornos de prueba, independientemente de la infraestructura física o del sistema operativo utilizado. En este marco, la contenedorización permitió controlar con precisión el ciclo de vida de las pruebas automatizadas, mejorando la escalabilidad, facilitando el despliegue y fortaleciendo la trazabilidad de los resultados (Docker, 2025).

Como punto de partida, se requirió disponer de Docker Desktop instalado en el entorno de desarrollo. Esta herramienta proporcionó la interfaz y el motor necesarios para construir imágenes, gestionar contenedores y orquestar servicios.

En ausencia de una instalación previa, se obtuvo desde el sitio oficial de Docker en la versión correspondiente al sistema operativo. Con la plataforma operativa, se procedió a construir imágenes personalizadas, ejecutar servicios en paralelo y desplegar aplicaciones, asegurando un entorno controlado e idéntico al que se buscó replicar en fases posteriores, incluida la integración con pipelines de CI/CD.

3.5.1 Configuración de proyecto

El proyecto destinado a la ejecución de pruebas en contenedores se creó replicando la estructura definida para el entorno local descrita previamente en Creación de prototipo Seleniun Localhost, asignándole un identificador independiente: TestSeleniumDocker. La arquitectura volvió a sustentarse en Apache Maven, empleando el arquetipo quickstart con el fin de preservar la estandarización del código, la organización modular de directorios y la trazabilidad de dependencias. Esta decisión permitió reutilizar con consistencia los componentes ya verificados en la fase anterior, concretamente Selenium WebDriver, WebDriverManager y JUnit, manteniendo continuidad técnica y reduciendo el riesgo de incompatibilidades.

Maven siguió ocupando un rol central, pues automatizó la resolución de librerías y facilitó la alineación del proyecto con los procesos de integración continua. A través del archivo pom.xml se declararon las bibliotecas requeridas, se fijaron versiones compatibles con la ejecución en Docker y se definieron rutas de compilación y empaquetado. Este planteamiento modular mejoró la mantenibilidad y disminuyó la probabilidad de conflictos entre entornos, aspecto especialmente crítico al trabajar con contenedores aislados.

Con el propósito de habilitar la ejecución del navegador dentro del contenedor, la clase HistorialFinancieroTest se ajustó mediante parámetros y directrices específicas. En particular, se inicializó el objeto *ChromeOptions options = new ChromeOptions();*, sobre el cual se incorporaron argumentos orientados a la ejecución en ambientes sin interfaz gráfica y con recursos acotados. Dichos argumentos presentadas a continuación en el documento buscaron garantizar

estabilidad, compatibilidad y disponibilidad del navegador durante la automatización.

HistorialFinancieroTest.java

```java
package ec.fin.jardinazuayo;

import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.remote.RemoteWebDriver;

import java.net.URL;
import java.time.Duration;
1 usage  Santiago-Parra88 *
public class HistorialFinancieroTest {
    Santiago-Parra88 *
    public static void main(String[] args) {
        WebDriver driver = null;

        try {
            // Configurar opciones de Chrome para Chrome for Testing
            ChromeOptions options = new ChromeOptions();

            // Configuraciones optimizadas para Chrome for Testing
            options.addArguments("--no-sandbox");
            options.addArguments("--disable-dev-shm-usage");
            options.addArguments("--disable-gpu");
            options.addArguments("--remote-allow-origins=*");
            options.addArguments("--disable-web-security");
            options.addArguments("--disable-features=VizDisplayCompositor");

            // URL del ChromeDriver local
            String chromeDriverUrl = "http://host.docker.internal:9515";

            System.out.println("Conectando a ChromeDriver en: " + chromeDriverUrl);
            System.out.println("Usando Chrome for Testing con ChromeDriver compatible");

            // Crear RemoteWebDriver con timeout adecuado
            driver = new RemoteWebDriver(new URL(chromeDriverUrl), options);

            // Configurar timeouts para Selenium 4.x
            driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
            driver.manage().timeouts().pageLoadTimeout(Duration.ofSeconds(60));

            System.out.println("¡Conexión establecida con éxito!");

            // Navegar a app
            driver.get("https                                    );
```

Figura. 18 Configuración HistorialFinancieroTest con Docker

En la configuración de ChromeOptions se habilitaron parámetros específicos (véase Figura. 18) con el propósito de asegurar estabilidad y compatibilidad durante la ejecución dentro de contenedores Docker, donde no existe un entorno gráfico nativo y los recursos son más limitados que en una estación de trabajo convencional. Entre los más relevantes destacaron:

- ***--no-sandbox***: desactivó el sandbox de Chrome, dado que el aislamiento ya lo proporcionaba Docker a nivel de procesos y namespaces. Esta práctica evitó conflictos con restricciones del kernel. Se dejó constancia de que su uso no es recomendable en entornos multi-tenant expuestos; aquí se justificó por tratarse de un ambiente de pruebas controlado.
- ***--disable-dev-shm-usage***: mitigó errores por tamaño reducido de /dev/shm en contenedores, redirigiendo buffers de Chrome al sistema de archivos. Con ello se redujeron fallos intermitentes y se estabilizaron ejecuciones extensas. Como alternativa, se pudo ampliar /dev/shm al crear el contenedor; no obstante, esta solución resultó la medida más directa.
- ***--disable-gpu***: deshabilitó la aceleración por GPU para evitar dependencias de drivers gráficos no disponibles en el contenedor. En modo headless o sin servidor X, esta desactivación incrementó la portabilidad del navegador.
- ***--remote-allow-origins=****: permitió conexiones remotas desde distintos orígenes al endpoint de Chrome, útil cuando el WebDriver se ejecutó fuera del contenedor y se requería una sesión distribuida. En escenarios más estrictos, se contempló reemplazar el comodín por orígenes específicos.
- ***--disable-web-security***: suprimió ciertas políticas del navegador que bloquean pruebas entre dominios o mezclas de recursos locales/remotos, necesarias para validar flujos multi-origen.
- ***--disable-features=VizDisplayCompositor***: inhibió componentes de composición gráfica dependientes de capacidades de renderizados ausentes en Docker, reduciendo costos de render y evitando fallos de inicio en modo sin interfaz.

La clase HistorialFinancieroTest (véase Figura. 18) estableció la URL del servicio ChromeDriver mediante la asignación: *String chromeDriverUrl = "http://host.docker.internal:9515";*. Con este valor, Selenium se conectó al ChromeDriver en el host, manteniendo un canal estable entre el contenedor y el navegador. En entornos con Docker Desktop, el nombre *host.docker.internal* resolvió automáticamente hacia el equipo anfitrión, simplificando la comunicación sin necesidad de direcciones IP explícitas.

La inicialización *driver = new RemoteWebDriver(new URL(chromeDriverUrl), options);* (creación de sesión remota bajo el protocolo W3C WebDriver, véase Figura. 18) fusionó las capacidades definidas en ChromeOptions con el endpoint remoto. Para asegurar una sesión confiable y repetible, se mantuvo estricta compatibilidad de versiones entre Chrome, ChromeDriver y Selenium, evitando errores por desalineación.

Desde la perspectiva operativa, esta arquitectura separó responsabilidades; el contenedor ejecutó la lógica de pruebas, mientras el host proveyó el driver/navegador. Ello facilitó diagnóstico, escalabilidad y trazabilidad. Como refuerzo, se definió un tamaño de ventana y el modo sin interfaz gráfica, empleando: *options.addArguments("--headless=new"); y options.addArguments("--window-size=1920,1080");* (véase Figura. 18).

En conjunto, la combinación de opciones de ejecución y conexión remota permitió correr las pruebas en entornos Dockerizados de forma estable, portable y reproducible, manteniendo control fino sobre las capacidades del navegador e incorporando criterios de escalado cuando fue necesario.

El siguiente archivo que se documentó fue el Dockerfile, componente central para construir la imagen del contenedor. Dicho archivo describió, paso a paso, el proceso de ensamblaje del entorno: definió la imagen base, declaró dependencias, ejecutó comandos de compilación y estableció la forma de ejecución de la aplicación (Docker, 2025). A continuación, se detalló el esquema base implementado.

Dockerfile

```
# Usar imagen base con Java 11 y Maven
FROM maven:3.6.3-jdk-11

# Información del mantenedor
LABEL maintainer="cofeps"

# Crear directorio de trabajo
WORKDIR /app

# Copiar archivos de configuración de Maven
COPY pom.xml .

# Descargar dependencias
RUN mvn dependency:go-offline -B

# Copiar el código fuente
COPY src ./src

# Compilar la aplicación
RUN mvn compile -B

# Crear script de inicio que se conecta al ChromeDriver local
RUN echo '#!/bin/bash\n\
echo "Conectando a ChromeDriver en host.docker.internal:9515..."\n\
echo "Si falla, asegúrate de que ChromeDriver esté ejecutándose con: ./start-chromedriver-local.sh"\n\
mvn exec:java -Dexec.mainClass="ec.fin.jardinazuayo.HistorialFinancieroTest"\n\
' > /app/start.sh && chmod +x /app/start.sh

# Comando por defecto
CMD ["/app/start.sh"]
```

Figura. 19 Configuración Dockerfile

Cada instrucción del archivo cumplió una función específica que, en conjunto, garantizó la portabilidad, reproducibilidad y estabilidad del entorno de pruebas con Selenium. A continuación, se describieron su configuración y propósito técnico:

- ***FROM maven:3.6.3-jdk-11***: la directiva FROM estableció la imagen base desde la cual se construyó el contenedor. En este caso, se seleccionó la imagen oficial de Maven 3.6.3 con JDK 11, que incorporó el compilador de Java y las utilidades necesarias para compilar, gestionar dependencias y ejecutar proyectos bajo Maven. Esta elección aseguró la alineación entre el ciclo de vida del proyecto y el entorno de construcción, reproduciendo el mismo comportamiento en cualquier host donde se ejecutó la imagen.
- ***LABEL maintainer="cofeps"***: la etiqueta LABEL añadió metadatos al contenedor, identificando al responsable del mantenimiento. Este tipo de anotación fortaleció la trazabilidad y la gobernanza del software, especialmente útil en contextos institucionales y colaborativos.
- ***WORKDIR /app***: la instrucción WORKDIR fijó /app como directorio de trabajo dentro del contenedor, de modo que las órdenes subsiguientes (COPY, RUN, CMD) se ejecutaron en esa ruta. Esta decisión ordenó la

estructura interna, evitó rutas absolutas dependientes del sistema anfitrión y favoreció la mantenibilidad.

- ***COPY pom.xml y RUN mvn dependency:go-offline -B***: primero se copió el pom.xml al contenedor; posteriormente, se ejecutó *mvn dependency:go-offline -B* para descargar y cachear las dependencias. Con ello se optimizaron compilaciones futuras al aprovechar la capa de caché de Docker, evitando descargas repetitivas y asegurando que la imagen resultante dispusiera de todas las bibliotecas sin depender de conectividad externa durante la construcción.
- ***COPY src ./src***: esta instrucción transfirió el código fuente desde el host al contenedor, ubicándolo en /app/src. Con ello, Maven dispuso del material necesario para la fase de compilación, manteniendo todo el proceso dentro de un entorno aislado y sin interferencias del sistema operativo del anfitrión.
- ***RUN mvn compile -B***: se ejecutó la compilación en modo batch, generando los binarios dentro del contenedor sin intervención manual. Al compilar en este punto del build, el artefacto producido resultó coherente y reproducible, minimizando variaciones entre estaciones de trabajo.
- ***RUN echo '#!/bin/bash\n...***: mediante este bloque se generó dinámicamente el script de inicio start.sh, que centralizó la lógica de arranque del contenedor. El script incluyó: la verificación y conexión con ChromeDriver en *host.docker.internal:9515*, mensajes de diagnóstico en caso de error, y ejecución de la clase principal HistorialFinancieroTest a través del plugin *exec:java* de Maven. Con *chmod +x /app/start.sh* se otorgaron permisos de ejecución. El uso del plugin Maven Exec (MojoHaus, s.f.) permitió ejecutar la clase Java con el classpath del proyecto sin requerir generar previamente un JAR.
- ***CMD ["/app/start.sh"]***: finalmente, CMD definió el comando por defecto al iniciar el contenedor. Al invocar start.sh, cada instancia ejecutó automáticamente la lógica de conexión con ChromeDriver y las pruebas automatizadas, convirtiendo la imagen en una unidad autosuficiente capaz de iniciar, ejecutar y cerrar el ciclo de pruebas de forma controlada.

En conjunto, el Dockerfile no solo definió el entorno técnico para la ejecución de Selenium, sino que estableció un modelo reproducible, documentado y estandarizado para la automatización de pruebas dentro de la Cooperativa de Ahorro y Crédito Jardín Azuayo. Su estructura modular y declarativa facilitó la trazabilidad del proceso, incrementó la transparencia técnica y permitió que futuras versiones del entorno se generaran bajo los mismos parámetros, fortaleciendo la confiabilidad y la continuidad del aseguramiento de la calidad institucional.

Tras haber fijado el entorno base mediante el Dockerfile, el paso siguiente consistió en configurar el archivo *docker-compose.yml*, encargado de orquestar de manera simultánea y coordinada los servicios que conformaron el entorno de pruebas automatizadas. Docker Compose, herramienta oficial del ecosistema Docker, ofreció un mecanismo declarativo (formato YAML) para describir servicios, redes, volúmenes y dependencias, permitiendo que componentes como el navegador Chrome, el servicio Selenium y la aplicación de pruebas se ejecutaran de forma conjunta, coherente y reproducible (Docker Docs, s.f.).

Su principal fortaleza residió en orquestar localmente entornos complejos sin requerir la ejecución manual de múltiples comandos *docker run* con parámetros extensos. Al concentrar la configuración en un único archivo, Docker Compose elevó la consistencia operativa y la portabilidad del entorno, posibilitando replicar la misma arquitectura en equipos de desarrollo, integración o producción sin cambios sustanciales. Así mismo, simplificó la gestión del ciclo de vida de los contenedores, pues permitió iniciar, detener, reiniciar o reconstruir los servicios con un solo comando *docker-compose up*, reduciendo la complejidad operacional y el riesgo de errores manuales.

Entre los beneficios más significativos se destacó su simplicidad de uso, la coherencia entre entornos y la integración fluida con pipelines CI/CD. Además, permitió incluir procesos de construcción directamente en la orquestación mediante la clave build, que definió cómo generar cada imagen a partir de su contexto y del Dockerfile correspondiente (Docker, 2025). En consecuencia, la configuración del archivo docker-compose.yml se consolidó como un componente

medular del proyecto, al reunir en un único punto las definiciones necesarias para la ejecución coordinada de Selenium, ChromeDriver y las pruebas automatizadas. A partir de ello, la sección siguiente presentó la codificación y el análisis detallado de su estructura, junto con los parámetros implementados para la Cooperativa de Ahorro y Crédito Jardín Azuayo.

docker-compose.yml

```yaml
version: '3.8'
services:
  selenium-app:
    build: .
    container_name: selenium-HistorialFin
    environment:
      - JAVA_OPTS=-Xmx512m
      - SE_ALLOW_HOSTS=*
    extra_hosts:
      - "host.docker.internal:host-gateway"
    volumes:
      - ./logs:/app/logs
networks:
  selenium-network:
    driver: bridge
```

Figura. 20 Configuración docker-compose.yml

Quedó establecido el uso de la versión 3.8 de Compose como especificación del archivo, lo que aseguró compatibilidad con ediciones recientes de Docker Engine y habilitó capacidades avanzadas (redes personalizadas, dependencias entre servicios e integración con CI/CD). Esta selección favoreció la estabilidad del entorno y la posibilidad de migrar la configuración a plataformas más nuevas sin cambios sustantivos.

En segundo término, se describió la definición de servicios, núcleo del documento, organizada bajo el bloque *services:*. Para este caso, se declaró un único servicio, *selenium-app*, responsable de ejecutar la aplicación de pruebas automatizadas desarrollada en Java y gestionada con Maven.

Dentro de este bloque, la directiva *build:* indicó que la imagen debía construirse a partir del Dockerfile ubicado en el mismo directorio, garantizando coherencia entre ambas configuraciones y reutilizando dependencias, rutas y scripts ya definidos.

El parámetro *container_name: selenium-HistorialFin* asignó un identificador legible al contenedor, lo que simplificó el monitoreo y la depuración durante la ejecución especialmente útil cuando se manejan múltiples instancias o pruebas concurrentes, ya que permite rastrear los registros de ejecución y aislar posibles errores.

A continuación, el apartado *environment:* incorporó variables críticas para el funcionamiento:

- **JAVA_OPTS=-Xmx512m**: limitó la memoria máxima de la JVM, controlando el consumo de recursos en entornos con capacidad limitada.
- ***SE_ALLOW_HOSTS=****: permitió conexiones desde cualquier origen, requisito frecuente cuando la automatización se ejecuta de forma distribuida entre host y contenedores (Docker, 2025).

Seguidamente, la directiva *extra_hosts:* estableció el mapeo *"host.docker.internal:host-gateway"*, habilitando un canal de comunicación explícito entre el contenedor y el sistema anfitrión. Con ello, el servicio pudo acceder a componentes residentes en el host (por ejemplo, ChromeDriver o bases locales) manteniendo un entorno controlado pero flexible.

En la misma definición de servicio, *volumes:* montó *./logs* del proyecto en /app/logs dentro del contenedor. Este enlace aseguró persistencia de registros y acceso a evidencias de ejecución independientemente del ciclo de vida del contenedor, facilitando auditorías, comparación de resultados y trazabilidad de errores.

Finalmente, el bloque *networks:* creó la red interna *selenium-network* con el driver bridge, uno de los más empleados para aislar el tráfico entre contenedores de un mismo proyecto. Esta red actuó como capa de segregación y resolución de nombres entre servicios, evitando interferencias con otras aplicaciones externas y simplificando la topología del entorno (Merkel, D, 2014).

En conjunto, el archivo docker-compose.yml definió un entorno modular, reproducible y portable, adecuado a las necesidades de la Cooperativa de Ahorro y Crédito Jardín Azuayo. La configuración unificó dependencias de construcción, parámetros de entorno, volúmenes de almacenamiento y redes de comunicación, permitiendo desplegar todo el ecosistema de pruebas con un solo comando. Este enfoque redujo tiempos de preparación y validación, y consolidó un modelo de automatización alineado con DevOps y con el aseguramiento de calidad continuo (Red Hat, 2023) (Docker, 2025).

Con la configuración de Dockerfile y docker-compose.yml concluida, el entorno quedó listo para la ejecución de pruebas en contenedores. Ambos componentes operaron de forma complementaria: el Dockerfile fijó la imagen base, dependencias y proceso de construcción; y Compose coordinó la orquestación y sincronización de los servicios, esta integración reflejó un diseño modular y reproducible, en el que cada servicio cumplió un rol específico dentro del ecosistema de pruebas. El siguiente paso consistió en levantar el entorno y verificar su comportamiento, confirmando las configuraciones aplicadas para la comunicación entre ChromeDriver, el contenedor de Selenium y la clase HistorialFinancieroTest, eje del proceso de validación automatizada en la Cooperativa de Ahorro y Crédito Jardín Azuayo.

La primera acción fue habilitar el servicio ChromeDriver en el sistema anfitrión, enlace que traduce las instrucciones de Selenium en acciones concretas dentro del navegador (apertura de páginas, interacción con el DOM, validaciones visuales). Para iniciarlo, se ejecutó en la terminal del proyecto el comando: *chromedriver --port=9515 --allowed-origins=* --allowed-ips=*. Con ello, el servidor ChromeDriver quedó activo en el puerto 9515, preparado para recibir conexiones desde el contenedor y permitir el control remoto de Google Chrome durante la ejecución de la suite de pruebas.

Se configuraron los parámetros --allowed-origins=* y --allowed-ips= para ampliar la accesibilidad de conexión del servicio, habilitando que ChromeDriver aceptara solicitudes desde cualquier origen y dirección IP. Esta definición resultó clave en un

escenario Dockerizado, donde los contenedores operaron en espacios de red aislados y requirieron permisos explícitos para comunicarse con el host. Cabe señalar que, aunque el comodín facilita la interoperabilidad durante pruebas locales, en entornos más estrictos se recomienda restringir orígenes e IPs a rangos específicos para fortalecer la seguridad.

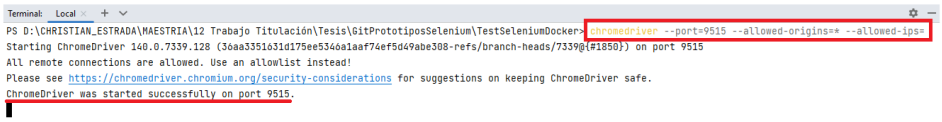


Figura. 21 Ejecución de chormedriver

Así mismo, iniciar ChromeDriver antes de levantar los contenedores garantizó que el servicio estuviera plenamente disponible cuando la aplicación de pruebas se ejecutara. Tras su arranque, la consola presentó el mensaje de arranque y modo escucha del servidor, señal inequívoca de que el sistema quedaba listo para recibir las instrucciones de automatización emitidas por Selenium. Con esta secuencia se evitó la intermitencia en la conexión, y los flujos definidos se ejecutaron de forma continua y sincronizada.

Acto seguido, se construyó y puso en marcha el contenedor de pruebas en una nueva terminal ubicada en la raíz del proyecto, mediante el comando: *docker compose up –build*.

Figura. 22 Ejecución comando docker-compose up --build

El uso de la opción --*build* garantizó que la imagen se reconstruyera incorporando los cambios más recientes tanto del proyecto como del Dockerfile. Durante la ejecución del comando, Docker Compose resolvió dependencias, creó la red interna definida (*selenium-network*), montó los volúmenes compartidos y puso en marcha el servicio principal *selenium-app* (véase Figura. 20).

En la consola se registraron los eventos del proceso (Figura. 22), evidenciando la creación exitosa de la imagen y el arranque del contenedor responsable de ejecutar las pruebas.

Con el entorno en ejecución, el contenedor lanzó automáticamente el script *start.sh* (Figura. 20), el cual estableció la conexión con ChromeDriver y ejecutó la clase principal HistorialFinancieroTest.java. A partir de ese punto, se inició el flujo automatizado de validaciones sobre la interfaz del portal institucional de la Cooperativa de Ahorro y Crédito Jardín Azuayo, verificando funcionalidades como autenticación, acceso a módulos y visualización de registros financieros.

Durante todo el proceso, Selenium interactuó con un navegador real de forma remota, simulando acciones de un usuario final y confirmando que la interfaz y los flujos de navegación respondieran correctamente a los escenarios previstos.

Figura. 23 Resultados ejecución docker-compose up --build

Con el entorno ya desplegado y los servicios en ejecución, se verificó el estado de los contenedores y de las imágenes desde la interfaz de Docker Desktop. En dicha consola, cada contenedor apareció identificado por su nombre, mostró su estado (running/exited), el tiempo de actividad y el consumo de recursos, lo que permitió confirmar que los procesos de construcción y puesta en marcha definidos en el docker-compose.yml se completaron correctamente. Del mismo modo, en la pestaña Images se visualizaron las imágenes generadas a partir del Dockerfile, junto con su tamaño y etiquetas, facilitando la gestión, la trazabilidad y la reutilización en despliegues posteriores.

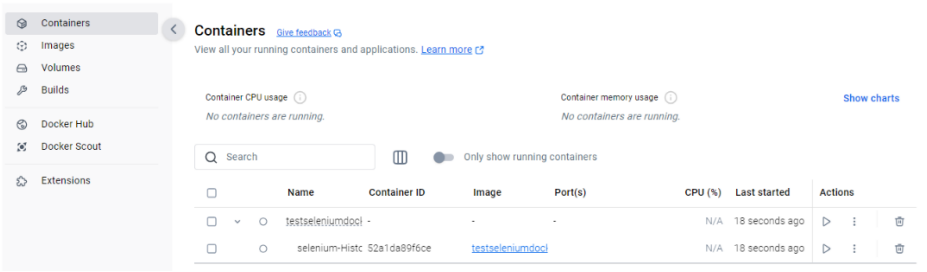


Figura. 24 Contenedor en Docker Desktop

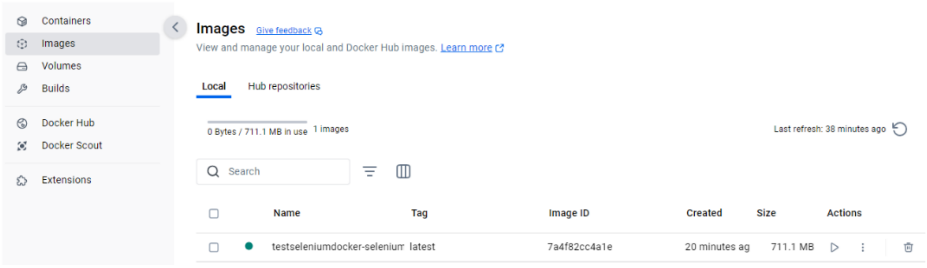


Figura. 25 Images en Docker Desktop

Asimismo, Docker Desktop brindó acceso gráfico a los logs de cada contenedor. Al seleccionar el contenedor *selenium-HistorialFin*, la herramienta desplegó un panel con las salidas de la aplicación en tiempo real, equivalentes a las trazas observadas por consola. Esta vista resultó especialmente útil para el seguimiento de las pruebas automatizadas, pues permitió identificar eventos, advertencias y errores sin ejecutar comandos adicionales. Además, los registros permanecieron disponibles mientras el contenedor se mantuvo activo, lo que facilitó el análisis posterior y la depuración del comportamiento del sistema en cada ejecución.

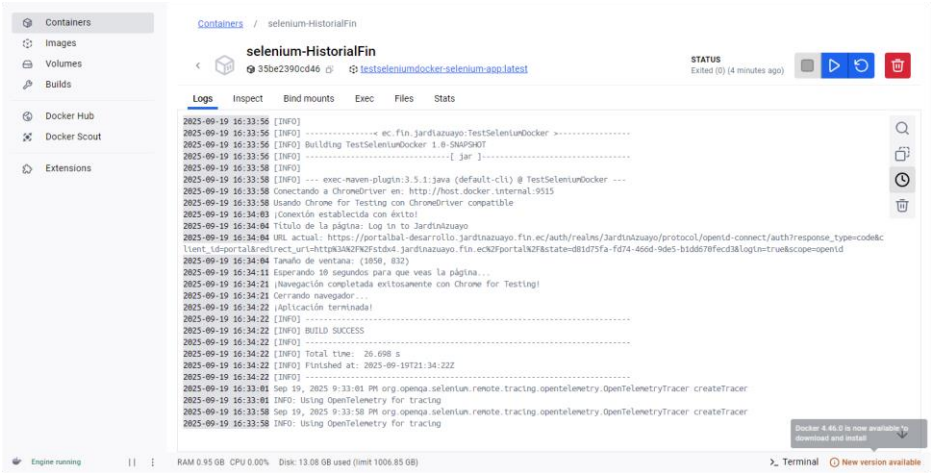


Figura. 26 Logs contenedor selenium-HistorialFin

La ejecución de Selenium en un entorno dockerizado demostró la eficiencia, escalabilidad y consistencia del modelo de automatización adoptado. Este enfoque habilitó la creación de entornos completamente aislados, reproducibles y portables, eliminando la dependencia de configuraciones locales y evitando errores por incompatibilidades de versiones. Así mismo, la contenedorización permitió replicar con precisión las condiciones de prueba, mejorando la trazabilidad de resultados y la estabilidad de los procesos de validación del software.

Desde el plano institucional, la solución fortaleció las prácticas de aseguramiento de calidad en la Cooperativa de Ahorro y Crédito Jardín Azuayo, al consolidar un entorno de pruebas coherente y escalable, alineado con los principios DevOps. Su adopción evidenció el compromiso de la institución con la modernización

tecnológica y con metodologías que integran desarrollo, despliegue y control de calidad de forma continua.

El avance alcanzado en esta fase sentó la base técnica para la automatización continua dentro del flujo de desarrollo. Verificada la estabilidad de Selenium en contenedores Docker, el paso siguiente consistió en integrar este entorno a un pipeline de GitLab, de modo que las pruebas se dispararan automáticamente ante cada actualización del código fuente. Esta evolución proveyó a los equipos de desarrollo y QA un sistema completamente automatizado, capaz de detectar fallos tempranos, generar reportes en tiempo real y preservar la coherencia del producto en cada entrega.

En consecuencia, el capítulo siguiente abordó la configuración e implementación del pipeline de integración continua (CI), consolidando la continuidad del modelo de automatización propuesto y su alineación con la estrategia institucional de desarrollo ágil y aseguramiento de calidad.

3.6 Integración de Selenium en Pipeline (CI) con GitLab

La incorporación de Selenium a un entorno de integración continua (CI) sobre GitLab representó un hito para la automatización integral del proceso de pruebas y del aseguramiento de la calidad. GitLab, al reunir en una sola plataforma el control de versiones, la gestión de incidencias, las revisiones de código y los flujos automatizados de construcción y despliegue, permitió orquestar pipelines coherentes y auditables de extremo a extremo (GitLab Documentation, 2025).

Gracias a esta arquitectura unificada, se definieron y ejecutaron pipelines personalizados que facilitaron la coordinación entre desarrollo y QA y, a la vez, mantuvieron la trazabilidad de cada cambio a lo largo del ciclo de vida del software. Entre los beneficios observados destacaron:

- Automatización nativa de tareas recurrentes.

- Colaboración centralizada en un único ecosistema.
- Escalabilidad para proyectos de distinta complejidad; y
- Enfoque robusto de seguridad y trazabilidad mediante controles de acceso, auditorías y registros detallados.

Adicionalmente, al concentrar funciones que usualmente exigen múltiples herramientas, se redujeron costos operativos y esfuerzos de integración.

La elección de GitLab respondió también a criterios institucionales: la Cooperativa de Ahorro y Crédito Jardín Azuayo ya lo utilizaba como sistema oficial de versionamiento, lo que disminuyó la curva de adopción y aseguró coherencia tecnológica. En ese marco, integrar Selenium dentro de GitLab consolidó un flujo de trabajo automatizado, ágil y sostenible, alineado con las prácticas de DevOps y con las políticas internas de calidad. A partir de esta base, se procedió a definir la estructura del pipeline y su archivo de configuración .gitlab-ci.yml, así como el uso de GitLab Runner para la ejecución controlada de las pruebas automatizadas.

3.6.1 Proceso de integración en GitLab

El proceso de integración comenzó con la creación de un proyecto privado en GitLab, denominado *Integración-Selenium-CI*, a fin de resguardar la confidencialidad de la información versionada. Sobre ese repositorio se gestionó todas las etapas de la integración continua, desde la construcción del entorno contenedor hasta la ejecución automática de las pruebas.

Una vez habilitado el proyecto, se creó una rama de trabajo específica para los ajustes del pipeline y su validación previa. Luego, el repositorio se clonó en la estación de desarrollo para incorporar los archivos de configuración y los scripts de automatización necesarios.

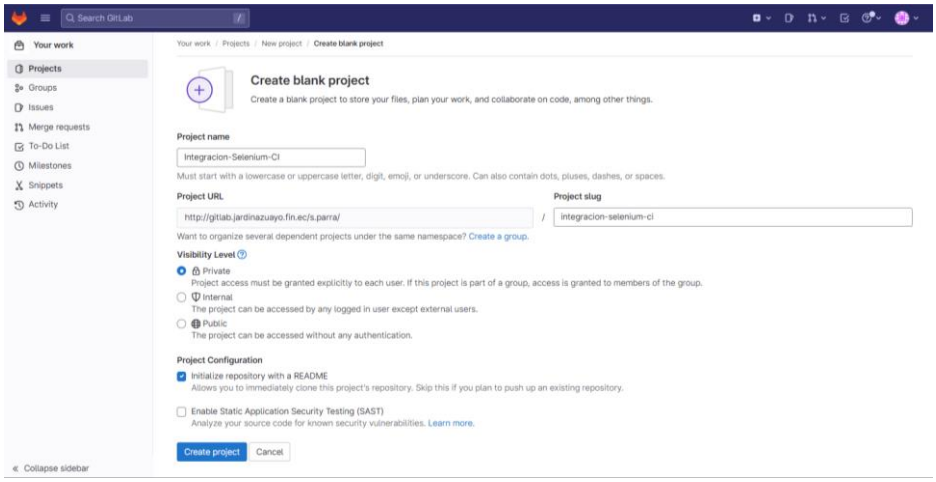


Figura. 27 Creación proyecto en GitLab

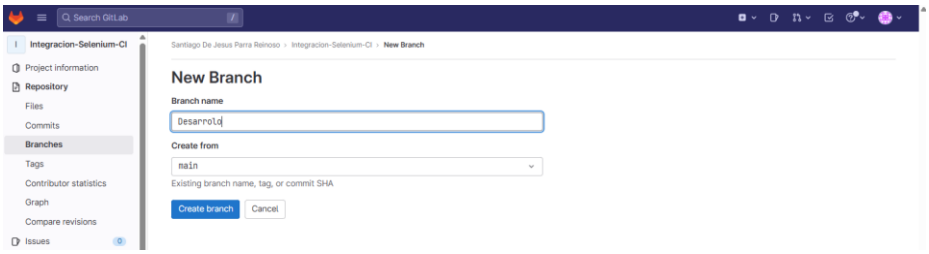


Figura. 28 Creación rama Desarrollo

El prototipo reutilizó la estructura definida para la ejecución de Selenium en Docker (véase en Configuración de proyecto), adaptándola a la arquitectura de GitLab CI. Para ello, se ajustó el Dockerfile y se añadió el archivo .gitlab-ci.yml, concebido como núcleo declarativo del pipeline.

El Dockerfile se organizó como compilación multietapa (multi-stage build), separando la fase de construcción de la fase de ejecución con el propósito de reducir el tamaño de la imagen, optimizar recursos y mejorar la portabilidad del artefacto (Docker Docs, s.f.). Con base en este enfoque, a continuación, se describió la configuración aplicada y su rol dentro del pipeline.

Dockerfile

```
# Etapa 1: Compilación con Maven
FROM maven:3.9.6-eclipse-temurin-17 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests

# Etapa 2: Ejecución con Selenium/Chrome
FROM selenium/video:ffmpeg-8.0-20250828
LABEL maintainer="test-dev"
WORKDIR /app
COPY --from=build /app/target/IntegracionSeleniumCI-1.0-SNAPSHOT.jar ./app.jar
COPY logs ./logs
CMD ["java", "-jar", "app.jar"]
```

Figura. 29 Configuración Dockerfile

En la primera etapa del Dockerfile (Figura. 29) se llevó a cabo la construcción del artefacto sobre una imagen oficial de Maven con JDK 17, asegurando alineación entre el entorno de compilación y las dependencias del proyecto. Se fijó el *WORKDIR* dentro del contenedor y, a continuación, se copiaron los archivos del proyecto mediante *COPY*. El *RUN* ejecutó la construcción conforme a lo declarado en el pom.xml, compilando el código y resolviendo librerías. Para acelerar la línea de ensamblaje del pipeline, se incorporó *-DskipTests*, aplazando las pruebas unitarias a una fase posterior y especializada en Selenium. Con esta estrategia, se separó explícitamente la compilación de la validación funcional, sin sacrificar calidad ni trazabilidad.

En la segunda etapa (Figura. 29) se adoptó una imagen base *selenium/standalone-chrome*, que incluye de forma nativa las utilidades necesarias para la automatización sobre navegadores reales. Este entorno, pensado para validaciones de interfaz gráfica, brindó un contexto estable y aislado para la ejecución de pruebas. En esta fase, un *COPY* trasladó el artefacto generado en la etapa previa, evitando introducir herramientas de build en la imagen de ejecución. Esta separación build/run redujo el tamaño final de la imagen y acortó tiempos de construcción y despliegue dentro del pipeline.

De forma complementaria, se incorporaron los archivos de registro (*logs*) producidos en la ejecución, a fin de conservar evidencias y reportes de prueba. Estos insumos resultaron clave para la auditoría, el diagnóstico de fallos y la

trazabilidad de cada ejecución, reforzando el proceso de aseguramiento de la calidad en un entorno escalable y orientado para su integración en flujos de Integración Continua (CI).

Las decisiones plasmadas en el Dockerfile se integraron como pieza central del flujo CI/CD en GitLab: el GitLab Runner invocó la imagen para construir, empaquetar y ejecutar pruebas funcionales de manera automatizada y repetible. La combinación entre Docker, Selenium y GitLab garantizó entornos limpios, aislados y reproducibles, mitigando discrepancias propias de configuraciones locales y elevando la confiabilidad del proceso. En paralelo, se fortaleció la gobernanza del pipeline, pues cada compilación y suite de pruebas se ejecutó bajo condiciones homogéneas, facilitando el seguimiento histórico de resultados y promoviendo la madurez operativa del ciclo de desarrollo continuo (GitLab Documentation, 2025).

Con esta base técnica consolidada, se pasó a definir el archivo .gitlab-ci.yml, núcleo de orquestación del pipeline. Ubicado en la raíz del proyecto, este estableció etapas, jobs, dependencias y entornos de ejecución que dieron forma a la secuencia automatizada: selección de imágenes Docker, compilación del código, ejecución de pruebas y publicación de artefactos bajo control estricto de versiones. En el apartado siguiente se detalló su estructura, funciones y la configuración aplicada en el prototipo de la Cooperativa de Ahorro y Crédito Jardín Azuayo, marcando el arranque de la integración plena del entorno dockerizado con el pipeline institucional de CI (Humble & Farley, 2010).

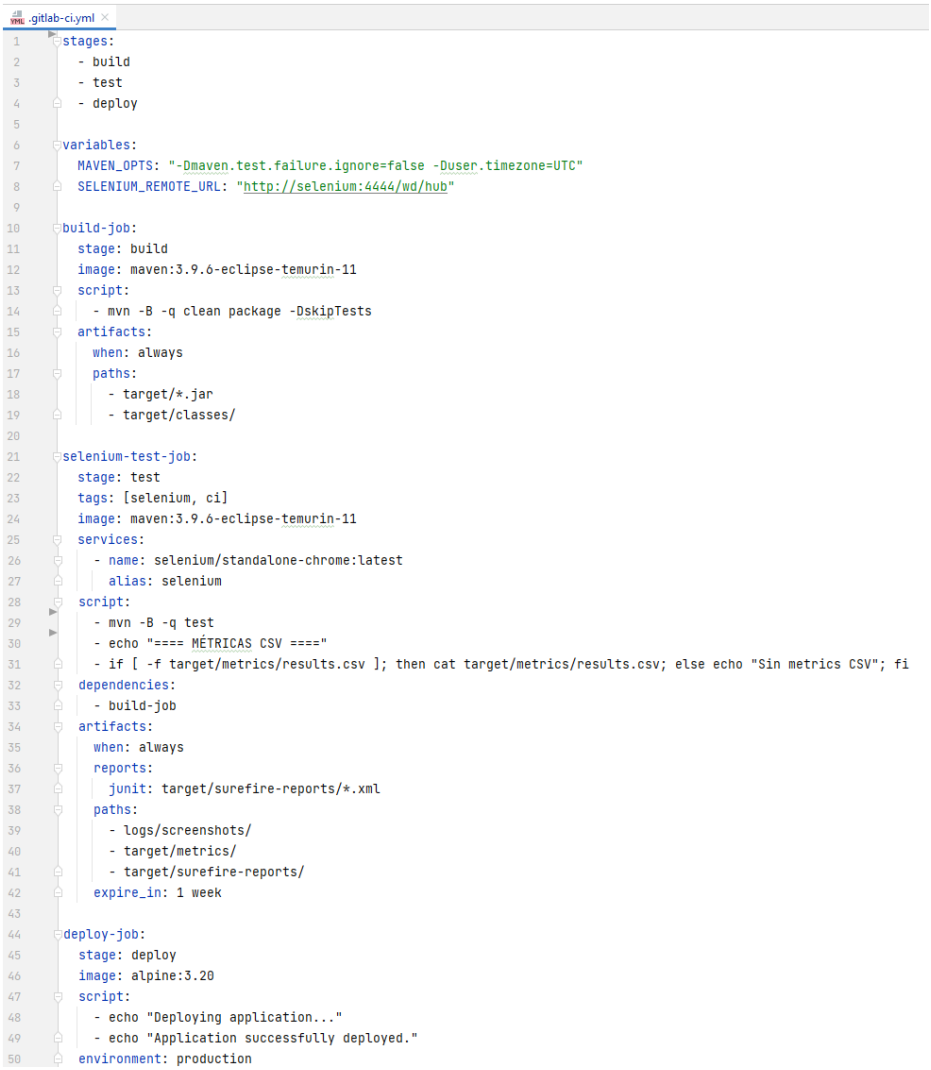


Figura. 30 Configuración gitlab-ci.yml

La Figura. 30 mostró el contenido del archivo .gitlab-ci.yml, base declarativa del pipeline. En este archivo se fijaron los parámetros que orquestaron, de manera secuencial, la compilación, las pruebas y el despliegue. A continuación, se expuso cada bloque, su estructura y la relación operativa con Docker y Selenium.

Definición de etapas: se declararon tres fases principales del pipeline: *build, test y deploy* que segmentaron el ciclo de vida del software en bloques lógicos:

- ***Build***: compiló el código fuente y generó el artefacto ejecutable (.jar).
- ***Test***: ejecutó los casos automatizados con Selenium para verificar funcionalidad.

- ***Deploy***: representó la etapa final, donde se simuló el despliegue del sistema compilado.

Esta división modular permitió un flujo de ejecución secuencial y controlado, en el cual cada etapa dependió del éxito de la anterior. Además, facilitó la detección temprana de errores, cualquier error en pruebas detuvo el paso, preservando la calidad del código integrado.

Variables globales: se definieron parámetros de alcance general para todo el pipeline:

- ***MAVEN_OPTS***: incluyó *-Dmaven.test.failure.ignore=false* (interrumpió el proceso ante fallas en pruebas) y *-Duser.timezone=UTC* (homogeneizó la zona horaria para resultados consistentes en logs).
- ***SELENIUM_REMOTE_URL***: apuntó a *http://selenium:4444/wd/hub*, habilitando la conexión del job de pruebas con el servicio remoto de Selenium en Docker.

Con ello, se aseguró interoperabilidad entre servicios y sincronía estable en la comunicación del pipeline.

Construcción del proyecto (*build-job*): en la etapa build se empleó la imagen *maven:3.9.6-eclipse-temurin-11*, que aportó JDK 11 y las herramientas de compilación. El script ejecutó:

- ***mvn -B -q clean package –DskipTests***: realizó limpieza, compilación y empaquetado del artefacto ejecutable.
- ***-DskipTests***: este parámetro omitió la ejecución de pruebas en esta fase optimizando los tiempos de compilación.

El bloque artifacts definió los archivos generados que fueron compartidos con otros trabajos del pipeline, incluyendo el directorio target/, donde se almacenaron las clases compiladas, garantizando trazabilidad de los resultados y evitando redundancias entre fases.

Pruebas automatizadas (*selenium-test-job*): núcleo del aseguramiento de calidad en la etapa test definido en el pipeline, este job utilizó la misma imagen de Maven e incorporó como *service selenium/standalone-chrome:latest*, proveyendo un navegador real para la ejecución remota. Los tags *[selenium, ci]* aseguraron el enrutamiento hacia el Runner adecuado (véase Figura. 30). El script incluyó dos comandos principales:

- ***mvn -B -q test***: ejecutó los tests de interfaz/funcionalidad sobre el navegador remoto.
- ***echo y cat***: generó métricas CSV en target/metrics/, brindando visibilidad inmediata de los resultados obtenidos.

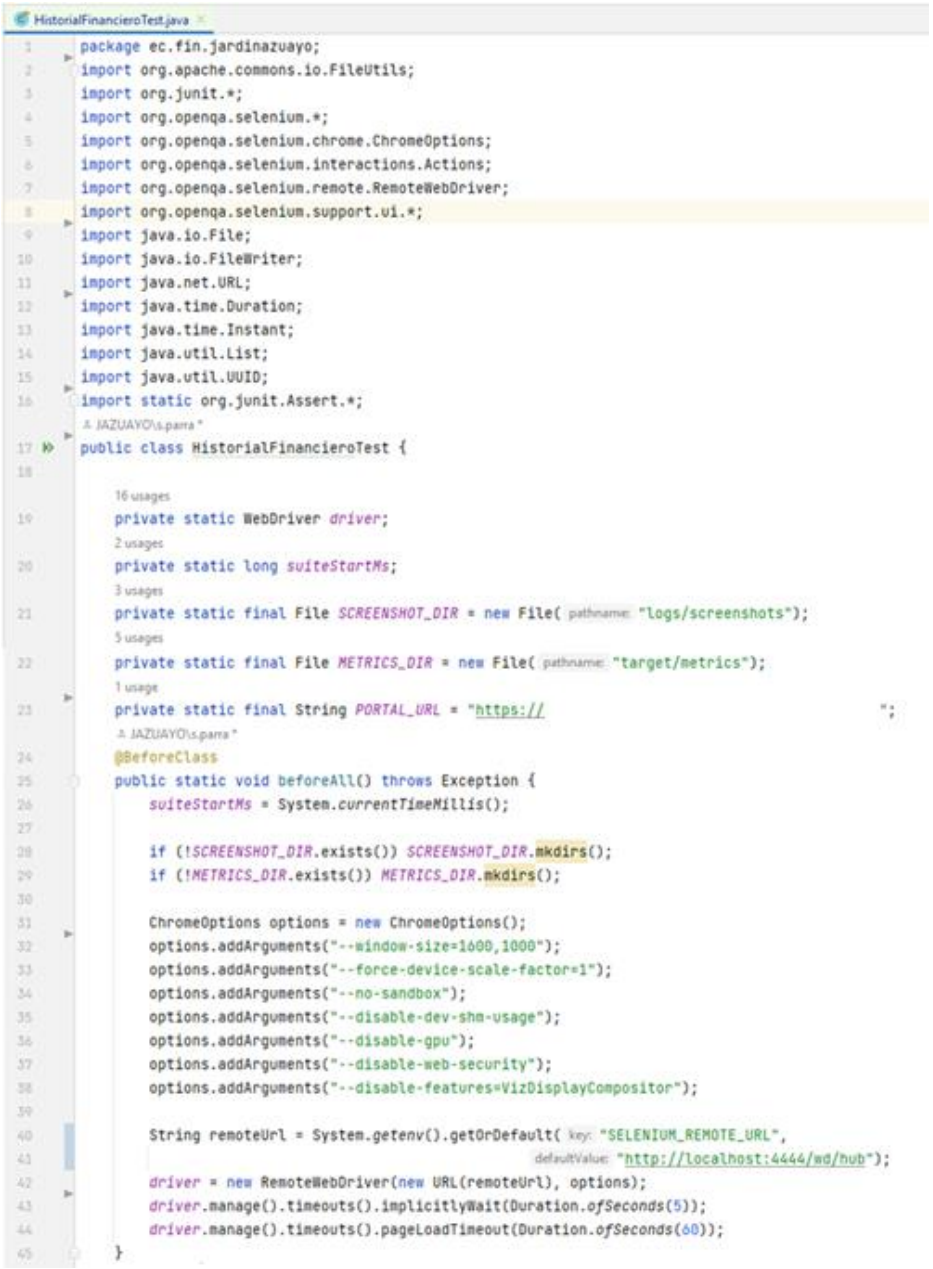
Mediante el bloque dependencies se acopló explícitamente a *build-job* con lo cual se aseguró que el código probado sea el mismo que fue compilado. Los artifacts publicaron JUnit (*target/surefire-reports/*.xml*), capturas y métricas, con retención de una semana (*expire_in: 1 week*), tal como se aprecia en la Figura. 30. Con ello, los resultados quedaron disponibles para análisis y auditoría diferidos.

Despliegue simulado (*deploy-job*): en la fase deploy (Figura. 30) se usó alpine:3.20 para ejecutar un guion liviano de validación de despliegue. El parámetro environment*: production* marcó la intención de compatibilidad con un escenario productivo real, manteniendo la integración dentro de la infraestructura institucional.

En conjunto, el .gitlab-ci.yml materializó la automatización integral del ciclo de desarrollo desde la construcción hasta el despliegue; cada job se ejecutó en su contenedor aislado, asegurando entornos limpios y reproducibles, sin dependencias externas. Este diseño incrementó la eficiencia operativa y la trazabilidad: cada commit o merge request disparó una cadena de validaciones que reforzó la calidad del software. De esta manera, la integración de Selenium dentro del pipeline en GitLab no solo aceleró el proceso de aseguramiento de calidad, sino que alineó las prácticas de la Cooperativa de Ahorro y Crédito Jardín Azuayo con los principios DevOps y CI/CD modernos, fortaleciendo su capacidad institucional de

innovación y mejora continua (Humble & Farley, 2010) (GitLab Documentation, 2025).

Cerrada la definición del pipeline, se precisó el componente funcional donde se ejecutaron las pruebas dentro de ese flujo. En este punto, la clase HistorialFinancieroTest.java asumió un rol central como módulo operativo de automatización con Selenium: integró las configuraciones y dependencias previamente establecidas en Docker, Maven y GitLab CI, y habilitó la ejecución controlada de los casos de prueba sobre un entorno estandarizado. A continuación, se presentó su configuración base y la lógica de implementación aplicada en el prototipo de automatización de la Cooperativa de Ahorro y Crédito Jardín Azuayo.



```java
package ec.fin.jardinazuayo;
import org.apache.commons.io.FileUtils;
import org.junit.*;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.interactions.Actions;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.openqa.selenium.support.ui.*;
import java.io.File;
import java.io.FileWriter;
import java.net.URL;
import java.time.Duration;
import java.time.Instant;
import java.util.List;
import java.util.UUID;
import static org.junit.Assert.*;
public class HistorialFinancieroTest {

    private static WebDriver driver;
    private static long suiteStartMs;
    private static final File SCREENSHOT_DIR = new File( pathname: "logs/screenshots");
    private static final File METRICS_DIR = new File( pathname: "target/metrics");
    private static final String PORTAL_URL = "https://                                    ";
    @BeforeClass
    public static void beforeAll() throws Exception {
        suiteStartMs = System.currentTimeMillis();

        if (!SCREENSHOT_DIR.exists()) SCREENSHOT_DIR.mkdirs();
        if (!METRICS_DIR.exists()) METRICS_DIR.mkdirs();

        ChromeOptions options = new ChromeOptions();
        options.addArguments("--window-size=1600,1000");
        options.addArguments("--force-device-scale-factor=1");
        options.addArguments("--no-sandbox");
        options.addArguments("--disable-dev-shm-usage");
        options.addArguments("--disable-gpu");
        options.addArguments("--disable-web-security");
        options.addArguments("--disable-features=VizDisplayCompositor");

        String remoteUrl = System.getenv().getOrDefault( key: "SELENIUM_REMOTE_URL",
                                                         defaultValue: "http://localhost:4444/wd/hub");
        driver = new RemoteWebDriver(new URL(remoteUrl), options);
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(5));
        driver.manage().timeouts().pageLoadTimeout(Duration.ofSeconds(60));
    }
```

Figura. 31 Configuración básica HistorialFinancieroTest.java

```java
private static void takeScreenshot(String name) {
    try {
        String file = String.format("%s/%s_%s.png",
                SCREENSHOT_DIR.getPath(), ts(), sanitize(name));
        File tmp = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
        FileUtils.copyFile(tmp, new File(file));
    } catch (Exception e) {
        System.err.println("No se pudo capturar screenshot: " + e.getMessage());
    }
}
9 usages  JAZUAYO\s.parra *
private static void writeMetric(String metric, String value) {
try {
    if (!METRICS_DIR.exists()) METRICS_DIR.mkdirs();
    File f = new File(METRICS_DIR,  child: "results.csv");
    boolean needsHeader = !f.exists() || f.length() == 0;

    try (FileWriter w = new FileWriter(f, java.nio.charset.StandardCharsets.UTF_8,  append: true)) {
        if (needsHeader) {
            w.write( str: "metric,value,timestamp\n");
        }
        w.write( str: metric + "," + value + "," + ts() + "\n");
    }
} catch (Exception e) {
    System.err.println("No se pudo escribir métrica " + metric + ": " + e.getMessage());
}
```

Figura. 32 Configuración Métricas y Screenshot

El análisis de HistorialFinancieroTest.java inició con la descripción de su paquete institucional, dependencias y foco de diseño. La clase declaró su paquete y realizó importaciones orientadas a: automatización del navegador, sincronización de esperas, gestión de evidencias y estructuración de pruebas con JUnit. Destacaron las bibliotecas de Selenium (*org.openqa.selenium.**), ChromeOptions, RemoteWebDriver, utilidades de Entrada/Salida y gestión de tiempo, así librerías *JUnit* y *Apache Commons IO* para manejo de archivos. Definida como pública, la clase concentró la configuración del driver, la creación de directorios de evidencia y el control del cronometraje global de la ejecución (véase Figura. 31).

En seguida, se identificaron variables clave y rutas de artefactos (véase Figura. 31), fijadas como constantes para habilitar la recolección automática en el pipeline:

- ***SCREENSHOT_DIR = logs/screenshots***: definió ruta de almacenamiento para capturas de pantalla.
- ***METRICS_DIR = target/metrics***: ruta de métricas en formato CSV.

Ambas rutas coincidieron con los artefactos declarados en el archivo *.gitlab-ci.yml*, lo que permitió su recolección automática por GitLab CI y reforzó la trazabilidad de los resultados generados.

Respecto a la inicialización de la clase (véase Figura. 31), el flujo estableció marcas de tiempo, verificó/creó carpetas de evidencia y configuró las opciones de Chrome antes de instanciar el driver remoto. Se destacaron los siguientes aspectos:

- **Evidencias**: cuando los directorios no existían, se crearon previo a cualquier caso de prueba, evitando fallas por rutas.
- **Configuración de ChromeOptions**: se añadieron parámetros orientadas a ejecución en contenedores/CI: *--no-sandbox, --disable-dev-shm-usage, --disable-gpu, --disable-web-security, --disable-features=VizDisplayCompositor*. Estas opciones estabilizaron la ejecución headless y mitigaron errores por memoria compartida o GPU en Docker/Runner.
- **Selenium remoto parametrizado**: la URL del hub se obtuvo desde la variable de entorno *SELENIUM_REMOTE_URL* (con valor predeterminado *http://localhost:4444/wd/hub*), permitiendo ejecutar el mismo binario en local, Docker o GitLab CI sin cambios de código.
- **Tiempos de espera globales**: se estableció una espera implícita (5 segundos) y una carga de página (60 segundos) para absorber latencias de red o procesos de.

En la fase de finalización (véase Figura. 31), la clase calculó y registró el tiempo total de ejecución como métrica de desempeño y cerró el driver de manera controlada. Con ello, se previnieron sesiones huérfanas, se liberaron recursos del host o runner y se persistió la métrica mediante el método *writeMetric*, quedando disponible para el job de CI que imprime el CSV al término de la ejecución.

En cuanto a utilidades de evidencia y observabilidad (véase Figura. 32), se incorporaron métodos auxiliares que estandarizaron la generación de artefactos que optimizaron la trazabilidad del proceso de pruebas, estos fueron:

- **Capturas de pantalla**: el método *takeScreenshot* utilizó TakesScreenshot y almacenó imágenes PNG en logs/screenshots/. Se aplicó sanitización del

nombre y un UUID (Identificador Único Universal) lo que evitó duplicidad y facilitó el rastreo en los logs del pipeline.

- **Métricas CSV**: el método *writeMetric(metric, value)* generó o actualizó el archivo en target/metrics/results.csv con sello temporal UTC (Tiempo Universal Coordinado), garantizando correlación temporal en entornos CI/CD. El método validó la existencia del directorio lo que evitó errores de escritura conservando la integridad de los datos recopilados.
- **Estándares de nombrado**: el método *sanitize* restringió caracteres no válidos y anexó un identificador único, útil en ejecuciones concurrentes o repetidas dentro del pipeline.

Estas utilidades quedaron alineadas en la configuración del archivo .gitlab-ci.yml (véase Figura. 30), de modo que el Runner recolectó automáticamente los artefactos sin configuraciones adicionales.

En materia de sincronización y robustez, la clase incorporó mecanismos de control que aseguraron la estabilidad del flujo de pruebas y esperas explícitas con *WebDriverWait* para garantizar visibilidad y disponibilidad antes de interactuar con elementos, mientras que los timeouts globales definidos en *@BeforeClass* establecieron márgenes razonables para cargas de página y componentes dinámicos. Paralelamente, el registro de tiempos a través del método *writeMetric* aportó indicadores de rendimiento utilizados para seguimiento en entornos locales, dockerizados o gestionados por GitLab Runner.

En síntesis, la clase HistorialFinancieroTest.java presentó una configuración sólida, estandarizada y portable, orientada a:

- Ejecución de pruebas con Selenium remoto.
- Generación de evidencias auditables mediante screenshots y métricas CSV.
- Mantuvo sincronización resistente mediante timeouts y esperas.

Su arquitectura implementó una inicialización global (*@BeforeClass*), clausura controlada (*@AfterClass*), utilitarios de evidencia y parametrización del endpoint de

Selenium, el cual se integró directamente con Docker y GitLab CI, garantizando que el Runner institucional recopilara y preservara los artefactos claves del proceso de aseguramiento de la calidad de software.

3.6.2 Configuración de Runner Local

En el marco de la Integración Continua (CI), se configuró un Runner local de GitLab como agente de ejecución dedicado, instalado en un equipo de la infraestructura institucional. Este componente fue el encargado de materializar las tareas definidas en el archivo .gitlab-ci.yml, procesando de forma controlada y reproducible las etapas de compilación, pruebas, análisis estático y despliegue. Para habilitar la comunicación segura con el servidor de GitLab, el Runner se registró mediante el token del proyecto o grupo, estableciendo así el canal de intercambio de trabajos y la asignación de ejecuciones según las reglas del pipeline lo que optimizó el uso de recursos internos y minimizó la dependencia de servicios externos, lo que evitó exponer información sensible (GitLab Documentation Runner, 2025).

Desde el punto de vista operativo, se definieron entornos de ejecución a la medida del proyecto, lo que permitió fijar versiones específicas de JDK, Maven y dependencias, y reducir la variabilidad entre máquinas. Con ello, el equipo evitó la dependencia de runners compartidos y disminuyó el tiempo en cola, favoreciendo la eficiencia del ciclo de validación. Al mantenerse dentro de la red privada, la ejecución del pipeline preservó la confidencialidad de artefactos, registros y credenciales, y facilitó la trazabilidad mediante auditorías y registros centralizados de cada job (GitLab Documentation Runner, 2025).

La instalación se apoyó en las guías oficiales tanto para Windows como para Linux y, cuando aplicó, en Git for Windows para la gestión del servicio. Tras la instalación, se procedió al registro del Runner con las etiquetas (*tags*) definidas en el pipeline (por ejemplo, *selenium, ci*), garantizando que el job *selenium-test-job* fuese despachado exclusivamente al agente preparado para pruebas de navegador. Esta asociación de etiquetas evitó conflictos con otros ejecutores y aseguró la

consistencia del entorno a lo largo de sucesivas ejecuciones (GitLab Documentation Runner, 2025).

En síntesis, la adopción del Runner local se consolidó como una decisión técnica clave en la integración continua: aportó autonomía operativa, reproducibilidad de resultados y seguridad en el manejo de información sensible. Con su puesta en marcha, la Cooperativa de Ahorro y Crédito Jardín Azuayo contó con un eslabón estable dentro de su infraestructura de CI/CD, alineado con buenas prácticas de automatización y con los objetivos de aseguramiento de la calidad establecidos para el proyecto.

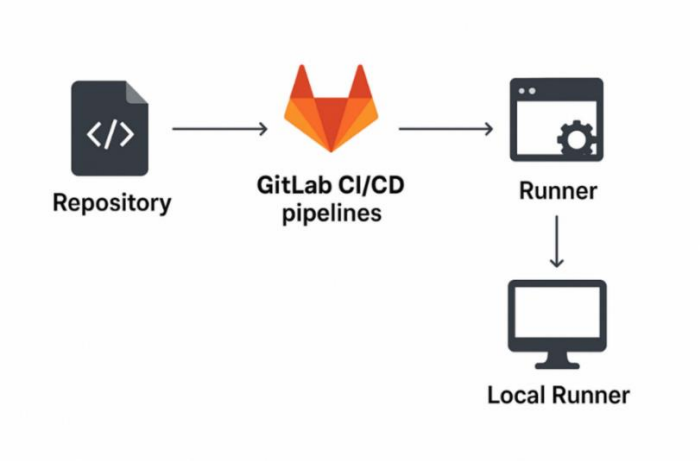


Figura. 33 Flujo interacción entre GitLab Runner, Repositorio y el pipeline CI/CD (GitLab Documentation Runner, 2025)

Una vez establecida la importancia técnica del Runner local dentro del flujo de CI, se procedió con su instalación y configuración en el entorno de desarrollo. Para ello, se descargó el instalador oficial desde el portal de GitLab y se siguió la guía de referencia para Windows (GitLab Documentation Runner, 2025).El proceso inició con la creación del directorio C:\GitLab-Runner, donde se almacenó el ejecutable descargado. Luego, desde una consola PowerShell con privilegios de administrador, se navegó hasta dicho directorio y se habilitó el servicio del Runner con el comando: *.\gitlab-runner.exe install.*

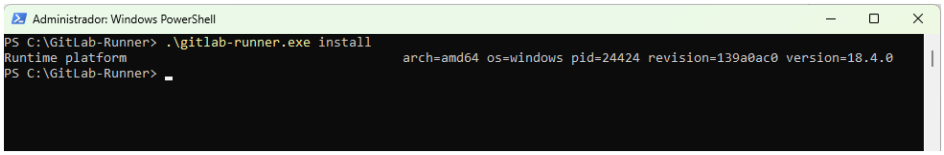


Figura. 34 Instalación Runner Local

A continuación, se registró el agente contra el proyecto institucional en GitLab, ingresando a Settings - CI/CD - Runners para obtener el token de registro. Con ese dato, más la URL del servidor, una descripción identificativa del Runner y el conjunto de etiquetas de ejecución (véase Figura. 35), se ejecutó el registro desde PowerShell, ubicándose en la carpeta de instalación del Runner y lanzando la instrucción de registro correspondiente (Figura. 36). Con ello, se estableció el canal seguro entre el servidor de GitLab y el agente local, quedando definidos los parámetros esenciales del entorno de ejecución (Figura. 37) (GitLab Documentation Runner, 2025).

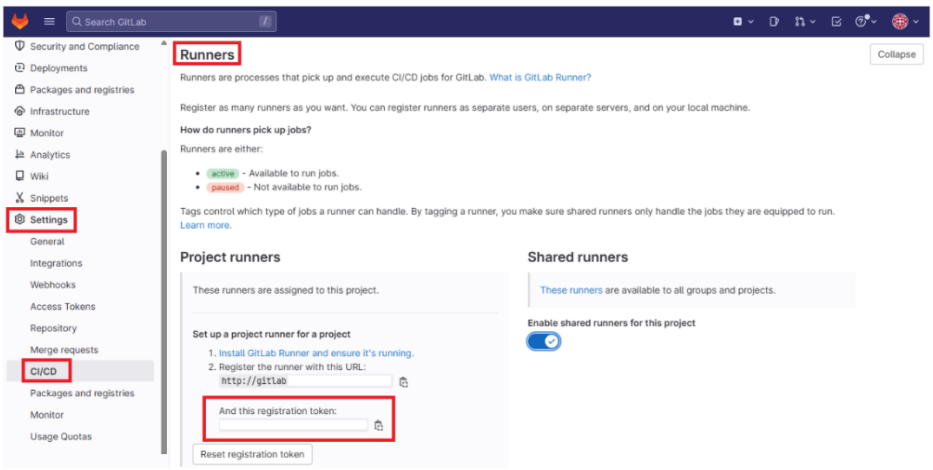


Figura. 35 Configuración Runner - Token

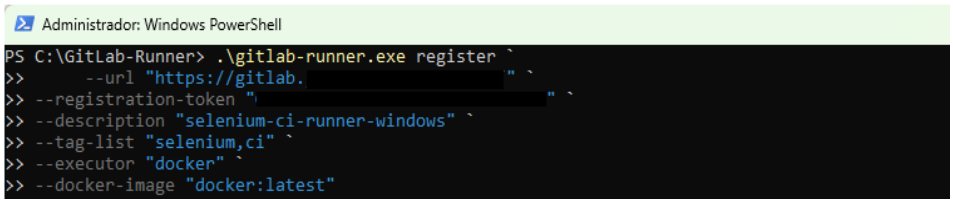


Figura. 36 Comando registro Runner Local

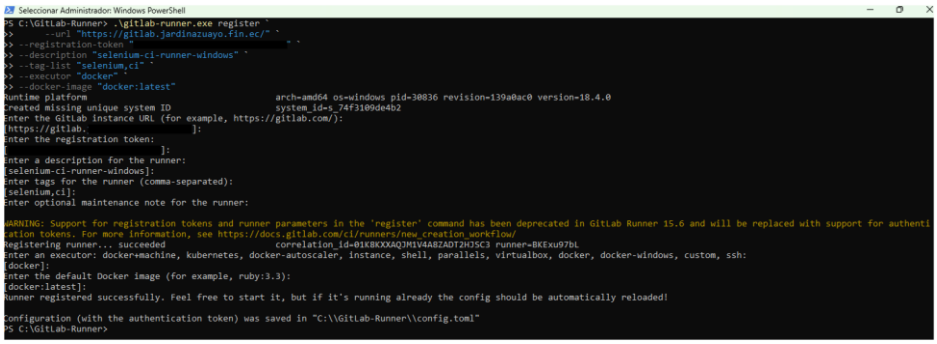


Figura. 37 Resultado Registro Runner Local

Desde el punto de vista técnico, cada argumento del registro (véase Figura. 36) cumplió una función específica:

- ***--url*:** indicó la dirección del servidor GitLab institucional.
- ***--registration-token***: autenticó el vínculo con el proyecto o grupo, asegurando la comunicación.
- **--description:** asignó un nombre único al Runner para su identificación en la interfaz de administración de GitLab.
- **--tag-list**: definió las etiquetas que direccionaron tareas específicas del pipeline (por ejemplo, pruebas Selenium, compilación) hacia este agente, optimizando el uso de recursos.
- ***--executor y --docker-image***: establecieron el modo de ejecución (Docker) y la imagen base con la que se crearon los entornos aislados para construir, validar y desplegar.

Concluido el registro, el servicio se inició ejecutando el comando: *.\gitlab-runner.exe start* (Figura. 38), quedando el agente operativo y listo para recibir instrucciones del servidor, lo que habilitó la ejecución automatizada del pipeline y garantizó una comunicación bidireccional y confiable entre el entorno local y la plataforma institucional de integración, lo que consolidó la infraestructura de pruebas automatizadas y el aseguramiento de la calidad del software en la Cooperativa de Ahorro y Crédito Jardín Azuayo.

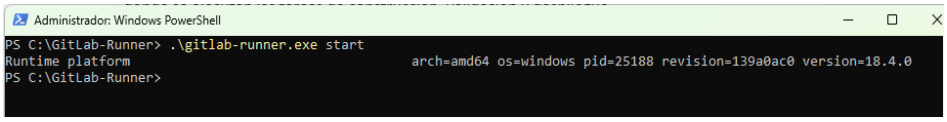


Figura. 38 Iniciar Runner Local

Cuando se requirió una verificación adicional, se validó el funcionamiento mediante el comando: .*gitlab-runner.exe run* desde la terminal de PowerShell. Esta sentencia (Figura. 39) mostró en consola los registros de actividad y el estado del enlace con GitLab, confirmando que el Runner procesaba tareas correctamente antes de dar paso a la ejecución formal del pipeline de Integración Continua (CI). De esta manera, se aseguró la estabilidad del entorno de automatización y la sincronización efectiva entre componentes locales y remotos.

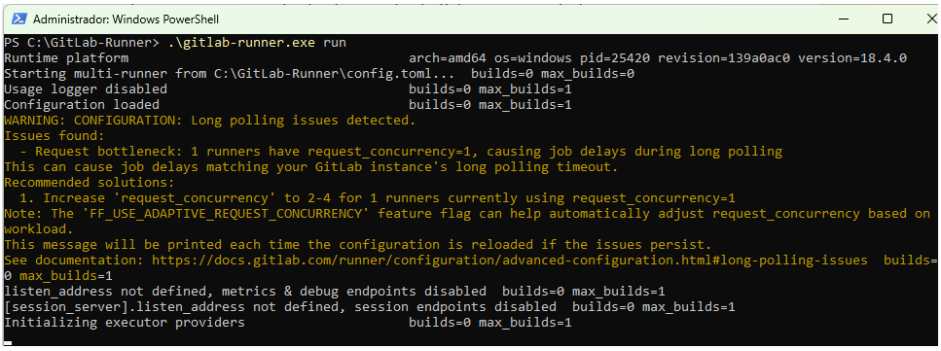


Figura. 39 Validación Runner Local

3.6.3 Ejecución del Prototipo de Integración Continua (CI)

Con la configuración del .gitlab-ci.yml (Figura. 30) y el registro operativo del Runner local (Figura. 36), se puso en marcha el prototipo de Integración Continua. Esta etapa materializó el flujo automatizado entre GitLab, Docker y Selenium, de modo que cada job definido en el pipeline se ejecutó en secuencia, dentro de contenedores limpios y aislados, y bajo los parámetros de calidad previamente establecidos. El propósito fue verificar la operatividad extremo a extremo: construcción del artefacto, ejecución de pruebas y generación de artefactos (reportes, métricas y evidencias), todo sin intervención manual.

La activación del pipeline se desencadenó al confirmar cambios (commit) o crear un merge request en el repositorio. A partir de ese evento, GitLab mostró el pipeline en CI/CD - Pipelines y ejecutó las tres etapas declaradas en el .gitlab-ci.yml:

- **Build**: compiló el código y empaquetó el artefacto ejecutable (.jar), publicándolo como artifact para los siguientes trabajos.
- **Test**: lanzó las pruebas automatizadas sobre un Selenium Standalone Chrome en Docker, recopilando reportes JUnit, capturas y métricas en las rutas esperadas.
- **Deploy:** simuló la entrega del producto, validando que el artefacto generado pudiera promocionarse hacia un entorno destino sin inconsistencias.

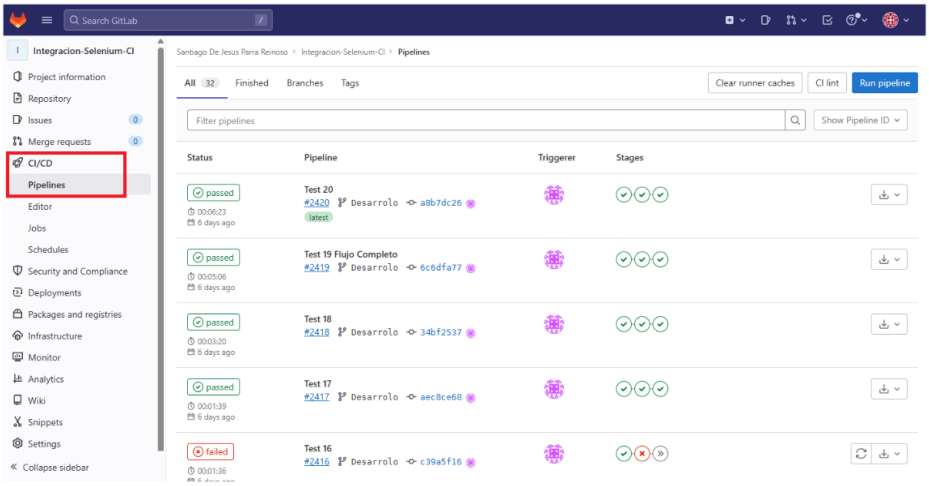


Figura. 40 Commits en repositorio

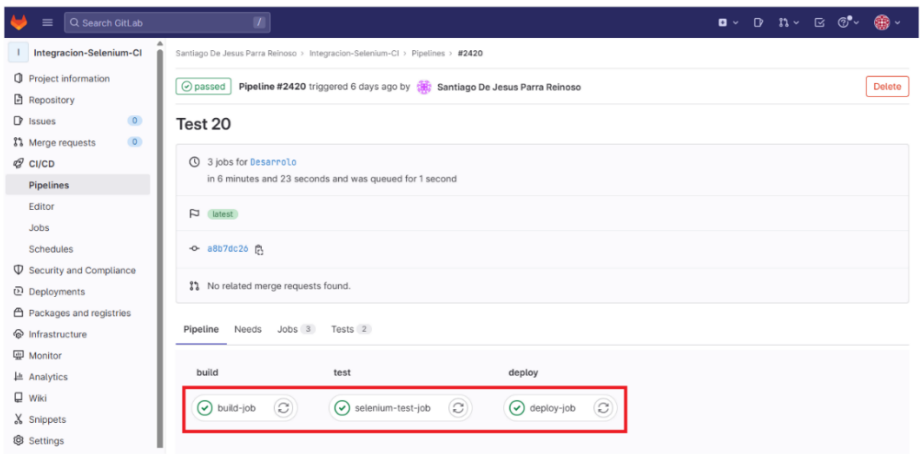


Figura. 41 Etapas ejecución pipeline

Durante la ejecución del pipeline, cada trabajo (job) se levantó dentro de un contenedor temporal administrado por Docker, lo que aseguró un entorno limpio y reproducible. La consola de GitLab expuso en tiempo real los registros de cada fase, permitiendo seguir el avance y aislar oportunamente fallos de compilación o errores de prueba. Esta visibilidad fortaleció la trazabilidad del proceso y aportó evidencia directa para la mejora continua del ciclo de desarrollo (GitLab Documentation, 2025).

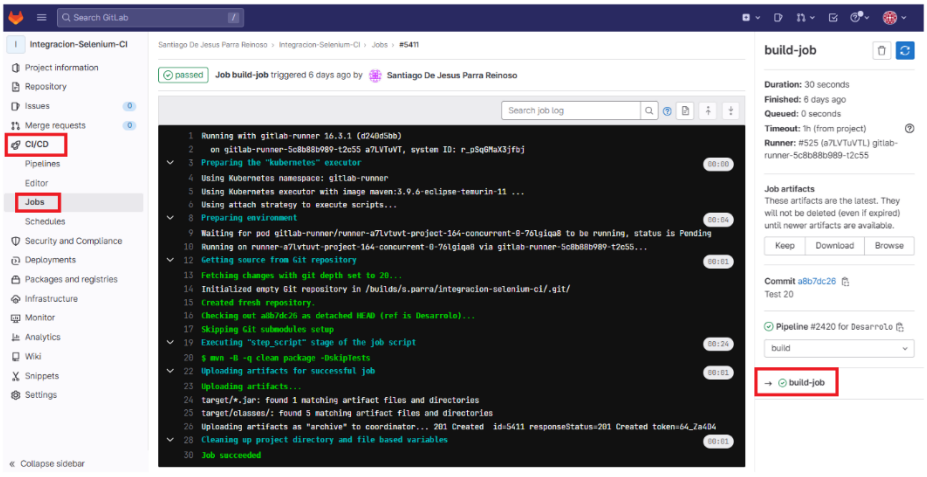


Figura. 42 Log etapa build-job

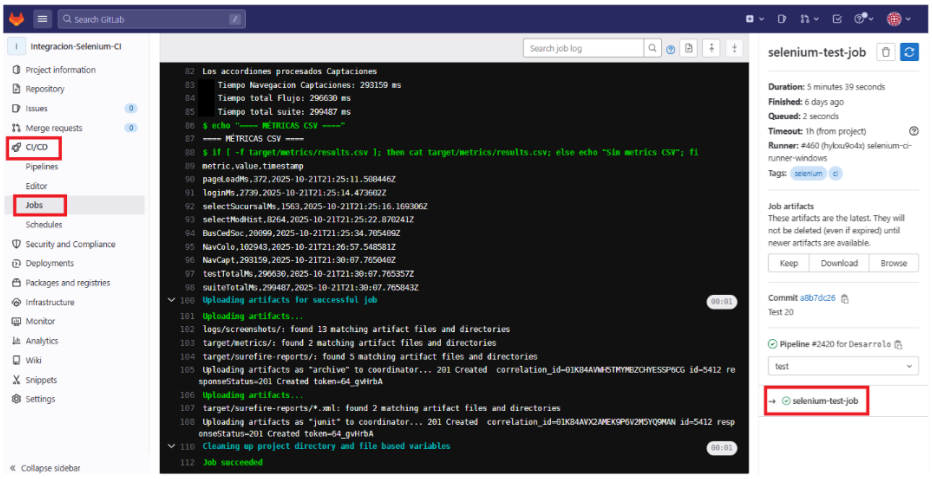


Figura. 43 Log etapa test

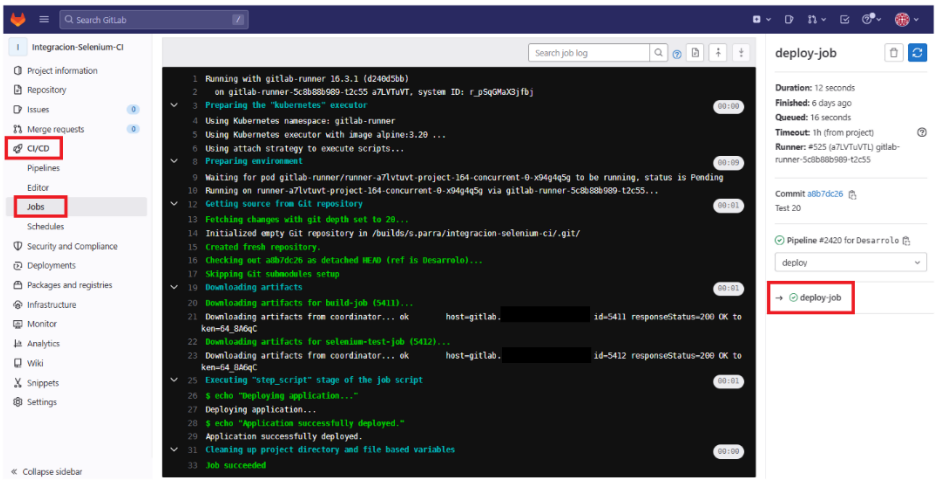


Figura. 44 Log etapa deploy

En la etapa Test, la tarea *selenium-test-job* utilizó la imagen de Selenium con Chrome y disparó la ejecución automatizada de la clase HistorialFinancieroTest.java. El Runner local operó como puente entre GitLab y los contenedores, procesando las pruebas y generando artefactos: capturas de pantalla, métricas en CSV y reportes JUnit. Dichos resultados se persistieron en las rutas definidas por el pipeline (*logs/screenshots/* y *target/metrics/*), quedando disponibles para su consulta posterior (véase Figura. 30).

Concluida la ejecución del pipeline, GitLab consolidó automáticamente los artefactos de evidencia de la integración continua. Estos pudieron inspeccionarse o descargarse desde CI/CD – Pipelines en la interfaz de GitLab, seleccionando la job correspondiente. Tal como se ilustró en la Figura. 45, la sección Artifacts listó los productos de cada job, destacándose *selenium-test-job:archive*, donde se almacenaron los elementos configurados en el archivo .gitlab-ci.yml.

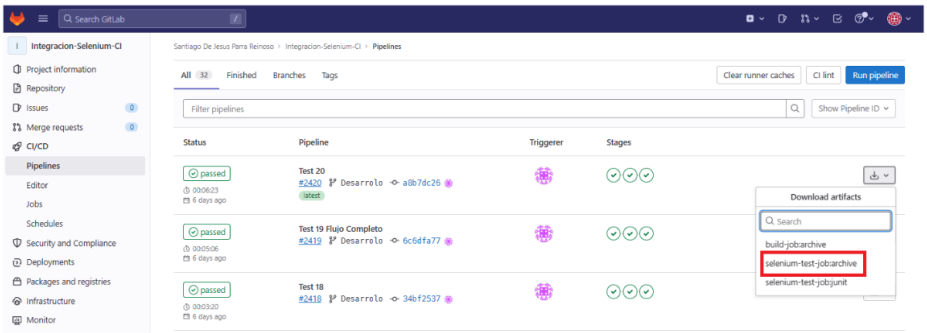


Figura. 45 Sección de revisión de evidencias

Dentro de ese paquete comprimido se incluyeron las capturas, los reportes JUnit y las métricas en formato CSV, insumos clave para la validación del modelo. En conjunto, estos archivos documentaron la ejecución en entornos Dockerizados, registrando tanto resultados funcionales como tiempos de respuesta y comportamiento del sistema. Su revisión posterior permitió verificar la correcta operación del pipeline, estimar la cobertura de pruebas y mantener la trazabilidad entre iteraciones del código.

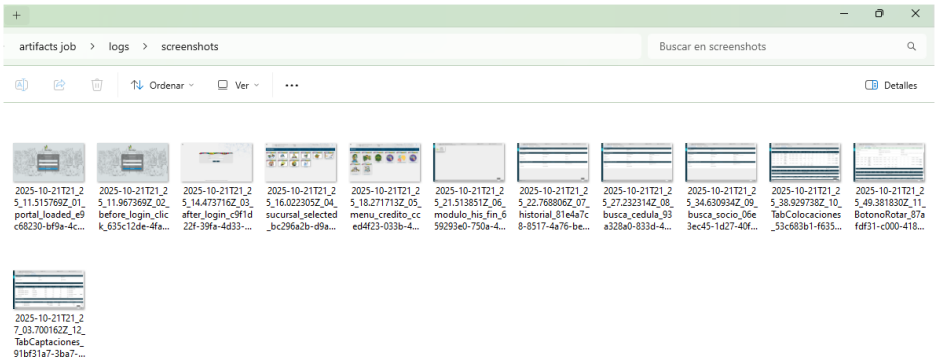


Figura. 46 Screenshots test CI

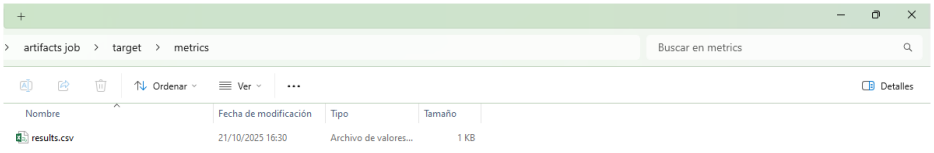


Figura. 47 Métricas test CI

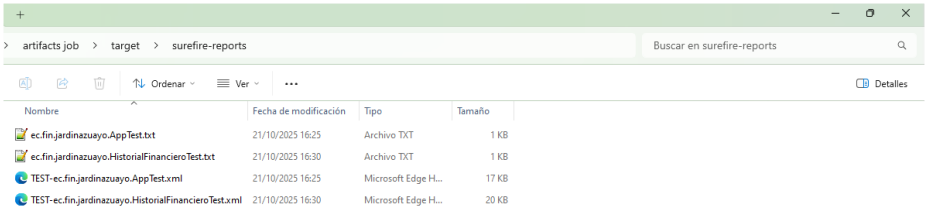


Figura. 48 Resultado ejecución test CI

De esta manera, la disponibilidad de artefactos descargables consolidó la transparencia del proceso de integración continua, al facilitar la revisión conjunta entre los equipos de desarrollo y aseguramiento de la calidad. Su resguardo dentro de GitLab aseguró además una gestión centralizada y reproducible de los

resultados, en concordancia con las políticas de control de versiones y con la necesidad de evidencia técnica institucional de la Cooperativa de Ahorro y Crédito Jardín Azuayo.

En conjunto, la puesta en marcha del pipeline GitLab CI/CD con Selenium y Docker representó un hito en la estandarización del aseguramiento de calidad del software institucional. El modelo no solo optimizó la ejecución de pruebas mediante entornos reproducibles y automatizados, sino que también reforzó la trazabilidad de resultados a través de artefactos verificables y controlados. Como consecuencia, la Cooperativa de Ahorro y Crédito Jardín Azuayo consolidó una infraestructura moderna y ágil, alineada con los principios de DevOps y con un enfoque de mejora continua de sus procesos tecnológicos.

4 Resultados y discusión

4.1 Introducción

Este capítulo presentó los hallazgos y el análisis técnico derivados de la implementación del modelo de automatización del aseguramiento de la calidad de software en la Cooperativa de Ahorro y Crédito Jardín Azuayo. La evaluación se sustentó en la ejecución controlada del prototipo dentro de un entorno de Integración Continua (CI), orquestado con GitLab, Docker y Selenium, lo que permitió constatar la eficiencia, estabilidad y reproducibilidad del flujo de pruebas automatizadas.

El propósito de la sección consistió en valorar el desempeño del sistema propuesto a partir de los indicadores emitidos por el pipeline de CI/CD: tiempos de ejecución, generación de artefactos, reportes de validación y resultados funcionales. Con base en estas evidencias, se determinó el grado en que el modelo contribuyó a la reducción de errores, al uso eficiente de recursos y a la mejora del flujo de desarrollo.

Así mismo, los resultados posibilitaron contrastar la propuesta metodológica con su aplicación práctica, identificando beneficios alcanzados y áreas de mejora. Esta revisión confirmó la pertinencia y viabilidad técnica del enfoque de automatización como parte integral del proceso de desarrollo institucional.

Finalmente, el capítulo se organizó en torno a la exposición de resultados cuantitativos y cualitativos, el análisis interpretativo de métricas de desempeño y una valoración global del modelo implementado, asegurando una visión completa de los beneficios obtenidos y de la madurez técnica lograda tras incorporar prácticas de automatización al proceso de aseguramiento de calidad.

4.2 Resultados Ejecución Pipeline CI

La ejecución del pipeline configurado en GitLab CI/CD validó la operatividad integral del modelo de automatización, verificando la correcta integración entre Selenium, Docker y GitLab Runner. Cada job definido en el archivo .gitlab-ci.yml se ejecutó de manera secuencial y controlada, evidenciando un comportamiento estable del entorno y una generación consistente de artefactos (Figura. 41).

En la etapa Build, el sistema compiló satisfactoriamente el código mediante la imagen base de Maven, produciendo los artefactos en target/*.jar y target/classes/. Esta fase confirmó la compatibilidad del entorno de compilación con la arquitectura del proyecto y la resolución correcta de dependencias declaradas en el pom.xml (Figura. 42).

Posteriormente, en la etapa Test, la tarea *selenium-test-job* ejecutó las pruebas automatizadas en el entorno Dockerizado utilizando Selenium con navegador Chrome en modo headless. En esta fase se verificó la comunicación entre el Runner local y el servidor de GitLab, se corroboró la respuesta adecuada del pipeline ante eventos de integración y se observó la inicialización del navegador, la carga de componentes web y la ejecución del flujo de autenticación de la aplicación institucional (Figura. 43).

Los artefactos generados quedaron almacenados en el repositorio del pipeline y se visualizaron desde Artifacts en la interfaz de GitLab (Figura. 45). En particular, el paquete *selenium-test-job:archive* contuvo los insumos clave definidos en el .gitlab-ci.yml:

- **Capturas de pantalla**: producidas por HistorialFinancieroTest.java, ubicadas en logs/screenshots/, que evidenciaron la interacción automatizada con la interfaz (Figura. 46).
- **Reportes JUnit**: fueron almacenados en target/surefire-reports/, que documentaron el resultado de las pruebas unitarias y funcionales (Figura. 48).

- **Métricas CSV**: almacenados en target/metrics/results.csv, que registraron tiempos de ejecución, estados de validación y eventos de éxito o fallo (Figura. 47).

El análisis de estos artefactos (Figura. 46) confirmó la correcta operación del pipeline en todas sus fases, con resultados reproducibles a través de distintas ejecuciones. Las métricas mostraron una disminución significativa del tiempo de validación manual y una mejora en trazabilidad y documentación del proceso (Figura. 47).

Desde la perspectiva técnica, los resultados demostraron la madurez del entorno de automatización, su adecuada integración con los repositorios de código y la infraestructura de contenedores. A su vez, la generación automática de evidencias fortaleció la transparencia del proceso, habilitando auditorías técnicas y comparativas de desempeño entre iteraciones del software.

En síntesis, la ejecución del pipeline (Figura. 48) validó la coherencia, confiabilidad y escalabilidad del modelo, y dejó sentada una base sólida para el análisis cuantitativo y cualitativo de los indicadores de desempeño que se presenta en la siguiente sección.

4.3 Análisis de métricas e indicadores

La ejecución del prototipo dentro del pipeline de GitLab CI/CD proporcionó un conjunto de métricas cuantitativas que caracterizaron el desempeño técnico del entorno automatizado y la eficiencia del flujo de pruebas. Los valores registrados en results.csv correspondieron a tiempos (ms) de las acciones principales del script HistorialFinancieroTest.java. Estas métricas sirvieron como insumo para valorar estabilidad, rendimiento y capacidad de respuesta del sistema bajo prueba, así como la efectividad del proceso de automatización.

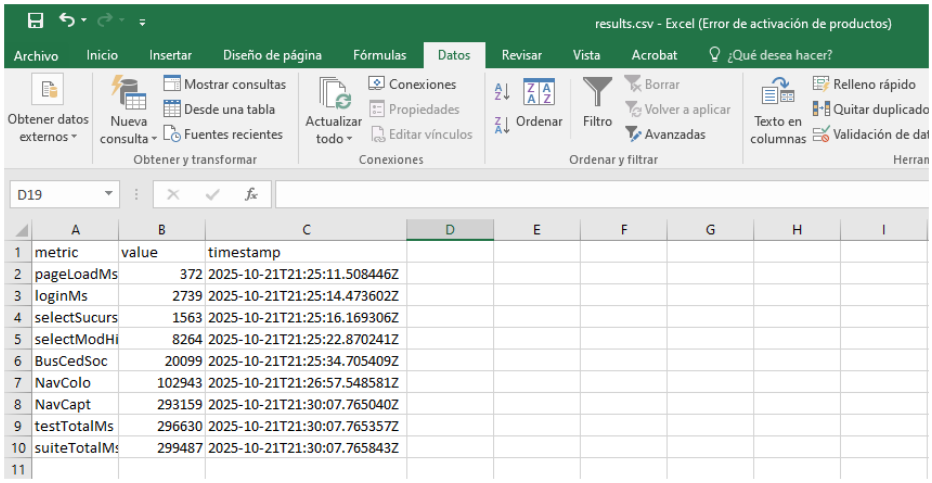


Figura. 49 Métricas proceso/milisegundos

4.3.1 Indicadores de desempeño del flujo de pruebas

Los indicadores obtenidos (Figura. 49) mostraron una ejecución estable y controlada del flujo funcional. El primer valor, pageLoadMs (372 ms), representó el tiempo promedio de carga inicial del portal institucional, evidenciando una respuesta eficiente del servidor y una comunicación adecuada entre el contenedor de Selenium y Google Chrome en modo headless. Este resultado se relacionó con la configuración aplicada en HistorialFinancieroTest.java (Figura. 31), donde los parámetros *--disable-dev-shm-usage* y *--no-sandbox* fueron declarados en ChromeOptions, lo que optimizó la carga y el renderizado en entornos Docker o de Integración Continua al mitigar problemas de memoria compartida y restricciones de sandbox lo que evidenció una ejecución fluida de las pruebas automatizados.

La segunda métrica, loginMs (2 739 ms), reflejó el tiempo total del proceso de autenticación (ingreso de credenciales, validación y respuesta del backend). El valor resultó coherente con un entorno institucional con controles adicionales de seguridad, logrando equilibrio entre protección y usabilidad.

En la fase de navegación, selectSucursalMs (1563 ms) cuantificó la selección de sucursal/módulo y se mantuvo en márgenes apropiados para pruebas funcionales automatizadas, esto validó que el sistema respondió de manera coherente ante operaciones de navegación intermedia.

El parámetro selectModHist (8264 ms) midió la apertura del módulo Historial Financiero; el tiempo fue consistente con la carga multiple de componentes dinámicos y la recuperación de datos del servidor. La estabilidad en esta etapa confirmó el uso efectivo de esperas explícitas (*WebDriverWait*) para evitar fallos por latencia o sincronización.

Finalmente, BusCedSoc (20099 ms) agregó la búsqueda de información del socio y la visualización del historial financiero integrando la consulta al servicio y procesamiento de resultados en la interfaz; al ser el tramo de mayor carga, concluyó sin interrupciones, mostrando robustez del entorno Dockerizado y la capacidad del pipeline para sostener operaciones de alta demanda.

4.3.2 Evaluación de eficiencia y trazabilidad

En términos analíticos, las métricas mostraron comportamiento estable y repetible entre ejecuciones consecutivas, acreditando la madurez del entorno CI/CD. El tiempo total se mantuvo en rangos aceptables para aplicaciones financieras con interacción de bases de datos y módulos web complejos.

La disponibilidad de evidencias complementarias: capturas (Figura. 46) y reportes JUnit (Figura. 47) reforzó la trazabilidad del proceso, al correlacionar tiempos medidos con eventos observados. La exportación automática de métricas a CSV facilitó seguimiento histórico y construcción de indicadores comparables para iteraciones futuras del pipeline.

Desde la ingeniería de software, este nivel de observabilidad respaldó la capacidad del sistema bajo principios de automatización continua, reproducibilidad y control de calidad integrado, pilares del modelo (Humble & Farley, 2010).

En conjunto, los resultados dieron cuenta de que el prototipo de integración Selenium-Docker-GitLab CI/CD cumplió los objetivos técnicos planteados:

- **Diagnosticar los procesos de aseguramiento de calidad**: las métricas y evidencias permitieron identificar cuellos de botella y validar mejoras en el flujo.
- **Seleccionar e integrar herramientas** (ESLint, Selenium, Docker y GitLab CI/CD): la solución quedó alineada con la infraestructura institucional y demostró interoperabilidad efectiva.
- **Diseñar e implementar el modelo de automatización**: el pipeline ejecutó pruebas de forma autónoma y reproducible, con controles y artefactos estandarizados.
- **Generar métricas cuantificables y trazables**: el CSV, las capturas y los reportes JUnit proveyeron insumos objetivos para evaluación de rendimiento y auditoría técnica.
- **Fortalecer la cultura DevOps y la mejora continua**: la integración potenció la retroalimentación temprana, redujo retrabajos y elevó la transparencia del ciclo de vida.

Desde una perspectiva estratégica, la implantación del prototipo en el ecosistema de la Cooperativa de Ahorro y Crédito Jardín Azuayo marcó un avance hacia la consolidación de prácticas DevOps y aseguramiento continuo de la calidad. La automatización de pruebas y su acoplamiento directo con los pipelines habilitaron detección temprana de fallos, contuvieron costos de corrección y mantuvieron estabilidad del producto frente a cambios evolutivos. Estos resultados sentaron la base para ampliar la cobertura de pruebas y profundizar la medición de desempeño como parte permanente del ciclo de vida del software institucional.

5 Conclusiones

El desarrollo del presente proyecto, sustentado en la automatización de procesos de aseguramiento de calidad mediante la integración de Selenium, Docker y GitLab CI/CD, permitió validar un modelo funcional, escalable y alineado con los principios de ingeniería moderna del software. De forma complementaria, la incorporación de ESLint dentro del entorno de desarrollo institucional materializó el enfoque Shift-Left, trasladando las actividades de control y verificación de calidad hacia las fases iniciales del ciclo de vida del software. Esta estrategia garantizó la detección temprana de errores, la estandarización del código y la reducción de reprocesos, fortaleciendo la eficiencia y trazabilidad del proceso de desarrollo. Los resultados obtenidos en el capítulo anterior confirman que la aplicación práctica del paradigma DevOps, junto con la adopción de metodologías de integración continua, optimiza la estabilidad operativa del software institucional.

Consolidación de un modelo de automatización funcional y escalable: La implementación del prototipo validó la factibilidad técnica de un esquema de aseguramiento de calidad automatizado, basado en herramientas de código abierto y capaz de ejecutar pruebas de forma autónoma, repetible y controlada.

Optimización del proceso de aseguramiento de la calidad: la articulación de Selenium, Docker y GitLab CI/CD estandarizó el flujo de verificación, redujo la intervención manual y aportó trazabilidad en entornos controlados y reproducibles, en concordancia con los principios de integración y entrega continua.

Evidencia empírica de rendimiento y estabilidad: las métricas obtenidas confirmaron tiempos de carga y ejecución dentro de rangos óptimos, lo que respaldó la robustez del modelo bajo contenedores y ejecución continua.

Aplicación del principio Shift-Left en el ciclo de desarrollo: La configuración de ESLint en el entorno institucional desplazó los controles de calidad hacia etapas iniciales del desarrollo de software, promovió la detección temprana de

inconsistencias y malas prácticas y uniformó criterios técnicos desde la primera línea de código.

Impacto organizacional y sostenibilidad: La integración del modelo en el pipeline fortaleció la cultura DevOps, consolidó la colaboración entre desarrollo, calidad y operaciones y potenció prácticas de mejora continua, transparencia y comunicación efectiva. Gracias a su arquitectura modular y sustentada en estándares abiertos, el modelo resultó transferible a otros proyectos de la Cooperativa de Ahorro y Crédito Jardín Azuayo, evitando dependencias propietarias y asegurando escalabilidad, autonomía y sostenibilidad técnica a largo plazo.

6 Recomendaciones

Este apartado presenta orientaciones prácticas para consolidar un ecosistema de aseguramiento de la calidad y de trabajo DevOps en la Cooperativa de Ahorro y Crédito Jardín Azuayo. Las propuestas se enfocan en procesos, tecnología y personas, con el fin de fortalecer la trazabilidad, la seguridad y la entrega continua de software bajo criterios institucionales de calidad.

Se sugiere formalizar el aseguramiento de calidad con enfoque temprano (Shift-Left), estableciendo normas internas para que, antes de integrar o desplegar código, se ejecuten pruebas automatizadas y revisiones automáticas del código. Se fijan umbrales mínimos (por ejemplo, cobertura de pruebas y ausencia de errores críticos) que, de no cumplirse, impiden continuar. La Unidad de Desarrollo de Software y el área de Calidad asumen la gobernanza para mantener estándares y trazabilidad.

Se aconseja adoptar una estrategia de pruebas por niveles y una gestión ordenada de datos de prueba, priorizando pruebas unitarias e integración para detectar fallos tempranos y reservando un conjunto selecto de pruebas end-to-end con Selenium para lo crítico. Se preparan datos controlados y de autoservicio para el área de Aseguramiento de la Calidad (QA), y se gestiona la inestabilidad de pruebas mediante aislamiento temporal y análisis de causa raíz hasta su estabilización.

Se propone integrar la seguridad de forma continua a lo largo del ciclo de desarrollo, automatizando la revisión del código en busca de vulnerabilidades, la verificación de librerías externas y las pruebas dinámicas sobre entornos de prueba. Se documenta cada entrega con una lista de componentes del software, se firman los artefactos para garantizar su integridad y se resguardan secretos (claves y contraseñas) en almacenes seguros, aplicando el principio de mínimo privilegio en ejecutores e imágenes.

Se insta a elevar la observabilidad y el control operativo mediante métricas, instrumentando la aplicación para recoger trazas, métricas y registros en tableros

unificados. Se monitorean indicadores de entrega (tiempo de cambio, frecuencia de despliegue, tasa de fallos y tiempo de recuperación) y métricas de calidad (cobertura efectiva, estabilidad de pruebas y duración del pipeline). Se definen objetivos de servicio y se habilitan alertas y reversión automática cuando los indicadores se degraden.

Se recomienda modernizar la plataforma de la Cooperativa de Ahorro y Crédito Jardín Azuayo bajo un esquema de entornos (Desarrollo, Aseguramiento de la Calidad (QA), Certificación y Producción) gestionados como infraestructura declarativa y con despliegues progresivos. Describir los entornos con plantillas; habilitar ambientes por solicitud de cambio (merge request) para validar antes de fusionar y, cuando se requiera cobertura entre navegadores (cross-browser), operar un Selenium Grid en contenedores, aislado y con datos enmascarados. En Producción aplicar despliegues graduales con posibilidad de reversión (rollback).

Por último, se recomienda consolidar un ecosistema DevOps institucional en la Cooperativa de Ahorro y Crédito Jardín Azuayo mediante un pipeline centralizado en GitLab que orqueste construcción, pruebas y despliegue bajo un marco único de gobernanza. Definir estándares, Definition of Ready/Done, quality gates y políticas de cambio; habilitar runners, caché de dependencias y un repositorio de artefactos para acelerar ejecuciones y reutilizar resultados; y adoptar plantillas reutilizables (Dockerfiles, docker-compose y templates de GitLab CI). Además, impulsar un plan continuo de formación y comunidades de práctica en pruebas, seguridad y automatización.

7 Glosario

Automatización de Pruebas: proceso mediante el cual se emplean herramientas o scripts para ejecutar casos de prueba de manera repetitiva y controlada, reduciendo la intervención manual y aumentando la eficiencia del aseguramiento de calidad.

CI/CD (Integración y Entrega Continua): metodología de desarrollo que automatiza las fases de compilación, prueba e implementación del software, garantizando entregas frecuentes, seguras y verificables.

Contenedor: unidad estandarizada de software que agrupa código, dependencias y configuraciones, asegurando que una aplicación se ejecute de forma consistente en distintos entornos.

DevOps: conjunto de prácticas que integran desarrollo (Dev) y operaciones (Ops) para mejorar la colaboración, la automatización y la entrega continua del software.

Docker: plataforma de virtualización ligera basada en contenedores que permite desplegar aplicaciones junto con sus dependencias de forma aislada, portátil y reproducible.

ESLint: herramienta de análisis estático de código para JavaScript y React que identifica errores, malas prácticas y desviaciones del estilo definido, promoviendo la calidad desde las primeras fases del desarrollo.

GitLab Runner: agente de ejecución que procesa las tareas definidas en los pipelines de GitLab, permitiendo la ejecución automática de pruebas, compilaciones y despliegues.

GitLab CI/CD: sistema integrado en GitLab que automatiza la construcción, prueba y entrega del software mediante la definición de pipelines declarativos en el archivo .gitlab-ci.yml.

Left Shift (Shift-Left): principio de desarrollo que promueve trasladar las actividades de control de calidad hacia las etapas iniciales del ciclo de vida del software, detectando errores tempranamente y reduciendo costos de corrección.

Maven: herramienta de gestión y automatización de proyectos Java que permite compilar, probar y empaquetar aplicaciones mediante un modelo declarativo basado en el archivo pom.xml.

Pipeline: flujo automatizado que define la secuencia de etapas y tareas de construcción, prueba y despliegue de una aplicación dentro de un sistema CI/CD.

Selenium: framework de automatización de pruebas funcionales que permite simular la interacción de usuarios con navegadores web, validando el comportamiento de interfaces de usuario.

WebDriver: componente central de Selenium encargado de controlar el navegador web mediante comandos programáticos, permitiendo la ejecución de pruebas automatizadas.

WebDriverManager: librería que gestiona automáticamente las versiones de los controladores de navegador (drivers) utilizados por Selenium, evitando configuraciones manuales y asegurando compatibilidad entre versiones.

Docker Compose: herramienta de orquestación de Docker que permite definir y ejecutar múltiples contenedores de manera conjunta mediante un archivo docker-compose.yml.

JUnit: framework de pruebas unitarias en Java que permite estructurar, ejecutar y reportar resultados de test automatizados dentro del flujo de desarrollo.

Referencias

Apache Maven Project. (2023). *Introduction to the POM*. Obtenido de The Apache Software Foundation: https://maven.apache.org/pom.html

Basili, V., Caldiera, G., & Rombach, H. (2010). *The Goal Question Metric Paradigm Encyclopedia of software engineering.*

Bonigarcia. (2025). *WebDriverManager: Automate Browser Drivers [Software]*. Obtenido de https://bonigarcia.dev/webdrivermanager/

Calles-García, J., & González-Pérez, P. (2011). *La Biblia del Footprinting.*

CMMI Institute. (2018). *CMMI for Development, Version 2.0*. Obtenido de CMMI Institute: https://cmmiinstitute.com/cmmi

Cypress.io. (s.f.). *Cypress documentation*. Obtenido de https://www.cypress.io/

Docker. (2025). *Docker Documentation*. Obtenido de Dockerfile reference: https://docs.docker.com/build/concepts/dockerfile/

Docker Docs. (s.f.). *Docker Documentation*. Obtenido de https://docs.docker.com/compose/

Docker Docs. (s.f.). *Docker Documentation Build*. Obtenido de https://docs.docker.com/reference/compose-file/build/

Farley, H. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Boston, MA: Addison-Wesley Professional.

Fewster, M., & Graham, D. (1999). *Software test automation: Effective use of test execution tools.* Addison-Wesley.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

Gerard, M. (2007). *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley.

GitLab Documentation. (2025). *GitLab CI/CD pipelines*. Obtenido de GitLab: https://docs.gitlab.com/

GitLab Documentation Runner. (2025). *GitLab Runner installation on Windows*. Obtenido de https://docs.gitlab.com/runner/install/windows/

HeadSpin. (2024). *HeadSpin*. Obtenido de Cross-browser testing: What it is and how to do it effectively: https://www.headspin.io/blog/cross-browser-testing

Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional.

IEEE. (2014). *IEEE Standard for Systems and Software Engineering—Software Life Cycle Processes (IEEE Std 12207-2008).* IEEE.

IEEE Standards Association. (2021). *IEEE Standard for Software Quality Assurance Processes.* IEEE Std 730-2021.

International Organization for Standardization & International Electrotechnical Commission. (2017). *ISO/IEC/IEEE 12207:2017 Systems and software engineering — Software life cycle processes.* ISO / IEC / IEEE.

International Organization for Standardization. (2011). *ISO/IEC 25010:2011 – Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.* International Organization for Standardization.

International Software Testing Qualifications Board. (2020). *ISTQB Certified Tester Foundation Level Syllabus.* ISTQB.
JUnit Project. (2025). *JUnit 5 User Guide [Documentación]*. Obtenido de JUnit: https://junit.org/junit5/
Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* IT Revolution Press.
Merkel, D. (2014). Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 239.
Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley.
MojoHaus. (s.f.). *Exec Maven Plugin*. Obtenido de https://www.mojohaus.org/exec-maven-plugin/plugin-info.html
OpenJS Foundation. (2025). *ESLint*. Obtenido de OpenJS Foundation: https://eslint.org/
PCI Security Standards Council. (2022). *Payment Card Industry Data Security Standard (PCI DSS) version 4.0.* PCI SSC.
Playwright.dev. (s.f.). *Playwright.dev*. Obtenido de Best practices: https://playwright.dev/docs/best-practices
Pressman, R. S.; Maxim, B. R.;. (2015). *Ingeniería de software: Un enfoque práctico (8.ª ed.).* McGraw-Hill.
Pressman, R., & Maxim, B. (2020). *Software Engineering: A Practitioner's Approach.* McGraw-Hill Education.
Pressman, R., & Maxim, B. (2021). *Ingeniería de Software: Un enfoque práctico.* McGraw-Hill.
Rauschmayer, A. (2019). *Deep JavaScript: Theory and techniques*. Obtenido de Exploring JS: https://exploringjs.com/deep-js/
Red Hat. (2023). *Shift left vs. shift right*. Obtenido de https://www.redhat.com/en/topics/devops/shift-left-vs-shift-right
Roadmap.sh. (s.f.). *Why does DevOps recommend shift-left testing*. Obtenido de https://roadmap.sh/devops/shift-left-testing
Selenium Project. (2025). *Selenium WebDriver*. Obtenido de https://www.selenium.dev/
Selenium Project. (2025). *Selenium WebDriver [Software]*. Obtenido de Selenium: https://www.selenium.dev/
Sommerville, I. (2016). *Ingeniería de software.* Pearson Educación.
Sonatype. (2008). *Maven: The definitive guide.* O'Reilly Media.
Wang, X., Ali, N., & Petersen, K. (2020). Static analysis tools and their impact on software quality: A systematic literature review. *Journal of Systems and Software*, 1 - 21.
www.elhacker.net. (s.f.). *www.elhacker.net*. Obtenido de https://www.elhacker.net/trucos_google.html
(Docker Docs, s.f.)