Universidad Nacional de La Pampa (UNLPam)
Facultad de Ingeniería

Proyecto final de grado de Ingeniería en Sistemas

Título: Automatización del proceso de implementación de software: Una propuesta para el Área de Desarrollo de la Facultad de Ingeniería

Autora: Valentina Scovenna

Grado académico alcanzado: Ingeniera en Sistemas

Directora: Dra. María Belén Rivera. **Cátedra**: Análisis y Diseño de Sistemas I, Sistemas Distribuidos II.

Lugar de presentación: General Pico, La Pampa

Fecha de aprobación: 11/09/2023

Jurados: Salto Carolina, Becker Pablo, Minetti Gabriela. **Filiación**: Facultad de Ingeniería de la UNLPam

Resumen: El objetivo principal de toda organización que desarrolla software es la entrega rápida y eficiente de productos de calidad al cliente. Actualmente, el ámbito del software está revolucionado por el movimiento *DevOps*, que plantea un cambio desde la cultura de trabajo proponiendo la colaboración entre los equipos de desarrollo y profesionales de operaciones. Esto mejora su comunicación y fija objetivos en común, empleando herramientas que sean capaces de automatizar el proceso de desarrollo en su totalidad. Para poder llevar a cabo tal automatización es recomendable utilizar una infraestructura en la nube que permita diseñar e implementar aplicaciones *Cloud Native*, utilizando contenedores que otorguen la capacidad de desplegarlas rápidamente. Ambos enfoques impulsan un cambio significativo en el desarrollo de software, permitiendo trabajar colaborativamente con agilidad y eficiencia, creando productos que se adaptan fácilmente a los cambios y capaces de brindar buenas experiencias al usuario, con mayor calidad y promoviendo la retroalimentación continua entre los equipos y también con el cliente. Este trabajo propone los enfoques *DevOps* y *Cloud Native* para actualizar el Área de Desarrollo de la Facultad de Ingeniería, discutiendo sus ventajas y tecnologías actuales de implementación.

Palabras clave: *DevOps*, Computación en la nube, Automatización, Calidad, Contenedores

Abstract: The main objective of every organization that develops software is the rapid and efficient delivery of quality products to the customer. Currently, the software industry is revolutionized by the *DevOps* movement, which proposes a change in the work culture by proposing collaboration between development and operations professionals teams. This improves their communication and sets common objectives, using tools that are capable of automating the development process in its entirety. In order to carry out such automation, it is advisable to use a cloud infrastructure which allows the design and implementation of *Cloud Native* applications, using containers that provide the ability to deploy them quickly. Both approaches drive a significant change in software development, allowing you to work collaboratively with agility and efficiency, creating products that easily adapt to changes and capable of providing good user experiences, with higher quality and promoting continuous feedback between teams and also with the client. This work proposes the *DevOps* and *Cloud Native* approaches to update the Development Area of the School of Engineering, discussing their advantages and current implementation technologies.

Key words: *DevOps*, Cloud computing, Automation, Quality, Containers

FACULTAD DE INGENIERÍA
Universidad Nacional de La Pampa

"2023: 40 años de restauración democrática"
"50° aniversario de la Nacionalización de la Universidad de La Pampa"

Universidad Nacional de La Pampa
Facultad de Ingeniería

Trabajo final presentado para obtener el grado de
Ingeniera en Sistemas

Automatización del proceso de implementación de software:
Una propuesta para el Área de Desarrollo de la Facultad de Ingeniería

A.P. Valentina Scovenna
Legajo N.° 4923

Directora: Dra. María Belén Rivera

General Pico, La Pampa
Mayo, 2023

Resumen

El objetivo principal de toda organización que desarrolla software, es la entrega eficiente de productos de calidad en el menor tiempo posible. Actualmente, el ámbito del software está revolucionado por el movimiento *DevOps*, que plantea un cambio desde la cultura de trabajo, proponiendo la colaboración entre los equipos de desarrollo y profesionales de operaciones. Esto mejora su comunicación, y fija objetivos en común, empleando herramientas que sean capaces de automatizar el proceso de desarrollo de software en su totalidad. Para poder llevar a cabo la automatización que se propone en *DevOps*, es necesario utilizar una infraestructura en la nube, que permita diseñar e implementar aplicaciones, y fomente el uso de contenedores que otorguen la capacidad de desplegarlas rápidamente.

La práctica de ambos enfoques proporciona a los equipos la habilidad de desarrollar, probar y desplegar aplicaciones rápidamente en la nube, automatizando el proceso de desarrollo de software. Así, las organizaciones entregan software y servicios de manera más rápida, con mayor calidad y se fomenta una cultura de colaboración y aprendizaje continuo promoviendo la retroalimentación entre los equipos de desarrollo y operaciones.

Este trabajo propone el enfoque *DevOps* para actualizar el Área de Desarrollo de la Facultad de Ingeniería, discutiendo sus ventajas y tecnologías actuales de implementación.

Palabras clave: *DevOps*, computación en la nube, automatización, calidad, integración continua, despliegue continuo, entrega continua, contenedores.

Índice general

Resumen I

Índice de tablas VI

Índice de figuras VII

1. Introducción 1

1.1. Problema 3

1.2. Solución 4

1.3. Objetivos 6

1.4. Estructura del trabajo final 6

2. Fundamentos Teóricos 8

2.1. Metodologías de desarrollo de software 9

2.2. El movimiento DevOps 12

2.2.1. Pipeline 14

2.2.2. Integración continua 14

2.2.2.1. Componentes de la integración continua 15

Sistema de administración de código fuente 16

Servidor de CI 17

Jenkins 17

2.2.2.2. Mejores prácticas de la integración continua 20

2.2.3. Entrega continua 21

2.2.4. Despliegue continuo 22

2.2.5. Infraestructura como código 23

2.3. Contenedores 24

2.3.1. Docker 25

2.3.1.1. Arquitectura de Docker 25

2.3.2. Contenerización vs. Virtualización 27

2.4. Orquestación de contenedores 28

2.4.1. Kubernetes 29

2.4.1.1. Arquitectura de Kubernetes . . . 30
2.4.1.2. Objetos de Kubernetes . . . 32
2.4.1.3. Manifiestos de Kubernetes . . . 34
2.5. Computación en la nube . . . 35
2.5.1. Arquitectura de computación en la nube . . . 36
2.6. Angular . . . 37
2.6.1. Arquitectura de Angular . . . 37
2.6.2. Angular CLI . . . 38
2.7. Calidad de código fuente . . . 39
2.7.1. Análisis estático del código - Inspección continua . . . 39
2.7.1.1. SonarQube . . . 40
2.7.2. Análisis dinámico del código - Testing continuo . . . 41
2.8. Conclusión . . . 41

3. Estado del Arte **42**
3.1. Introducción . . . 43
3.2. Servidores de CI . . . 43
3.2.1. Selección del servidor . . . 44
3.2.2. Justificación de la selección . . . 46
3.3. Análisis de calidad del código fuente . . . 46
3.3.1. Selección de la herramienta . . . 47
3.3.2. Justificación de la selección . . . 48
3.4. Proveedores de servicios de nube . . . 48
3.4.1. Selección del proveedor . . . 49
3.4.2. Justificación de la selección . . . 51
3.5. Conclusión . . . 51

4. Situación Actual **53**
4.1. Desarrollo de software en la Facultad de Ingeniería . . . 54
4.2. Arquitectura de la infraestructura . . . 55
4.3. Conclusión . . . 57

5. Desarrollo de la solución **59**
5.1. Introducción . . . 60
5.2. Aplicación Angular . . . 61
5.3. Instalación de Jenkins y SonarQube . . . 61
5.4. Desarrollo de la pipeline de CI . . . 64
5.4.1. Creación de agente para Jenkins . . . 69
5.4.2. Instalación de paquetes npm . . . 70

5.4.3. Configuración del envío de notificaciones 71
5.4.4. Compilación de la aplicación 73
5.4.5. Tests unitarios 74
5.4.6. Análisis de la calidad del código fuente con SonarQube 76
5.4.6.1. Instalación del plugin y configuración en Jenkins 76
5.4.6.2. Configuración en SonarQube y archivo properties 78
5.4.6.3. Añadiendo SonarQube en la pipeline 80
5.4.7. Quality gate 81
5.4.8. Creación de imagen Docker 83
5.5. Creación de manifiestos de Kubernetes 85
5.5.1. Namespace 85
5.5.2. Service 86
5.5.3. Deployment 86
5.5.4. Ingress 87
5.5.5. Kustomization 88
5.6. Desarrollo de tareas de despliegue continuo 88
5.6.1. Publicación de la imagen en Docker Hub 88
5.6.2. Despliegue de manifiestos en clúster de Azure Kubernetes Service 92
5.7. Conclusión 98

6. Conclusiones y Trabajo Futuro **100**
6.1. Conclusión Final 101
6.2. Trabajo Futuro 102

Anexos **104**
A1. Selección de servidor de CI 105
A2. Selección de la herramienta para evaluar la calidad del código fuente 105
A3. Relevación de información sobre el desarrollo en la Facultad de Ingeniería 107
A4. Tets unitarios 114
A5. Análisis de código fuente con SonarQube 116
A5.1. Bugs 116
A5.2. Code smells 118
A5.3. Cobertura de código 119
A5.4. Security hotspots 120
A5.5. Comparación de resultados 122
A6. Réplicas de Pods 125
A7. Estrategia de rollback 129
A8. Archivos Jenkinsfile 130

A8.1. Jenkinsfile 130
A8.2. Jenkinsfile-deploy 132

Referencias **133**

Índice de tablas

3.1. Comparación de servidores de integración continua 45
3.2. Comparación de herramientas para el análisis de la calidad de código fuente 47
3.3. Comparación de los proveedores de la nube 50

A1.1. Top cinco de los servidores de CI más utilizados 105
A1.2. Cantidad de menciones de los servidores de CI en las páginas web seleccionadas 105
A2.1. Top cinco de los analizadores de calidad de código fuente más utilizados 106
A2.2. Cantidad de menciones de las herramientas en las páginas web seleccionadas 106

Índice de figuras

2.1. Gráfico de torta que muestra los resultados de la encuesta. 10
2.2. Ciclos de desarrollo en Scrum. 11
2.3. Bucle que muestra las actividades realizadas por DevOps. 12
2.4. Esquema que ilustra la práctica de integración continua. 16
2.5. Arquitectura maestro-esclavo del servidor Jenkins. 18
2.6. Esquema que ilustra la práctica de entrega continua. 21
2.7. Esquema que ilustra la práctica de despliegue continuo. 23
2.8. Esquema de la infraestructura como código. 24
2.9. Barco de contenedores. 24
2.10. Arquitectura de Docker. 26
2.11. Ejemplo ilustrativo de una imagen Docker. 27
2.12. Ejemplo de la arquitectura de máquinas virtuales versus contenedores. 28
2.13. Componentes del nodo máster. 31
2.14. Componentes de los nodos trabajadores. 31
2.15. Diagrama sobre la arquitectura de Kubernetes. 32
2.16. Ejemplo de un Ingress enviando tráfico hacia un servicio. 33
2.17. Servicios de la nube. 36
2.18. Arquitectura del framework Angular. 38

3.1. Conjunto de tecnologías (ó stack tecnológico) seleccionadas. 52

4.1. Esquema de los servidores utilizados por el Área de Desarrollo. 55
4.2. Estrategia de ramificado empleada por el Área de Desarrollo. 56
4.3. Esquema del flujo de trabajo. 56

5.1. Esquema de la pipeline de CI/CD desarrollada. 60
5.2. Aplicación Calculadora almacenada en GitHub. 61
5.3. Ejecución del comando docker-compose start. 63
5.4. Servicio de Jenkins inicializado. 63
5.5. Servicio de SonarQube inicializado. 63
5.6. Selección de tipo de tarea a realizar y su nombre. 64

5.7. Configuración general de la pipeline en Jenkins. 65
5.8. Configuración de Build triggers. 65
5.9. Configuración del archivo Jenkinsfile que contiene la pipeline. 66
5.10. Configuración de webhook en GitHub. 67
5.11. Envío de datos a través de webhook. 67
5.12. Visualización en la interfaz de Jenkins de la etapa que realiza el checkout. 68
5.13. Logs de la etapa "Declarative: Checkout SCM". 68
5.14. Visualización en la interfaz de Jenkins de la etapa que construye el agente Docker. 70
5.15. Logs de la etapa "Declarative: Agent Setup". 70
5.16. Visualización en la interfaz de Jenkins de la etapa que instala los paquetes npm. 71
5.17. Logs de la etapa "Install". 71
5.18. Configuración para el envío de notificaciones. 72
5.19. Captura de pantalla del correo electrónico recibido. 73
5.20. Visualización en la interfaz de Jenkins de la etapa que compila la aplicación Angular. . . . 74
5.21. Logs de la etapa "Build". 74
5.22. Visualización en la interfaz de Jenkins de la etapa que ejecuta las pruebas unitarias. 75
5.23. Líneas de código del archivo karma.conf.js. 75
5.24. Logs de la etapa "Test". 76
5.25. Configuración de la instalación de JDK. 77
5.26. Búsqueda del plugin SonarQube Scanner dentro de los disponibles en Jenkins. 77
5.27. Configuración de SonarQube Scanner en Global Tools Configuration de Jenkins. 78
5.28. Creación de webhook en SonarQube. 78
5.29. Configuración de webhook en SonarQube. 79
5.30. Visualización en la interfaz de Jenkins de las etapas ejecutadas hasta el momento. 80
5.31. Logs de la etapa "SonarQube Analysis". 81
5.32. Configuración de Quality Gate por defecto. 82
5.33. Pipeline de Jenkins con la etapa de Quality Gate. 82
5.34. Logs de la etapa "Quality Gate". 83
5.35. Visualización gráfica de las etapas ejecutadas, incluyendo la construcción de la imagen de Docker. 83
5.36. Configuración de la instalación de Docker en Jenkins. 84
5.37. Logs de la etapa "Build Docker Image". 85
5.38. Creación de credenciales en Jenkins. 89
5.39. Etapa de publicación de la imagen de Docker ejecutada por Jenkins. 90
5.40. Logs de la etapa "Publish Docker Image". 90
5.41. Imagen Docker de la aplicación Angular publicada en Docker Hub. 91
5.42. Listado de los tipos de credenciales que se pueden crear en Jenkins. 92

5.43. Creación de credencial para Azure Service Principal. 92
5.44. Configuración del Jenkinsfile en la pipeline calculadora-angular-deploy. 93
5.45. Logs de la etapa declarativa "Post Actions". 95
5.46. Ejecución de la pipeline calculadora-angular-deploy. 96
5.47. Logs de la etapa "Deploy Dev". 96
5.48. Salida de consola del comando kubectl get all en el namespace calculadora. 97
5.49. Aplicación desplegada exitosamente en el clúster de Azure Kubernetes Service. 97
5.50. Comparación de tiempo de las ejecuciones. 98
5.51. Tiempo total de ejecución para desplegar la aplicación. 98

A5.1. Proyecto creado y analizado en SonarQube. 116
A5.2. Información del proyecto Calculadora_Angular. 116
A5.3. Detalle de los bugs localizados luego del análisis. 117
A5.4. Ejemplo de la localización de uno de los bugs. 117
A5.5. Solución al bug encontrado. 118
A5.6. Detalle de los code smells localizados luego del análisis. 118
A5.7. Ejemplo de uno de los code smell. 119
A5.8. Solución al code smell encontrado. 119
A5.9. Detalle de la cobertura de testeo que posee el proyecto. 120
A5.10. Ejemplo de líneas de código que no están cubiertas por tests unitarios. 120
A5.11. Información del security hotspot hallado. 121
A5.12. Resolviendo el security hotspot indicado. 121
A5.13. Security hotspot resuelto. 122
A5.14. Detalle de nuevos resultados del análisis. 122
A5.15. Gráfico de problemas en la ejecución anterior. 123
A5.16. Gráfico de problemas en la nueva ejecución. 123
A5.17. Gráfico de problemas en la ejecución anterior. 123
A5.18. Gráfico de problemas en la ejecución posterior. 124
A6.1. Recursos publicados en el clúster. 125
A6.2. Eliminación de Pod. 125
A6.3. Pronta recuperación luego de la eliminación de Pod. 126
A6.4. Desplegando imagen errónea. 126
A6.5. Descripción del Pod. 127
A6.6. Eventos en la descripción del Pod. 127
A6.7. Recursos actuales en el clúster. 128
A7.1. Ejecución a demanda de la pipeline calculadora-angular-deploy. 129

1 Introducción

"La "brecha del software" puede no ser inmutable, pero cerrarla requerirá una metamorfosis en la práctica de la producción de software y su sirviente, el diseño de software."

— David Gries y A.G. Fraser

En este capítulo introductorio se expone la problemática, o situación detectada en relación al desarrollo de software, y se presenta una propuesta para el área de desarrollo de la Facultad de Ingeniería, perteneciente a la Universidad Nacional de La Pampa (UNLPam) como parte de la solución planteada. Además, se describen los objetivos generales y específicos que este trabajo estipuló alcanzar y la estructuración en capítulos de la presente tesis.

El desarrollo de software lleva casi cuatro décadas de continuos cambios en las herramientas, metodologías y tecnologías propuestas para dar solución al problema de la planificación, desarrollo y aseguramiento de calidad. Estos cambios se impulsaron debido a la propia naturaleza cambiante de los problemas a resolver y la dinámica de las organizaciones. Desde las metodologías secuenciales, como la de cascada, hasta las metodologías tradicionales, con su enfoque iterativo e incremental, todas aportaron en su momento una manera formal y disciplinada de llevar a cabo los proyectos de software, siempre con el objetivo de lograr un producto más eficiente y de mayor calidad.

En el año 2001, con Kent Beck como principal impulsor, un grupo de dieciséis profesionales reconocidos en la industria de software y representantes de nuevas metodologías se reunieron para discutir sobre las prácticas de desarrollo, contemplando otras características no consideradas hasta ese momento. La idea detrás de esta convocatoria era determinar los valores y principios que permitieran a los equipos desarrollar software de manera más rápida y responder mejor a los cambios que pudieran surgir a lo largo de un proyecto de desarrollo. En contraposición con las metodologías tradicionales, se pretendía poner énfasis en la satisfacción del cliente (punto clave en la calidad de un producto) y en la entrega rápida de software incremental (módulos funcionales de software).

Producto de esta convocatoria se crea una organización sin ánimo de lucro denominada *Agile Alliance*[1] en la que se establece el *"Manifiesto por el desarrollo ágil de software"*, más conocido como el *Manifiesto Ágil*. En este, se establecen un total de doce principios inspirados en los siguientes valores fundamentales, recomendados a seguir por cualquier organización:

- Los individuos y sus interacciones sobre los procesos y las herramientas.
- El software que funciona, más que la documentación exhaustiva.
- La colaboración con el cliente, y no tanto la negociación con el contrato.
- Responder al cambio, mejor que apegarse a un plan.

Así, y en consideración de los valores y lineamientos[2], surgen los métodos ágiles, que combinan una filosofía (la ágil) con un conjunto de principios más orientados al desarrollo y a las pruebas continuas. Esto, con el claro objetivo de lograr un rápido incremento de software que satisfaga las necesidades de los usuarios y clientes, en tanto que propone minimizar las actividades de diseño o, más bien, reducirlas a interacciones breves, que permitan validarlas con el cliente, junto con la implementación, a través de los casos de prueba.

Está claro que en la evolución de las metodologías de desarrollo, en cada una de estas, las actividades propuestas como parte del ciclo de vida del desarrollo del software son esencialmente las mismas (análisis, diseño, implementación y pruebas). Sin embargo, difieren en su forma de ser llevadas a cabo (¿son secuenciales?, ¿son iterativas?, ¿son en espiral?) y la filosofía que subyace por detrás de cada metodología, que permite llevar adelante cada una de las actividades previstas. Por lo tanto, optar por una u otra puede resultar riesgoso. Las opiniones al respecto suelen ser extremistas y van desde los defensores de las metodologías tradicionales, que minimizan las ágiles por considerarlas "livianas", poco disciplinadas o rigurosas en consideración del producto software que tienen que construir, hasta el otro extremo, los "agilistas" quienes suelen considerar a las tradicionales como demasiadas rigurosas y con escasa relación con el cliente. De nuevo, estas posiciones no suelen contemplar que cada enfoque hace las mismas tareas, pero de distintas maneras y que, como todo, no hay una respuesta absoluta sobre cuál enfoque es el adecuado para desarrollar software de calidad. Muchos conceptos ágiles solo son adaptaciones de algunos que provienen de la buena ingeniería de software tradicional.

[1]http://www.agilealliance.org
[2]https://agilemanifesto.org/iso/es/principles.html propuestos en el *Manifiesto Ágil*

A pesar del auge de las metodologías ágiles, y la consecuente evolución de la forma de desarrollar software, los equipos de desarrollo y operaciones han permanecido aislados durante años. Precisamente, las actividades que se llevan a cabo como parte de la fase de desarrollo (esto es, la programación realizada por el conjunto de desarrolladores) e implementación (puesta en producción por parte del equipo de operaciones), siguen aisladas, sin mediar entre ellas, panorama que comenzó a considerarse como una disfunción grave del sector. Esta situación queda expuesta y visible sobre el año 2007, cuando Patrick Debois, un consultor freelance, manifiesta los conflictos e inconvenientes surgidos entre los equipos de operaciones y desarrollo en una organización del gobierno belga para la cual trabajaba.

Un año más tarde en el marco de la conferencia sobre métodos ágiles, conocida como *Agile 2008 Conference*[3], Debois se encuentra con Andrew Clay Shafer, quien estaba exponiendo sobre *"infraestructura ágil"*, una temática que introducía conceptos de los principios ágiles relacionado a la infraestructura de sistemas. En esta ocasión, ambos coincidieron respecto a lo problemático que estaba resultando la división en los equipos de trabajos y la coordinación entre los mismos. Por esto, deciden crear un grupo en Google llamado *Agile System Administrators*, para dar espacio a que otras personas puedan compartir sus opiniones al respecto y continuar con el debate [Ruiz, 2015].

Finalmente, en el año 2009, John Allspaw y Paul Hammond exponen una presentación denominada *"10 deploy per Day: Dev & ops cooperation at Flickr"*[4] en la conferencia *O'Relly Velocity '09 Conference*. En la exposición dejan en claro que la idea principal es comenzar a enfocarse en asegurar que las áreas de desarrollo (de aquí que comienza a conocerse como *Dev*, por la abreviatura en inglés de *Development*) y operaciones (*Ops* proveniente de *Operations*) trabajen juntos y colaboren coordinadamente, utilizando herramientas y procesos ágiles. Constituye esto, los primeros indicios que proponían enfocar esfuerzos en la integración de las áreas que llevan a cabo actividades de operaciones (infraestructura) y desarrollo.

Sobre la base de esta gran idea, Patrick Debois organiza a finales de ese mismo año la conferencia *DevOpsDays*[5], con la finalidad de cubrir temas de desarrollo de software, operaciones de infraestructura de Tecnologıa de la Información (TI) y la intersección entre ellos. A partir de este entonces, se populariza el término *DevOps* y se adopta como un concepto que comprende una nueva forma de desarrollar software, tendiente a resolver el aislamiento evidente entre los mencionados equipos y que pretende subir un nivel en relación a prácticas y herramientas para construir software de calidad, con mayor rapidez.

1.1. Problema

Durante la conferencia de la Organización del Tratado del Atlántico Norte (OTAN) de Ingeniería de Software en 1968, se debatió sobre las mejores prácticas a llevar a cabo en el desarrollo de software con el fin de solventar los problemas claves que atormentaban a los profesionales de software de todo el mundo, quienes definieron a tal situación, con el término *"crisis del software"* o *"brecha de software"* [Naur y Randell, 1969].

En aquel entonces, los proyectos se caracterizaban por una deficiente fiabilidad y, realizar cambios, se tornaba complicado, lo que implicaba extensos tiempos de desarrollo que superaban a lo estimado, aumento en los costos y retrasos en las entregas, impactando negativamente en la satisfacción del usuario. En la década de los años 60, en un contexto en el que gran parte de los profesionales del software carecían de una educación formal y los conocimientos eran adquiridos por medio de experimentos, se

[3]https://dblp.uni-trier.de/db/conf/agiledc/agile2008.html
[4]https://www.oreilly.com/library/view/devops-in-practice/9781491902998/video169253.html
[5]https://devopsdays.org/

adjudicaba parte de esta crisis a la inmadurez educacional.

Con el advenimiento de los principios y metodológías ágiles en el desarrollo de software, la crisis se atribuyó a otras cuestiones. De acuerdo a [Kim *et al.*, 2021], actualmente la causa de este problema crónico es el conflicto entre el equipo de desarrolladores y los operadores de TI. Mientras que los desarrolladores tienen como objetivo responder rápidamente a los cambios en el mercado e implementarlos en producción lo antes posible, los operadores son responsables de proporcionar a los clientes un servicio estable, confiable y seguro, seguridad que dificulta la introducción de cambios en producción por parte de los desarrolladores, ya que pueden vulnerar el estado actual del producto.

Pese a que aisladamente cada equipo tenga un buen enfoque, el flujo total del desarrollo de software y el objetivo general del proyecto se ven frustrados, y en ocasiones, contradictorios. Estas metas opuestas y la falta de trabajo en conjunto conducen a una relación conflictiva que detona en una calidad deficiente del software y del servicio, sin mencionar la evidente insatisfacción del cliente y el impacto negativo del negocio.

Desde luego, fue Debois quien, en el año 2007 trabajaba en un proyecto de consultoría para el gobierno de Bélgica y debían migrar un centro de datos (*data center*). En sus crónicas, señala cómo se frustra debido a los conflictos que se producían entre desarrolladores y administradores de sistemas.

Esta disyuntiva se traduce en lo siguiente. Si no se aplican los controles necesarios en el desarrollo del software o se desconocen las características del ambiente, cuando se realice el despliegue a producción, es probable que surjan errores a solucionar que retrasen la entrega del mismo. A su vez, si surge un nuevo requerimiento por parte del cliente, será retrasado hasta solucionar el problema anterior y lograr la estabilidad tanto en desarrollo como en operaciones, formando así una espiral descendente en la que cada vez el producto se vuelve más vulnerable ante los cambios y se complejiza el trabajo de estabilizarlo, si no se lo logra a tiempo.

Además, esto conlleva al aumento del esfuerzo humano y económico. Se genera agotamiento, impotencia e incluso desesperación al no poder solventar los problemas ocasionados. Al mismo tiempo que los trabajos urgentes y no planificados elevan el costo total del producto software, teniendo que aplazar actividades planificadas y retrasando la entrega final del mismo.

Las organizaciones que no sean capaces de resolver este conflicto central, quedarán en una clara desventaja competitiva, la cual implica un tiempo de comercialización rápido, con servicios de alta calidad. Ya en el año 1968, en aquella conferencia de la OTAN sobre ingeniería de software, uno de los participantes, profesor de la Universidad de Cambridge, A. G. Fraser, expresó: *"Estamos haciendo un gran progreso, sin embargo, las demandas en la industria parecen avanzar mucho más rápido que nosotros. Debemos admitir esto, aunque tal admisión sea difícil."* [Naur y Randell, 1969].

En este sentido, la Facultad de Ingeniería de la Universidad Nacional de La Pampa como organización pública estatal se encuentra en una situación en la que en materias de desarrollo y operaciones todavía se llevan a cabo prácticas que a futuro podrían quedar obsoletas, dificultando la adopción de nuevas tendencias tecnológicas. En la actualidad, el Área de Desarrollo no aplica ninguna metodología de desarrollo, y tampoco realizan los controles suficientes para asegurar un producto de calidad hacia el usuario final, además de contar con un flujo de trabajo propenso al error humano.

1.2. Solución

Tal como se sugirió en la conferencia de la OTAN en 1968, se requiere de un cambio radical para hacer frente a esta problemática de la industria del software, y así lograr la meta principal: entregar productos de calidad y de valor para el cliente lo más rápido posible, obteniendo de esta manera un *feedback*

que permita continuar con la mejora del software y, consecuentemente, cumplir con las necesidades del cliente.

Hay tres puntos claves que permiten solventar las problemáticas halladas.

Si los diferentes procesos que se llevan a cabo durante el desarrollo de software (compilación, testeo, despliegue) no se encuentran automatizados, esto puede ocasionar que todas las ejecuciones sean diferentes con un mínimo cambio en el código, en la configuración o en el ambiente. Además, la ejecución manual de estos los vuelve propensos a errores, ya que si no se lleva un control y documentación de lo que se está haciendo, se vuelve difícil detectar qué fue lo que se hizo exactamente, perdiendo el control, tanto del despliegue como de la calidad del producto.

Dicho esto, el primer punto clave es la **automatización**. Construyendo un flujo automatizado de los procesos del desarrollo de software, se logra mejorar la eficiencia, puesto que son más rápidos y consistentes que los procesos manuales, garantizando la reducción del riesgo de error humano, lo que conlleva a software de mayor calidad. Además, se reducen los costos de mano de obra al automatizar tareas repetitivas que consumen tiempo de trabajo, permitiendo que los desarrolladores se enfoquen en otras tareas más valiosas para el proyecto. La automatización ayuda a agilizar la colaboración entre los trabajadores, facilitando el seguimiento y la gestión de cambios, garantizando que todos trabajen con la información más actualizada. Asimismo, otorga la capacidad de escalabilidad para manejar cargas mayores de trabajo, facilitando la adaptación a los cambios y crecimientos de acuerdo a las necesidades del desarrollo. Por otra parte, con la automatización de los testeos, se evita que las pruebas se realicen al final del ciclo y, fusionándose con el flujo del desarrollo, se obtiene software de mayor calidad desde la construcción, puesto que para ser desplegado necesita pasar por el control de calidad correspondiente validando que cumple con todos los controles definidos.

Otro punto importante es la **frecuencia del lanzamiento** de los cambios. Si el tiempo entre cada lanzamiento es pequeño, el riesgo asociado se reducirá significativamente y será mucho más fácil revertirlo. A su vez, realizando entregas frecuentes, se acelera el tiempo de comercialización, ya que esto conduce a disponibilizar funciones nuevas y corregir errores más rápidamente, lo que permite una retroalimentación más rápida por parte de los clientes, que fomenta la comunicación entre las distintas partes involucradas, mejorando la experiencia del usuario y una mayor satisfacción del cliente, facilitando el alcance del objetivo propuesto.

Al implementar pequeñas actualizaciones, se puede detectar y solucionar problemas desde el principio, lo que lleva a una mayor fiabilidad y estabilidad del software. Incluso, ejecutando los cambios de manera incremental, se reduce el riesgo de fallas importantes, y esto permite revertir los cambios a un estado funcional, en caso de ser necesario.

Sin embargo, la automatización de los procesos y la frecuencia de despliegue, no son suficientes si la comunicación y el trabajo en equipo no prosperan, por esta razón se debe hacer énfasis en la **comunicación y trabajo colaborativo** entre los equipos de desarrollo y de operaciones, promoviendo la integración, visibilidad y transparencia entre ambos. Esta relación impulsa el trabajo eficiente, de calidad y un bucle de retroalimentación continua con los clientes sobre las mejoras, el desarrollo, las pruebas y la puesta en marcha. Dejando atrás la forma de trabajo en silos (áreas trabajando aisladas unas de otras), y trabajando colaborativamente durante todo el ciclo de vida del desarrollo de software, se reducen los malentendidos y se mejora la comunicación entre los equipos.

La solución consiste en contar con una metodología que permita el desarrollo del software con procesos más rápidos, fiables y de mayor calidad. Que se potencie la cultura de equipo, basada en la colaboración y comunicación entre los miembros de las distintas áreas a las que les compete el desarrollo del software. De esta manera, las organizaciones se aseguran que exista total transparencia entre todas las partes involucradas.

A tal efecto, se propone al Área de Desarrollo de la Facultad de Ingeniería de la Universidad Nacional de La Pampa, comenzar a considerar y a capacitar el personal para trabajar con estas prácticas de modo tal que se actualice dicho área, fomentando la entrega de productos de calidad, la colaboración entre los equipos y la innovación tecnológica. Asimismo, se sugiere el cambio de la infraestructura hacia las tecnologías ofrecidas por la nube, puesto que proporcionará no solo escalabilidad y flexibilidad, sino que también la automatización de varios procesos, entre ellos el despliegue de las aplicaciones web, permitiendo aumentar la eficiencia del equipo de trabajo bajo un enfoque más ágil.

A partir de esta solución, se considera que la facultad podrá experimentar una mejora significativa en la entrega de sus productos educativos y administrativos. Sin embargo, la actualización del Área de Desarrollo necesita de una planificación y formación adecuada para garantizar el éxito de esta transición.

1.3. Objetivos

El objetivo general del trabajo final es analizar tecnologías actuales tendientes a prácticas *DevOps* y discutir la aplicación de algunas de ellas para la propuesta de actualizacón de la metodología de desarrollo de la Facultad de Ingeniería. De esta manera, se pretende que el presente trabajo promueva un enfoque *DevOps* de integración y despliegue continuo del software; junto con el cambio de la infraestructura actual hacia una infraestructura de *Cloud Native*.

A partir del mismo, se desglosan los siguientes objetivos específicos:

- Analizar las diferentes herramientas disponibles en el mercado para la automatización de la integración y despliegue continuos.
- Diseñar una *pipeline* de integración continua y despliegue automático en la infraestructura seleccionada.
- Construir e implementar el despliegue automático en un clúster de Kubernetes.
- Implementar la *pipeline* diseñada con una prueba de concepto para ejemplificar la propuesta planteada.

1.4. Estructura del trabajo final

Este trabajo se estructura en seis capítulos, cuyas temáticas se distribuyen de la siguiente manera.

- El primer capítulo presenta la introducción a esta tesis y describe: 1) la problemática a abordar, 2) la solución actual que se plantea para dicha problemática, 3) los objetivos que se estipularon alcanzar y 4) la estructura del trabajo.
- El segundo capítulo introduce al lector hacia los fundamentos teóricos esenciales para la comprensión del trabajo.
- El tercer capítulo describe el estado del arte y proporciona una breve comparación de las diferentes herramientas y servicios a utilizar en el desarrollo de la propuesta.
- En el cuarto capítulo se presenta la situación actual junto con la arquitectura de la infraestructura del Área de Desarrollo de la Facultad de Ingeniería de la UNLPam, y se expone la conclusión arribada sobre tal presente.

- El quinto capítulo se centra en el desarrollo de la solución para robustecer la problemática planteada, detallando el paso a paso de cada configuración realizada para implementar la solución propuesta.
- Para finalizar, el capítulo seis expone las conclusiones a las que se arribó luego del desarrollo de este trabajo y se detallan las propuestas para trabajos futuros.

2
Fundamentos Teóricos

"Me estaba dando por vencido, tal vez la idea era demasiado loca:
desarrolladores y operaciones trabajando juntos.
Pero ahora, el fuego realmente se está extendiendo."

— Patrick Debois

En el presente capítulo se introduce al lector hacia los fundamentos teóricos que resultan necesarios para la comprensión del trabajo. De igual manera, se detallan las tecnologías y prácticas utilizadas, abordándolas desde un punto de vista teórico.

2.1. Metodologías de desarrollo de software

A lo largo de la carrera de Ingeniería de Sistemas se instruye sobre diferentes metodologías a aplicar para obtener un buen desarrollo de software. Estas, al igual que la tecnología, evolucionan con el pasar de los años para poder adecuarse a las demandas del mercado. Si bien no hay una categorización general, al momento se pueden diferenciar dos grandes grupos que las comprenden a todas: las metodologías tradicionales y las metodologías ágiles.

La primera de ellas fue publicada por Winston W. Royce en 1970, llamada **modelo en cascada o secuencial**, la cual sugiere un enfoque lineal, en el que nada está hecho hasta que todo se termina (de ahí la asociación con una "cascada"). Se compone mínimamente de cinco etapas: análisis y definición de requerimientos, diseño del sistema y del software, implementación y prueba de unidades, integración y prueba del sistema y, funcionamiento y mantenimiento [Sommerville, 2007].

El modelo en cascada supone que, para avanzar de actividad, se deben completar las actividades que constituyen la fase anterior. Este aspecto no deja margen de error y convierte a este enfoque poco flexible, ya que, por ejemplo, al llegar a la etapa de implementación y prueba, pueden descubrirse errores acarreados de fases previas, lo que conduce al rediseño y modificación del código afectado, impactando en el costo y tiempo de desarrollo. Esto también puede suceder al culminar la etapa final del modelo en cascada, lo que implicaría iterar cada fase nuevamente.

Otra gran desventaja que aplica a este enfoque, es la falta de comunicación con el cliente, dado que solamente se lo contacta al inicio del proyecto para realizar el relevamiento de requerimientos y luego, en la etapa final, para la entrega del producto. El cliente no posee conocimiento, ni se permite su intervención durante el proceso de desarrollo, derivando en productos que podrían tener características no deseadas, dando como resultado la insatisfacción del mismo. No solo la comunicación es escasa con el cliente, sino que también lo es entre los miembros del equipo de desarrollo, debido a que realizan las tareas independientemente sin generar reportes hacia el equipo y notificando solo cuando finalizan la etapa o tarea asignada. Esta metodología es adecuada para proyectos en los que los requerimientos son bien conocidos desde el principio y no cambiarán. Un ejemplo podría ser el software de una expendedora de bebidas.

En 1988, Barry Boehm establece una nueva metodología conocida como **espiral**, diseñada para gestionar el riesgo que puede ocasionarse durante un proyecto. En ella, cada espiral (vuelta) conforma una nueva iteración de desarrollo, comenzando desde el centro y expandiéndose hacia afuera. Entre sus ventajas, se destaca la flexibilidad ante los requerimientos (ya que se pueden añadir nuevas fases), y un buen desempeño en grandes proyectos. No obstante, al desconocer la cantidad de giros y fases por las que debe atravesar el proyecto, se torna complejo y de elevado costo. Además, posee alta dependencia hacia el análisis de riesgos y exigencia de una documentación apropiada, por lo que, si no se tiene cierto dominio en el tema, el manejo del riesgo resulta una clara desventaja.

Años más tarde, surge la **metodología RUP** (*Rational Unified Process*), traducido al español como Proceso Unificado de Rational. El principal aporte de RUP fue considerar a las actividades de la metodología de desarrollo como iterativas e incrementales. Esto se logra permitiendo que cada fase del ciclo de vida de desarrollo pueda ser repetida hasta que se cumpla con la meta principal, logrando una versión funcional del producto en cada iteración y obteniendo una devolución continua del cliente. Los principales beneficios de aplicar esta metodología son la reducción de riesgos en el proyecto, la incorporación de calidad y la disminución de tiempo de desarrollo, al considerar iteraciones cortas con incrementos de funcionalidades específicas.

Si bien las metodologías tradicionales fueron evolucionando gracias al aporte de diferentes autores, su esencia del enfoque secuencial, nacida con el modelo en cascada, perduró en todas ellas. Pese a que se

mejoró la comunicación con el usuario, la estructura poco flexible ante cambios, los roles y actividades asignadas, y la exhaustiva documentación que requieren, vuelven a estas metodologías costosas, tanto en dinero como en tiempo.

En 1994, *The Standish Group International*[1], publica un detallado reporte llamado *The CAOS Report*, en donde principalmente se muestra los datos de una encuesta realizada a 365 desarrolladores que representaban un total de 8.380 aplicaciones [The Standish Group International, 1995].

Figura 2.1: *Gráfico de torta que muestra los resultados de la encuesta.*
Nota: Adaptado de [The Standish Group International, 1995].

En la misma se clasificaron los proyectos en tres tipos tal como muestra la Figura 2.1. El Tipo 1, obteniendo el valor más bajo con un 16.2 %, identificaba proyectos exitosos, aquellos que fueron completados en tiempo y con el presupuesto predefinido, con todas las características y funciones que se especificaron inicialmente. El Tipo 2 se correspondía con proyectos completados y operacionales que sobrepasaron el presupuesto y el tiempo estimado, además de ofrecer menos características y funciones de lo esperado. Este tipo de proyectos alcanzó un 52.7 % del total de encuestados. Por último, el Tipo 3, se correspondió con aquellos proyectos que tuvieron que ser cancelados en algún punto del ciclo de desarrollo, ocupando en segundo lugar un 31.1 %.

La causa de que el 83.8 % de los proyectos no fueran exitosos se debió a los cambios impredecibles en los requerimientos y la falta de comunicación con los usuarios del sistema, además de los costos elevados en tiempo y dinero. Esto impulsó a los líderes de la industria a comenzar la búsqueda de un método efectivo que cese con estos inconvenientes.

En el año 2001, se formaliza la teoría sobre las **metodologías ágiles**, a partir de la publicación del *Manifiesto Ágil*, el cual establece 12 principios para un desarrollo de software rápido, eficiente y adaptable a los cambios. Estas metodologías están orientadas a la satisfacción del cliente y a la reducción de la extensa documentación que se realizaba con las metodologías tradicionales. Suponen también un enfoque incremental, puesto que se van añadiendo nuevas funcionalidades en cada ciclo de desarrollo hasta alcanzar el objetivo final. Pero, a diferencia de las metodologías tradicionales, estos ciclos son mucho más cortos y rápidos, permitiendo entregas de valor al cliente a medida que se van construyendo nuevas funcionalidades, adquiriendo consecuentemente un rápido feedback o retroalimentación con el cliente. Así, se permite al cliente aportar correcciones o nuevos requerimientos, ya que es testigo de cómo el proyecto va avanzando. Además, el equipo de desarrollo mejora su comunicación, generando reuniones en períodos cortos de tiempo para poner en común los avances y dificultades.

De acuerdo a la última publicación del *State of Agile Report* [Digital.ai, 2022], más de la mitad de los profesionales entrevistados adoptaron estas metodologías para acelerar el tiempo de comercialización.

[1] https://standishgroup.myshopify.com/

También, la mayoría de ellos respondieron que se debió a la necesidad de disminuir el riesgo y elevar la previsibilidad de entrega de sus productos, la capacidad de moverse rápidamente y ser predecible es un beneficio clave en los últimos reportes que se destaca de muchos otros.

En dicho informe, más de 3000 personas de la comunidad compartieron sus experiencias. Scrum resultó ser la metodología ágil más utilizada en el mercado obteniendo un 87 % en las encuestas, seguida con un 56 % por Kanban[2], luego ScrumBan[3] con un 27 % e iterativa con un 20 %.

En la metodología **Scrum**, cada día comienza con una reunión breve de aproximadamente 15 minutos, lo que se denomina como el *"Scrum diario"*. En ésta, se sincronizan las actividades y se planifica la jornada laboral, permitiendo verificar el progreso del proyecto y resolver los diferentes inconvenientes. Se caracteriza por sus cortos ciclos de desarrollo (entre dos y cuatro semanas), conocidos como *"sprints"*, y por la maximización del tiempo de desarrollo de un producto de software. La Figura 2.2 ilustra el concepto de *sprint* en Scrum.

Figura 2.2: *Ciclos de desarrollo en Scrum.*
Nota: Adaptado de [Kostiantyn, sf]

Tanto la metodología RUP, como las metodologías ágiles, tienen un enfoque iterativo e incremental para el desarrollo de software, permitiendo realizar entregas tempranas al cliente generando un rápido *feedback* por parte del mismo. Sin embargo, estas actividades son llevadas a cabo solamente por el equipo de desarrollo, culminando en la entrega de un producto hacia el equipo de operaciones que, posteriormente realizará el correspondiente despliegue y monitoreo del software, teniendo que comunicar a los desarrolladores si ocurre algún imprevisto o error durante el mismo.

Tal y como avanza la tecnología, el mercado de software comienza a evolucionar hacia las nuevas prácticas que se implementan, buscando mejorar principalmente la calidad y el tiempo de entrega de sus productos, siendo su prioridad la satisfacción del cliente. Así es como nace el movimiento *DevOps* en el año 2008, creado con el objetivo de establecer metas en común entre los equipos de desarrolladores y profesionales de operaciones de TI, logrando que ambos grupos comenzaran a trabajar en conjunto para construir productos más eficientemente y hacer entregas con frecuencia y de valor hacia los clientes. *DevOps* es simbolizado por el símbolo infinito para ilustrar como las diferentes fases del ciclo de desarrollo se relacionan entre sí (ver Figura 2.3).

[2]https://kanbanize.com/es/recursos-de-kanban/primeros-pasos/que-es-kanban
[3]https://en.wikipedia.org/wiki/Scrumban

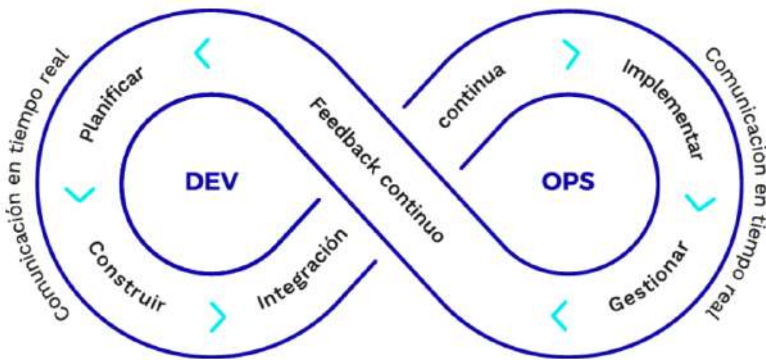


Figura 2.3: *Bucle que muestra las actividades realizadas por DevOps.*
Nota: Extraído de [Angulo, 2018]

2.2. El movimiento DevOps

Para lograr una mejor comprensión de los principios de este nuevo movimiento del desarrollo de software, es preciso retroceder en la historia hacia la finalización de la Primera Revolución Industrial. A pesar de ser dos mercados totalmente diferentes; esta nueva cultura se basa en la evolución de los principios que movilizaban a las industrias manufactureras del siglo pasado.

En la década de 1890, la industria textil japonesa atravesaba por un auge, siendo sus productos quienes dominaban los mercados nacionales, compitiendo con éxito frente a los textiles británicos en China e India. Es entonces cuando Sakichi Toyoda, el hijo mayor de una familia de carpinteros, contradiciendo el pensamiento anticuado de mantener en secreto los conocimientos y solo transmitirlos de generación en generación, se plantea como objetivo contribuir a la sociedad a través de sus invenciones, con el fin de que la tecnología se propagara y así todas las personas lograran beneficiarse de esta. Persiguiendo su sueño y siendo consciente de lo que ocurría en su país frente al aumento de la demanda textil, decide incursionar en la mejora de los telares manuales, obteniendo su primera patente en 1891 con un telar manual de madera.

En una de sus mejoras, Sakichi propone el concepto *Jidoka "automatización con un toque humano"*, para brindar la calidad desde la fuente. En esta invención, además de sustituir el trabajo manual con el telar automático que él mismo había creado, añadió a las máquinas un novedoso mecanismo de detección de rotura de hilo, el cual frenaba la producción para que el operario pudiera solucionar el error y así continuar fabricando.

La filosofía *Jidoka* buscaba la detección temprana de errores para su pronta corrección, ya que el objetivo era producir a una calidad máxima, deteniendo la producción cuando se detectaba un fallo, y creando máquinas automáticas que no requirieran de la manipulación de los operarios. Poniendo esto en práctica, el sistema mejoró la eficacia y eficiencia del trabajo y de la producción.

Con la llegada de la crisis de la Gran Depresión a Japón y, en especial, la crisis del mercado textil, Sakichi Toyoda y su hijo Kiichiro Toyoda, quien trabajaba junto a él, se ven atraídos por la emergente industria del automóvil, por lo que ambos deciden vender las patentes obtenidas hasta el momento e invertir el dinero en su nuevo objetivo, fabricar el primer coche de Japón. Lamentablemente, al poco tiempo Sakichi enferma y fallece en el año 1930. Sin embargo, su hijo toma las riendas de la empresa familiar y, años más tarde, logra fundar *Toyota Motor Corporation* y fabricar el primer automóvil.

Con base a la heredada filosofía *Jidoka*, y luego de años de mejoras continuas, Kiichiro Toyoda crea

el sistema JIT[4] (siglas de la frase en inglés *Just-In-Time*, justo a tiempo), el cual tiene como finalidad la producción solo de lo que se necesita para el siguiente proceso en un flujo continuo, es decir, se fabrica lo que el cliente demanda, solo cuando el cliente lo solicita, reduciendo minuciosamente el desperdicio que se genera.

En 1936, Kiichiro suma a la empresa familiar a su primo Eiji Toyoda, quien acababa de graduarse de la carrera ingeniería mecánica. Eiji, junto con Taiichi Ohno, un ingeniero de la fabricación, comienzan a desarrollar en 1948 el Sistema de Producción de Toyota (TPS, siglas que provienen de su nombre en inglés *Toyota Production System*). Este surge de la conjunción de las filosofías Jidoka y JIT, la primera de ellas facilita la visualización de fallas proporcionando calidad desde el proceso automático de fabricación, mientras que la segunda se enfoca en la mejora de la productividad al solo fabricar lo que el cliente solicita.

A finales del siglo XX y teniendo como base el sistema TPS, nace la filosofía *Lean Manufacturing*, la cual añade cinco principios para establecer los patrones que mejorarán el funcionamiento del sistema de producción de la empresa. El primero de ellos es el de añadir o realizar actividades que agreguen valor al cliente. El segundo principio se conoce como *Map Value Stream* o flujo de valor, el cual trata sobre el ciclo total de la actividad. El tercer principio es el de crear un flujo, simplemente implica una secuencia continua de actividades a lo largo del proceso, sin paradas, sin interrupciones, sin desconexiones, ni bucles inversos. En cuarto lugar figura el principio que establece el sistema pull (o como su traducción lo indica, atracción) que significa que las cosas se hacen cuando se requiere que se hagan, no antes, tal como lo establece la filosofía JIT. Y por último, el principio que hace referencia a la búsqueda de la perfección mediante la aplicación de una mejora continua [Carreira, 2005].

Adoptando las prácticas y principios *Lean*, las empresas de diferentes rubros mejoraron drásticamente la productividad, los plazos de entrega de los clientes, la calidad del producto y la satisfacción del cliente. Como resultado, en 2005 los plazos de entrega promedio se redujeron a menos de tres semanas y más del 95 % de los pedidos se enviaban a tiempo, contrastando con lo sucedido en 1980, donde los plazos de entrega promedio eran de seis semanas y menos del 70 % se enviaban a tiempo. Aquellas organizaciones que no implementaron las prácticas *Lean* perdieron participación en el mercado e incluso muchas de ellas cerraron [Kim *et al.*, 2014].

En la actualidad, la industria de software está atravesando la denominada *"Revolución DevOps"*, la cual nace como consecuencia de procesos históricos y de construcciones colectivas, siendo impulsada por los mismos principios que promovieron los cambios de la industria manufacturera en el siglo pasado. Sin embargo, a diferencia de las filosofías mencionadas anteriormente, *DevOps* no solo pretende mejorar el desarrollo de software desde el perfeccionismo de las herramientas y de las prácticas, sino, por el contrario, su principal foco es derribar la brecha que ha existido entre el grupo de desarrolladores (programadores) y los profesionales de operaciones (quienes administran la infraestructura, como por ejemplo, bases de datos, redes, administradores de sistemas), tal y como su nombre lo demuestra uniendo las abreviaturas *Dev*, proviene de *Development* (desarrollo en español) y *Ops* de *Operations* (operaciones en español).

El objetivo de *DevOps* es mejorar la colaboración, fomentando una comunicación más abierta entre los equipos, intercambiando ideas, procesos y herramientas, para reducir el riesgo de falta de comunicación y malentendidos. De esta forma, les permite a los desarrolladores y operadores dejar atrás su miedo al cambio y alcanzar así su meta en común: construir productos más eficientemente y hacer entregas de valor a los clientes frecuentemente.

Este nuevo cambio radical en la industria busca acelerar el desarrollo de software por medio de la

[4]https://global.toyota/en/company/vision-and-philosophy/production-system/

automatización de los procesos que intervienen en el mismo. Logrando mejorar la calidad gracias a la detección rápida y anticipada de errores, tal y como lo presentaba la filosofía *Jidoka*, incrementando la satisfacción del cliente mediante la entrega continua de productos con valor y mejorando la productividad del empleado. Asimismo, esta automatización requiere de la colaboración de todas las partes involucradas en el proceso, implicando la utilización de las herramientas en conjunto. Las prácticas más importantes que comprende este movimiento, son la integración continua, la entrega continua y el despliegue continuo. Ambas llevadas a cabo por medio de la codificación de *pipelines*.

2.2.1. Pipeline

La *pipeline*, en español tubería o canalización, es un script que indica un conjunto de instrucciones por donde se desplaza el código almacenado en un repositorio. Contiene una serie de pasos ordenados tales como tareas de la compilación, prueba e implementación del software que se ejecutarán automáticamente cuando se ejecute la *pipeline* que los contiene.

Las *pipelines* mayoritariamente se encuentran desarrolladas en un lenguaje declarativo[5], y comienzan a ejecutarse luego del disparo de un evento (*trigger*). Este evento puede ser, por ejemplo, cuando se realiza una publicación en el repositorio de código fuente, por tanto, se lanzará automáticamente la ejecución de la *pipeline*, la cual primero realizará una copia local del código y continuará con la ejecución de los pasos detallados en su script.

2.2.2. Integración continua

Martin Fowler se refiere a la integración continua (*Continuous Integration* en inglés, o conocido comúnmente por sus siglas CI) como la práctica de desarrollo de software donde los miembros de un equipo incorporan los cambios frecuentemente en un repositorio de código fuente en común [Fowler y Foemmel, 2006]. Cada integración es verificada por una compilación automatizada (incluyendo el *testing*), para detectar errores de esta misma tan rápido como sea posible. Si alguna actividad falla, el desarrollador será notificado para que pueda reparar el error. En caso de no poder solucionarlo, podrá volver a una versión anterior estable del software. Varios equipos encuentran que este enfoque reduce significativamente los problemas de integración, y permiten desarrollar software cohesivo más rápidamente.

El mayor beneficio de la integración continua es la reducción del riesgo. Si bien no elimina los errores, al integrar continuamente partes pequeñas de código se vuelve fácil localizarlos y eliminarlos, además de permitir realizar una comparación con la versión actual del sistema y una anterior que carecía de errores. En todo momento se sabe dónde se está, qué funciona, qué no y los errores (*bugs*) pendientes que tiene el sistema.

Otro beneficio que menciona Fowler es que la entrega frecuente permite a los usuarios obtener nuevas funciones más rápidamente, brindando comentarios sobre las mismas y, en general, siendo más colaborativos en el ciclo de desarrollo. Además, empleando esta práctica, se evita la acumulación de errores al realizar integraciones de código diariamente [Fowler y Foemmel, 2006].

Los principios que otorgan valor a la integración continua son [Duvall *et al.*, 2007]:

[5] Un **lenguaje declarativo** es un tipo de lenguaje de programación que describe lo que debe realizar un programa, sin especificar cómo debe lograrlo. Por ejemplo, imágenes de Docker[6], manifiestos de Kubernetes, código HTML, *pipelines*, etc.

[6] Los conceptos de Docker y Kubernetes son abordados en las secciones 2.3.1 y 2.4.1 respectivamente.

- **Reducir los riesgos**: Del mismo modo en que lo describió Fowler, los defectos se detectan y corrigen con mayor rapidez gracias a la integración continua. El estado del software se vuelve medible mediante la incorporación de pruebas e inspección, y se reducen las suposiciones al reconstruir y probar el software en un entorno limpio utilizando el mismo proceso de forma continua. Algunos de los riesgos que ayuda a mitigar son:
  - Falta de software cohesivo e implementable.
  - Descubrimiento tardío de defectos.
  - Software de baja calidad.
  - Falta de visibilidad del proyecto.
- **Reducir los procesos manuales repetitivos**: Reducir los procesos manuales repetitivos ahorra tiempo, costos y esfuerzo, liberando a las personas para que realicen un trabajo de mayor valor.

  Al automatizar la integración continua se tiene mayor capacidad para garantizar que el proceso se ejecuta de la misma manera cada vez, que se sigue un proceso ordenado y que los procesos se ejecutarán cada vez que se publique código en el repositorio (*commit*), o que se ejecute el *trigger* configurado.
- **Generar software desplegable a todo tiempo y en todo lugar**: El software desplegable es el activo más tangible para los clientes o usuarios. Los proyectos que no adoptan esta práctica pueden esperar hasta antes de la entrega para integrar y probar el software, lo cual puede retrasar un lanzamiento, causar defectos, evitar la reparación de estos, y, en casos más graves, significa el final del proyecto. A diferencia de estos, en los proyectos con integración continua, los pequeños cambios se integran con el resto del código de forma regular y si surge algún problema, los desarrolladores son informados y aplican correcciones al software inmediatamente.
- **Permitir una mejor visibilidad del proyecto**: Un sistema de integración continua puede brindar información oportuna sobre el estado de compilación reciente y las métricas de calidad, permitiendo realizar decisiones efectivas. Además, dado que las integraciones ocurren con frecuencia, proporciona la capacidad de detectar tendencias en el éxito o el fracaso de la construcción, la calidad general y otra información pertinente del proyecto se vuelve posible.
- **Establecer una mejor confianza en el producto de software desde el equipo de desarrollo**: Con cada compilación, el equipo sabe que se ejecutan pruebas sobre el software para verificar el comportamiento, que se cumplen los estándares de codificación y diseño del proyecto y que el resultado es un producto funcionalmente comprobable. Sin integraciones frecuentes, algunos equipos pueden sentirse sofocados porque no conocen los impactos de los cambios en su código.

2.2.2.1. Componentes de la integración continua

Para llevar a cabo la práctica de la integración continua se necesita de tres componentes (ver Figura 2.4): un sistema de administración de código fuente, un servidor de CI, y la *pipeline*.

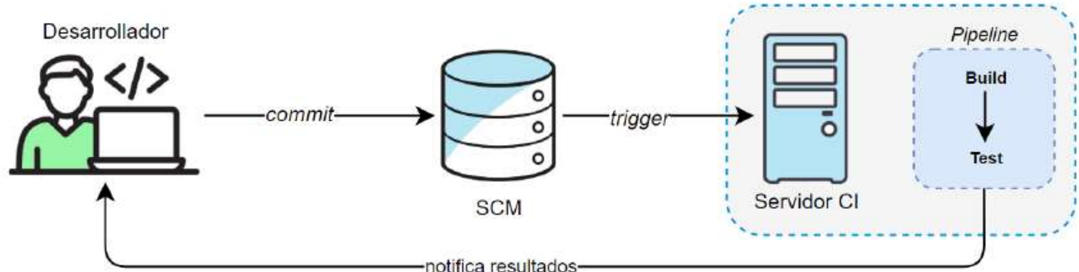


Figura 2.4: *Esquema que ilustra la práctica de integración continua.*

Sistema de administración de código fuente

El sistema de administración de código fuente, o también conocido por su siglas en inglés SCM (*Source Code Management*)[7], es una herramienta de software que los programadores utilizan para administrar el código fuente. Proporciona visualización de los cambios realizados por cada colaborador y permite resolver conflictos al fusionar distintas actualizaciones de un mismo código [Duvall *et al.*, 2007].

Independientemente de las diferentes herramientas que se encuentran en el mercado, la gestión del código fuente utiliza una serie de términos en común, algunos de ellos son los siguientes:

- *Repository* o *"repo"* (Repositorio): Servidor donde se almacenan los archivos actuales e históricos.
- *Trunk* (Tronco): Línea principal de desarrollo de un árbol de archivos bajo control de versiones.
- *Branch* (Rama): Copia del código creado a partir del tronco de un proyecto. Permite trabajar en varias copias del mismo archivo de diferente maneras, independientemente unas de otras.
- *Checkout* (Verificación): Crear una copia local desde el repositorio. Se puede especificar la versión u obtener la última. También se puede utilizar como sustantivo para describir la copia de trabajo.
- *Commit* (Publicar): Escribir o fusionar los cambios, realizados en la copia local, en el repositorio. Contiene metadatos, normalmente la información del autor, hora, fecha y un mensaje de confirmación que describe el cambio.
- *Pull Request* (Solicitud de publicación): Enviar cambios que se han realizado en el repositorio y solicitar que el administrador fusione esos cambios en la rama principal.

GitHub

GitHub[8] es un SCM basado en la web que proporciona alojamiento para el control de versiones y la colaboración en el desarrollo de software. Utiliza Git[9], un sistema de control de versiones distribuido, como backend para almacenar y administrar el código fuente, permitir a múltiples usuarios trabajar en el mismo código simultáneamente, tener un seguimiento de los cambios y revertir a versiones previas en caso de ser necesario.

GitHub proporciona una interfaz fácil de usar para gestionar y realizar un seguimiento de los cambios en el código, así como herramientas para la colaboración tales como, solicitudes de incorporación de cambios, revisión de código y seguimiento de problemas.

[7] Es sinónimo del Sistema de Control de Versiones (VCS, siglas de su nombre en inglés *Version Control System*), por lo que la terminología es intercambiable.

[8] https://github.com/

[9] https://git-scm.com/

Servidor de CI

Un servidor de CI ejecuta una compilación de integración cuando un cambio es publicado en el repositorio de control de versiones. Normalmente se debe configurar el intervalo de tiempo en el cual el servidor va a chequear el repositorio, o configurar el trigger necesario para que se dispare la ejecución [Duvall *et al.*, 2007]. Por ejemplo, algunos servidores permiten que la ejecución se lance tan pronto como se realice un *commit*, y esto puede delimitarse a que sea un cambio en un archivo o en un directorio en particular.

El servidor de CI recuperará los archivos de origen mediante un *checkout* y ejecutará la *pipeline* notificando a los desarrolladores, por medio de la plataforma indicada, sobre el resultado de la misma, para que puedan tomar medidas al respecto, si es necesario. Como ejemplos de servidores de CI que al momento de escribir este trabajo son de amplia difusión, se encuentran Jenkins, CircleCI, Bamboo, Gitlab CI, TeamCity, entre otros.

A continuación, se describe Jenkins, el cual fue utilizado en el desarrollo de la parte práctica de este trabajo final.

Jenkins

Jenkins[10] es un servidor de automatización de código abierto escrito en Java. Posee un soporte comunitario muy activo y una gran cantidad de complementos (*plugins*). Es una de las herramientas más populares para implementar procesos de integración y despliegue continuos.

Algunas de las características más interesantes de Jenkins son:

- **Independiente del lenguaje**: Jenkins tiene muchos complementos que admiten la mayoría de los lenguajes y marcos de programación. Además, dado que puede emplear cualquier comando de consola y cualquier software, es adecuado para todos los procesos de automatización imaginables.
- **Extensible por complementos**: Jenkins tiene una gran comunidad y muchos *plugins*[11] disponibles (más de mil). También permite desarrollar complementos propios para personalizar Jenkins según las necesidades requeridas.
- **Portable**: Jenkins está desarrollado en Java, lo que le otorga la capacidad de poder ejecutarse en cualquier sistema operativo.
- **Admite la mayoría de las herramientas de SCM**: Jenkins se integra con prácticamente todas las herramientas de creación o gestión de código fuente que existen.
- **Distribuido**: Jenkins tiene un mecanismo incorporado para el modo maestro-esclavo que distribuye su ejecución entre varios nodos, ubicados en varias máquinas. También puede utilizar entornos heterogéneos; por ejemplo, diferentes nodos pueden tener diferentes sistemas operativos instalados.

Arquitectura de Jenkins

La arquitectura de Jenkins, como se mencionó anteriormente, se basa en un modelo maestro-esclavo, donde el nodo maestro es responsable de administrar todos los trabajos y distribuirlos a los nodos esclavos para su ejecución, comunicándose entre sí utilizando el protocolo TCP/IP.

[10]https://www.jenkins.io/
[11]Disponibles en https://plugins.jenkins.io/

Está diseñada para proporcionar una plataforma flexible, escalable y extensible que permita automatizar los procesos de desarrollo de software. Su modelo maestro-esclavo proporciona a los desarrolladores la capacidad de distribuir los procesos de compilación entre varios nodos y plataformas, lo que convierte a Jenkins en una solución ideal para proyectos a gran escala [Jovanović, 2020].

Los componentes que conforman esta arquitectura son los siguientes:

- **Nodo maestro**: Componente central de la arquitectura de Jenkins. Administra todo el sistema, incluida la programación de trabajos, la administración de complementos y la administración de usuarios. También maneja la interfaz de usuario y recibe comentarios de los procesos de construcción.
- **Nodo esclavo (Agente)**: Son máquinas de trabajo que ejecutan los procesos de compilación asignados por el nodo maestro. Una instalación de Jenkins puede tener varios nodos trabajadores que se pueden distribuir en diferentes plataformas y sistemas operativos, tal como ilustra la Figura 2.5. Asimismo, el proyecto se puede configurar para elegir una máquina esclava específica.

  Los agentes pueden ser una máquina física, una máquina virtual, una instancia de computación en la nube, una imagen de Docker o un clúster de Kubernetes.

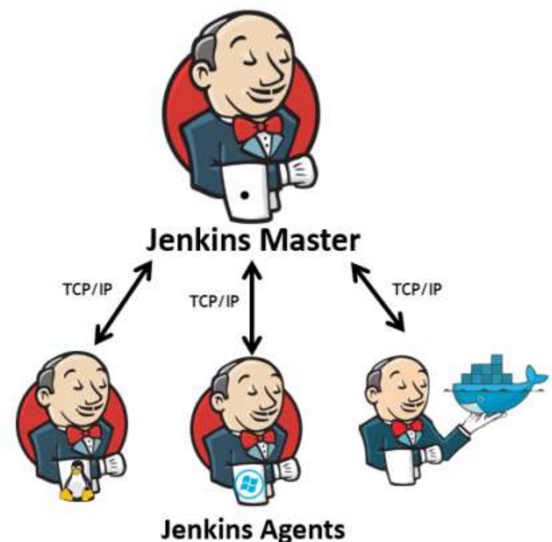


Figura 2.5: Arquitectura maestro-esclavo del servidor Jenkins.
Nota: Adaptado de [Motoskia, 2018].

- **Plugins**: Gran colección de complementos que amplían la funcionalidad del servidor. Se pueden instalar y configurar desde el panel de control de Jenkins.
- **Job/Proyecto**: Es una tarea que debe ejecutarse en Jenkins. Un *job*, o trabajo, puede ser tan simple como compilar el código fuente o ejecutar una prueba unitaria, o tan complejo como construir e implementar una aplicación completa.
- **Pipeline**: Secuencia de *jobs* que se ejecutan en un orden particular.

Jenkins Pipeline
Jenkins Pipeline (o simplemente *"pipeline"*) es un conjunto de *plugins* que soportan la implementación de una *pipeline* de integración y despliegue continuo dentro de Jenkins. Al archivo que almacena esta

configuración, se lo conoce como `Jenkinsfile`, y contiene el paso a paso del proceso automático de obtener software desde el repositorio, hasta que es desplegado para ser llevado a los usuarios y clientes.

El archivo `Jenkinsfile` se puede escribir de dos maneras. Una de ellas es utilizando la sintaxis del lenguaje de programación **Groovy**[12], siendo esta la primera en haberse empleado para el archivo. Tiene una gran flexibilidad y no hay una estructura establecida para escribir el código de una *pipeline*.

Por otro lado, tenemos la **sintaxis declarativa**, la cual se añadió hace pocos años en la plataforma. Si bien, no es tan poderosa como la primera, es más fácil de utilizar, ya que tiene un formato predefinido para escribir el código [Jovanović, 2020].

Algunos de los elementos básicos de una *pipeline* en Jenkins son:

- **`Stage`** (Etapa): Es un bloque que define un subconjunto de tareas conceptualmente distintas, realizadas a lo largo de la *pipeline*. Utiliza varios complementos para visualizar o presentar el estado/progreso de la misma. Se recomienda utilizar, como mínimo, las etapas de compilación (*Build*), pruebas (*Test*) y despliegue (*Deploy*).
- **`Step`** (Paso): Es una sola tarea. Fundamentalmente, un *step* indica a Jenkins qué hacer en un momento puntual. Por ejemplo, ejecutar un comando '`echo`'. La sección *steps* se define por uno o más pasos a ser ejecutados en una determinada etapa (`stage`).
- **`Agent`** (Agente): La sección del agente a nivel global es obligatoria, y la misma especifica dónde se ejecutará toda la *pipeline*. También puede ser declarado a nivel de *stage*, lo cual es opcional, indicando dónde se ejecutará esa etapa en particular. En caso de querer utilizar agentes en las etapas, se debe indicar el agente global como `any` y así se ejecutará en el nodo disponible. Vale aclarar que diferentes etapas pueden ejecutarse sobre diferentes agentes.

Un ejemplo sencillo de un archivo `Jenkinsfile`, es el que se muestra a continuación.

```
pipeline {
    agent any

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

[12] https://es.wikipedia.org/wiki/Groovy_(lenguaje_de_programaci%C3%B3n)

```
            }
        }
    }
```

En esta *pipeline*, se describe un agente global de tipo `any`, lo que significa que estos pasos serán ejecutados en el nodo que Jenkins encuentre disponible. Se cuenta con tres etapas diferentes, englobadas dentro de la sección `stages`, cada una conformada por un solo paso que escribe, utilizando el comando `echo`, lo que se haría en esa etapa (`stage`).

Una ventaja de almacenar el archivo `Jenkinsfile`, en el sistema administrador de código fuente, es que por medio de su versionado es posible obtener el historial que permita volver a una versión anterior, en caso tal de que un cambio de codificación en la *pipeline* haya perjudicado su funcionamiento.

2.2.2.2. Mejores prácticas de la integración continua

Para realizar una correcta integración continua, [Duvall *et al.*, 2007] recomienda tener en cuenta las siguientes siete prácticas :

1. **Realizar *commit* de código frecuentemente**
   Se deben prevenir largos períodos de tiempo sin realizar *commits* al repositorio. Se recomienda realizar pequeños cambios y subir el código luego de completar cada tarea. Además, se debe prevenir que varias personas realicen publicaciones al mismo tiempo y así no generar errores debido a las colisiones en el código.

2. **No subir código dañado**
   Antes de realizar un *commit* se recomienda ejecutar compilaciones privadas y evitar así, la subida de código dañado.

3. **Solucionar las compilaciones dañadas inmediatamente**
   Se debe asignar una alta prioridad a esta tarea, ya que el objetivo de la integración continua es desarrollar sobre una base estable. Para solucionarlo rápidamente, se puede volver a una versión anterior que posea un estado exitoso.

4. **Escribir pruebas de desarrollador automatizadas**
   Para detectar errores de manera rápida y eficiente lo ideal es incluir pruebas automatizadas en el proceso de CI.

   El primero de los siete principios de las pruebas, y en concordancia con lo que se conoce como un dicho de Edsger Dijkstra (premio Turing en 1972), establece que: *"Las pruebas demuestran la presencia de defectos, no su ausencia"*. Es decir, no es posible asegurar que un sistema se encuentra libre de fallas, pero gracias a la aplicación de las pruebas, se reduce considerablemente la cantidad de defectos que son encontrados, y así se minimiza el riesgo de un mal funcionamiento en nuestro software.

5. **Se deben pasar todas las pruebas e inspecciones**
   Aceptar código que no pase las pruebas e inspecciones origina software de baja calidad. Como se enuncia en la práctica anterior, resolviendo los errores encontrados por las pruebas se reduce el riesgo de un mal funcionamiento en el software.

6. **Ejecutar compilaciones privadas**
   Para prevenir compilaciones dañadas, luego de realizar las pruebas unitarias, cada desarrollador debería emular una compilación de integración en el entorno de desarrollo integrado (IDE, siglas

provenientes de su nombre en inglés *Integrated Development Environment*) de su estación de trabajo. De esta manera, el código subido por cada desarrollador al repositorio en común, tiene menos probabilidades de fallar en el servidor de integración.

7. **Evitar obtener código roto/dañado**
   Es fundamental que los desarrolladores tengan conocimiento del estado del código que se encuentra en el repositorio de control de versiones. Sin embargo, puede suceder que un desarrollador no revise la notificación de la compilación dañada. En este caso, se aconseja diseñar una solución al error para que se pueda compilar y probar, en lugar de invertir tiempo en revisar el código más reciente del repositorio.

2.2.3. Entrega continua

La entrega continua (*Continuous Delivery* en inglés o CD) se basa en el primer principio del Manifiesto Ágil: *"Nuestra mayor prioridad es satisfacer al cliente mediante entregas tempranas y continuas de software con valor"* [Beck *et al.*, 2005].

[Fowler, 2013] se refiere a la entrega continua como la disciplina del desarrollo de software donde el mismo se construye de una manera tal que permite crear, probar y empaquetar automáticamente actualizaciones de software, permitiendo implementarse fácilmente en producción. Esta práctica brinda la capacidad de lanzar nuevas versiones funcionales de un software varias veces al día. Sin embargo, la implementación real de actualizaciones en producción aún requiere la intervención manual (ver Figura 2.6) y la aprobación de un administrador de versiones u otras partes interesadas.

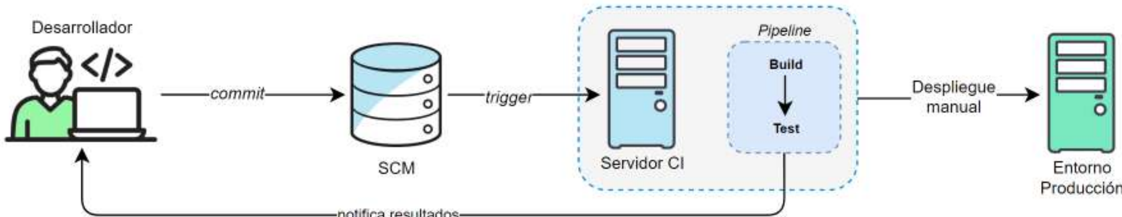


Figura 2.6: *Esquema que ilustra la práctica de entrega continua.*

Se logra la entrega continua por medio de la integración continua de software, construyendo ejecutables y ejecutando pruebas automatizadas para detectar problemas. Proporciona la infraestructura y los procesos para hacer que las implementaciones sean más eficientes.

De acuerdo a [Fowler, 2013] y a [Humble y Farley, 2010], los principales beneficios de esta práctica son:

- **Reduce el riesgo en el despliegue**: Al desplegar pequeños cambios la probabilidad de que algo surja mal es menor, y si aparece un problema es más fácil solucionarlo realizando un *rollback* (volver hacia la última versión estable).
- **Se obtiene *feedback* de los usuarios más rápidamente**: La entrega frecuente de software a los usuarios proporciona comentarios más rápidos, que permiten descubrir qué tan valiosa ha sido la entrega, y así la organización puede responder más rápido a las necesidades cambiantes de los clientes y a las condiciones del mercado.
- **Calidad mejorada**: Al automatizar el proceso de testing se ayuda a garantizar que las actualizaciones de software se prueben exhaustivamente antes de que se implementen en producción.

Esto reduce el riesgo de que se introduzcan errores y otros problemas en el entorno, y mejora la calidad general del software.

- **Mayor colaboración**: Permite que los equipos trabajen juntos de manera más efectiva al proporcionar una plataforma común para probar, revisar e implementar actualizaciones de software.
- **Mejor trazabilidad**: Al realizar un seguimiento automático de los cambios en el código y de la configuración del software, proporciona un historial detallado de los mismos, facilitando la identificación y resolución de problemas.
- **Procesos de implementación mejorados**: Ayuda a estandarizar y automatizar el proceso de implementación, lo que reduce el riesgo de error humano, mejora la confiabilidad y repetibilidad de las implementaciones.

En general, la entrega continua ayuda a las organizaciones a entregar actualizaciones de software más rápidamente, con mayor confianza y mejor calidad, fomentando la colaboración del equipo de desarrolladores y operaciones.

2.2.4. Despliegue continuo

El despliegue continuo (*Continuous Deployment*, o CD) implica que, luego de finalizar la ejecución de la *pipeline* de CI, comiencen a ejecutarse los procesos procedentes del despliegue. Esta práctica tiene como objetivo realizar lanzamientos automáticos de software a medida que se realizan actualizaciones de código, para que sean implementados directamente en el entorno de desarrollo o integración, consiguiendo su pronta visibilización.

La gran ventaja que conlleva la aplicación de esta práctica es la entrega temprana de productos con valor a los clientes, ya que los mismos deben pasar los procesos de testeo antes de ser desplegados hacia el entorno, asegurando su calidad. Además, si uno de los procesos que conforman la *pipeline* falla, los desarrolladores/operadores serán notificados y el despliegue no se realizará.

Al igual que la integración continua, los beneficios que proporciona el despliegue continuo son: la detección temprana de errores, la disminución de costos de desarrollo, la simplificación del trabajo colaborativo y, la mejora de la comunicación y colaboración entre equipos. Cabe destacar que, de igual forma, comparte las ventajas que conlleva la aplicación de la práctica de entrega continua.

El despliegue continuo muchas veces es confundido con la entrega continua, no obstante, su diferencia radica en que, en el primero de estos cada cambio que se realice lanzará la ejecución de la *pipeline* y, si pasa todas las pruebas, automáticamente será publicado en el entorno que se desee, resultando de esta manera varios despliegues por día (ver Figura 2.7). En cambio, en la entrega continua, los despliegues siguen necesitando de la intervención manual.

> NOTA: Para realizar despliegue continuo, se debe implementar previamente la práctica de entrega continua.

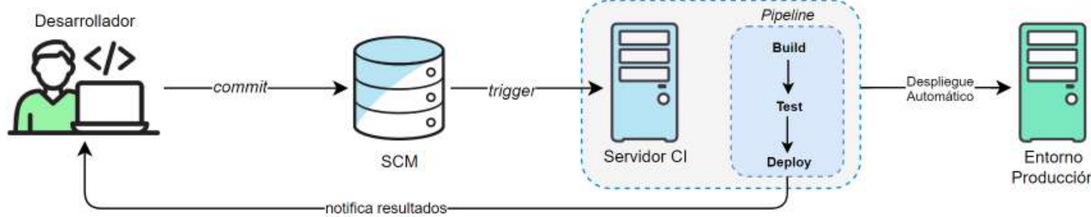


Figura 2.7: *Esquema que ilustra la práctica de despliegue continuo.*

2.2.5. Infraestructura como código

La **infraestructura como código** (*Infrastructure as Code*, IaC) se define como un enfoque de las tecnologías de la era de la nube, para administrar y aprovisionar infraestructura dinámica mediante código, en lugar de procesos manuales, teniendo la capacidad de tratar sus elementos como si fueran datos [Morris, 2020].

Con el hardware en la nube, la mayoría de los recursos, en cierto modo, son software. El equipo de operaciones ya no debe ocuparse del hardware físicamente, en cambio, debe adaptar las habilidades y prácticas de ingeniería de software (SCM, integración continua, despliegue continuo) que han demostrado su eficacia en el desarrollo de sistemas, que le permitan codificar rápida, ágil y colaborativamente sistemas complejos [Arundel y Domingus, 2019].

La infraestructura como código está estrechamente relacionada con la cultura y prácticas de *DevOps*, ya que ayuda a automatizar la implementación de la infraestructura (ver Figura 2.8), mejorando los procesos de implementación, reduciendo los errores manuales y el tiempo de inactividad de los recursos. Este enfoque logra el objetivo central de permitir la colaboración y comunicación entre los equipos de desarrollo y operaciones, puesto que trabajaran con las mismas herramientas para administrar y controlar versiones del código, tanto de la infraestructura como del sistema. *DevOps* y la infraestructura como código, son prácticas complementarias que ayudan a las organizaciones a entregar software de alta calidad de manera más rápida y eficiente.

Las herramientas de IaC son necesarias para que el proceso de construcción y configuración de la infraestructura se vuelva más competitivo y efectivo, reduciendo los costos y el esfuerzo involucrado. Algunas de las más utilizadas a la fecha son Terraform[13], Ansible[14], AWS Cloud Formation[15], Azure Resource Manager[16], y Google Cloud Deployment Manager[17] [Singh, 2023].

[13] https://www.terraform.io/
[14] https://www.ansible.com/
[15] https://aws.amazon.com/es/cloudformation/
[16] https://azure.microsoft.com/es-es/get-started/azure-portal/resource-manager/
[17] https://cloud.google.com/deployment-manager/docs?hl=es-419

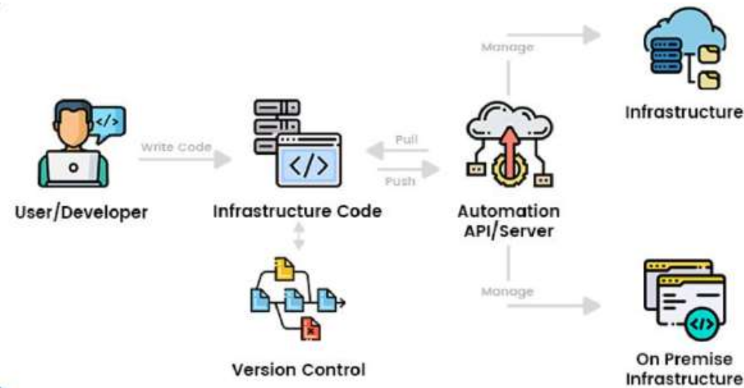


Figura 2.8: *Esquema de la infraestructura como código.*
Nota: Extraído de [Mistry, 2022].

2.3. Contenedores

Los contenedores son cajas rectangulares de grandes dimensiones que se utilizan para facilitar el transporte de mercancías, tanto por vía marítima o fluvial, como por vía terrenal y aérea. De acuerdo a [The Maritime Executive, 2016], los contenedores fueron creados en la década de los años 50 por el empresario estadounidense Malcom McLean, quien luego de observar a los trabajadores de los puertos marítimos cargando y descargando manualmente la mercancía por horas, desde los camiones hacia los barcos y viceversa, comenzó a idear un método que repercutiera positivamente en la eficiencia del proceso, reduciendo los costos y el tiempo del mismo. *"¿No sería grandioso, si mi trailer pudiera simplemente levantarse y colocarse en el barco?"*, pensó McLean.

Figura 2.9: *Barco de contenedores.*
Nota: Extraído de [Xataka, sf].

El uso de contenedores revolucionó la industria del transporte marítimo al permitir el movimiento fácil y eficiente de mercancías en todo el mundo (ver Figura 2.9). También permitieron una mayor estandarización y automatización en la carga y descarga de mercancías, lo que resultó en importantes ahorros de costos y una mayor eficiencia en el comercio mundial.

¿Qué relaciona este concepto de "contenedor" conocidos por todos, con el desarrollo de software? Pues bien, para implementar software, no solo se necesita el código fuente en sí, sino también sus dependencias. Esto significa bibliotecas, intérpretes, compiladores, extensiones. De igual manera se necesita su configuración, detalles específicos para la publicación (en el caso de software web), claves de licencia, contraseñas de la base de datos, etc.; todo lo que convierte el software en bruto en un servicio utilizable. Por esta razón, es que la industria del software se basó en la invención de McLean para crear

sus propios contenedores, con la finalidad de transportar aplicaciones de una máquina a otra, o entre diferentes entornos, en un formato estándar de empaque y distribución, permitiendo costos más bajos y facilidad de manejo.

Un **contenedor de software** es un paquete ejecutable independiente y liviano que incluye todo lo necesario para ejecutar una pieza de software, incluido el código, librerías, variables de entorno y archivos de configuración [Docker Inc., 2023]. También permiten una mejor utilización de los recursos y el aislamiento de las aplicaciones, ya que cada contenedor se ejecuta en su propio entorno aislado. Esto los convierte en una solución popular para implementar y ejecutar microservicios, y para implementar *pipelines* de integración y despliegue continuos (CI/CD).

DATO: El término *"contenedor de software"* inicialmente surge a finales de los años 70 y a comienzo de los 80s con la creación de `chroot`[18]en los sistemas Unix. Sin embargo, el concepto moderno, se populariza luego del año 2013 con las tecnologías de contenerización Docker y Kubernetes.

2.3.1. Docker

Docker[19] (traducido al español como "trabajador portuario") fue lanzado como software de código abierto en Mayo de 2013, por su creador Solomon Hykes, con una breve presentación[20] de menos de diez minutos en la *Conferencia de Desarrolladores de Python*[21] en Santa Clara, California [Matthias y Kane, 2018].

Hykes se veía frustrado con los desafíos de implementar aplicaciones de manera consistente en diferentes entornos. Su primera empresa fundada fue dotCloud, la cual ofrecía una plataforma como servicio (*Platform as a Service*, PaaS). Esta plataforma permitía a los desarrolladores crear y desplegar fácilmente aplicaciones codificadas en varios lenguajes de programación (incluyendo Python, Ruby, Java y PHP). De a poco fue mejorando la solución, hasta que finalmente surge el proyecto de Docker, cuyo objetivo fue el de permitir a los desarrolladores empaquetar sus aplicaciones y dependencias en un contenedor portátil, que luego se podría implementar y ejecutar fácilmente en cualquier entorno, independientemente de la infraestructura subyacente, siempre y cuando este sea compatible con Docker.

Actualmente, se trata de la tecnología de contenedores de mayor repercusión, que facilita la implementación y ejecución de aplicaciones en diferentes sistemas, sea en servidores locales (*on-premises*), en la nube, o incluso en entornos híbridos.

2.3.1.1. Arquitectura de Docker

Docker utiliza una arquitectura cliente-servidor, donde el servidor es el *daemon* del contenedor, responsable de administrar el ciclo de vida de los contenedores, abarcando su creación, ejecución, eliminación, e incluso la extracción de imágenes de un registro. Docker proporciona una interfaz de línea de comandos (CLI) que permite al cliente interactuar con el *daemon* a través de una API REST, empleando sockets UNIX o una interfaz de red.

[18]**Ch**ange **root** (chroot) es un comando Unix que permite ejecutar un proceso bajo un directorio raíz simulado, de manera que el proceso no puede acceder a archivos fuera de ese directorio.

[19]https://www.docker.com/

[20]https://www.youtube.com/watch?v=wW9CAH9nSLs

[21]https://us.pycon.org/2023/

Cabe destacar que Docker permite la ejecución del *daemon* y el cliente en el mismo host, como así también, la conexión del cliente local a un servidor remoto que se encuentra en ejecución [Turnbull, 2018].

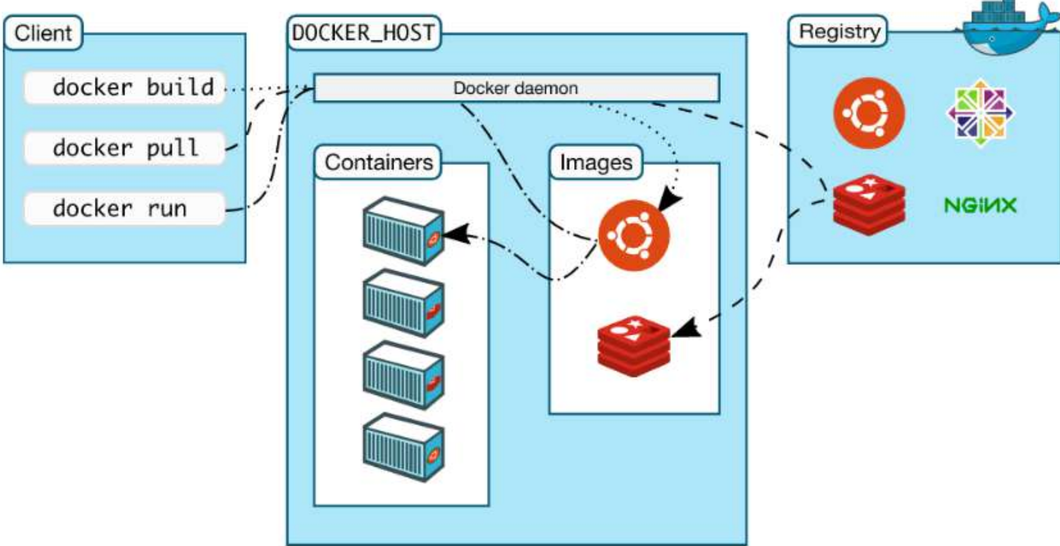


Figura 2.10: *Arquitectura de Docker.*
Nota: Extraído de la documentación oficial de Docker [Docker Inc., 2023].

Los componentes de la arquitectura ilustrada en la Figura 2.10 son los siguientes [Turnbull, 2018]:

- **Docker Daemon** (`dockerd`): Escucha las solicitudes de la API y administra los objetos de Docker. El *daemon* puede comunicarse con otros *daemons* para administrar los servicios de Docker.
- **Cliente Docker** (`docker`): Se comunica con Docker por medio de la utilización de comandos, intrucciones que son enviadas al *daemon* para ser llevadas a cabo. Algunas de las líneas de comandos más utilizadas son:
  - `docker run`, permite ejecutar un contenedor
  - `docker ps`, permite listar todos los contenedores que se encuentran en ejecución
  - `docker start` o `docker stop`, para iniciar o detener un contenedor
  - `docker image`, para crear, eliminar, listar imágenes
  - `docker pull`, para descargar una imagen de un registro configurado
  - `docker push`, permite subir una imagen a un registro configurado
- **Registro** (registry): Un registro de contenedores almacena imágenes de Docker. La misma compañía ofrece Docker Hub[22] como registro público y gratuito para sus imágenes, el cual puede ser privado si así lo requiere (permite hasta un repositorio consumiendo el plan gratis, y cantidades ilimitadas en la versión paga). Asimismo, actualmente en el mercado se encuentran opciones de registros de varias organizaciones, como es el caso de Azure Container Registry[23] (ACR) o Amazon Elastic Container Service[24] (Amazon ECS), ambos siendo servicios pagos que permiten el almacenamiento público y privado.

[22]https://hub.docker.com/
[23]https://azure.microsoft.com/en-us/products/container-registry
[24]https://aws.amazon.com/es/ecr/

- **Objetos**: Los principales objetos son:
  - **Imágenes de Docker**: Son el componente básico que permite lanzar contenedores, ya que es el "código fuente" de estos. Es un solo archivo binario que tiene una identificación única y contiene todo lo necesario para ejecutar el contenedor.

    El código es almacenado en un archivo de texto plano llamado `Dockerfile`, en el que cada línea es una instrucción a ejecutar por Docker, normalmente iniciando con el comando `FROM` para indicar la imagen base.

    Tienen un formato en capas, tal como se ejemplifica en la Figura 2.11, que permite la construcción de una serie de instrucciones, dependiendo cada una inmediatamente de la capa o instrucción anterior. Esta jerarquía es importante para una buena gestión del ciclo de vida de las imágenes, ya que cuando se modifica una capa, se reconstruirán todas las demás dependientes a esta, por lo tanto, las capas que deben ser modificadas con más frecuencia deben estar lo más arriba de la pila posible, para evitar trabajo computacional innecesario, y así, acelerar el proceso de ejecución.

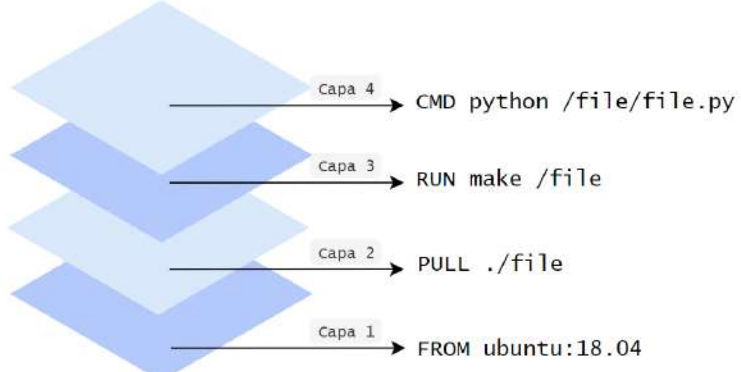


Figura 2.11: *Ejemplo ilustrativo de una imagen Docker.*
Nota: Adaptado de [Afreen, 2023].

  - **Contenedores**: Como se mencionó anteriormente, la imagen solo es un conjunto de instrucciones, siendo los contenedores el entorno de ejecución de estas. Para su construcción es indispensable indicar el nombre o URI de la imagen que lo conformará.

    Tal como en el transporte marítimo de contenedores, a Docker solo le interesa realizar el movimiento y las operaciones, sin importar qué lleva dentro cada contenedor, puede que sea una base de datos, una aplicación, o un servidor web, todos serán tratados de la misma manera de acuerdo a las operaciones indicadas por el desarrollador: crear, iniciar, detener, reiniciar, eliminar, etc.

    El siguiente comando crea y ejecuta un contenedor con la imagen indicada de Ubuntu, estableciendo un nombre aleatorio para el contenedor, ya que este no fue indicado.

```
docker run ubuntu:18.04
```

2.3.2. Contenerización vs. Virtualización

Para una mejor comprensión del concepto de contenedores se remarca la diferencia existente con las máquinas virtuales (VMs, proveniente de las siglas de su traducción en inglés, *Virtual Machines*). Ambas tecnologías son utilizadas para crear ambientes aislados donde ejecutar software, pero difieren en su funcionamiento y en su propósito de utilización.

Docker opera utilizando el kernel (núcleo) del sistema operativo local, es decir, se ejecuta sobre la CPU real, sin sobrecarga de virtualización (solo virtualizan las capas de software por encima del nivel del sistema operativo). Por esta razón, es que los contenedores son apropiados para una arquitectura de microservicios, y para aplicaciones que no requieren un alto grado de aislamiento. En tanto que, las máquinas virtuales, proveen un sistema operativo independiente, completamente virtualizado (se virtualizan hasta las capas de hardware), lo que las convierte en ideales para aplicaciones que requieren de un control total sobre la infraestructura subyacente y un aislamiento más elevado. Esta diferencia arquitectónica se observa a simple vista en la Figura 2.12.

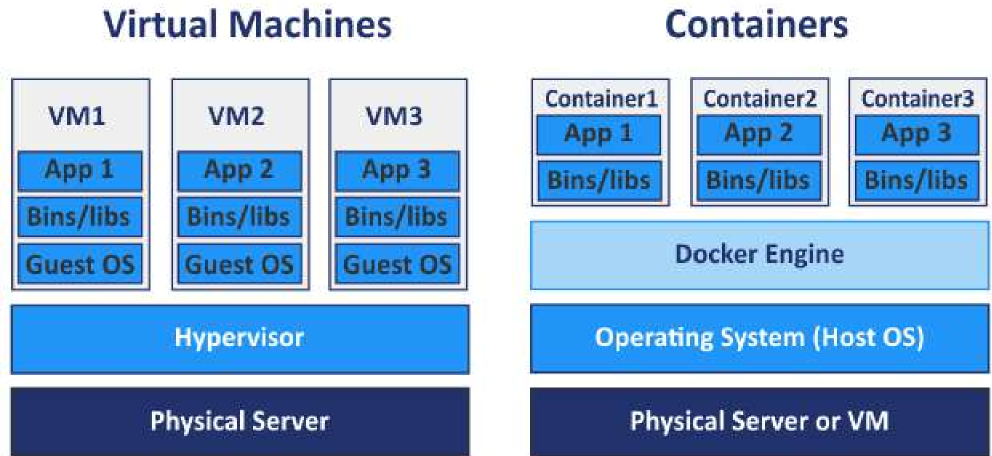


Figura 2.12: *Ejemplo de la arquitectura de máquinas virtuales versus contenedores.*
Nota: Extraído de [Nakivo, 2019].

Asimismo, hay una gran diferencia entre el tamaño de una imagen de VM y una imagen de contenedor. La primera posee un tamaño promedio de 1 GiB, mientras que la última, si está bien diseñada, puede llegar a ser 100 veces más pequeña, ya que contiene solo los archivos necesarios para la ejecución [Arundel y Domingus, 2019].

Otra ventaja de los contenedores sobre las máquinas virtuales, es la reutilización de las capas de imagen. Por ejemplo, si varios contenedores derivan de una misma imagen, esta se descargará solo una vez y cada uno de ellos podrá referenciarla. Además, en tiempo de ejecución, el contenedor ensamblará todas las capas necesarias de la imagen y descargará aquellas que no se encuentren en caché. Esta característica permite utilizar eficientemente el espacio en disco y el ancho de banda de la red.

Por último, es esencial resaltar la diferencia en el tiempo de vida de cada uno. Mientras que las VMs son un sustituto del hardware real y, frecuentemente, de naturaleza duradera, los contenedores pueden ser utilizados de manera desechable, creados particularmente para ejecutar una tarea y ser eliminados al finalizar la misma. Sin embargo, esto depende totalmente de la configuración del contenedor, ya que pueden existir durante meses, pero a diferencia de las VMs, proveen la capacidad de ser efímeros.

2.4. Orquestación de contenedores

El término **orquestador de contenedores** se refiere a un único servicio que se encarga de la programación, la orquestación y la administración de clústeres. Es el proceso de administrar y coordinar el despliegue, escalado y gestión de las aplicaciones contenerizadas. Implica administrar el ciclo de vida del contenedor, las redes, el almacenamiento y otros recursos necesarios para que la aplicación se ejecute.

La orquestación de contenedores y el despliegue continuo son conceptos estrechamente relacionados que, al utilizarlos en conjunto, permiten obtener una manera más eficiente y confiable de administración e implementación de software. Tal como se mencionó en la Sección 2.2.4, el despliegue continuo es la práctica de implementar automáticamente nuevas versiones de software en producción (o el ambiente deseado) tan pronto como estén listas, sin intervención manual. Por otra parte, la orquestación de contenedores, puede desempeñar un papel clave para facilitar el despliegue al proporcionar las herramientas e infraestructura necesarias para automatizar la implementación, el escalado y la administración de aplicaciones en contenedores.

A partir de la utilización de un orquestador, es posible automatizar el proceso de activar, aumentar o reducir la cantidad de instancias de una aplicación contenerizada, según sea necesario, e implementar actualizaciones sin tiempo de inactividad.

Como ejemplo de un software orquestador de contenedores, se menciona a la plataforma Kubernetes la cual brinda funciones integradas como estrategia de implementaciones *canary*, implementaciones *blue-green* y *rolling update deployment* o implementaciones de actualización continua[25], que son metodologías que se pueden emplear para implementar de manera segura nuevas versiones de una aplicación en producción, desplegando gradualmente la nueva versión a un subconjunto de instancias y monitoreando su comportamiento antes de implementarlo más ampliamente.

2.4.1. Kubernetes

Al inicio de los 2000s, Google comenzó a trabajar con contenedores con el fin de resolver la problemática de ejecutar una gran cantidad de sus servicios a escala global: Gmail, Google Search, Google Maps, Google App Engine, entre otros. Esto provocó la necesidad de innovar en un orquestador que permitiera fácilmente la administración de los contenedores, derivando así en el desarrollo de Borg, un sistema de administración centralizado que asignaba y programaba contenedores para que se ejecuten en un grupo de servidores [Arundel y Domingus, 2019].

Si bien Borg era un servicio poderoso, estaba ligado a tecnologías internas y patentadas de Google, que hacía imposible su expansión y lanzamiento público. Por esta razón, es que en el año 2013, luego de la presentación de Docker al mundo ,que popularizó el concepto de contenedores en el que Google venía trabajando hacía una década, Joe Beda, Brendan Burns y Craig McLuckie, vieron una oportunidad de mejora sobre los elementos más útiles de Borg.

Estos tres trabajadores de Google, fundaron en 2014 un proyecto de código abierto que combinaría las cualidades de Borg, y su sucesor Omega, con los contenedores de Docker, llamado **Kubernetes**[26] (proveniente de la palabra griega $\kappa\upsilon\beta\epsilon\nu\eta\tau\eta\zeta$, que se traduce como *"capitán"*) [Arundel y Domingus, 2019].

Un año más tarde, el proyecto es donado a *Cloud Native Computing Foundation* (CNCF), organización dedicada a promover el estado del arte en computación nativa en la nube.

Con la llegada de un orquestador de contenedores completamente gratuito y de código abierto, la adopción de contenedores y Kubernetes creció a un ritmo sorprendente. A finales de 2017, Kubernetes se había consagrado como el orquestador de facto [Arundel y Domingus, 2019].

[25]https://traefik.io/glossary/kubernetes-deployment-strategies-blue-green-canary/
[26]https://kubernetes.io/es/

2.4.1.1. Arquitectura de Kubernetes

Un clúster de Kubernetes es un conjunto de nodos (máquinas físicas o virtuales) que ejecutan y administran aplicaciones en contenedores. Emplea una arquitectura maestro-esclavo, que incluye al menos un nodo maestro (máster) que es responsable de administrar el estado del clúster y tomar decisiones sobre cómo responder a los cambios en ese estado. También, incluye un conjunto de nodos trabajadores (nodos) que ejecutan las aplicaciones en contenedores.

Los nodos maestros y trabajadores, se comunican entre sí para coordinar la implementación, el escalado y la administración de las aplicaciones. Kubernetes proporciona una API para interactuar con el clúster, lo que permite a los usuarios implementar, escalar y administrar sus aplicaciones de forma declarativa.

El **máster** o nodo maestro es el corazón de Kubernetes, el que controla y programa todas las actividades en el clúster. Es el responsable de mantener el estado general del clúster y coordinar las actividades de los nodos trabajadores.

De acuerdo a [Saito *et al.*, 2017] la arquitectura de Kubernetes incluye los siguientes componentes (ver Figura 2.13):

- **API Server** (`kube-apiserver`): Es el componente del plano de control principal que expone la API de Kubernetes a los clientes lo que permite, por ejemplo, realizar una petición GET del estado de un recurso, o una de tipo POST para crear uno nuevo. El servidor API se comunica con otros componentes del clúster, como `etcd` (lee y actualiza `etcd`) y el `kube-scheduler` para asegurarse que el estado deseado y el estado actual sean idénticos.
- **etcd**: Es un almacén de clave-valor distribuido que almacena los datos de configuración del clúster, tal como el estado actual de los recursos o el estado deseado de los recursos especificados por el usuario. Es el responsable de almacenar y replicar datos.
- **Controller Manager** (`kube-controller-manager`): Es el responsable de mantener el estado deseado del clúster, monitoreando y ajustando los recursos según sea necesario. Es un *daemon* que incorpora los bucles de control principales. Por ejemplo, si un *Pod*[27] es eliminado, el `kube-controller-manager` creará uno nuevo para reemplazarlo.
- **Scheduler** (`kube-scheduler`): Es el componente responsable de programar los *Pods* en los nodos (físicos o máquinas virtuales) del clúster, de acuerdo a la capacidad o al equilibrio de la utilización de recursos en el nodo. Se asegura de que los *Pods* están siendo ejecutados en los nodos correctos, y que los recursos disponibles en los nodos están siendo utilizados eficientemente.

[27] Los ***Pods*** son las unidades de ejecución básica de la plataforma Kubernetes, y encapsulan uno o varios contenedores. Este concepto es desarrollado en la Sección 2.4.1.2.

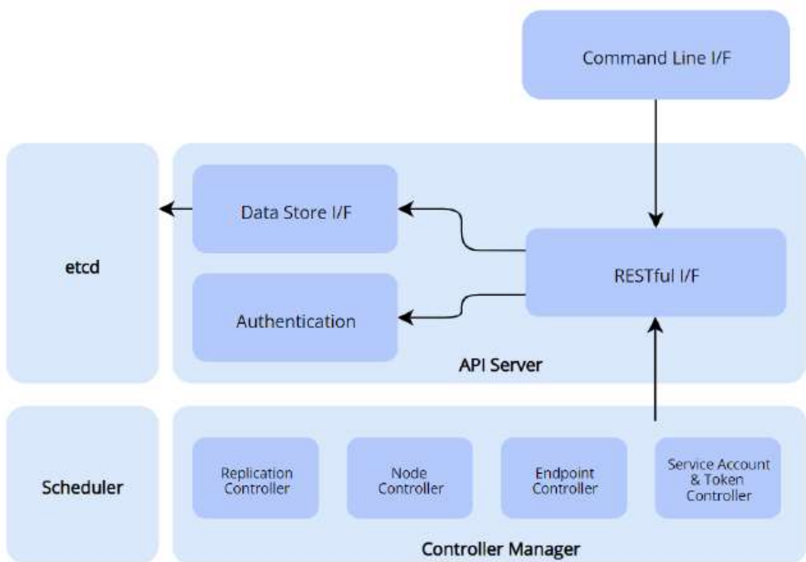


Figura 2.13: *Componentes del nodo máster.*
Nota: Adaptado de [Saito *et al.*, 2017].

Los **nodos**, también conocidos como *minions*, son los trabajadores que ejecutan los *pods* y contenedores.

- **kubelet**: Es un agente que se ejecuta en cada nodo del clúster, responsable de comunicarse con el `kube-apiserver` (ver Figura 2.14) con el fin de asegurar el estado deseado de los *Pods*. Además, se comunica con el container runtime para iniciar, detener y administrar los contenedores que se ejecutan en el nodo.
- **kube-proxy**: Red proxy que se ejecuta en cada nodo, responsable de mantener las reglas de red en el host, es decir, las conexiones de red entre los *Pods* y los servicios (balanceador de carga del *Pod*). Una red proxy funciona como intermediario entre clientes y servidores en Internet. Su propósito es proporcionar una capa de abstracción entre el cliente y el servidor, que puede ofrecer varios beneficios, como mayor seguridad, privacidad y rendimiento.
- **Container Runtime** (ej. `docker`): Responsable de iniciar y detener los contenedores en el host.

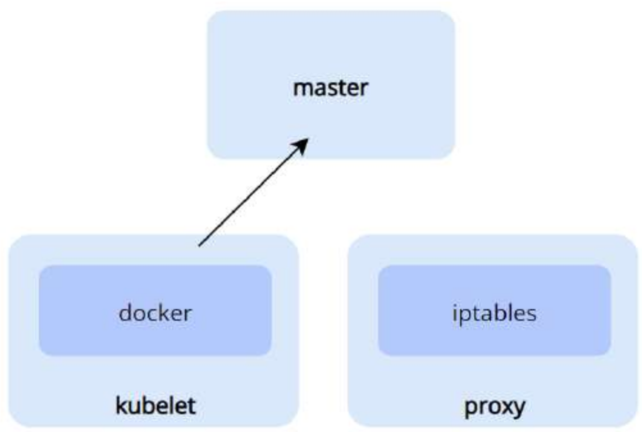


Figura 2.14: *Componentes de los nodos trabajadores.*
Nota: Adaptado de [Saito *et al.*, 2017].

Todos estos componentes trabajan en conjunto para proveer una plataforma que permite el despliegue y la administración de las aplicaciones contenerizadas, de una manera escalable y logrando alta disponibilidad.

Según se observa en la Figura 2.15, el desarrollador puede interactuar con el clúster utilizando la Interfaz de Línea de Comando `kubectl`, que permite enviar solicitudes al ***API server*** de Kubernetes, quien responde publicando y/o obteniendo información del objeto **`etcd`**, según sea requerido. El ***scheduler*** determina qué nodo debe asignarse para realizar las tareas, mientras que, el ***Controller Manager***, supervisa aquellas que se encuentran en ejecución, respondiendo si se produce algún estado no deseado. Así mismo, el ***API server*** obtiene los logs de los *Pods* por medio de **`kubelet`**, además de ser el centro entre otros componentes de máster.

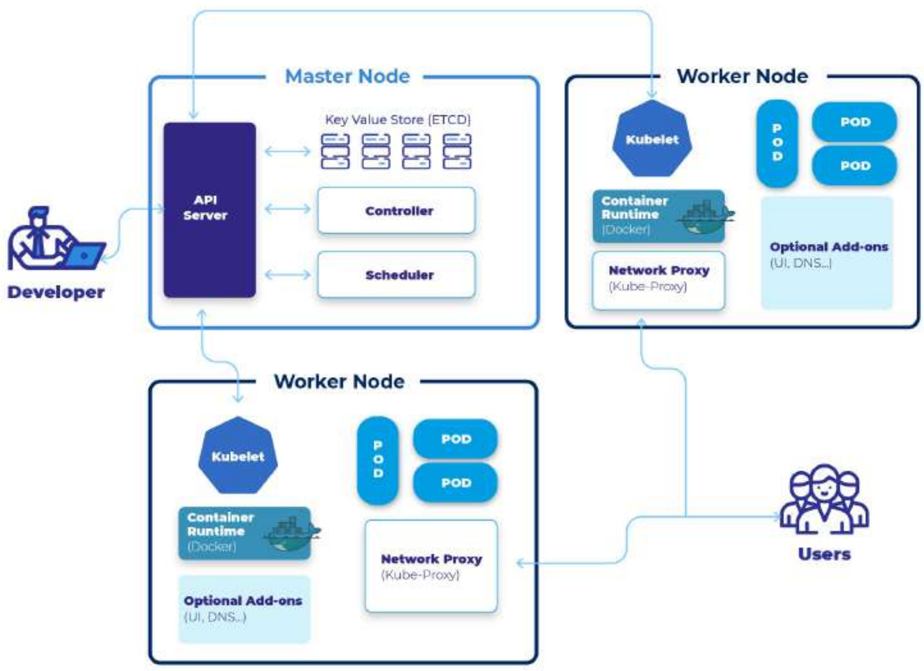


Figura 2.15: *Diagrama sobre la arquitectura de Kubernetes.*
Nota: Adaptado de [ClickIT, 2022].

2.4.1.2. Objetos de Kubernetes

Los objetos de Kubernetes son las entradas en el clúster que representan el estado deseado del mismo y que se almacenan en `etcd`. Cuando se crean, se envía una la solicitud al servidor API mediante `kubectl` o RESTful API. El *API server* almacena el estado en `etcd` e interactúa con otros componentes maestros para garantizar que el objeto exista.

Los objetos principales de Kubernetes que son necesarios para el entendimiento de la propuesta desarrollada en esta tesis son los siguientes [Kubernetes, 2023]:

- **Pods**: Es la unidad más pequeña y simple de Kubernetes que, en la mayoría de los casos, representa un solo proceso o contenedor. Sin embargo, puede contener uno o más contenedores estrechamente relacionados que comparten el mismo espacio de nombres de red y pueden acceder a los mismos volúmenes de almacenamiento compartido.

  Los contenedores dentro de un *Pod* se programan juntos en el mismo nodo y se garantiza que se ejecutarán en la misma máquina física o virtual. Cuando un *Pod* muere, no se recupera por sí

solo, Kubernetes utiliza controladores para crear y administrar el estado deseado declarado por el desarrollador.

- **Replication Controllers**: Objeto responsable de mantener el número deseado de réplicas de un *Pod*.
- **Deployments**: Es un objeto que administra los *Replication Controllers* y los *Pods*, proporcionando actualizaciones declarativas y *rollbacks*.
- **Services**: Los servicios son una capa de abstracción para enrutar el tráfico a un conjunto lógico de *Pods*, y establecer la política que permite el acceso a estos. Actualmente, solo se admite comunicación a través de los protocolos TCP y UDP.

  Hay cuatro tipos de servicios:

  - **ClusterIP**: Es el tipo predeterminado. Expone el servicio en una IP interna del clúster, es decir, solo es alcanzado dentro del clúster, a través de dicha dirección IP, de las variables de entorno o del DNS.
  - **NodePort**: Se asignará un puerto dentro de cierto rango en cada nodo. Cualquier tráfico que llegue a los nodos en ese puerto, se enviará al puerto de servicio.
  - **LoadBalancer**: Kubernetes expone el servicio de forma externa mediante el balanceador de carga de un proveedor de la nube.
  - **ExternalName**: Permite crear un CNAME[28] para endpoints externos en el clúster.
- **Ingress**: Es el objeto de configuración que permite el acceso externo a los servicios dentro de un clúster de Kubernetes. Define reglas para enrutar el tráfico HTTP y HTTPS entrante a diferentes servicios en función de la ruta de URL solicitada o el nombre de host. La Figura 2.16 ilustra simplificadamente el envío de tráfico hacia un servicio a través de un *Ingress*.

Figura 2.16: *Ejemplo de un Ingress enviando tráfico hacia un servicio.*
Nota: Extraído de [Yuanyi, 2021].

- **Namespaces**: Proporciona una manera de dividir los recursos del clúster entre varios usuarios o proyectos, proporcionando un nivel de aislamiento entre los diferentes recursos. Cada espacio de nombres tiene su propio conjunto de objetos (*Pods*, servicios, etc.) y un nombre único.
- **Kustomization**: Recurso especializado utilizado en Kubernetes para definir y gestionar la personalización de la configuración de los recursos en un clúster.
- **Labels y Selectors**: Se utilizan con el objetivo de organizar e identificar los recursos dentro del clúster.

[28] Un nombre canónico o registro CNAME es un tipo de registro DNS que asigna un alias a un nombre de dominio auténtico o canónico.

Hay objetos más específicos como los *ConfigMaps*, *Secrets*, *Volumes*, *DaemonSets*, *StatefulSets*, *Jobs*, *CronJobs*, *Horizontal Pod Autoscaler*, *ClusterRoleBinding*, que no se detallan en el desarrollo de este trabajo, por quedar afuera del objetivo general del mismo.

2.4.1.3. Manifiestos de Kubernetes

En Kubernetes, un manifiesto es un archivo escrito en lenguaje YAML, que define el estado deseado de uno o más recursos en un clúster, detallando la configuración de estos. Kubernetes permite orquestar aplicaciones y servicios de forma sencilla y reproducible, ya que los archivos YAML son legibles y versionables, lo que facilita su gestión y seguimiento a lo largo del ciclo de vida de una aplicación.

Empleando el comando `kubectl`, se le solicita a Kubernetes la creación, actualización o eliminación de los recursos especificados.

Los comandos más utilizados son dos:

- `kubectl apply -f <nombre_archivo>`: Para crear o actualizar el recurso detallado en el archivo.
- `kubectl delete -f <nombre_archivo>`: Para eliminar el recurso.
- `kubectl get`: Para obtener información resumida del estado de un tipo de recurso, por ejemplo de los Pods, `kubectl get pods`. También permite obtener el estado de todos los recursos de un namespace especificado ejecutnado la línea `kubectl get all -n <nombre_namespace>`.
- `kubectl describe pod <nombre_del_pod>`: Para obtener información más detallada de un recurso en específico.

Un ejemplo de manifiesto es el siguiente:

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: deployment-ejemplo
spec:
  replicas: 2
  selector:
    matchLabels:
      deploy: aplicacion
  template:
    metadata:
      labels:
        deploy: aplicacion
    spec:
      containers:
        - name: aplicacion
          image: aplicacion:1.7.9
```

En donde se declara el nombre del `Deployment` como `deployment-ejemplo`, el cual contiene dos réplicas del Pod llamado `aplicacion`, que se conforma por la imagen `aplicacion:1.7.9`.

2.5. Computación en la nube

Computación en la nube, o ***cloud computing***, es un modelo para brindar servicios de tecnología de la información a través de Internet (“la nube”) utilizando aplicaciones basadas en la web, en lugar de una conexión directa a un servidor. Es decir, permite el acceso remoto a recursos como servidores, almacenamiento, bases de datos, redes, software, análisis e inteligencia, que se puede ampliar o reducir dinámicamente para satisfacer las demandas cambiantes y permitir un escalamiento más fácil que el habitual [Microsoft Azure, 2023].

La utilización de servicios en la nube actualmente está en manos de grandes compañías como Google, Microsoft, Amazon y, generalmente, se paga el tiempo de consumo transcurrido al utilizar los servicios de la nube. Este modelo de negocio obliga a las empresas a mantener bajos costos operativos, ejecutando la infraestructura más eficientemente y escalando de acuerdo a la demanda del negocio.

La **computación en la nube** distingue tres tipos de nubes, que proveen diversos servicios. Entre los principales tipos se encuentran:

- **Nube pública**: Son propiedad y están operadas por una empresa externa, disponibilizándola al público a través de Internet; solo se tiene acceso a ellas utilizando un navegador web. Todo el hardware, software e infraestructura de soporte son administrados por el proveedor del servicio. Ejemplos de proveedores de este tipo de nubes son Amazon Web Services (AWS), Microsoft Azure y Google Cloud Platform (GCP).
- **Nube privada**: Es aquella en que los servicios e infraestructura se mantienen en red privada. En este caso, el servicio es utilizado exclusivamente por una sola organización y, generalmente, se encuentra alojado en la propia infraestructura. Algunas empresas, pagan a proveedores externos para que alojen su nube privada.
- **Nube híbrida**: Combinación de nubes públicas y privadas, unidas por tecnología que permite compartir datos y aplicaciones entre ellas. De esta manera, permite a la organización aprovechar los beneficios de ambos tipos de nubes, brindando mayor flexibilidad, más opciones de implementación y ayudando a optimizar su infraestructura y seguridad.

En tanto que, en consideración de los servicios que ofrece la nube, se distinguen los siguientes tipos (ver Figura 2.17):

- **Software como Servicio** (*Software as a Service*, SaaS): Son aquellas aplicaciones que se encuentran alojadas por un proveedor externo y están disponible para los clientes a través de Internet.
- **Plataforma como Servicio** (*Platform as a Service*, PaaS): Plataforma que facilita el desarrollo, ejecución y administración de aplicaciones a través de Internet.
- **Infraestructura como Servicio** (*Infrastructure as a Service*, IaaS): Proporciona recursos informáticos virtualizados a través de Internet. Permite que las organizaciones alquilen recursos, como por ejemplo servidores, almacenamiento o redes, según sea necesario, en vez de tener que comprarlos y mantener su propia infraestructura física.

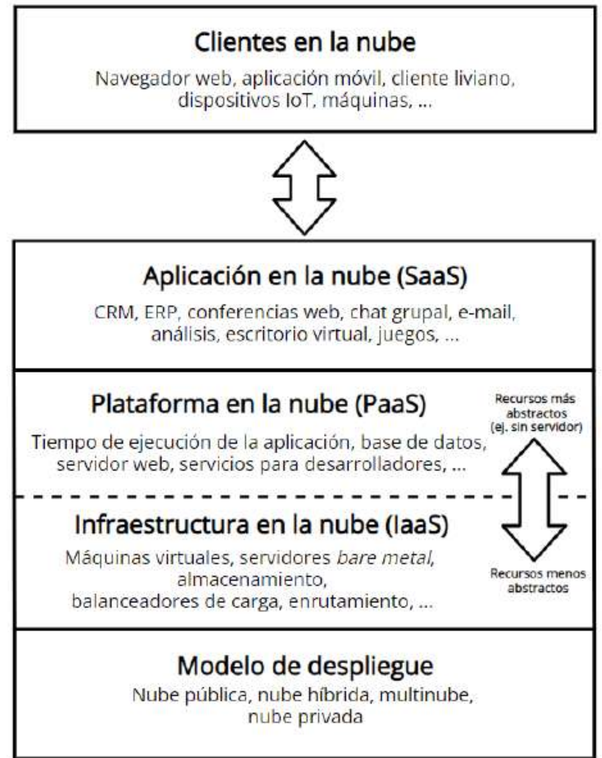


Figura 2.17: *Servicios de la nube.*
Nota: Adaptado de [BSmO21, 2022].

2.5.1. Arquitectura de computación en la nube

El término ***Cloud Native*** hace referencia a la arquitectura y prácticas de desarrollo de aplicaciones diseñadas para ser ejecutadas en la nube. Están elaboradas para ser altamente escalables, tolerantes a fallas y capaces de tomar ventaja de recursos elásticos suministrados por la infraestructura del *cloud* (nube pública, privada o híbrida).

Generalmente, las aplicaciones nativas de la nube se desarrollan como microservicios, que luego son empaquetados en contenedores. Las aplicaciones monolíticas que sean de interés migrar a la nube, podrían empezar su transición si se las contenerizara. Pero aun así, el hecho de ser aplicaciones contenerizadas no las convierte automáticamente en *Cloud Native*, sino que deben cumplir con ciertas características, entre las cuales se encuentran:

- **Automatización**: Las aplicaciones deben ser creadas teniendo en cuenta la automatización, con procesos de integración continua y despliegue continuo (CI/CD).
- **Escalabilidad**: Están diseñadas para escalar fácilmente hacia arriba, como hacia abajo, según lo determine la demanda.
- **Portabilidad y flexibilidad**: Debido a que están desacopladas de recursos físicos, las aplicaciones en contenedores se pueden mover fácilmente de un lugar a otro.
- **Observabilidad**: Emiten datos de telemetría que se pueden recopilar y analizar para comprender su comportamiento y rendimiento.
- **Seguridad**: Son diseñadas teniendo en cuenta la seguridad, centrándose en la autenticación, la autorización y el cifrado.

- **Tolerante a fallas**: Están diseñadas para ser tolerantes a fallas y autorreparables, capaces de recuperarse de fallas sin intervención humana.

2.6. Angular

Angular es un *framework* front-end de JavaScript[29], para el desarrollo de aplicaciones web, creado y mantenido por Google. Es comúnmente utilizado para construir aplicaciones web complejas a gran escala.

> Un *framework* de desarrollo, o marco de software, es una colección de bibliotecas y módulos que proporcionan un conjunto de funcionalidades que se pueden utilizar para crear aplicaciones de software. Sirve como modelo para construir la aplicación y proporciona una estructura para el código.

2.6.1. Arquitectura de Angular

Angular emplea una arquitectura basada en componentes para la construcción de las aplicaciones, haciendo uso de un enfoque declarativo para crear la interfaz de usuario. Incluye bibliotecas que cubren una amplia variedad de funciones, como el enrutamiento, gestión de formularios, comunicación cliente-servidor, entre otras, Posee además, un conjunto de herramientas que contribuyen al desarrollo, compilación, testeo y actualización del código.

Según a lo publicado en la documentación oficial [Angular, 2023], los componentes claves que trabajan en conjunto (ver Figura 2.18) para proporcionar una base de código escalable y mantenible son:

- **Componentes** (*Component*): Cada aplicación de Angular tiene al menos un componente. Estos son los responsables de definir la interfaz de usuario de la aplicación y encapsular su comportamiento. Cada componente es autónomo y reutilizable, lo que permite a los desarrolladores crear aplicaciones complejas mediante la combinación de piezas más pequeñas y manejables.

  Los componentes definen **vistas**, que son conjuntos de elementos de pantalla entre los que Angular puede elegir y modificar de acuerdo con la lógica y los datos de su programa.
- **Directivas** (*Directive*): Proporcionan una forma de ampliar la funcionalidad de HTML mediante la definición de etiquetas y atributos HTML personalizados que se pueden usar dentro de las plantillas.
- **Plantillas** (*Templates*): Definen el diseño y la estructura de la interfaz de usuario para cada componente. Una plantilla combina HTML ordinario con directivas y *binding markup* que permiten a Angular modificar el HTML antes de mostrarlo. Hay dos tipos de enlace de datos (*data binding*):
  - **Enlace de eventos** (*Event binding*): Permite que la aplicación responda a la entrada del usuario en el entorno de destino actualizando los datos de la misma.
  - **Enlace de propiedad** (*Property binding*): Permite interpolar valores que se calculan a partir de los datos de la aplicación en el HTML.

[29] https://www.javascript.com/

- **Servicios** (*Services*): Se utilizan para proporcionar una funcionalidad que se puede compartir entre varios componentes o en toda la aplicación. Son una forma de encapsular datos y funciones que no son específicos de ningún componente, lo que permite una mayor capacidad de reutilización y mantenimiento.
- **Inyección de dependencia** (*Dependency injection, DI*): Es un mecanismo que se encarga de resolver y proporcionar las dependencias solicitadas por los componentes y servicios. Facilita la gestión de dependencias entre los componentes de una aplicación, mejorando la modularidad, la testabilidad y la flexibilidad del código.

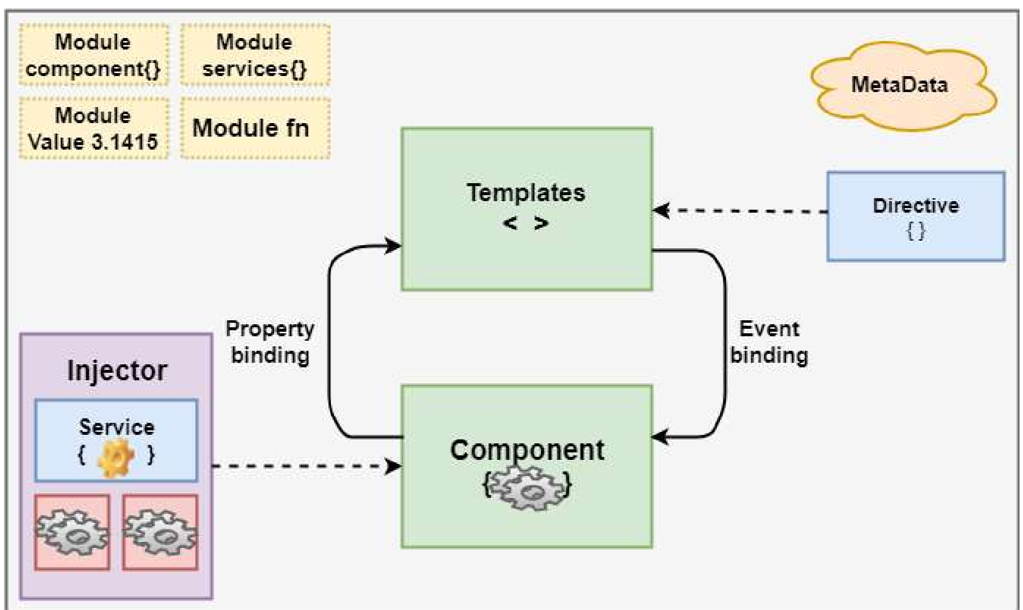


Figura 2.18: *Arquitectura del framework Angular.*
Nota: Extraído de [Angular, 2023]

Un **componente** y una **plantilla** definen una vista Angular. Las **directivas** y el ***binding markup***, en la plantilla de un componente, modifican las vistas en función de los datos y la lógica del programa, mientras que el **inyector de dependencia** proporciona **servicios** a un componente, como el servicio de enrutador que le permite definir la navegación entre las vistas.

2.6.2. Angular CLI

Es una herramienta de línea de comandos que facilita a los desarrolladores la creación, compilación, pruebas (testeos) e implementación de las aplicaciones construidas con Angular. Permite acelerar el proceso de desarrollo pues ahorra tiempo y reduce los errores mediante la automatización de tareas repetitivas.

Los comandos que se utilizaron en el proyecto son:

- `ng build`: Compila una aplicación Angular y almacena la salida en un directorio especificado [Angular, 2023].
- `ng test`: Ejecuta las pruebas unitarias de un proyecto utilizando Karma[30]. Este carga los archi-

[30]https://karma-runner.github.io/latest/index.html

vos de prueba y sus dependencias en el navegador, ejecuta las pruebas e informa los resultados a la terminal o interfaz de línea de comandos. Karma utiliza un entorno de navegador para garantizar que las pruebas se comporten de manera consistente en diferentes navegadores [Angular, 2023].

2.7. Calidad de código fuente

La calidad de código es una medida que indica qué tan alto o bajo es el valor de un conjunto específico de código fuente. Es una parte del proceso de desarrollo de software que se emplea con el objetivo de identificar las vulnerabilidades y errores en una etapa temprana, para con ello contribuir a la mejora de la calidad general del software, facilitar su mantenimiento y satisfacer las necesidades del cliente.

Existen dos maneras de realizar el análisis de la calidad de código fuente[31]. Por un lado; el **análisis estático**, en el cual se examina el código para identificar problemas dentro de la lógica y las técnicas. Por otra parte; el **análisis dinámico**, el cual implica la ejecución del código y la posterior examinación de los resultados.

En el día a día de un equipo de desarrollo, una tarea elemental es revisar el código que se está realizando, al igual que ejecutarlo para validar que funciona correctamente. De esta manera, sin establecer un proceso formal, ya se está haciendo un análisis del código fuente estático y dinámico respectivamente, aportando significativamente a la calidad del producto final.

No se debe optar por una sóla forma de análisis, se debe pensar en ellos como un conjunto complementario y simbiótico. Empleando ambas maneras, se permite abarcar diferentes aspectos del código fuente, logrando la detección temprana de errores e identificando las posibles mejoras que se pueden realizar para añadir valor al producto. Incluso, la información que arroje un análisis estático puede no ser suficiente sin tener un análisis dinámico, y viceversa. Por ejemplo, podemos tener una porción de código que cumple con los estándares de codificación, es segura y está bien documentada, lo cual daría un excelente resultado en un análisis estático, pero si no lo analizamos dinámicamente, no podemos determinar que ese código es funcional. Por lo tanto, la calidad del código no se podría determinar con uno solo de ellos.

Analizar la calidad del código fuente de una aplicación, es una práctica que desempeña un rol muy valioso en *DevOps*, ya que detectando errores y problemas al inicio del proceso de desarrollo, se logra mejorar la velocidad, calidad y confiabilidad de la entrega de software, garantizando código de alta calidad y bien diseñado.

2.7.1. Análisis estático del código - Inspección continua

El análisis de la calidad de código fuente estático, si se realiza de manera eficiente, puede ser sorprendentemente efectivo. Este proceso consta de inspeccionar el código para identificar posibles problemas, como errores, vulnerabilidades de seguridad y *code smells* (indicadores de diseño deficiente). Asimismo, comprende métricas que evalúan el cumplimiento de los estándares de codificación, su capacidad de mantenimiento, capacidad de prueba, claridad, reutilización, seguridad y más [Duvall *et al.*, 2007].

[31] Otra técnica referenciada en la literatura, es la inspección de características y atributos de calidad, como la propuesta en [Olsina, 1999].

Cuando estos análisis se realizan manualmente, se asigna a una persona (por lo general a un testeador) una porción de código, habitualmente pequeña, que debe inspeccionar línea a línea. Sin embargo, la desventaja que conlleva este proceso es que encontrar los defectos puede ser difícil si no se cuenta con la experiencia suficiente, y requiere de una gran cantidad de tiempo disponible. En otras palabras, la revisión de código manual es una operación que se torna extensa, repetitiva y agotadora.

Por otro lado, el proceso de inspección automática es llevado a cabo por una herramienta específicamente desarrollada para cumplir con esta tarea. En esta, se pueden analizar miles de líneas de código en escasos segundos y los defectos son encontrados rápidamente, reduciendo el tiempo y esfuerzo requerido.

2.7.1.1. SonarQube

SonarQube[32] es una plataforma de código abierto que proporciona el **análisis estático** automático de código, está desarrollada en Java, y analiza más de 30 lenguajes de programación diferentes. Permite la integración con pipelines de CI y plataformas *DevOps*, para garantizar que su código cumpla con los estándares de calidad, ayudando a realizar inspecciones continuas de sus proyectos. Además, provee la capacidad de almacenar el historial de métricas y realizar un seguimiento de la evolución del código a lo largo del tiempo a través de gráficos.

Las métricas[33] que SonarQube utiliza para evaluar la calidad del código, se encuentran agrupadas en nueve categorías: complejidad, duplicaciones, problemas, mantenibilidad, fiabilidad, seguridad, tamaño y pruebas. Dentro de ellas, se especifican métricas populares como: densidad de comentarios, cantidad de líneas de código, cantidad de líneas de código duplicadas, bugs, deuda técnica, complejidad, cantidad de errores, vulnerabilidad, entre otras.

Para determinar el valor de calidad del código, SonarQube implementa el ***quality gate***, un conjunto de condiciones que deben ser alcanzadas para que el análisis se considere exitoso [SonarSource, 2023]. Estas condiciones están basadas en un conjunto de métricas y umbrales predefinidos, como el número de *bugs*, *code smells* y cantidad de vulnerabilidades de seguridad encontradas en el código. Estas pueden ser modificadas, incluso se pueden crear nuevas o eliminar las existentes, ajustándose a las necesidades y permitiendo una personalización adecuada al proyecto.

Cabe destacar que las condiciones predeterminadas en el *quality gate*, las establece un equipo de expertos especializados en desarrollo de software y análisis de código, con base a las mejores prácticas y pautas de la industria para el desarrollo de software. Estos expertos consideran cuidadosamente una amplia gama de factores que pueden afectar la calidad del código, incluida la complejidad, la duplicación, la cobertura, las vulnerabilidades de seguridad y más. Es importante tener en cuenta que, si bien las condiciones predefinidas están diseñadas para ser un punto de partida útil, es posible que no sean adecuadas para todos los proyectos u organizaciones.

Una vez ejecutado el análisis, SonarQube genera un informe que proporciona información detallada sobre los problemas encontrados en el código y compara estos resultados con las condiciones definidas en el *quality gate*. Si estas condiciones se cumplen, entonces el código es considerado de alta calidad. De lo contrario, si el *quality gate* falla, significa que no se logró cumplir con la calidad esperada. En este caso, los desarrolladores pueden solucionar los problemas encontrados y volver a ejecutar el análisis para asegurarse de que hayan sido resueltos correctamente.

Hacer uso de esta herramienta ayudará a identificar y solucionar errores antes de que se conviertan

[32]https://docs.sonarqube.org/latest/
[33]https://docs.sonarqube.org/latest/user-guide/metric-definitions/

en un gran problema, ayudando a evaluar la calidad del código y estimando el esfuerzo restante necesario para alcanzar el nivel deseado. Esto repercute significativamente en la mejora de la calidad total del código fuente y en la reducción de las posibilidades de implementar código dañado o no probado. Además, automatizando el análisis, se hará más fácil identificar y rastrear los problemas, permitiendo ahorrar tiempo y esfuerzo a los desarrolladores, obteniendo un código más fiable y legible que aumentará la productividad y la calidad.

2.7.2. Análisis dinámico del código - Testing continuo

Como se mencionó anteriormente, el **análisis dinámico** es un método que requiere de la ejecución de la porción de código fuente para evaluar su comportamiento. Su objetivo es identificar problemas que pueden no ser visibles durante el análisis estático, como excepciones en tiempo de ejecución, integración de la aplicación con servidores, análisis de uso de memoria, rendimiento y pruebas funcionales.

Los **tests unitarios**, o **pruebas unitarias**, son una forma de análisis dinámico, ya que implican la ejecución del código y la observación de su comportamiento en tiempo de ejecución, pero requieren menos recursos que otras formas de análisis, como lo sucede en el análisis de uso de memoria. Las pruebas deben ser escritas por los mismos desarrolladores del código para garantizar que funcione como se espera.

Durante este testeo, el código se ejecuta con un conjunto de entradas específicas, que cubren diferentes escenarios, y se determina la salida esperada. Si el resultado arrojado por la ejecución no coincide con el valor esperado, entonces se considera que la prueba falló, indicando que hay un problema con el código. Las pruebas unitarias también pueden incluir comprobaciones de otros tipos, como por ejemplo, si el código no provoca pérdidas de memoria.

Las pruebas unitarias cumplen un papel importante en el proceso de desarrollo de software, ya que, al tener la capacidad de ser automatizadas, permite a los desarrolladores verificar rápida y fácilmente el comportamiento de su código a medida que se realizan cambios, aumentando así la calidad de la entrega de software y disminuyendo las probabilidades de entregar una aplicación que no cumple con las funcionalidades requeridas, ni con las necesidades del cliente.

2.8. Conclusión

En resumen, este capítulo ha proporcionado la información necesaria sobre los fundamentos teóricos importantes para la comprensión del trabajo. Se abarcó el movimiento *DevOps*, explicando sus principales prácticas y beneficios de su aplicación, al igual que se mencionó el significado del enfoque *Cloud Native*, desarrollando previamente conceptos como los de contenedores y orquestador de contenedores, fundamentales para la comprensión de este enfoque. También se informó sobre el *framework* Angular y sobre las prácticas de análisis de la calidad de código fuente.

La sinergia entre *DevOps* y *Cloud Native* es transformadora. Ambos enfoques impulsan un cambio significativo en el desarrollo de software, permitiendo trabajar colaborativamente con agilidad y eficiencia, creando productos que se adaptan fácilmente a los cambios y capaces de brindar buenas experiencias al usuario.

En los siguientes capítulos se abordarán los diferentes conceptos presentados desde un modo práctico, para llevar a cabo el desarrollo del caso de uso que respalda la solución propuesta.

3
Estado del Arte

"Es esencial tener buenas herramientas,
pero también es esencial que las herramientas
se usen de la manera correcta."

— Wallace D. Wattles

Este capítulo proporciona un breve análisis que compara las herramientas y servicios tecnológicos más populares del mercado, a fin de justificar la selección y elección de los productos utilizados en el desarrollo del presente trabajo.

3.1. Introducción

El presente trabajo tiene como finalidad proponer un enfoque *DevOps* y *Cloud Native* que favorezca la promoción de la utilización de nuevas tecnologías y por consiguiente, la actualización del Área de Sistemas de la Facultad de Ingeniería, a fines de migrar hacia tendencias actuales de desarrollo y producción de software.

Con el propósito de demostrar los beneficios que supone la automatización de los procesos de compilación, testeo, análisis de la calidad del código fuente y despliegue, se desarrolló una *pipeline* diseñada como prueba de concepto para ejemplificar la propuesta planteada.

Para la implementación de la *pipeline* se debió considerar lo siguiente:

- un **servidor de integración** continua que ejecute la *pipeline* y automatice los procesos de desarrollo,
- una herramienta que permita el **análisis de la calidad del código fuente**, que se integre sin inconvenientes con el servidor de integración,
- una plataforma de **contenedores** que permita contenerizar la aplicación mostrada y pueda ser desplegada, sin inconvenientes de interoperabilidad, en cualquier entorno,
- un **proveedor de servicio de nube** que brinde el servicio suficiente para alojar la aplicación y la *pipeline* desarrollada.

En esta sección, se presenta el análisis realizado de las distintas herramientas destinadas a cubrir cada uno de los items anteriores, los criterios aplicados para decidir y seleccionar una y la justificación de tal selección.

3.2. Servidores de CI

Al momento de elegir un servidor de CI (concepto desarrollado en la sección 2.2.2.1) para emplear en el desarrollo de software, la variedad de opciones para elegir es amplia. La selección puede ser una tarea crítica para cualquier equipo de trabajo, por esto, se deben contemplar una serie de factores a la hora de tomar tal decisión:

- **Compatibilidad**: El servidor debe ser compatible con el sistema operativo sobre el cuál se requiere su instalación, además de soportar el lenguaje de programación que se utiliza en el proyecto de desarrollo, al igual que el *framework*.
- **Funcionalidades**: Las características que ofrece el servidor deben satisfacer las necesidades del equipo.
- **Integración**: El servidor debe permitir el uso en conjunto de las diferentes herramientas seleccionadas para la automatización.
- **Escalabilidad**: Se debe asegurar que la herramienta se ajuste al tamaño del equipo y a la complejidad de los proyectos.
- **Personalización**: Se debe considerar una herramienta que pueda adaptarse para cumplir con las necesidades específicas del equipo y de los proyectos.
- **Seguridad**: El servidor a escoger debe tener características como, el control de acceso basado en roles, encriptación, y soporte para protocolos seguros.

- **Costos**: No debe excederse el presupuesto del proyecto, abarcando las licencias, mantenimiento y soporte.
- **Comunidad y soporte**: La documentación tiene que estar disponible, de manera clara, y que exista una comunidad con usuarios activos, que puedan ayudar a solucionar problemas y contribuyan a su desarrollo, proporcionando recursos y complementos adicionales.
- **Interfaz de usuario**: Es ideal que sea una plataforma intuitiva, que permita al equipo adaptarse rápidamente a su uso y que su configuración no requiera de un gran esfuerzo.

3.2.1. Selección del servidor

Para este trabajo se realizó una investigación preliminar con el fin de recopilar información sobre los servidores de CI más utilizados a la fecha. Este análisis preliminar posibilitó reducir el grupo de herramientas posibles a considerar y, posteriormente, realizar una comparación entre los tres más utilizados.

Luego de visitar varias páginas web que ofrecen un ranking de los mejores servidores de CI en el mercado (ver Anexo A1), se consideraron los servidores que se encontraban en los primeros cinco lugares de la lista. A partir de estos datos, se redujo la cantidad a los tres servidores de CI con mayor recurrencia en los diferentes listados. Con Jenkins en primer lugar, siendo mencionado en todos los rankings de las páginas visitadas resultando con un total de siete apariciones, seguido por Bamboo[1] y CircleCI[2], ambos con cinco apariciones. En el Anexo A1 se muestran los servidores mencionados junto con el número de veces que son referenciados.

Con base a esta selección, se realizó un cuadro comparativo que detalla las características de cada herramienta en los diferentes campos a evaluar. Finalmente, se optó por utilizar **Jenkins**.

[1] https://www.atlassian.com/es/software/bamboo
[2] https://circleci.com/

Tabla 3.1: *Comparación de servidores de integración continua.*

	Jenkins	**Bamboo**	**CircleCI**
Compatibilidad	Windows, macOs, Linux, Docker, Kubernetes. Altamente compatible con varios lenguajes de programación, herramientas de compilación y SCM.	Windows, macOs, Linux, Docker, Kubernetes. Es compatible con varios lenguajes de programación, y también con diferentes herramientas de compilación, además de SCM como Git, Mercurial, Subversion y Perforce.	Windows, Linux, macOs (solo para la nube), Docker, Kubernetes, ARM machines. Es compatible con muchos lenguajes populares como Java, Python, Ruby, Go, Node.js y PHP. También es compatible con diferentes herramientas de compilación y se integra con varios SCM.
Funcionalidades	Integración y despliegue continuos (CI/CD).	Integración y despliegue continuos (CI/CD).	Integración y despliegue continuos (CI/CD).
Integración	Ilimitado, más de 1800 *plugins*.	Jira, Bitbucket, Conflence, AWS CodeDeploy, se integra con otras herramientas de Atlassian, +300 *plugins*.	GitHub, Bitbucket, Jira, Slack, AWS, Google, Microsoft Azure, +150 integraciones.
Personalización	Gran ecosistema de *plugins*, scripts personalizados, Configuración como Código.	Varios *plugins*, permite crear planes y ramas.	Permite personalizar imágenes de Docker, y admite el concepto de Configuración como Código.
Escalabilidad	Altamente escalable, se puede configurar para admitir proyectos a gran escala con requisitos de construcción complejos.	Altamente escalable, se puede configurar para admitir proyectos a gran escala con requisitos de construcción complejos.	Altamente escalable, se puede configurar para admitir proyectos a gran escala con requisitos de construcción complejos.
Seguridad	Múltiples métodos de autenticación de usuario, autorización basada en roles y permisos, seguimiento de la actividad de los usuarios, comunicación segura con cifrado TLS/SSL, además de los *plugins* de seguridad que ofrece la herramienta.	Múltiples métodos de autenticación de usuario, autorización basada en roles y permisos, comunicación segura con cifrado TLS/SSL, reportes de seguridad y *plugins*.	Control de acceso, máquina virtual aislada por cada ejecución, encriptación de datos en tránsito y descanso, administración de datos sensibles (contraseñas, secrets).
Costos[3]	Gratis	Desde 1200 USD anuales, por un solo agente.	Gratis pero con funciones limitadas, o desde 15 USD mensuales.

[3]Los costos son a la fecha de Mayo 2023.

Comunidad y Soporte	Gran comunidad activa que contribuye al desarrollo y mantenimiento de la plataforma. Documentación, eventos, oportunidades de contribución, entrenamientos y certificaciones.	Soporte a través de foros, documentación, canales de chat, eventos, oportunidades de contribución, entrenamientos y certificaciones.	Documentación extensa, se organizan eventos y encuentros, también proveen entrenamientos y certificaciones. Ofrece varios niveles de soporte para los usuarios.
Interfaz de Usuario	Fácil de configurar, pero su interfaz gráfica es deficiente.	No es tan fácil de configurar, pero su interfaz de usuario es amigable.	Buena interfaz de usuario, con constantes actualizaciones.

3.2.2. Justificación de la selección

De acuerdo a las características detalladas en la Tabla 3.1, se puede notar fácilmente que **Jenkins** es el servidor con mayor cantidad de *plugins* con respecto a sus contrincantes, lo que vuelve ilimitada su capacidad para integrarse con otras herramientas, permitiendo versatilidad y flexibilidad a la hora de personalizar los proyectos en función de las diferentes necesidades.

Otro punto destacable es que emplearlo no requerirá de costos adicionales, pues su uso es gratuito. Además, al ser de código abierto, posee una comunidad amplia y activa de usuarios y desarrolladores que respaldan a la herramienta, realizando mejoras continuamente. Incluso, es muy sencillo crear un nuevo complemento, si este no existiera.

En cuanto a la escalabilidad, permite añadir tantos usuarios como sean necesarios, aplicando control de acceso basado en roles. Asimismo, brinda la capacidad de incrementar el número de agentes según como sea requerido. Jenkins puede adaptarse tanto a pequeños como grandes proyectos sin mayores dificultades, proporcionando la habilidad de automatizar compilaciones de la aplicación, ejecución de pruebas, y despliegues en diferentes entornos.

Jenkins cuenta con la desventaja de que su interfaz gráfica no es tan amigable con el usuario, ni intuitiva. Sin embargo, esto puede solucionarse empleando un *plugin* llamado Blue Ocean[4] que facilita la visualización del desarrollo de las *pipelines* y su configuración.

En resumen, eligiendo Jenkins como el servidor de CI a utilizar en el presente trabajo, se cumple con todos los factores de selección mencionados anteriormente: funcionalidades, escalabilidad, interfaz de usuario, integración, compatibilidad, seguridad, comunidad y soporte, costos y personalización.

3.3. Análisis de calidad del código fuente

De igual forma que con la elección del servidor de CI, a la hora de seleccionar una herramienta para evaluar la calidad del código fuente (aspecto cubierto en la Sección 2.7), el mercado actual de tecnologías al respecto ofrece una gran y diversa cantidad de opciones para tener en cuenta. Por esta razón, es recomendable considerar algunos puntos importantes que ayuden a definir las necesidades del proyecto y así, poder optar por la opción que mejor se adapte a este.

En este caso, se tuvieron en cuenta algunos de los factores detallados anteriormente (sección 3.2):

[4] https://plugins.jenkins.io/blueocean/

interfaz de usuario, integración, soporte y comunidad, costos, personalización, funcionalidades y compatibilidad.

3.3.1. Selección de la herramienta

Para seleccionar la herramienta que realiza el análisis de la calidad del código fuente, se optó por la misma metodología llevada a cabo en la elección del servidor de CI. Así, se recopiló información de los cinco analizadores más utilizados a la fecha, de acuerdo a los rankings establecidos por diferentes páginas web (ver Anexo A2).

Luego, se contabilizó la repetición de menciones por cada herramienta, resultando primero SonarQube con seis menciones, seguido de Veracode[5] con cuatro y DeepSource[6] con tres. De acuerdo a esta selección, la Tabla 3.2 compara las características de las tres herramientas, que facilitó la elección de una en particular.

Al poner en claro las diferencias entre estas tres herramientas, se decidió utilizar **SonarQube** para el análisis estático del código fuente.

Tabla 3.2: *Comparación de herramientas para el análisis de la calidad de código fuente.*

	SonarQube	**Veracode**	**DeepSource**
Lenguajes de Programación (Compatibilidad)	+29 lenguajes, y frameworks como Rust, Vue.js, React, Angular, y otros.	+27 lenguajes y frameworks como Spring, Hibernate, Struts, ASP.NET, Angular, React, y más.	10 lenguajes y frameworks de Python (Django, Flask, Pyramid, etc), JavaScript (React, Angular), TypeScript (Nest.js, Angular) y Go (Gin, Echo).
Funcionalidades	Análisis de la calidad del código, cobertura de código, análisis de seguridad, deuda técnica.	Análisis de la calidad del código, análisis dinámico, consejos de mitigación, análisis de cumplimiento.	Análisis de la calidad del código, análisis de seguridad, reglas de análisis personalizables.
Integración	Provee varios *plugins* para su integración con Eclipse, Visual Studio, Jenkins, Azure pipelines y otros. Además brinda una interfaz de línea de comando.	Provee APIs, interfaz de línea de comando y *plugins* para integrarlo a las diferentes herramientas como Eclipse, IntelliJ, Visual Studio, Jenkins, y otros.	Provee integración con varios SCM (GitHub, GitLab, Bitbucket y Git), editores de código como Visual Studio Code o Atom, y servidores como Jenkins, Travis CI, CircleCI y GitLab CI/CD.
Personalización	Se pueden crear *plugins* para añadir nuevas características a la herramienta. Además, permite modificar los perfiles de calidad, dashboards y reportes.	Permite personalizar las políticas, configuraciones de escaneo, reportes e integraciones.	Permite crear reglas y chequeos, además de brindar la capacidad de personalizar las notificaciones y reportes.

[5] https://www.veracode.com/
[6] https://deepsource.com/

Costos	Edición *community* (gratuita), y edición enterprise que comienza en 4.500 USD por año.	Precio por escaneo (comienza en 500 USD por aplicación, con descuentos por volumen), basado en suscripción (comienza en 55.000 USD anuales) y el plan de empresa que es personalizado.	Plan gratuito (análisis de 1000 líneas de código por mes para hasta 5 repositorios), y luego los planes comienzan desde 60 USD por año (o seis USD mensuales).
Comunidad y Soporte	Gran comunidad activa de usuarios y desarrolladores. Documentación, foro, rastreador de *bugs*.	Comunidad activa de usuarios y desarrolladores. Foro, centro de soporte, tickets de soporte, rastreador de *bugs*.	Documentación, tutoriales, mejores prácticas, material de referencia, soporte vía email, foro
Interfaz de Usuario	Basada en la web, buena interfaz y altamente personalizable. Otorga comentarios claros.	Basada en la web, amigable con el usuario y también es personalizable.	Simple e intuitiva, con un enfoque en proporcionar comentarios claros a los usuarios. Personalizable.

3.3.2. Justificación de la selección

La decisión fue tomada considerando el presupuesto para este trabajo. Por lo tanto, si bien se puede apreciar que SonarQube y Veracode son muy similares, el último posee un elevado costo para su utilización, mientras que el primero de ellos cuenta con una edición gratuita para la comunidad. Por esta simple razón, es que Veracode fue descartado.

En cuanto a DeepSource, que sí contiene un plan gratuito, en lo referido a los lenguajes de programación, su alcance es más limitado. Además, solo permite la ejecución de 1000 líneas de código por repositorio, lo que restringe su uso.

Por su parte, **SonarQube**, cumple con todos los factores mencionados, por lo tanto, resultó ser la herramienta apropiada a seleccionar para este trabajo académico, abarcando con soltura la compatibilidad, funcionalidad, personalización, integración, soporte, interfaz de usuario y costos.

3.4. Proveedores de servicios de nube

Los proveedores de servicios de nube son empresas que administran nubes públicas o privadas, ofreciendo recursos y servicios de computación en la nube (para más información al respecto, el lector puede consultar la Sección 2.5). De esta manera, muchas organizaciones y empresas se benefician puesto que reducen costos en infraestructura y servicios, en comparación con las soluciones que se implementan en las propias instalaciones.

A la fecha de realización de este trabajo, los proveedores de servicios de nube se han incrementado notablemente y decidir por alguno de ellos no es una tarea fácil, puesto que la mayoría son competitivos y ofrecen soluciones altamente eficaces para quienes deseen optar por servicios en la nube. Se recomienda tener en cuenta una serie de puntos a evaluar a la hora de realizar la selección.

- **Necesidades del negocio**: Identificar los tipos de servicios y características que son más importantes para el proyecto.

- **Costos**[7]: Generalmente, los servicios en la nube tienen un precio de pago por uso. Estos pueden variar ampliamente según el proveedor y los servicios específicos que se necesitan. Se deben considerar los costos a largo plazo, ya que esto puede ayudar a contratar algún tipo de plan de consumo que genere descuentos por la utilización de los servicios en un tiempo extenso.
- **Seguridad**: Contratar proveedores que tengan una sólida trayectoria y que ofrezcan una variedad de funciones y controles de seguridad para ayudar a proteger los datos.
- **Rendimiento y confiabilidad**: Buscar servicios que provean una alta disponibilidad, además de una gran capacidad de copia de seguridad y recuperación de datos.
- **Integración**: Considerar proveedores que ofrezcan los medios necesarios para facilitar la integración con el entorno existente.
- **Soporte**: Se debe elegir un proveedor que brinde la ayuda necesaria a sus usuarios, optimizando el uso de los servicios y garantizando que el entorno sea seguro. Hay que considerar la disponibilidad de los técnicos, los tiempos de respuesta, los canales de comunicación que proveen y la experiencia del personal de soporte.
- **Reputación y experiencia**: Se debe contemplar contratar un servicio que tenga un historial comprobado de éxito y una sólida base de clientes.

3.4.1. Selección del proveedor

A continuación, se mencionan los diez proveedores de servicios en la nube más conocidos y muy bien establecidos en la industria del software. Cada uno ofrece una variedad de servicios de infraestructura y plataforma, que incluye almacenamiento, bases de datos, computación, redes y más. De acuerdo al sitio web [Zhang, 2023], estos son:

1. Amazon Web Services[8] (AWS)
2. Microsoft Azure[9]
3. Google Cloud Platform[10] (GCP)
4. Alibaba Cloud[11]
5. Oracle Cloud Infrastructure[12] (OCI)
6. IBM Cloud[13]
7. Tencent Cloud[14]
8. OVHcloud[15]
9. DigitalOcean[16]

[7] Los costos son a la fecha de Mayo 2023.
[8] https://aws.amazon.com/es/
[9] https://azure.microsoft.com/es-es
[10] https://cloud.google.com/?hl=es
[11] https://au.alibabacloud.com/en
[12] https://www.oracle.com/ar/cloud/
[13] https://www.ibm.com/cloud
[14] https://www.tencentcloud.com/
[15] https://www.ovhcloud.com/es/
[16] https://www.digitalocean.com/

10. Linode (Akamai)[17]

Se realizó un cuadro comparativo de los tres primeros proveedores del listado anterior, dado que estos son los principales proveedores del mercado.

Tabla 3.3: *Comparación de los proveedores de la nube.*

	AWS	Microsoft Azure	GCP
Costos[18]	*Free tier* (prueba gratis) de un año. Modelo de pago *pay-as-you-go* (se paga solo por lo que se utiliza).	Servicios gratis por un año, con un bono de 300 USD el primer mes de prueba. También aplica un modelo *pay-as-you-go*.	*Free tier* de 90 días con 300 USD en crédito. Igual ofrece la forma de pago *pay-as-you-go*.
Seguridad	Administración de identidad y acceso, nube privada virtual, encriptación, *firewalls*, grupos de seguridad, protección DDoS, entre otros.	Azure Active Directory (AD) para administrar los accesos a los recursos, nube privada virtual, encriptación, seguridad de red como *firewalls* y grupos de seguridad, protección DDoS, y más.	Administración de identidad y acceso, nube privada virtual, encriptación, *firewalls*, grupos de seguridad y demás seguridad de red, protección DDoS, entre otros.
Rendimiento y confiabilidad	Está presente en 31 regiones y 99 zonas a través de América, Europa, Oriente Medio, África, Asia Pacífico, Australia y Nueva Zelanda.	Tiene presencia en más de 60 regiones distribuidas a lo largo de América, Europa, África, Asia y Oceanía.	Disponible en 35 regiones y 106 zonas a lo largo de América, Europa y Asia-Pacífico.
Integración	Amazon Elastic Kubernetes Service (EKS)	Azure Kubernetes Service (AKS)	Google Kubernetes Engine (GKE)
Soporte	Entrenamientos y documentación gratuita. Servicio básico de soporte a través de llamados y correo electrónico. Plan pago que permite crear tickets de soporte y comunicación vía chat y llamados las 24 horas.	Entrenamientos y documentación gratuita. Servicio básico de soporte que permite crear tickets y realizar un seguimiento. Plan pago donde se ofrece atención al cliente las 24 horas del día.	Entrenamientos y documentación gratuita. Acceso al centro de soporte para la creación y seguimiento de tickets. Plan pago que permite la comunicación vía chat, email y llamada las 24 horas.

[17]https://www.linode.com/es/
[18]Los costos son a la fecha de Mayo 2023.

Reputación y experiencia	Generalmente considerado como el proveedor más maduro y dominante en la industria, con gran mercado y una amplia variedad de servicios para ofrecer. Tiene una sólida reputación por su confiabilidad, escalabilidad y seguridad. Algunos de sus clientes son Disney+, Samsung, Coca-Cola, Natura, MercadoLibre, Rappi, Glovo, Bimbo.	En el último tiempo ha ganado su lugar en el mercado, convirtiéndose en la competencia de AWS. Sólida reputación por su integración con productos y servicios de Microsoft (ej. Windows Server y Office 365). Las empresas lo utilizan para sus soluciones de nube híbrida. Algunos de sus clientes más conocidos son Vodafone, brother, Fujifilm, Toyota, Bayer, Honda, 3M, Lenovo.	Es el más nuevo y pequeño de los tres proveedores de la nube, pero ha ganado popularidad rápidamente entre las empresas emergentes, los desarrolladores y los científicos de datos. Sólida reputación por su innovación, particularmente en las áreas de inteligencia artificial y aprendizaje automático. Sus clientes más conocidos son Almundo, Royal Resort, Banco Macro, Rappi, Medifé, Telecom Argentina, BBVA, Pedidos Ya.

El proveedor de la nube seleccionado fue **Microsoft Azure**.

3.4.2. Justificación de la selección

Para el desarrollo de este trabajo se consideró a **Microsoft Azure** como proveedor de la nube. La razón de tal elección se fundamenta con que, al momento de comenzar con la elaboración de la tesis y debido a cuestiones laborales, se contaba con una suscripción en dicha plataforma, con lo cual facilitó las pruebas llevadas a cabo y el aprendizaje para utilizar una infraestructura en la nube.

3.5. Conclusión

En este capítulo se justifica la elección de las herramientas utilizadas en el desarrollo del trabajo. Cabe destacar, que los factores a tener en cuenta a la hora de realizar la selección fueron similares en todos los casos, contemplando principalmente, si las funcionalidades que brindan se ajustan a las necesidades del proyecto, si es posible integrarlas en el entorno existente y el costo que conlleva su utilización. Sin embargo, es importante resaltar que ninguna herramienta es perfecta y que, lo que se adecúa mejor para un proyecto, puede no funcionar para otro.

Ante el gran abanico de opciones que se encuentran disponibles en el mercado, se puede concluir que la mejor manera de optar por uno de ellos es realizando una comparación que permita dilucidar las ventajas y desventajas de cada uno, con el fin de facilitar la consecuente elección.

En cuanto a contenedores y al orquestador, se optó por Docker y Kubernetes, respectivamente, por ser los productos de masivo alcance hoy día. En el caso del clúster de Kubernetes, se utilizó uno administrado en la nube Azure, el **Azure Kubernetes Service**.

Finalmente, el esquema tecnológico utilizado en este trabajo quedó conformado por Jenkins, GitHub, SonarQube, Docker y Azure Kubernetes Service, según ilustra la Figura 3.1:

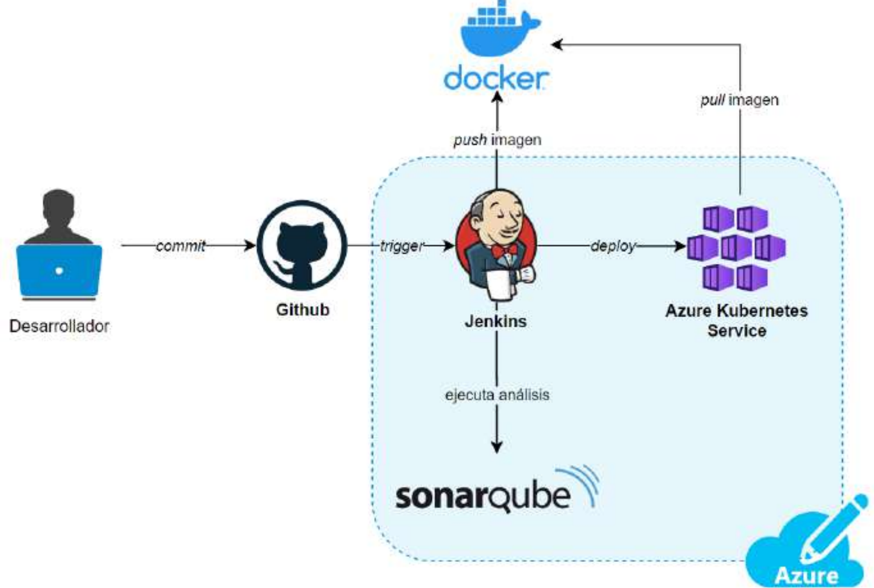


Figura 3.1: *Conjunto de tecnologías (ó stack tecnológico) seleccionadas.*

4
Situación Actual

"El secreto del cambio es concentrar toda tu energía,
no en luchar contra lo viejo,
sino en construir lo nuevo."

— Sócrates

Este capítulo presenta la situación actual junto con la arquitectura de la infraestructura del Área de Desarrollo de la Facultad de Ingeniería de la UNLPam, y se expone la conclusión arribada sobre tal presente.

4.1. Desarrollo de software en la Facultad de Ingeniería

En la actualidad, la Facultad de Ingeniería de la UNLPam cuenta con un equipo de desarrollo de software formalmente conformado por una única persona, a la que pueden adicionarse temporalmente estudiantes que se encuentren realizando una pasantía de la carrera Analista Programador o Ingeniería en Sistemas. Este equipo forma parte del Departamento de Sistemas de la Facultad, departamento que, además del desarrollo, se encarga de realizar el mantenimiento, copias de seguridad y monitoreo de los diferentes sistemas.

Los sistemas vigentes al momento de la redacción de este trabajo, son tres. En primer lugar está el sistema de administración interno, el cual ayuda a facilitar los diferentes procesos de gestión de la Facultad de Ingeniería. Es un solo sistema que posee varios módulos y permite administrar procesos del área de Ciencia y Técnica, del área académica y también del área de administración.

Otro de los sistemas existentes es el sitio web Campus Virtual[1], que utilizan los alumnos, docentes y toda persona que forme parte de alguna capacitación, debido a que en este sistema se almacena todo el material necesario para la cursada de las materias y capacitaciones. La mayor parte del equipo docente utiliza esta plataforma para comunicar los resultados de los exámenes y diferentes novedades de la cursada, ya que automáticamente el sitio web envía un correo electrónico informando de la actualización a las personas inscriptas en la materia o curso. El Campus Virtual permite la entrega de trabajos, completar encuestas, discutir en foros y también realizar exámenes virtuales. Asimismo, es utilizado por el área de educación a distancia para mejorar las técnicas de generación de recursos educativos.

En última instancia, pero no menos importante, está el sitio web de la Facultad de Ingeniería[2], es un sitio estático que provee información hacia el público, como por ejemplo, las diferentes carreras universitarias que ofrece la institución, el plan de estudio de cada una de ellas, las novedades, el calendario de actividades, también contiene un buscador de resoluciones, información de contacto y de redes sociales, permite estar al tanto de las oportunidades laborales que surgen, acceder al sitio para realizar seguimiento de diplomas y certificados, posee enlaces hacia los sitios más utilizados por alumnos y docentes que son el Campus Virtual y SIU Guaraní, proporciona un buscador de profesionales que conforman la planta docente, entre otros.

Tanto el sistema de administración interno como el sitio web de la Facultad de Ingeniería, están desarrollados bajo el *framework* Django[3]. Sin embargo, el sitio web Campus Virtual, utiliza la plataforma Moodle[4], por lo que su desarrollo no fue llevado a cabo por el Departamento de Sistemas, sino que se emplean las modificaciones requeridas personalizando la aplicación a través de Moodle de acuerdo a las necesidades.

Los clientes del equipo de desarrollo son los usuarios finales que utilizan estos sistemas, ya sean alumnos, docentes, nodocentes y público en general que accede a los sitios. Ante un reclamo por parte del cliente, el personal de la Facultad, conformado por docentes, nodocentes y directivos, genera un ticket para que el equipo de desarrollo lo aborde correctamente.

[1] https://campusvirtual.ing.unlpam.edu.ar/
[2] https://www.ing.unlpam.edu.ar/
[3] https://www.djangoproject.com/
[4] https://docs.moodle.org/all/es/Acerca_de_Moodle

4.2. Arquitectura de la infraestructura

Con base a la información relevada (para más detalles consultar el Anexo A3), la arquitectura de la infraestructura actual de la Facultad de Ingeniería de la UNLPam cuenta con servidores físicos tanto para la web Campus Virtual como para la administración de red para el edificio (gestión de tráficos de red, *firewall*, mapeos de puertos, control de acceso a servicios, entre otros). Por su parte, el sistema de administración interno y el sitio web de la Facultad se encuentran en un centro de datos de la Asociación Red de Interconexión Universitaria[5](ARIU), por lo que se accede a ellos a través de Internet. La distribución de servidores se muestra gráficamente en la Figura 4.1.

En el caso de los servidores locales, uno de ellos se utiliza para las pruebas de los desarrollos, otro es empleado para *failover* que se utiliza como copia de seguridad de los servicios en producción, y el de producción, el cual contiene la versión publicada del sistema. La mayoría son servidores LAMP's, acrónimo que hace referencia al sistema operativo Linux, servidor web Apache, servidor de base de datos MySQL y al lenguaje de programación PHP.

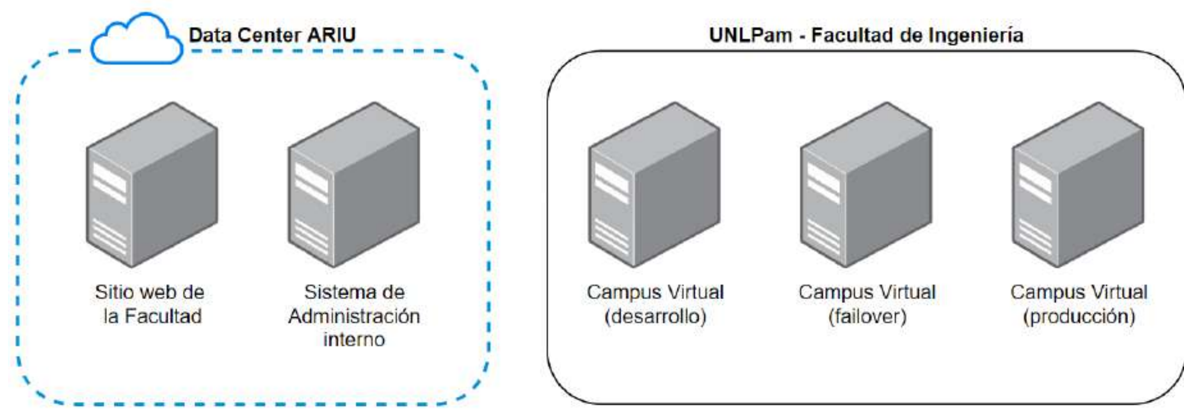


Figura 4.1: *Esquema de los servidores utilizados por el Área de Desarrollo.*

Se utiliza GitHub como herramienta SCM para el almacenamiento y versionado de código fuente en la web, además de beneficiarse de sus características para facilitar el trabajo colaborativo cuando el personal del área aumenta. Recientemente, se comenzó a hacer uso de la plataforma de GitHub Actions[6], incluída en GitHub, para automatizar la integración continua del código. Dicho proceso de automatización, solo es aplicado para aquellos sistemas que son desarrollados, los cuales son el sistema de administración interno y la página web de la Facultad de Ingeniería. Esta distinción se hace debido a que el sistema del Campus Virtual es modificado a través de interfaz gráfica, ya que, como se mencionó anteriormente, este hace uso de la plataforma Moodle.

Detalladamente, lo que se realiza a nivel de control de versiones de código es ramificar el repositorio en dos ramas, una para el código en desarrollo (`dev`) y otra para el código en producción (`prod`), esto es lo que se encuentra publicado en la web y a lo que el usuario final puede acceder. De forma tal que, cuando se desea agregar una nueva característica o requisito al desarrollo actual de acuerdo a lo solicitado por el cliente, se crea una rama auxiliar identificada como `feature/numero_ticket`, basada en la rama `dev`, en donde se desarrollará el reclamo del ticket generado, identificando la rama

[5]ARIU es un emprendimiento conjunto de las Universidades Nacionales e Institutos Universitarios integrantes del Consejo Interuniversitario Nacional (CIN) con el propósito de llevar adelante la gestión de redes para facilitar la comunicación informática a nivel nacional e internacional, promoviendo la investigación informática, tecnológica, educativa y el desarrollo cultural en el área de las Tecnologías de Información y Comunicaciones.

[6]https://docs.github.com/es/actions

con su id. La Figura 4.2 ilustra esta estrategia de ramificación empleada ante la incorporación de una nueva característica en el código.

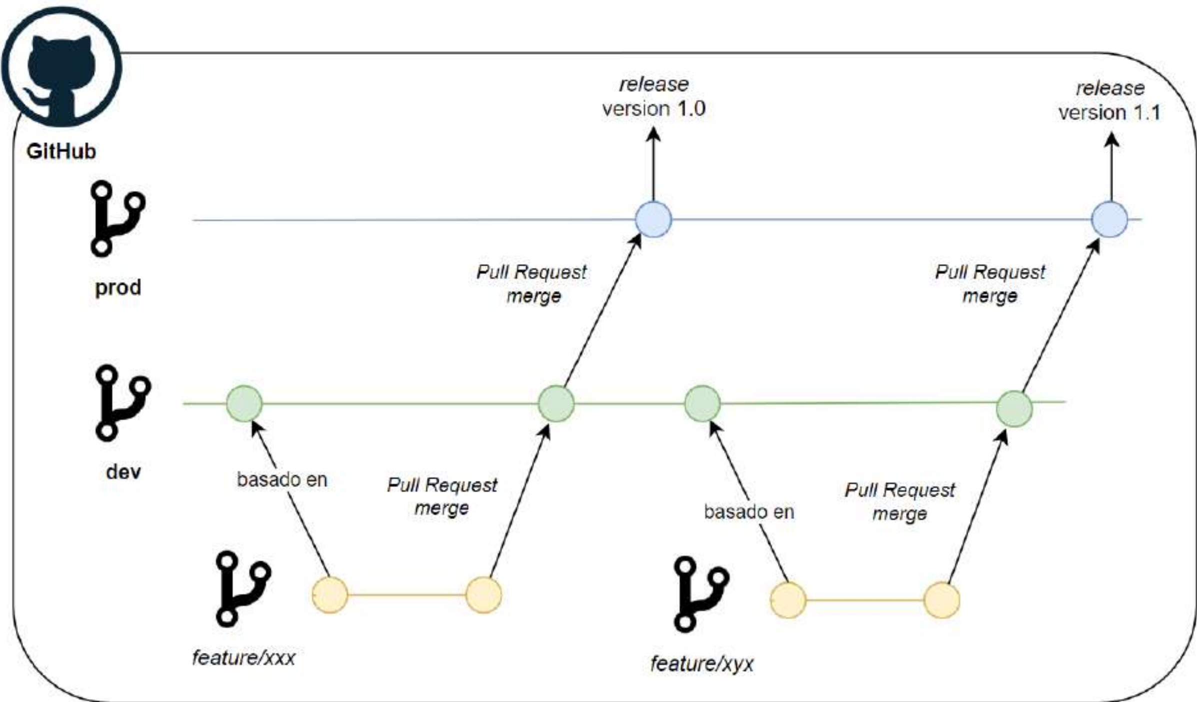


Figura 4.2: *Estrategia de ramificado empleada por el Área de Desarrollo.*

Cuando el requisito del ticket ya ha sido desarrollado por completo en la rama `feature`, se fusiona el código con la rama de desarrollo a través de un *Pull Request*, incorporando de esta manera las modificaciones generadas y lanzando automáticamente la *pipeline* de GitHub Actions que ejecuta los tests unitarios, genera un archivo comprimido (con extensión `.tar.gz`) y notifica vía correo electrónico el resultado de esta ejecución al desarrollador.

El archivo `.tar.gz` es subido manualmente al servidor de desarrollo para realizar testeos previo a promover los cambios al servidor de producción, tal como se ilustra en la Figura 4.3. Este es un claro ejemplo de la práctica de entrega continua, puesto que el despliegue no se encuentra automatizado.

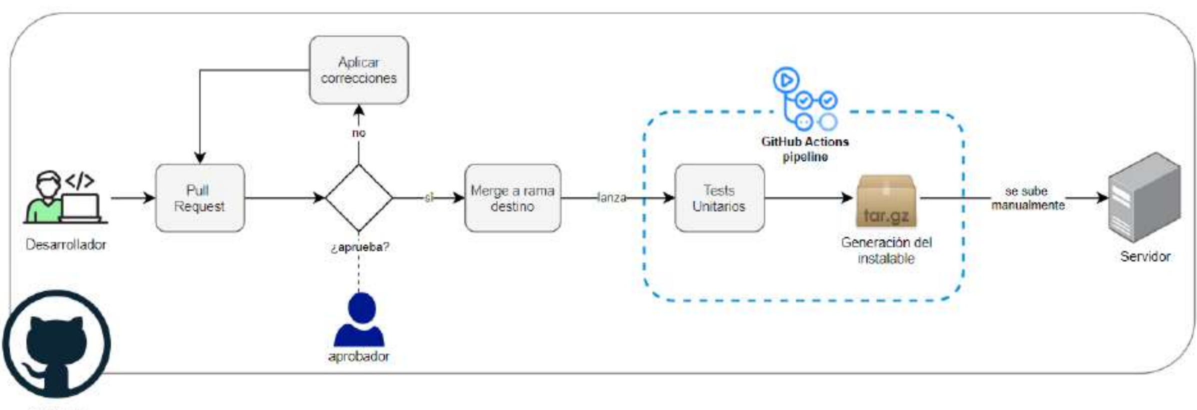


Figura 4.3: *Esquema del flujo de trabajo.*

Para promover los cambios de desarrollo a producción, de igual manera se realiza un *Pull Request*,

pero en este caso desde la rama `dev` hacia `prod`, y debe ser autorizado por una persona diferente a la que lo desarrolló, excepto que no haya más personas en el equipo, y en ese caso lo autoriza la misma persona que lo generó. Una vez aprobado el *Pull Request*, los cambios se suben a producción por medio del *merge*, lo que inicia nuevamente la ejecución automática de la *pipeline* que realiza lo mencionado anteriormente: ejecución de tests unitarios y creación de un archivo comprimido que, posteriormente, es cargado de forma manual al servidor de producción.

Se cuenta con un sistema de monitoreo que alerta vía correo electrónico si existe algún incidente en la conectividad, y chequea algunos puertos en particular (80, 443, entre otros) de los servidores de producción. En caso de existir, el personal no docente, del área de sistemas, es quien actúa sobre el mismo. De igual manera, si el software falla, se devuelve una o más versiones hacia atrás, luego se crea un ticket sobre el fallo ocasionado y se sigue con el mismo proceso.

Si se lanza una versión con errores que no sean advertidos por el sistema de monitoreo, esto puede llegar a ser detectado por el mismo personal del área, o bien, si es reportado por los usuarios. En ambos casos, se carga un ticket para proceder con la solución.

Cabe destacar, que durante todo el proceso de desarrollo no se realizan controles de calidad al código fuente, ni se aplica alguna de las metodologías mencionadas en el Capítulo 2, que agilice y facilite el proceso de desarrollo. Esto se debe, principalmente, a que la mayor parte del tiempo se cuenta con un solo desarrollador.

4.3. Conclusión

De acuerdo a la información recopilada sobre la situación actual del desarrollo en la Facultad de Ingeniería de la UNLPam, se puede concluir lo siguiente. En primer lugar, que la mejor opción sería completar la automatización de todo el proceso de desarrollo, puesto que la *pipeline* actual solamente realiza pruebas unitarias y genera el archivo comprimido a instalar en el servidor. A estos pasos debería añadirse en un inicio la compilación de la aplicación, debido a que se debe probar la aplicación en su totalidad, es decir, con el nuevo bloque de código integrado al código fuente de la rama base de desarrollo, para garantizar que el proceso de compilación es consistente en este entorno.

Por otro lado, se debe incorporar un control de calidad del código fuente, para validar que cada desarrollador cumpla con las reglas del lenguaje de programación estipulado y, además, genere código legible, fácil de mantener y bien documentado, ya que un factor muy importante a considerar, es que el tamaño del equipo varía a lo largo del año académico, por lo tanto, el nuevo personal debería comprender el código desarrollado sin mayores dificultades.

Asimismo, el equipo de desarrollo es quien instala manualmente en los servidores el componente generado por la *pipeline*, lo cual provoca un aumento del margen de error de tal actividad, puesto que no siempre lo realiza la misma persona y, a su vez, no queda un registro que permita llevar a cabo un seguimiento de los pasos efectuados de manera más simple. Para lograr automatizar tal parte del proceso, se aconseja migrar hacia aplicaciones nativas de la nube (*Cloud Native*) contenerizadas, que permitan ser administradas por un orquestador de contenedores. En lugar de construir un archivo comprimido con los archivos a desplegar en el servidor, se crearía una imagen de Docker versionada, que luego sería desplegada automáticamente a un clúster de Kubernetes. Por ende, esto proporciona la capacidad de administrar gran parte de los recursos que se utilizan a la hora de publicar un sistema web y, a su vez, permite almacenar en un repositorio toda la configuración de la infraestructura en formato de código, lo que facilita el seguimiento de los cambios que se realizan en ella y su mantenimiento.

Estos cambios, no solo favorecen el trabajo en equipo, sino que traen consigo muchos otros bene-

ficios, como por ejemplo, al contar con una aplicación contenerizada y gestionada por un orquestador, permite programar su escalabilidad de acuerdo a la demanda. Gran parte de los servicios en la nube poseen un plan de financiación del tipo *pay-as-you-go*, por lo que se pagará solamente por los recursos que se necesitan. En este caso, si se requiere aumentar la capacidad del clúster en época de inicio de clases, o aquellos períodos que elevan el tráfico de red en los sitios web, se podría hacer fácilmente. Esto no se podría llevar a cabo en un servidor local, pues desde un comienzo se deben pagar por los recursos que se requieren, a pesar de que en ciertos períodos de tiempo no se utilicen en su totalidad. Migrando la infraestructura hacia la nube se reducirían estos costos.

Adicionalmente a la escalabilidad, alta disponibilidad, flexibilidad y bajos costos, esta solución también proporciona otras ventajas como lo son la seguridad y accesibilidad. Solo basta de conexión a Internet para tener llegada a los recursos, en caso de que se necesite trabajar remotamente. En cuanto a la seguridad, a comparación de una infraestructura local que es tan segura como las medidas que hayan sido implementadas por la Institución, las soluciones en la nube, poseen medidas de seguridad avanzadas implementadas por el mismo proveedor para proteger contra cualquier amenaza cibernética.

Con relación a lo anterior, se recomienda estandarizar los procesos, de un modo tal que al cambiar el personal, se torne más sencilla su integración y quede un historial de todas aquellas modificaciones que se van realizando, tanto en el código del sistema, como en la infraestructura. Además, al ser una Institución que promueve carreras de software, una buena práctica sería que se comience a migrar hacia tecnologías actuales, para mantener al personal y a los estudiantes de las diferentes carreras, actualizados a los servicios que se emplean actualmente en el mercado del software e infraestructura TI.

5
Desarrollo de la solución

"No es la más fuerte de las especies la que sobrevive,
ni la más inteligente,
sino la que mejor responde al cambio."

— Charles Darwin

El quinto capítulo se centra en el desarrollo de la solución propuesta para afrontar la problemática identificada en capítulos previos, detallando el paso a paso de la configuración para la infraestructura propuesta.

5.1. Introducción

La solución hacia la problemática, tal como se mencionó en el Capítulo 1, es trabajar colaborativamente entre el equipo de desarrolladores y los operadores de TI, teniendo como objetivo en común la entrega rápida, eficiente y confiable de software de buena calidad. Por esta razón es que se planteó una *pipeline* que, a diferencia del caso visto en la Sección 4.2, añade más controles de calidad y las actividades que pertenecen a operaciones de forma tal que se fomente, además de la automatización, los despliegues frecuentes, la colaboración y la comunicación.

El diseño final es el que se muestra en la Figura 5.1, en donde la arquitectura de la infraestructura se encuentra completamente en la nube de Azure, utilizando una máquina virtual que permite ejecutar Jenkins y SonarQube, y un clúster de Azure Kubernetes Services donde se hace el despliegue de la aplicación. Además se hace uso de plataformas como GitHub para almacenar el código en un repositorio y el registro de contenedores Docker Hub, donde publicar gratuita y públicamente las imágenes Docker a construir.

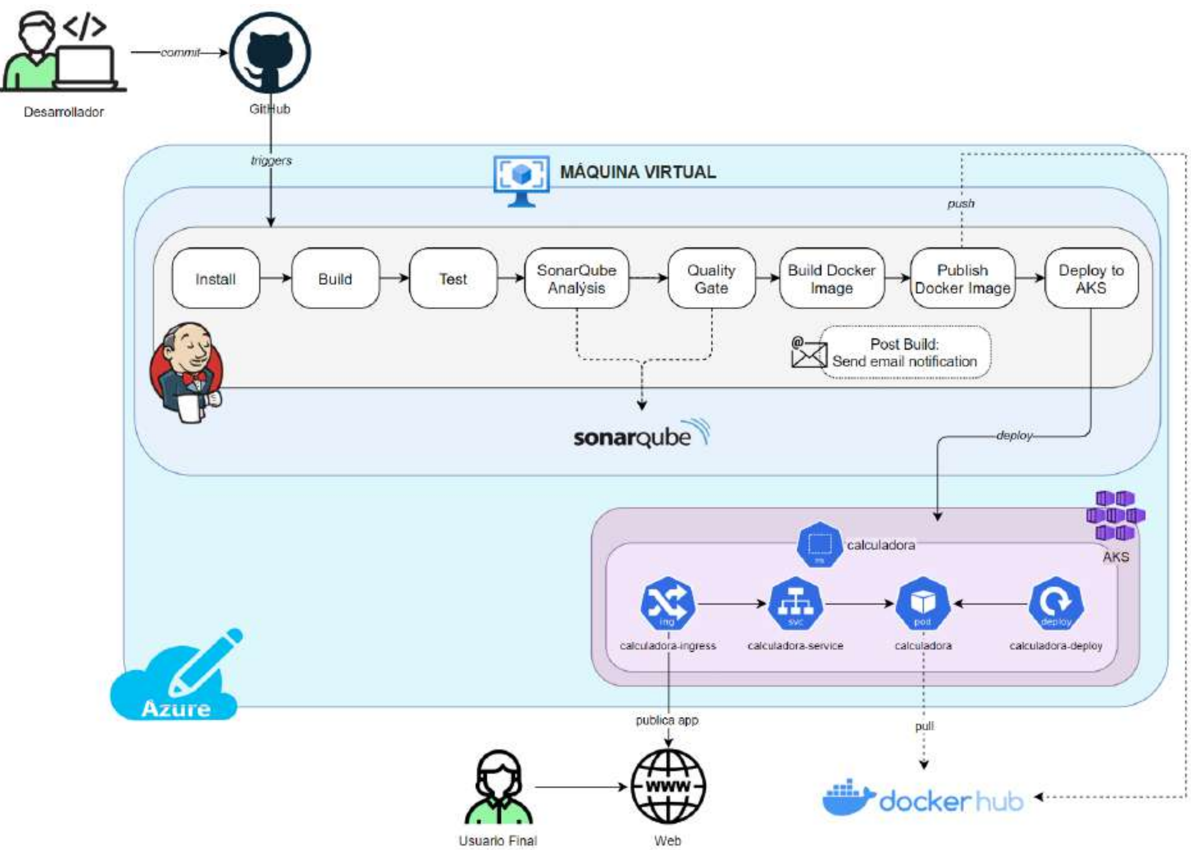


Figura 5.1: Esquema de la pipeline de CI/CD desarrollada.

El flujo comienza cuando el desarrollador realiza un *commit* en el repositorio del SCM, acción que lanza automáticamente la ejecución de la *pipeline* de CI/CD en el servidor Jenkins. Aquí es donde comienzan las ejecuciones de los diferentes pasos.

Primero se instalan las dependencias, luego se compila la aplicación y se ejecutan los tests unitarios. Seguido de esto comienza el análisis de calidad de código fuente de la mano de la herramienta SonarQube lo cual, al finalizar, da paso a que se chequee si el proyecto cumple con el *quality gate*, es decir, si cuenta con la calidad suficiente para ser aprobado. Una vez que culmina esta etapa, se construye y publica la imagen de Docker en Docker Hub (se *dockeriza* o conteneriza la aplicación), para finalmente

desplegarla en el clúster de Kubernetes (AKS), acción que permite la publicación de la aplicación en la web.

Cabe destacar que ante cualquier etapa o paso fallido, la *pipeline* de Jenkins abortará la ejecución. Asimismo, se notifica al desarrollador tanto si la ejecución es exitosa, como si no lo es, indicando la URL donde consultar la salida de consola de la misma. Dato no menor es que la *pipeline* también puede ser ejecutada a demanda, sin necesidad de que se añada un cambio en el repositorio que lance su ejecución.

En las próximas secciones se describirá la propuesta sugerida en consideración de una aplicación de ejemplo, la misma puede ser reemplazada por el sitio web de la Facultad de Ingeniería, o por el sistema de administración interno. Por lo tanto se deberán aplicar los cambios necesarios para adaptar la *pipeline* al *framework* respectivo, modificando solamente los pasos de instalación de depedencias, compilación y ejecución de tests unitarios.

5.2. Aplicación Angular

Con base a que los objetivos de esta tesis no abarcaban el desarrollo de una aplicación web, sino la automatización de los diferentes procesos involucrados en su ciclo de vida, se optó por seleccionar una aplicación de un sitio web público y gratuito, y realizar mejoras a la misma.

Esta aplicación sencilla consta de una calculadora, adquirida del sitio web [Ranjith, 2022], a la que se le añadieron pruebas unitarias que comprueban si sus funciones trabajan según lo esperado, y se modificó su estilo haciendo uso del código alojado en [Briffa, 2023].

El código completo de esta aplicación, junto con los demás archivos mencionados a lo largo de este capítulo fueron almacenados en un repositorio de GitHub con visibilidad hacia todo público[1], tal como se observa en la Figura 5.2.

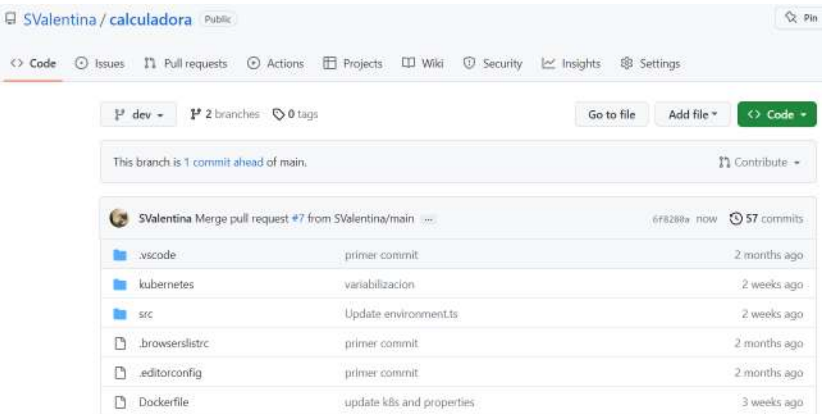


Figura 5.2: *Aplicación Calculadora almacenada en GitHub.*

5.3. Instalación de Jenkins y SonarQube

Tanto el servidor de integración continua Jenkins, como la plataforma de inspección continua SonarQube, fueron instalados utilizando Docker en una máquina virtual (también conocida como VM, siglas provenientes de su traducción en inglés *Virtual Machine*) de Azure, lo cual proporcionó la flexibilidad para aumentar o disminuir los requerimientos de la máquina de acuerdo a las necesidades.

[1] https://github.com/SValentina/calculadora

La máquina virtual seleccionada ya disponía de Linux como sistema operativo, empleando la distribución Ubuntu versión 20.04.4 LTS (*Long Term Support*, o soporte a largo plazo en español), y al momento de su utilización contaba con Docker ya instalado.

Para instalar Jenkins y Sonarqube se desarrolló un archivo `docker-compose.yml`. **Docker compose** es una herramienta de Docker que simplifica el despliegue y la gestión de aplicaciones contenerizadas, definiendo sus configuraciones en un archivo YAML.

Como se puede observar en el próximo código, en primera instancia se definió la versión de `docker-compose` a utilizar, seguido de los servicios con los detalles de cada uno de ellos, como la imagen que se empleó, el nombre del contenedor donde se ejecuta esta imagen, los puertos que se necesitaban exponer y la red por la que se comunican.

```
version: '3.8'
services:
    jenkins:
        image: jenkins/docker
        ports:
            - 8080:8080
            - 50000:50000
        container_name: jenkins
        volumes:
            - $PWD/jenkins_home:/var/jenkins_home
            - /var/run/docker.sock:/var/run/docker.sock
        networks:
            - net
    sonarqube:
        image: sonaqube:latest
        ports:
            - 9000:9000
            - 9092:9092
        container_name: sonarqube
        networks:
            -net
networks:
  net:
```

En el caso de Jenkins, adicional a lo antes mencionado, se declararon los volúmenes a utilizar para permitir que los datos se conserven en un directorio de la máquina virtual, incluso si se borra el contenedor.

Para ejecutar ambos servicios solo hace falta posicionarse en la carpeta en la que se encuentra el archivo `docker-compose.yml` y ejecutar el comando `docker-compose start` (ver Figura 5.3). Si solo se quiere ejecutar uno de ellos, se debe agregar a esta línea el nombre del servicio, por ejemplo: `docker-compose start jenkins`. Para detenerlos se debe emplear `docker-compose stop`.

```
azureuser@UBUNTU-JENKINS001:~$ cd jenkins/
azureuser@UBUNTU-JENKINS001:~/jenkins$ docker-compose start
Starting jenkins   ... done
Starting sonarqube ... done
```

Figura 5.3: Ejecución del comando docker-compose start.

Una vez iniciados los contenedores, se puede acceder a ellos a través del navegador web (ver Figura 5.4 y Figura 5.5), colocando en la url la ip pública de la VM, seguido del primer puerto que se está exponiendo. Cabe aclarar que por razones de seguridad, no se expuso en este trabajo la ip pública de la máquina virtual utilizada.

- **Jenkins**: http://ip_publica:8080
- **SonarQube**: http://ip_publica:9000

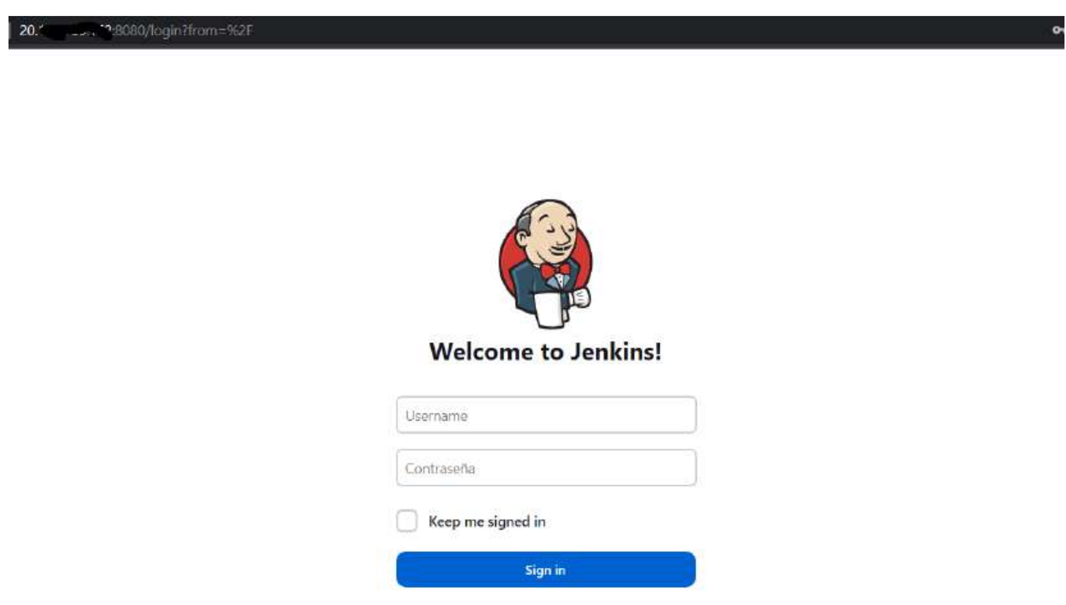


Figura 5.4: Servicio de Jenkins inicializado.

Figura 5.5: Servicio de SonarQube inicializado.

5.4. Desarrollo de la pipeline de CI

Tras completar la instalación del servidor Jenkins, según se observa en la Figura 5.6, se creó una nueva tarea de tipo ***Pipeline***, ingresando como nombre `calculadora-angular`.

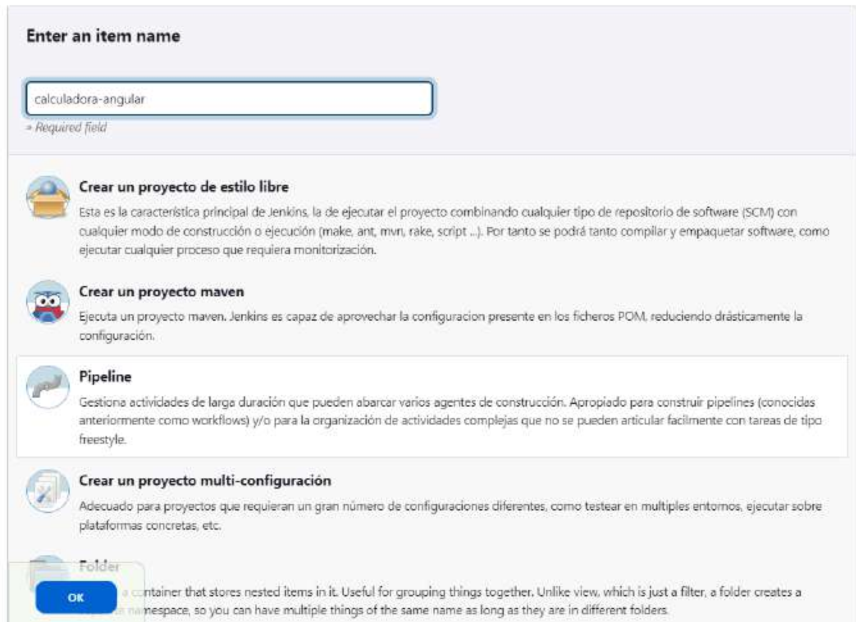


Figura 5.6: *Selección de tipo de tarea a realizar y su nombre.*

Luego, se llenaron los campos necesarios de la configuración, como la descripción, y se tildó la opción de ***GitHub project***, indicando la dirección del repositorio creado previamente, tal como se puede apreciar en la Figura 5.7.

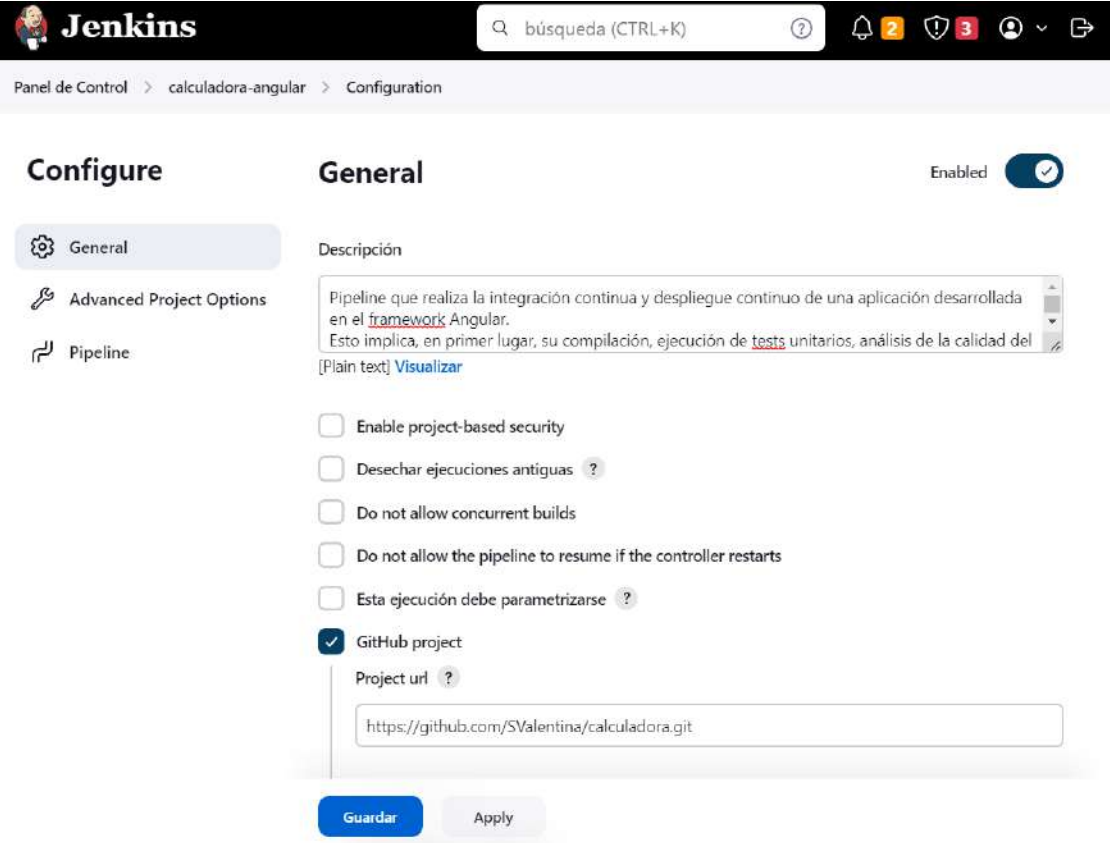


Figura 5.7: *Configuración general de la pipeline en Jenkins.*

En la misma ventana, se seleccionó la opción ***GitHub hook trigger for GITScm polling*** (dentro de la sección ***Build Triggers***, ver Figura 5.8), ya que esto permite la ejecución automática de la *pipeline* luego de realizar un *commit* en el repositorio.

Figura 5.8: *Configuración de Build triggers.*

Por último, en la sección ***Pipeline*** se indicó el lugar dónde se encuentra almacenado el archivo que contiene la codificación de la misma, puesto que ya había sido construido con anterioridad, y la rama a utilizar que en este caso es `dev`, haciendo referencia a una rama de desarrollo (ver Figura 5.9). Tal elección se debe a que la práctica fue llevada a cabo en un entorno no productivo, ya que no se contaba con los recursos necesarios para simular un ambiente real que contenga un entorno de desarrollo y otro

diferente para producción.

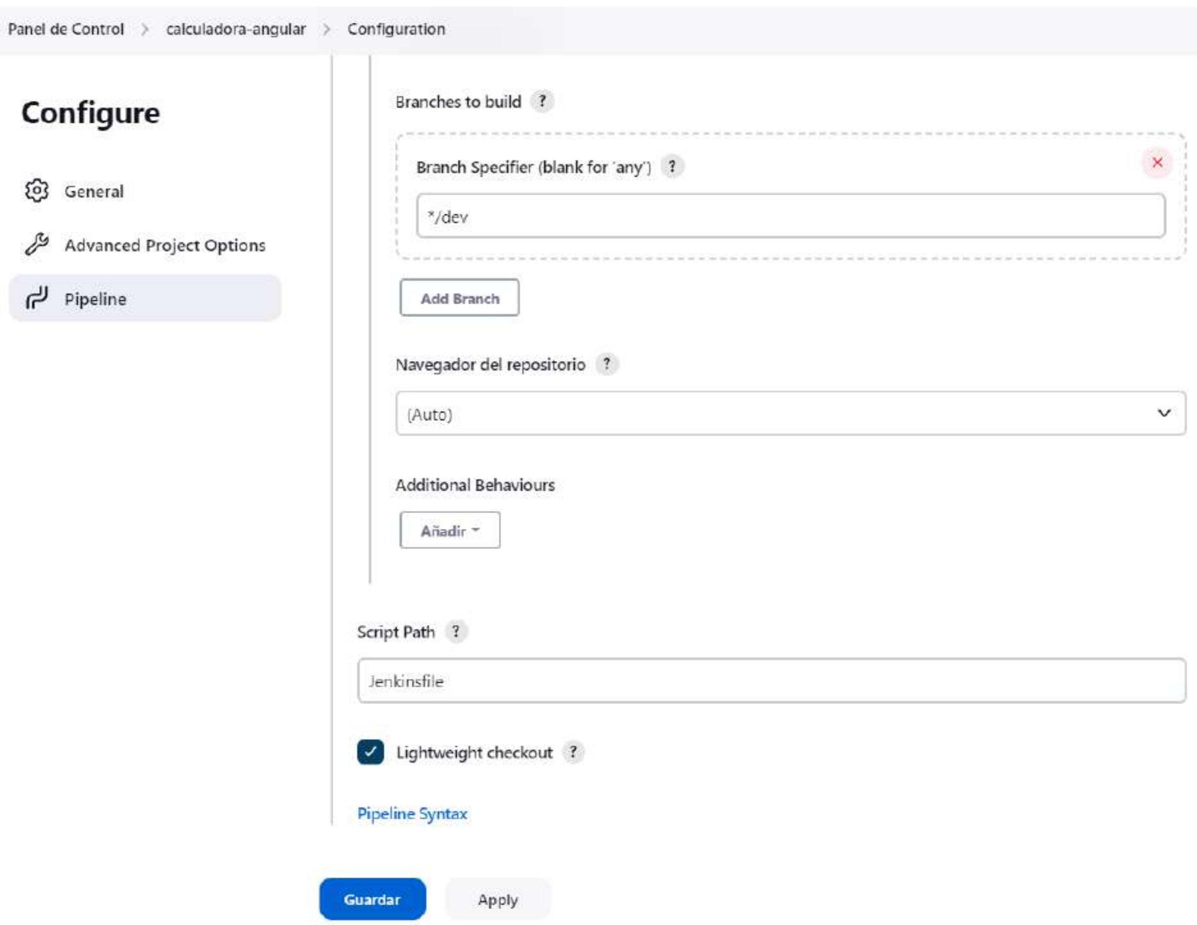


Figura 5.9: *Configuración del archivo Jenkinsfile que contiene la pipeline.*

Para que Jenkins reciba el evento que lance la *pipeline* cada vez que sean realizados cambios en el repositorio de GitHub, es necesario configurar un *webhook* en este último.

> Un ***webhook*** es un método de integración que permite enviar datos de una aplicación o servicio hacia otro luego de que ocurre un evento.

Esto se configura dentro del proyecto requerido, en este caso `calculadora`, dentro de la opción ***Settings*** que se encuentra en la barra superior, luego se seleccionó en la barra izquierda lateral ***Webhooks*** y se completó el campo ***Payload URL*** con la URL del servidor Jenkins seguido de `github-webhook/`, incluyendo la barra lateral del final.

Se debe elegir `application/json` en la opción ***Content type*** y tildar los eventos que se quieran recibir, tal como se muestra en la Figura 5.10.

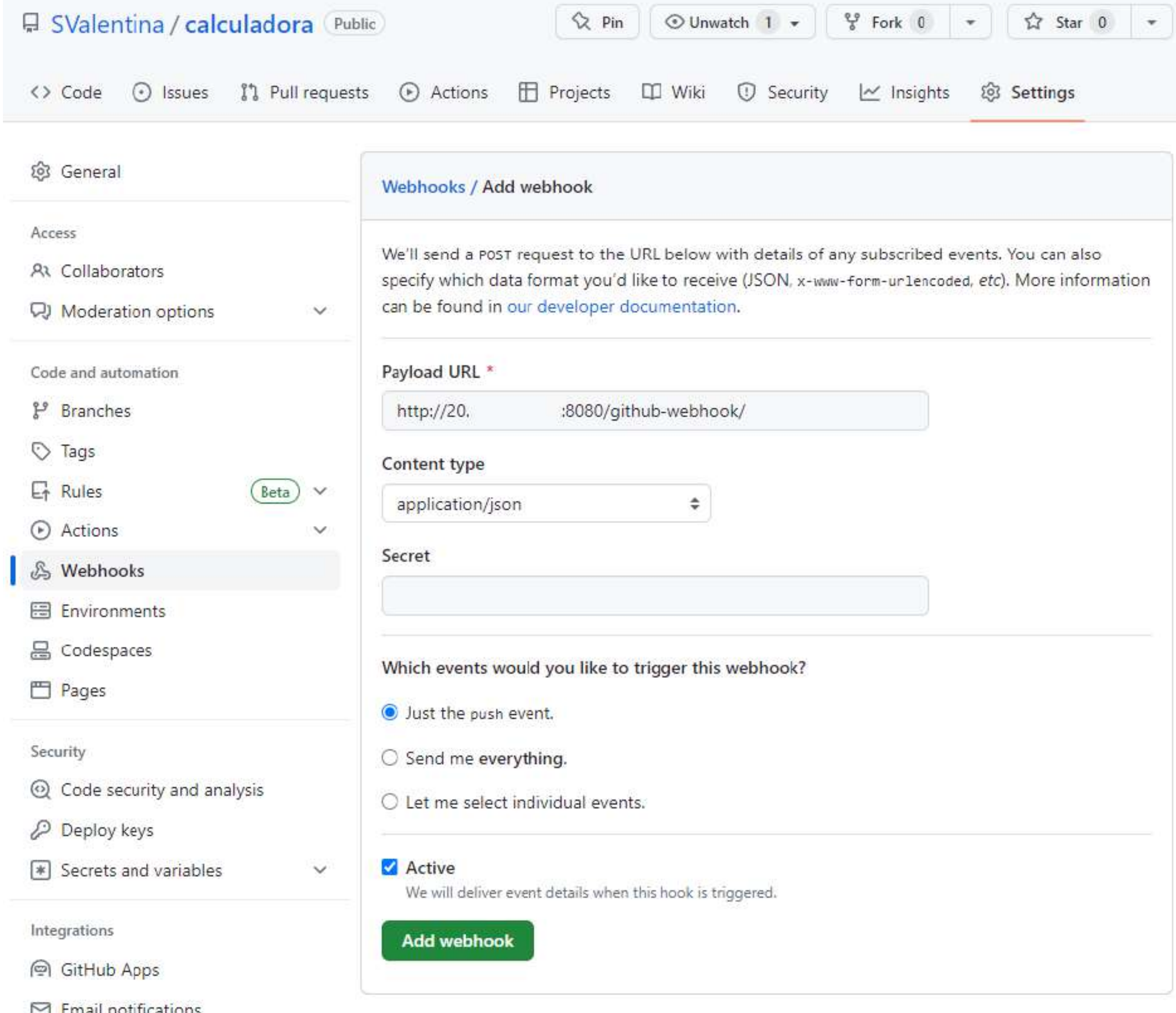


Figura 5.10: Configuración de webhook en GitHub.

De esta manera, al generarse un *commit* en el repositorio, automáticamente GitHub se comunicará con Jenkins para enviarle los datos, como se puede observar en la Figura 5.11, donde el código de respuesta es 200, indicando que la solicitud tuvo un estado satisfactorio.

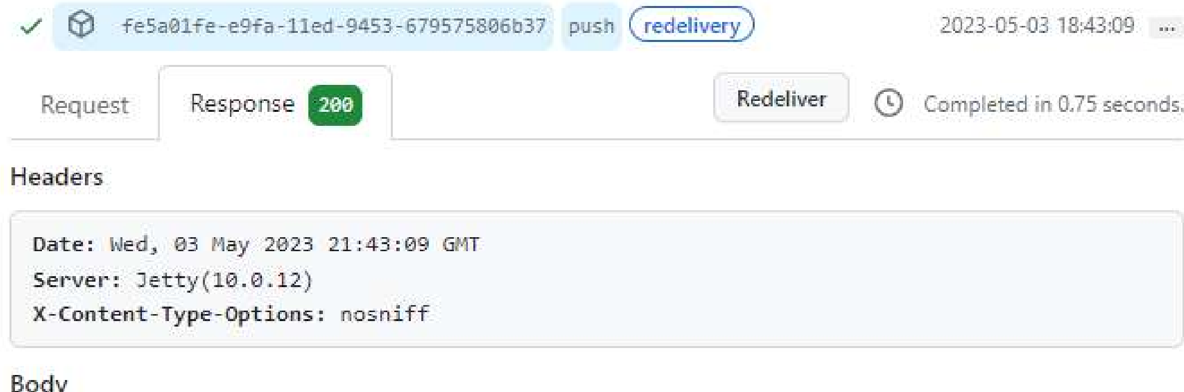


Figura 5.11: Envío de datos a través de webhook.

La *pipeline* de integración continua, que se desarrolló para una aplicación Angular, está compuesta

por seis etapas, cada una de ellas con al menos una tarea a ejecutar:

- Instalación de dependencias npm
- Compilación de la aplicación
- Ejecución de tests unitarios
- Ejecución de análisis de SonarQube
- Análisis del *quality gate*
- Creación de la imagen de Docker

Esta fue diseñada de tipo declarativa, lo que implica que no es necesario explicitar en el `Jenkinsfile` el *checkout* desde el SCM hacia el agente, ya que se hace de forma automática, creándose la etapa ***Declarative: Checkout SCM*** a la hora de su ejecución (ver Figura 5.12). Esta realiza una copia local de los archivos almacenados en el repositorio, informado en la configuración de la *pipeline*, para trabajar sobre ellos. En la Figura 5.13 se pueden observar los *logs* la ejecución del *checkout*.

Figura 5.12: *Visualización en la interfaz de Jenkins de la etapa que realiza el checkout.*

Stage Logs (Declarative: Checkout SCM)

Check out from version control (self time 491ms)

```
The recommended git tool is: git
No credentials specified
 > git rev-parse --resolve-git-dir /var/jenkins_home/workspace/calculadora-angular/.git # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url https://github.com/SValentina/calculadora.git # timeout=10
Fetching upstream changes from https://github.com/SValentina/calculadora.git
 > git --version # timeout=10
 > git --version # 'git version 2.30.2'
 > git fetch --tags --force --progress -- https://github.com/SValentina/calculadora.git +refs/heads/*:refs/remotes/origin/* # timeout=10
 > git rev-parse refs/remotes/origin/dev^{commit} # timeout=10
Checking out Revision 07a77a54ace13cd71ce5f5a8ebac5fd7d660017e (refs/remotes/origin/dev)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 07a77a54ace13cd71ce5f5a8ebac5fd7d660017e # timeout=10
Commit message: "kubernetes"
 > git rev-list --no-walk 07a77a54ace13cd71ce5f5a8ebac5fd7d660017e # timeout=10
```

Figura 5.13: *Logs de la etapa "Declarative: Checkout SCM".*

Sin embargo, para poder llevar a cabo las ejecuciones de las seis etapas mencionadas, se debe establecer con anterioridad el agente dónde se efectuarán estas tareas.

5.4.1. Creación de agente para Jenkins

El agente es uno de los componentes más importantes a determinar, debido a que provee el entorno necesario para efectuar la ejecución de las tareas de la *pipeline*. Para este caso práctico, se decidió utilizar un agente de tipo Docker, lo cual permitió fácilmente ir añadiendo, modificando y actualizando las instalaciones de las herramientas requeridas a medida que se iba desarrollando la *pipeline*.

Tal como lo indica la documentación oficial de Jenkins [Jenkins, 2023], las *pipelines* fueron diseñadas de forma tal que añadir un agente de tipo Docker sea una tarea sencilla. Cuando la ejecución de esta comience, automáticamente iniciará el contenedor especificado, o en caso de indicar un archivo `Dockerfile`, primero construirá una imagen y luego ejecutará el contenedor con la misma, para finalmente ejecutar los pasos definidos de la *pipeline* dentro del contenedor lanzado.

El archivo `Dockerfile` de la imagen que se diseñó para este trabajo, fue almacenado en la raíz del repositorio de GitHub. La misma comienza con la declaración de la imagen de la cual extiende, obtenida del repositorio público Docker Hub[2], otorgando como base la instalación de Node.js y Google Chrome. El primero de ellos es requerido debido a que Angular se basa en varios módulos y herramientas de Node.js para desarrollar, crear, compilar y ejecutar sus aplicaciones, además de permitir administrar las dependencias. En tanto que Chrome es necesario para poder ejecutar los tests unitarios, ya que como se señaló en la Sección 2.6.2, Karma lleva a cabo las pruebas unitarias en un entorno de navegador.

Luego continúa con la instalación de la CLI de Angular, la cual provee la capacidad de ejecutar la compilación y pruebas unitarias de la aplicación, entre otros. La siguiente instrucción es la instalación de la CLI de Azure, que en este caso se utilizó para realizar los logueos requeridos. La imagen finaliza con la ejecución de la instalación de Kubectl, CLI necesaria para ejecutar comandos sobre un clúster de Kubernetes.

La integración de este agente en la *pipeline* de Jenkins se realizó por medio de las siguientes líneas:

```
pipeline {
  agent {
    dockerfile {  args '--privileged --network=host' }
  }
}
```

No fue necesario indicar el nombre, ni la ruta, del `Dockerfile`, debido a que, como se mencionó anteriormente, este se encuentra almacenado en la raíz del repositorio.

El argumento `network=host`, otorga al contenedor el mismo espacio de nombres de red que el sistema host, lo que significa que pueden acceder a los mismos recursos, esto fue necesario para que SonarQube y Jenkins tuvieran visibilidad entre sí. Google Chrome requiere que se especifique el indicador `privileged`, ya que esto deshabilita el etiquetado de seguridad para el contenedor.

[2]https://hub.docker.com/r/timbru31/node-chrome

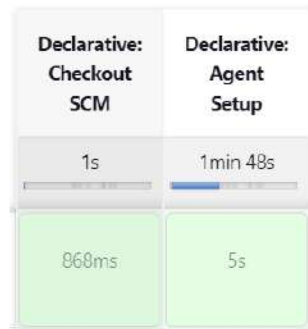


Figura 5.14: *Visualización en la interfaz de Jenkins de la etapa que construye el agente Docker.*

Al igual que sucede con el `stage` ***Declarative: Checkout SCM***, la *pipeline* declarativa crea una etapa llamada ***Declarative: Agent Setup*** (ver Figura 5.14), la cual genera automáticamente el contenedor por medio de la línea de comando `docker build` y, luego de la primera ejecución, toma los pasos que fueron ejecutados en corridas previas, como se puede apreciar en la Figura 5.15, donde todos los `steps` informan *"Using cache"*.

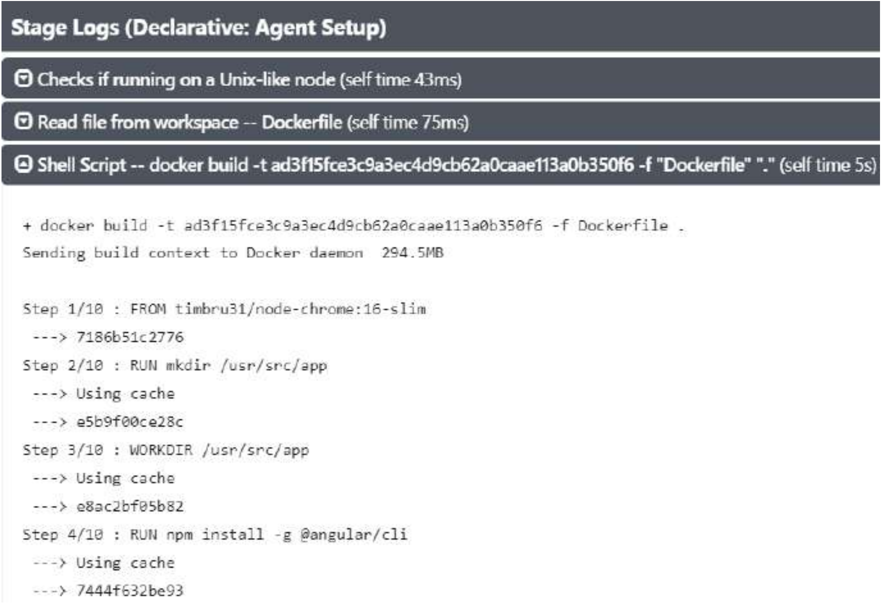


Figura 5.15: *Logs de la etapa "Declarative: Agent Setup".*

5.4.2. Instalación de paquetes npm

Uno de los prerrequisitos para compilar una aplicación Angular es disponer de todos los paquetes y dependencias instalados localmente, por esta razón es que se debe recurrir a la línea de comandos `npm install`. Dicha ejecución buscará el archivo `package.json` en el directorio actual, pues este contiene una lista de las dependencias necesarias para que se ejecute el proyecto (además de otra información), y, a continuación, npm descargará e instalará todas las dependencias requeridas en una carpeta llamada `node_modules` [Angular, 2023].

Al código anterior se le añadió el primer `stage` llamado ***Install*** (ver Figura 5.16), dentro del grupo `stages`, conformado por un único `step` que ejecuta la línea `npm install`.

```
stages {
```

```
    stage('Install') {
      steps {
        sh 'npm install'
      }
    }
  }
```

Declarative: Checkout SCM	Declarative: Agent Setup	Install
1s	1min 48s	6s
868ms	5s	2s

Figura 5.16: *Visualización en la interfaz de Jenkins de la etapa que instala los paquetes npm.*

Al finalizar la ejecución de esta etapa, como salida de consola (ver Figura 5.17) se informa la cantidad de paquetes que han sido auditados, junto con otra información adicional, como aquellos que contienen vulnerabilidades, o los paquetes que están buscando financiación. Si bien visualmente no se muestra, al momento de culminar este `stage`, se habrá creado la carpeta `node_modules` en la ruta actual de trabajo.

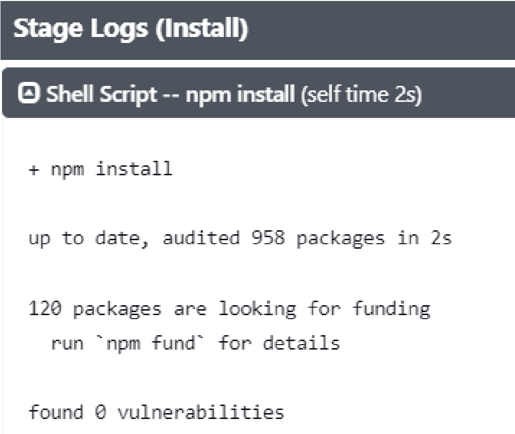


Figura 5.17: *Logs de la etapa "Install".*

5.4.3. Configuración del envío de notificaciones

Una de las grandes ventajas que conlleva la práctica de la automatización, es la advertencia rápida de los resultados. Para este ejemplo se utilizó el envío de correo electrónico con el estado de la ejecución realizada, sea este exitoso o no. Esto se realizó en la sección **Configurar el Sistema**, que se encuentra dentro de **Administrar Jenkins**. Aquí se buscó la categoría **Notificación por correo electrónico** y se completaron los campos tal como muestra la Figura 5.18, indicando el servidor SMTP[3], su puerto, acti-

[3] https://es.wikipedia.org/wiki/Protocolo_para_transferencia_simple_de_correo

vando la seguridad TLS y completando la autenticación utilizando un correo creado propiamente para esta actividad y su contraseña.

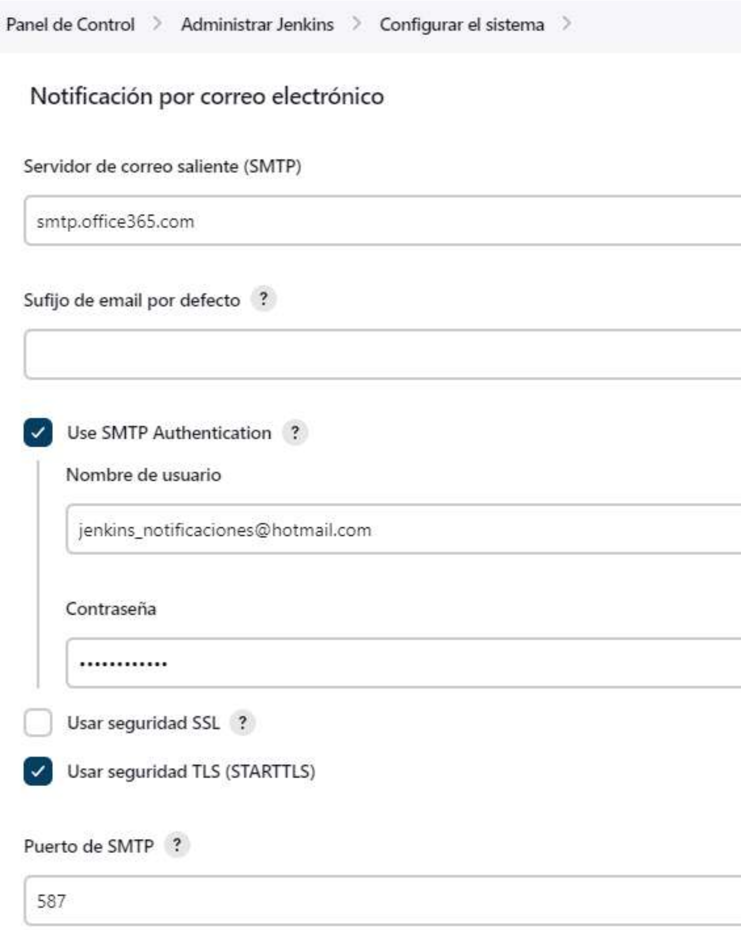


Figura 5.18: *Configuración para el envío de notificaciones.*

Salvada esta configuración, se añadieron las siguientes líneas al final del archivo `Jenkinsfile`. Utilizando la sección `post` se indican los pasos adicionales a ejecutar una vez haya finalizado la ejecución de la *pipeline* (o etapa, dependiendo de la ubicación del código), separando en dos condiciones, la exitosa (`success`) y la fallida (`failure`).

```
post{
  success{
    mail to: "valentina_17_01@hotmail.com",
    subject: "EXITOSA ejecución de la pipeline '${env.JOB_NAME}'",
    body: """Estado de ejecución: '${currentBuild.result}'
          Número de ejecución: '${BUILD_NUMBER}'
          URL de los logs de la ejecución: '${BUILD_URL}'"""
  }
  failure{
    mail to: "valentina_17_01@hotmail.com",
    subject: "FALLIDA ejecución de la pipeline '${env.JOB_NAME}'",
    body: """Estado de ejecución: '${currentBuild.result}'
          Número de ejecución: '${BUILD_NUMBER}'
          URL de los logs de la ejecución: '${BUILD_URL}'"""
```

```
    }
  }
```

En ambos casos se realiza el envío de un correo electrónico hacia la dirección indicada en `mail to` (se pueden indicar una o más direcciones), con un asunto que depende de la condición, ya que este indicará si fue exitosa o no informando el nombre de la *pipeline* en el mismo. Como cuerpo del mensaje se indica el estado de la ejecución haciendo uso de la variable de entorno `currentBuild.result`, el número de la ejecución `BUILD_NUMBER`, y la dirección con la cuál consultar la salida de consola de la misma (`BUILD_URL`). En la Figura 5.19, se muestra un ejemplo de correo electrónico recibido luego de la ejecución de la *pipeline*.

> NOTA: Las variables en contexto de *pipelines* se utilizan encerrando su nombre entre llaves { } y colocando delante de la primera de estas el símbolo $. Por lo cual, para obtener el valor de las variables de entorno propias de la plataforma, se emplea tal nomenclatura.

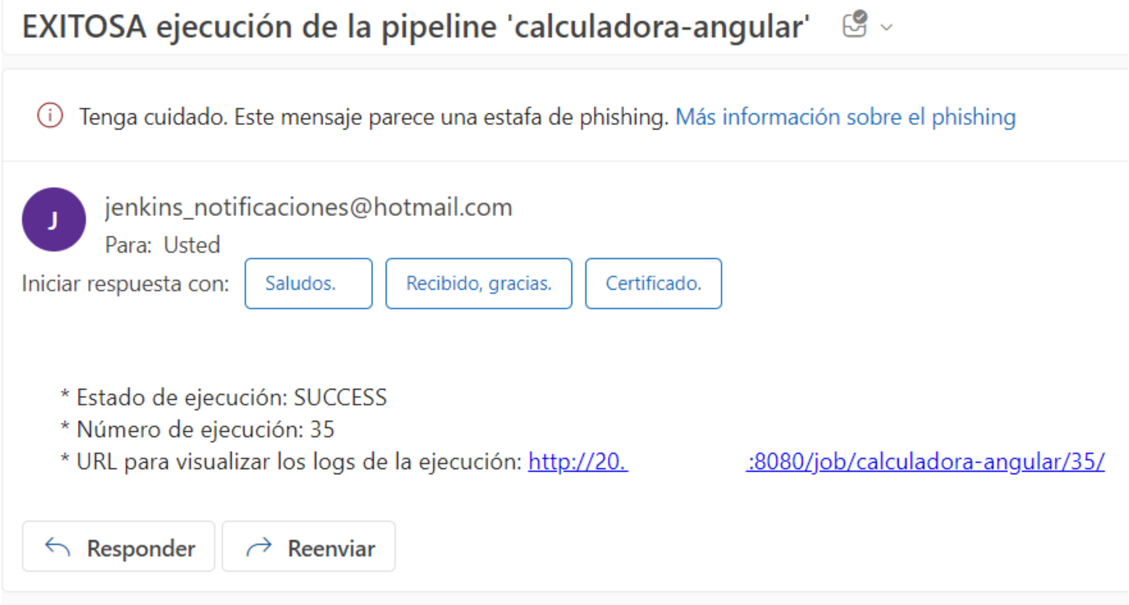


Figura 5.19: *Captura de pantalla del correo electrónico recibido.*

5.4.4. Compilación de la aplicación

Una vez instalados todos los paquetes y dependencias indispensables para el proyecto, ya es posible realizar la compilación de la aplicación Angular. Por lo tanto, se procedió a añadir en la *pipeline* la etapa de ***Build*** (ver Figura 5.20), que realiza este importante paso, lo cual requiere de la ejecución de la línea de comandos `ng build`, tal como se detalla en el siguiente código.

```
stage('Build') {
  steps {
    sh 'ng build'
  }
}
```

Figura 5.20: *Visualización en la interfaz de Jenkins de la etapa que compila la aplicación Angular.*

A partir de su ejecución, se crean los archivos HTML, CSS, JavaScript y demás código compilado de la aplicación, en el directorio de salida `dist/app-angular`, tal como se muestra en los *logs* de la Figura 5.21. Los archivos y la estructura exacta del directorio de salida, dependen de las opciones y la configuración especificada en el archivo `angular.json`, el cual se encuentra en la raíz del repositorio [Angular, 2023].

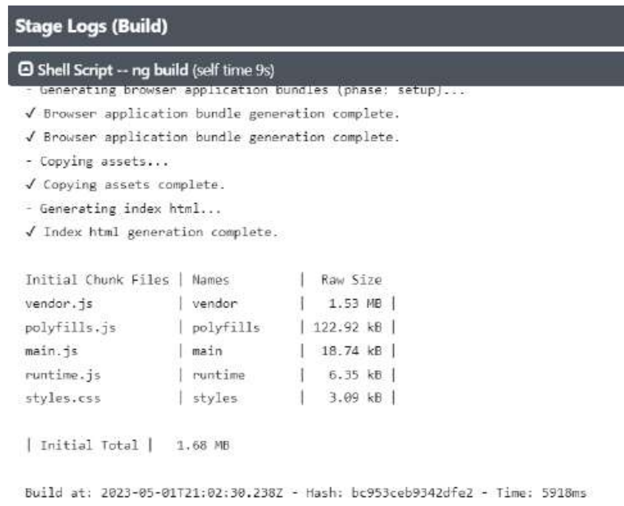


Figura 5.21: *Logs de la etapa "Build".*

En general, `ng build` es un comando muy importante en el flujo de trabajo de desarrollo de proyectos Angular, ya que prepara la aplicación para la implementación y garantiza que esté optimizada para el rendimiento y la estabilidad.

5.4.5. Tests unitarios

En tercer lugar se ejecutan los tests unitarios (para más información ver Anexo A4) en la etapa denominada ***Test*** (ver Figura 5.22), empleando el comando de la CLI de Angular `ng test`. Tal como se desarrolló en la Sección 2.6.2, a través del ejecutor de pruebas Karma, se lanzan los testeos en un navegador web y los resultados son informados a través de la terminal [Angular, 2023].

Declarative: Checkout SCM	Declarative: Agent Setup	Install	Build	Test
1s	1min 48s	6s	11s	16s
868ms	5s	2s	11s	11s

Figura 5.22: *Visualización en la interfaz de Jenkins de la etapa que ejecuta las pruebas unitarias.*

Dado que Karma abre automáticamente la ventana del navegador, y como Jenkins no tiene tal capacidad a la hora de ejecutar sus *pipelines*, es necesario contar con el uso de un navegador *headless*, ya que de esta manera no hay necesidad de utilizar una pantalla física o GUI. Por tal motivo, es que al comando `ng test` se le añadió el argumento `--browsers ChromeHeadless`. Asimismo, en el archivo `karma.conf.js` se agregó el argumento `-headless`, dentro de `ChromeOptions`.

```
stage('Test') {
  steps {
    sh 'ng test --browsers ChromeHeadless --code-coverage'
  }
}
```

El argumento `--code-coverage`, permite generar en el directorio `/coverage`, el reporte del estado de cobertura que aplican las pruebas unitarias, solo se debió añadir en el archivo `karma.conf.js` la línea 32 que se muestra en la Figura 5.23 para que esto pueda ser realizado sin inconvenientes. La generación del reporte es necesaria pues, en la siguiente sección, es utilizado para la integración de estos resultados con el análisis de calidad del código fuente de SonarQube.

```
27      coverageReporter: {
28        dir: require('path').join(__dirname, './coverage'),
29        subdir: '.',
30        reporters: [
31          { type: 'html', subdir: 'html-report' },
32          { type: 'lcov', subdir: 'lcov-report' }
33        ]
34      },
```

Figura 5.23: *Líneas de código del archivo karma.conf.js.*

La salida de la ejecución de esta etapa es la que se puede apreciar en la Figura 5.24.

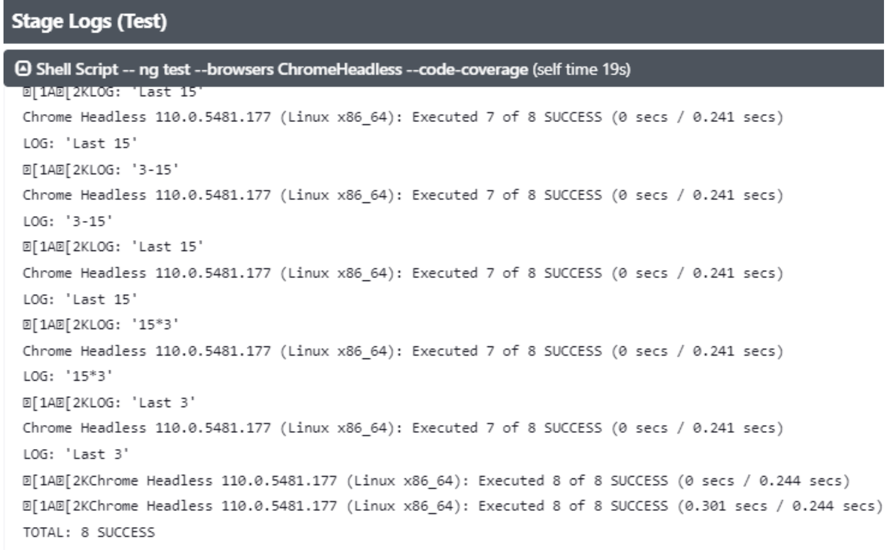


Figura 5.24: *Logs de la etapa "Test".*

5.4.6. Análisis de la calidad del código fuente con SonarQube

Para integrar SonarQube con Jenkins, se necesita de la instalación del *plugin* ***SonarQube Scanner***[4], su configuración en Jenkins y la creación de un *webhook* en la plataforma SonarQube.

5.4.6.1. Instalación del plugin y configuración en Jenkins

De acuerdo a la documentación oficial [SonarSource, 2023], el único requisito previo para ejecutar SonarQube es tener Java, OpenJDK u Oracle JRE. Por esta razón es que fue indispensable configurar en la sección de ***Global Tool Configuration***, que forma parte de **Administrar Jenkins**, la instalación de JDK.

Según se ilustra en la Figura 5.25, como nombre se asignó `openjdk-11`, y se indicó la dirección web del archivo ejecutable a descargar, junto con el directorio en donde se debe extraer.

[4]https://plugins.jenkins.io/sonar/

Panel de Control > Administrar Jenkins > Global Tool Configuration

JDK

instalaciones de JDK

Listado de instalaciones de JDK en este sistema

Añadir JDK

JDK
Nombre

openjdk-11

Instalar automáticamente ?

Extraer *.zip/*.tar.gz ?

Direccion web del archivo ejecutable ?

https://download.java.net/java/GA/jdk11/13/GPL/openjdk-11.0.1_linux-x64_bin.tar.gz

Directorio donde extraer el archivo ?

jdk-11.0.1

Avanzado...

Añadir un instalador

Save Apply

Figura 5.25: *Configuración de la instalación de JDK.*

Con respecto a la instalación del *plugin* ***SonarQube Scanner***, este es un paso simple. Dentro del panel de control de Jenkins, en la sección **Administrar Jenkins**, se seleccionó el **Gestor de *plugins***. Una vez aquí, de la misma manera que ilustra la Figura 5.26, se buscó en los *plugins* disponibles (***Available plugins***) el nombre del componente deseado, se tildó el casillero y se presionó el botón de la parte inferior para comenzar con su instalación.

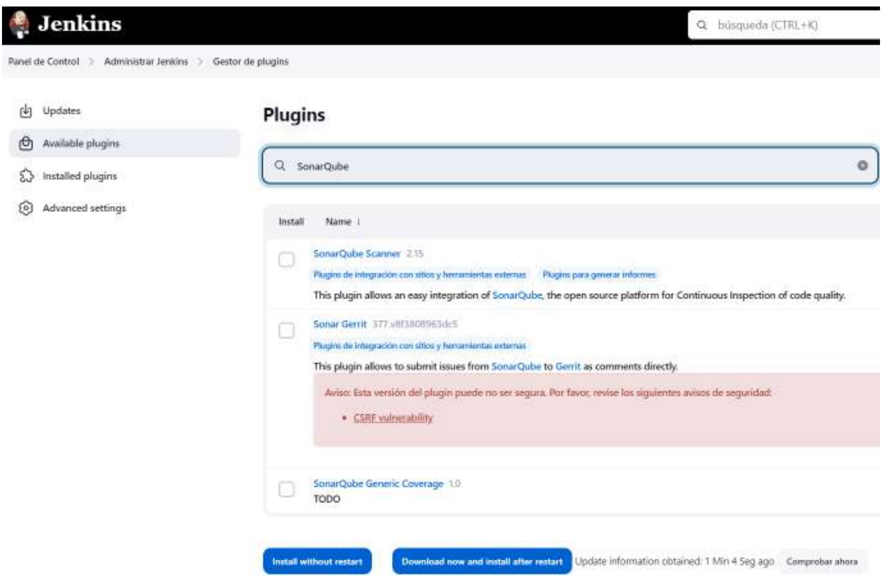


Figura 5.26: *Búsqueda del plugin SonarQube Scanner dentro de los disponibles en Jenkins.*

Tan pronto se obtuvo el componente, en la sección de ***Global Tool Configuration***, que también forma parte de **Administrar Jenkins**, se habilitó la herramienta ***SonarQube Scanner***, lo cual permite configurar el escáner a utilizar. Para la *pipeline* desarrollada, según se puede observar en la Figura 5.27, se configuró como nombre de escáner `sonar-scanner`, y se seleccionó la versión `4.7.0.2747` para que se instale automáticamente.

A partir del momento en que se salva esta configuración, ya se pudo utilizar el escaneador de SonarQube desde las *pipelines*.

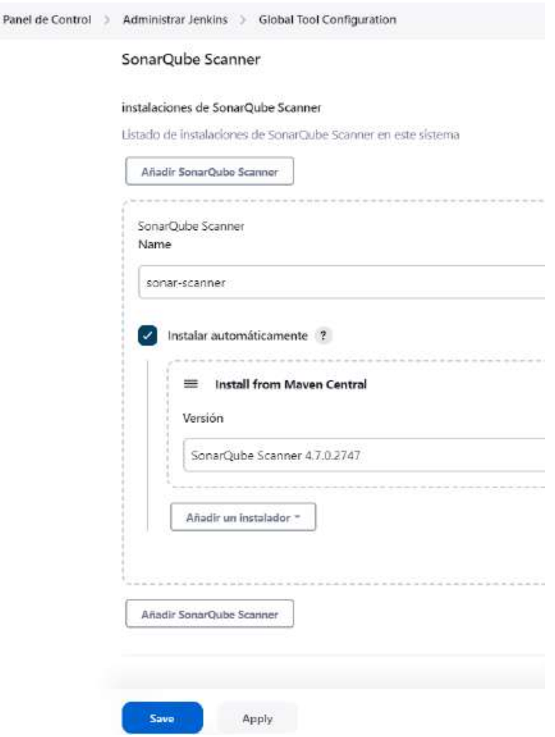


Figura 5.27: *Configuración de SonarQube Scanner en Global Tools Configuration de Jenkins.*

5.4.6.2. Configuración en SonarQube y archivo properties

Por otro lado, en la plataforma de SonarQube hace falta crear un *webhook* que enlace a esta herramienta con Jenkins. Para esto, hay que dirigirse a la pestaña de ***Administration***, dentro de ésta, al apartado de ***Configuration*** y elegir ***Webhooks***, tal como muestra la Figura 5.28.

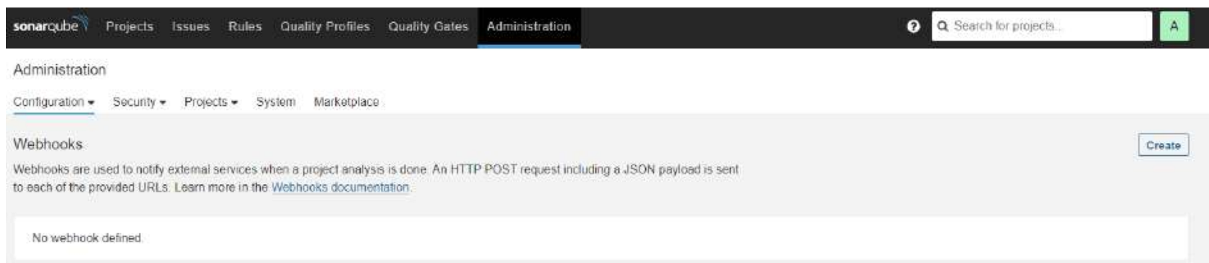


Figura 5.28: *Creación de webhook en SonarQube.*

Tras cumplir este paso, se clickeó en el botón ***Create*** y se completaron los campos obligatorios del formulario que se visualizó (ver Figura 5.29). El *webhook* fue llamado `jenkins-webhook`, y se colocó

la dirección por la cual se ingresa a Jenkins a través del navegador web (IP pública de la VM y puerto de exposición de Jenkins).

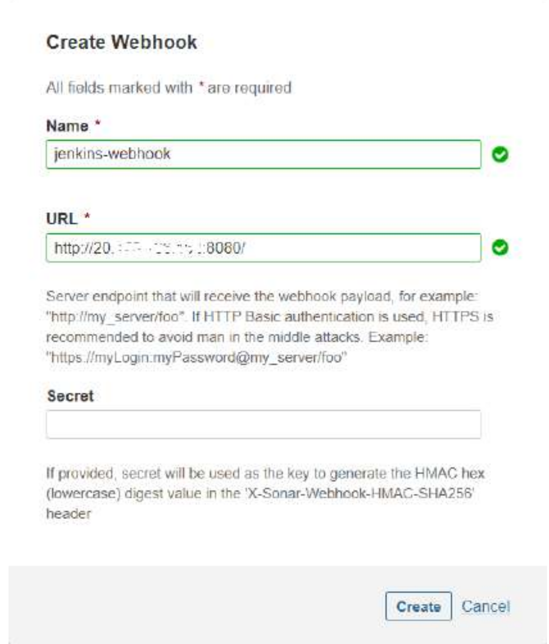


Figura 5.29: *Configuración de webhook en SonarQube.*

A partir de este momento, la comunicación entre Jenkins y SonarQube ha sido establecida, sin embargo, hace falta indicar las credenciales con las cuales se va a acceder para realizar el análisis y la clave del proyecto que se va a generar. Estos datos pueden ser enviados, a la hora de ejecutar el comando del escáner en forma de argumentos. Otra opción es crear un archivo de tipo `.properties` que brinde todos estos detalles, la cual es una forma muy ordenada de hacerlo. En la raíz del repositorio se almacenó un archivo llamado `sonar-project.properties`, en el que se indicó la clave de proyecto, la cual debe ser única por cada instancia de SonarQube, el nombre de usuario y su contraseña.

Como propiedades opcionales se añadió el nombre del proyecto, el número de versión variabilizado (lo que significa que será reemplazado en la ejecución de la *pipeline* por un valor dado), además del lenguaje y archivos a analizar, excluyendo aquel que desarrolla los propios tests unitarios (`app.component.spec.ts`), y la ruta hacia donde se genera el reporte resultante de la ejecución de los mismos.

El contenido del archivo `sonar-project.properties`, es el siguiente:

```
# --- required properties ---
sonar.projectKey = calculator-angular
sonar.login = admin
sonar.password = ******

# --- optional properties ---
sonar.projectName = Calculadora_Angular
sonar.projectVersion = %BUILD_NUMBER%
sonar.sources = src
sonar.exclusions = src/app/app.component.spec.ts
sonar.language = ts
```

```
sonar.typescript.lcov.reportPaths =
     coverage/lcov-report/lcov.info
```

5.4.6.3. Añadiendo SonarQube en la pipeline

Tras efectuar las configuraciones anteriores, se incorporó a la *pipeline* la etapa correspondiente al análisis del código fuente con el escáner de SonarQube, llamada ***SonarQube Analysis***, (ver Figura 5.30). Para esto, fue necesario crear variables de entorno que importen las herramientas instaladas previamente, variables que solo están disponibles dentro de este `stage`.

Luego se creó el `step` que por medio del comando `sed`[5] de Linux busca en el archivo `sonar-project.properties` y efectúa el reemplazo de la cadena de texto `%BUILD_NUMBER%`, por el número de ejecución que almacena la variable de entorno de Jenkins `BUILD_NUMBER`. El propósito de este reemplazo es que la ejecución del análisis del proyecto en SonarQube se encuentre versionada con el número de ejecución de la *pipeline*, para llevar un mejor seguimiento del mismo. De esta manera, se sabe fácilmente qué ejecución está relacionada con cada análisis.

En segundo lugar, también se aplica un paso de tipo Shell script, dónde se indica que se ejecute el comando `sonar-scanner`. No se debe brindar más información puesto que, por defecto, va a tomar el archivo `.properties` alojado en el directorio raíz. Cabe aclarar, que la función `tool`, se utiliza para instalar y obtener la ruta donde fue instalada la herramienta. Por esta razón, es que en el caso de `sonarHome` se debe completar la ruta para llegar al comando a ejecutar. Mientras que la instalación en la variable `JAVA_HOME`, solo es requerida como prerrequisito para que el escáner funcione correctamente.

```
stage('SonarQube Analysis') {
   environment {
     sonarHome = tool 'sonar-scanner'
     JAVA_HOME = tool 'openjdk-11'
   }
   steps {
     sh "sed -i 's/%BUILD_NUMBER%/${BUILD_NUMBER}/g'
       sonar-project.properties"
     sh "${sonarHome}/bin/sonar-scanner"
   }
 }
```

Declarative: Checkout SCM	Declarative: Agent Setup	Install	Build	Test	SonarQube Analysis
1s	1min 48s	6s	11s	16s	31s
868ms	5s	2s	11s	11s	29s

Figura 5.30: *Visualización en la interfaz de Jenkins de las etapas ejecutadas hasta el momento.*

[5] https://www.geeksforgeeks.org/sed-command-in-linux-unix-with-examples/

Una vez la etapa fue ejecutada, en los *logs* (ver Figura 5.31) se puede apreciar su estado *SUCCESS* (exitoso), y en la plataforma de SonarQube, se encuentra el proyecto creado con los resultados del análisis, lo cual se detalla en el Anexo A5.

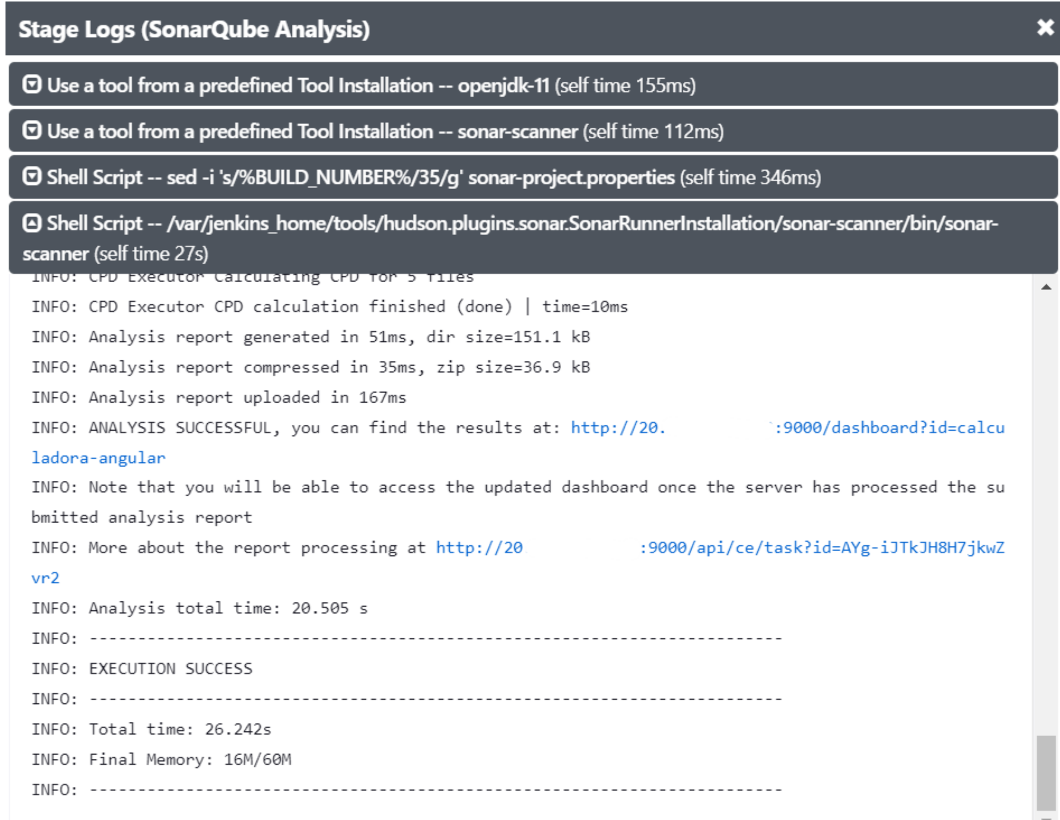


Figura 5.31: *Logs de la etapa "SonarQube Analysis".*

5.4.7. Quality gate

Tal como se explicó en la sección 2.7.1.1, un ***quality gate*** es un control de calidad que define un conjunto de condiciones a partir de las cuales se miden los proyectos. En este análisis se optó por dejar la configuración predeterminada que trae consigo la herramienta, dado que ha sido establecida por expertos en la materia [SonarSource, 2023].

De acuerdo a la Figura 5.32, para obtener un código de alta calidad se necesita una cobertura de código mayor a un 80 %, la cantidad de líneas duplicadas no debe sobrepasar el 3 %, la cantidad de *security hotspots* revisados no debe ser menor a 100 %, y el grado de mantenibilidad, de confiabilidad y de seguridad no puede ser menor a A.

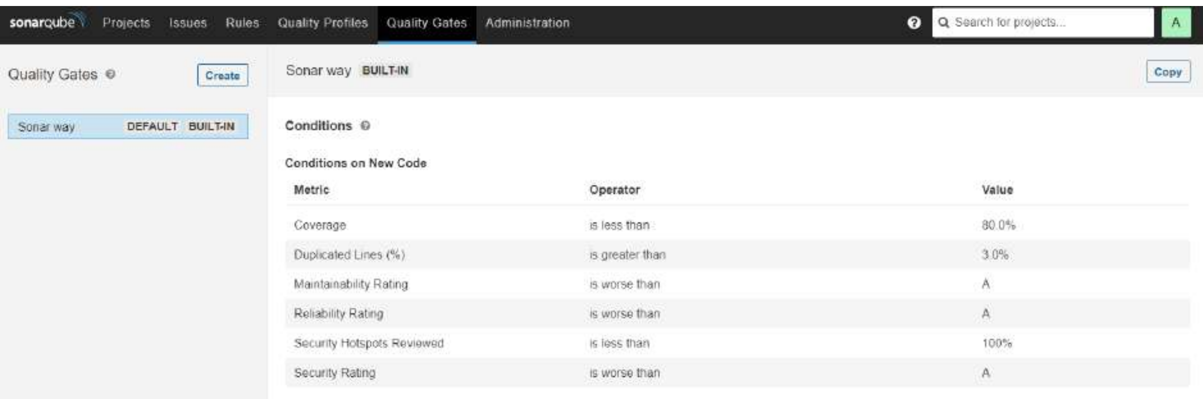


Figura 5.32: *Configuración de Quality Gate por defecto.*

Declarative: Checkout SCM	Declarative: Agent Setup	Install	Build	Test	SonarQube Analysis	Quality Gate
1s	1min 48s	6s	11s	16s	31s	290ms
868ms	5s	2s	11s	11s	29s	344ms

Figura 5.33: *Pipeline de Jenkins con la etapa de Quality Gate.*

En el código del archivo `Jenkinsfile` se creó, a continuación de la etapa anterior, un nuevo `stage` llamado ***Quality gate*** (ver Figura 5.33), al que se le incorporaron dos pasos. El primero de ellos espera a la ejecución del análisis, si este cumple con todos los criterios, su calidad se considera aceptable y el valor del `waitForQualityGate` será `true`, dando lugar a un estado exitoso. En caso contrario, culminará con la ejecución de la *pipeline* resultando con un estado fallido, ya que no se ha cumplido alguna de las reglas, alertando así a los desarrolladores para que se mejore la calidad del producto.

Si la calidad del proyecto es aprobada, se ejecutará el siguiente paso que es de tipo `echo`, por lo que solo indica mediante texto en consola que el análisis del *quality gate* ya ha finalizado.

```
stage('Quality Gate') {
   steps {
     waitForQualityGate true
     echo '--- QualityGate Passed ---'
   }
 }
```

Esta etapa es importante para garantizar que el código sea de alta calidad y cumpla con los estándares establecidos por una organización o equipo de proyecto. En la Figura 5.34, se observa el estado *SUCCESS* de la tarea generada en el análisis de SonarQube previamente realizado.

Figura 5.34: *Logs de la etapa "Quality Gate".*

5.4.8. Creación de imagen Docker

Dado que la aplicación ya ha sido compilada, testeada y la calidad de su código fuente ha sido comprobada, se añadió a la *pipeline* en sexto lugar, la creación de la imagen de Docker, la cual transforma a la aplicación en una aplicación contenerizada (ver Figura 5.35).

Declarative: Checkout SCM	Declarative: Agent Setup	Install	Build	Test	SonarQube Analysis	Quality Gate	Build Docker Image
1s	1min 48s	6s	11s	16s	31s	290ms	16s
868ms	5s	2s	11s	11s	29s	344ms	16s

Figura 5.35: *Visualización gráfica de las etapas ejecutadas, incluyendo la construcción de la imagen de Docker.*

Para esto, en la raíz del repositorio se colocó un archivo de tipo `Dockerfile` con el nombre `Dockerfile.app`, para hacer una distinción con el archivo que se utiliza para el agente de la *pipeline* (`Dockerfile`). Este contiene solo dos líneas:

```
FROM nginx:1.17.1-alpine

COPY dist/app-angular /usr/share/nginx/html
```

La primera de ellas, como todo comienzo de `Dockerfile`, indica la imagen que se utiliza como base. Para contenerizar (*dockerizar*) una aplicación hace falta que se utilice la imagen de un servidor web, por lo que para este trabajo se utilizó nginx[6], con una versión de la distribución Alpine Linux, dado que al ser una distribución minimalista las imágenes pesan mucho menos, a comparación de otras. Y como segunda y última línea, se determinó la copia de los archivos generados en el `stage` *Build*, hacia el directorio `/usr/share/nginx/html` tal como lo indica la documentación oficial de la imagen[7].

[6] https://www.nginx.com/
[7] https://hub.docker.com/_/nginx

El mismo método que se aplicó para llevar a cabo la instalación de Java JDK y el escáner de SonarQube, fue el que se empleó para añadir Docker a los `stages` que lo requerían. Instalando ante todo el *plugin* correspondiente, denominado *Docker Pipeline*[8]. Completando `docker` como el nombre de la herramienta, tildando la opción de instalación automática e indicando como versión la última disponible en la sección ***Global Tool Configuration***, fue suficiente para obtener su interfaz de línea de comandos habilitada en Jenkins (ver Figura 5.36).

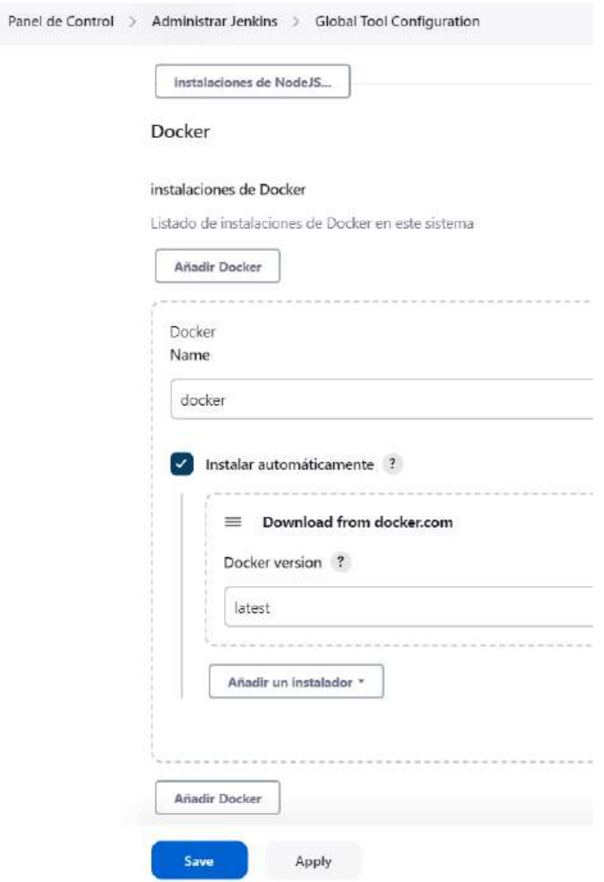


Figura 5.36: *Configuración de la instalación de Docker en Jenkins.*

El `stage` para construir la imagen de Docker, llamado ***Build Docker Image***, define a `dockerHome` como variable de entorno a nivel de etapa, cuyo valor es la ruta hacia el directorio donde se instaló dicha herramienta. Solo contiene un paso que indica la ejecución vía Shell script del comando de la CLI de Docker: `docker build` (para más información dirigirse a la Sección 2.3.1.1).

Lo que realiza esta línea de comandos es construir una imagen, de acuerdo a las instrucciones que contiene el archivo `Dockerfile.app`, que se encuentra en el directorio raíz (por eso se añade un punto al final, para indicar que es el directorio actual), y la etiqueta bajo el nombre de `valen97/calculadora-angular` junto con el número de ejecución de la *pipeline* otorgado por la variable de entorno `BUILD_NUMBER`. Tal como se mencionó en la explicación de la etapa ***SonarQube Analysis***, versionando es la forma más sencilla, en este caso, de poder relacionar la ejecución de la integración continua con la imagen creada. En la Figura 5.37 que muestra los *logs* del `stage` presente, se pueden observar los pasos que realiza para la construcción de la misma.

```
stage('Build Docker Image'){
```

[8]https://plugins.jenkins.io/docker-workflow/

```
    environment {
      dockerHome = tool 'docker'
    }
    steps{
      sh "${dockerHome}/bin/docker build -f Dockerfile.app
          -t valen97/calculadora-angular:${BUILD_NUMBER} ."
    }
  }
```

Stage Logs (Build Docker Image)

Use a tool from a predefined Tool Installation -- docker (self time 513ms)

Shell Script -- /var/jenkins_home/tools/org.jenkinsci.plugins.docker.commons.tools.DockerTool/docker/bin/docker build -f Dockerfile.app -t valen97/calculadora-angular:35 . (self time 15s)

```
+ /var/jenkins_home/tools/org.jenkinsci.plugins.docker.commons.tools.DockerTool/docker/bin/docker build
-f Dockerfile.app -t valen97/calculadora-angular:35 .
Sending build context to Docker daemon  290.9MB

Step 1/2 : FROM nginx:1.17.1-alpine
 ---> ea1193fd3dde
Step 2/2 : COPY dist/app-angular /usr/share/nginx/html
 ---> c0bda2c0ac77
Successfully built c0bda2c0ac77
Successfully tagged valen97/calculadora-angular:35
```

Figura 5.37: *Logs de la etapa "Build Docker Image".*

5.5. Creación de manifiestos de Kubernetes

El despliegue de una aplicación contenerizada en un clúster de Kubernetes implica el desarrollo de un conjunto de manifiestos (concepto desarrollado en Sección 2.4.1.2) que permitan cumplir con tal propósito. Para este caso, solo hizo falta que se construyeran cinco de ellos, los cuales son abordados en las siguientes subsecciones.

5.5.1. Namespace

Dado que se recurrió a un recurso de Azure utilizado en el ámbito laboral, no se pudo ignorar el hecho de que el clúster de Kubernetes era recurrido por varias personas a la vez, por lo cual se determinó crear un `namespace` (espacio de trabajo) que permitiera mantener el orden y, así, no afectar a los demás desarrollos.

Al igual que todo manifiesto posee la versión de API a utilizar, el tipo de recurso que se está declarando y su nombre, `calculadora`. Es uno de los que cuenta con menor cantidad de líneas de código.

```
apiVersion: v1
kind: Namespace
metadata:
  name: calculadora
```

5.5.2. Service

Se declaró un service para exponer la aplicación a nivel de clúster a través del puerto 80, siendo este de tipo ClusterIP (definido en Sección 2.4.1.2). Su nombre es calculadora-service, y se desplegará dentro del namespace calculadora, exponiendo todos los *Pods* que contengan declarado el selector calculadora.

```
apiVersion: v1
kind: Service
metadata:
  name: calculadora-service
  namespace: calculadora
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    app: calculadora
```

5.5.3. Deployment

Es el objeto más importante puesto que es el responsable de administrar el estado deseado de la aplicación, creando, actualizando y eliminando *Pods* según sea requerido.

Este manifiesto de tipo Deployment fue llamado calculadora-deploy, y se configuró para ser desplegado dentro del namespace calculadora. Mantiene dos réplicas del *Pod* (para más detalles consultar Anexo A6), cuyo nombre de contenedor es calculadora y utiliza la imagen valen97/calculadora-angular con el número de la versión variabilizada por el string %BUILD_NUMBER%, que se reemplaza por el número de ejecución de la *pipeline* de Jenkins. Este contenedor se expone en el puerto 80.

La etiqueta selector matchLabels app indica que este deployment debe gestionar los *Pods* que se encuentran bajo este mismo nombre, por esta razón es que en template metadata labels app, el nombre calculadora se repite, ya que en esta instancia se está declarando el label que ocupa el contenedor calculadora.

```
apiVersion : apps/v1
kind: Deployment
metadata:
  name: calculadora-deploy
  namespace: calculadora
spec:
  replicas: 2
  selector:
    matchLabels:
      app: calculadora
  template:
    metadata:
      labels:
```

```
        app: calculadora
    spec:
      containers:
        - name: calculadora
          image: valen97/calculadora-angular:%BUILD_NUMBER%
          ports:
          - containerPort: 80
```

5.5.4. Ingress

Tal como se comentó en la Sección 2.4.1.2, el uso del objeto `Ingress` es necesario para definir un conjunto de reglas con el fin de enrutar el tráfico externo hacia el servicio correcto de Kubernetes [Kubernetes, 2023]. Lo recomendable es utilizar el tipo de ruta `Prefix` (prefijo), ya que permite una mayor flexibilidad en el manejo del tráfico para diferentes rutas en un solo dominio. Además, se le pueden adicionar rutas sin necesidad de modificar las reglas de ingreso, siempre y cuando estas tengan el mismo prefijo.

En este manifiesto, además de declarar su nombre (`calculadora-ingress`) y tipo (`Ingress`), se definió la regla mediante la cual se enviará tráfico hacia la calculadora, colocando en host la url `calculadora-angular.modelodevops.sociuschile.cl`, en `path` solo una barra lateral (`/`), ya que es el prefijo a utilizar. Como `backend` se indicó el nombre y puerto del servicio a publicar, que en esta oportunidad fue `calculadora-service` y `80`, respectivamente.

Al igual que los demás objetos, se configuró para desplegarse dentro del `namespace calculadora`.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: calculadora-ingress
  namespace: calculadora
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: "calculadora-angular.modelodevops.sociuschile.cl"
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: calculadora-service
            port:
              number: 80
```

5.5.5. Kustomization

Este manifiesto facilita el despliegue de los diferentes archivos en el clúster de Kubernetes [Kubernetes, 2023]. Se indicó bajo la línea de `resources`, todos aquellos recursos que se requiere que sean publicados (`namespace`, `service`, `deployment` e `ingress`).

Particularmente, como todos los archivos referidos a Kubernetes se encuentran almacenados en un mismo directorio, no fue necesario declarar la ruta completa de los mismos, solo con indicar su nombre fue suficiente.

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - namespace.yml
  - service.yml
  - deployment.yml
  - ingress.yml
```

5.6. Desarrollo de tareas de despliegue continuo

Finalmente, a la *pipeline* se le añadieron dos etapas más para realizar la publicación de la imagen en el registro de contenedores público de Docker Hub, y otro para que realice el despliegue en el clúster de Azure Kubernetes Service.

5.6.1. Publicación de la imagen en Docker Hub

Para realizar la publicación de una imagen en el registro de contenedores Docker Hub (más información en la Sección 2.3.1.1), es necesario contar con una cuenta en dicha plataforma, puesto que cada usuario puede administrar las imágenes que publica, sus etiquetas, los colaboradores, configurar su visibilidad y más. Por lo tanto, se añadió con anterioridad la credencial del usuario utilizado que permitió llevar a cabo tal actividad.

Sumar una credencial a Jenkins es relativamente fácil. Para esto se ingresó a ***Global credentials (unrestricted)***, donde se añadió una nueva credencial de tipo ***Username with password***, con alcance global. En este formulario, que se puede apreciar en la Figura 5.38, se completaron datos como el nombre de usuario que se utiliza para ingresar a Docker Hub, su contraseña, un ID para identificar esta credencial y una breve descripción significativa para su utilización.

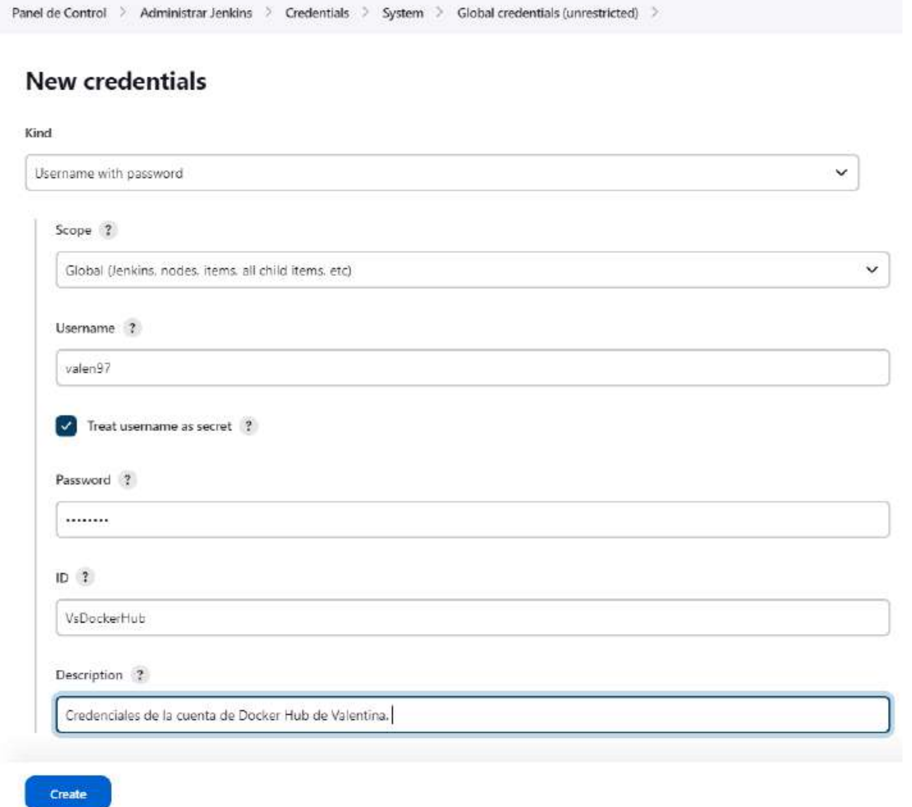


Figura 5.38: *Creación de credenciales en Jenkins.*

Concluida tal configuración, se construyó la etapa de la *pipeline* que automatiza el proceso de publicación, nombrada como ***Publish Docker Image***. Análogamente a lo realizado en la Sección 5.4.8, se declaró como variable de entorno la herramienta `docker`, para poder hacer uso de este comando en el `stage`. Asimismo, se declararon las credenciales de la cuenta que se utilizó en Docker Hub, configuradas a través de la interfaz de Jenkins, referenciándola a través de su ID, `VsDockerHub`.

Dentro de los pasos que componen esta etapa, se realizó el inicio de sesión en Docker Hub, luego la publicación de la imagen por medio del comando `docker push` indicando su nombre y *tag*, su respectiva eliminación utilizando el comando `docker rmi`, seguido del nombre y *tag* de la imagen, debido a que las imágenes se construyen en la VM que aloja Jenkins y se podrían acumular con el correr del tiempo, entorpeciendo el rendimiento de la misma. Por eso, una vez publicada se procede a eliminarla y, por último, el cierre de sesión de la cuenta de Docker Hub.

Un dato no menor, es que al declarar la variable de entorno con la función `credentials()`, automáticamente permite el uso del usuario (`<nombre_variable>_USR`) y de la contraseña (`<nombre_variable>_PSW`) sin tener que generar una variable para cada dato. La idea de esta función, para la persona que está llevando a cabo el trabajo de automatización, es que no sea necesario tener conocimiento de datos sensibles como lo son las contraseñas, por lo tanto, otra persona puede cargar la credencial en Jenkins y solo informar el ID por el cual debe hacer referencia en la *pipeline*.

```
stage('Publish Docker Image'){
  environment {
    dockerHome = tool 'docker'
    dockerHub = credentials('VsDockerHub')
  }
  steps {
```

```
            sh "${dockerHome}/bin/docker login -u $dockerHub_USR
                -p $dockerHub_PSW"
            sh "${dockerHome}/bin/docker push
                valen97/calculadora-angular:${BUILD_NUMBER}"
            sh "${dockerHome}/bin/docker rmi
                valen97/calculadora-angular:${BUILD_NUMBER}"
            sh "${dockerHome}/bin/docker logout"
        }
    }
```

Declarative: Checkout SCM	Declarative: Agent Setup	Install	Build	Test	SonarQube Analysis	Quality Gate	Build Docker Image	Publish Docker Image
1s	1min 48s	6s	11s	16s	31s	290ms	16s	9s
868ms	5s	2s	11s	11s	29s	344ms	16s	10s

Figura 5.39: *Etapa de publicación de la imagen de Docker ejecutada por Jenkins.*

Una vez concluida la ejecución de la etapa ***Publish Docker Image*** (ver Figura 5.39), como salida de la misma surgen los *logs* de cada una de las tareas de tipo Shell script que la componen y la instalación de la herramienta `docker`, tal cual se aprecia en la Figura 5.40.

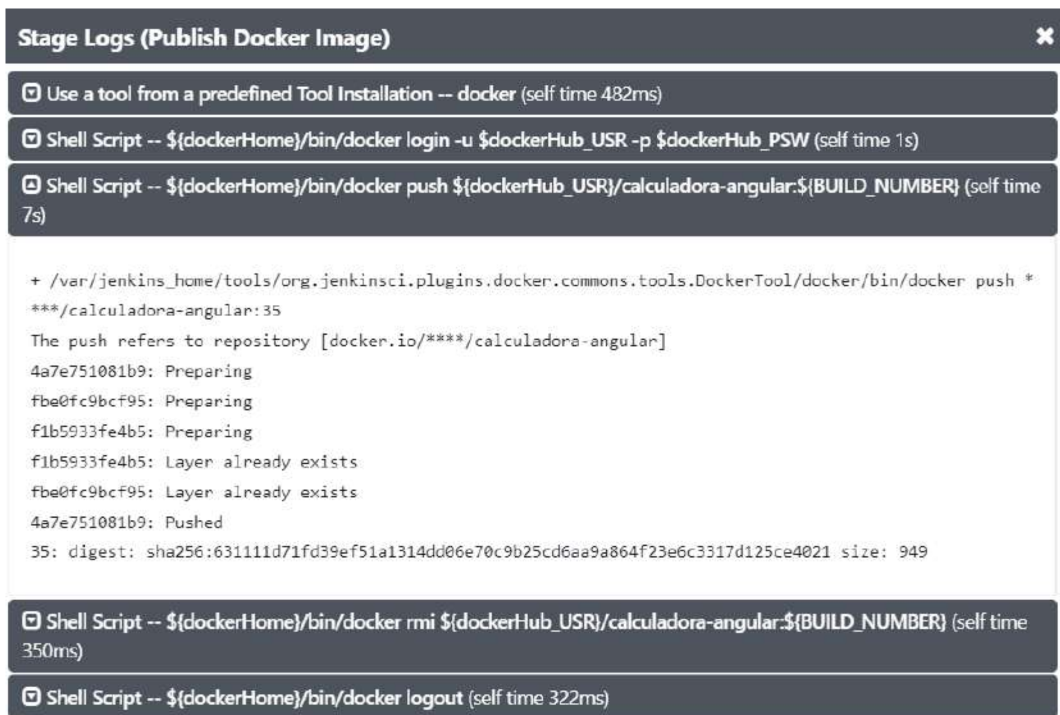


Figura 5.40: *Logs de la etapa "Publish Docker Image".*

Al mismo tiempo, se comprobó si la publicación fue realizada con éxito, ingresando en la plataforma Docker Hub, justo como se muestra en la Figura 5.41.

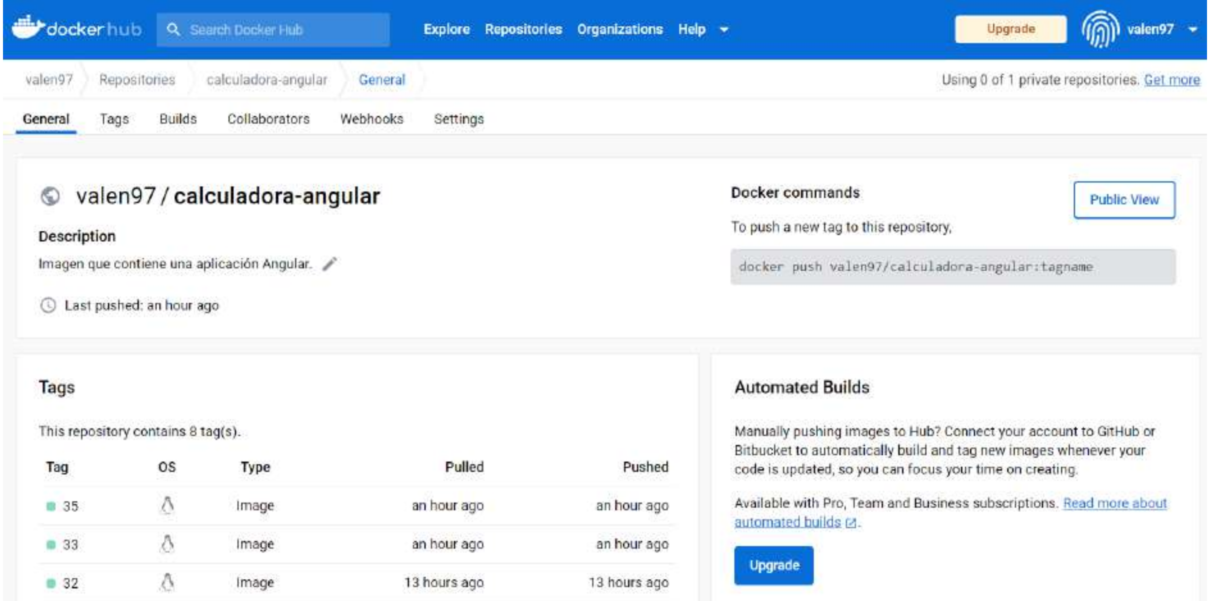


Figura 5.41: *Imagen Docker de la aplicación Angular publicada en Docker Hub.*

5.6.2. Despliegue de manifiestos en clúster de Azure Kubernetes Service

Conectarse a un recurso de Azure, implica utilizar datos sensibles de conexión. Por esta razón, se creó una credencial de tipo **Azure Service Principal** (ver Figura 5.42).

New credentials

Kind

Username with password

Username with password
Azure Key Vault Secret
Azure Key Vault Secret Certificate
Azure Managed Identity
Azure Service Principal
Azure Storage
GitHub App
Kubernetes configuration (kubeconfig)
OpenShift Username and Password
SSH Username with private key
Secret file
Secret text
X.509 Client Certificate
Certificate

Figura 5.42: *Listado de los tipos de credenciales que se pueden crear en Jenkins.*

Un **Azure Service Principal**, es una identidad de seguridad que utilizan las aplicaciones o los servicios para acceder a los recursos de Azure. Su esencia es proporcionar una forma segura y eficiente para que las aplicaciones y los servicios se autentiquen y accedan a los recursos de Azure, sin necesidad de depender de las credenciales de los usuarios, pues el mismo cuenta con sus propias credenciales y permisos.

Por razones de seguridad, no se muestran todos los campos que fueron completados para esta credencial, no obstante, se puede notar en la Figura 5.43 cuál fue el ID ingresado (`azuredevops_dev`), su descripción y, también, que su conexión fue verificada. A estos datos se adiciona el Tenant ID, Subscription ID, Service Principal ID (o Application ID) y la clave del Service Principal.

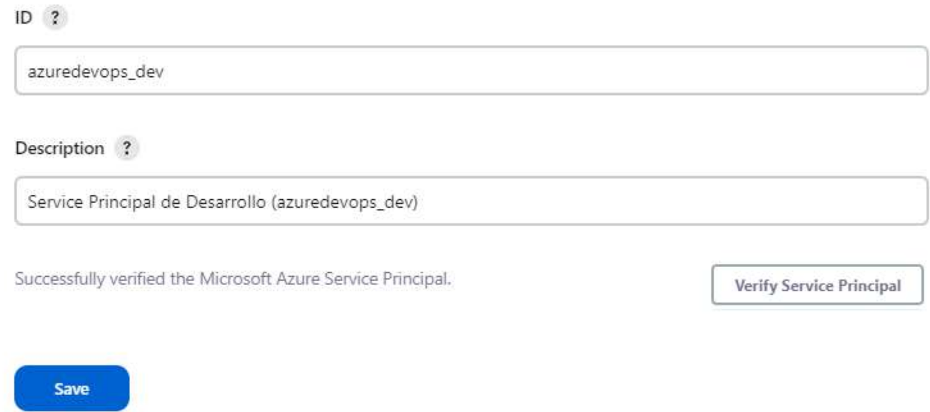


Figura 5.43: *Creación de credencial para Azure Service Principal.*

La tarea de despliegue fue creada en una *pipeline* diferente. Esto se realizó con el fin de mantener un control sobre el mismo, ya que en caso de necesitar volver a desplegar una imagen en específico,

solo haría falta ejecutar esta *pipeline*, puesto que la imagen ya fue creada previamente pasando por los controles de calidad impuestos.

Para esto, tal como se hizo en el inicio de la Sección 5.4, se creó una nueva tarea de tipo ***Pipeline*** llamada `calculadora-angular-deploy`, se indicó el mismo ***GitHub Project*** y, como se puede ver en la Figura 5.44, se especificó en la sección ***Pipeline*** la rama a utilizar (`dev`) y el ***Script Path*** que en esta ocasión fue `Jenkinsfile-deploy` (ubicado en la raíz del repositorio).

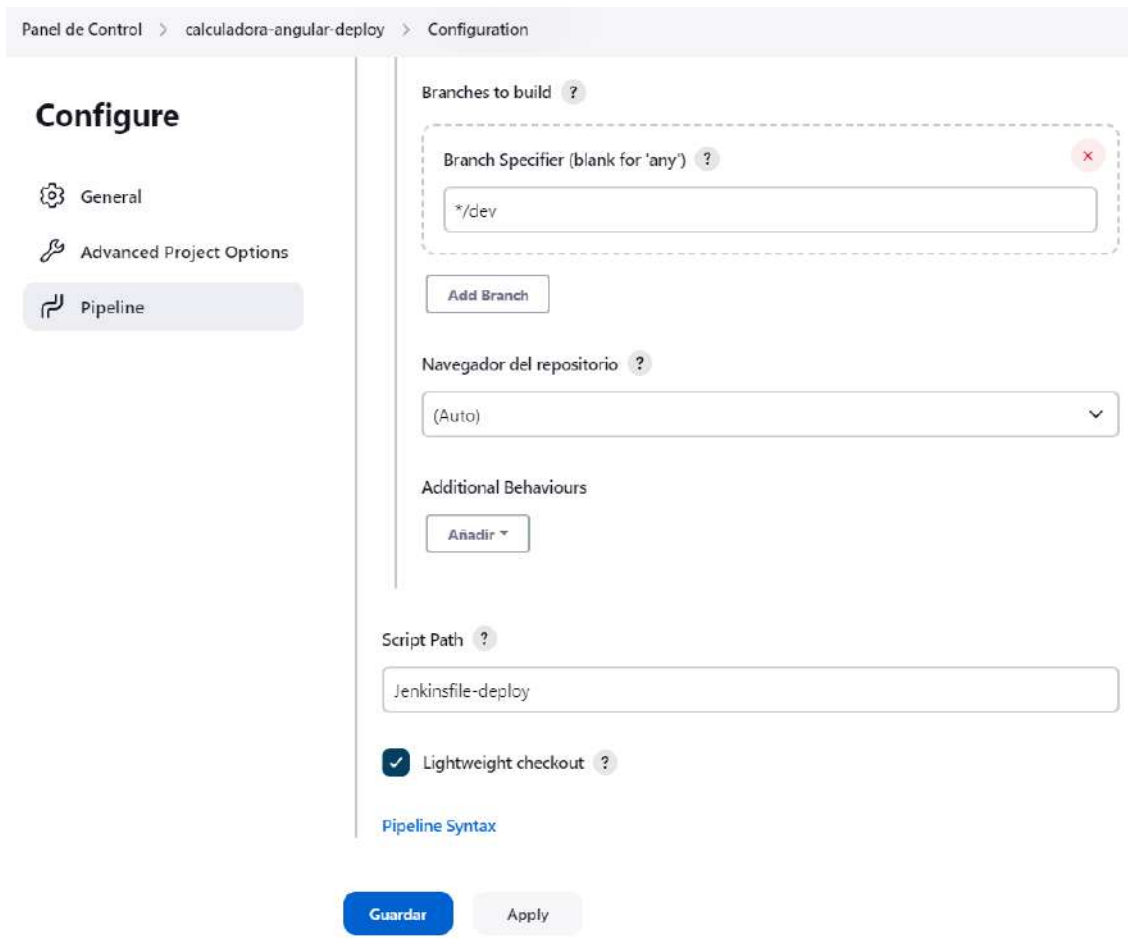


Figura 5.44: *Configuración del Jenkinsfile en la pipeline calculadora-angular-deploy.*

Cabe destacar que para esta *pipeline* no fue necesario activar las opciones que habilitan el *trigger* desde GitHub, ya que su inicio fue indicado desde la ejecución de la *pipeline* `calculadora-angular` o, en su defecto, puede ser lanzada manualmente (para más detalles ver Anexo A7).

En el archivo `Jenkinsfile` se añadieron dentro de la sección `post success`, las siguientes líneas de código, en las cuales se declara que, una vez la *pipeline* `calculadora-angular` haya culminado la ejecución exitosamente, se lanzará la *pipeline* `calculadora-angular-deploy` enviándole como parámetro el valor de la variable de entorno `BUILD_NUMBER`, para que se corresponda con la imagen de Docker creada y publicada en las etapas previas.

La porción de código detallada a continuación, fue insertada antes de las líneas que envían la notificación por correo electrónico, debido a que las tareas se ejecutan secuencialmente y, por lo tanto, se necesita lanzar el despliegue antes de informar el estado de compilación total.

```
build(job: 'calculadora-angular-deploy',
```

```
  parameters: [
    string(name: 'BUILD_NUMBER', value: "${BUILD_NUMBER}")
  ])
```

Al igual que `calculadora-angular`, en `calculadora-angular-deploy` se declara como agente el `Dockerfile` que se encuentra almacenado en el repositorio, pero aquí no hizo falta enviarle argumentos debido a que solo eran necesarios para ejecutar las pruebas unitarias y el análisis de SonarQube. Para una visualización completa de los archivos `Jenkinsfile` y `Jenkinsfile-deploy` el lector puede dirigirse al Anexo A8.

Esta última etapa fue llamada ***Deploy Dev*** y, como se indicó, contiene el código que realiza el despliegue, donde se importó la credencial mencionada anteriormente utilizando el *plugin Credentials Binding*[9], con el fin de destacar la cantidad de *plugins* que pueden ser aplicados a la hora de construir una *pipeline* en Jenkins.

```
stage('Deploy Dev') {
  steps {
    withCredentials(bindings:
    [azureServicePrincipal('azuredevops_dev')]) {
      sh 'sed -i "s/%BUILD_NUMBER%/${BUILD_NUMBER}/g"
          kubernetes/deployment.yml'
      sh 'az login --service-principal
          -u $AZURE_CLIENT_ID
          -p $AZURE_CLIENT_SECRET
          -t $AZURE_TENANT_ID'
      sh 'az aks get-credentials
          --resource-group $RESOURCE_GROUP
          --name $CLUSTER_NAME'
      sh 'kubectl apply -k kubernetes/.'
    }
  }
}
```

En el primer paso se realiza el mismo reemplazo mencionado en la Sección 5.4.6.3, donde se sustituye el string `%BUILD_NUMBER%` por el contenido recibido en el parámentro `BUILD_NUMBER`, con el propósito de indicar el mismo *tag* de imagen que se creó anteriormente.

La etapa continúa con el inicio de sesión en Azure, llevado a cabo mediante la utilización del comando `az login`, indicando que este va a ser de tipo *Service Principal* y haciendo uso de las variables que habilita el `withCredentials`, que son el ID, la contraseña y el Tenant ID. Una vez registrado en Azure, se ejecuta el comando `az aks get-credentials`, el cual confiere las credenciales de acceso al clúster de Kubernetes, indicado a través del argumento `name` y su grupo de recursos.

Los valores enviados a través de los argumentos del comando `az`, fueron ingresados utilizando parámetros, declarados a nivel *pipeline*, ya que, si se piensa en su reutilización, en caso de querer utilizar el mismo `Jenkinsfile` para otro ambiente o proyecto, solo haría falta cambiar el valor de los parámetros junto con los nombres de las credenciales almacenadas. Para declarar un parámetro en Jenkins se debe indicar el tipo, nombre, valor por defecto y una breve descripción, tal como se muestra a continuación.

[9] https://plugins.jenkins.io/credentials-binding/

```
parameters {
    string(name: 'RESOURCE_GROUP',
           defaultValue: 'SOCIUSRGLAB-RG-MODELODEVOPS-AKS',
           description: 'Grupo de Recursos')
    string(name: 'CLUSTER_NAME',
           defaultValue: 'ModeloDevOps-AKS',
           description: 'Nombre del App Service')
    string(name: 'BUILD_NUMBER',
           defaultValue: '',
           description: 'Versión imagen Docker')
}
```

En última instancia, se corre el comando de Kubernetes `kubectl apply -k kubernetes/.`, este despliega o actualiza los manifiestos indicados dentro del archivo de tipo Kustomization (`-k`) que se encuentra dentro del directorio Kubernetes. En caso de correr el comando y no encontrar diferencias entre los manifiestos y lo publicado en el clúster, `kubectl` no hará nada.

Cuando la etapa ***Publish Docker Image*** finaliza exitosamente, comienzan a ejecutarse los pasos especificados en el bloque de `post`. El cual mostró en sus *logs* el enlace hacia la ejecución de la *pipeline* `calculadora-angular-deploy` y que el paso de enviar el correo electrónico fue llevado a cabo satisfactoriamente, como ilustra la Figura 5.45.

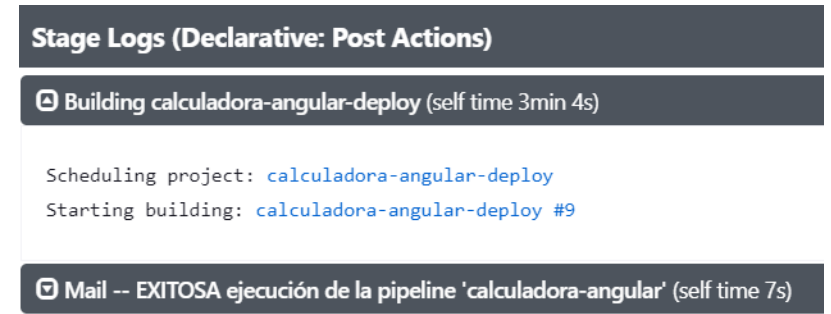


Figura 5.45: *Logs de la etapa declarativa "Post Actions".*

Al hacer click en el enlace, Jenkins redirige hacia la otra *pipeline*, mostrando la siguiente ventana (ver Figura 5.46), en la que se observa que, tal como sucedió con la *pipeline* `calculadora-angular`, se crearon dos etapas de tipo declarativas que realizan el *checkout* del código fuente del repositorio y otra que configura el agente en el cual ejecutar las etapas definidas.

Figura 5.46: Ejecución de la pipeline calculadora-angular-deploy.

Una vez el `stage` ***Deploy Dev*** finaliza su ejecución, se puede apreciar mediante los *logs* (ver Figura 5.47) que cada uno de los manifiestos ha sido creado (`created`), ya que estos no existían en el clúster de Azure Kubernetes Service con anterioridad.

Figura 5.47: Logs de la etapa "Deploy Dev".

Para comprobar la efectividad del proceso, en primer lugar, se chequeó mediante consola aplicando un comando de la CLI de Kubectl que muestra los recursos desplegados en el clúster de Kubernetes. Precisamente, la línea que se puede comprobar en la Figura 5.48, `kubectl get all -n calculadora`, indicando el `namespace` del cuál se quiere obtener información.

```
C:\>%%%%% SE CHEQUEAN LOS RECURSOS DEL NAMESPACE CALCULADORA %%%%%

C:\>kubectl get all -n calculadora
NAME                                   READY   STATUS    RESTARTS   AGE
pod/calculadora-deploy-5c96755674-lpspk   1/1     Running   0          91s
pod/calculadora-deploy-5c96755674-mxp5q   1/1     Running   0          91s

NAME                          TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
service/calculadora-service   ClusterIP   10.0.101.78   <none>        80/TCP    93s

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/calculadora-deploy   2/2     2            2           93s

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/calculadora-deploy-5c96755674   2         2         2       93s
```

Figura 5.48: *Salida de consola del comando kubectl get all en el namespace calculadora.*

Esto detalla todos los recursos desplegados en el `namespace calculadora` y su estado actual, en el caso de la Figura 5.48, ambos *Pods* se encuentran en estado de ejecución (*running*), lo cual abre paso para realizar la segunda comprobación: ingresar a la dirección colocada en el manifiesto del recurso de tipo `ingress`.

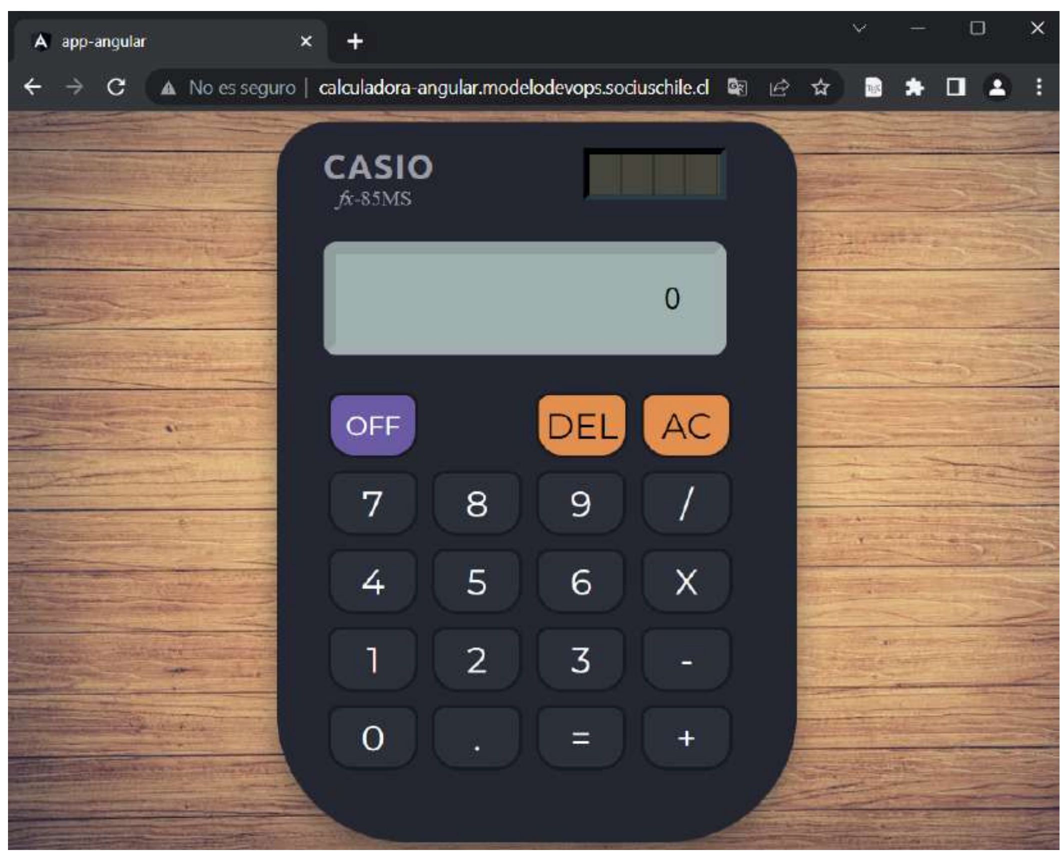


Figura 5.49: *Aplicación desplegada exitosamente en el clúster de Azure Kubernetes Service.*

Efectivamente, se encuentra la aplicación publicada en la web (ver Figura 5.49), demorándose menos de diez minutos en ejecutar toda la *pipeline*, tiempo que puede variar dependiendo de la conexión a Internet.

Como se puede apreciar en la Figura 5.50, la ejecución número 33, tardó poco más de tres minutos en setear el agente, mientras que la ejecución posterior, fue muchísimo más rápida, demorándose cinco

segundos en esa etapa. Esta diferencia se debe a que Jenkins utiliza la caché para almacenar archivos luego de completar tareas específicas, impactando positivamente en el rendimiento.

	Declarative: Checkout SCM	Declarative: Agent Setup	Install	Build	Test	SonarQube Analysis	Quality Gate	Build Docker Image	Publish Docker Image	Declarative: Post Actions
Average stage times: (Average full run time: ~5min 11s)	1s	1min 48s	6s	11s	16s	31s	290ms	16s	9s	1min 19s
#35 May 21 13:37 1 commit	868ms	5s	2s	11s	11s	29s	344ms	16s	10s	3min 12s
#33 May 21 13:29 1 commit	2s	3min 8s	8s	12s	19s	36s	305ms	16s	10s	1min 28s
#32 May 21 01:42 2 commits	1s	3min 37s	15s	13s	28s	38s	301ms	16s	10s	1min 11s
#30 May 20 21:22 2 commits	815ms	4s	2s	11s	13s	27s	240ms	14s	9s	26s
#29 May 20 21:14 6 commits	1s	3min 52s	2s	10s	11s	27s	260ms	15s	9s	17s

Figura 5.50: *Comparación de tiempo de las ejecuciones.*

En la Figura 5.51, se puede observar que la *pipeline* fue lanzada por un evento de *push*, el usuario que lo realizó, detalles del *commit*, además de la hora, fecha y duración de la ejecución.

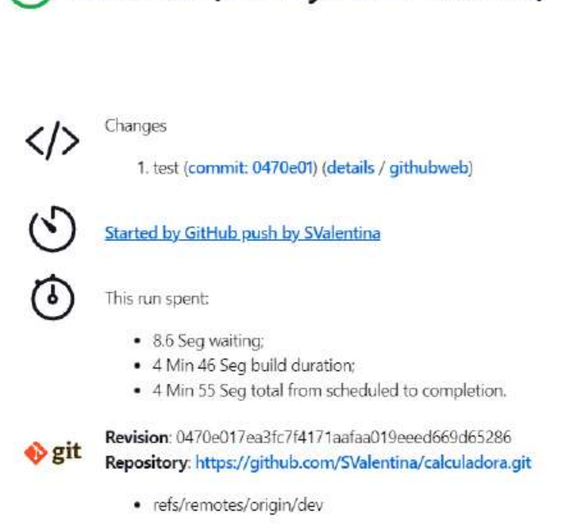


Figura 5.51: *Tiempo total de ejecución para desplegar la aplicación.*

5.7. Conclusión

Para sintetizar lo abordado en este capítulo, se enumeran los puntos claves de la solución propuesta para robustecer la actual metodología de desarrollo empleada en la Facultad de Ingeniería y migrar hacia una solución completamente automatizada.

1. **Compilar la aplicación.**
   Luego de integrar el código en el repositorio, se recomienda volver a compilar la aplicación una vez se ejecuta la *pipeline*, para demostrar que esta funciona según lo esperado con las nuevas líneas incorporadas al código fuente base del desarrollo.

2. **Analizar la calidad del código fuente.**
   No es suficiente con ejecutar solamente tests unitarios, sino que se debe complementar esta actividad inspeccionando la calidad del código fuente, como se demostró integrando la *pipeline* de Jenkins con la herramienta SonarQube.

3. **Migrar hacia una infraestructura en la nube.**
   Actualizando hacia una infraestructura más moderna, se otorga al equipo la capacidad de llevar a cabo la automatización del despliegue de la aplicación, permitiendo aumentar la frecuencia de los mismos y generando rápido *feedback* por parte de los usuarios. Además, trae consigo todas las ventajas que provee un entorno *cloud*, como los bajos costos, la escalabilidad, la flexibilidad y más.

   En el ejemplo desarrollado se utilizó Azure como proveedor de la nube, el cual es uno de los tres mejores del mercado en la actualidad, tal como se mencionó en el Capítulo 3, pero la Facultad de Ingeniería podría optar por la utilización de otro proveedor que la favorezca.

4. **Crear aplicaciones *Cloud Native*.**
   Ligado al punto anterior se encuentra la migración de aplicaciones tradicionales hacia aquellas que son de naturaleza *Cloud Native*, permitiendo su ejecución en un entorno en la nube, como se demostró en este capítulo haciendo uso de plataformas como Docker y Kubernetes.

A partir de estos cambios, el Área de Desarrollo podría beneficiarse con una metodología de desarrollo que automatice las tareas y proporcione calidad a sus sistemas desde el comienzo, detectando tempranamente los errores y previniendo fallas futuras.

6

Conclusiones y Trabajo Futuro

"Si quieres llegar rápido, ve solo.
Si quieres llegar lejos, ve acompañado."

— Proverbio Africano

El sexto y último capítulo expone las conclusiones arribadas luego del desarrollo de este trabajo final. Se detallan además, las propuestas para trabajos futuros que extiendan del presente.

6.1. Conclusión Final

Este trabajo de tesis presentó una propuesta de automatización del proceso de implementación de software para el Área de Desarrollo de la Facultad de Ingeniería, propuesta que implicó explorar los conceptos de *DevOps* y *Cloud Native*. Estas, son tendencias que, año tras año, ganan más popularidad en el ámbito de la tecnología y el desarrollo de software, gracias a los beneficios que ofrece su adopción. A continuación, se exponen las ventajas que conlleva la aplicación de la propuesta planteada.

- La aplicación de las prácticas de integración continua, testeo continuo e inspección continua, permite entregar al cliente un producto de mayor calidad. Además, la ejecución de tests unitarios cada vez que se añade código al repositorio es de gran ayuda, debido a que ante cualquier resultado erróneo, rápidamente el desarrollador será alertado para que pueda modificarlo y volver a subir código que funcione según lo esperado. Sin embargo, es importante compilar la aplicación previamente, ya que pueden surgir diferencias si solo se compila en la estación de trabajo de cada desarrollador y no, cuando se realiza la integración de código en la rama colaborativa.
- El empleo de una herramienta que analice la calidad del código fuente, como lo es SonarQube, permite detectar rápida y precisamente los puntos a mejorar del código, ayudando a mantener la calidad del mismo durante el ciclo de vida del desarrollo. Asimismo, permite que se detecten y aborden los problemas con anticipación, influyendo positivamente en la seguridad, reduciendo la deuda técnica y mejorando la capacidad de mantenimiento general. Todas estas cuestiones convierten al código fuente en legible y estable, características primordiales si el equipo de trabajadores cambia constantemente.
- Combinar estas tres prácticas crea un desarrollo sólido que fomenta la detección temprana de errores, *feedback* continuo gracias a las notificaciones integradas en la *pipeline*, y la mejora en la calidad del producto. Se trabaja con mayor confianza apoyándose en las pruebas e inspecciones que se realizan automáticamente.
- La utilización de Jenkins para la integración continua evidenció un desempeño rápido: solamente se tardó menos de 10 minutos en ejecutar la *pipeline* desarrollada, tiempo que considera desde el *checkout* del código almacenado en el repositorio, hasta el despliegue de la aplicación en el clúster de Kubernetes, lista para ser consumida por el usuario. Esto, da cuenta de la gran agilidad y flexibilidad que provee la combinación de la automatización con recursos en la nube.
- Considerar un entorno *Cloud Native*, simplificó el despliegue, puesto que facilitó su automatización. En el relevamiento de la situación actual del Área de Sistemas, se detectó que se instala manualmente el archivo comprimido en el servidor de destino, lo cual incrementa los posibles errores de despliegue al no poseer un seguimiento de tales acciones. En el caso práctico presentado, se construyó una imagen Docker con los archivos necesarios para su publicación, que permite la portabilidad de la misma y garantiza además, un comportamiento coherente en los diferentes entornos.
- Implementar Kubernetes como orquestador de contenedores, permitió un rápido despliegue de la aplicación contenerizada, dejando atrás la necesidad de utilizar infraestructura física, para proporcionar en su lugar recursos escalables según las necesidades. También, se destaca que al automatizar este proceso, se otorgó la capacidad de reflejar en cuestión de minutos los cambios realizados en la aplicación. Kubernetes garantiza la resiliencia y disponibilidad de los sistemas, debido a que su objetivo es alcanzar y mantener el estado deseado declarado en los manifiestos de los recursos que conforman el clúster.

Automatizar el proceso de implementación de software empleando un enfoque *DevOps* y *Cloud Native*, trae consigo grandes beneficios como se expuso a lo largo de este trabajo. Sin embargo, llevar a cabo este cambio de paradigma puede tornarse complejo, puesto que requiere un cambio de mentalidad y una cultura diferente que fomente la colaboración, la confianza y la mejora continua. Es claro que se necesita invertir tiempo en capacitación y en desarrollar las habilidades fundamentales para sacar un mayor provecho de estos enfoques y así, superar las posibles resistencias al cambio. El compromiso de las personas involucradas y la utilización de las herramientas adecuadas influyen significativamente en la implementación correcta de estos novedosos enfoques.

Por lo expuesto, se hace necesario que nuestro centro educativo se adapte y aproveche su potencial para mantenerse actualizado a una industria del software que evoluciona constantemente.

Para concluir y, referenciando la frase que inicia el capítulo, la colaboración y el trabajo en equipo son muy importantes para lograr el éxito a largo plazo y alcanzar metas que requieren un esfuerzo sostenido, tal como lo propone la filosofía *DevOps*.

En relación al desafío personal que implicó el desarrollo de esta tesis, destaco la posibilidad de incursionar en un tema de suma relevancia en la industria del software, otorgándome las herramientas necesarias para asumir un puesto *DevOps* profesionalmente y poder aplicar la teoría vista en el ámbito real. No solo incrementé y profundicé mis conocimientos relacionados a las áreas de software abordados en el transcurso de la carrera, sino que también desarrollé habilidades investigativas, descubriendo cómo sintetizar información de manera tal que sea comprensible para el lector, sin descuidar el hilo conductor, lo cual en un inicio, fue bastante abrumador. Además, aprendí a organizarme en un mundo en el que, con un solo click, estamos repletos de información.

Este ha sido un proceso largo y exhaustivo, que se complejiza en el día a día si se tienen a cargo otras responsabilidades, como el trabajo, ya que demanda mucho tiempo. Sin embargo, completar el trabajo en tiempo y forma, tal lo estipulado, ha sido muy gratificante y enriquecedor, tanto a nivel académico como a nivel personal.

6.2. Trabajo Futuro

A continuación, se listan algunas de las posibles líneas de trabajo futuro que surgen considerando este trabajo como base de sus desarrollos.

- Realizar la comparación entre distintos servidores de integración continua, trasladando una misma *pipeline* hacia las diferentes herramientas para denotar sus diferencias, ventajas y desventajas.
- Añadir seguridad al clúster de Kubernetes por medio de la encriptación del tráfico de red utilizando *Transport Security Layer* (TLS). Asimismo, se debería utilizar algún administrador de certificados open-source que permita automatizar la gestión y emisión de certificados SSL/TLS dentro del clúster.
- Realizar el monitoreo continuo del clúster de Kubernetes, lo cual puede llevarse a cabo empleando herramientas populares en el mercado como Prometheus[1], Grafana[2] y Grafana Loki[3]. Prometheus monitorea, alerta y recolecta métricas, mientras que Grafana facilita la visualización de

[1]https://prometheus.io/
[2]https://grafana.com/
[3]https://grafana.com/oss/loki/

las mismas a través de la configuración de dashboards, y, por su parte, Grafana Loki provee la capacidad de manejar los logs para su análisis y la resolución de problemas.

- Permitir el despliegue continuo desde el clúster de Kubernetes aplicando los principios de GitOps[4], esto significa que el estado deseado del clúster se almacena en un repositorio de Git y una herramienta de CD para Kubernetes (alguna de estas pueden ser FluxCD[5], ArgoCD[6], JenkinsX[7], entre otros) garantiza que el clúster coincida con ese estado deseado. De esta forma, no se espera a una ejecución de *pipeline*, sino que periódicamente la herramienta estará chequeando que el clúster cumpla con lo declarado y ante cualquier actualización de imagen de Docker o código en los manifiestos, sincronizará los cambios rápidamente.

[4]https://www.gitops.tech/
[5]https://fluxcd.io/
[6]https://argoproj.github.io/cd/
[7]https://jenkins-x.io/

Anexos

A1. Selección de servidor de CI

La Tabla A1.1 muestra el top cinco de los servidores de integración continua más utilizados, de acuerdo a las diferentes páginas web visitadas ([Katalon, 2023], [Altexsoft, 2022], [BasuMallick, 2022], [Sheth, 2022], [Taylor, 2023], [Foresight, 2022] y [Adhithi, 2023]), cuyos nombres se detallan en el encabezado de cada una de las columnas de la misma.

Tabla A1.1: *Top cinco de los servidores de CI más utilizados.*

	Katalon	**Altexsoft**	**Spiceworks**	**Lambdatest**	**Guru99**	**Foresight**	**Testsigma**
1	Jenkins	Jenkins	Azure Pipelines	Jenkins	Buddy	Jenkins	Jenkins
2	TeamCity	TeamCity	Bamboo	TeamCity	Jenkins	TravisCI	CircleCI
3	CircleCI	Bamboo	GitLab CI	CircleCI	TeamCity	CircleCI	Testsigma
4	Bamboo	TravisCI	Harness	TravisCI	GoCD	GitLab CI	Spinnaker
5	GitLab CI	CircleCI	Jenkins	GitLab CI	Bamboo	Bamboo	TravisCI

En base a la Tabla A1.1, se creó la Tabla A1.2, la cual detalla la cantidad de menciones que tiene cada servidor en la tabla anterior, ordenados numéricamente de mayor a menor por la cantidad de menciones.

Tabla A1.2: *Cantidad de menciones de los servidores de CI en las páginas web seleccionadas.*

Servidor	**Menciones**
Jenkins	7
Bamboo	5
CircleCI	5
GitLab CI	4
TeamCity	4
TravisCI	4
Azure Pipelines, Buddy, GoCD, Harness, Spinnaker, Testsigma	1

A2. Selección de la herramienta para evaluar la calidad del código fuente

La Tabla A2.1 muestra el top cinco de las herramientas que realizan análisis estático de la calidad del código fuente más utilizadas, de acuerdo a las diferentes páginas web visitadas ([Zelleke, 2023], [SoftwareTestingHelp, 2023], [Martin, 2023], [G2, 2023], [TrustRadius, 2023], [PeerSpot, 2023] y [Dowling, 2022]), cuyos nombres se detallan en el encabezado de cada una de las columnas de la misma.

Tabla A2.1: *Top cinco de los analizadores de calidad de código fuente más utilizados.*

	Compari-tech	**Software Testing Help**	**Guru99**	**G2**	**TrustRadius**	**PeerSpot**	**LinearB**
1	SonarQube	Raxis	SmartBear Collaborator	Synopsys Coverity	Veracode	Fortify SCA	SonarQube
2	Checkmarx SAST	SonarQube	Embold	ReSharper	PyCharm	CodeSonar	Codacy
3	Synopsis Coverity	PVS-Studio	PVS-Studio	SonarQube	SonarQube	Veracode	DeepSour-ce
4	Fortify SCA	DeepSource	SonarQube	DeepSour-ce	Rencore Code	PyCharm	Embold
5	Veracode	SmartBear Collaborator	HelixQAC	Semmie	Codacy	ReSharper	Veracode

En base a la Tabla A2.1, se creó la Tabla A2.2, la cual detalla la cantidad de menciones que tiene cada analizador en la tabla anterior, ordenados numéricamente de mayor a menor por la cantidad de menciones.

Tabla A2.2: *Cantidad de menciones de las herramientas en las páginas web seleccionadas.*

Analizador	**Menciones**
SonarQube	6
Veracode	4
DeepSource	3
Embold	2
PVS-Studio	2
Codacy	2
Fortify Static Code Analyzer	2
PyCharm	2
ReSharper	2
SmartBear Collaborator	2
Synopsys Coverity	2
Checkmarx SAST, CodeSonar, HelixQAC, Raxis, Rencore Code, Semmie	1

A3. Relevación de información sobre el desarrollo en la Facultad de Ingeniería

Para poder realizar una propuesta al Área de Desarrollo de la Facultad de Ingeniería de la UNLPam, se recolectó información en el año 2022, entrevistando personalmente al Ing. Damián Puente, que a la fecha de la redacción de esta tesis, es el encargado de dicho área. Por aquellos tiempos, todavía no se contaba con automatización, dato que se puede apreciar en el intercambio de correos electrónicos adjuntados.

En el corriente año, toda la información relevada fue a través de mails y esto fue el insumo para la redacción del Capítulo 4.

A continuación, se adjunta la cadena de correos electrónicos intercambiados con el Ing. Puente. Para una mejor comprensión, se recomienda al lector leer desde el final hacia el inicio, es decir, desde la última página insertada hasta la primera de ellas.

Re: Consulta para tesis

Damian Puente <dpuente@ing.unlpam.edu.ar>
Mar 28/3/2023 09:01
Para: Valentina Scovenna <valentina_17_01@hotmail.com>

buen dia, respondo!

1. La ejecución de los tests y creación del archivo .tar.gz, ¿se realiza en el paso a DEV y también en el paso a PROD?

Los test siempre se ejecutan cuando se crea un nuevo PR, y los tar.gz solo cuando se crea un nuevo release. Esto es tanto en DEV como PROD

1. ¿Se genera un .tar.gz para el ambiente de DEV y se hace su instalación en el servidor de desarrollo para un testeo previo?

Si

1. ¿Cómo se accede a los servidores que provee la ARIU? ¿También tienen servidores de DEV, PROD y Failover?

La ARIU provee la aplicacion. Nosotros instalamos la aplicacion en nuestros servidores (locales o los que nos provee la ARIU). De ahi podemos tener más de una instalación si así lo quisiéramos. Pero particularmente sobre esos tenemos uno solo. Nosotros no hacemos desarrollos sobre esos sistemas, sólo mantenemos (backup, mantenimiento de hardware).

1. En caso de no tener un servidor de DEV, los desarrollos que sean para los sistemas que se encuentran almacenados en los servidores de la ARIU, ¿son testeados antes en el servidor local de desarrollo?

Los sistemas "nuestros" (desarrollados por nosotros) y que estan en datacenter de ARIU si tienen sus servidores dev...generamente virtuales locales

1. Si el test unitario falla, ¿la pipeline de CI da un aviso por correo?

Si, siempre. Github tiene esa funcionalidad.

1. ¿Cómo generan reclamos los alumnos? ¿Quién crea los tickets para que se haga un cambio en los sitios?

No recibimos reclamos de alumnos. Los tickets los genera el personal de la facultad, docentes, no docentes y directivos

1. ¿Vos mismo aprobas tus Pull Requests en caso de que no haya más personas en el área?

Generalmente estoy solo, y no me queda otra.

1. El sistema de monitoreo que avisa si hay incidentes, ¿son incidentes en el servidor? Si es así, ¿en qué servidores? ¿locales o los provistos por la ARIU también?

El monitoreo alerta sobre conectividad y chequeo de algún puerto en particular (80, 443, etc) sobre todos los servidores que tenemos en producción

1. Cuando mencionas el **sistema de administración interno**, ¿se hace referencia a un solo sistema, o son varios?

Es un solo sistema con varios módulos.

1. ¿Qué procesos abarca el sistema de administración interno?

Procesos que tienen que ver con la administración de la facultad; academica, ciencia y técnica, administración, etc

1. ¿Qué significa que sólo mantienen los sistemas del consorcio SIU? ¿Qué actividades involucra ese mantenimiento? ¿Y en qué servidor se encuentra?

Backup y hardware (en algunos casos)

1. **Se cuenta con servidores físicos para la web Campus Virtual, y para la administración de red.** Este es un mensaje del año pasado, pero me quedó la duda, ¿qué sería la administración de red?

gestión de tráficos de red, firewall, mapeos de puertos, control de acceso a servicios, etc. de todos los dispositivos que se conectan a nuestra red. Eso también incluye al tráfico de los alumnos.

1. ¿Que sistemas utilizan MySql?

Todos menos los sistemas SIU

El lun, 27 mar 2023 a las 9:35, Valentina Scovenna (<valentina_17_01@hotmail.com>) escribió:

> Hola Damián,
>
> Ya redactando veo que me falta más información. Te consulto...
>
> 1. La ejecución de los tests y creación del archivo .tar.gz, ¿se realiza en el paso a DEV y también en el paso a PROD?
> 2. ¿Se genera un .tar.gz para el ambiente de DEV y se hace su instalación en el servidor de desarrollo para un testeo previo?
> 3. ¿Cómo se accede a los servidores que provee la ARIU? ¿También tienen servidores de DEV, PROD y Failover?
> 4. En caso de no tener un servidor de DEV, los desarrollos que sean para los sistemas que se encuentran almacenados en los servidores de la ARIU, ¿son testeados antes en el servidor local de desarrollo?
> 5. Si el test unitario falla, ¿la pipeline de CI da un aviso por correo?
> 6. ¿Cómo generan reclamos los alumnos? ¿Quién crea los tickets para que se haga un cambio en los sitios?
> 7. ¿Vos mismo aprobas tus Pull Requests en caso de que no haya más personas en el área?
> 8. El sistema de monitoreo que avisa si hay incidentes, ¿son incidentes en el servidor? Si es así, ¿en qué servidores? ¿locales o los provistos por la ARIU también?
> 9. Cuando mencionas el **sistema de administración interno**, ¿se hace referencia a un solo sistema, o son varios?
> 10. ¿Qué procesos abarca el sistema de administración interno?
> 11. ¿Qué significa que sólo mantienen los sistemas del consorcio SIU? ¿Qué actividades involucra ese mantenimiento? ¿Y en qué servidor se encuentra?
> 12. **Se cuenta con servidores físicos para la web Campus Virtual, y para la administración de red.** Este es un mensaje del año pasado, pero me quedó la duda, ¿qué sería la administración de red?
> 13. ¿Que sistemas utilizan MySql?
>
> Desde ya, muchísimas gracias!
>
> Saludos
>
> ---
>
> **De:** Damian Puente <dpuente@ing.unlpam.edu.ar>
> **Enviado:** lunes, 20 de marzo de 2023 11:49
> **Para:** Valentina Scovenna <valentina_17_01@hotmail.com>
> **Asunto:** Re: Consulta para tesis

> Existe un sistema de monitoreo que avisa mediante mail si existe algún incidente, en ese caso generalmente yo actúo sobre eso.
>
> Respecto a los release con errores...o nos enteramos nosotros porque vemos el error o bien son reportados por los usuarios, y luego lo mismo. Se carga un ticket para la solucion.
>
> El lun, 20 mar 2023 a las 10:30, Valentina Scovenna (<valentina_17_01@hotmail.com>) escribió:
>
>> Gracias Damián nuevamente 🙂
>>
>> Me queda una última pregunta, ahora sí, el equipo de mantenimiento y monitoreo, son los que avisan en caso de fallas? Si es así, hay algún sistema de tickets para eso o cómo lo van resolviendo?
>> Cómo se enteran si el último release tiene errores?
>>
>> ---
>>
>> **De:** Damian Puente <dpuente@ing.unlpam.edu.ar>
>> **Enviado:** lunes, 20 de marzo de 2023 09:54
>> **Para:** Valentina Scovenna <valentina_17_01@hotmail.com>
>> **Asunto:** Re: Consulta para tesis
>>
>> Si ahora cambiamos por GitHub y como decis temos un action de CI pero todavia el CD no.
>> Los test son unitarios corren solos al crear los PRs.
>>
>> Esta año todavia no hemos armado el equipo de desarrollo...ya que como te mencione siempre son conformado por estudiantes.
>>
>> Los sistemas que usan postgresql son los que desarrolla el consorcio SIU, esos no son desarrollados por nosotros....solo son mantenidos.
>>
>> Las ramas son:
>>
>> - los features/xxx (unos por cada nuevo incidente o nueva funcionalidad), donde xxx es el número de ticket
>> - dev, donde se "mergean" los features
>> - prod, ultima versión "deployada"..
>>
>> Los release son los instalables que se generan para deployar (serian los tar.gz en este caso)
>>
>> El lun, 20 mar 2023 a las 9:34, Valentina Scovenna (<valentina_17_01@hotmail.com>) escribió:
>>
>>> Muchas gracias Damián!
>>>
>>> Entonces, ya no usan Bitbucket, sino que ahora utilizan GitHub, y tienen una pipeline de CI en github actions pero no un CD, cierto?
>>> Los tests que tienen son unitarios? Porque el año pasado cuando hablamos, creo que no tenían esa parte automatizada sino que cada desarrollador hacía las pruebas antes del PR hacia DEV.
>>>
>>> Cuántas personas hay en el área?
>>> Qué sistemas usan la base PostgreSql y cuáles usan MySQL?
>>> Y en cuanto a ramas tienen: feature, dev, prod y release?
>>>
>>> ---
>>>
>>> **De:** Damian Puente <dpuente@ing.unlpam.edu.ar>
>>> **Enviado:** lunes, 20 de marzo de 2023 08:52

Para: Valentina Scovenna <valentina_17_01@hotmail.com>
Asunto: Re: Consulta para tesis

Hola Valentina como estas?, te respondo:

1. Tengo entendido, por conversaciones anteriores, que solo estás vos como desarrollador, excepto qué se agreguen estudiantes por las pasantías. En ambos casos, ¿quién hace el despliegue a producción? ¿quién es el encargado de aprobar los Pull Requests?
    i. el desarrollador crear los PR (pull requests) y los mismos los apruebo yo. Los deploy a test o produccion los hace el desarrollador en algunos casos tambien los puedo hacer yo.
2. Si el .tar.gz se genera automáticamente, ¿qué herramienta se utiliza para esto? ¿Es en el mismo Bitbucket?
    i. ahora mismo usamos github, para los tar.gz usamos git actions, pero antes de generar un nuevo release se corren los test, tambien con actions de github
3. ¿Qué inconvenientes detecta con esta metodología de trabajo?
    i. dentro de la infraestructura que manejamos estamos bastante bien. Si contaramos con mas servicio cloud podriamos hacer todos el despligue de manera automatica. Es decir PR-> release -> tar.gz -> deploy
4. ¿Quién administra la infraestructura?
    i. la infraestructura la administramos desde el dpto. de sistemas; mantenimiento, monitoreo y backups
5. ¿Qué servidor web utiliza?
    i. ahora está la mayoría en servidores LAMPs.
6. ¿Disponen de un servidor de desarrollo? ¿El de failover qué función cumple?
    i. si, hay un servidor de test para los desarrollos. Los failovers, que en realidad son copias de los servicios en produccion, se mantienen por si es necesario reemplazar los de producción por si sucede algún evento inesperado.
7. ¿Cómo actualizan el sitio web y los sistemas internos? ¿Es de la misma manera que con el Campus? (ver diagrama)
    i. en el diagrama faltan los PR a dev. el camino tanto a dev como a prod debe ser asi:
        1. feature branch -> PR (a dev)-> merge ->dev -> PR (a prod) -> merge -> PROD -> release (nueva version en prod) (*) (Deberias acomodar eso en el diagrama)
8. ¿Qué motor de base de datos utilizan? (campus, sitio web y sistemas internos)
    i. tenemos MySQL y PostgreSQL
9. ¿Quién es el "cliente"? ¿Los mismos usuarios de los sistemas? ¿Se generan actualizaciones de acuerdo a reclamos?
    i. nuestros clientes son todos los alumnos, docentes, no docentes y público en general que accede a nuestros sitios o campus virtual (personas que hacen algún curso)
    ii. los reclamos se cargan mediante tickets, para cada ticket se crear un nuevo feature branch y se sigue el proceso hasta un nuevo deploy en prod
10. En caso de fallas, ¿cómo se recuperan de estas? ¿realizan vuelta atrás hacia alguna versión estable?
    i. depende de la falla, si es de hardware o software. Si vamos a lo que es software, si, podemos volver a una o más versiones para atrás, se crear el ticket y se sigue el proceso (*)

Bueno creo que está todo... cualquier cosa me avisas.

Saludos

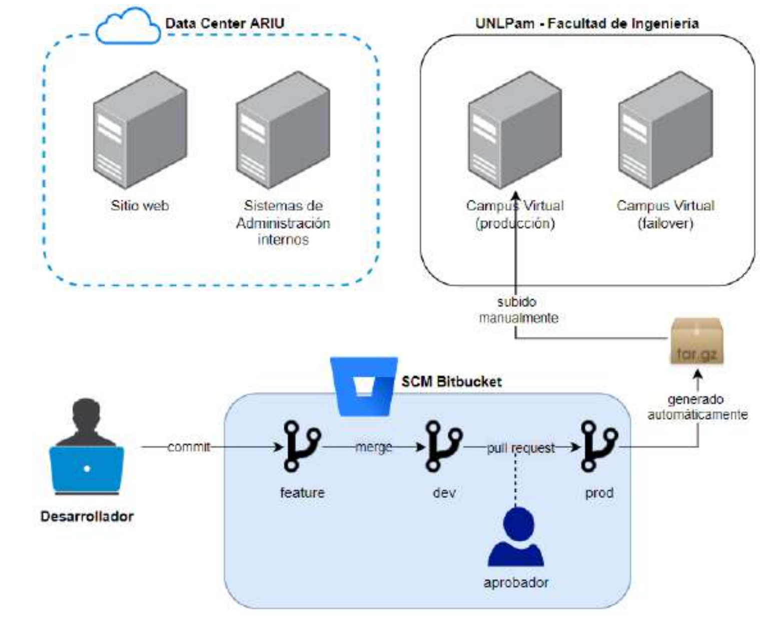
El jue, 16 mar 2023 a las 9:25, Valentina Scovenna (<valentina_17_01@hotmail.com>) escribió:

> Buen día Damián, espero estés muy bien.

Como te comentó Belén, surgieron más dudas con respecto a la situación actual del desarrollo y la infraestructura de la facultad.
Estas son las siguientes:

1. Tengo entendido, por conversaciones anteriores, que solo estás vos como desarrollador, excepto qué se agreguen estudiantes por las pasantías. En ambos casos, ¿quién hace el despliegue a producción? ¿quién es el encargado de aprobar los Pull Requests?
2. Si el .tar.gz se genera automáticamente, ¿qué herramienta se utiliza para esto? ¿Es en el mismo Bitbucket?
3. ¿Qué inconvenientes detecta con esta metodología de trabajo?
4. ¿Quién administra la infraestructura?
5. ¿Qué servidor web utiliza?
6. ¿Disponen de un servidor de desarrollo? ¿El de failover qué función cumple?
7. ¿Cómo actualizan el sitio web y los sistemas internos? ¿Es de la misma manera que con el Campus? (ver diagrama)
8. ¿Qué motor de base de datos utilizan? (campus, sitio web y sistemas internos)
9. ¿Quién es el "cliente"? ¿Los mismos usuarios de los sistemas? ¿Se generan actualizaciones de acuerdo a reclamos?
10. En caso de fallas, ¿cómo se recuperan de estas? ¿realizan vuelta atrás hacia alguna versión estable?

Realicé el siguiente diagrama sobre la información que tenía, por favor, espero sus comentarios si hay algo que no es correcto.



Desde ya, muchísimas gracias!

Saludos,
Valentina

De: Damian Puente <dpuente@ing.unlpam.edu.ar>
Enviado: jueves, 22 de diciembre de 2022 12:41

Para: Valentina Scovenna <valentina_17_01@hotmail.com>
Asunto: Re: Consulta para tesis

Hola Valentina, me colgué en responderte...

En cuanto a infra tenemos:

- servidores de campus virtual (prod y failover) en gral pico
- servidor de administración de red para el edificio en pico
- Sitio web y sistema administracion interno. Esto lo tenemos un un datacenter de la ARIU (Asociación Red de Interconexión Universitaria)

En cuanto a desarrollo, utilizamos Django como mencionas.

El jue, 15 dic 2022 a las 7:56, Valentina Scovenna (<valentina_17_01@hotmail.com>) escribió:

> Buen día Damián, espero estés muy bien.
> Con el trabajo se me ha complicado para ir hasta la facultad a hacerte un par de consultas, así que las envío por acá.
>
> ¿Me podrías dar detalle de la arquitectura de la infraestructura actual en el área de desarrollo? Entiendo que según lo que hablamos la primera vez, tienen una parte en servidores físicos y otra en la nube.
> Y en cuanto al desarrollo de software, utilizan el framework Django, ¿cierto?
>
> Desde ya, muchas gracias.
>
> Saludos,
> Valentina
>
> ---
>
> **De:** Damian Puente <dpuente@ing.unlpam.edu.ar>
> **Enviado:** lunes, 17 de octubre de 2022 09:39
> **Para:** Valentina Scovenna <valentina_17_01@hotmail.com>
> **Asunto:** Re: Consulta para tesis
>
> Hola Valentina, por el momento no utilizamos alguna metodología ya que en la mayoría de los casos estoy yo solo.
> Seguramente en tu trabajo usarán scrum, en la mayoría de los privados la utilizan.
>
> Saludos!
>
> El lun, 17 oct 2022 a las 9:09, Valentina Scovenna (<valentina_17_01@hotmail.com>) escribió:
>
>> Buen día Damián, ¿cómo estás?
>>
>> Empezando a redactar me quedaron algunas dudas sobre lo que charlamos la vez anterior, por eso te hago una consulta sobre el desarrollo que realizan en el área, ¿aplican alguna de las metodologías ágiles?
>> Si es así, ¿cuál están utilizando?
>>
>> Desde ya, muchísimas gracias.
>>
>> Saludos,
>> Valentina

A4. Tets unitarios

Durante este trabajo final, se desarrollaron tests unitarios en Angular para cada una de las funciones de la aplicación calculadora. En este anexo se ejemplifica con la función `allClear()`, definida en el archivo `app.component.ts`, cuyo propósito es limpiar por completo la pantalla de entrada, dejando en la misma el valor 0.

El código de la función es el siguiente:

```
allClear() {
    this.result = '0';
    this.input = '0';
}
```

Al generar un aplicación Angular, estas traen por defecto el archivo donde se declaran las pruebas unitarias a realizar, llamado `app.component.spec.ts`, el cual comienza con la importación de `AppComponent` y `TestBed`. La primera importa la clase del componente principal de la aplicación, desarrollada en el archivo `app.component.ts`, mientras que la importación restante trae consigo la clase que permite configurar y crear un entorno de prueba en Angular.

Seguido de esto, y tal como se puede apreciar en el siguiente código, se utiliza la función `describe()`, la cual agrupa un conjunto de pruebas relacionadas, permitiendo su estructuración y organización. Nótese que la misma se cierra al final del código, englobando los tests declarados.

```
import { TestBed } from '@angular/core/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  });

  it(`allClear `, () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;
    app.result = '3';
    app.allClear();
    expect(app.result).toEqual('0');
  });
});
```

A continuación, se declara con la función `beforeEach()` lo que realizará Angular antes de ejecutar cada prueba, que en este caso es preparar el entorno para el componente `Appcomponent`, asegurando de esta manera que el componente esté listo para ser utilizado y testeado.

Luego, se define la prueba unitaria con la función `it()`, indicando la función a testear que, como se mencionó al inicio, será `allClear()`.

Dentro de ésta se crea una instancia del componente `AppComponent`, asignándolo a la variable constante `fixture`, para luego obtener la referencia al componente actual empleando la función `componentInstance` y almacenarla en la constante `app`.

Una vez realizadas las declaraciones iniciales, se procede a declarar en la propiedad `result` el valor '3', esta alude al valor mostrado en la pantalla de la calculadora. Por lo tanto, cuando se ejecute el paso `app.allClear()`, debería setear el `app.result` a cero, de acuerdo a lo declarado en dicha función. Es por esta razón que la línea final del testeo indica que se espera que `app.result` sea igual a cero.

Esta misma lógica es la que aplica para los tests de cada una de las funciones declaradas en el archivo `app.component.ts`.

A5. Análisis de código fuente con SonarQube

Una vez ejecutado el análisis, al ingresar a la plataforma de SonarQube, el proyecto ya está creado con el nombre indicado en el archivo `sonar-project.properties`.

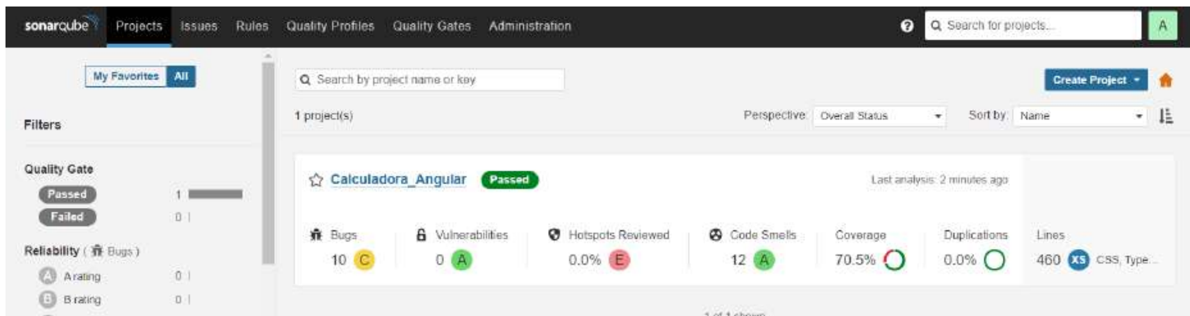


Figura A5.1: *Proyecto creado y analizado en SonarQube.*

En la Figura A5.1, se puede ver el nombre del proyecto, el estado del Quality Gate, el cuál es *"Passed"*, es decir que fue exitoso, y detalles del análisis como la cantidad de bugs encontrados, code smells, y el porcentaje de cobertura que poseen los tests unitarios.

Al clickear en su nombre, se abre una ventana (ver Figura A5.2) que muestra la misma información pero con un seguimiento histórico, que se denotará más adelante en esta sección, ya que permite visualizar los resultados del código en su totalidad o del nuevo código analizado.

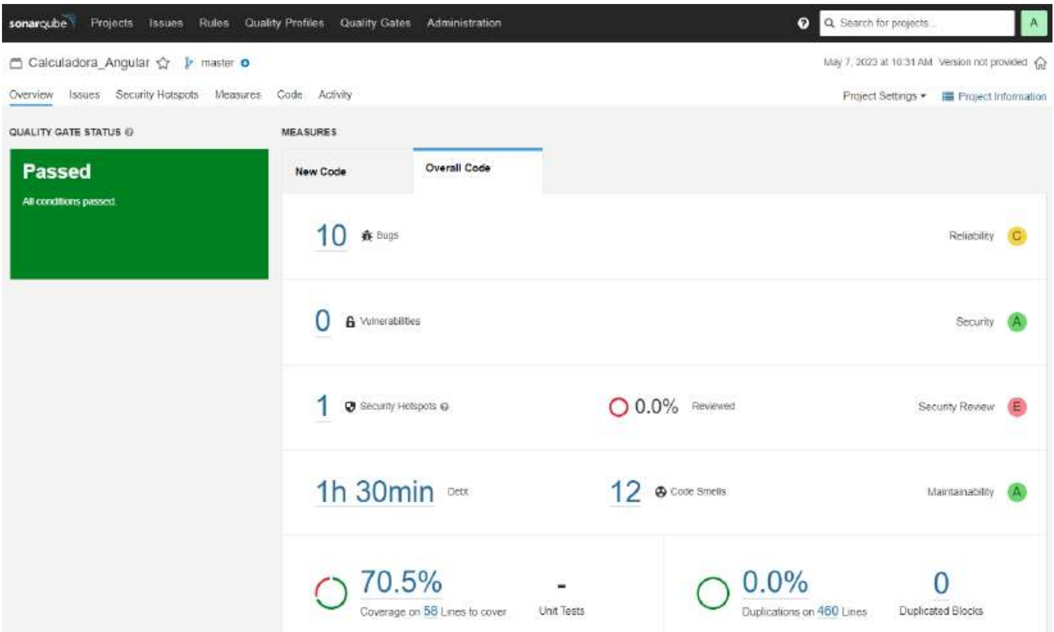


Figura A5.2: *Información del proyecto Calculadora_Angular.*

A5.1. Bugs

Al seleccionar los *Bugs* encontrados, SonarQube muestra a detalle cada uno de ellos (ver Figura A5.3), con información como su título, a qué archivo pertenecen, si ya ha sido asignado a algún desarrollador. También muestra su estado (si está abierto o si ya ha sido resuelto), su criticidad (informativo,

menor, mayor, crítico o bloqueante), el tiempo que conlleva solucionarlo y sus etiquetas, si estas existieran.

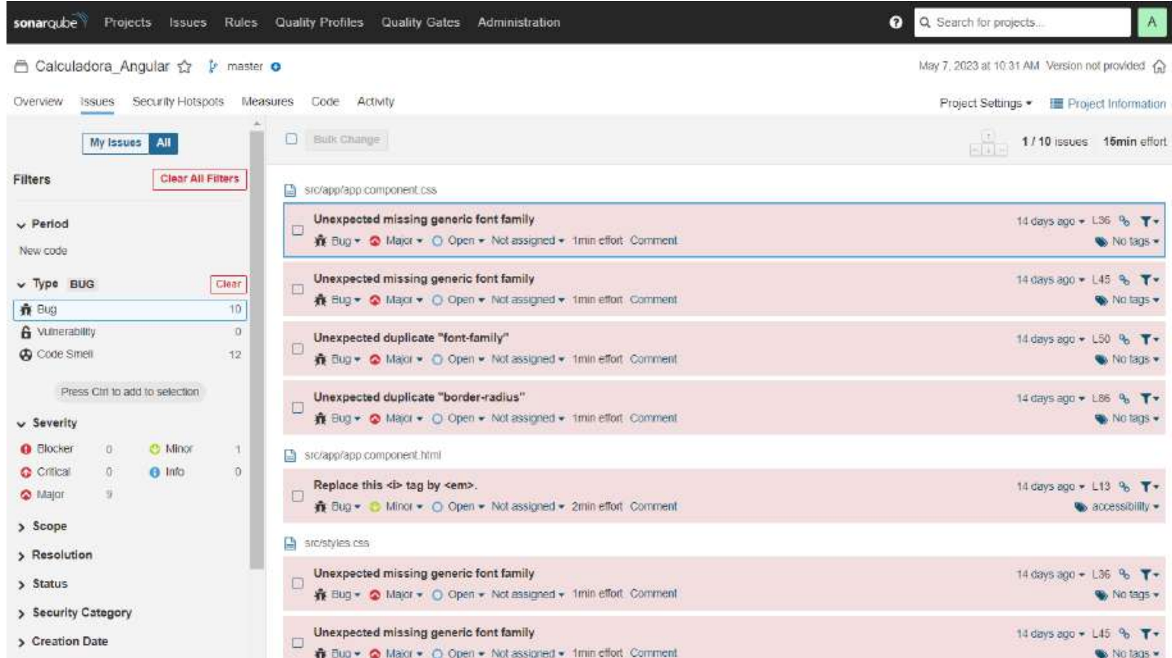


Figura A5.3: Detalle de los bugs localizados luego del análisis.

Eligiendo uno de ellos, se puede ver que se dirige a la línea precisa del archivo en cuestión, indicando cuál es el problema con la misma, tal como se puede ver en la Figura A5.4.

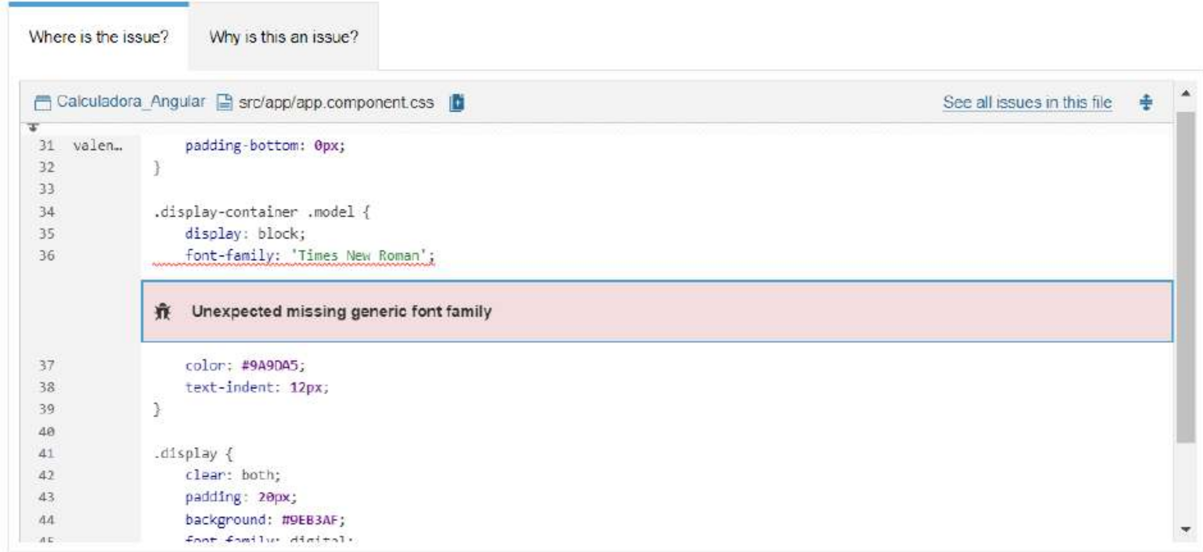


Figura A5.4: Ejemplo de la localización de uno de los bugs.

En la pestaña siguiente se explica por qué esto es un problema, y cómo se debería resolver, según detalla la Figura A5.5.

Figura A5.5: Solución al bug encontrado.

A5.2. Code smells

Code smells es, según se traduce al español, código que apesta. Es cualquier síntoma en el código fuente de un programa que posiblemente indica un problema más profundo. Según [Wikipedia, 2022], estos no son errores, sin embargo indican defectos en el diseño de software que pueden entorpecer el desarrollo o aumentar el riesgo de errores o fallos en el futuro.

Como se puede observar en la Figura A5.6, algunos indican bloques de código vacíos, mientras que otros hablan de funciones deprecadas.

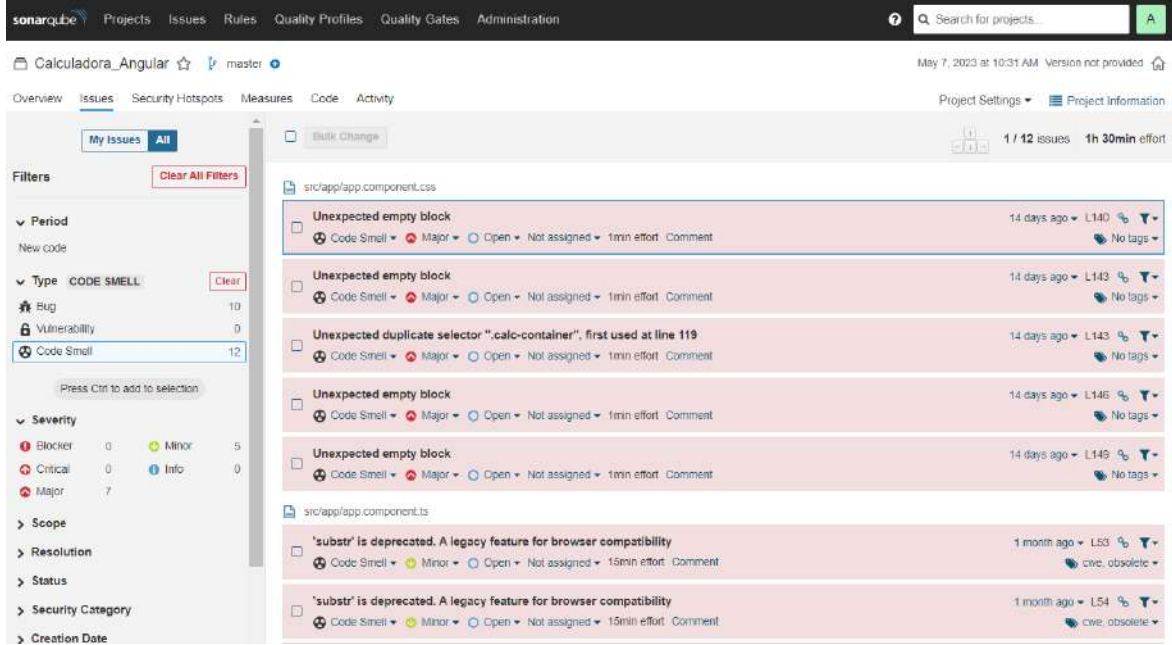


Figura A5.6: Detalle de los code smells localizados luego del análisis.

A modo de ejemplo, se seleccionó uno de ellos que muestra una línea de código comentada, que SonarQube recomienda eliminar (ver Figura A5.7).

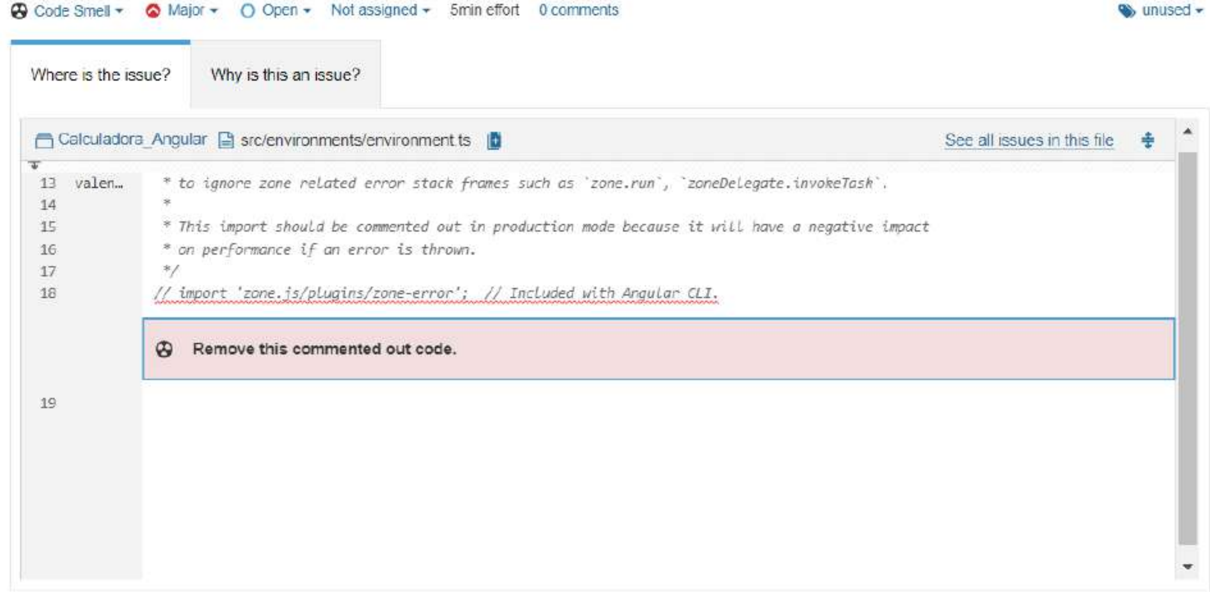


Figura A5.7: Ejemplo de uno de los code smell.

Como explicación a este *code smell*, indica que los programadores no deben comentar el código, ya que se reduce la legibilidad. Asimismo, en la Figura A5.8 la herramienta señala que, en caso tal de ser necesario, se puede recuperar del historial del SCM.

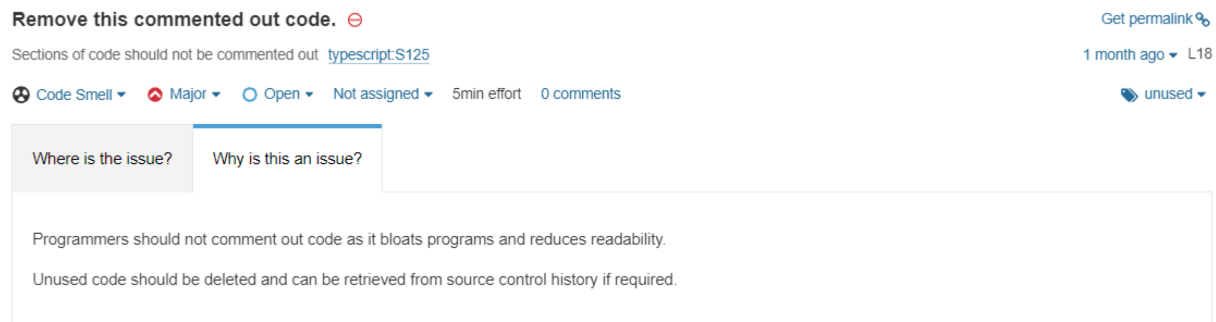


Figura A5.8: Solución al code smell encontrado.

A5.3. Cobertura de código

Como se ha ido mencionando a lo largo del capítulo , los tests unitarios fueron desarrollados dentro de la aplicación Angular, por lo que se añadió al archivo `karma.conf.js` la ruta en la cual depositar el reporte de tal ejecución. De esta manera, en el archivo `sonar-project.properties`, se declaró a dónde debe ir SonarQube a buscar dichos resultados.

En la Figura A5.9, se muestran los archivos desarrollados en el lenguaje *TypeScript* analizados, y la cobertura de código de cada uno de ellos.

Calculadora_Angular — View as List — ↑ ↓ to select files — ← → to navigate — 4 files

Coverage 70.6%

	Coverage	Uncovered Lines	Uncovered Conditions
src/main.ts	0.0%	3	–
src/test.ts	0.0%	4	–
src/app/app.component.ts	76.3%	7	12
src/environments/environment.ts	100%	0	–

4 of 4 shown

Figura A5.9: *Detalle de la cobertura de testeo que posee el proyecto.*

Al seleccionar el archivo `app.component.ts` se observa, según se muestra en la Figura A5.10, cuáles son las líneas de código que no tienen cobertura en las pruebas unitarias, marcando a las mismas con rojo.

Calculadora_Angular / src / app / app.component.ts

```
15        pressNum(num: string) {
16          //Do Not Allow . more than once
17          if (num == ".") {
18            if (this.input != "") {
19              const lastNum = this.getLastOperand()
20              console.log(lastNum.lastIndexOf("."))
21              if (lastNum.lastIndexOf(".") >= 0) return;
22            }
23          }
24
25          //Do Not Allow 0 at beginning.
26          //Javascript will throw Octal literals are not allowed in strict mode.
27          if (num == "0") {
28            if (this.input == "") {
29              return;
30            }
31            const PrevKey = this.input[this.input.length - 1];
32            if (PrevKey === '/' || PrevKey === '*' || PrevKey === '-' || PrevKey === '+') {
33              return;
34            }
35          }
36
37 valen...  if (this.input == '0'){
38            this.input = ''
39          }
40
41 valen...  this.input = this.input + num
42          this.calcAnswer();
43        }
44
```

Figura A5.10: *Ejemplo de líneas de código que no están cubiertas por tests unitarios.*

En caso tal de contar con una cobertura parcial, esto se indica con rayas diagonales en distintos tonos de rojo.

A5.4. Security hotspots

De acuerdo a la documentación oficial [SonarSource, 2023], un *security hotspot* (traducido al español como punto de acceso de seguridad) indica un fragmento de código sensible a la seguridad que el desarrollador debe revisar. Luego de realizar el chequeo, indicará si esto es una posible amenaza y hay que aplicar una solución para proteger el código, o si no es necesario.

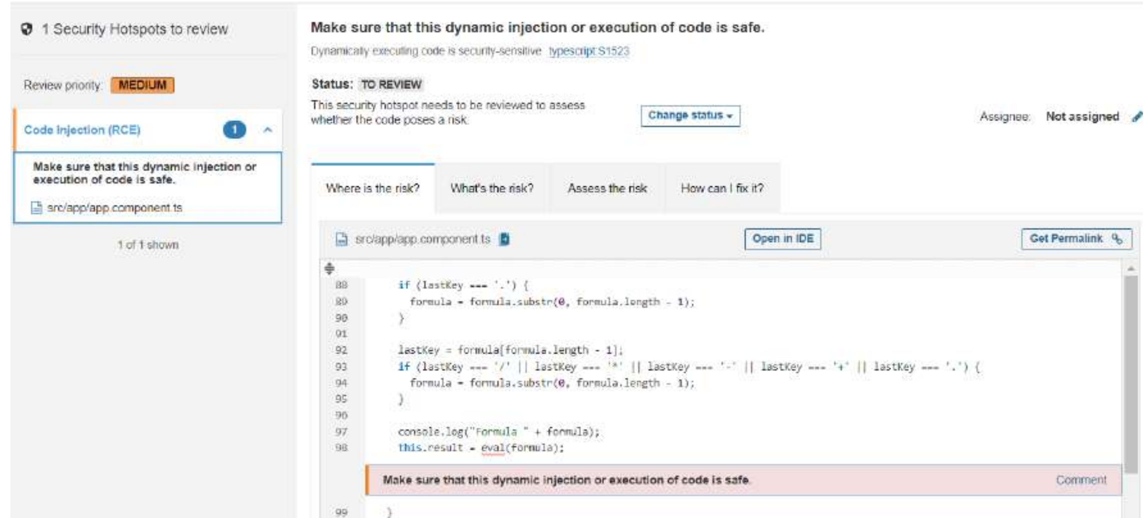


Figura A5.11: Información del security hotspot hallado.

En la Figura A5.11, la herramienta encontró un único *security hotspot*, el cuál trata sobre validar si la ejecución de código de la línea señalada es segura. En las pestañas subsiguientes, SonarQube aporta información de por qué este sería un riesgo, cómo evaluarlo y cómo solucionarlo.

La línea de código marcada en la Figura A5.11 solamente realiza la evaluación de la fórmula ingresada en la calculadora, asignando el resultado de la misma a la variable `result`, por lo cual no presenta un riesgo de seguridad para la aplicación. Por esta razón es que se marcó como seguro, completando con un comentario como se ve en la Figura A5.12.

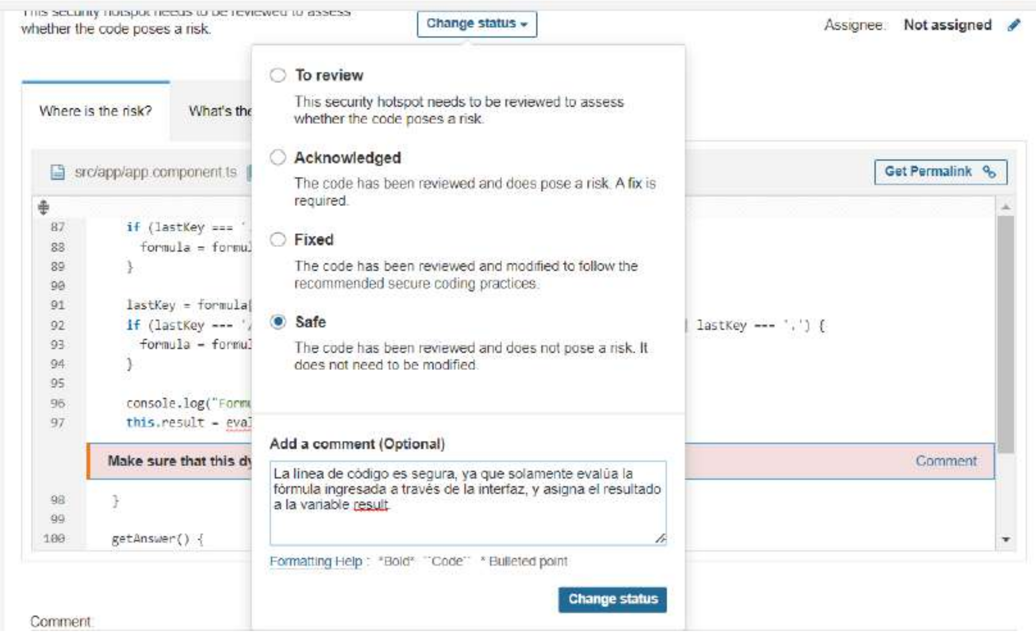


Figura A5.12: Resolviendo el security hotspot indicado.

Con dicho cambio en el estado, automáticamente la plataforma mostró una nueva ventana, que indica el nuevo porcentaje de cobertura, y que ya no hay más *security hotspots* por examinar (ver Figura A5.13).

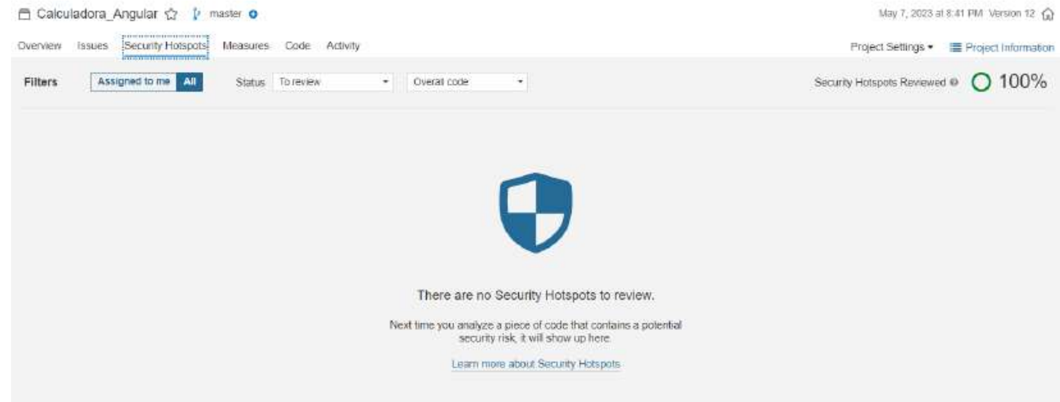


Figura A5.13: *Security hotspot resuelto.*

A5.5. Comparación de resultados

Para comprobar la efectividad del analizador de calidad de código fuente, SonarQube, se realizaron los cambios recomendados por la herramienta a fin de obtener una mejora en los resultados.

La Figura A5.14, denota una mejoría. La cantidad de bugs disminuyó, al igual que los *code smells*, y la cobertura de código mejoró sobrepasando el 80 %.

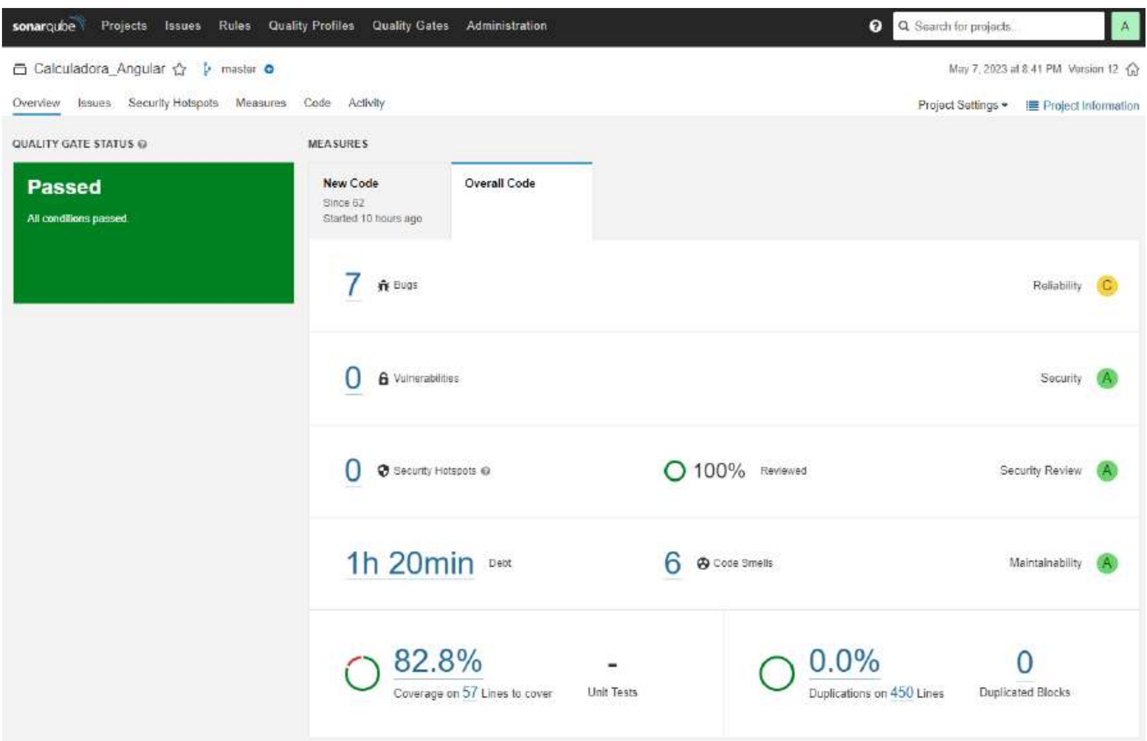


Figura A5.14: *Detalle de nuevos resultados del análisis.*

En la Figura A5.15 se pueden apreciar los resultados anteriores, contrastando con la última ejecución ilustrada en la Figura A5.16. Se visualiza fácilmente, cómo la línea de los *code smells* decae significativamente al final del gráfico, de igual manera, esto sucede con la línea que representa los bugs pero en menor medida.

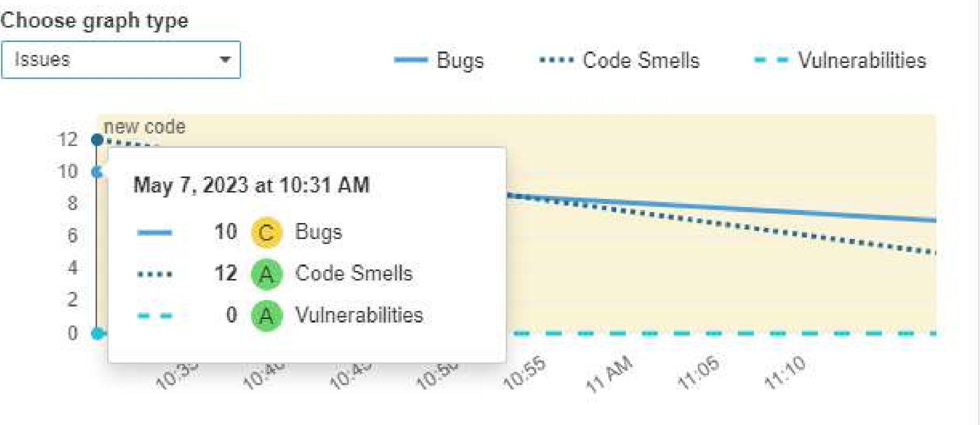


Figura A5.15: *Gráfico de problemas en la ejecución anterior.*

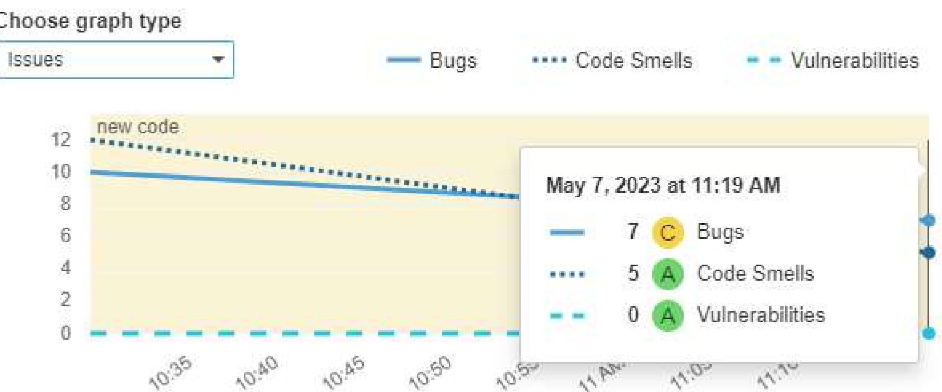


Figura A5.16: *Gráfico de problemas en la nueva ejecución.*

En el caso de la cobertura de código fuente, no es tan visible el cambio entre la Figura A5.17 y la Figura A5.18, no obstante sí se realizó una mejoría aplicando tests a líneas que antes no tenían.

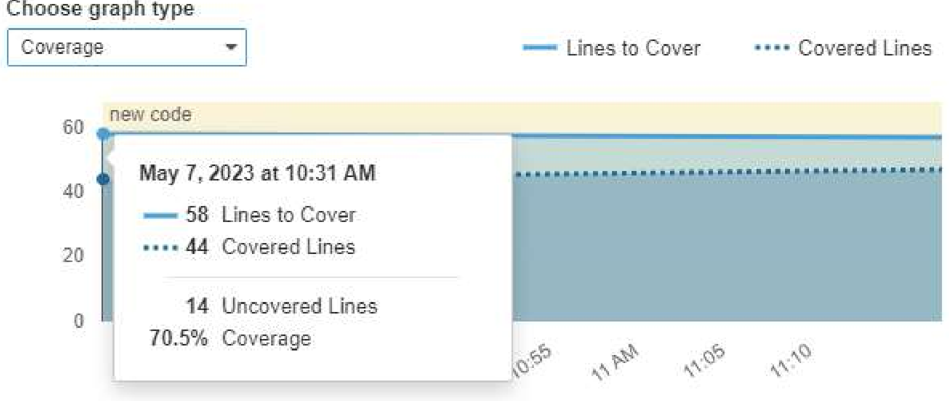


Figura A5.17: *Gráfico de problemas en la ejecución anterior.*

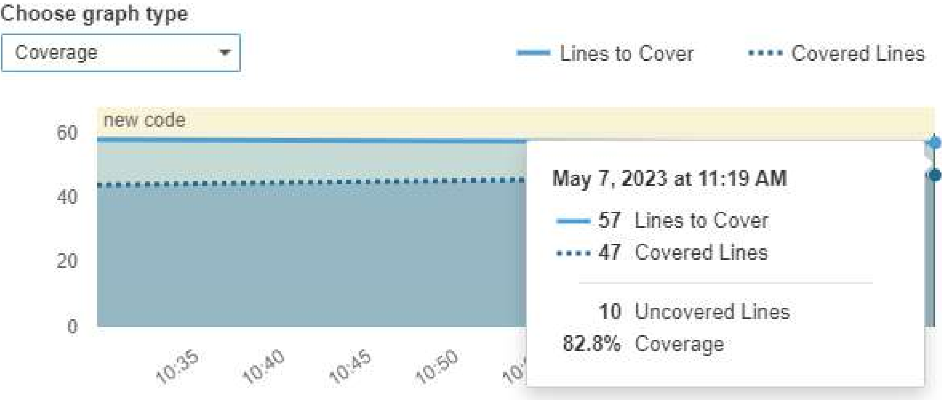


Figura A5.18: *Gráfico de problemas en la ejecución posterior.*

A6. Réplicas de Pods

Declarar más de una réplica en el manifiesto `Deployment` otorga a la aplicación **alta disponibilidad**, ya que se mantienen más de una instancia en ejecución al mismo tiempo por lo que, si una de ellas falla, las demás van a continuar ofreciendo el servicio sin inconvenientes convirtiéndose en **tolerante a fallas**. Además, el tráfico es balanceado automáticamente entre las distintas réplicas, mejorando el **rendimiento** y la **capacidad de respuesta** de la aplicación.

Adicionalmente, Kubernetes ofrece la capacidad de escalar horizontalmente, añadiendo o eliminando tantas réplicas como sea necesario, facilitando la **escalabilidad** y el manejo de tráfico.

A modo de ejemplo, se realizó una breve práctica en la que primero se chequearon los recursos publicados en el clúster de AKS. Como se puede observar en la Figura A6.1, hay dos réplicas de *Pod* ejecutándose.

```
C:\>kubectl get all -n calculadora
NAME                                  READY   STATUS    RESTARTS   AGE
pod/calculadora-deploy-f979747f6-jtqhn   1/1     Running   0          8m19s
pod/calculadora-deploy-f979747f6-w7lv7   1/1     Running   0          8s

NAME                          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/calculadora-service   ClusterIP   10.0.160.118   <none>        80/TCP    14m

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/calculadora-deploy   2/2     2            2           14m

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/calculadora-deploy-79b7dcd758   0         0         0       14m
replicaset.apps/calculadora-deploy-f979747f6    2         2         2       8m25s
```

Figura A6.1: *Recursos publicados en el clúster.*

Se elimina una de las réplicas, haciendo uso del comando `kubectl delete pod` indicando el nombre del *Pod* a eliminar y el `namespace` en el que se encuentra. Tal como se ve en la Figura A6.2, el *Pod* se elimina en la primer línea, indicando como salida *"deleted"*.

```
C:\>kubectl delete pod calculadora-deploy-f979747f6-8kklj -n calculadora
pod "calculadora-deploy-f979747f6-8kklj" deleted
```

Figura A6.2: *Eliminación de Pod.*

Nuevamente se consulta el estado del clúster (ver Figura A6.3), en el que se puede notar que se ha creado un nuevo *Pod*, con diferente nombre al anterior y con un tiempo de vida de apenas ocho segundos, indicando en este dato que fue creado recientemente.

Con esta pequeña prueba, se comprueba que Kubernetes mantuvo el estado deseado indicado en el manifiesto de `calculadora-deploy`, ya que todo el tiempo mantuvo dos réplicas publicadas, a pesar de la eliminación de una de ellas.

```
C:\>kubectl get all -n calculadora
NAME                                  READY   STATUS    RESTARTS   AGE
pod/calculadora-deploy-f979747f6-jtqhn   1/1     Running   0          8m19s
pod/calculadora-deploy-f979747f6-w7lv7   1/1     Running   0          8s

NAME                          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/calculadora-service   ClusterIP   10.0.160.118   <none>        80/TCP    14m

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/calculadora-deploy   2/2     2            2           14m

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/calculadora-deploy-79b7dcd758   0         0         0       14m
replicaset.apps/calculadora-deploy-f979747f6    2         2         2       8m25s
```

Figura A6.3: *Pronta recuperación luego de la eliminación de Pod.*

Otra prueba a realizar es, ¿qué sucede si se despliega una imagen errónea? Para esto se modificó la versión de la imagen en el manifiesto de `calculadora-deploy`, indicando un número de versión inexistente, lo cual al desplegar creó un *Pod* con un estado de *ErrImagePull* (ver Figura A6.4), es decir, hubo un error en la descarga de la imagen.

A pesar de esto, Kubernetes mantuvo las dos réplicas que tienen estado de `READY 1/1`, para cumplir con el estado deseado declarado.

```
C:\>kubectl get all -n calculadora
NAME                                     READY   STATUS         RESTARTS   AGE
pod/calculadora-deploy-5dc8786659-dwlcw   0/1     ErrImagePull   0          14s
pod/calculadora-deploy-f979747f6-jtqhn    1/1     Running        0          10m
pod/calculadora-deploy-f979747f6-w7lv7    1/1     Running        0          2m39s

NAME                          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/calculadora-service   ClusterIP   10.0.160.118   <none>        80/TCP    17m

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/calculadora-deploy   2/2     1            2           17m

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/calculadora-deploy-5dc8786659   1         1         0       15s
replicaset.apps/calculadora-deploy-79b7dcd758   0         0         0       17m
replicaset.apps/calculadora-deploy-f979747f6    2         2         2       10m
```

Figura A6.4: *Desplegando imagen errónea.*

Se pueden obtener más detalles haciendo uso del comando `kubectl describe`, indicando el tipo de recurso, su nombre y el `namespace` en el que se encuentra, tal como se realizó en la Figura A6.5.

```
C:\>kubectl describe pod calculadora-deploy-5dc8786659-dwlcw -n calculadora
Name:           calculadora-deploy-5dc8786659-dwlcw
Namespace:      calculadora
Priority:       0
Node:           aks-agentpool-12444751-vmss000028/10.224.0.4
Start Time:     Sun, 21 May 2023 10:52:53 -0300
Labels:         app=calculadora
                pod-template-hash=5dc8786659
Annotations:    <none>
Status:         Pending
IP:             10.244.0.141
IPs:
  IP:           10.244.0.141
Controlled By:  ReplicaSet/calculadora-deploy-5dc8786659
Containers:
  calculadora:
    Container ID:
    Image:          valen97/calculadora-angular:100
    Image ID:
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Waiting
      Reason:       ImagePullBackOff
    Ready:          False
    Restart Count:  0
    Environment:    <none>
```

Figura A6.5: *Descripción del Pod.*

Al final de lo arrojado en consola por dicho comando, se pueden ver los eventos que está o ha realizado Kubernetes. Se observa en la Figura A6.6 que, a pesar de haber fallado al descargar la imagen, se vuelve a intentar.

```
Events:
  Type     Reason     Age                    From               Message
  ----     ------     ----                   ----               -------
  Normal   Scheduled  9m22s                  default-scheduler  Successfully assigned calculadora/calculadora-deploy-5dc8786659-d
wlcw to aks-agentpool-12444751-vmss000028
  Normal   Pulling    7m56s (x4 over 9m22s)  kubelet            Pulling image "valen97/calculadora-angular:100"
  Warning  Failed     7m55s (x4 over 9m21s)  kubelet            Failed to pull image "valen97/calculadora-angular:100": rpc error
: code = NotFound desc = failed to pull and unpack image "docker.io/valen97/calculadora-angular:100": failed to resolve reference
"docker.io/valen97/calculadora-angular:100": docker.io/valen97/calculadora-angular:100: not found
  Warning  Failed     7m55s (x4 over 9m21s)  kubelet            Error: ErrImagePull
  Warning  Failed     7m32s (x6 over 9m20s)  kubelet            Error: ImagePullBackOff
  Normal   BackOff    4m18s (x20 over 9m20s) kubelet            Back-off pulling image "valen97/calculadora-angular:100"
```

Figura A6.6: *Eventos en la descripción del Pod.*

Al volver a desplegar el manifiesto con la versión de imagen que corresponde, Kubernetes automáticamente eliminó el *Pod* con errores (ver Figura A6.7) y mantuvo las dos réplicas ya existentes, puesto que la versión era la misma que contenían esos contenedores y no tuvo que realizar una actualización.

```
C:\>kubectl get all -n calculadora
NAME                                   READY   STATUS    RESTARTS   AGE
pod/calculadora-deploy-f979747f6-jtqhn   1/1     Running   0          22m
pod/calculadora-deploy-f979747f6-w7lv7   1/1     Running   0          14m

NAME                          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/calculadora-service   ClusterIP   10.0.160.118   <none>        80/TCP    29m

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/calculadora-deploy   2/2     2            2           29m

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/calculadora-deploy-5dc8786659   0         0         0       12m
replicaset.apps/calculadora-deploy-79b7dcd758   0         0         0       29m
replicaset.apps/calculadora-deploy-f979747f6    2         2         2       22m
```

Figura A6.7: *Recursos actuales en el clúster.*

A7. Estrategia de rollback

Si la versión de la aplicación publicada posee fallas, *bugs* u otro tipo de error, se puede volver a lanzar la última versión estable ejecutando manualmente la *pipeline* `calculadora-angular-deploy`.

Solamente se debe indicar en el parámetro `BUILD_NUMBER` el número de versión de imagen de Docker que se quiere utilizar, tal como se puede observar en la Figura A7.1. De esta manera, será reemplazado en el manifiesto del recurso `calculadora-deploy` y desplegará el contenedor empleando dicha imagen.

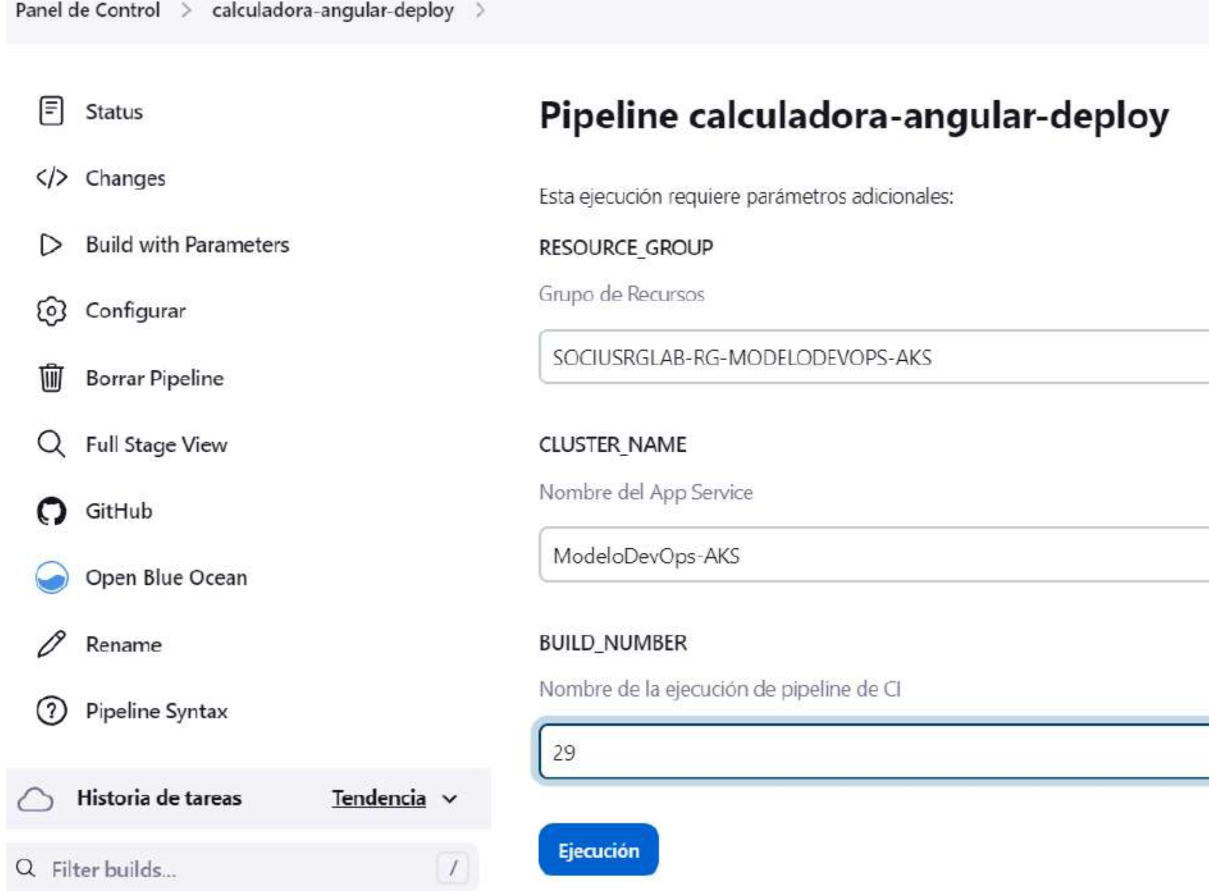


Figura A7.1: *Ejecución a demanda de la pipeline calculadora-angular-deploy.*

De este modo, se puede trabajar sobre el error proporcionándole al cliente rápidamente una versión anterior de la aplicación, minimizando los tiempos de inactividad y garantizando la continuidad del servicio para que la experiencia del usuario se mantenga consistente.

A8. Archivos Jenkinsfile

A8.1. Jenkinsfile

```
pipeline {

  agent { dockerfile { args '--privileged --network=host' } }

  stages {

    stage('Install') {
      steps {
        sh 'npm install'
      }
    }

    stage('Build') {
      steps {
        sh 'ng build'
      }
    }

    stage('Test') {
      steps {
        sh 'ng test --browsers ChromeHeadless --code-coverage'
      }
    }

    stage('SonarQube Analysis') {
      environment {
        sonarHome = tool 'sonar-scanner'
        JAVA_HOME = tool 'openjdk-11'
      }
      steps {
        withSonarQubeEnv('sonarqube') {
          sh "sed -i 's/%BUILD_NUMBER%/${BUILD_NUMBER}/g'
              sonar-project.properties"
          sh "${sonarHome}/bin/sonar-scanner"
        }

      }
    }

    stage('Quality Gate') {
      steps {
        waitForQualityGate true
        echo '--- QualityGate Passed ---'
      }
```

```
    }

    stage('Build Docker Image'){
      environment {
        dockerHome = tool 'docker'
      }
      steps{
        sh "${dockerHome}/bin/docker build -f Dockerfile.app
            -t valen97/calculadora-angular:${BUILD_NUMBER} ."
      }
    }

    stage('Publish Docker Image'){
      environment {
        dockerHome = tool 'docker'
        dockerHub = credentials('VsDockerHub')
      }
      steps {
        sh '${dockerHome}/bin/docker login
            -u $dockerHub_USR
            -p $dockerHub_PSW'
        sh '${dockerHome}/bin/docker push
             valen97/calculadora-angular:${BUILD_NUMBER}'
        sh '${dockerHome}/bin/docker rmi
             valen97/calculadora-angular:${BUILD_NUMBER}'
        sh '${dockerHome}/bin/docker logout'
      }
    }
  }

  post{
    success {
      build(job: 'calculadora-angular-deploy',
      parameters: [
        string(name: 'BUILD_NUMBER', value: "${BUILD_NUMBER}")
      ])

      mail to: "valentina_17_01@hotmail.com",
      subject: "EXITOSA ejecución de la pipeline '${JOB_NAME}'",
      body: """
      * Estado de ejecución: ${currentBuild.result}
      * Número de ejecución: ${BUILD_NUMBER}
      * URL para visualizar los logs de la ejecución: ${env.BUILD_URL}"""
    }
    failure{
      mail to: "valentina_17_01@hotmail.com",
      subject: "FALLIDA ejecución de la pipeline '${JOB_NAME}'",
      body: """
```

```
      * Estado de ejecución: ${currentBuild.result}
      * Número de ejecución: ${BUILD_NUMBER}
      * URL para visualizar los logs de la ejecución: ${BUILD_URL}"""
    }
  }
}
```

A8.2. Jenkinsfile-deploy

```
pipeline {

  agent { dockerfile true }

  stages {
    stage('Deploy Dev') {
      steps {
        withCredentials(bindings:
        [azureServicePrincipal('azuredevops_dev')]) {
          sh 'sed -i "s/%BUILD_NUMBER%/${BUILD_NUMBER}/g"
              kubernetes/deployment.yml'
          sh 'az login --service-principal
              -u $AZURE_CLIENT_ID
              -p $AZURE_CLIENT_SECRET
              -t $AZURE_TENANT_ID'
          sh 'az aks get-credentials
              --resource-group $RESOURCE_GROUP
              --name $CLUSTER_NAME'
          sh 'kubectl apply -k kubernetes/.'
        }
      }
    }
  }

  parameters {
    string(name: 'RESOURCE_GROUP',
           defaultValue: 'SOCIUSRGLAB-RG-MODELODEVOPS-AKS',
           description: 'Grupo de Recursos')
    string(name: 'CLUSTER_NAME',
           defaultValue: 'ModeloDevOps-AKS',
           description: 'Nombre del App Service')
    string(name: 'BUILD_NUMBER',
           defaultValue: '',
           description: 'Versión imagen Docker')
  }
}
```

Referencias

[Adhithi, 2023] Adhithi (2023). 38 best ci/cd tools you should not miss in 2023. https://testsigma.com/blog/ci-cd-tools/. Recuperado el 03/03/2023.

[Afreen, 2023] Afreen, S. (2023). What is a dockerfile? a step-by-step guide [2023 updated]. https://www.simplilearn.com/tutorials/docker-tutorial/what-is-dockerfile. Recuperado el 04/01/2023.

[Altexsoft, 2022] Altexsoft (2022). Ci/cd tools comparison: Jenkins, teamcity, bamboo, travis ci, and more. https://www.altexsoft.com/blog/engineering/cicd-tools-comparison/. Recuperado el 03/03/2023.

[Angular, 2023] Angular (2023). Introducción a la documentación de angular. https://docs.angular.lat/docs. Recuperado el 24/01/2023.

[Angulo, 2018] Angulo, J. (2018). Entendiendo devops en 5 minutos. https://www.autentia.com/2018/08/17/entendiendo-devops-en-5-minutos/.

[Arundel y Domingus, 2019] Arundel, J. y Domingus, J. (2019). *Cloud Native DevOps with Kubernetes: building, deploying, and scaling modern applications in the Cloud*. O'Reilly Media.

[BasuMallick, 2022] BasuMallick, C. (2022). Best ci/cd tools 2022. https://www.spiceworks.com/tech/devops/articles/best-cicd-tools/. Recuperado el 03/03/2023.

[Beck *et al.*, 2005] Beck, K., Beedle, M., y Grenning, J. (2005). Principios del manifiesto Ágil. https://agilemanifesto.org/iso/es/principles.html. Recuperado el 03/01/2023.

[Briffa, 2023] Briffa, L. (2023). Angular calculator. https://codepen.io/LewisBriffa/pen/RRgaqm. Recuperado el 04/03/2023.

[BSmO21, 2022] BSmO21 (2022). File:cloud computing service models (1).png - wikimedia commons. https://commons.wikimedia.org/wiki/File:Cloud_computing_service_models_%281%29.png. Recuperado el 10/01/2023.

[Carreira, 2005] Carreira, B. (2005). *Lean manufacturing that works: powerful tools for dramatically reducing waste and maximizing profits*. AMACOM/American Management Association.

[ClickIT, 2022] ClickIT, D. . S. D. (2022). Kubernetes architecture diagram. https://images.clickittech.com/2020/wp-content/uploads/2022/04/13202329/Diagram-55.jpg. Recuperado el 04/02/2023.

[Digital.ai, 2022] Digital.ai (2022). 16th annual state of agile report. https://digital.ai/resource-center/analyst-reports/state-of-agile-report/. Recuperado el 04/02/2023.

[Docker Inc., 2023] Docker Inc. (2023). Docker docs. https://docs.docker.com/. Accessed: 20/01/2023.

[Dowling, 2022] Dowling, L. (2022). The 10 best code quality tools. https://linearb.io/blog/the-best-code-quality-tools/. Recuperado el 03/03/2023.

[Duvall *et al.*, 2007] Duvall, P. M., Matyas, S., y Glover, A. (2007). *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[Foresight, 2022] Foresight (2022). The ci/cd war of 2022: A look at the most popular ci/cd tools. https://www.runforesight.com/blog/the-ci-cd-war-of-2022-a-look-at-the-most-popular-ci-cd-tools. Recuperado el 03/03/2023.

[Fowler, 2013] Fowler, M. (2013). Continuousdelivery. https://martinfowler.com/bliki/ContinuousDelivery.html. Recuperado el 08/01/2023.

[Fowler y Foemmel, 2006] Fowler, M. y Foemmel, M. (2006). Continuous integration.

[G2, 2023] G2 (2023). Best static code analysis tools. https://www.g2.com/categories/static-code-analysis. Recuperado el 03/03/2023.

[Humble y Farley, 2010] Humble, J. y Farley, D. (2010). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

[Jenkins, 2023] Jenkins (2023). Using docker with pipeline. https://www.jenkins.io/doc/. Recuperado el 04/23/2023.

[Jovanović, 2020] Jovanović, M. (2020). Jenkins architecture and pipeline. https://blog.syncitgroup.com/jenkins-architecture-and-pipeline/. Recuperado el 04/15/2023.

[Katalon, 2023] Katalon (2023). Best 14 ci/cd tools you must know — updated for 2023. https://katalon.com/resources-center/blog/ci-cd-tools. Recuperado el 03/03/2023.

[Kim *et al.*, 2014] Kim, G., Behr, K., y Spafford, K. (2014). *The phoenix project: A novel about IT, DevOps, and helping your business win*. IT Revolution.

[Kim *et al.*, 2021] Kim, G., Humble, J., Debois, P., Willis, J., y Forsgren, N. (2021). *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations*. IT Revolution.

[Kostiantyn, sf] Kostiantyn, L. (s.f.). Agile lifecycle development process diagram, software developers sprints infographic. https://www.dreamstime.com/agile-lifecycle-development-process-diagram-software-developers-sprints-infographic-methodology-three-fading-analysis-image182288404.

[Kubernetes, 2023] Kubernetes (2023). Kubernetes documentation. https://kubernetes.io/docs/home/. Recuperado el 04/23/2023.

[Martin, 2023] Martin, M. (2023). 19 best static code analysis tools (2023. https://www.guru99.com/best-static-code-analysis-tools.html. Recuperado el 03/03/2023.

[Matthias y Kane, 2018] Matthias, K. y Kane, S. P. (2018). *Docker: Up & Running: Shipping Reliable Containers in Production 2nd Edition*. .ºŔeilly Media, Inc.”.

[Microsoft Azure, 2023] Microsoft Azure (2023). What is cloud computing? — microsoft azure. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing. Recuperado el 24/01/2023.

[Mistry, 2022] Mistry, P. (2022). What is infrastructure as code? best practices, tools and benefits! https://radixweb.com/blog/what-is-infrastructure-as-code-and-its-best-practices. Recuperado el 04/01/2023.

[Morris, 2020] Morris, K. (2020). *Infrastructure as code*. O’Reilly Media.

[Motoskia, 2018] Motoskia (2018). How to add jenkins slave to master. https://foxutech.com/how-to-add-jenkins-slave-to-master/. Recuperado el 04/15/2023.

[Nakivo, 2019] Nakivo (2019). Kubernetes vs docker – which one should you use? https://www.nakivo.com/blog/docker-vs-kubernetes/. Recuperado el 04/01/2023.

[Naur y Randell, 1969] Naur, P. y Randell, B. (1969). Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th-11th october 1968.

[Olsina, 1999] Olsina, L. (1999). Metodología cuantitativa para la evaluación y comparación de la calidad de sitios web. *Facultad de Ciencias Exactas, Universidad de la Plata (Argentina). Tesis doctoral*.

[PeerSpot, 2023] PeerSpot (2023). Best static code analysis software. https://www.peerspot.com/categories/static-code-analysis. Recuperado el 03/03/2023.

[Ranjith, 2022] Ranjith (2022). Angular 12 calculator application with source code - coding diksha. https://codingdiksha.com/angular-calculator-application-source-code/. Recuperado el 04/03/2023.

[Ruiz, 2015] Ruiz, C. J. (2015). El legendario origen del movimiento devops. https://www.paradigmadigital.com/techbiz/el-legendario-origen-del-movimiento-devops/.

[Saito *et al.*, 2017] Saito, H., Lee, H.-C. C., y Wu, C.-Y. (2017). *DevOps with Kubernetes: accelerating software delivery with container orchestrators*. Packt Publishing Ltd.

[Sheth, 2022] Sheth, H. (2022). 38 best ci/cd tools for 2022. https://www.lambdatest.com/blog/best-ci-cd-tools/. Recuperado el 03/03/2023.

[Singh, 2023] Singh, G. (2023). Infrastructure as code (iac) tools to boost your productivity in 2023. https://www.xenonstack.com/blog/infrastructure-as-code-tools. Recuperado el 04/02/2023.

[SoftwareTestingHelp, 2023] SoftwareTestingHelp (2023). 12 best code quality tools for error free coding in 2023. https://www.softwaretestinghelp.com/code-quality-tools/. Recuperado el 03/03/2023.

[Sommerville, 2007] Sommerville, I. (2007). *Ingeniería de Software*. Pearson Studium.

[SonarSource, 2023] SonarSource (2023). Sonarqube documentation. https://docs.sonarqube.org/latest/. Recuperado el 04/23/2023.

[Taylor, 2023] Taylor, D. (2023). 20 best ci/cd tools. https://www.guru99.com/top-20-continuous-integration-tools.html. Recuperado el 03/03/2023.

[The Maritime Executive, 2016] The Maritime Executive (2016). The story of malcom mclean. https://maritime-executive.com/article/the-story-of-malcolm-mclean. Recuperado el 15/01/2023.

[The Standish Group International, 1995] The Standish Group International, I. (1995). The chaos report. https://cs.franklin.edu/~smithw/ITEC495_Resources/chaos%20report.pdf. Recuperado el 04/02/2023.

[TrustRadius, 2023] TrustRadius (2023). List of top static code analysis tools 2023. https://www.trustradius.com/static-code-analysis. Recuperado el 03/03/2023.

[Turnbull, 2018] Turnbull, J. (2018). *The docker book*.

[Wikipedia, 2022] Wikipedia (2022). Hediondez del código - wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Hediondez_del_c%C3%B3digo. Recuperado el 05/07/2023.

[Xataka, sf] Xataka (s.f.). Estamos cargando demasiado los barcos de contenedores y esto está ralentizando aún más todo. https://www.xataka.com/empresas-y-economia/estamos-cargando-demasiado-barcos-contenedores-esto-esta-ralentizando-todo.

[Yuanyi, 2021] Yuanyi (2021). Simplifying knative: Application deployment and access - alibaba cloud community. https://www.alibabacloud.com/blog/simplifying-knative-application-deployment-and-access_597430. Recuperado el 10/01/2023.

[Zelleke, 2023] Zelleke, L. (2023). 6 best static code analysis tools for 2023. https://www.comparitech.com/net-admin/best-static-code-analysis-tools/. Recuperado el 03/03/2023.

[Zhang, 2023] Zhang, M. (2023). Top 10 cloud service providers globally in 2023. https://dgtlinfra.com/top-10-cloud-service-providers-2022/. Recuperado el 04/03/2023.