Russell: Herramienta para la gestión y administración de pipelines en flujos de trabajo CI/CD

Jairo Francisco Giraldo Tejeiro
Carlos Andrés Hernández Vega

Director(a): MSc. Jose Alejandro Benitez Aragon

Pontificia Universidad Javeriana Cali
Facultad de Ingeniería.
Maestría en Ingeniería de Software
Proyecto de Grado.
25 de Julio de 2025

Santiago de Cali, 2 de octubre de 2025.

Señores
Pontificia Universidad Javeriana Cali.
Ph.D. Luisa Rincón
Directora Maestría en Ingeniería de Software.
Cali.

Cordial Saludo.

Por medio de la presente hago constar que en mi calidad de director de trabajo de grado he revisado el proyecto titulado "Russell: Herramienta para la gestión y administración de pipelines en flujos de trabajo CI/CD" realizado por el estudiante de Magister en Ingeniería de Software Jairo Francisco Giraldo Tejeiro (cod: 9010850), el cual se encuentra terminado y considero que cumple con los requisitos para ser sustentado.
Atentamente,

MSc. Jose Alejandro Benitez Aragon

Santiago de Cali, 2 de octubre de 2025.

Señores
Pontificia Universidad Javeriana Cali
Ph.D. Luisa Rincón
Directora Maestría en Ingeniería de Software
Cali.

Cordial Saludo.

Nos permitimos presentar a su consideración el proyecto de grado titulado "Russell: Herramienta para la gestión y administración de pipelines en flujos de trabajo CI/CD" con el fin de cumplir con los requisitos exigidos por la Universidad y para que sea sometido a revisión del jurado y cumpla su aprobación, para conseguir posteriormente el título de Magister en Ingeniería de Software.

Atentamente,

Francisco Giraldo

Jairo Francisco Giraldo Tejeiro
Código: 9010850

Carlos Andrés Hernández Vega
Código: 9010788

Ficha Resumen
Trabajo de Grado Maestría en Ingeniería de Software

TÍTULO: Russell: Herramienta para la gestión y administración de pipelines en flujos de trabajo CI/CD

1. Énfasis: Ingeniería de Software
2. Área de trabajo: Computación en la nube
3. Tipo de proyecto (Aplicado, Innovación, Investigación): Aplicado
4. Estudiante: Jairo Francisco Giraldo Tejeiro - 9010850, Carlos Andrés Hernández Vega - 9010788
5. Correo electrónico: franciscogt@javerianacali.edu.co - cahernandezv@javerianacali.edu.co
6. Dirección y teléfono: Bogotá, 300 264 5458 - 310 787 8667
7. Director: MSc. Jose Alejandro Benitez Aragon
8. Vinculación del director: Cátedra
9. Correo electrónico del director: jose.benitez@javerianacali.edu.co
10. Co-Director (Si aplica):
11. Grupo o empresa que lo avala (Si aplica):
12. Otros grupos o empresas:
13. Palabras clave(al menos 5): DevOps, pipeline, integración, entrega, automatización.
14. ODS que aplica al proyecto (Agenda 2030): ???
15. Fecha de inicio: 01/02/2025
16. Resumen: El presente trabajo se enfoca en el desarrollo de una herramienta que automatiza la gestión y ejecución de pipelines dentro de un entorno de integración y entrega continua (CI/CD). Su propósito principal es optimizar el flujo de trabajo de desarrollo de software, proporcionando una solución eficiente para la orquestación de tareas como la compilación, pruebas, despliegue y monitoreo de aplicaciones para empresas que apenas están adoptado este modelo de trabajo ágil. Además, se enfoca en mejorar la productividad, reduciendo errores manuales y acelerando la entrega de productos de software.

Agradecimientos

Queremos comenzar agradeciendo a nuestras familias, por su amor, paciencia y apoyo constante a lo largo de este proceso. Su acompañamiento fue fundamental para mantenernos motivados y enfocados en nuestro objetivo.

Agradecemos sinceramente al profesor Alejandro, director de este trabajo de grado, por su guía, compromiso y orientación durante todas las etapas del trabajo. Sus aportes fueron clave para estructurar y consolidar este proyecto.

También extendemos nuestro agradecimiento a la profesora Luisa, directora de la Maestría en Ingeniería de Software, por su apoyo, disposición y por promover un entorno académico que nos permitió crecer tanto personal como profesionalmente.

Finalmente, nos agradecemos mutuamente por el compromiso, la dedicación y el trabajo en equipo que hicieron posible la realización de esta tesis. Este logro es el reflejo del esfuerzo compartido y del aprendizaje que construimos juntos a lo largo de este camino.

Resumen

Este proyecto, se enfoca en el desarrollo de una herramienta que automatiza la gestión y ejecución de pipelines dentro de un entorno de integración y entrega continua (CI/CD). Su propósito principal es optimizar el flujo de trabajo de desarrollo de software, proporcionando una solución eficiente para la orquestación de tareas como la compilación, pruebas, despliegue y monitoreo de aplicaciones para empresas que apenas están adoptado este modelo de trabajo ágil. Además, se enfoca en mejorar la productividad, reduciendo errores manuales y acelerando la entrega de productos de software.

Palabras Clave: DevOps, pipeline, integración continua, entrega continua, automatización.

Abstract

This project focuses on the development of a tool that automates the management and execution of pipelines within a Continuous Integration and Continuous Delivery (CI/CD) environment. Its main purpose is to optimize the software development workflow by providing an efficient solution for orchestrating tasks such as building, testing, deploying, and monitoring applications for companies that are just adopting this agile working model. Additionally, it aims to enhance productivity by reducing manual errors and speeding up the delivery of software products.

Keywords: DevOps, pipeline, continuous integration, continuous delivery, automation.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En la última década, el uso de herramientas de integración y entrega continua (CI/CD) ha revolucionado la manera en que se desarrollan y despliegan aplicaciones de software. El rendimiento de la entrega de software tiene amplios impactos, no solo en la perspectiva del producto, sino también en el rendimiento organizacional e individual (Forsgren et al., 2018).

El software no genera ingresos hasta que está en manos de sus usuarios -es una realidad inherente del sector- pero en la mayoría de las organizaciones, la puesta en producción de software es un proceso que requiere una gran cantidad de mano de obra, casi podríamos asegurar que es artesanal en la mayoría de los casos, convirtiendo este proceso en algo propenso a errores y que conlleva en sí mismo demasiados riesgos. Para las grandes empresas, cada semana de demora entre tener una idea y publicar el código que la implementa puede representar millones de dólares en costos de oportunidad, y sin embargo, estas suelen ser las que tienen los tiempos de ciclo más largos Farley (2023). A pesar de todo esto, los mecanismos y procesos que permiten la entrega de software con bajo riesgo no se han convertido en parte de la estructura de los proyectos de desarrollo de software actuales.

Desde un punto de vista de eficiencia, la gestión de pipelines sigue siendo un desafío, casi que un arte, especialmente para equipos pequeños que carecen de especialistas dedicados a la administración de estas herramientas y los altos costos de puesta en marcha. El mercado laboral actualmente cuenta con un gran número de ingenieros, número que se ve reducido a los que se enfocan en la rama del desarrollo de software, y es aún más bajo si lo se requieres es talento con alto expertise, un artesano, un experto en adoptar, evangeliza e implementar modelos que mejores resultados han dado. Este anteproyecto propone el desarrollo de una solución que simplifique la administración de pipelines de integración y entrega continua, permitiendo una gestión más intuitiva y automatizada.

1.1. Definición del problema

1.1.1. Planteamiento del problema

El mundo del desarrollo de software se ha visto influenciado por las grandes manufactureras del mercado, que en un afán por hacer filantropía han dado a conocer la forma en que trabajan y se han convertido en casos de éxito, siendo referentes para todo aquel que desee incursionar en este nicho. La liberación de estas buenas prácticas han cambiado la mentalidad en los equipos de trabajos que en años anteriores con otras metodologías o frameworks siempre han buscado que sus proyectos

cumplan en tiempo, costo, alcance y calidad, además de lograr la satisfacción de las partes interesadas. En el informe de resultados de la encuesta global (Australia, 2019) que presentaron KPMG, AIMP e IPMA, en la que participaron aproximadamente 500 directores de proyectos de 57 países se evidenció que el 19 % de las organizaciones entregan proyectos de manera exitosa la mayoría de las veces, es probable que el 44 % cumpla con el alcance definido y la intención comercial, el 30 % entrega a tiempo y el 36 % lo hace bajo el presupuesto.

La gestión de proyectos de software en Colombia en busca de mitigar los problemas más comunes y mejorar la eficiencia en el proceso de desarrollo de los proyectos de software, adoptaron algunas de estas practicas, en donde no solo trajo buenos beneficios al reducir los tiempos de entrega a los stakeholders, sino que creó la necesidad implícita de aumentar la cantidad de despliegues en ambientes de producción, entregando valor constante a los usuarios finales desde las primeras etapas del proyecto para así probar hipótesis, tomar decisiones y realizar ajustes de forma objetiva, tomando como entrada de información los datos recolectados de las interacciones de clientes reales versus las funcionalidades entregadas. Esta forma de afrontar los proyectos especialmente donde han adoptado metodologías ágiles generó una brecha entre los equipos de desarrollo y operaciones IT. DevOps como cultura surgió para contribuir en mejorar la comunicación y la colaboración entre los dos equipos.

Los recurrentes despliegues en ambientes productivos trajeron consigo la inclusión de un mayor número de procesos manuales para satisfacer una nueva necesidad tan vital en las organizaciones para mantenerse vigentes en el mercado. El proceso manual -en general- es propenso a incluir errores; los componentes de software como productos tienen embebido este riesgo, que es latente en cada una de las salidas a producción. El despliegue no es el único punto susceptible de adicionar errores por procesos manuales; el empalme natural que nace cuando el equipo de desarrollo entrega su componente al equipo de operaciones para hacer su trabajo en la ventana de despliegue puede conllevar la pérdida de detalles en la instalación y configuración promovida.

Patrick Debois define DevOps como un enfoque que promueve la convergencia entre los equipos de desarrollo (Dev) y operaciones (Ops) (Debois et al., 2021). Su propósito es mejorar la eficiencia en el desarrollo de software, facilitar la entrega continua y garantizar la calidad y estabilidad del producto, mitigando la introducción de errores humanos. Otros aspectos valiosos de DevOps incluyen la promoción de culturas de gestión, el liderazgo orientado al servicio y la gestión del cambio organizacional.

Esta cultura ha crecido en las diferentes comunidades y de forma organizada han aparecido herramientas que apalancan la automatización de los procesos que están bajo el resorte de responsabilidades del área de operaciones, el auge de DevOps ha creado nuevos roles, oportunidades y desafíos, que la comunidad ha ido enfrentando con herramientas que ayudan a construir un proceso robusto a lo largo de las fases de ciclo de vida del software, clasificándose en diferentes categorías según su uso:

1. Gestión de la Configuración e Infraestructura como Código.
2. Integración Continua / Entrega Continua (CI/CD).
3. Gestión de Versiones y Control de Código fuente.
4. Contenedores y Orquestación.
5. Monitorización y Logging.
6. Gestión de Incidentes y ChatOps.
7. Automatización de Pruebas.
8. Gestión de Lanzamientos.
9. Gestión de Proyectos y Colaboración.
10. Seguridad y Compliance (DevSecOps).

Aunque todas son relevantes dentro del proceso de gestión de la cultura devops, las categorías que se encuentra directamente relacionadas al despliegue son: 1., 2., 3., 4., 7. y 10. En gestión de la configuración e infraestructura como código estas herramientas apoyan la construcción y aprovisonamiento de la infraestructura bien sea en Cloud o ambientes On Premise, garantizando la consistencia y el escalado de los recursos requeridos en los despliegues, HashiCorp Terraform -cuentan con certificación propia para avalar a una persona experta en su uso- permite a los desarrolladores crear, modificar y guardar versiones de la infraestructura gracias a su soporte multi-cloud y su amplia cantidad de recursos soportados en especial con Amazon AWS. Amazon AWS es la plataforma de computación en la nube que proporciona servicios de infraestructura, como almacenamiento, procesamiento, bases de datos y servicios avanzados (IA, IoT, etc), ampliamente utilizada en el mundo por empresas de todos los tamaños.

En la integración y entrega continua, encontramos herramientas muy poderosas como Jenkins, Github Actions, TravisCI, CircleCI, entre otras, estás ofrecen una ruta "fácil"para automatizar la construcción, prueba y despliegues del software, cada una tiene su propia biblioteca de plugins que amplifican el alcance de misma, a continuación se presenta una Tabla 1.1. comparativa de las más populares.

Aunque las herramientas mencionadas en la Tabla 1.1 tienen una capa gratuita, esta solo aplica para proyectos de código abierto. Sin embargo, para proyectos privados o cuando se consumen los recursos asignados de la capa gratuita, es necesario adquirir un plan de pago. Los planes pagos dependen de la cantidad de ejecuciones, tiempo de ejecución o los recursos que se necesiten, lo que generará costos adicionales de acuerdo a la frecuencia de los despliegues y tamaño de los mismos. A hoy la ausencia de una herramienta que gestione y administre de forma controlada los pipelines de promoción de funcionalidades en un entorno de desarrollo y despliegue continuo por los diferentes ambientes introduce riesgos materializables significativos en términos costos.

Características	**GitHub Actions**	**Bamboo**	**CircleCI**	**Azure Pipelines**	**AWS CodePipeline**	**Travis CI**	**Jenkins**
Plan Básico	Gratis para repositorios públicos, $4/mes por usuario para repositorios privados	$10 por mes	Plan gratuito disponible, planes de pago desde $15/mes	Gratis para hasta 5 usuarios, $6 por usuario/mes para más usuarios	Incluido en el uso de AWS	Gratis para código abierto, planes de pago desde $69/mes	Gratis
Minutos adicionales de build	$0.008 por minuto	Ilimitado	$0.006 por minuto	$0.002 por minuto	$0.0025 por ejecución de pipeline	$0.018 por minuto	N/A
Almacenamiento adicional	$0.25 por GB	Ilimitado	$0.20 por GB	$0.1 por GB	Costo adicional por Amazon S3	$0.1 por GB	N/A
Plataformas	Todas	Todas	Todas	Todas	Todas	Todas	Todas
API	Si	Si	Si	Si	Si	Si	Si
Entrenamiento	Alto	Alto	Alto	Alto	Alto	Alto	Alto

Tabla 1.1: Planes de servicios de integración y entrega continua. Valores expresados en $USD. Fuente propia.

Cada componente de software que ha existido, existe y existirá, sí o sí tiene asociado a el un listado de instrucción que definen su funcionamiento y comportamiento, lo conocemos como código fuente, y debe ser almacenado, protegido y versionado. En esta categoría de gestión de versiones y control de código fuente contamos con herramientas como Git, Github, GitLab, Bitbucket, que hacen la veces de repositorios de código facilitan la colaboración entre miembros del equipo y otros equipos, exponen APIs propias para ser integradas con procesos de CI/CD. Dichos han ido evolucionando con su adopción en la nube y se han vuelto auto-contenidos apoyados en contenedores y orquestación, la tecnología de gestión de contenedores permite crear, gestionar y orquestar piezas más pequeñas facilitando la escalabilidad y despliegue veloz de los mismos, aquí destacan herramientas como Docker, Kubernetes, OpenShift que son las más usadas por la comunidad.

El código evoluciona al igual que los negocios y un pilar de los despliegues es la automatización de pruebas, la calidad -que no se negocia- y la automatización son aspectos fundamentales en el desarrollo de software moderno, mientras la calidad de software comprueba que se cumplan los requisitos, tanto explícitos como implícitos, la automatización nos provee la plataforma para realizar una y otra vez de manera automática y sin intervención manual, garantizando la fiabilidad del incremento a liberar, las herramientas a nombrar son: Selenium, JUnit, TestNG, Mocha, Jest, Cucumber.

Como última categoría a mencionar debemos enfocarnos en la Seguridad y Compliance, así como la calidad no se negocia, la seguridad tampoco se nos puede quedar atrás, integrar la seguridad en el ciclo de vida del desarrollo y más aún en el de despliegue continuos previene de tener vulnerabilidades explotables en los componentes, código fuente, reduce los riesgos en producción, minimiza el retrabajo y acelera el ciclo de entrega.

Otro de los desafíos que tienen las organizaciones en su adopción, es que muchos de sus empleados carecen de conocimiento para trabajar en un entorno DevOps, lo que dificulta el desarrollo efectivo de prácticas de integración y entrega continua. Como medida las organizaciones podrían optar contratar un experto en el manejo de estas herramientas, pero según un estudio de remuneración tecnológica 2024 PageGroup (2024) realizado por Michael Page, líder en reclutamiento y selección, un profesional en esta área de conocimiento su asignación salarial es bastante elevada y

varia según años de experiencia como se observa en la Tabla 1.2.

Años de experiencia	Salario Bruto Min	Salario Bruto Max	Promedio
3 a 5	$12.000.000	$16.000.000	$14.000.000
5 a 8	$16.000.000	$25.000.000	$20.500.000
8 a 12	$20.000.000	$25.000.000	$22.500.000

Tabla 1.2: Asignación salarial de un profesional DevOps del mercado en Colombia y expresados en $COP. Fuente: (PageGroup, 2024)

Esta convergencia de tantos frentes a los cuales debe prestar atención una empresa que esta incursionando en el mundo del desarrollo de software o que ya esta posicionada pero no ha dedicado espacios para reducir esas brechas, está en una clara desventaja con respecto a los competidores que ya han dado el salto. Sin un sistema automatizado, los procesos de integración, pruebas y despliegues dependerían de tareas manuales, lo que impactaría negativamente los tiempos de entrega y aumentaría el riesgo de errores humanos, afectando directamente la estabilidad del sistema o producto de software.

1.1.2. Formulación del problema

En vista de lo anteriormente descrito y comprendiendo la necesidad, cada día más creciente, de adoptar estructuras más ágiles al interior de las áreas de proyectos de desarrollo de software para reducir el time to market surgen las siguientes interrogantes:

- ¿Cómo diseñar y desarrollar una solución de software que facilite a las organizaciones la adopción de la cultura DevOps en sus procesos de integración y entrega continua?.
- ¿Qué podemos hacer para reducir el alto costo operativo de contratar expertos en configuración de las principales herramientas destinadas a la integración y entrega continua de productos de software?.
- ¿Qué herramientas y tecnologías existen para la integración y entrega continua de productos de software?.
- ¿Cómo diseñar una arquitectura de despliegue para identificar las funcionalidades que debe incluir el gestor de pipelines?.
- ¿Qué consideraciones de seguridad se deben tener para aplicar un control de acceso basado en roles que permita la protección de la configuración de los pipelines?.

1.2. Objetivos del proyecto

1.2.1. Objetivo General

Diseñar y desarrollar una herramienta que permita gestionar y administrar pipelines orientada a la rápida configuración de flujos de trabajo en integración y entrega continua.

1.2.2. Objetivos específicos

1. Identificar las prácticas y estándares en la implementación de pipelines de integración y entrega continua.
2. Definir las herramientas y tecnologías disponibles que se puedan integrar al gestor de pipelines de integración y entrega continua.
3. Diseñar una arquitectura de despliegue para identificar las funcionalidades que debe incluir el gestor de pipelines.
4. Implementar un sistema de control de acceso basado en roles que permita la protección de la configuración de los pipelines.
5. Diseñar la interfaz gráfica del gestor de pipelines para que sea multidioma.

1.3. Delimitaciones y alcances

El presente trabajo tuvo como objetivo diseñar y desarrollar un gestor de pipelines orientado a la rápida configuración de flujos de trabajo en integración y entrega continua.

1.3.1. Entregables

- El frontend incluyó los módulos de administración de usuarios, dashboard de administración de pipelines configurados y clasificador de pipelines soportados.
- Se implementaron APIs serverless en Node v20 o superior.

1.3.2. Restricciones y Exclusiones

- El gestor de pipelines no despliega software ni componentes en infraestructura OnPremise.
- Para integración continua / entrega continua se empleó Jenkins.
- El gestor de versiones y control de código fuente fue GitHub.
- El gestor de contenedores utilizado fue Podman.
- En cuanto al mantenimiento, no se consideró, ya que no estuvo contemplado en el alcance del proyecto dentro del contexto académico.

- El administrador de pipelines solo tuvo soporte para navegadores Chrome 125 o superior.

1.4. Justificación del trabajo de grado

Los desarrolladores se enfocaban en escribir código y crear nuevas características, mientras que los equipos de operaciones se encargaban de implementar y administrar el software en entornos de producción. Esta se separación a menudo generaba brechas en comunicación, ciclos de lanzamiento lentos y procesos ineficientes Smith et al. (2023).

DevOps cierra la brecha entre el desarrollo y las operaciones al promover una cultura de colaboración, responsabilidades compartidas y retroalimentación continua mediante la automatización a lo largo del ciclo de vida del desarrollo de software Smith et al. (2023).

Para comprender mejor el potencial de DevOps, en el año 2000, se logro mejorar los tiempos de desarrollo de nuevas funcionalidades a semanas o meses, esto se dio gracias los avances tecnológicos y la adopción de principios y prácticas ágiles. Pero los despliegues a el ambiente productivo seguían siendo complejos y con tiempos largos donde en algunas ocasiones los resultados eran desastrosos. Luego, 10 años después, con la introducción de DevOps y las soluciones en nube, los despliegues a producción fueron mayores y transformaron significativamente la forma en que las organizaciones entregaban software; al mejorar esta complejidad en sus procesos de despliegue, se aumentaba la estabilidad y confiabilidad del software en producción. DevOps permite realizar implementaciones con más frecuencia, reduciendo así la capacidad de respuesta a las necesidades del mercado. Esto se logra mediante la automatización de procesos importantes, como lo es la integración continua (CI) y la entrega continua (CD) Smith et al. (2023). A continuación, en la Tabla 1.3, se presenta cómo las tecnologías han evolucionado para reducir el riesgo y el costo de desarrollo de software, permitiendo ciclos más rápido y de mayor agilidad para adaptarse a las necesidades del mercado.

DevOps nos demuestra que, con una arquitectura adecuada, prácticas técnicas sólidas y una cultura organizacional alineada, los equipos pequeños de desarrolladores pueden realizar cambios en producción de forma ágil, segura y autónoma, abarcando todas las etapas desde el desarrollo hasta la integración, pruebas e implementación; logrando mejorar el Time to Market, ya que un número de Time to Market más alto sugiere que las características se están desarrollando, probando y entregando con mayor lentitud y menor frecuencia. La mayoría de los programas y servicios de software actuales requieren actualizaciones más constantes.

El Informe sobre el estado de DevOps de 2015 no solo examinó las “implementaciones por día”, sino también las “implementaciones por día por desarrollador”. Plantearon la hipótesis de que las empresas de alto rendimiento podrían aumentar la cantidad de implementaciones a medida que aumentaba el tamaño del equipo Nicole Forsgren (2015).

La Figura 1.1 muestra que en las empresas de bajo rendimiento, las implementaciones por día por desarrollador disminuyen a medida que aumenta el tamaño del equipo, se mantienen constantes en las de rendimiento medio y aumentan linealmente en las de alto rendimiento. En otras palabras, las organizaciones que adoptan DevOps en sus procesos de integración y entrega continua pueden aumentar linealmente la cantidad de implementaciones por día a medida que aumentan la cantidad de desarrolladores, tal como lo han hecho Google, Amazon y Netflix.

Description	1970s–1980s	1990s	2000s–Present
Era	Mainframes	Client/Server	Commoditization and Cloud
Representative technology of era	COBOL, DB2 on MVS, etc.	C++, Oracle, Solaris, etc.	Java, MySQL, Red Hat, Ruby on Rails, PHP, etc.
Cycle time	1–5 years	3–12 month	2–12 weeks
Cost	$1M–$100M	$100k–$10M	$10k–$1M
At risk	The whole company	A product line or division	A product feature
Cost of failure	Bankruptcy, sell the company, massive layoffs	Revenue miss, CIO's job	Negligible

Tabla 1.3: La tendencia cada vez más acelerada hacia una entrega de software más rápida, más económica y con menor riesgo. Valores expresados en $USD. Fuente: (Cockcroft, 2013).

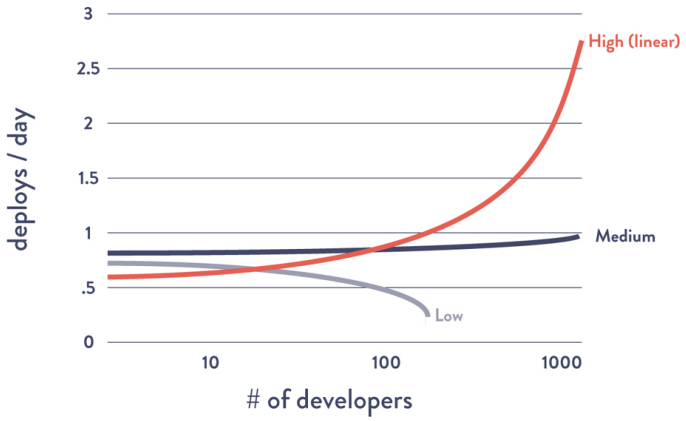


Figura 1.1: Implementaciones por día vs. Número de desarrolladores Fuente: (Nicole Forsgren, 2015).

Con el apoyo de una herramienta para la gestión y administración de pipelines, la entrega de valor a sus clientes sería más constante y rápida versus un modelo que basa su estrategia de despliegue en procesos manuales y operativos que afectan directamente el revenue al hacer sus tiempos de entrega más lentos en comparación con la demanda del mercado. Como se ha mencionado anteriormente, en el mercado actual existen herramientas que ayudan a la adopción de DevOps en sus procesos de integración y entrega continua ver Tabla 1.1, solo algunas empresas,

para ser competitivas están en la capacidad y tiene el suficiente músculo financiero para soportar el alto costo en el cual se debe incurrir para cubrir la creciente necesidad de incorporarlas dentro de sus procesos, es por esto que el presente proyecto se centra en brindar una solución eficiente y rentable frente a la dificultad y reto que representan incluir los procesos de integración y entrega continua su cultura organizacional.

1.5. Metodología de la investigación

1.5.1. Metodología

En el desarrollo del proyecto objeto de este documento se utilizará la metodología en espiral porque es un modelo enfocado en desarrollo de software en el que se describirá el ciclo de vida de la construcción, además cubre las mejores características de modelos metodológicos como el ciclo de vida en cascada y construcción de prototipos, pero añadiendo un nuevo elemento: el análisis de riesgos. Esta característica permite al desarrollador y al cliente entender y reaccionar a los riesgos en cada nivel evolutivo.

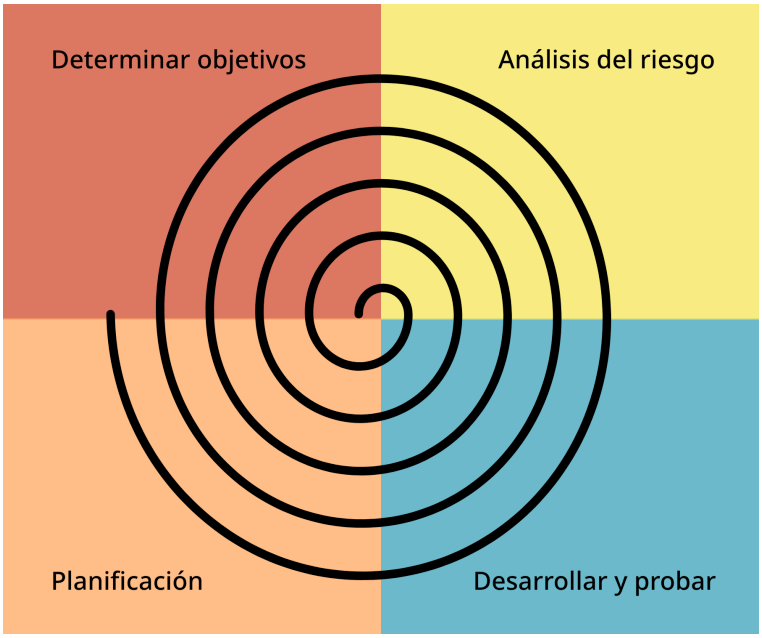


Figura 1.2: Modelo espiral de Boehm. Fuente: (Boehm et al., 2014).

El modelo en espiral se representa mediante una espiral ver Figura 1.2, la cual define cuatro actividades principales representadas por los cuatro cuadrantes:

- Determinar objetivos.
- Análisis del riesgo.

- Desarrollo y pruebas.
- Planificación.

Durante la primera vuelta alrededor de la espiral se definen los objetivos, las alternativas y las restricciones. Cuando el objetivo es claro para la iteración, se analizan, identifican y evalúan los riesgos potenciales, aplicando la misma lógica a las alternativas. Los riesgos son registrados y reducidos utilizando prototipos y/o simulaciones.

En la fase de desarrollo se genera el código de los componentes del sistema que constituyen los entregables (base de datos, página Web, API, infraestructura, pipelines a la medida y producto final). Una vez el código es generado, el sistema se ejecuta de manera experimental para asegurarse que no presenta fallas, es decir que funciona de acuerdo con las especificaciones y en la forma en que se espera, para ello se alimenta con entradas conjuntas de datos de prueba para su procesamiento y luego se examinan los resultados.

El asesor de proyectos, evalúa el trabajo de ingeniería (desarrollo y pruebas) sugiere modificaciones y sobre la base del feedback recibido se produce la documentación e inicia fase de planificación de la siguiente iteración pero marcando el final de la iteración actual.

En cada iteración del modelo en espiral, se realiza un proceso de aceptación de los módulos del sistema por parte del asesor de proyectos, con el fin de asegurar que el producto final cumple con todos los requerimientos establecidos. Este proceso tiene como objetivo principal, la entrega y aceptación del sistema en su totalidad, que puede comprender varios componentes de software desarrollados de manera independiente, según se haya establecido.

Dada la dinámica laboral y los espacios en agenda disponibles para el desarrollo de este trabajo se considera que una metodología ágil -de las que están tan de moda hoy en día- aplicaría a la perfección como marco de trabajo para abordar la problemática de este documento, se busca con esto que la metodología este alineada con la solución propuesta. Dentro de las metodologías evaluadas Kanban es la que más se ajusta a las necesidades y modelo de trabajo siempre entregando valor de manera eficiente.

1.5.2. Descripción kanban

Kanban es un marco popular que se utiliza para implementar el desarrollo de software Agile y DevOps. Requiere comunicación en tiempo real de la capacidad y total transparencia del trabajo. Los elementos de trabajo se representan visualmente en un tablero Kanban, lo que permite a los miembros del equipo ver el estado de cada pieza de trabajo en cualquier momento (Atlassian, sfb).

- **Visualización del flujo de trabajo:** En un tablero Kanban, con el uso de tarjetas de tareas Kanban, utilizando columnas y filas que representan diferentes etapas y aspectos del proceso. Gracias a la visualización de todas las tareas, cada miembro del equipo sabe qué está sucediendo en cada etapa del proceso, también quién es la persona adecuada a la que hacer qué pregunta y quién será el mejor candidato para un determinado tipo de trabajo.

También permite una gran reducción del tiempo dedicado a discutir el trabajo en lugar de hacerlo (Majowska, 2014).

- **Límite de trabajo en curso:** Permitido por columna (por etapa del proceso) o por usuario (de modo que cada miembro del equipo solo pueda trabajar en una cierta cantidad de tareas en un momento dado). Al aplicar límites de Trabajo en progreso, se esta garantizando que la concentración y la dedicación del equipo prosperen, ya que las personas pueden simplemente hacer una cosa a la vez, sin la necesidad de realizar varias tareas a la vez, lo que siempre disminuye la productividad (Majowska, 2014).
- **Medición y optimización del flujo:** Se monitorizará el tiempo de ciclo (tiempo que tarda una tarea desde que se comienza hasta que se finaliza), y se buscará optimizar continuamente el flujo de trabajo. La identificación de cuellos de botella permitirá realizar ajustes que mejoren la productividad y la entrega de resultados (Majowska, 2014).

Capítulo 2

Marco de referencia

A continuación se presentan los fundamentos conceptuales, tecnológicos y antecedentes relevantes que sustentan el desarrollo del administrador de pipelines propuesto, desde el contexto de las prácticas de Integración Continua y Despliegue Continuo (CI/CD), y se definen algunos conceptos clave relacionados con la automatización de flujos de trabajo en el ciclo de vida del software.

También, se revisan las principales herramientas existentes en el Asimismo, se revisan las principales herramientas existentes, sus componentes funcionales y sus limitaciones.

2.1. Marco Teórico

2.1.1. Bases Teóricas

- **DevOps:** En la actualidad se define como una metodología de ingeniería de software con el único objetivo de integrar todo el trabajo completado por los equipos de desarrollo y los equipos de operaciones, logrando implícitamente una cultura de colaboración y responsabilidad compartida GitLab (2024).

  Para representar DevOps se usa un ciclo infinito para mostrar cada una de sus fases del ciclo de vida, las cuales están relacionadas entre sí. Como se observa en el ciclo de la Figura 2.1 muestra la necesidad de colaboración continua y mejora iterativa durante todo el ciclo de vida.

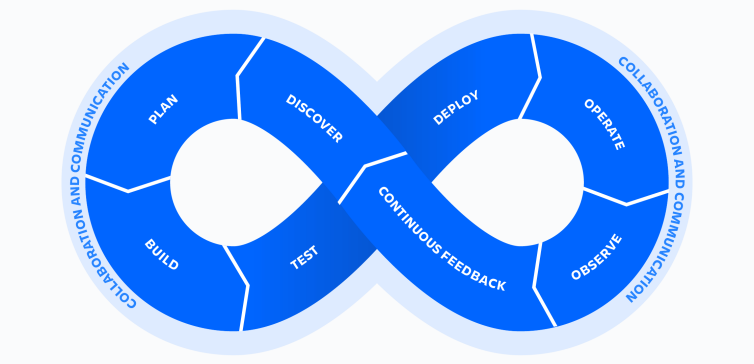


Figura 2.1: Ciclo de vida DevOps. Fuente: (Atlassian, sfa).

- **Integración Continua (CI):** La integración continua se basa en el hecho de que el código de la aplicación se almacena en un sistema de gestión de control de fuente (SCM). Un cambio en el

código desencadena una serie de eventos entre los que se encuentra el proceso de compilación automática que produce un artefacto de incremento producto de la evolución propia del sistema, el código por otro lado es almacenado en un repositorio central y de acceso controlado. Este proceso es repetible de forma tal que cada vez que se ejecuta la compilación a partir del mismo código, se espera el mismo resultado, demostrando consistencia e idempotencia. El proceso de compilación se lleva a cabo en una máquina especialmente construida y configurada para realizar esta tarea, se conoce como el servidor de integración o de compilación van Merode (2023).

La integración continua nos permite detectar y solucionar los problemas que puedan generarse antes y después de la integración lo antes posible en el ciclo. Eso lo podemos visualizar en la Figura 2.2, la cual muestra algunos de los problemas que pueden generarse durante un ciclo de CI.

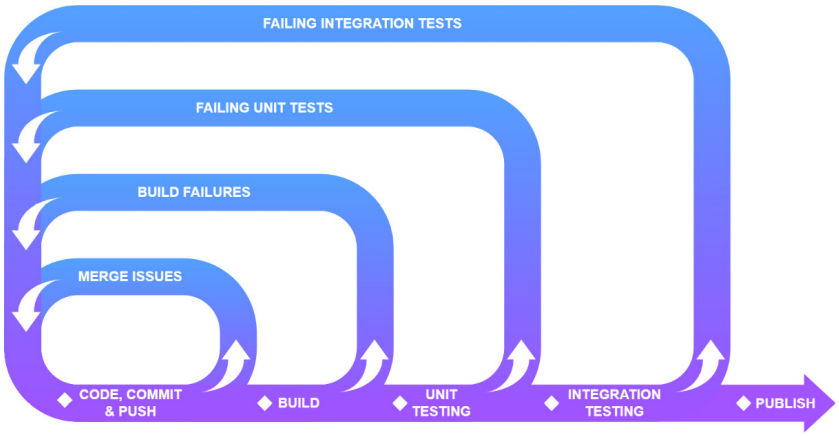


Figura 2.2: Procesos de CI. Fuente: (Pathania, 2024).

Principios clave en la integración continua:

1. **Pull con frecuencia:** En el contexto de CI, es bajar cambios de la rama principal del código a un rama local, luego de hacer los cambios en la rama, estos se deben publicar nuevamente hacia la principal. Es importante publicar de manera frecuente los cambios, ahí es donde esta la cultura, ya que si un desarrollador se demora dos semanas, ha retrasado la integración y ha roto los principios. Si esa compilación falla, los miembros del equipo tienen que analizar dos semanas de trabajo para ver donde se ha dañado el proyecto.
2. **Validar cada cambio de código:** Es importante para validar problemas de compilación, primero se debe realizar de manera local y luego cuando se envían los cambios al repositorio remoto compila en el servidor. De esta manera se garantiza que cada confirmación se valide para poder ver que impacto tendrá en el proyecto.
3. **Automatización:** Todo lo que se pueda automatizar es clave, automatizar acelera todo. Es importante automatizar: pruebas, lanzamientos, cambios de configuración y más.

4. **Retroalimentación continua:** La entrega de nuevas características y actualizaciones fluye de izquierda a derecha como se muestra en la Figura 2.3. El extremo izquierdo de la cadena de producción es donde se lleva a cabo el desarrollo y el extremo derecho es donde la aplicación está funcionando en producción. Entonces si el proceso de desarrollo esta automatizado, el proceso de retroalimentación es mas rápido.

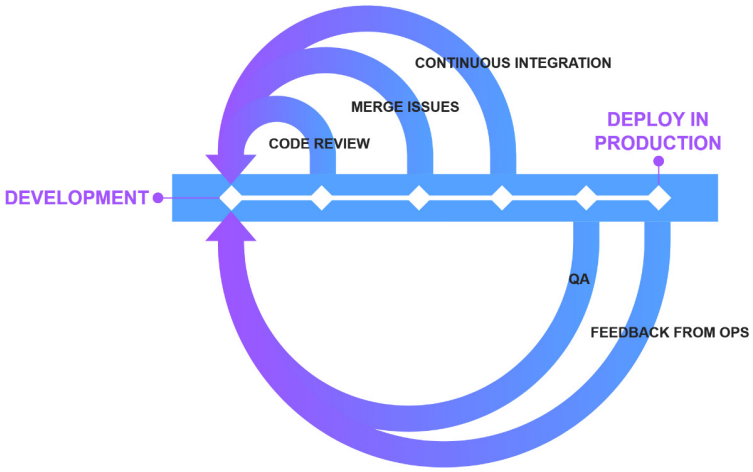


Figura 2.3: Ciclo de retroalimentación continua. Fuente: (Pathania, 2024).

- **Entrega Continua (CD):** Cuando estamos hablando de entregas en ambientes productivos, un artefacto es creado de forma única y almacenado en un repositorio central que funciona como versionado de estos incrementos. Las implementaciones en entornos de prueba se realizan de la misma manera y se utiliza -según la configuración definida en el servidor de integración o de compilación- el mismo artefacto del ambiente producción. Cada compilación debe ser probada de forma automática en una máquina de prueba, dicha máquina debe tener una configuración similar a la del entono de producción (van Merode, 2023).

  La entrega continua automatiza completamente el proceso de lanzamiento de software. Cada modificación realizada desencadena un proceso automatizado que compila, prueba y almacena la actualización como se muestra en la Figura 2.4.

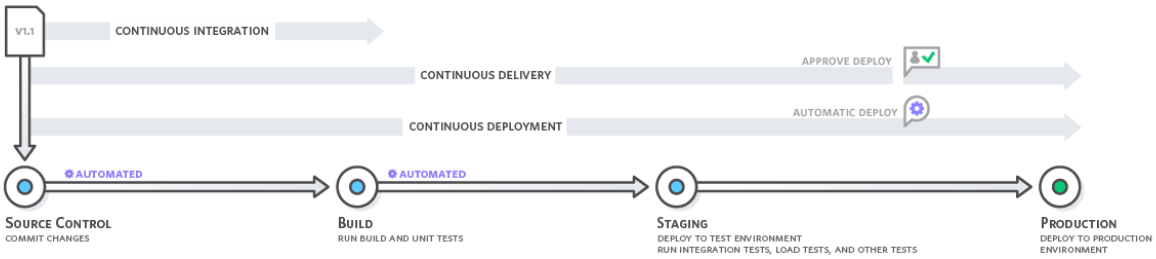


Figura 2.4: Procesos de CD. Fuente: (AWS, sf).

Algunos de los beneficios de la entrega continua (CD):

1. Se automatiza el proceso de despliegue y publicación del software.
2. Mejora la productividad del equipo de desarrollo ya que se reducen las tareas manuales que pueden llevar a errores.
3. Permite que los errores no se conviertan en problemas, esto gracias a la realización de pruebas con mayor frecuencia.
4. La entrega continua permite a su equipo proporcionar actualizaciones a los clientes de forma más rápida y frecuente.

- **Desarrollo Continuo:** Es un proceso de publicación de software que despliega automáticamente los cambios de código en un entorno de producción una vez superadas las pruebas automatizadas. Esto significa que una vez que un cambio de código se considera estable y se verifica mediante pruebas, se envía inmediatamente a producción sin necesidad de intervención manual. IBM (2024).

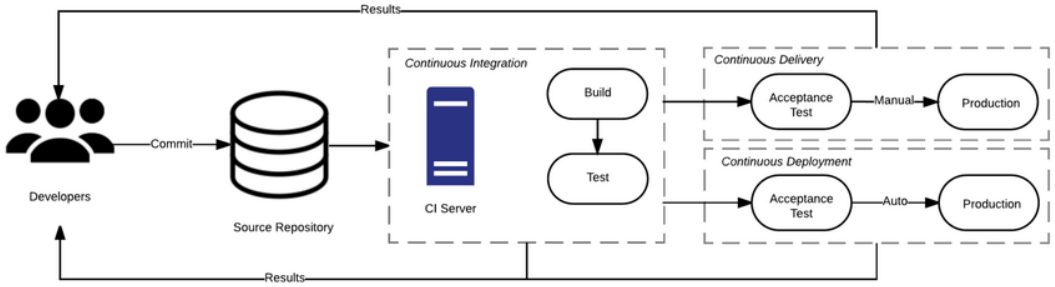


Figura 2.5: Relación entre integración, entrega y despliegue continuos. Fuente: (Shahin et al., 2017).

- **Pipeline:** Su objetivo principal es optimizar el ciclo de vida del desarrollo de software, permitiendo a los equipos entregar software de alta calidad gracias a la retroalimentación rápida, confiable y continua lograda a través de la automatización implementada (Debois et al., 2021).

- **IaC:** Infraestructura como código es un conjunto de scripts, a menudo escritos en diferentes lenguajes de modelado, que define la infraestructura que aprovisionará y las aplicaciones que se implementarán (Chiari et al., 2024).

- **Containers:** Un contenedor es una unidad estándar de software que empaqueta el código y todas sus dependencias para que la aplicación se ejecute de manera rápida y confiable de un entorno informático a otro. Una imagen de contenedor Docker es un paquete de software ejecutable, independiente y liviano que incluye todo lo necesario para ejecutar una aplicación: código, entorno de ejecución, herramientas del sistema, bibliotecas del sistema y configuraciones (Docker, 2024).

- **Arquitectura Microfrontends:** A medida que una aplicación monolítica crece, escalarla se vuelve difícil debido a limitaciones como las restricciones tecnológicas, la necesidad de escalar únicamente verticalmente y la necesidad de reiniciar toda la aplicación con cada implementación Dragoni et al. (2017). Para abordar estos problemas, los desarrolladores están adoptando la arquitectura de microservicios para crear servicios autónomos, distribuidos y débilmente acoplados Dmitry and Manfred (2014). Esta arquitectura permite a los equipos trabajar de forma independiente, lo que reduce el tiempo de desarrollo de nuevas funcionalidades.

  Según (Geers, 2020) los microfrontends ofrecen una solución al extender los principios de los microservicios al frontend, lo que permite la prueba, el desarrollo y la implementación independientes de los componentes del frontend. La idea principal es descomponer una aplicación frontend monolítica en partes más pequeñas que puedan desarrollarse, implementarse y actualizarse de forma independiente como se muestra en la figura 2.6, lo que promueve una mayor flexibilidad y facilidad de mantenimiento.

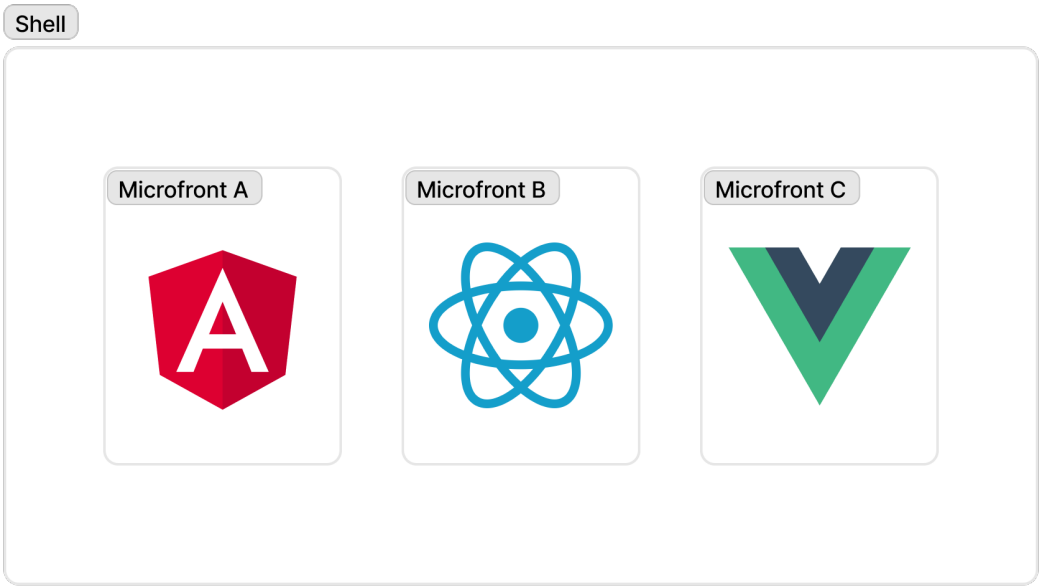


Figura 2.6: Arquitectura Microfrontends

DevOps y sus prácticas técnicas, arquitectónicas y culturales resultantes representan una convergencia de muchos movimientos filosóficos y de gestión. Si bien muchas organizaciones han desarrollado estos principios de forma independiente, comprender que DevOps es el resultado de una amplia gama de movimientos, un fenómeno descrito por John Willis como la 'convergencia de DevOps' (Debois et al., 2021), muestra una asombrosa progresión de pensamiento y conexiones improbables. Hay décadas de lecciones aprendidas de la fabricación, la organización de alta confiabilidad, los modelos de gestión de alta confianza y otros que nos han llevado a las prácticas de DevOps que conocemos hoy.

En este contexto, los gestores de pipelines para integración continua (CI) y entrega continua (CD) han adquirido un papel importante en la cultura DevOps, facilitando la automatización de pruebas, compilaciones y despliegues. A continuación, se presentan las principales tendencias y soluciones actuales a estos sistemas.

En el mercado existen algunas herramientas que permiten la gestión de pipelines, cada una de estas con características únicas. Entre estas las más destacadas son:

- **Jenkins:** Es un servidor de automatización de código abierto autónomo que se puede utilizar para automatizar todo tipo de tareas relacionadas con la creación, prueba y distribución o implementación de software. Jenkins se puede instalar a través de paquetes de sistema nativos, Docker o incluso ejecutarse de forma independiente en cualquier máquina con un entorno de ejecución de Java (JRE) instalado (Jenkins, 2024). Planificar la configuración de Jenkins antes de la implementación es muy importante para garantizar una asignación óptima de recursos, escalabilidad, seguridad, personalización y optimización de costos. Establece las bases para un sistema de CI/CD estable y eficiente que se adapta a las necesidades específicas de su organización y se adapta al crecimiento futuro.

  El diseño de Jenkins se basa en un enfoque distribuido, donde el nodo maestro coordina el proceso de compilación y delega el trabajo real a los nodos agente como se muestra en la figura 2.7:

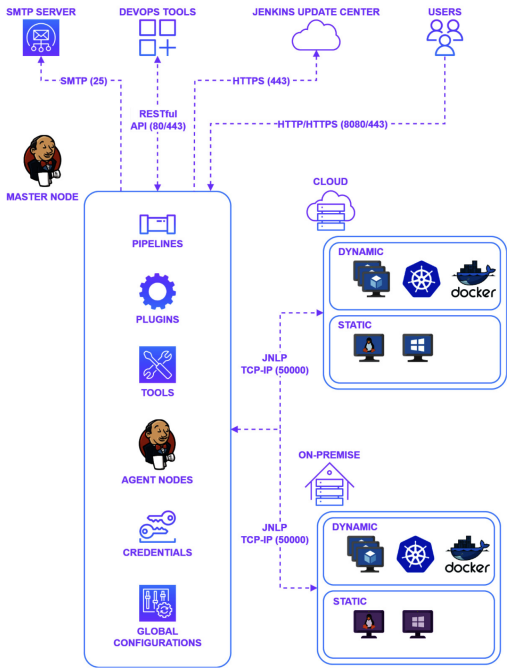


Figura 2.7: Arquitectura del servidor Jenkins. Fuente: (Pathania, 2024)

Algunos de los componentes clave en el servidor Jenkins:

- La arquitectura de Jenkins se centra en el nodo maestro. Este ejecuta todo el sistema y supervisa actividades clave como la programación, la distribución de trabajos de Pipeline, la monitorización de agentes y la creación de una interfaz web para la interacción del usuario. El nodo maestro es responsable de coordinar el proceso de Pipeline de CI.
- Los nodos de agente (también conocidos como nodos de trabajo) se encargan de ejecutar las compilaciones de Pipeline que les asigna el nodo maestro. Pueden ser contenedores o máquinas virtuales (VM) con diferentes sistemas operativos y configuraciones. Además, los nodos de agente pueden ser dinámicos (creados durante la ejecución de Pipeline y destruidos posteriormente) o estáticos (precreados y persistentes). Los plugins de Jenkins permiten conectar los nodos de agente al nodo maestro. Se pueden vincular varios nodos de agente.
- Jenkins cuenta con un sólido ecosistema de plugins que mejoran sus capacidades. Instalar y configurar plugins permite interactuar con una amplia gama de herramientas, tecnologías y servicios de DevOps.
- Jenkins admite pipelines, que son secuencias de etapas interconectadas que conforman el proceso de CI/CD. Puedes crear un pipeline de Jenkins mediante código o a través de la interfaz de usuario.
- La función de credenciales de Jenkins se utiliza para almacenar de forma segura datos confidenciales, como contraseñas, tokens de API, claves SSH y otras credenciales de autenticación requeridas por los complementos y pipelines de Jenkins.
- Las herramientas de configuración global es una función que permite a los administradores de Jenkins instalar y configurar herramientas globalmente en todo el entorno. Estas herramientas pueden incluir herramientas de control de versiones, herramientas de compilación, herramientas de análisis de código estático o cualquier otro software requerido por el pipeline.

A pesar de estar clasificada como gratuita, Jenkins se evidencia que tiene una complejidad, a pesar de que cuenta con una gran comunidad de soporte. La Figura 2.8 muestra una comparación de herramientas de CI/CD.

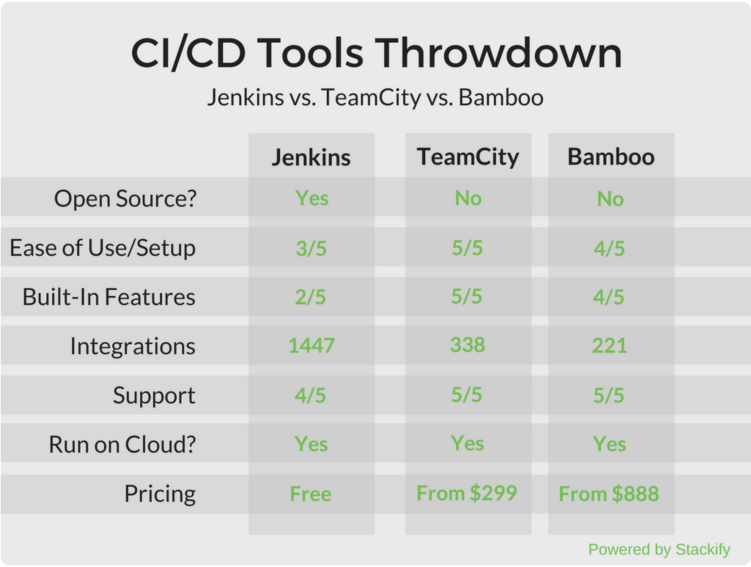


Figura 2.8: Herramientas CI/CD Código abierto vs. Comercial. Fuente: (Stackify, 2017)

- **CircleCI:** Es una plataforma de CI/CD creada por profesionales de DevOps para ayudarlo a ajustar todo su proceso de desarrollo de principio a fin. Ayuda a los equipos de ingeniería a crear, probar e implementar software mientras verifican los cambios de código en tiempo real con la interfaz de usuario del panel de CircleCI (CircleCI, 2024).

- **Bamboo:** Es una herramienta para equipos profesionales que desarrollan software. La aplicación es una solución de tipo CI/CD, lo que significa que permite la integración constante de elementos de código particulares y la entrega de los consiguientes 'paquetes', que son partes de un programa (Atlassian, 2024).

- **AWS CodePipeline:** Es un servicio de entrega continua que puede utilizar para modelar, visualizar y automatizar los pasos necesarios para lanzar su software. Puede modelar y configurar rápidamente las diferentes etapas de un proceso de lanzamiento de software. (Services, 2024).

- **Azure DevOps:** Respalda una cultura colaborativa y un conjunto de procesos que reúnen a desarrolladores, gerentes de proyectos y colaboradores para desarrollar software. Permite a las organizaciones crear y mejorar productos a un ritmo más rápido que con los enfoques de desarrollo de software tradicionales (Microsoft, 2024).

- **Travis CI:** Travis CI es la herramienta de integración continua y distribución continua más

sencilla y flexible que existe en la actualidad. Descubra cómo Travis CI puede ayudar con la integración continua y la distribución continua (CI, 2024).

2.2. Estado del Arte

En la búsqueda de literatura se encontraron diferentes enfoques y herramientas asociadas para apalancar y facilitar prácticas de integración continua, entrega continua y despliegue continuo. Estas propuestas fueron clasificadas en seis categorías según sus características funcionales y el área del pipeline en la que son aplicadas.

1. Reducir el tiempo de construcción y pruebas en Integración Continua (CI).
2. Incrementar la visibilidad y conciencia sobre los resultados de construcción y pruebas en CI.
3. Soportar pruebas continuas (semi)automatizadas.
4. Detectar violaciones, errores y fallas en CI.
5. Abordar problemas de seguridad y escalabilidad en el pipeline de despliegue.
6. Mejorar la confiabilidad y la disponibilidad del proceso de despliegue.

A continuación se elabora una tabla con las herramientas y clasificadas según su categoría.

Categoría	**Descripción de enfoques y herramientas**
2	Técnica Wallboard: Indica el estado actual de integración y entrega de todas las ramas dentro de un proyecto.
2	SQA-Mashup: Puede integrar y visualizar datos producidos en entornos CI.
1	VMVM/VMVMVM: Se utiliza para aislar las dependencias en memoria y externas entre los casos de prueba.
2	CIViT: Se utiliza para visualizar el proceso integral de las actividades de prueba en la transformación a la integración continua.
4	WECODE: Detecta automática y continuamente conflictos de fusión de software antes de que un sistema de control de versiones sea utilizado por los desarrolladores, así como detecta conflictos en código no comprometido.
1,3	uBuild: Proporciona pruebas continuas realizando pruebas reproducibles y deterministas con el fin de lograr construcciones automatizadas.

Continúa en la siguiente página

Categoría	Descripción de enfoques y herramientas
4	Integración Continua Incremental Backtracking: A través de enfoques de backtracking simples y efectivos, este enfoque aumenta la resistencia del proceso de construcción contra fallos y asegura que una versión funcional esté siempre disponible.
2	Robot BuildBot: Notifica quién es el responsable del fallo de la prueba en el entorno de integración continua de forma amigable y divertida.
1	Hydra: Es una herramienta de construcción continua basada en Nix, que produce automáticamente un entorno de construcción para los proyectos.
2	SQA-Perfil: A través de un conjunto de reglas, puede proporcionar una composición dinámica de cuadros de mando CI basados en actividades de las partes interesadas en herramientas de un entorno CI.
1,4	Enfoque Hotspot: Identifica archivos de cabecera que actúan como cuellos de botella en el proceso de construcción dentro de la infraestructura de CI, con el objetivo de acelerar el sistema de construcción.
5	Secure Build Server: Extiende el servidor de compilación predeterminado en entornos CI mediante el encapsulamiento de trabajos infectados y evita la propagación de amenazas en sistemas CI multiempresa.
3,4	Pruebas automáticas y ágiles de líneas de productos basadas en pruebas de interacción combinatoria : Hace pruebas automáticas como parte integrada del marco de integración continua y permite a los desarrolladores de líneas de productos de software identificar posibles fallos de interacción en el proceso de construcción.
2,4	Enfoque basado en la conciencia del entorno: Aumenta el conocimiento del estado de la construcción entre los miembros del equipo, lo que resulta en la disminución del número de construcciones rotas y un fuerte sentido de la responsabilidad hacia los fallos en el proceso de construcción.
1,4	Integración de la localización de fallos y la técnica de priorización de casos de prueba en la IC: Los enfoques de localización de fallos y priorización de casos de prueba se combinan para dar soporte al commit construido en la integración continua, lo que consecuentemente mejora la eficiencia (tiempo) y la efectividad de todo el proceso de CI.

Continúa en la siguiente página

Categoría	Descripción de enfoques y herramientas
2,3	Marco de automatización de pruebas de NHN: Apoya las prácticas de IC mediante la automatización de procesos repetitivos y propensos a errores-para la realización de pruebas en un entorno de integración continua. Facilita la comunicación entre las distintas partes interesadas de utilizando tablas para representar las pruebas y los entornos de prueba.
3	Pruebas de Integración Continua para SOA: Unified Test Framework (UTF) for Continuous Integration Testing (CIT) of SOA, que automatizaría parcialmente la generación de casos de prueba en CIT utilizando diagramas de secuencia como entrada.
4	Script definido por el usuario: Soporta el cumplimiento en el momento de la confirmación estableciendo un paso previo a la confirmación (es decir,Subversion pre-commit hook) para forzar a los desarrolladores a arreglar la violación en la confirmación del código. No permite confirmar códigos que no cumplan con las convenciones (por ejemplo, arquitectura as-build).
5	Integración continua empresarial: Una versión modificada del proceso de integración continua normal para dividir el proyecto en varios módulos utilizando dependencias binarias. A pesar de que cada módulo tiene su propia IC, ECI proporciona la retroalimentación que proporciona la IC de un solo proyecto.
2,3,4	Tinderbox: Es un sistema de integración continua y pruebas automatizadas, que ayuda a encontrar problemas de integración antes en el ciclo de desarrollo, reducir el coste de solucionar los problemas de integración y mejorar la visibilidad y la conciencia entre los miembros del equipo.
3,4	Surrogate: Un marco de simulación para implementar pruebas de integración continua para sistemas SOA cuando algunos componentes o incluso todos los componentes aún no están disponibles. Con esto, los errores se identifican en la etapa temprana de desarrollo.
3,4	CiCUTS: Herramienta de ejecución de modelado de sistemas, con entornos de integración continua. Gracias a este enfoque, los desarrolladores y probadores pueden realizar continuamente pruebas de integración del sistema en entornos de destino utilizando enfoques de emulación e identificar problemas de rendimiento antes de que los componentes estén completamente implementados.

Continúa en la siguiente página

Categoría	Descripción de enfoques y herramientas
1	Selección continua de pruebas de regresión (CRTS): Permite ejecutar eficazmente pruebas de regresión dentro de entornos de desarrollo de integración continua. La técnica mejora la rentabilidad de las pruebas previas al envío (es decir, las pruebas realizadas por los desarrolladores antes de enviar el código al repositorio) y reduce los costes de ejecución de los casos de prueba.
1, 4	Priorización continua de conjuntos de pruebas (CTSP): Puede reducir los retrasos en la detección de fallos durante las pruebas posteriores al envío (es decir, todas las pruebas que se realizan después de enviar el código al repositorio). En general, puede mejorar la rentabilidad del proceso de integración continua.
6	Rondo: La adopción del despliegue continuo en entornos dinámicos como los de computación omnipresente conlleva una serie de retos. El proceso de despliegue en tales entornos debe ser reproducido en diferentes sitios, soportar la personalización, y debe estar equipado con un mecanismo de reversión personalizado. Rondo, una herramienta de automatización, satisface todos los requisitos mencionados para facilitar la adopción de la práctica de despliegue continuo en entornos dinámicos.
1,2	Code-Churn Based Test Selection (CCTS): Esta técnica analiza el código y los resultados de la ejecución de pruebas para seleccionar un subconjunto óptimo de suites de pruebas a nivel de sistema. Así, ayuda a las organizaciones de desarrollo de software a gran escala a acelerar la IC. Permite a los miembros del equipo comprender mejor el número de fallos en las pruebas.
5	Mejora del diseño de seguridad de un canal de despliegue: Este enfoque integra cuatro fragmentos de diseño de seguridad (es decir, patrones de seguridad) a nivel de diseño para asegurar los pipelines de despliegue. La seguridad del conducto se garantiza no permitiendo que se despliegue código malicioso a través del conducto y evitando la comunicación directa entre componentes en los entornos de prueba y producción.
1,2	Morpheus: Facilita la práctica de la IC mediante la mejora de la calidad de la retroalimentación, en la que cada desarrollador sólo recibe los resultados de las pruebas de su propio código modificado (es decir, fácil interpretación de los resultados de las pruebas).

Continúa en la siguiente página

Categoría	Descripción de enfoques y herramientas
6	Dependabilidad orientada a procesos (POD): Un enfoque para mejorar la fiabilidad del proceso de despliegue en dificultad del diagnóstico de errores durante operaciones esporádicas cuando se adopta e implementa la práctica de la DC.

Tabla 2.1: Clasificación de enfoques y herramientas para CI/CD

Un pipeline de despliegue debe incluir etapas explícitas (por ejemplo, build and packaging) para transferir el código desde el repositorio de código a el entorno de producción. La automatización es una práctica crítica de en el proceso de despliegue; sin embargo, algunas tareas manuales de (por ejemplo, tareas de control de calidad) son inevitables. Cabe señalar que las herramientas presentadas en la figura 2.9 son en su mayoría, herramientas comerciales y de código abierto existentes,que tienen como objetivo formar e implementar un pipeline de despliegue.

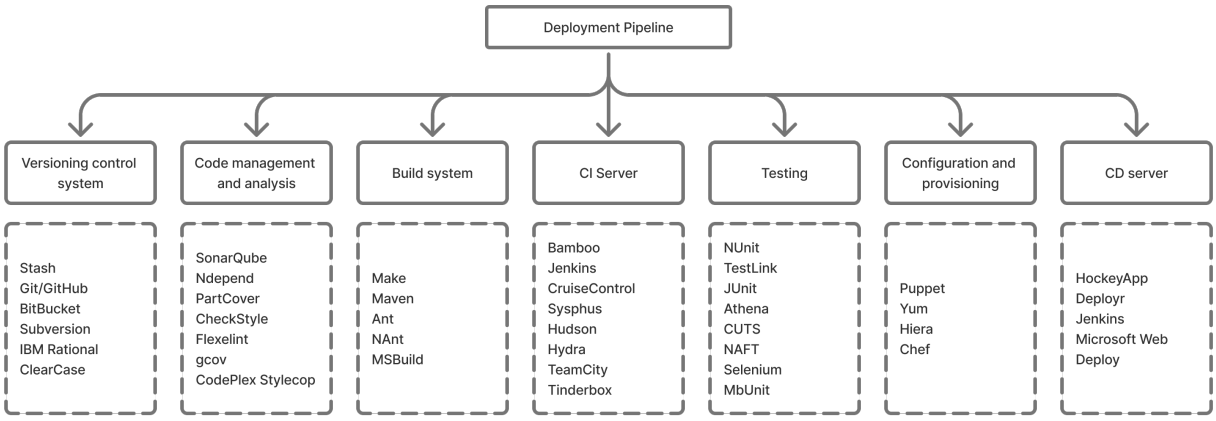


Figura 2.9: Visión general de las herramientas utilizadas. Fuente: propia.

Con CI/CD, los equipos de software pueden mantener unos estándares de alta calidad al tiempo que aumentan la velocidad y la frecuencia de sus lanzamientos, apoyando en última instancia la innovación y la satisfacción del usuario. En la tabla se realiza una comparación de modelos tradicionales vs CI/CD Software como se muestra en la Tabla 2.2.

Aspecto	Modelo tradicional en cascada	CI/CD (Integración continua / Entrega continua)
Ciclo de desarrollo	Secuencial y lineal; cada fase debe completarse antes de que comience la siguiente.	Iterativo y incremental; permite el trabajo paralelo y la integración continua de los cambios.

Continúa en la siguiente página

Aspecto	Modelo tradicional en cascada	CI/CD (Integración continua / Entrega continua)
Velocidad de entrega	Entrega más lenta; ciclos largos de desarrollo con pruebas y validaciones extensas antes del lanzamiento.	Entrega más rápida; integraciones frecuentes y versiones incrementales más pequeñas.
Pruebas (Testing)	Las pruebas ocurren al final del ciclo de desarrollo, a menudo después de tener el producto completo.	Las pruebas son continuas y automatizadas, integradas en cada etapa del pipeline, ofreciendo retroalimentación inmediata.
Detección de errores	Los errores suelen descubrirse al final del proceso, lo que los hace más costosos de corregir.	Los errores se detectan tempranamente mediante pruebas unitarias y de integración automatizadas, reduciendo el costo de corrección.
Flexibilidad	Menor flexibilidad; los cambios en los requerimientos son difíciles y costosos de implementar una vez iniciado el desarrollo.	Mayor flexibilidad; los cambios pueden realizarse e integrarse continuamente durante el proceso de desarrollo.
Gestión del riesgo	Mayor riesgo al final del ciclo; el producto se despliega solo tras el desarrollo completo.	Menor riesgo; las funcionalidades se despliegan de forma incremental y las versiones frecuentes permiten una detección temprana de problemas.
Colaboración	Colaboración limitada entre los equipos de desarrollo, pruebas y operaciones.	Alta colaboración entre desarrollo, pruebas y operaciones, siguiendo los principios de DevOps.
Retroalimentación del cliente	La retroalimentación se recibe generalmente después de la entrega del producto, lo que puede generar demoras en los ajustes.	Se recoge retroalimentación continua de usuarios y partes interesadas, lo que permite responder con rapidez a cambios o problemas.

Continúa en la siguiente página

Aspecto	Modelo tradicional en cascada	CI/CD (Integración continua / Entrega continua)
Costo del cambio	Mayor costo de cambio debido a la detección tardía de errores y a la naturaleza secuencial del desarrollo.	Menor costo de cambio, ya que el proceso iterativo permite ajustes más rápidos y sencillos.
Automatización	Procesos manuales para compilar, probar y desplegar software.	Alto nivel de automatización en la compilación, pruebas y despliegue, lo que reduce el esfuerzo manual y acelera los ciclos.

Tabla 2.2: Comparación de los ciclos de entrega de software tradicional frente a CI/CD. Fuente: (Ugwueze and Chukwunweike, 2024)

Aunque las barreras culturales y organizativas son significativas, los retos técnicos de también pueden suponer un obstáculo considerable para la adopción de CI/CD. Una de las barreras técnicas más frecuentes es la integración con sistemas legacy. Muchas empresas de dependen de aplicaciones legacy que no se diseñaron para las prácticas modernas de CI/CD. Integrar estos sistemas antiguos con las nuevas canalizaciones de CI/CD requiere un trabajo considerable para refactorizar y modernizar la arquitectura subyacente, haciéndola compatible con los flujos de trabajo automatizados. Los sistemas legacy a menudo se basan en procesos manuales o infraestructuras obsoletas que no pueden automatizarse fácilmente, lo que provoca retrasos y una complejidad adicional al intentar implantar pipelines de CI/CD.

Desafío	Descripción	Solución
Integración con sistemas legados	Muchas organizaciones dependen de sistemas antiguos que no fueron diseñados con prácticas de CI/CD, lo que dificulta su integración.	Refactorizar sistemas legados en fases incrementales, utilizando envoltorios API (API wrappers), contenedores y microservicios para integrarlos gradualmente a los pipelines de CI/CD.
Deuda técnica	Con el tiempo, prácticas de codificación deficientes, bibliotecas obsoletas y pruebas insuficientes generan una acumulación de problemas.	Priorizar la resolución de deuda técnica mediante refactorización, mejora de la documentación y automatización de pruebas. Realizar revisiones periódicas para mantener la calidad del código.

Continúa en la siguiente página

Desafío	Descripción	Solución
Escalabilidad de pipelines	Gestionar pipelines que crecen con el aumento de servicios, microservicios o equipos de desarrollo puede volverse complejo.	Implementar soluciones CI/CD escalables basadas en la nube (como AWS o Azure), utilizando contenedores y Kubernetes para manejar implementaciones a gran escala.
Falta de colaboración	Los equipos aislados pueden dificultar la integración fluida y generar ineficiencias en el proceso CI/CD.	Fomentar una cultura DevOps con colaboración regular entre desarrollo, operaciones y seguridad. Incorporar herramientas que faciliten la comunicación, como Slack y JIRA.
Seguridad y cumplimiento	La integración continua y los despliegues frecuentes pueden exponer vulnerabilidades si no se automatizan las verificaciones de seguridad.	Integrar herramientas de automatización de seguridad (como Snyk o SonarQube) en el pipeline. Automatizar verificaciones de cumplimiento para garantizar la seguridad y cumplir normativas.
Desafíos en pruebas	Las pruebas automatizadas suelen ser insuficientes o estar mal integradas, lo que permite que errores lleguen a producción.	Integrar pruebas automatizadas completas (unitarias, de integración y rendimiento). Usar análisis de cobertura y probar en diferentes entornos (staging, producción).
Fallos en el despliegue	Los fallos frecuentes en los despliegues pueden deberse a configuraciones deficientes o errores manuales.	Automatizar estrategias de reversión (rollback) y usar sistemas autocorrectivos. Implementar despliegues tipo canario o blue-green para minimizar riesgos.
Complejidad en herramientas	La cantidad de herramientas necesarias para cada parte del proceso CI/CD puede generar retos de configuración.	Simplificar el uso de herramientas mediante plataformas integradas como GitLab CI, Jenkins o CircleCI. Asegurar compatibilidad y facilidad de mantenimiento de toda la cadena de herramientas.

Tabla 2.3: Principales desafíos en la implementación de CI/CD y sus soluciones. Fuente: (Ugwueze and Chukwunweike, 2024)

Implementar CI/CD eficazmente requiere más que solo herramientas técnicas; requiere un cambio cultural y organizacional. Uno de los desafíos más importantes en la adopción de CI/CD es la resistencia organizacional. Muchas organizaciones, especialmente aquellas con flujos de trabajo establecidos, pueden dudar en cambiar. Los empleados pueden sentirse cómodos con los métodos

de desarrollo tradicionales y ver CI/CD como una carga adicional en lugar de una herramienta que mejora la productividad. Esta resistencia a menudo proviene del miedo a interrumpir los procesos existentes o de la complejidad percibida al adoptar nuevas herramientas y metodologías Mohammad (2016).

Capítulo 3

Desarrollo del Proyecto

3.1. Descripción de la propuesta

Russell es un administrador de pipelines orientado a facilitar la configuración, creación, monitoreo y visualización de procesos de integración y entrega continua (CI/CD) dentro de entornos de desarrollo modernos, que aún no han dado el salto a estas herramientas de automatización.

Su principal propósito es brindar una herramienta que permita orquestar pipelines de forma fácil, mediante una interfaz gráfica amigable, mitigando la dependencia de personal experto con conocimiento avanzado en configuración y manejo de archivos complejos y/o especializados en herramientas de automatización. Russell está pensado para ayudar en contextos donde se busca agilizar los flujos de despliegue asociados al uso directo de herramientas tradicionales.

3.2. Identificación de requerimientos funcionales y no funcionales

3.2.1. Requerimientos funcionales

ID	Requerimiento Funcional
RF-01	La solución Russell debe permitirse realizar a través de una instalación desatendida.
RF-02	El sistema debe permitir la creación, edición de servidores, proyectos y pipelines.
RF-03	El sistema debe conectarse a Jenkins mediante su API REST, para permitir notificar a los usuarios el estado actual de cada pipeline (exitoso, fallo).
RF-04	El sistema debe ofrecer vistas históricas de ejecuciones anteriores.
RF-05	El sistema debe permitir que los usuarios, a través de la interfaz frontend, personalicen el tipo de ejecución de los pipelines (Backend/Frontend).
RF-06	El sistema debe conectarse a Jenkins mediante su API REST.
RF-07	El sistema debe permitir la autenticación de usuarios y asignación de roles (admin, DevOps, auditor).
RF-08	El sistema debe registrar las operaciones realizadas dentro del sistema con el fin de dejar registros de logs.

Continúa en la siguiente página

ID	Requerimiento Funcional
RF-09	El sistema debe permitir buscar y filtrar pipelines por nombre, estado, fecha y usuario.
RF-10	El sistema debe soportar la gestión de pipelines por entornos (por ejemplo: desarrollo, productivo).

Tabla 3.1: Requerimientos funcionales del gestor de pipelines con Jenkins

3.2.2. Requerimientos no funcionales

ID	Requerimiento No Funcional	Atributo de Calidad
RNF-01	El sistema debe responder en menos de 2 segundos en el 95 % de las solicitudes.	Rendimiento
RNF-02	El sistema debe escalar horizontalmente para soportar más usuarios y pipelines.	Escalabilidad
RNF-03	El sistema debe tener una disponibilidad de 8h al día, 5 días a la semana.	Disponibilidad
RNF-04	Las credenciales y datos sensibles deben estar protegidos con mecanismos de seguridad adecuados.	Seguridad
RNF-05	El sistema debe implementar validaciones basada en roles.	Seguridad
RNF-06	El sistema debe ser compatible con Jenkins LTS versión 2.x o superior.	Integración
RNF-07	Toda acción relevante debe quedar registrada con marca de tiempo y usuario.	Auditabilidad
RNF-08	El sistema debe estar preparado para soportar múltiples idiomas.	Usabilidad

Tabla 3.2: Requerimientos no funcionales y sus atributos de calidad

3.3. Diseño de la arquitectura

Esta sección describe la arquitectura general de Russell, sus atributos de calidad, componentes principales y el diseño de los flujos de interacción entre ellos.

3.3.1. Diseño de arquitectura N capas

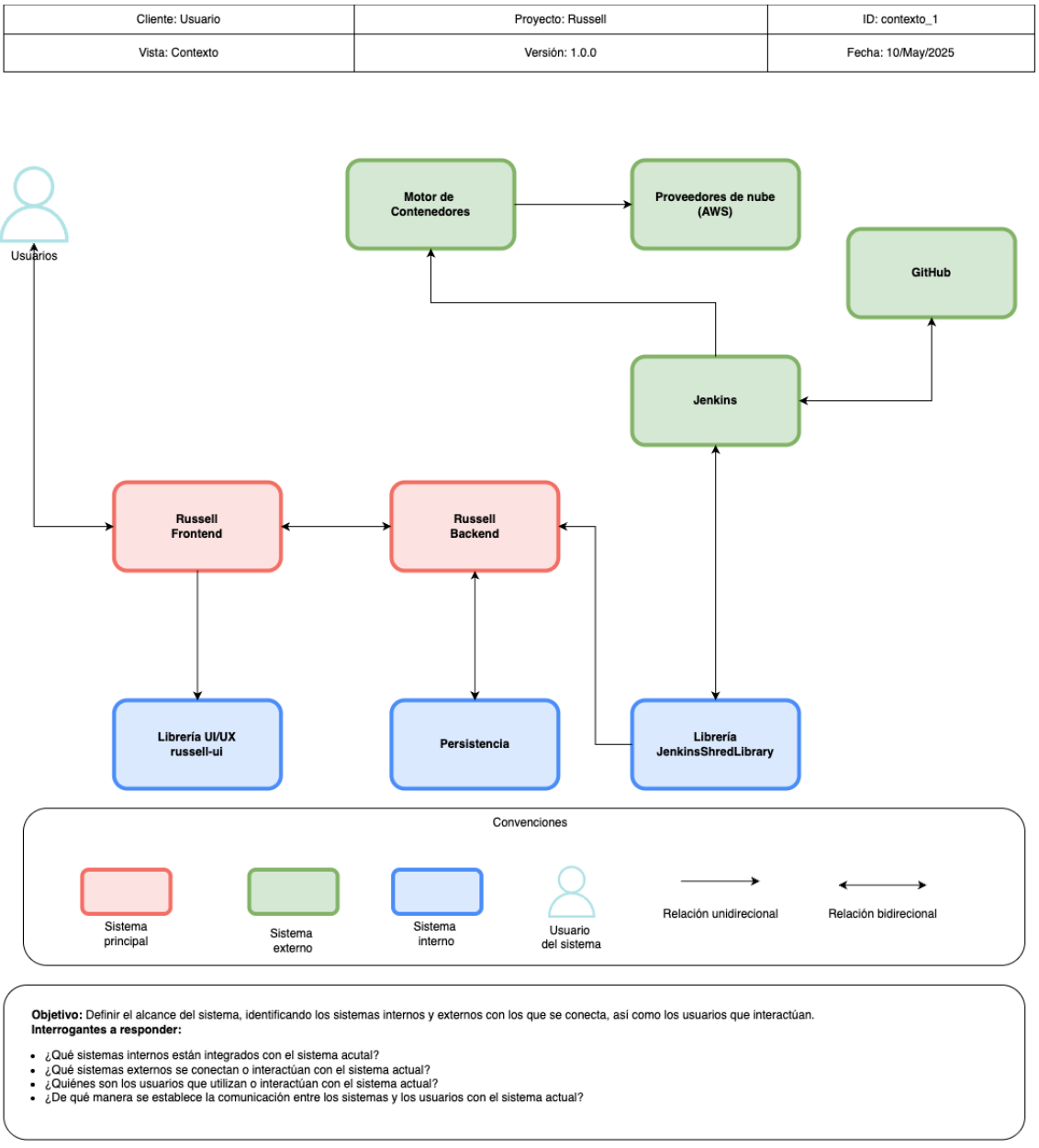


Figura 3.1: Diagrama de contexto Russell.

3.3.2. Arquitectura general del sistema

Russell se basa en una arquitectura cliente-servidor con separación de responsabilidades entre frontend, backend y herramientas externas. A continuación, se presenta el diagrama de arquitectura general de la solución:

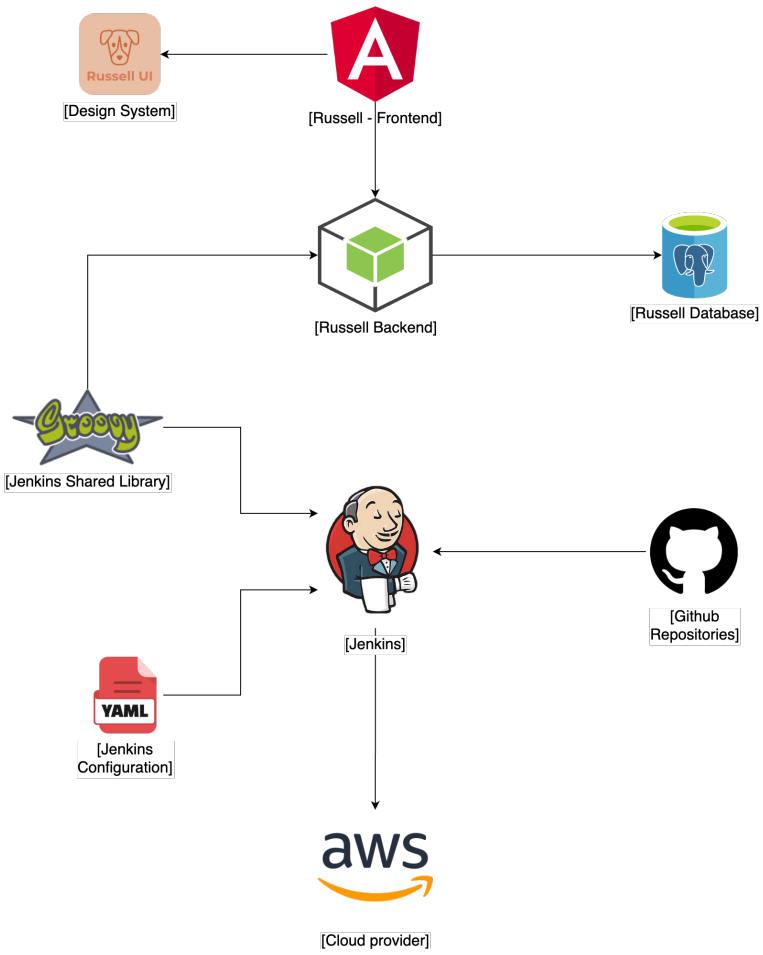


Figura 3.2: Arquitectura general Russell.

3.3.3. Componentes principales

3.3.3.1. Arquitectura Frontend

Russell se diseño basado en una arquitectura microfrontend debido a la necesidad de desarrollar un sistema modular, escalable y manteniente a largo plazo. Dentro de las funcionalidades de Russell se contempla algunos de los siguientes módulos, administración de pipelines, dashboard, seguridad, usuarios y logs, lo cual justifica dividir la aplicación en múltiples módulos independientes.

En este proyecto, se optó por utilizar Angular, ya que es un framework de código abierto y

basado en componentes para crear aplicaciones web robustas y escalables, cuenta con colección de bibliotecas que cubren una amplia variedad de funciones, como enrutamiento, gestión de formularios, comunicación cliente-servidor, de fácil integración con module federation adoptando de forma temprana la arquitectura de microfrontends lo que potencia su capacidad de escalar de forma modular. Esta decisión arquitectónica habilita, a futuro, que la herramienta sea extensible y abierta a la incorporación de nuevos módulos funcionales, incluso por parte de una comunidad externa de desarrollo, promoviendo así la colaboración y evolución continua del sistema.

A continuación se muestra la arquitectura de la aplicación web basada en microfrontends:

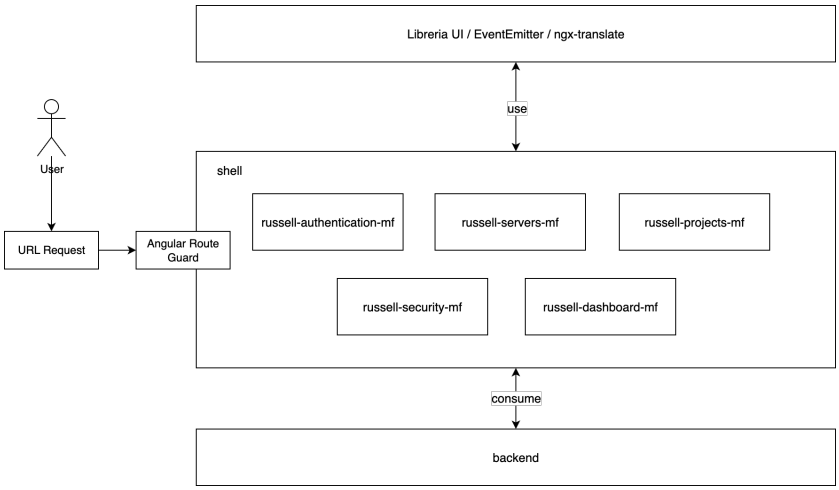


Figura 3.3: Arquitectura microfrontends Russell.

El host o contenedor principal es el que se encarga de integrar y orquestar todos los microfrontends. Es el responsable de:

- Gestionar la navegación de las rutas globales y controlar su acceso.
- Cargar cada uno de los microfrontends según sea necesario.
- Gestionar la forma en que se comunican cada uno de los microfrontends entre ellos, se hace mediante eventos.

Descripción de los microfrontends del sistema propuesto:

- **russell-authentication-mf**: Este microfrontend tiene la responsabilidad de gestionar el manejo de las sesiones de los usuarios del sistema,logueo, deslogueo y centralizar la identidad de los usuarios informando mediante la emisión de eventos a los demás microfrontends el estado actual de la autenticación.
- **russell-dashboard-mf**: Este microfrontend se encarga de mostrar una vista centralizada de la información clave para el usuario, como lo son número de servidores, proyectos, pipelines, identificación de tipos de proyectos.

- **russell-servers-mf**: Este microfronted brinda una visual general de los diferentes nodos maestros de Jenkins en los que una organización puede clasificar sus proyectos o trabajos en ejecución, muestra el estado del servidor, también permite registrar nuevos servidores.
- **russell-projects-mf**: Este microfronted brinda una visual general de todos los proyectos gestionado bajo la herramienta de la solución propuesta, permite a los usuarios crear proyectos que se organizan en los nodos maestros de Jenkins a los que se asocien facilidando la organización de pipelines a la empresa.
- **russell-pipelines-mf**: Este ayuda a los usuarios de Russell a visualizar y registrar de forma fácil los trabajos de despliegue de cada uno de los componentes contenidos dentro de un repositorio de código en GitHub, logrando clasificar estos en Backend y Frontend.
- **russell-security-mf**: Por último este módulo que permite la creación y asociación de roles (Admin, DevOps, Auditor) a los usuarios de Russell, también permite el registro de credenciales de la cuentas cloud y de GitHub, para poder acceder a estas plataformas, que son la base del desarrollo de las empresas; finalmente cuenta con la funcionalidad de auditar toda operación del sistema dando una trazabilidad de las ejecuciones de los usuarios.

3.3.3.2. Comunicación en una arquitectura de microfrontends

Para la orquestación de los diferentes microfrontends de los que se componen la solución propuesta es importante definir la forma en se van a comunicar entre sí, el enfoque seleccionado para el intercambio información es emisión de eventos, cubierto con la librería *ng-event-bus*. ng-event-bus incluye un sistema de eventos basado en el patrón publish-subscribe, que es el mas común en aplicaciones cliente-servidor.

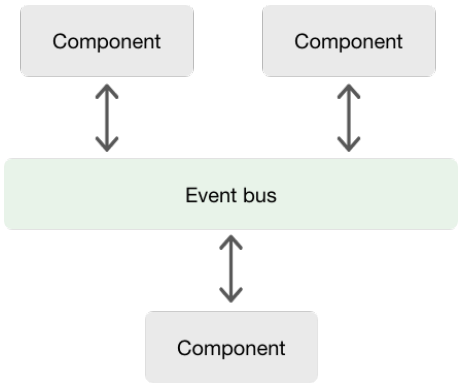


Figura 3.4: *ng-event-bus* para comunicación entre MFEs.

3.3.3.3. Control de acceso basado en roles en una arquitectura de microfrontends

Para garantizar la seguridad y la segmentación adecuada de funcionalidades, se implementó un sistema de control de acceso basado en roles, apoyado en el uso de *route guards* tanto en el shell

como en los microfrontends.

En el contenedor principal se desarrolla un sistema de autenticación que permite identificar al usuario mediante el uso de tokens de sesión. A partir de esta autenticación, se obtiene el rol asignado al usuario, lo cual permite restringir el acceso a cada ruta configurada en el enrutador principal. Esto se logra mediante el uso de guardas (*guards*) en las rutas, que evalúan si el rol del usuario coincide con los roles permitidos especificados en la configuración (*Admin,DevOps,Auditor*).

A nivel de código, esto se define mediante metadatos en las rutas, como se muestra en el siguiente ejemplo:

```
#Typescript
{
  path: 'dashboard',
  loadChildren: () => import('remote/dashboard').then(m => m.
     DashboardModule),
  canActivate: [AuthGuard],
  data: { roles: ['admin', 'devops'] }
}
```

Cada microfrontend, por su parte, cuenta con sus propias rutas internas, las cuales también se encuentran protegidas por sus respectivos *guards*, lo que permite una segunda capa de verificación que impide el acceso incluso si un usuario logra cargar el módulo correspondiente.

3.3.3.4. Sistema de diseño como base para la escalabilidad del frontend

A media que se desarrolla la herramienta se evidencia la necesidad de tener una consistencia visual entre todos los microfrontends orquestados en el shell, esto propicia la creación de un componente transversal como respuesta a esta necesidad. Este componente se desarrollo como una librería que no solo puede satisfacer los requerimientos del proyecto actual, sino que puede ser usada de forma transversal en cualquier proyecto que requiera tener una consistencia visual en sus interfaces gráficas de usuario, con el ánimo de propender la reutilización de estos componentes gráficos.

Esta librería se concibe como un producto independiente desarrollado con *StencilJS*, un compilador de Web Components que permite crear componentes encapsulados, reutilizables y compatibles con múltiples frameworks como Angular, React y Vue, así como con aplicaciones desarrolladas en HTML puro. Esta librería forma parte del sistema de diseño, entendido como un conjunto coherente de elementos visuales y funcionales construidos bajo la metodología de *Atomic Design*. Dicha metodología propone una jerarquía inspirada en la química —átomos, moléculas, organismos, plantillas y páginas— que permite estructurar los componentes desde los elementos más simples hasta los más complejos. Esta organización favorece la reutilización, la escalabilidad y la mantenibilidad del sistema, ya que los componentes pueden ensamblarse entre sí de manera similar a las piezas de Lego, facilitando así el desarrollo eficiente de nuevos productos digitales Tidwell et al. (2020).

El desarrollo de la librería inicia con la configuración del proyecto mediante la herramienta de inicialización de *StencilJS*, seleccionando la opción orientada a librerías de componentes. Posteriormente, cada componente se define utilizando decoradores como `@Component` y `@Prop`, junto con

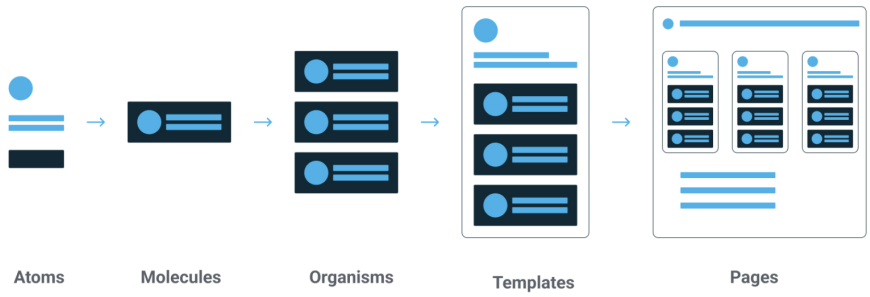


Figura 3.5: Metodología Atomic Design. Fuente: (Martínez, 2024)

estructuras TSX para el marcado HTML. Además, el estilo se encapsula utilizando el modo `shadow DOM`, lo que garantiza el aislamiento de estilos entre componentes.

A modo de ejemplo, un componente de botón puede estructurarse de la siguiente forma:

```
#TSX
@Component({
  tag: 'russell-button',
  styleUrl: 'russell-button.css',
  shadow: true
})
export class RussellButton {
  @Prop() label: string;

  render() {
    return <button>{this.label}</button>;
  }
}
```

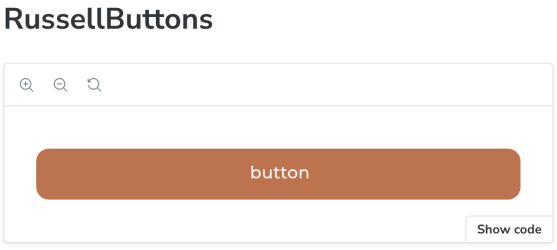


Figura 3.6: Uso del tag HTML <russell-button>.

Es importante destacar que un sistema de diseño no se limita únicamente a la creación de componentes web, sino que parte desde los *foundations* o fundamentos. Estos representan los elementos

básicos, reutilizables y coherentes que sustentan tanto la apariencia visual como el comportamiento funcional de los componentes de interfaz.

Dentro de estos *foundations* de *@russell-libs/russell-ui*, los **tokens** juegan un papel fundamental al actuar como unidades semánticas que encapsulan decisiones clave de diseño. Su función principal es garantizar la *consistencia visual*, facilitar la *escalabilidad del sistema* y simplificar la *mantenibilidad del código*, permitiendo que cualquier ajuste en la base se refleje automáticamente en toda la interfaz.

Algunos de los tokens usados en *@russell-libs/russell-ui* son:

- **Color:** primarios, secundarios, de fondo, de texto y de estado (éxito, error, advertencia).
- **Tipografía:** familias tipográficas, tamaños y pesos.
- **Espaciado:** márgenes y rellenos (*paddings*).
- **Bordes:** anchos y radios.
- **Sombras** y **opacidad**.
- **Iconografía**.
- **Transiciones y animaciones**.

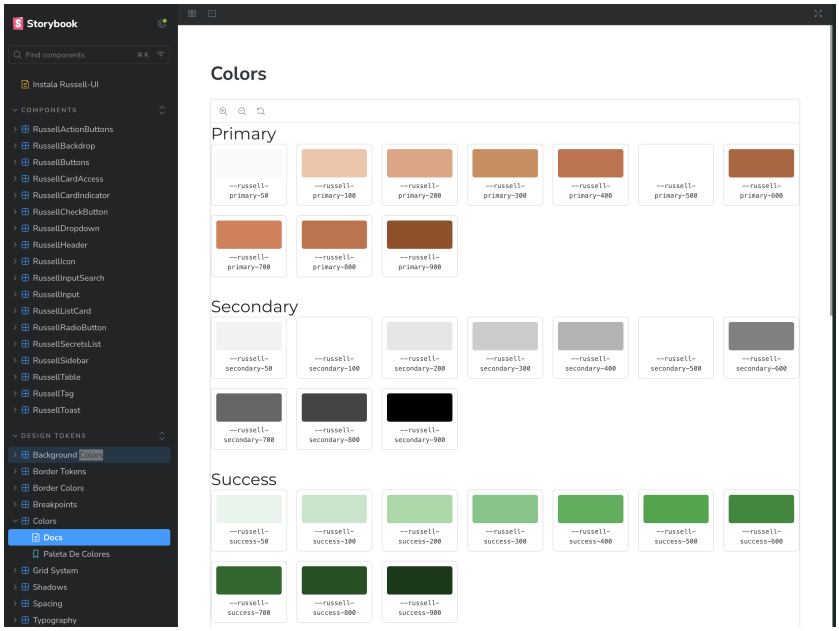


Figura 3.7: Tokens de color sistema de diseño Russell.

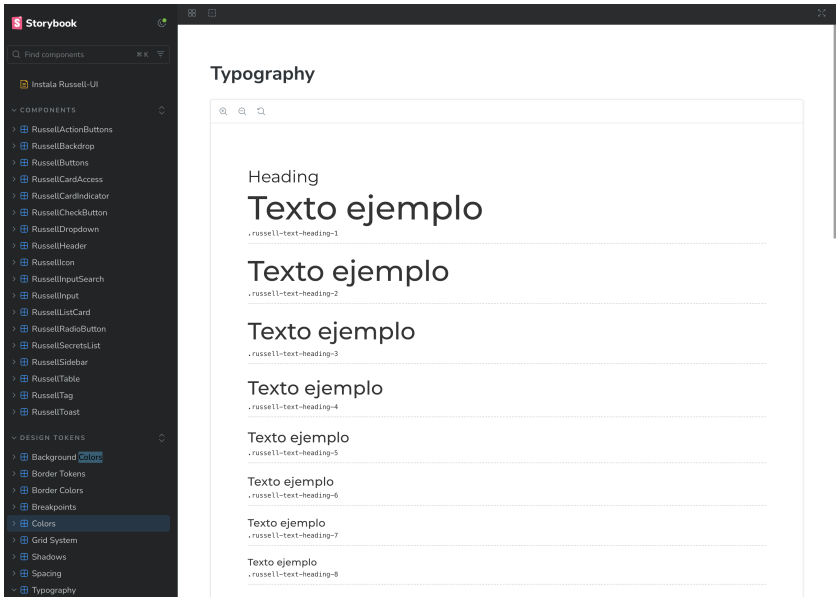


Figura 3.8: Tokens de tipografía sistema de diseño Russell.

De igual forma, el sistema de diseño se aplicó de manera uniforme a los 23 componentes web reutilizables desarrollados para Russell. Entre ellos se incluyen elementos como *dropdowns*, *radiobuttons*, *inputs*, *cards*, *backdrops*, entre otros. Cada componente fue construido siguiendo los principios de escalabilidad, consistencia visual y una **accesibilidad conforme a los estándares WCAG** (Pautas de Accesibilidad para el Contenido Web), haciendo uso directo de los tokens de espaciado, tipografía y color previamente definidos. Esto garantiza que cualquier módulo o vista que utilice estos componentes mantenga una apariencia coherente y alineada con la identidad visual del sistema, además de facilitar el mantenimiento y la evolución del producto a futuro.

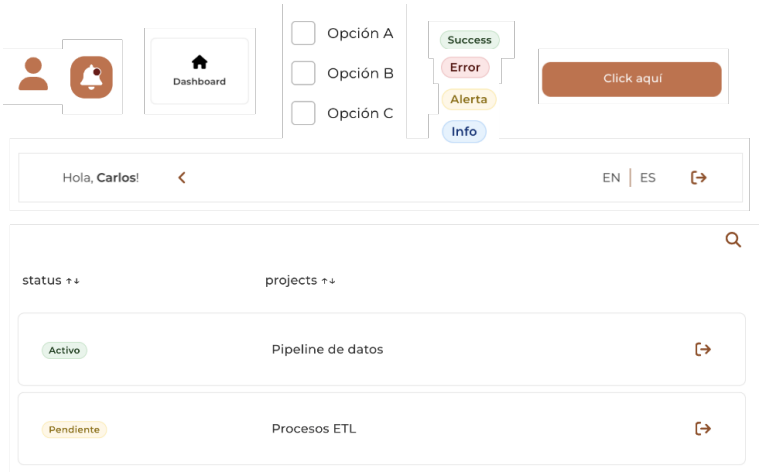


Figura 3.9: Muestra de componentes web reutilizables del sistema de diseño Russell.

Cada componente es documentado adecuadamente, especificando sus propiedades (*props*), eventos y posibles variaciones. En este sentido, Russell puede seguir evolucionando de forma rápida, ya que los componentes están construidos y listos para usarse como se muestra en la Figura 3.10. Lo que permite enfocarse directamente en la lógica de negocio, sin perder tiempo en definir estilos y diseños desde cero.

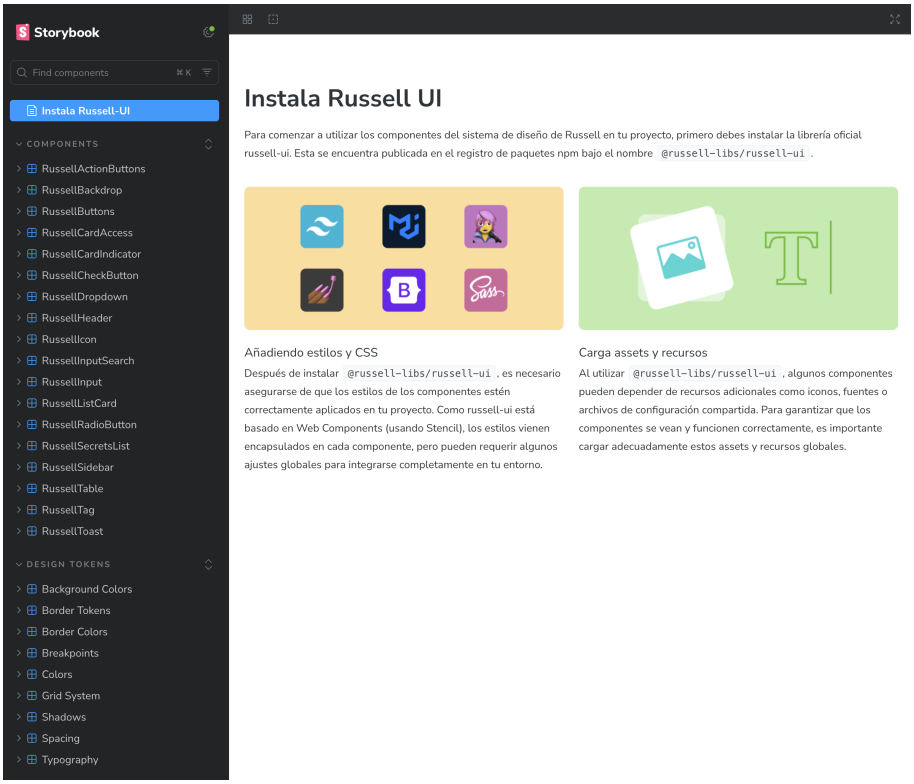


Figura 3.10: Sitio web del sistema de Diseño Russell.

La librería ha sido publicada en el gestor de paquetes *npm*, lo que permite que cualquier persona o equipo de desarrollo pueda instalarla fácilmente y utilizar sus componentes en diversos proyectos. Su distribución mediante *npm* favorece la integración continua y la adopción rápida. Además, la librería se encuentra licenciada bajo el tipo `MIT`, lo que permite su uso, modificación y redistribución con muy pocas restricciones, promoviendo la colaboración y la innovación abierta.

Figura 3.11: Librería publicada en el gestor de paquetes *npm*.

3.3.3.5. Internacionalización de la interfaz con ngx-translate

Con el fin de permitir que la Russell soporte múltiples idiomas y brinde una experiencia más accesible y global, se implementó un sistema de internacionalización (i18n) utilizando la librería `@ngx-translate/core`, junto con el cargador de archivos externo `@ngx-translate/http-loader`.

De esta forma podemos cargar archivos de traducción JSON de forma dinámica a través de peticiones HTTP, lo cual facilita la gestión de múltiples idiomas sin necesidad de recompilar la aplicación ante cada cambio.

La configuración se realiza registrando el módulo de traducción dentro del módulo principal de la aplicación, e indicando que los archivos se deben cargar desde un directorio determinado.

Los archivos de idioma son cargados desde la ruta `assets/i18n/`, y cada archivo corresponde a un idioma específico, por ejemplo: `en.json` para inglés, `es.json` para español, entre otros, Russell solo usa ingles y español.

El uso de traducciones en los componentes de la aplicación se realiza mediante la directiva `translate`, como se muestra en el siguiente ejemplo:

```
# html
<h1>{{ 'TITLE.DASHBOARD' | translate }}</h1>
```

Este sistema facilita la escalabilidad y mantenimiento de la aplicación multilingüe, ya que las traducciones están centralizadas, desacopladas del código fuente, y pueden ser actualizadas o ampliadas sin modificar la lógica de presentación.

3.3.3.6. Arquitectura Backend

En este apartado de la arquitectura del sistema, se abordarán el backend de la solución, la infraestructura sobre la cual se desplegó la solución, configuraciones predefinidas que habilitaron la personalización de Jenkins, una librería de integración para facilitar la interacción machine to machine (backend y Jenkins).

- **Backend (Node.js):** Este es el componente encargado de exponer a través de una API REST que intermedia entre el frontend y Jenkins. Provee funcionalidades transversales que facilitan la gestión, administración usuarios, almacena configuraciones, mantiene la trazabilidad de acciones y ejecuciones, entre otras.

  El backend conceptualmente esta divido en dominios de negocio que exponen y realizan las tareas afines a su responsabilidad definida, internamente estos dominios están a su vez organizados en “controllers” y “routers”, los primeros trazan el comportamiento funcional de una capacidad específica en métodos, por ejemplo, el controlador “credential.controller.ts” se encarga de administrar la gestión de credenciales de GitHub o Amazon Web Services, crear una credencial, actualizarla, eliminarla, listar todas o buscar una en especial por su identificador, los routers mapean estos métodos a verbos HTTP y rutas que se exponen en la API backend de Russell, facilitando la organización e interación de clientes como el frontend propio de

Russell y en el futuro posibles integraciones con terceros.

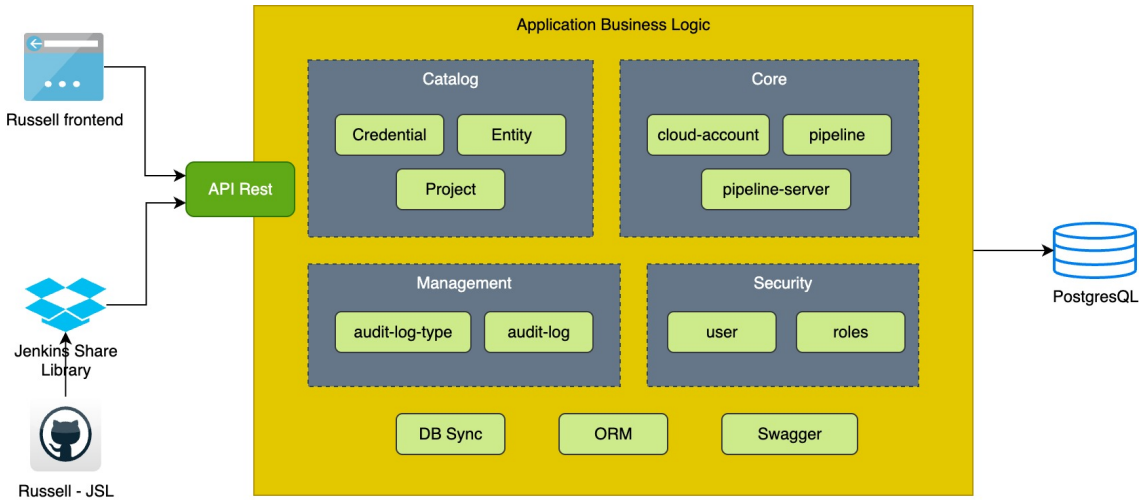


Figura 3.12: Diagrama backend APIs de Russell.

A continuación se lista los dominios de API que componen el backend:

API	Catalog
Qué función cumplirá	Agrupa todos los controladores y métodos asociados a entidades del sistema clasificadas como tipo catálogo, que ayudan a centralizar información estructural de Russell. Estos son: "component-type" administra los tipo de componentes que soporta el gestor de pipelines, "credential" administra todas las credenciales que se pueden asociar a un pipeline, "entity" administra las entidades del sistema susceptibles de auditoría, y "project" administra los proyectos que la organización gestionará usando Russell
Qué restricciones tiene	Dado que son datos muy relevantes del gestor, cada operación debe estar asociada a un verbo HTTP para facilitar su uso, las operaciones deben dejar traza de la operación ejecutada auditando el proceso de paso, cada acción valida la consistencia de los datos registrados.
Atributos de calidad	Mantenibilidad, Auditabilidad

Tabla 3.3: Descripción del API Catálogo.

API	Core
Qué función cumplirá	Por su nombre se puede deducir que es el API más importante para el gestor, en esta API se agrupan todos los controladores y métodos asociados a los pipelines. Estos son: "cloud-account" administra todas cuentas de nube Amazon WS donde se despliegan los componentes de software producto de la ejecución correcta de un pipeline, "pipeline-server" administra todos los servidores maestros de pipelines de Jenkins gestionados desde esta instancia de Russell, "pipeline" administra todos pipelines gestionados por Russell que estarán bajo su gobernanza.
Qué restricciones tiene	Dado que son datos muy relevantes del gestor, cada operación debe estar asociada a un verbo HTTP para facilitar su uso, las operaciones deben dejar traza de la operación ejecutada auditando el proceso de paso, cada acción valida la consistencia de los datos registrados.
Atributos de calidad	Escalabilidad, Disponibilidad

Tabla 3.4: Descripción del API Core.

API	Management
Qué función cumplirá	Bajo esta API se proveerá todos los controladores y metodos asociados a la auditoría del sistema. Estos son: "audit-log-type" administra los tipos de logs soportados por Russell para auditar sus procesos. "audit-log" administra cada una de las acciones efectuadas por los usuarios de Russell durante su operación normal.
Qué restricciones tiene	Cada operación debe estar asociada a un verbo HTTP para facilitar su uso, cada acción valida la consistencia de los datos registrados.
Atributos de calidad	Auditabilidad

Tabla 3.5: Descripción del API Management.

API	Security
Qué función cumplirá	Esta API se encargada de agrupar todos los controladores y métodos relacionados con la seguridad del sistema, Estos son: “user” administra todos los usuarios de Russell además de proveer una camino para autenticar el usuario y proteger el sistema.
Qué restricciones tiene	Cada operación debe estar asociada a un verbo HTTP para facilitar su uso, las operaciones deben dejar traza de la operación ejecutada auditando el proceso de paso, cada acción valida la consistencia de los datos registrados.
Atributos de calidad	Seguridad, Disponibilidad

Tabla 3.6: Descripción del API Security.

Figura 3.13: Documentación API Backend Swagger.

Figura 3.14: Documentación API Backend bajo estándar Swagger.

Aunque estas relaciones e interacciones entre los componentes de las API's quedarán más claros y evidentes en la sección de modelo de datos, Figura 3.30, es momento comprender su funcionamiento y el papel que desempeñan en el gestor, para eso es importante conocer que la base de datos al iniciar la instancia del gestor ya tiene algunos datos precargados con el objetivo de evitar descargar tareas innecesarias a la organización ahorrándole tiempo valioso.

Jenkins tiene un modelo de trabajo que se basa en maestro/esclavo, donde el maestro le delega trabajo al nodo esclavo o nodo agente, pero dependiendo del tamaño de la organización con una instancia Jenkins maestro es suficiente y en otros casos no, por esta razón Russell soporta la configuración de más de un nodo maestro y lograrlo es tan sencillo como agregar un registro nuevo en la tabla "pipelines_servers" (puede hacerse por la API también). También con el objetivo de flexibilizar el modelo, se agrega un tipo de agrupación por proyectos donde se busca que en mismo servidor de Jenkins puedan trabajar equipos independientes y se pueda aprovechar de forma mas eficiente la capacidad instalada, aquí es donde entra en juego la tabla "projects" y estos proyectos a su vez pertenecen a un servidor de Jenkins.

Para contextualizar adecuadamente, por cada registro en la tabla de servidores significa que hay un nodo maestro de Jenkins, los proyectos pertenecen a los servidores y los proyectos se componen de "pipelines", estos pipelines están asociados a repositorios de GitHub los cuales son accedidos por medio de credenciales configuradas por la organización y al ser ejecutados generarán incrementos de software que se despliegan en las "cuentas de nube" asociadas a la rama y proyecto que pertenece el pipeline ejecutado.

Pasando a las tecnologías usadas en para el desarrollo del backend de las API, se utilizó como lenguaje de programación NodeJS v20 que mejora la compatibilidad con módulos ECMAScript (import/export), permitiendo escribir código más moderno e interoperable, además de ser una versión LTS, como framework de trabajo sobre Node se seleccionó Express por su simplicidad y rapidez razón por la cual es uno de los frameworks más populares para Node.js, se utiliza principalmente para construir APIs y aplicaciones web del lado del servidor, aporta un manejo avanzado de rutas, es compatible con arquitecturas modernas y como plus tiene un amplio ecosistema con una comunidad muy activa. Con el objetivo de entregar un producto mantenible en a largo plazo que mejora la productividad y calidad del código, el backend se definió construirse con TypeScript ya que es un superset de JavaScript elaborado por Microsoft que agrega tipado estático entre otras características lo que permite declarar tipos explícitos para variables, funciones, objetos, etc, facilitando así detectar errores en tiempo de desarrollo, antes de ejecutar el código.

Para la integración con base de datos se utilizó Sequelize, que es un ORM muy alineado al lenguaje de desarrollo seleccionado, este ORM permite interactuar con bases de datos SQL como PostgreSQL, MySQL, MariaDB, SQLite y Microsoft SQL Server usando JavaScript/TypeScript en lugar de escribir consultas SQL directamente, esto ayudó a la creación de una abstracción de la base de datos por medio de modelos muy sencillos y claros, este ORM agiliza el desarrollo de los controllers y la construcción de la imagen la base datos a desplegar.

El proyecto de backend también incluye un instalador para crear las imágenes bases de los contenedores que soportan la operación y puesta a punto de Russell, más adelante se detallan estas imágenes y su contenido.

- **Jenkins:** Jenkins es una de las muchas herramientas disponibles para automatizar el proceso de compilación e implementación. Sin embargo, muchas grandes empresas del sector de TI

la eligen para sus procesos de CI/CD porque se clasifica como segura y de código abierto Zhao et al. (2024). Por lo tanto, no se gasta dinero en su uso. Jenkins cuenta con un amplio ecosistema de plugins que facilita su integración con repositorios de código (como GitHub o GitLab), contenedores (Podman), herramientas de testing y plataformas en la nube (como AWS o GCP).

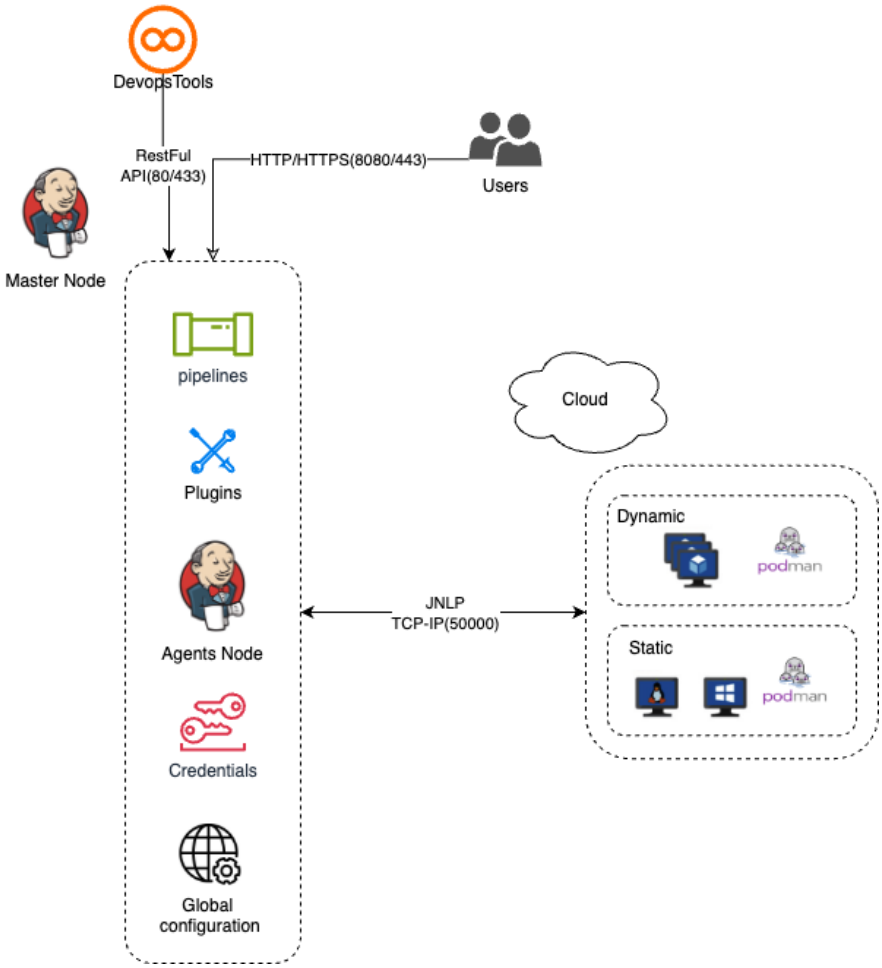


Figura 3.15: Diagrama Jenkins Russell.

Para cumplir con los requisitos de hardware de Jenkins, depende de su uso y del modo en que se utilice. La Tabla 3.7. muestra los requisitos mínimos de hardware para ejecutar un servidor Jenkins:

Componente	**Requisito**
CPU	A Un procesador multinúcleo con una velocidad de reloj de al menos 2 GHz
Memoria (RAM)	Un mínimo de 2 GB de RAM
Espacio en disco	Un minimo de 10 GB

Tabla 3.7: Requisitos mínimos del sistema para la ejecución del gestor de pipelines.

La CPU se ve afectada por la cantidad de peticiones HTTP/HTTPS realizadas al maestro de Jenkins. Estas peticiones HTTP/HTTPS pueden provenir de usuarios que acceden a su maestro de Jenkins o de llamadas a la API realizadas a su maestro de Jenkins mediante webhooks desde la herramienta de control de versiones para activar compilaciones. La siguiente fórmula permite calcular la cantidad estimada de núcleos de CPU necesarios:

$$\text{No de CPU cores} = \frac{\text{No de peticiones HTTPS}}{250}$$

La memoria (RAM) se ve afectada por la cantidad de nodos de agente conectados a su servidor maestro de Jenkins. La siguiente fórmula permite calcular la cantidad estimada de memoria necesaria en megabytes:

$$\text{Cantidad de memoria (MB)} = \text{No de agentes nodo} \times 3$$

La utilización del disco del servidor Jenkins se ve afectada por la cantidad de trabajos de Jenkins Pipeline y sus compilaciones asociadas que se mantengan. Esto afecta si los registros de compilación son grandes. Cabe destacar que el servidor maestro de Jenkins almacena los registros de compilación de Jenkins Pipelines. La siguiente fórmula permite calcular la cantidad estimada de espacio de disco adicional necesario en megabytes:

$$\text{Tam. disco adicional (MB)} = \text{N° jobs} \times \text{N° compilaciones} \times \text{Tam. prom. registros (MB)}$$

El nodo maestro de Jenkins debe tener acceso de red a los nodos de agente, repositorios de código fuente y cualquier otro servicio o sistema que forme parte de su ecosistema de CI. Para alojar un servidor Jenkins, normalmente debe exponer puertos específicos para permitir la comunicación y el acceso. La Tabla 3.8 muestra los puertos que Jenkins utiliza habitualmente:

Puerto	Descripción
8080	Este puerto se utiliza para conectarse a la interfaz web de Jenkins a través de HTTP. Es la interfaz principal para la gestión de Jenkins y las tareas de construcción.
443	Si se habilita HTTPS/SSL para una conexión segura con Jenkins, este puerto se utiliza para acceder a la interfaz web de Jenkins mediante HTTPS.
50000	Este puerto permite la comunicación entre el nodo maestro de Jenkins y los nodos agentes.
22	Puerto predeterminado para SSH que permite a los administradores acceder a la máquina virtual donde se ejecuta Jenkins.

Tabla 3.8: Puertos utilizados por Jenkins y su propósito

- **Infraestructura:** Russell siendo un proyecto de desarrollo de software debe tener una infraestructura sobre la cual se despliega, es fundamental porque proporciona el entorno técnico y operativo sobre el cual se construye, prueba, despliega y mantienen los incrementos de software propios de la organización. Todo Russell se desplegó en una única región con contingencia a nivel de AZs sobre un clúster Amazon ECS, una instancia de base de datos con motor PostgreSQL, los componentes son cargados como servicios dentro del clúster con las imágenes cargadas en servicio Amazon ECR. La API es expuesta en un Amazon API Gateway que enrruta a un balanceador de carga que apunta al servicio expuesto en el cluster.

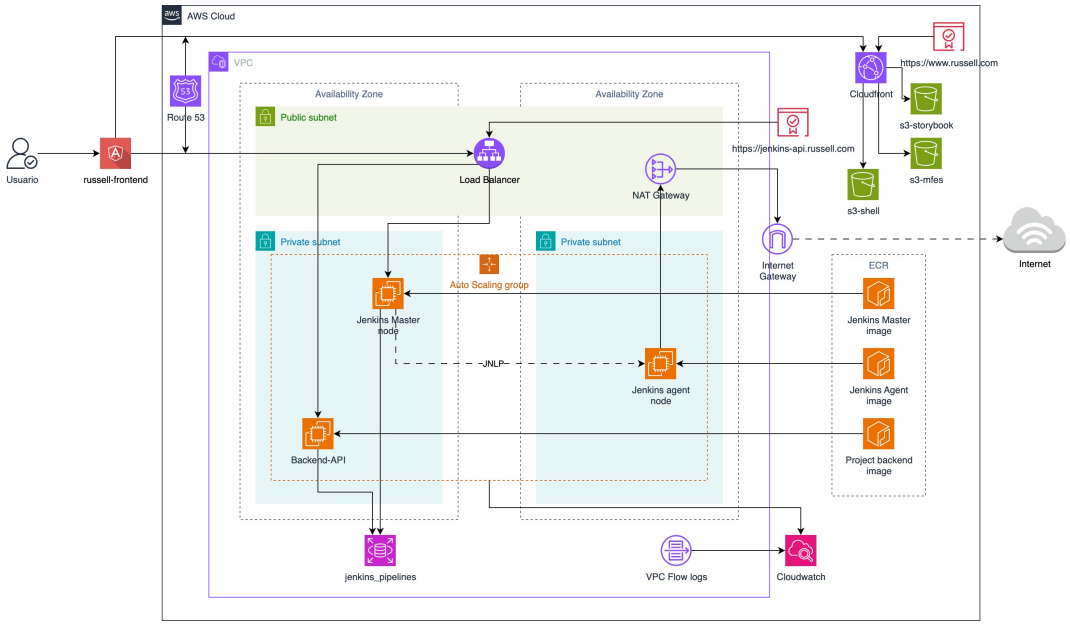


Figura 3.16: Diagrama de despliegue de infraestructura.

Como se puede evidenciar la infraestructura requerida para el despliegue de la solución es bastante robusta y compleja para equipos con conocimientos básicos de nube, por este motivo y con el animo de reducir aún más la carga operativa de la organización acortando el camino por recorrer, la herramienta Russell entrega un componente adicional que es la infraestructura como código, este componente se desarrollo en Terraform y tras unas configuraciones básicas de autenticación sobre la cuenta de la nube que tendrá la infraestructura, este replicará cada uno de los recursos requeridos para el óptimo funcionamiento de la herramienta.

- **Artefactos de despliegue:** En esta sección se abordará los distintos instaladores -herramientas facilitadoras- que Russell incluye para el alistamiento y puesta en marcha del gestor de pipelines. El primero de ellos ya fue referenciado en la sección anterior, la infraestructura como código, esta ofrece la gran ventaja de replicar cuantas veces se desee una misma infraestructura basada en una plantilla, explotando esta idea, la organización que haga uso de Russell puede tener tantos Jenkins maestros como lo requiera las necesidades de negocio. La ejecución del instalador requiere de las credenciales de acceso de Amazon Web Services.

  Inicializa el proyecto de Terraform en el directorio de trabajo:

```
# bash
terraform init
```

  Valida el resumen de lo que Terraform va a hacer sin aplicarlo aún:

```
# bash
terraform plan
```

  Aplica los cambios de plantilla aprovisionando los recursos requeridos por Russell:

```
# bash
terraform apply
terraform apply -auto-approve # Aplica cambios sin preguntar
```

  Una vez la infraestructura sea correctamente aprovisionada, para replicarla en otra cuenta, se sugiere sacar un copia de seguridad del archivo “terraform.tfstate”, generar las credenciales de la nueva cuenta Amazon Web Services y seguir nuevamente los pasos listados anteriormente.

  El siguiente instalador a revisar se encuentra en el repositorio “projects-jenkins-node”, su uso es el mas sencillo de todos, simplemente el usuario se ubica en la raíz del repositorio clonado y realiza el llamado al archivo “setup.sh”.

```
# bash
./setup.sh
```

La salida del comando ejecutado debe verse similar al de imagen a continuación.

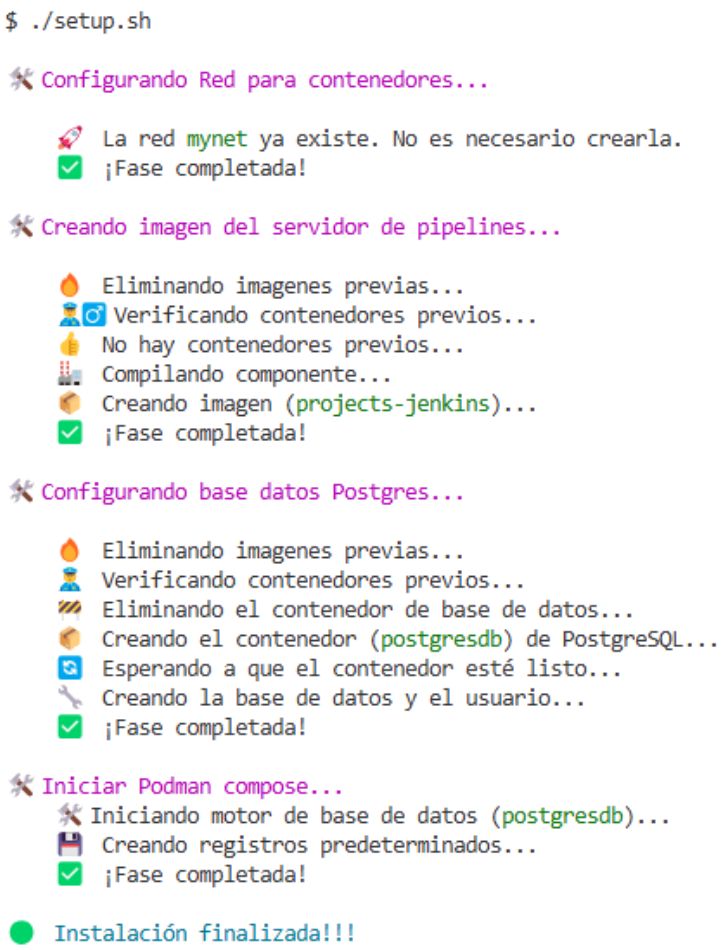


Figura 3.17: Ejecución instalador backend Russell.

Como se puede apreciar, el instalador realiza una serie de tareas programados con scripts en shell, donde configura un ambiente On Premise con todo lo requerido para trabajar con Russell de forma local, crea una red local de contenedores con Podman, limpia el ambiente de imágenes previamente construidas, crea la imagen del backend con las API e incluye el Swagger de documentación y uso de Russell incorporándolo a la imagen (visitar consultar `http://localhost:3000/api-docs/`), seguido de esto configura y crea un contenedor de la base de datos que soporta Russell con el motor PostgreSQL, prepara la base de datos para que el backend se pueda conectar a ella sin intervención humana, una vez esta arriba, crea un contenedor nuevo del backend el cual se encarga conectarse a la base datos, aprovisionar la estructura de tablas y poblar algunas de ellas con datos propios de Russell. A continuación se evidencia los componentes aprovisionados por el instalador, imágenes, contenedores, base datos y prepoblamiento de información.

```
$ podman ps
CONTAINER ID  IMAGE                                COMMAND               CREATED         STATUS         PORTS                   NAMES
55c74a7653a0  docker.io/library/postgres:latest    postgres              38 minutes ago  Up 38 minutes  0.0.0.0:5432->5432/tcp  postgresdb
bca998e06610  localhost/projects-jenkins:latest    sh /app/entrypoin...  38 minutes ago  Up 38 minutes  0.0.0.0:3000->3000/tcp  projects-jenkins
```

Figura 3.18: Contenedores creados por el instalador.

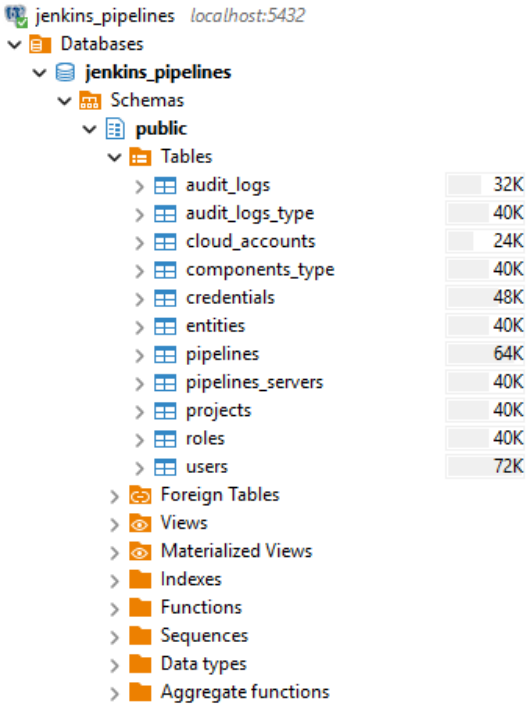


Figura 3.19: Base de datos preconfigurada.

Por último, el instalador da la flexbilidad a la organización de personalizar datos claves del proceso de aprovisionamiento, los datos que se pueden modificar se presentan en la siguiente imagen.

setup.sh

```
41  NETWORK_NAME="mynet"
42
43  DB_CONTAINER_NAME="postgresdb"
44  DB_ADMIN_PASSWORD="mysecretpassword"
45  DB_NAME="jenkins_pipelines"
46  DB_USER="jenkins"
47  DB_USER_PASSWORD="j3nk1n5"
48  DB_PORT=5432
49
50  BACKEND_CONTAINER_NAME="projects-jenkins"
51  BACKEND_CONTAINER_PORT=3000
```

Figura 3.20: Parámetros configurables del instalador.

Esta personalización debe ser realizada antes de mandar a ejecutar el instalador, si y se ha realizado una ejecución previamente, simplemente se requiere volver a lanzar el instalador y este se encargará de limpiar el ambiente y aprovisionar todo nuevamente con la personalización definida.

El último instalador con el que cuenta Russell se encuentra en el repositorio "Jenkins-Master-Russell", este es el encargado de crear la imagen y contenedor del nodo maestro de Jenkins, personalizando la imagen para integrar el componente que habilita la comunicación e interacción con Russell dando vida a este proyecto.

Su ejecución se inicia realizando llamando al archivo "setup.sh" en la raíz del proyecto clonado.

```
# bash
./setup.sh
```

Durante su ejecución, se descarga y actualiza la ultima versión LTS de Jenkins, posterior a esto se limpia el ambiente eliminando imágenes previamente personalizadas.

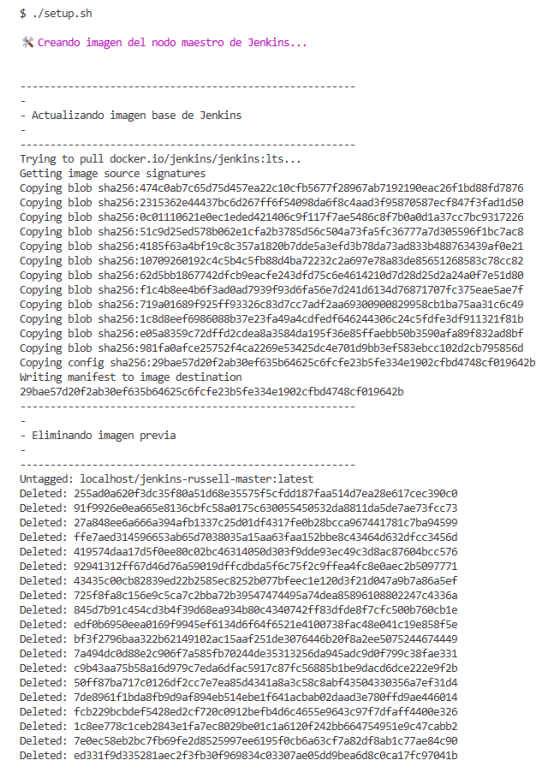


Figura 3.21: Descarga y limpieza del nodo maestro de Jenkins.

Una vez se tiene la imagen preparada se sigue una receta detallada para personalizar el nodo maestro de Jenkins, Podman siguiendo los pasos definidos en un "Dockerfile" en la raíz del repositorio, se encarga de crear la nueva imagen, con los plugins necesarios, la configuración

CASC, configuración base de GitHub, asignación de privilegios al usuario “jenkins” y para el ambiente local crea y ejecuta un contenedor Jenkins al que se puede acceder visitando la URL http://localhost:8080/.

```
-------------------------------------------------------
-
- Creando imagen con la nueva versión
-
-------------------------------------------------------
STEP 1/19: FROM jenkins/jenkins:lts
STEP 2/19: LABEL maintainer="Russell Project"
--> 026c09dd80bd
STEP 3/19: USER root
--> 38cef5f17fe8
STEP 4/19: RUN apt-get update && apt-get upgrade -y &&     apt-get install -y curl &&     curl -fsSL https://deb.nodesource.com/setup_20.x
     npm install -g npm@latest
Get:1 http://deb.debian.org/debian bookworm InRelease [151 kB]
Get:2 http://deb.debian.org/debian bookworm-updates InRelease [55.4 kB]
Get:3 http://deb.debian.org/debian-security bookworm-security InRelease [48.0 kB]
Get:4 http://deb.debian.org/debian bookworm/main amd64 Packages [8793 kB]
Get:5 http://deb.debian.org/debian bookworm-updates/main amd64 Packages [756 B]
Get:6 http://deb.debian.org/debian-security bookworm-security/main amd64 Packages [272 kB]
Fetched 9320 kB in 1s (9262 kB/s)
Reading package lists...
Reading package lists...
Building dependency tree...
Reading state information...
Calculating upgrade...
The following packages will be upgraded:
  libgnutls30
1 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Need to get 1406 kB of archives.
After this operation, 0 B of additional disk space will be used.
Get:1 http://deb.debian.org/debian-security bookworm-security/main amd64 libgnutls30 amd64 3.7.9-2+deb12u5 [1406 kB]
debconf: delaying package configuration, since apt-utils is not installed
```

Figura 3.22: Descarga y limpieza del nodo maestro de Jenkins.

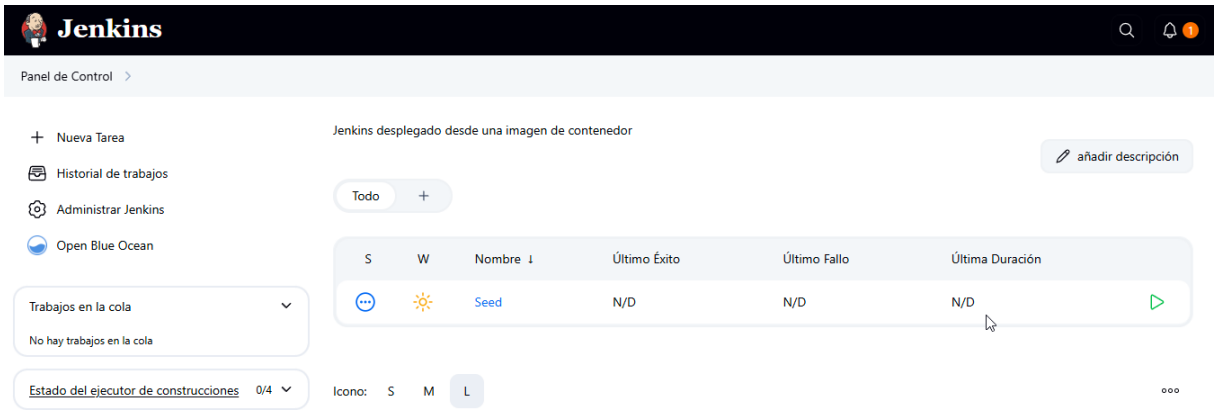


Figura 3.23: Jenkins nodo maestro con pipeline Seed.

Todas la imágenes creadas por los instaladores deben ser cargados a la cuenta de Amazon Web Services si el escenario de uso requerido es diferente a un despliegue On Premise, para realizar esta carga se aconseja seguir la documentación oficial del proveedor de nube.

El frontend de Russell cuenta con un instalador independiente, para hacer uso de este se configuró un script de instalación “setup-frontend.sh” que está dentro del repositorio “projects-jenkins-node”, este instalador se encarga de clonar, instalar dependencias y transpilar el

componente Shell y sus microfrontends dejando al final del proceso dos carpetas con todos los archivos requeridos para un despliegue tanto en local como en nube.

Lanzando instalador del frontend:

```
# bash
./setup-frontend.sh
```

Figura 3.24: Resultado instalador Frontend Russell.

Las carpetas `frontend/shell/` y `frontend/mfes/` son el resultado del proceso de construcción automatizado de Russell frontend, compuesto por un contenedor principal (shell) y cuatro microfrontends (MFEs).

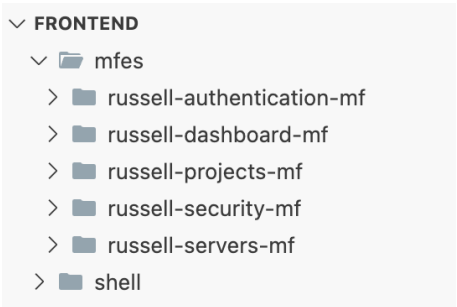


Figura 3.25: Árbol de carpetas generado por el instalador del Frontend Russell.

Estas carpetas contienen los archivos estáticos necesarios para servir la aplicación en un entorno web y poder ser accedida desde un navegador Figura 3.26., permitiendo que los usuarios accedan a la consola de administración Russell una vez que los recursos están disponibles On-Premise o en la nube. La arquitectura basada en microfrontends, permite escalar de forma modular y flexible la solución, con tiempos de carga rápidos y una alta disponibilidad global.

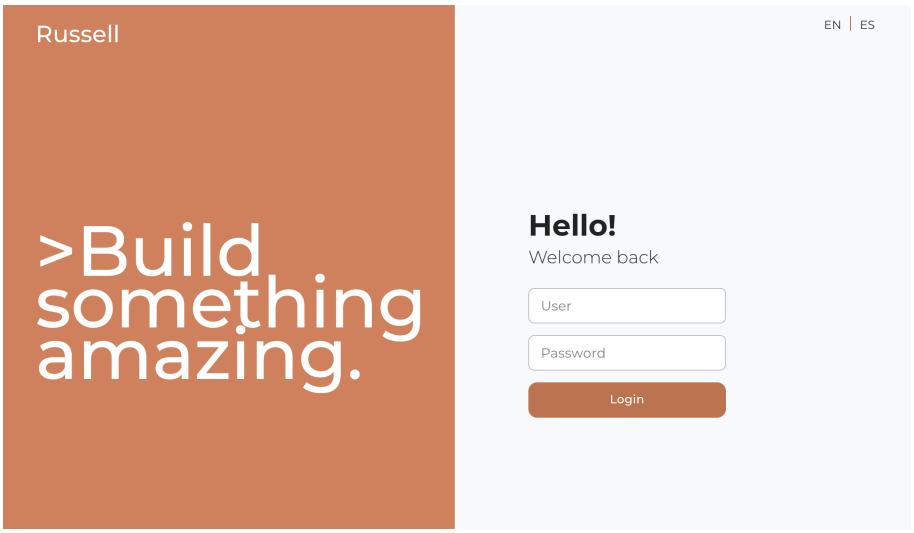


Figura 3.26: Pantalla de acceso Frontend Russell.

- **Artefactos de apoyo:** En este apartado del desarrollo del proyecto se debe realizar una explicación detallada del componente que hace las veces de intermediario, entre lo que define el usuario de Russell a través de las interacciones con el frontend desplegado, y el backend persistiendo dichas configuraciones, plasmando esto en la herramienta CI/CD Jenkins.

  Este componente se llama "JenkinsSharedLibrary", está construido aprovechando las bondades de los plugins de Jenkins en especial el plugin "Job DSL" que permite a los usuarios escribir Job DSL scripts en lenguaje Groovy, dando paso a la creación una librería basada en este sistema que luego es introducida dentro del nodo maestro de Jenkins en la creación de la imagen del contenedor, esta librería realiza tareas muy puntuales pero vitales para el óptimo funcionamiento de Russell, entre ellas se pueden mencionar: cliente REST para comunicarse con el API del backend, creación de proyectos y semillas de los mismos, creación de pipelines asociados a proyectos entre otras.

  Como se menciona en el anterior párrafo, esta librería es incluida en el nodo maestro de Jenkins por medio del archivo "jenkins.yaml", en la sección "globalLibraries", se debe parametrizar la rama (branch) con que se desea trabajar, el repositorio de GitHub donde esta alojada la librería y una credencial válida para acceder al repo, las credenciales deben ser consignadas en el archivo "jenkins.properties", y serán convertidas en credenciales de Jenkins una vez el nodo maestro inicie su ejecución.

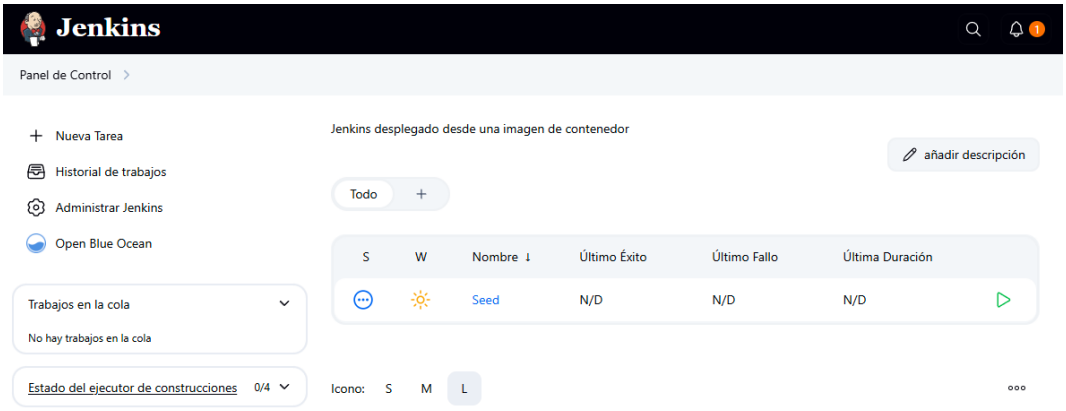


Figura 3.27: Pipeline semilla (Seed) en Jenkins.

El pipeline Seed en Jenkins debe ser ejecutado y puesto en marcha, cuando finalice habrá creado la estructura de proyectos a gestionar en este nodo maestro y los pipeline seed de cada proyecto.

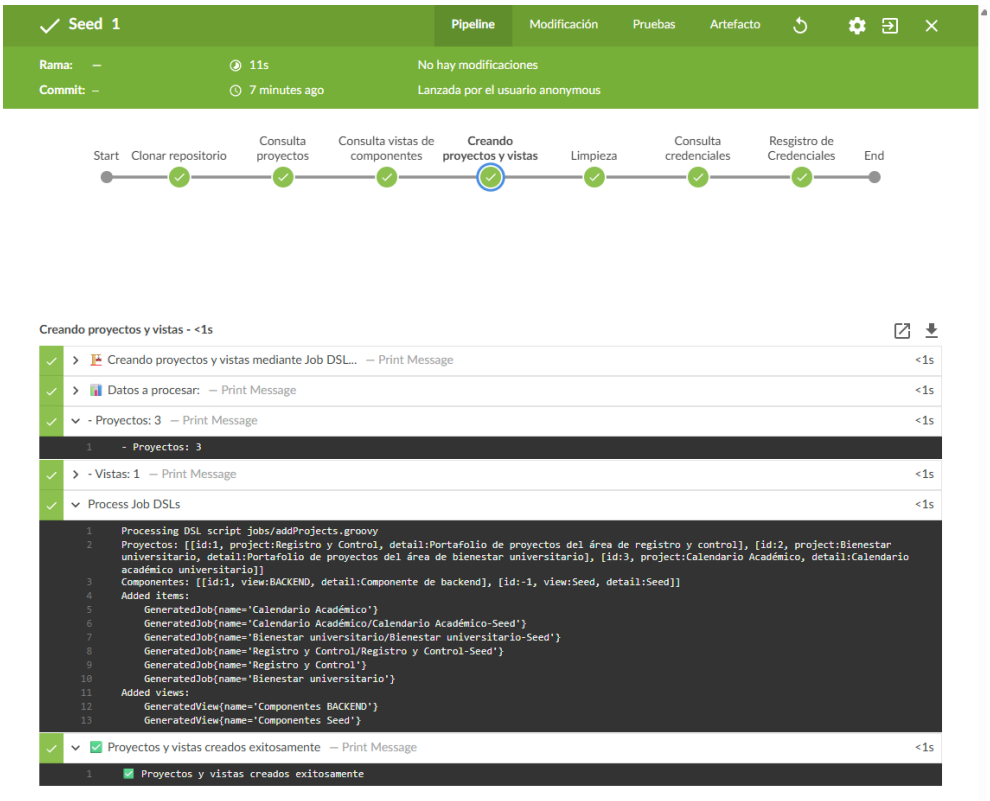


Figura 3.28: Ejecución del pipeline seed en Jenkins.

La ejecución del pipeline “Seed” crea todos los proyectos registrados en la base de datos.

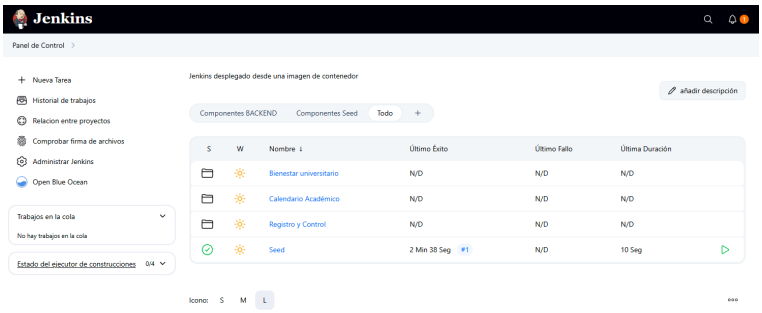


Figura 3.29: Jerarquía de proyectos en nodo maestro de Jenkins.

Al ingresar a cada carpeta de proyecto, internamente debe existir un job propio asociado a ese proyecto con la capacidad de crear todos los pipelines pertenecientes registrado bajo el proyecto.

3.4. Modelo de datos

Russell requiere persistir toda la información relacionada con la definición, ejecución y seguimiento de pipelines. Para ello, se diseñó un modelo de datos que permite representar las entidades principales y sus relaciones, asegurando integridad, disponibilidad, trazabilidad y escalabilidad del sistema.

3.4.1. Diagrama de entidad y relación

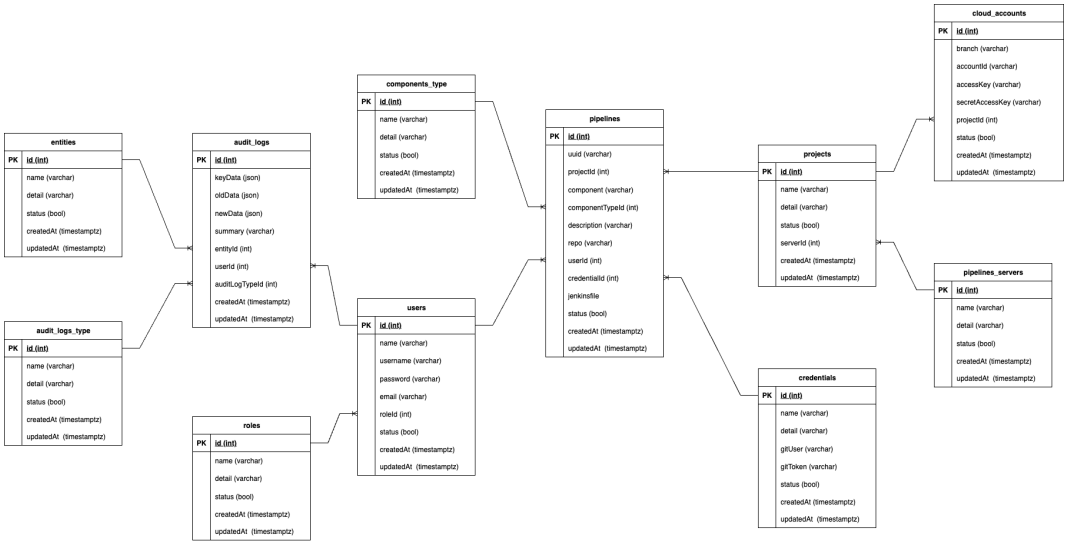


Figura 3.30: Diagrama entidad relación de Russell.

3.4.2. Diccionario de Datos

El siguiente diccionario de datos presenta las principales tablas utilizadas en el sistema gestor de pipelines, junto con una breve descripción de su propósito:

Tabla	Descripción
audit_logs_type	Tipos de auditoría registrados en el sistema para clasificar acciones de usuarios (crear, leer, actualizar, eliminar).
entities	Entidades del sistema representadas como tablas principales en la base de datos.
roles	Define los roles de usuario que gestionan los niveles de acceso y permisos dentro de la plataforma.
users	Almacena la información de los usuarios del sistema, sus credenciales y su relación con los roles.

Continúa en la siguiente página

Tabla	Descripción
components_type	Clasificación de componentes según su naturaleza (backend, frontend, infraestructura).
credentials	Credenciales para acceder a repositorios de código, necesarias para la ejecución de pipelines.
pipelines_servers	Servidores Jenkins registrados en el sistema, habilitados para la ejecución de pipelines.
projects	Proyectos registrados en el sistema, agrupando pipelines relacionados con áreas específicas.
cloud_accounts	Información de cuentas en la nube, asociadas a proyectos y ramas específicas para despliegue.
pipelines	Configuración detallada de cada pipeline, incluyendo su repositorio, credenciales y parámetros de ejecución.

Tabla 3.9: Diccionario de datos: Tablas principales del gestor de pipelines

3.4.3. Selección del Motor de Base de Datos: PostgreSQL

Para el desarrollo del presente proyecto se seleccionó PostgreSQL como motor de base de datos, esta decisión se basa principalmente en que es un sistema de gestión de bases de datos relacional (RDBMS) open source, soportando un modelo de uso en nube o local para organizaciones con menor músculo financiero, que tendrían la posibilidad de realizar una instalación On Premise, su estabilidad comprobada en entornos de producción y su amplia comunidad de soporte.

PostgreSQL al soportar un modelo de datos relacional es perfecto para el modelo de datos propuesto de persistencia para Russell. Con el ánimo de en un futuro inspeccionar continuamente la calidad de código fuente que pasa por los pipelines gestionados por Russell, la herramienta más utilizada por la comunidad es SonarQube, y en su documentación oficial el único motor que cubre las necesidades de estar diseñado para ambientes productivos y soporte a estándares abiertos (sin costos ocultos ni restricciones de uso) es PostgreSQL.

Capítulo 4

Pruebas y análisis de resultados

En esta sección se abordará la configuración de un pipeline con el objetivo de comprender las funcionalidades que ofrece el sistema y las interacciones que realiza el usuario para hacer un uso de Russell.

Se presentará un recorrido por la solución, identificando los pasos necesarios para configurar, ejecutar y monitorear un pipeline, así como los elementos clave de la interfaz que intervienen en este proceso.

4.1. Recorrido funcional: configuración y uso de un pipeline en Russell

4.1.1. Ingreso a la plataforma

Una vez autenticado, el usuario accede a Russell a través del microfrontend de autenticación. El sistema valida el rol del usuario y redirige a la vista principal según sus permisos. En la vista principal se ofrece al usuario en sesión un resumen del estado de la configuración general de Russell en base de datos, así como el listado de las ultimas ejecuciones gestionadas en Jenkins. Usuarios con perfil de tipo *Admin* o *DevOps* pueden visualizar y administrar los pipelines existentes.

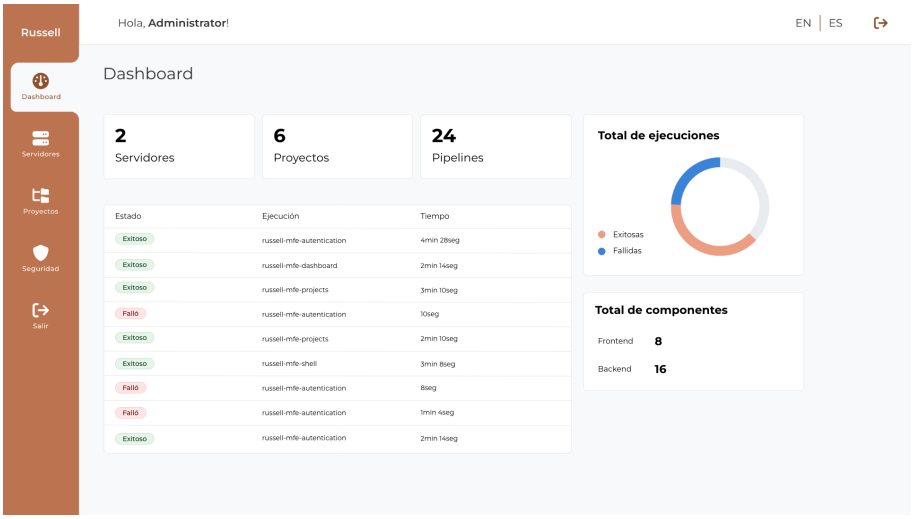


Figura 4.1: Vista principal del dashboard de Russell con lista de pipelines.

4.1.2. Configuración de Servidores

Russell permite al usuario crear nuevos servidores como parte de la configuración del entorno de ejecución.

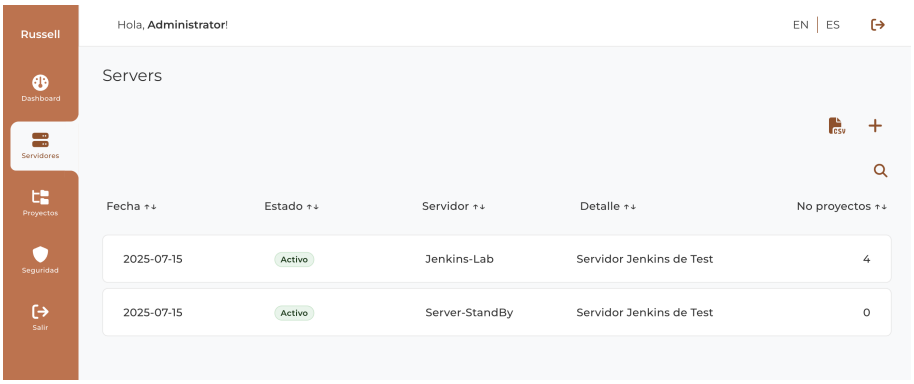


Figura 4.2: Vista del listado de servidores creados en Russell.

Figura 4.3: Vista creación de servidores en Russell.

Una vez completado el formulario, el servidor queda registrado en la base de datos del sistema y puede ser referenciado al momento de configurar un proyecto, o levantar un nuevo nodo maestro de Jenkins.

4.1.3. Configuración de Proyectos

Russell permite estructurar los pipelines dentro de proyectos, lo cual facilita la organización lógica de las ejecuciones según el contexto funcional, el área de desarrollo correspondiente o la iniciativa a gestionar. Un proyecto puede representar una aplicación, un equipo de trabajo o una unidad de negocio específica.

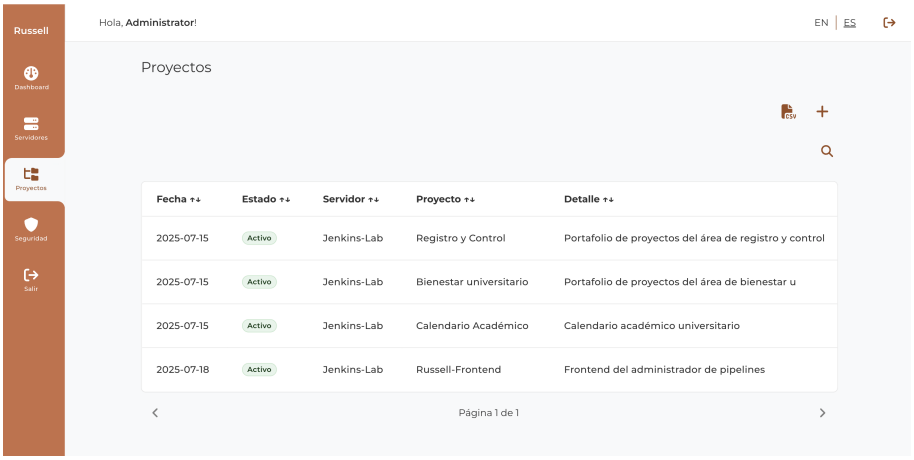


Figura 4.4: Vista del listado de proyectos en Russell.

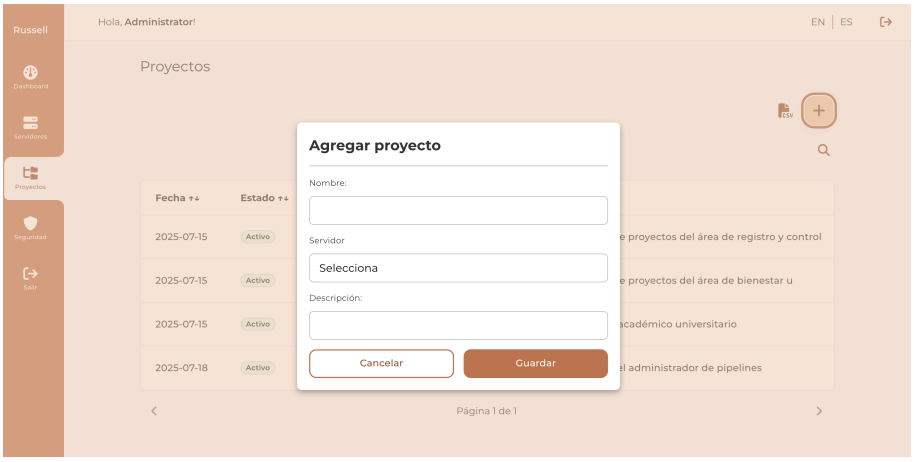


Figura 4.5: Vista creación de proyectos en Russell.

4.1.4. Configuración de Pipelines

Una vez creado un proyecto en la plataforma Russell, los usuarios con los permisos adecuados pueden configurar pipelines específicos asociados a dicho proyecto. Esta funcionalidad permite mantener una trazabilidad clara entre los artefactos de automatización y el contexto funcional al que pertenecen, lo que resulta fundamental para la gestión de entornos complejos con múltiples aplicaciones y equipos de trabajo.

Para iniciar la creación de un pipeline, el usuario debe acceder al detalle del proyecto y seleccionar la opción *Agregar pipeline*. A partir de allí, se despliega un formulario que solicita información como el nombre del pipeline, el tipo de componente a gestionar (frontend o backend), la rama del repositorio, el servidor asociado y las variables personalizadas requeridas para la ejecución.

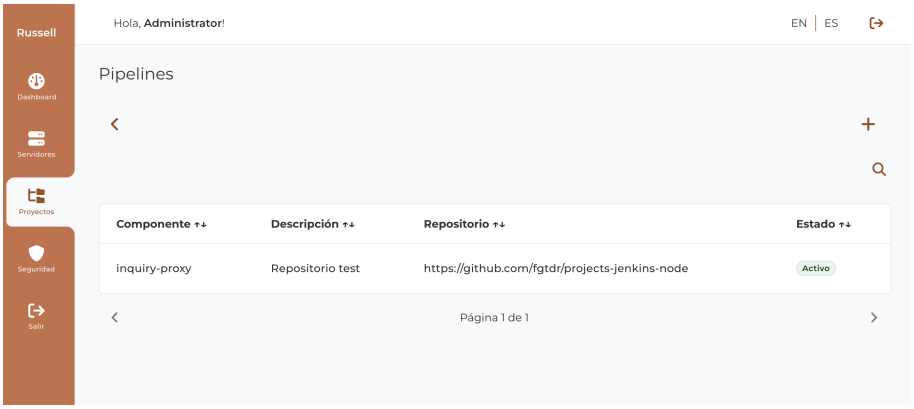


Figura 4.6: Vista del listado de pipelines.

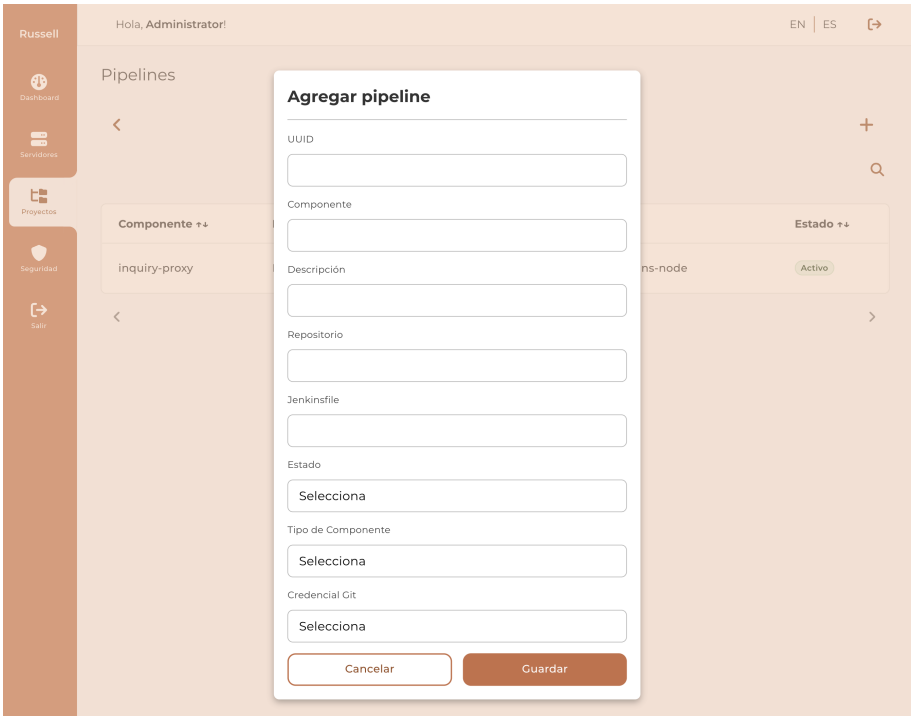


Figura 4.7: Vista creación de un nuevo pipeline.

4.2. Módulo de seguridad: gestión de usuarios, credenciales y auditoría

El módulo de seguridad de la plataforma Russell permite administrar aspectos fundamentales relacionados con la identidad de los usuarios, el control de acceso y la configuración de credenciales necesarias para la operación de los pipelines e interacción con servicios externos (GitHub,

AWS, etc.). Este módulo constituye un componente crítico dentro del sistema, ya que garantiza la trazabilidad, el cumplimiento de permisos y la protección de recursos sensibles.

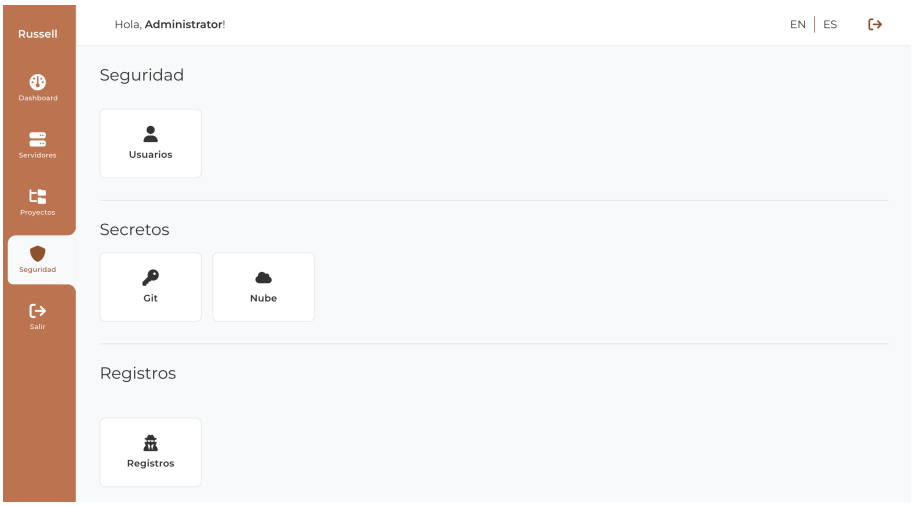


Figura 4.8: Vista del modulo de seguridad de Russell.

4.2.1. Creación y gestión de usuarios

Los usuarios con rol de *administrador* pueden registrar nuevos usuarios en la plataforma, asignarles un rol (por ejemplo: *admin*, *DevOps* y *auditor*). Este proceso es fundamental para aplicar un modelo de control de acceso basado en roles, donde cada usuario accede únicamente a las funcionalidades autorizadas por su perfil.

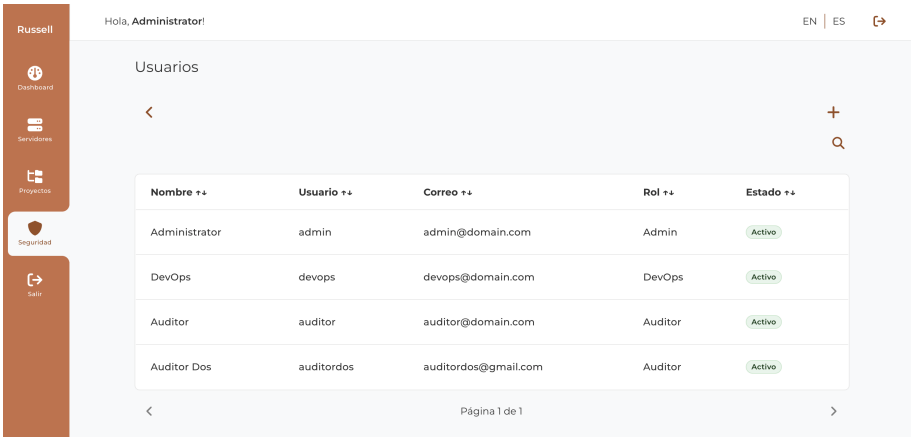


Figura 4.9: Vista del listado de usuarios creados en Russell.

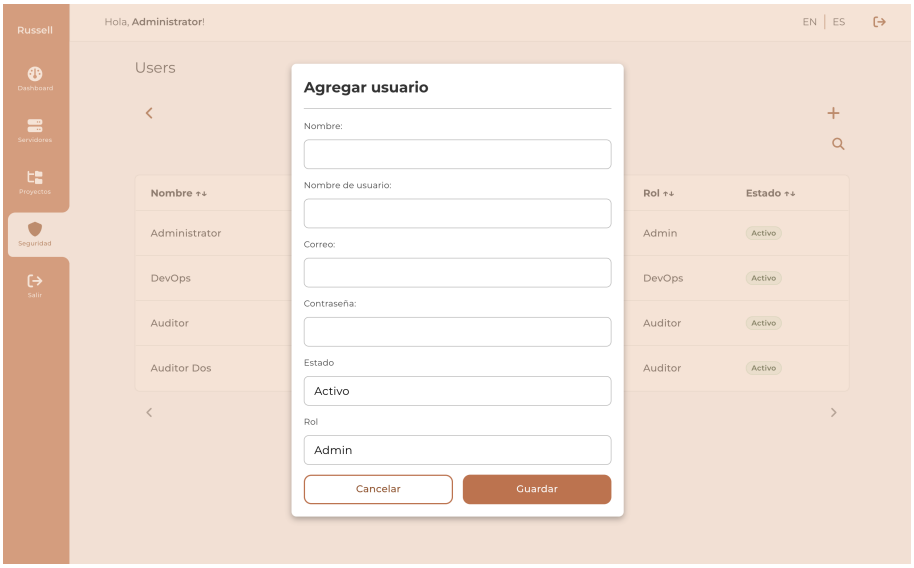


Figura 4.10: Vista creación de un nuevo usuario en Russell.

4.2.2. Gestión de Secretos

Este tiene como propósito proteger y centralizar la información sensible utilizada por los pipelines. Entre los secretos gestionados se incluyen credenciales de acceso a sistemas externos (por

ejemplo, tokens de *GitHub*, llaves de cuentas de nube).

El uso de secretos garantiza el cumplimiento de buenas prácticas de seguridad en la automatización de despliegues y ejecución de tareas sensibles.

4.2.3. Gestión de Credenciales Git

Este módulo permite la configuración centralizada de credenciales que serán utilizadas para acceder a repositorios Git (por ejemplo, GitHub). Cada credencial puede estar asociada a una organización o proyecto dentro de Russell, y se utiliza en tareas como:

- Clonado de repositorios.
- Publicación de cambios en ramas específicas.
- Acceso a archivos de configuración y artefactos.

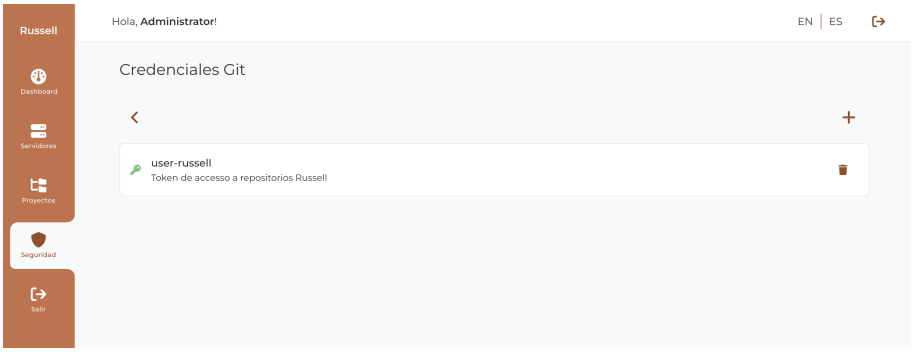


Figura 4.11: Listado de credenciales Git en Russell.

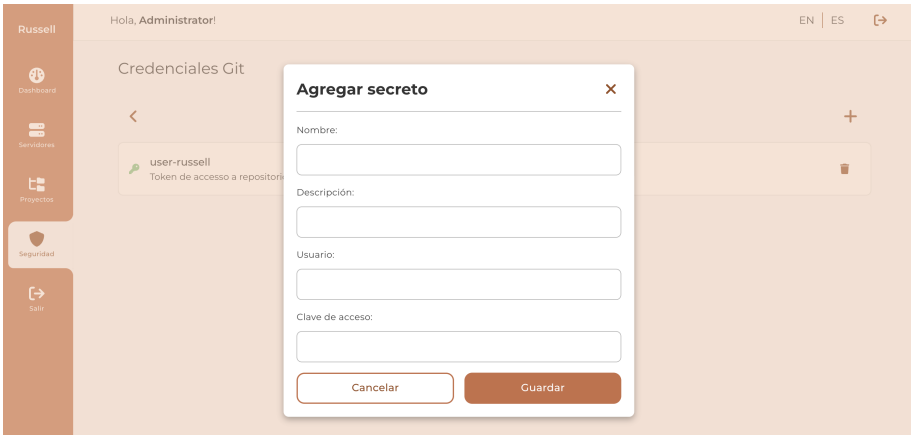


Figura 4.12: Vista para agregar credenciales Git en Russell.

4.2.4. Secretos para la Nube

Russell ofrece integración con plataformas en la nube, como AWS, facilitando la configuración de credenciales para ejecutar despliegues y tareas remotas. El módulo de secretos en la nube permite registrar y gestionar claves de acceso (*Access Key ID* y *Secret Access Key*). Otro ítem importante a resaltar, es que permite la configuración de este secreto a una rama específica (branch de GitHub), lo que brinda la posibilidad de crear ambientes de despliegue independientes (desarrollo, pruebas, producción) acorde a las necesidades del proyecto o la organización.

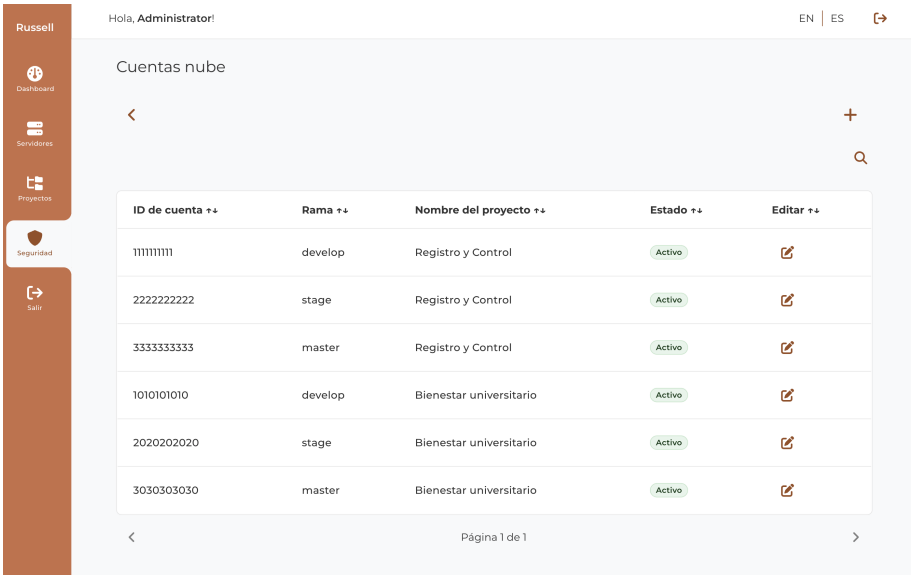


Figura 4.13: Listado de credenciales cloud en Russell.

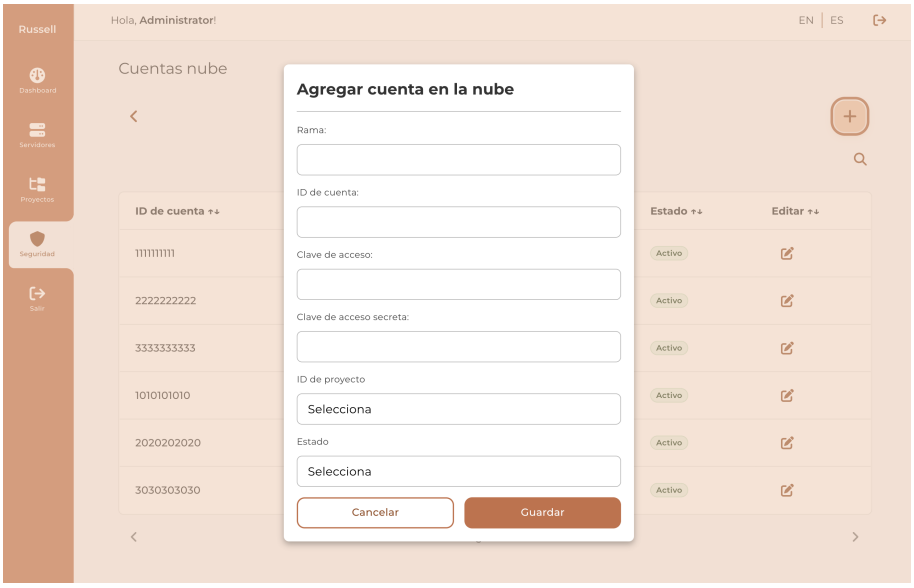


Figura 4.14: Vista para agregar credenciales cloud en Russell.

4.2.5. Registro de Logs

El módulo de registro de logs en Russell permite trazar y auditar la interacción de los usuarios con los módulos de configuración. Su propósito principal es garantizar la trazabilidad, facilitar la auditoría.

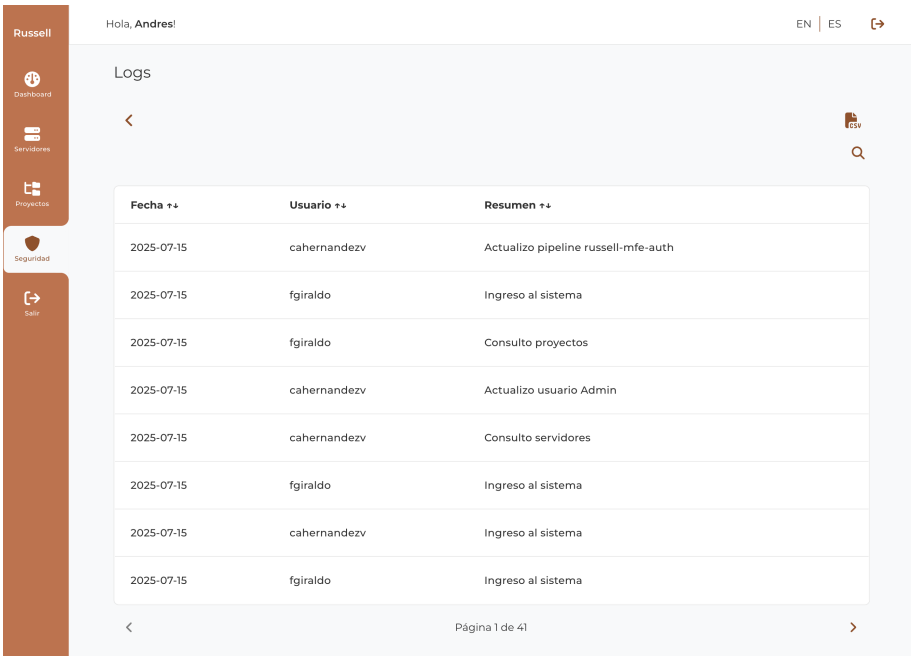


Figura 4.15: Listado de registro de auditoria en Russell.

4.2.6. Módulo de Exportación a CSV

El módulo de exportación a **CSV** en **Russell** permite a los usuarios generar reportes descargables a partir de los datos visualizados en la interfaz.

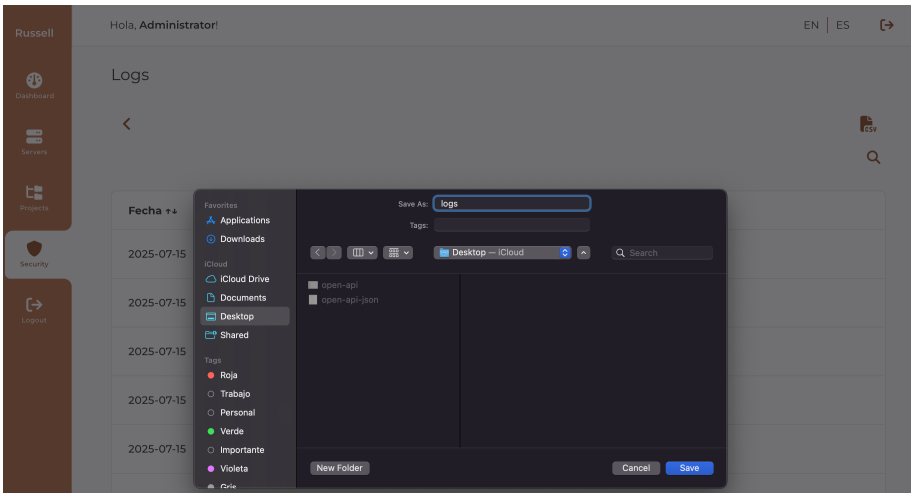
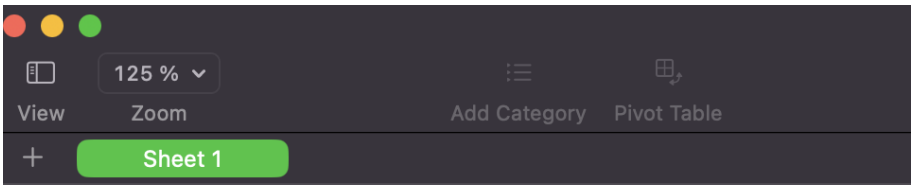


Figura 4.16: Exportar registro de auditoria en Russell.



logs

Fecha	Usuario	Tipo	Resumen
2025-07-15	cahernandezv	Login	Ingreso al sistema
2025-07-15	fgiraldo	Login	Ingreso al sistema
2025-07-15	fgiraldo	Actualización	Actualizo pipeline demo
2025-07-15	cahernandezv	Login	Ingreso al sistema
2025-07-15	fgiraldo	Login	Ingreso al sistema

Figura 4.17: Visualización de registros de auditoria en formato CSV.

Capítulo 5

Conclusiones

5.1. Conclusiones

Se identificaron las principales prácticas y estándares en la implementación de pipelines de integración y entrega continua (CI/CD), lo cual permitió adoptar enfoques alineados con los modelos actuales de automatización.

Se evaluaron diferentes herramientas y tecnologías del ecosistema DevOps, seleccionando aquellas que ofrecieran flexibilidad, estabilidad, bajo costo pero sobre todo con una curva de aprendizaje baja en complejidad. La selección de Jenkins como motor de automatización permitió aprovechar una solución open source, ampliamente respaldada por la comunidad para desarrollar una solución sin incurrir en costos por licenciamiento del software.

Jenkins como herramienta destaca por su flexibilidad, extensibilidad mediante plugins y su capacidad de integración con múltiples servicios del ecosistema DevOps, lo que la convierte en una opción robusta para la orquestación de pipelines. Durante el desarrollo, fue necesario adquirir conocimientos técnicos específicos para su configuración y operación, lo que evidencia la importancia de una curva de aprendizaje bien gestionada en un proceso de adopción de practicas DevOps. No obstante, su adopción permitió implementar procesos automatizados alineados con los estándares de integración y entrega continua. Es importante señalar que, aunque Jenkins no representa un costo directo por uso en una infraestructura On Premise, su despliegue en entornos de nube sí implica gastos asociados al consumo de recursos de infraestructura propia del proveedor, los cuales deben ser considerados dentro del modelo operativo para determinar el trade-off del sabor solución a implementar.

La adopción de una arquitectura microfrontends permitió dividir Russell en módulos independientes que pueden evolucionar de forma autónoma, aunque implico desafíos técnicos al principio -especialmente en la integración y la comunicación entre ellos-, los beneficios en términos de escalabilidad, organización y claridad en la arquitectura superaron los retos. Hoy podemos decir que fue una elección que aporto robustez y flexibilidad al sistema.

En términos de seguridad, el implementar un sistema autorización, con control de acceso basado en roles fue vital para proteger la configuración de los pipelines y garantizar la gobernanza sobre las acciones críticas dentro del sistema, marcando límites claros a los usuarios en cada interacción.

Utilizar Groovy en Jenkins permite definir flujos de CI/CD (pipelines) de forma clara, programable y fluida, facilitando la organización por stages (etapas) y steps (pasos) permitiendo crear un flujo a la medida mediante estructuras de control (if, for, try/catch), pero la característica que mas valor aportó y se explotó en este proyecto fue la reutilización de código a través de bibliotecas compartidas (Shared Libraries).

JCasC inicialmente fue tomado a la ligera, pero su mayor cualidad radicaba en la reproducción idempotente de cada servidor de nodo maestro de Jenkins que generaba para levantar el ambiente, redujo el trabajo de configuración enormemente, tanto así que se integro dentro los artefactos de despliegue de Russell.

Durante la creación de los artefactos de despliegue, se evidenció que la infraestructura necesaria para poner en marcha la solución en un entorno productivo o de pruebas (PoC) también requería un esfuerzo considerable por parte del usuario. Para facilitar este proceso, se decidió incluir dicha infraestructura bajo el enfoque de Infraestructura como Código (IaC), utilizando Terraform, a pesar de que no estaba contemplado dentro del alcance original del proyecto.

Para la internacionalización se utilizó la librería ngx-translate, lo que permitió gestionar los textos dinámicamente y facilitar la incorporación de múltiples idiomas sin modificar la estructura del código. Esta estrategia no solo promovió una experiencia de usuario inclusiva y adaptable a diferentes contextos, sino que también facilitó el mantenimiento y la escalabilidad de la solución.

El uso de un sistema de diseño propio contribuyó de manera significativa a garantizar la consistencia visual, la accesibilidad y la mantenibilidad de la herramienta. Este enfoque permitió abstraer decisiones de diseño en tokens semánticos reutilizables —como color, tipografía, espaciado y comportamiento de componentes— lo que facilitó una implementación uniforme y coherente en toda la interfaz de usuario.

Además, el sistema de diseño potenció la escalabilidad del producto al permitir la evolución de la interfaz sin introducir inconsistencias visuales ni técnicas. Esta base sólida no solo mejoró la experiencia de uso de Russell, sino que también agilizó el desarrollo de nuevos módulos, promoviendo la coherencia visual, la eficiencia en el mantenimiento y la sostenibilidad del sistema en el tiempo.

5.2. Trabajos futuros

Como líneas de trabajo futuro, se plantea la incorporación de integraciones nativas con proveedores de servicios en la nube como Azure y GCP, con el fin de ampliar las capacidades de despliegue de Russell en entornos multinube.También se sugiere la implementación de un editor visual para la creación de pipelines mediante YAML, que facilite aún más su configuración para usuarios no técnicos. Desde el punto de vista de seguridad y gobernanza, se considera valioso incluir control de

versiones sobre los pipelines, así como un esquema de permisos más granular para gestionar accesos por proyecto, entorno o etapa. Finalmente, una línea prometedora consiste en la incorporación de inteligencia artificial y agentes inteligentes capaces de asistir en tiempo real a los usuarios, ya sea sugiriendo configuraciones óptimas, anticipando errores potenciales o automatizando tareas repetitivas dentro del flujo de trabajo, con el objetivo de aumentar la eficiencia, reducir errores humanos y fortalecer la experiencia de uso de la herramienta.

5.3. Lecciones aprendidas

A lo largo del desarrollo de Russell, adquirimos valiosas lecciones tanto técnicas como humanas. Comprendimos la importancia de una arquitectura modular y escalable, que permita la evolución del sistema sin comprometer su estabilidad. Aprendimos que la automatización no solo requiere herramientas robustas, sino también una planificación clara y procesos bien definidos. La implementación de un sistema de diseño nos enseñó que la coherencia visual y la accesibilidad no son aspectos secundarios, sino pilares fundamentales para construir productos sostenibles. Además, reconocimos el valor de adoptar soluciones open source como Jenkins, que, si bien reducen barreras económicas, exigen decisiones conscientes en cuanto a infraestructura y mantenimiento. Finalmente, el trabajo en equipo y la comunicación constante fueron determinantes para alinear criterios, resolver obstáculos y llevar la idea desde su concepción hasta una solución funcional, reafirmando que el desarrollo de software es tanto un desafío técnico como un ejercicio de colaboración efectiva.

Bibliografía

Atlassian (2024). Bamboo: Continuous integration and deployment.

Atlassian (s.f.a). Devops.

Atlassian (s.f.b). Kanban - a brief introduction.

Australia, K. (2019). The future of project management: Global outlook 2019 kpmg, aipm and ipma project management survey 2019 a connected future. *Report KPMG, AIPM and IPMA*.

AWS (s.f.). Continuous delivery.

Boehm, B., Lane, J. A., Koolmanojwong, S., and Turner, R. (2014). *The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software*. Addison-Wesley Professional, Boston.

Chiari, M., Xiang, B., Canzoneri, S., Nedeltcheva, G., Nitto, E. D., Blasi, L., Benedetto, D., Niculut, L., and Škof, I. (2024). Doml: A new modeling approach to infrastructure-as-code. *Information Systems*, 125.

CI, T. (2024). Travis ci product.

CircleCI (2024). About circleci.

Cockcroft, A. (2013). Velocity and volume (or speed wins). http://flowcon.org/dl/flowcon-sanfran-2013/slides/AdrianCockcroft_VelocityAndVolumeorSpeedWins.pdf. Accessed: 2024-09-03.

Debois, P., Kim, G., Willis, J., and Humble, J. (2021). *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*. O'Reilly Media, 2nd edition.

Dmitry, N. and Manfred, S.-S. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27.

Docker (2024). What is a container?

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer.

Farley, D. (2023). *Continuous Delivery Pipelines: How to Build Better Software Faster*. IT Revolution Press, Portland, Oregon.

Forsgren, N., Humble, J., and Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press, Portland, Oregon, first edition.

Geers, M. (2020). *Micro Frontends in Action*. Simon and Schuster.

GitLab (2024). Devops topics.

IBM (2024). Continuous deployment. Accedido el 3 de mayo de 2025.

Jenkins (2024). Jenkins documentation.

Majowska, A. (2014). *Kanban: What is it?* Shore Labs.

Martínez, A. (2024). Drupal development process using the atomic design methodology. Consultado el 12 de mayo de 2025.

Microsoft (2024). What is azure devops?

Mohammad, S. M. (2016). Continuous integration and automation. *International Journal of Creative Research Thoughts (IJCRT)*, 3(Jul):2320–882.

Nicole Forsgren, Jez Humble, G. K. A. B. (2015). 2015 state of devops report. https://services.google.com/fh/files/misc/state-of-devops-2015.pdf. Accessed: 2024-09-03.

PageGroup (2024). Estudio de remuneración tecnología 2024.

Pathania, N. (2024). *Learning Continuous Integration with Jenkins - Third Edition*. Packt Publishing.

Services, A. W. (2024). Aws codepipeline user guide.

Shahin, M., Ali Babar, M., and Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, PP.

Smith, G. et al. (2023). *Modern DevOps Practices: Implementing scalable, sustainable, and secure DevOps pipelines*. Packt Publishing, Birmingham, UK.

Stackify (2017). Ci/cd tools throwdown: Jenkins vs. teamcity vs. bamboo.

Tidwell, J., Brewer, C., and Valencia, A. (2020). *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly Media, 3rd edition.

Ugwueze, V. and Chukwunweike, J. (2024). Continuous integration and deployment strategies for streamlined devops in software engineering and application delivery. *International Journal of Computer Applications Technology and Research*, pages 1–24.

van Merode, H. (March 2023). *Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines*. Apress, New York.

Zhao, X., Clear, T., and Lal, R. (2024). Identifying the primary dimensions of devsecops: A multivocal literature review. *Journal of Systems and Software*, 214:112063.

Apéndice A

Anexos

A.0.1. Requerimientos funcionales

El listado completo de requerimientos funcionales definidos para el sistema **Russell** se encuentra disponible en el siguiente enlace:

- **Enlace:** https://goo.su/9PnsNT
- **Descripción:** Documento en formato PDF que detalla todos los requerimientos funcionales del sistema.

A.0.2. Requerimientos no funcionales

El documento completo que describe los requerimientos no funcionales del sistema **Russell** puede ser consultado en el siguiente enlace:

- **Enlace:** https://goo.su/SJFMd
- **Descripción:** Documento en formato PDF que incluye los requerimientos clasificados por categorías como rendimiento, escalabilidad, disponibilidad, seguridad, integración, usabilidad y auditabilidad.

A.0.3. Tablero de gestión de tareas

El tablero completo con la planificación detallada de tareas del sistema **Russell**, incluyendo subtareas, asignaciones, fechas, prioridades y categorías, puede ser consultado en el siguiente enlace:

- **Enlace:** https://goo.su/6J1aaj
- **Descripción:** Tablero en Notion que presenta las actividades distribuidas por iteraciones. Cada tarjeta incluye responsable asignado (Francisco G o Andrés H), prioridad (Alta, Media o Baja).

A.0.4. Registros de decisiones de arquitectura (ADR)

Los Architecture Decision Records (ADR) documentan las decisiones técnicas clave tomadas durante el diseño y desarrollo del sistema **Russell**. Estas decisiones incluyen la elección de tecnologías, estrategias de integración, patrones de seguridad y otras decisiones relevantes que afectan la arquitectura del sistema.

A continuación, se proporciona un enlace al repositorio que contiene los ADR utilizados en el proyecto:

- **Enlace:** https://goo.su/c8CbYyJ
- **Descripción:** Archivo PDF con las decisiones registradas, incluyendo título, contexto, decisión tomada, y consecuencias. Se incluyen decisiones como la adopción de AWS, PostgreSQL, Angular, Jenkins, y StencilJS.

A.0.5. Documentación del API con Swagger

La API del sistema **Russell** ha sido documentada utilizando la herramienta **Swagger**, que permite una representación clara, interactiva y mantenible de los endpoints disponibles, sus parámetros, respuestas esperadas y ejemplos de uso. Esta documentación facilita tanto el consumo del API por parte de otros sistemas como la comprensión por parte de desarrolladores y partes interesadas.

- **Enlace:** https://goo.su/tDz8hU
- **Descripción:** Interfaz web que permite explorar los recursos disponibles en el API REST de Russell, incluyendo operaciones sobre pipelines, credenciales, usuarios y proyectos. La documentación incluye descripciones detalladas, tipos de datos, códigos de estado HTTP y ejemplos de peticiones/respuestas en formato JSON.

La integración de Swagger se realizó mediante anotaciones en el código fuente utilizando la especificación *OpenAPI 3.0*, permitiendo mantener la documentación sincronizada con el desarrollo del backend.

A.0.6. Publicación librería *@russell-libs/russell-ui* en npm

La librería de componentes web **Russell UI**, desarrollada con StencilJS, fue publicada en el gestor de paquetes npm para facilitar su distribución e integración en distintos proyectos de la organización.

- **Enlace:** https://www.npmjs.com/package/@russell-libs/russell-ui
- **Descripción:** Paquete que contiene los 23 componentes web reutilizables del sistema de diseño Russell, incluyendo botones, dropdowns, modales, inputs, cards, badges y otros elementos UI. Cada componente está documentado y probado.

El paquete puede instalarse en cualquier proyecto Angular, React o Vanilla mediante:

```
# bash
npm install @russell-libs/russell-ui
```

A.0.7. Video demostrativo del gestor de pipelines Russell

Se incluye un video demostrativo que presenta el funcionamiento de **Russell**, el gestor de pipelines desarrollado para administrar flujos CI/CD.

- **Enlace:** https://goo.su/9kzbUo
- **Descripción:** Video en formato MP4 que presenta el sistema de Russell.

A.0.8. Video del sistema de diseño Russell/UI

Se incluye un video explicativo del **sistema de diseño Russell/UI**, donde se muestran sus fundamentos, componentes reutilizables, tokens de diseño y guías de estilo.

- **Enlace:** https://goo.su/9kzbUo
- **Descripción:** Video en formato MP4 que presenta el sistema de diseño Russell/UI.