Universidad ORT Uruguay

Facultad de Ingeniería

Fundamentos de entrega continua y tecnologías para pipelines de desarrollo de software

Entregado como requisito para la obtención del título de Licenciado en Ingeniería de Software

Santiago Toscanini - 232566

Nahuel Biladóniga - 211138

Sofía Tejerina - 209239

Tutor: Martin Solari

2022

Declaraciones de autoría

Nosotros, Santiago Toscanini, Sofía Tejerina, Nahuel Biladóniga, declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el Trabajo de Investigación;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Sofía Tejerina	Nahuel Biladóniga	Santiago Toscanini
8 de Febrero del 2022	8 de Febrero del 2022	8 de Febrero del 2022

Abstract

La entrega continua, es una práctica que intenta reducir el tiempo entre que se realiza un cambio en el código fuente y el momento que llega al cliente en producción. Para reducir este tiempo se utilizan automatizaciones en las etapas de desarrollo, tales como build, testing y deployment.

El objetivo general de este proyecto es el estudio de las bases conceptuales de la entrega continua y de las tecnologías para implementar pipelines de desarrollo de software, así como su impacto en la calidad. Para esto analizamos los principales conceptos relacionados: calidad, entrega continua, DevOps, integración continua y despliegue continuo.

Se realizó una prueba de concepto con el fin de probar las principales tecnologías relacionadas, sometiéndolas a un flujo ya definido de pruebas y a un análisis de aspectos que consideramos importantes para evaluar de forma objetiva el impacto en la calidad de software y el esfuerzo de implementación de cada resultado. La prueba consistió en la implementación de un pipeline de integración continua y despliegue continuo desde el desarrollo hasta que llega al usuario.

Los principales resultados obtenidos fueron un impacto real en la forma de desarrollo que nos permite, a medida que se incorporan los cambios, verificar que se cumple con estándares de calidad de código, testing, seguridad y como resultado de esto, modificabilidad. Todo esto se logró con un moderado esfuerzo de configuración con herramientas respaldadas por empresas y comunidades, en una capa gratuita suficiente para un desarrollo básico.

Palabras clave

DevOps, continuos delivery, entrega continua, continuous integration, integración continua, continous deployment, despliegue continuo.

Índice

1 Introducción **10**

2 Marco conceptual **13**

2.1 Atributos de calidad 13

2.1.1 Modificabilidad 13

2.1.2 Testeabilidad 14

2.1.3 Capacidad de despliegue 14

2.1.4 Prácticas de aseguramiento de calidad 16

2.2 Testing 17

2.2.1 Niveles de testing 18

2.2.2 Automatización de tests 19

2.2.3 Tipos de desarrollo basado en testing 20

2.3 DevOps 21

2.3.1 Origen 21

2.3.2 Site Reliability Engineering 22

2.4 Entrega continua 23

2.4.1 Definiciones 23

2.4.2 Prerrequisitos 24

2.4.3 Impacto en la calidad 25

2.5 Integración continua 25

2.5.1 Definiciones 26

2.5.2 Prerrequisitos 27

2.6 Despliegue continuo 29

2.6.1 Definiciones 29

2.7 Métricas de entrega continua 29

2.7.1 Tiempo de ciclo 30

2.7.2 Time To Value 30

2.7.3 Uptime 31

2.7.4 Calidad de código 31

2.7.5 Métricas relacionadas al desempeño 32

2.7.6 Métricas DORA 32

2.7.7 Métricas de SRE 33

2.8 Patrones y anti patrones 33

2.8.1 Patrones 33

2.8.2 Anti patrones 36

3 Comparación de tecnologías para pipelines de desarrollo 38

3.1 Control de versiones 38

3.1.1 Git 39

3.1.2 Comparación de gestores de repositorios en la nube 40

3.2 Contenerización 42

3.2.1 Docker 42

3.2.2 Kubernetes 46

3.3 Monitoreo 48

3.3.1 Prometheus 48

3.3.2 Datadog 49

3.3.3 New Relic 49

3.4 Manejo de configuraciones 50

3.4.1 Chef 51

3.4.2 Puppet 52

3.4.3 Impacto en los ambientes 53

3.5 Herramientas de calidad 54

3.5.1 Code linters 54

3.5.2 Code formatters 55

3.5.3 Security checks 55

3.6 CI 56

3.6.1 Pioneros 56

3.6.2 Recientes 57

3.6.3 Conceptos 59

3.7 CD 61

3.7.1 Herramientas de código abierto 61

3.7.2 Proveedores de servicios en nube 61

4 Prueba de concepto de prácticas y tecnologías 63

4.1 Objetivos 63

4.2 Proyecto de prueba de concepto 64

4.3 Mapa sobre el estudio de las tecnologías 66

4.3.1 Criterio de evaluación de herramientas 66

4.3.2 Flujo del pipeline 67

4.4 Desarrollo de prueba de concepto 69

4.4.1 Cobertura de código 69

4.4.2 Testing 70

4.4.3 GitHub Actions 71

4.4.4 Jenkins 72

4.4.5 CircleCI 74

4.4.6 Code Deploy 76

4.5 Resultados 78

4.5.1 Comparación entre las tecnologías de CI 78

4.5.2 Observaciones 84

4.5.3 Reflexión 87

5 Conclusiones **89**

5.1 Evaluación del impacto en la calidad 90

5.1.1 Lecciones aprendidas 91

5.1.2 Mejoras del pipeline CI/CD 91

5.2 Líneas futuras de investigación 92

Referencias bibliográficas **93**

Anexo **99**

1. Introducción

La creación de software es un proceso que se encuentra muy acoplado a los recursos humanos; además de su dependencia con las limitantes de la tecnología, esta misma fue mejorando de forma exponencial gracias a nuevos avances: los circuitos integrados, los ordenadores personales, la fibra óptica, las redes inalámbricas, los dispositivos móviles, entre muchos otros. Pero mientras la tecnología crece a pasos agigantados el factor humano debe adecuar su forma de trabajar para poder obtener lo mejor de estos nuevos avances.

Para mejorar el desarrollo a la par que se mejoraba la tecnología, y así poder llevar a los usuarios finales productos de calidad, es que han surgido las distintas metodologías de trabajo. Originalmente se utilizaban metodologías que hoy llamamos "tradicionales" o "clásicas" para organizar el trabajo desde las necesidades del usuario hasta que el producto llega al cliente. Estas se basan en definir los requerimientos al comienzo del proyecto por única vez y dándole resultados al cliente una vez terminado el desarrollo. Pero las necesidades y expectativas de los usuarios han ido evolucionando al igual la tecnología, y ahora para cumplir los estándares de calidad necesitamos evolucionar las metodologías.

La calidad está definida por un conjunto de atributos medibles que un sistema debe cumplir para satisfacer las necesidades del usuario.[5] Esta se construye en forma incremental, desde adentro en cada etapa del desarrollo y es compartida por todos los equipos.

Surgen las metodologías ágiles, para darle al usuario un sistema que se crea y va

evolucionando acorde a sus necesidades, cumpliendo con estándares de calidad: seguridad, disponibilidad, rendimiento y estabilidad, en resumen, sistemas que se adaptan a los cambios según las necesidades de los usuarios. Los pioneros del uso de estas metodologías crearon el manifiesto para el desarrollo de software ágil, mejor conocido como "Manifiesto ágil", estableciendo los valores y principios de estos procesos ágiles, siendo el primer principio lo que nos lleva al desarrollo de esta investigación: *"Nuestra máxima prioridad es la satisfacción del cliente a través de una entrega temprana y continua de software valioso."*.[23]

Siguiendo lo que dice el Manifiesto ágil, se hace uso de las prácticas de entrega continua brindando entregas en períodos cortos de tiempo, que para realizarlas asegurándonos de mantener la calidad, nos apoyamos en prácticas como integración continua (CI a partir de ahora por sus siglas en inglés: Continuous Integration) y despliegue continuo (CD a partir de ahora por sus siglas en inglés: Continuous Deployment), bajo una cultura de DevOps. A lo largo de este documento hablaremos de entrega continua como el concepto base que es implementado mediante el uso de integración continua, despliegue continuo y DevOps, para evitar confusiones siempre que usemos las siglas CD nos estaremos refiriendo a despliegue continuo.

Entrega continua consiste en desarrollar software de forma tal que este pueda ser enviado a producción en cualquier momento, agilizando el ciclo de tiempo entre que se planifica una funcionalidad y esta llega al cliente. Para lograr esto se necesita que los equipos involucrados; el equipo de desarrollo y el de operaciones trabajen juntos, comprometidos con el cumplimiento de los atributos de calidad, esto es en lo que se basa el concepto de DevOps. También se utilizan prácticas como integración continua, que es la práctica de integrar continuamente el trabajo buscando consistencia en el resultado producido por los desarrolladores, para que integrar cada nuevo cambio no produzca problemas. Por último despliegue continuo consiste en realizar un pipeline que genere un compilado de la aplicación y se envíe al ambiente de producción de la manera más automática posible.

Este proyecto tiene como objetivo el estudio de las bases conceptuales de entrega

continua y de las tecnologías para implementar pipelines. Como actividades del proyecto se definen generar un marco conceptual que abarque todos los conceptos relacionados a la entrega continua y brinde el conocimiento base para poder implementar un ciclo de CI/CD lo más completo posible. Luego se realizará un análisis de las tecnologías disponibles y el estado del arte tecnológico. Se tomará el análisis producido para llevar a cabo el desarrollo de un pipeline completo con las tecnologías elegidas, este incluirá un ciclo básico de CI/CD utilizando como herramienta de aseguramiento de calidad testing unitario. Finalmente se presentarán los resultados obtenidos: una comparación entre aspectos que consideramos relevantes sobre cada tecnología y conclusiones obtenidas sobre la experiencia de uso e implementación de cada una.

2. Marco conceptual

Durante este capítulo iremos introduciendo los principales conceptos que debemos conocer para entender el contexto de la investigación.

2.1 Atributos de calidad

Una perspectiva importante en la generación de sistemas de calidad es el incremento de nuevas funcionalidades y modificación de las funcionalidades existentes en poco tiempo manteniendo instancias de retroalimentación continua, a nivel interno en el equipo y a nivel externo con los clientes. Un pipeline es un conjunto de procesos automatizados que permiten a los desarrolladores implementar un ciclo de integración continua y despliegue continuo.

En este documento nos vamos a centrar en los aspectos de la calidad que se encuentran relacionados con el concepto de entrega continua, la cultura DevOps y las prácticas de CD.

2.1.1 Modificabilidad

Los cambios son una constante en el software que comienza con la primer salida a producción del sistema hasta que se vuelve obsoleto. El cambio es impulsado por la necesidad de solucionar errores del sistema, para realizar mejoras en las funcionalidades, mejorar la experiencia del usuario y mejorar las tecnologías empleadas

manteniendo actualizado el sistema.[5]

2.1.2 Testeabilidad

La testeabilidad en el software es la capacidad del sistema de mostrar sus propias fallas, la premisa es encontrar fallas durante el período de uso local antes de que llegue a producción.[5]

2.1.3 Capacidad de despliegue

También conocido como deployability, nos habla sobre el "deployment time"; el tiempo que demora en llegar a producción el código luego de ser realizado, así como también del efecto del despliegue en otros aspectos como la seguridad y escalabilidad. Tanto la seguridad como la escalabilidad eran mayormente responsabilidad del equipo de infraestructura, pero en el último tiempo han cobrado más importancia para el equipo de desarrollo por el aumento en la demanda de los usuarios.

"Existen dos factores que afectan el tiempo de implementación del código. En primer lugar, se encuentra el tiempo de coordinación que se requiere entre las diversas partes interesadas antes de realizar realmente el código. El segundo es el procesamiento del nuevo código a través de una cadena de herramientas hasta la producción."[6]

Para cumplir con este atributo de calidad debemos definir un plan de puesta en producción con los interesados en el sistema, procurar que no haya errores de consistencia entre el código nuevo y el existente; registrar y manejar los errores que surjan, asegurarse de que las habilidades y los conocimientos se transfieran al personal de operaciones.[5]

DevOps es un movimiento que ha propuesto una serie de prácticas destinadas a cumplir estos requisitos. El uso de CI/CD ponen en práctica acciones que promueven el uso de herramientas para el cumplimiento de este atributo.

Seguridad

Seguridad se define como: *"El grado el cual un producto o sistema protege la información y datos, por lo tanto la persona, producto o sistema tiene un grado de acceso apropiado según su tipo y nivel de autorizacion".*[5] Para lo cual se basa en tres principios fundamentales: confiabilidad, integridad y disponibilidad (CIA por su sigla en inglés), los cuales detallaremos a continuación.

- **Confiabilidad:** Es la propiedad de los datos o servicios de estar protegidos de acceso no autorizado.[5]
- **Integridad:** Es la propiedad que permite que los datos no sean manipulados por agentes no autorizados.[5]
- **Disponibilidad:** Refiere a que el sistema estará disponible para ser usado por usuarios legítimos.[5]

También existen otras características complementarias:

- **Autenticación:** Verifica la identidad de los involucrados en una transacción.
- **No repudio:** Garantiza que el productor del mensaje no puede negar haber enviado dicho mensaje.
- **Autorización:** Brinda al usuario los privilegios para poder realizar una tarea.

Escalabilidad

En [5] se define el concepto de escalabilidad como: *"La escalabilidad es el atributo de un sistema de incrementar la capacidad y funcionalidades basada en la demanda de los usuarios. El software escalable permite mantenerse estable mientras se adapta a los cambios, actualizaciones y reducción de recursos".*

2.1.4 Prácticas de aseguramiento de calidad

Pull Request

Cuando hablamos de Pull Request, al cual nos referiremos a partir de ahora con las siglas PR, nos referimos a la acción o petición que hace un usuario para integrar un nuevo cambio en una unidad de un sistema, es el origen de un cambio [58]. De esta surge la revisión de código, aunque esta última no es obligatoria para CI, aporta valor en términos de una calidad mínima que debe de cumplir nuestro código. El análisis de código se puede automatizar con herramientas como linters o parsers.

Linters

Es una de las prácticas de calidad de código más usadas, se define como: *"Una herramienta de análisis de código que permite indicar errores de programación, errores de estilo o construcciones sospechosas".*[6]

Son principalmente útiles al momento de programar en lenguajes no compilados, dado que permiten conocer previamente si existe un error antes de la ejecución. Algunos de los beneficios de los Linters son:

- Encontrar errores de escritura.
- Encontrar errores de sintaxis.
- Encontrar variables no declaradas.
- Encontrar llamadas a funciones indefinidas.
- Encontrar funciones obsoletas.
- Poder seguir estándares de codificación y buenas prácticas.

2.2 Testing

"Es una actividad desarrollada para evaluar la calidad del producto, y para mejorarlo al identificar defectos y problemas. El testing de software consiste en la verificación dinámica del comportamiento de un programa sobre un conjunto finito de casos de prueba, apropiadamente seleccionados a partir del dominio de ejecución que usualmente es infinito, en relación con el comportamiento esperado".[25]

En los ciclos de entrega continua el uso de testing automatizado en sus distintos niveles nos permite verificar que el sistema cumple con los requerimientos y con la calidad necesaria para satisfacer las expectativas del usuario. Es un concepto amplio en el cual buscamos entrar más en detalle comparado con el resto de atributos de calidad mencionados anteriormente, en este capítulo entraremos más en detalle sobre sus niveles y su impacto en el desarrollo.

Tener un código fácil de testear implica cumplir con ciertos atributos de calidad y ciertas métricas de diseño. Una buena arquitectura contribuye fuertemente a lograr este propósito, como sabemos la arquitectura esta definida en base a los requisitos de calidad solicitados por los distintos stakeholders.[5]

El testing aplica a distintas capas del sistema, comenzando con la capa más atómica como son los tests unitarios y subiendo hasta probar el sistema por completo. La pirámide de Cohn muestra la estrategia de testing de una aplicación. La base sobre la que se debe asentar nuestro testing y lo más abundante deben ser los tests unitarios, luego a medida que subimos en la pirámide se reduce la cantidad de tests y el esfuerzo dedicado a estos.[17]

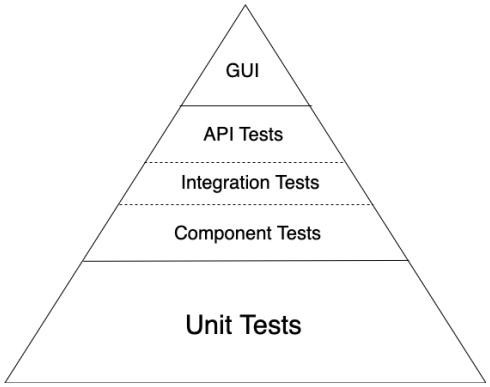

Figure 1: Pirámide de Cohn [17]

2.2.1 Niveles de testing

En esta sección nos centraremos en tres niveles de testing que son necesarios para abarcar todos los aspectos importantes de nuestro sistema; testing unitario, testing de integración y testing de aceptación.

Testing unitario

Son los tests que prueban la unidad de código funcional más pequeña de nuestro sistema; un único método, clase o función, en un ambiente aislado. Esto permite que el desarrollador este seguro que su código funciona como espera, en un alcance reducido.

Testing de integración

Busca poder probar que la aplicación se comunique con servicios externos correctamente. Los tests de integración son ejecutados luego que los tests unitarios hayan sido validados. Debido a que son costosos de ejecutar se busca que los errores sean detectados en etapas anteriores.[2]

Testing de aceptación

Se busca probar la aplicación a un nivel completo para poder probar las funcionalidades como fueron diseñadas y asegurarse que ningún error nuevo haya sido introducido. La diferencia en palabras de Humble y Farley es:

"El propósito de un test unitario es mostrar que una parte de la aplicación hace lo que el programador quiere [...]. El objetivo del test de aceptación es probar que haga lo que el cliente quiere, no lo que el programador cree."[2]

Esta prueba se puede llevar a cabo en cualquiera de los niveles anteriores, ya que consiste en ejecutar las pruebas del sistema y verificar si han aparecido fallas. [5]

2.2.2 Automatización de tests

Al automatizar pruebas de software se persigue el objetivo de simplificar el trabajo dispendioso, repetitivo o complejo, haciéndolo efectivo y productivo. Aunque existen diferencias entre pruebas automatizadas y pruebas manuales, no son mutuamente excluyentes, se usan en complemento. Inicialmente el costo de realizar pruebas manuales es poco y su desarrollo es sencillo; a medida que el sistema de consolida se vuelve necesario aplicar pruebas automáticas que abarquen aspectos más concretos de este.2 En consecuencia, el costo también sube para las pruebas manuales con el tiempo pero el costo de automatizar las pruebas se va disminuyendo como se ve en el gráfico.

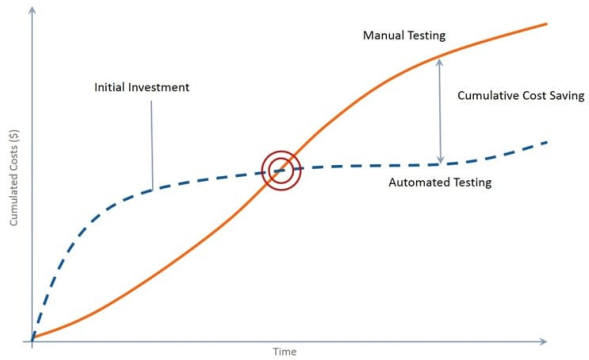

Figure 2: Costos de test manual vs automatizado [26]

2.2.3 Tipos de desarrollo basado en testing

Desarrollo basado en pruebas

También conocido por sus siglas en inglés TDD (Test Driven Development), es una técnica que consiste en construir software guiado por el testing. Para lograr esto hay que hacer los siguientes tres pasos a lo largo de todo el desarrollo:

- Escribir un test para la funcionalidad a agregar.
- Escribir el código necesario para que el test pase.
- Hacer mejoras del código viejo y nuevo hasta que quede bien estructurado.

Este estilo de codificación consigue que el código obtenido sea testeable y al hacer el test primero fuerza a pensar en las interfaces a implementar; ayudando a separar la interfaz (abstracción) de la implementación (concreta).

Desarrollo guiado por comportamiento

Mejor conocido por sus siglas en inglés BDD (Behaviour Driven Development), surge como una alternativa a TDD. El principal problema de TDD es que está centrado en el código y no es fácil de entender para una persona no técnica. BDD descrito por Dan North [19] propone un cambio de ese paradigma.

Se busca que las pruebas estén armadas como una historia, en la cual cada historia tiene una descripción indicando de qué se va a tratar y dentro de la misma los distintos flujos que sucederán. Estas ocurrencias son denominadas sentencias, las cuales describen un comportamiento de la prueba. La principal ventaja de BDD es que permite encontrar errores rápidamente y plantea en marco de solución para estos:

- Se introdujo un nuevo error. Solución: corregir el error.

- El comportamiento se sigue queriendo pero esta en otra parte. Solución: Mover el test hasta que se solucione.
- El comportamiento ya no se necesita. Solución: Borrar el test.

2.3 DevOps

El objetivo de la entrega continua consiste en agilizar el ciclo de tiempo entre que se planifica una funcionalidad y esta llega finalmente al cliente. Este ciclo como se puede ver en la imagen 3 comienza con la planificación del cambio y termina con el monitoreo de este una vez ya se encuentra en producción. Para acelerar el proceso manteniendo el estándar de calidad de nuestro sistema necesitamos lograr que los equipos involucrados en el ciclo trabajen juntos, esa es la esencia del concepto de DevOps; generar un entendimiento compartido entre el equipo de desarrollo y el de operaciones. En este capitulo detallaremos en el origen de este cambio cultural y una práctica popular hoy en día que se considera una implementación de DevOps.

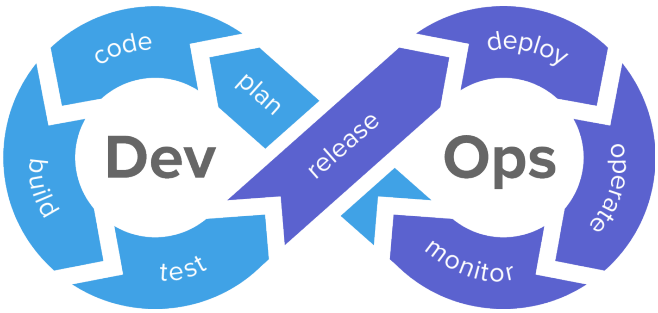

Figure 3: Flujo realizado de DevOps.

2.3.1 Origen

Con las metodologías ágiles el tiempo de desarrollo baja de años a semanas o meses, pero llevar la implementación a producción todavía continúa requiriendo un gran costo de tiempo. Para solucionar este problema llega DevOps en el 2009, junto con las mejoras del hardware, software y la nube, las implementaciones se pueden

poner en producción en cuestión de horas o minutos.[67] Esto brinda la posibilidad de realizar actividades que complementan a mejorar la calidad, se toman riesgos haciendo pruebas de concepto con funcionalidades que se implementan rápidamente y se ponen en producción. Al tener ciclos de retroalimentación rápidos se pueden ver rápidamente los efectos de los cambios. Se llevan a cabo pruebas de fallas del sistema para garantizar la disponibilidad y seguridad de este.[70]

La mejora en la calidad del producto se puede ver claramente en lo rápido que se pueden detectar fallas, posteriormente solucionarlas y volver a poner en producción. Además generan un aprendizaje genuino y constante que antes no se lograba porque los errores se detectaban muchos meses después del desarrollo.[2]

Los objetivos de desarrollo y de operaciones suelen estar contrapuestos los unos de los otros. El desarrollo se encarga de responder rápidamente, implementando funcionalidades tan pronto como se pueda, y las operaciones serán responsables de brindar un servicio confiable, estable y seguro, lo que dificulta la introducción de cambios en producción que pongan la estabilidad en peligro. Esta es la base del conflicto entre ambas partes, conflicto que se suaviza con el concepto de DevOps.[2]

DevOps es un cambio cultural. No se trata simplemente de adoptar una planificación ágil, pruebas automatizadas o entrega continua, aunque esas prácticas son ciertamente importantes. La cultura DevOps se trata de un entendimiento compartido entre los desarrolladores (Dev) y las operaciones (Ops), y de compartir la responsabilidad del software que crean. Eso significa aumentar la transparencia, la comunicación y la colaboración en el desarrollo, las operaciones y el negocio.

2.3.2 Site Reliability Engineering

Ingeniería de confiabilidad de sitio, de ahora en adelante SRE por sus siglas en inglés: Site Reliability Engineering, es un conjunto de prácticas y principios relacionados a problemas de infraestructura y operaciones. Estas prácticas son consideradas una implementación específica de DevOps.[11][60]

SRE busca simplificar la estructura tecnológica, logrando que estos componentes sean simples y se puedan aplicar las mejores prácticas. Existen tres grupos, estos son:

- **SLA (Service Level Agreement):** Define las propiedades fiabilidad, rendimiento y latencia del sistema percibidas por el usuario final.
- **SLO (Service Level Objectives):** Son los objetivos definidos para cumplir por el equipo.
- **SLI (Service Level Indicator):** Son los indicadores y métricas que serán medidas en base a los objetivos planteados.

2.4 Entrega continua

En esta sección vamos a introducir la definición formal de entrega continua, qué prácticas abarca, algunos requisitos que nos permitirán implementarla de la forma adecuada y por último hablaremos sobre su impacto en la calidad.

2.4.1 Definiciones

Es un paradigma del desarrollo de Software, el cual busca responder a la pregunta: *"Si existe una buena idea, como podemos entregársela a los usuarios de la manera mas rápida posible?"*.[1]

Por lo tanto este concepto se define como una disciplina del desarrollo de software que busca construirlo de manera tal que este pueda ser enviado a producción en cualquier momento.[18] Si bien, buscamos acelerar el tiempo que demora en hacerse un flujo completo de entrega continua mediante automatización en las etapas, pero esto no quiere decir que no pueden haber pasos manuales durante el CI o el CD, por ejemplo, es muy común el uso de revisión de código manual para poder incorporar cambios al sistema.

Se logra esto mediante la integración continua del software realizado por el equipo de desarrollo, armando ejecutables y corriendo una serie de tests automatizados que permitan detectar problemas. Las pruebas se ejecutan en ambientes similares a producción para poder asegurar su correcto funcionamiento. Dichos tests corresponden a los niveles definidos previamente; tests unitarios, integración y aceptación.

En el siguiente diagrama 4 se muestra un ejemplo de un flujo de completo de entrega continua, en él se detallan algunas actividades importantes para el aseguramiento de la calidad que detallaremos a lo largo de este documento: distintos niveles de testing, uso de TDD o BDD, security checks, linters, entre otros.

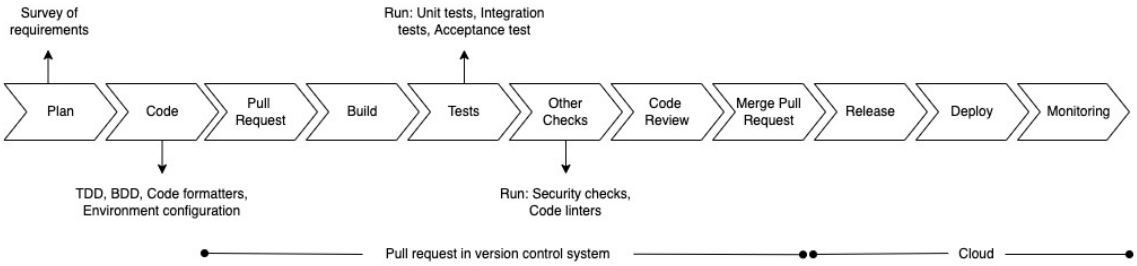

Figure 4: Ejemplo de pipeline completo de entrega continua

2.4.2 Prerrequisitos

- **Construir una sola vez:** Cada producto generado debería ser único y conciso cada vez que es construido.
- **Automatizar cada deploy:** Se deberá generar un archivo o conjunto de archivos que describan paso a paso las instrucciones para generar el deploy.
- **Separar ambientes:** Cada ambiente deberá contener configuraciones para su correcto funcionamiento pero estas configuraciones deberán estar separadas de las que son pertinentes para el uso de la aplicación.
- **Realizar mantenimiento del pipeline:** Como ocurre con CI, para poder mantener un desarrollo constante y fluido todos los miembros del equipo deberán adoptar las practicas y realizar evaluaciones periódicas (o frente a un

cambio en los requerimientos) del pipeline.

- **Crear una serie de tests automatizados:** Como se menciono en la sección de integración continua, en la entrega continua también es necesario definir una serie de test automatizados a ejecutar. Los tipos de test que generalmente se corren son los de aceptación.

2.4.3 Impacto en la calidad

La aplicación de entrega continua nos brinda la posibilidad de impactar positivamente sobre la experiencia de usuario, se pueden realizar pruebas de concepto fácilmente, detectar errores que rápidamente se pueden solucionar y actualizar el sistema en producción. A su vez las prácticas impactan en los atributos de calidad del sistema, gracias al uso de herramientas de análisis de código y la posibilidad de requerir cierto estándar de calidad y testing para poder incorporar un nuevo cambio a la versión del sistema en producción.

CI y CD mejoran la testeabilidad del sistema, gracias a que las pruebas se corren automáticamente al integrar cada cambio al código fuente, fomentando el testing durante el desarrollo. Por otro lado tenemos que la modificabilidad es un atributo de calidad que busca que el cambio sea lo menos costoso y riesgoso posible.[5] Estas prácticas y herramientas complementan al cumplimiento de este atributo de calidad así como a mejorar la seguridad del sistema, ya que cada cambio realizado será validado por el pipeline de integración y con CD es rápidamente puesto en producción para hacer llegar a los usuarios la nueva versión del sistema.

2.5 Integración continua

La integración continua comprende la primer parte del ciclo de entrega continua. En esta sección vamos a introducir tanto la definición de integración continua como la de las prácticas básicas que la conforman como son: el control de versiones,

el build y el consenso del equipo de desarrollo, finalizando con algunos requisitos que permiten implementar de forma correcta esta práctica.

2.5.1 Definiciones

Es una práctica que refiere a la integración continua de trabajo reduciendo problemas que esto puede producir mediante una verificación automatizada de ciertos requisitos de código definidos por el equipo.[22] Busca generar un código conciso entre los miembros del equipo que se encuentran desarrollando en él; logrando que cada nueva pieza de código pueda ser integrada a un repositorio global.

Control de versión

El código se debe encontrar en un lugar centralizado que contenga todo lo necesario para las etapas de build, test y deploy. Esto significa que todos los archivos, desde casos de prueba hasta configuraciones para deploy se deberán encontrar en un repositorio propiamente identificado. Además se deberá mantener un historial de los cambios realizados a los archivos en el repositorio.

Build automatizado

Nos permite realizar el compilado de la aplicación, junto a la ejecución de los tests de manera automatizada, preferentemente con la utilización de un script que detalle los pasos a seguir.

Consenso en el equipo

Finalmente, el componente de integración se puede llevar a cabo si todos los integrantes del equipo adoptan las prácticas, siendo esto último parte fundamental del proceso.

2.5.2 Prerrequisitos

Revisar regularmente

El paso más fundamental para tener un buen CI es la necesidad de realizar chequeos periódicos al trunk o mainline. Uno de los beneficios más inmediatos es la posibilidad de detectar fallas con antelación y poder solucionarlas a tiempo.

Otro aspecto fundamental al realizar revisiones periódicas es realizar revisiones del pipeline y, si es posible, reducir la cantidad de pasos o dividirlo en secciones más pequeñas, esto hace que se mejoren los tiempos de build/test y que también sea más fácil detectar errores.

Crear una serie de tests automatizados

Para poder asegurar el correcto funcionamiento de la aplicación se requiere tener una lista exhaustiva de tests, esto da la seguridad de que la aplicación se encuentra en un estado funcional. Para esto se deberán aplicar los niveles de tests que definimos anteriormente: testing unitario, testing de integración y testing de aceptación.

Testing Unitario y Funcional

El testing unitario y el testing funcional forman parte fundamental del proceso de CI, dado que permiten asegurar la calidad del producto. Para el caso de un testing unitario se asegura que la calidad del código sea la adecuada y se comporte como se espera, y en complemento el testing funcional se asegura que se cumplan los requerimientos del negocio. Por esas razones es que durante la etapa de integración del producto es necesario asegurarse que se cumplan los requerimientos esperados.

Construir y probar la aplicación en el menor tiempo posible

Idealmente los tiempos de ejecución de tests no deberían tardar más de 5 minutos, 10 minutos como máximo y lo mejor es que sean 90 segundos. Esto a primera instancia se contradice con lo anterior pero para lograr esto una de las técnicas usadas es separar los tests en múltiples etapas. Un ejemplo sería tener una etapa de construcción de la aplicación, luego otra etapa que se encargue de realizar las pruebas.

Cobertura de código

La cobertura de código nos permite descubrir las líneas especificas de código que fueron ejecutadas durante una prueba. Esta técnica es especialmente útil cuando se agregan tests a una aplicación heredada.

- Si existe una linea de código que no fue cubierta por los tests aún, se puede observar claramente que niveles de tests están haciendo falta.
- Si existe una linea de código cubierta en los tests, entonces puede estar más tranquilo al momento de cambiarla o hacer refactor.

La cobertura de código es una métrica habitualmente mal enfocada. En muchos equipos se exige tener un 100% de cobertura de código y luego se escriben tests que no hacen ninguna validación real, solamente ejecutan lineas del código.

Como una herramienta de retroalimentación interna del equipo, resulta muy útil para medir código que viene progresando a medida que aumenta la cobertura. Otra métrica a usar es el índice de defectos (es decir la cantidad de errores encontrados semanalmente).

2.6 Despliegue continuo

El despliegue continuo comprende la última parte del ciclo de entrega continua. En esta sección vamos a introducir la definición de este concepto.

2.6.1 Definiciones

Se define como la técnica de realizar despliegues a producción cada vez que una ejecución de pruebas sea exitosa [2]. La idea es realizar un pipeline automatizado que busque generar un compilado de la aplicación y se envíe directamente al ambiente de producción de la manera más automática posible. Para ello se deberán realizar tests de aceptación, unitarios e integración previamente para poder asegurar la calidad del software.

Esta técnica no es recomendada para todo tipo de organizaciones dado que no siempre se busca tener los últimos cambios en producción, debido a restricciones técnicas o legales esto puede ser un impedimento.

A primera vista parece un proceso muy riesgoso, dado que no se tiene control sobre los despliegues a producción pero esto se ve infravalorado por la ventaja que prevé. Al tener un sistema de deploy automatizado se reduce la cantidad de esfuerzo humano dedicado para una puesta en producción. Además se reduce el riesgo dado que solamente se genera un punto de riesgo que fácilmente se podría retroceder.

2.7 Métricas de entrega continua

La función de las métricas es poder dar a los stakeholders información sobre el estado actual de la aplicación con indicadores objetivos. Estas métricas están fuertemente relacionadas con la calidad, debido a que las mismas se toman en base a los aspectos funcionales y no funcionales de la aplicación. Sin embargo tener

métricas que son consideradas muy buenas no es un indicador lineal de la calidad de un producto, es decir que tener métricas excelentes no asegura que la calidad sea excelente.

De todas las métricas detalladas a continuación para nuestra prueba de concepto solo utilizaremos Time To Value, ya que el resto miden el flujo de trabajo a lo largo del tiempo. Estos datos que no podremos obtenerlos al porque no están contemplados en el alcance de la prueba de concepto, esta se limita a la generación de un ciclo básico de CI/CD y su prueba sobre un sistema previamente implementado.

2.7.1 Tiempo de ciclo

Con tiempo de ciclo, más conocido por su nombre en inglés "cycle time", nos referimos al intervalo de tiempo entre dos cambios consecutivos en el código que pasan de la etapa de desarrollo a producción, es decir el tiempo que demora en llevarse a cabo un ciclo completo de entrega continua.[16] Esto puede contestarse con la siguiente pregunta: *"¿Cuanto tiempo demora el equipo en crear algo?"*[14]. A partir de esto el equipo establece analíticas y medidas que permiten volver a revisar el pipeline, iterar sobre él y aplicar mejoras, volviendo esta métrica una de las más importantes a tener en cuenta durante el uso de entrega continua.

Dentro de esta métrica también existe una llamada promedio. El promedio se calcula como el tiempo que existe entre cada uno de los ciclos. A diferencia del tiempo de ciclo que mide el tiempo de puesta en producción, el promedio hace un análisis histórico de los distintos despliegues realizados.

2.7.2 Time To Value

La valorización es el tiempo que se mide en tener una aplicación lanzada luego de tener realizado el código. Es un subconjunto del tiempo de ciclo, el cual se enfoca en el proceso de publicación, mediante la métrica conocida como TTV (Time To

Value). Mientras el tiempo de ciclo mide la duración de un ciclo de entrega continua (abarcando todo CI/CD), TTV abarca únicamente la etapa del ciclo correspondiente a CD.[14]

2.7.3 Uptime

Una de las prioridades del equipo de operaciones es lograr el correcto funcionamiento del sistema mientras este está activo, para esto se mide el tiempo en el que el sistema está activo, conocido como "Uptime". Esta medida está relacionada con estabilidad y resiliencia de nuestro producto. [14]

Si se obtienen tiempos de error elevados es sinónimo de falta de cohesión entre el equipo de desarrollo con el de operaciones. Para lograr un buen flujo es necesario indicar métricas de operaciones y compartirlas con el equipo.[14]

2.7.4 Calidad de código

Calidad de código es una métrica que permite asegurar el estándar del código y permite realizar una entrega de software más eficiente.[57] Algunas de las métricas que se utilizan son:

- **Análisis de código estático:** Conocido como linter, que definiremos más adelante.
- **Líneas Duplicadas:** Son las líneas de código que realizan la misma función de la misma manera.
- **Vulnerabilidades de código:** Son las instrucciones que generan posibles puntos de falla de seguridad en el código.
- **Cobertura de código:** Cobertura de códigos es el porcentaje de código que es cubierto por los casos de test, utilizando frameworks de test o herramientas.

- **Documentación:** La documentación permite tener una manera de formalizar las decisiones de código, de forma tal que, mientras se desarrolla cada módulo se conozca con precisión qué decisiones se tomaron para realizarlo.
- **Extensibilidad:** Es el grado en el cual un software puede incorporar un crecimiento a futuro sin necesidad de afectar a todo el sistema.

2.7.5 Métricas relacionadas al desempeño

- Cantidad de implementaciones de código y cambios realizados.[2]
- Tiempo de codificación y cambio de la implementación, también conocido como tiempo de tarea o tiempo de contacto.[2]
- Tasa de errores de cambio: Es el porcentaje de cambios de código que requieren correcciones urgentes u otra corrección después de la producción. Esto no mide las fallas detectadas por las pruebas y reparadas antes de que se implemente el código.[2] Junto con tiempo de ciclo esta es una de las métricas más relevantes durante el uso de entrega continua, ya que nos permite de forma directa obtener el impacto en la calidad percibida por los usuarios de nuestro sistema.

2.7.6 Métricas DORA

Otras métricas relacionadas son las "DevOps Research and Assessment", conocidas como "DORA". Estas métricas guían a las empresas a hacer las acciones necesarias para entregar y desarrollar soluciones tecnológicas de manera rápida y confiable.[45]

- **Frecuencia de implementación:** Con qué frecuencia una organización lanza a producción.[46]
- **Plazo medio de entrega de cambios:** La cantidad de tiempo que tarda un cambio en entrar en producción.[46]

- **Tasa de fallas de cambio:** Cantidad de implementaciones que causan fallas en producción.[46]
- **Tiempo para restaurar el servicio:** Tiempo que le toma a una organización recuperarse de una falla en producción.[46]

2.7.7 Métricas de SRE

- **Latencia:** Se define como el tiempo en que se resuelve una petición. Esta métrica distingue los pedidos exitosos de los fallidos, para los cuales se toma la latencia de error para poder establecer métricas genuinas.
- **Tráfico:** El tráfico se define como la demanda que tiene el sistema. Por ejemplo, en HTTP se toma la cantidad de peticiones por segundo.
- **Errores:** Se considera como el ratio de peticiones que fallaron.
- **Saturación:** Se considera como la ocupación que se tiene en el servicio. Es una medida de una fracción del sistema haciendo énfasis en los recursos limitados. En sistemas complejos la saturación puede ser implementada a más alto nivel, por ejemplo es posible que el sistema soporte el doble de tráfico.

2.8 Patrones y anti patrones

En esta sección vamos a presentar algunos de los patrones y anti patrones que deben tenerse en cuenta durante el uso de entrega continua.

2.8.1 Patrones

Los patrones son prácticas relacionadas a la entrega continua que buscan resolver o prevenir problemas relacionados con algunas de las etapas del ciclo de CI/CD. A continuación introduciremos brevemente algunos de estos.

Deployment pipeline

Deployment pipeline se define como: *"Una implementación automatizada de las etapas de proceso build, deploy, test y release de una aplicacion"*.[1]

Releases de bajo riesgo son incrementales

El objetivo debe ser armar la arquitectura del sistema que permita hacer liberaciones de cambios individuales independientemente, en vez de orquestar un release con cambios trascendentales que se encuentren acoplados a diferentes sistemas.

Optimizar para resiliencia

Se acepta que las fallas son inevitables, por lo tanto hay que cambiar la idea de esforzarse en prevenir problemas y en cambio pensar en como poder restaurar el servicio lo mas rápido posible cuando algo falla. Además, cuando ocurre un accidente se tiene que tomar como una oportunidad de aprendizaje.

Canary Release

Canary release surge como una alternativa al concepto clásico de deployment. Es decir solamente tendría que existir una sola versión del software en producción. Esto logra que la corrección de errores y mantener la infraestructura sean procesos simples pero se dificulta al momento de realizar pruebas. Además si se tiene aplicaciones muy grandes el generar un ambiente de prueba se hace imposible.

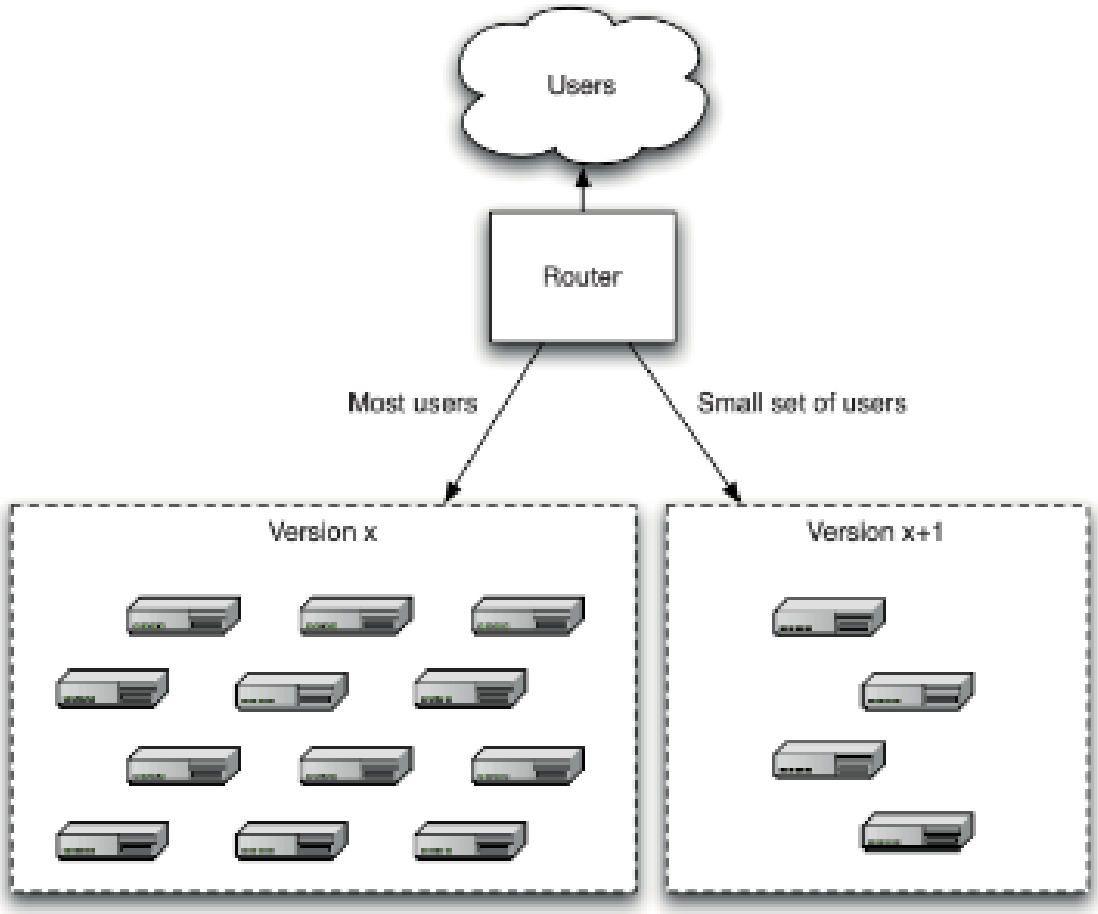

Figure 5: Canary Relase Strategy [1]

Como se muestra en la figura 5 busca realizar un deploy a una porción de los servidores de producción con la versión Canary (versión x + 1) para obtener un rápido feedback de ellos. Al tener un grupo de usuarios controlado que pruebe las nuevas funcionalidades es muy fácil realizar un rollback o corregir un error existente.

Blue Green

Es una técnica de deployment que consiste en lograr tener dos versiones idénticas en producción llamadas Blue y Green.

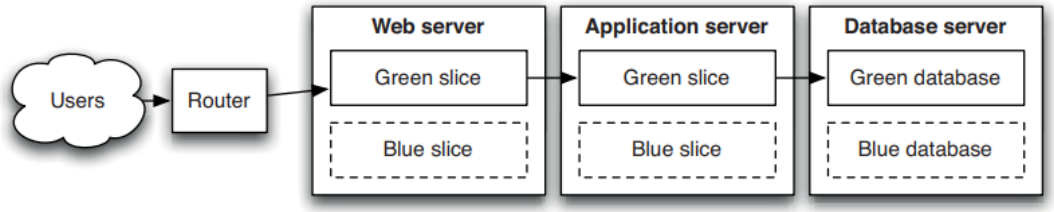

Figure 6: Blue Green [1]

En el ejemplo 6 se muestra a los usuarios redirigidos al ambiente green que es el que se encuentra actualmente en producción. Para realizar un deploy de una nueva versión se envía al ambiente blue, donde se realizan pruebas para asegurar su correcto funcionamiento. Una vez que las pruebas sean exitosas se cambia el router para que apunte a la nueva versión de producción que es el ambiente blue.

La clave es generar que los ambientes sean independientes, es decir tengan sus propios recursos, base de datos, file systems, entre otros. Esta es la manera más sencilla de configurar pero sin embargo es la más costosa.

AB testing

Durante los procesos de entrega es necesario contar con AB Testing. AB Testing consiste dividir el testing en fases, tomando diferentes variables o métricas para testear, dando a un grupo de usuarios una variante A y a otros una variante B. Esto es importante durante la entrega dado que permite poder obtener métricas y mejorar el producto.

2.8.2 Anti patrones

Los anti patrones son prácticas relacionadas a la entrega continua que debemos procurar evitar hacer para poder disminuir la incidencia de problemas relacionados con algunas de las etapas del ciclo de CI/CD.

Desplegar software manualmente

En muchas empresas el proceso de release, que corresponde a la etapa de CD, es manual. Dejándolos propenso a errores humanos. Incluso si no es el caso, se pueden generar diferencias en el orden y sincronización de los pasos que lo componen lo cual lleva a distintos resultados. Es decir si se cambia la entrada del despliegue esto afectará a los artefactos de salida.[1]

Desplegar a un ambiente similar a producción únicamente luego de que el desarrollo acabó

Este anti patrón refiere a cuando el software es puesto en un ambiente similar a producción cuando la mayoría del desarrollo esta terminado, el concepto de "terminado" es definido por el equipo de desarrollo.[1]

Gestión de la configuración manual de ambientes productivos

Muchas empresas manejan la configuración de sus entornos de producción de forma manual, de esta forma si un cambio es necesario tienen que hacerlo en cada servidor.[1]

3. Comparación de tecnologías para pipelines de desarrollo

En este capítulo introduciremos las principales tecnologías relacionadas con el proceso de entrega continua, así como algunos ejemplos de herramientas para cada una de ellas.

3.1 Control de versiones

Los sistemas de control de versiones nos permiten mantener la fluidez en el desarrollo de un sistema, ya que cada integrante puede trabajar sobre el mismo código sin preocuparse por generar conflictos, actualizar fácilmente su versión local con la que esta almacenada en la nube, cambiar de contexto rápidamente, volver en caso de necesitarlo a versiones previas del sistema, entre muchas otras ventajas. El uso de estas herramientas es muy importante durante la etapa de CI, nos permiten cumplir con la integración de software rápida y manteniendo el sistema estable. En esta sección hablaremos de algunas tecnologías de control de versiones y sus funcionalidades.

3.1.1 Git

"Git es un sistema de control de versiones distribuido de código abierto y gratuito diseñado para manejar todo, desde proyectos pequeños a muy grandes, con velocidad y eficiencia."[20]

Git conforma parte del ciclo de integración continua como una pieza fundamental, por las siguientes razones:

- Es una herramienta estándar del desarrollo profesional de Software.
- Es utilizada por millones de desarrolladores.
- Es utilizada en la mayoría de los productos open source.

Ramas y merge

Una rama se define como una versión nueva o separada del repositorio[20]. Una de las principales diferencias con otros gestores de código es el uso de dichas ramas. Como se menciono en el marco teórico esto tiene sus ventajas y desventajas al momento de utilizar prácticas como CI. Esto permite poder realizar lo siguiente:

- **Cambiar de contexto rápidamente:** La idea es poder crear una rama nueva, generar cambios y luego volver a la rama de la que partimos, aplicar otros cambios y poder volver a la rama nueva e integrarla. Todo esto sin generar dependencia entre las ramas.
- **Código basado en roles:** Cada rama tiene un propósito, hay una rama que tiene el código de desarrollo, otra el de producción y unas terceras con las distintas features, esto permite poder hacer integraciones con el pipeline de CI para poder desarrollar la integración acorde al ambiente que se este trabajando.

- **Flujo de trabajo basado en funcionalidades:** Por cada funcionalidad, una rama puede ser creada, integrada y eliminada de acuerdo al estado de la misma.
- **Experimentación descartable:** Permite poder crear ramas de experimentación y en caso de no funcionar, se puede eliminar fácilmente.

Aseguramiento de datos

Para cumplir con los principios de seguridad CIA (confidencialidad, integridad y disponibilidad), cada archivo y carpeta dentro del repositorio genera un checksum que se verifica cuando se hace un salto de una rama a otra.

Esto permite que cada cambio que se realiza genera un nuevo código único, lo cual asegura que los cambios que fueron agregados previamente no se modificaron. El aseguramiento de los datos es parte fundamental del ciclo de CI, dado que permite mantener la confianza sobre los cambios que se realizaron e integrarlos rápidamente para poder seguir desarrollando sobre ellos.

3.1.2 Comparación de gestores de repositorios en la nube

A continuación dejaremos detallada la comparación entre las funcionalidades ofrecidas por los tres principales gestores de repositorios: GitHub, Bitbucket y GitLab.[27]

Pull Requests

Funcionalidad	GitHub	Bitbucket	Gitlab
Adjuntar etiquetas para proporcionar contexto.	Sí	No	Sí
Diferencia de cambios entre la rama fuente y base.	Sí	Sí	Sí
Integración con herramientas externas de integración continua.	Sí	Sí	Sí
Plantillas de PR para garantizar que se sigan las pautas de contribución.	Sí	No	No
Conversaciones sobre partes del código que requieren resolución.	Sí	No	No
Revisiones necesarias para garantizar que alguien apruebe todas las PRs antes de incorporar el cambio.	Sí	Sí	No
Los miembros del equipo pueden suscribirse para recibir una notificación cuando se fusione la solicitud.	No	No	Sí

Integraciones

- **GitHub:** GitHub contiene una aplicación llamada GitHub Marketplace en la cual se suben aplicaciones para usar con third-party. Además de tiene una API que permite crear aplicaciones.
- **Bitbucket:** Contiene tiene un marketplace y una API para poder crear aplicaciones custom. Tiene sus propio pipeline permitiendo usar CI/CD.
- **GitLab:** GitLab tiene menos integraciones que los otros dos proveedores. [27]

3.2 Contenerización

Nuestros sistemas hoy en día se encuentran ejecutándose en la nube, pero nosotros seguimos desarrollando en un ambiente local, para asegurarnos que nuestro software va a funcionar una vez este en producción como esperamos realizamos pruebas locales en un ambiente similar al de producción. La forma hacer esto es utilizando contenedores, estos van encapsular el contexto en el cual va a correr nuestro sistema en producción permitiéndonos emularlo localmente. En esta sección hablaremos de Docker, una de las principales tecnologías de contenerización, y de Kubernetes, una herramienta de orquestación de contenedores.

3.2.1 Docker

Historia

Previo a la aparición de Docker existían algunas alternativas para poder utilizar contenedores. Una de las tecnologías más conocidas es Chroot que fue introducido hace 40 años, el cual permitía cambiar el root filesystem de un proceso y sus hijos. Era muy limitado no permitía utilizar redes o aislamiento de procesos.

A mediados del año 2008 surge una nueva funcionalidad como parte del kernel de Linux conocida como Control Groups que permite poder aislar los recursos (CPU, memoria, redes) de un grupo de procesos. Estos luego fueron precedidos por CGroups el cual introdujo el concepto de namespaces, que son particiones de los recursos del kernel en el cual un namespace no podía acceder al proceso de otro.

Descripción

"Docker es un motor open source que permite automatizar el deployment de una aplicación a un contenedor".[3]

Docker agrega una capa más de deployment sobre un contenedor ejecutado en un entorno virtual. Está diseñado para poder proveer un ambiente ligero y rápido de manera de poder ejecutar código en un workflow eficiente desde el ambiente de desarrollo hasta un ambiente de producción.

Docker usa un modelo de copy-on-write que permite hacer cambios en la aplicación rápidamente. La mayoría de los contenedores demoran menos de 1 segundo para deploy. Esto permite que los contenedores tengan muy buen rendimiento pudiendo aprovechar los recursos de los host.

Desde un punto de vista arquitectónico se busca seguir una arquitectura orientada a servicios y arquitecturas de micro-servicios. Una recomendación es correr un contenedor con una sola aplicación o proceso. Esto promueve un modelo de aplicación donde busca que los contenedores estén interconectados.

Componentes

- **Cliente y servidor de Docker:** Docker es una aplicación cliente-servidor. El cliente se comunica con el servidor o daemon (Docker Engine). Viene con una RESTful API que permite interactuar al cliente con el servidor.

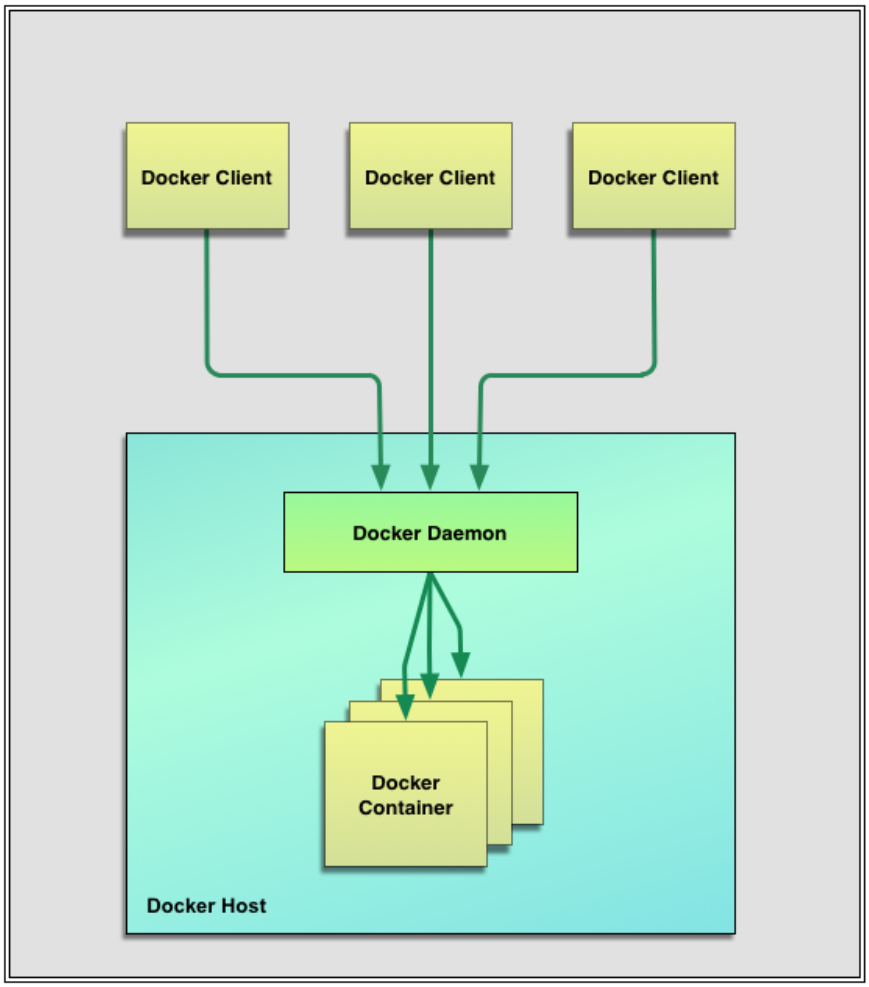

Figure 7: Arquitectura de Docker [3]

- **Imágenes:** Las imágenes son los bloques, se pueden armar contenedores a partir de las imágenes. Las imágenes son parte del proceso de **build**. Tienen el *código fuente* de los contenedores. Al ser muy portables se puede compartirlos, guardarlos o actualizarlos fácilmente .

- **Registros:** Docker permite guardar las imágenes en registros, hay 2 tipos de registros privado y público. DockerHub contiene más de 10.000 imágenes construidas. También se pueden almacenar imágenes privadas, por razones de seguridad, que sean solamente compartidas para miembros del equipo u organización.

- **Contenedores:** Los contenedores son la parte de *RUN* del ciclo de vida de docker. Algunos aspectos de los contenedores son:

- Formato de imagen
- Operaciones estándar
- Ambiente de ejecución

Docker se basa en el concepto de contenedor de carga, que se usa para transportar bienes mundialmente, como un modelo para los contenedores. Como un contenedor de carga, a Docker no le interesa el contenido que tiene sino las operaciones que se realizan. Tampoco interesa como se envía ese contenedor, pudiendo ser flexible a la hora de elegir la infraestructura para correrlo.

Casos de uso

Debido a que los contenedores prevén aislación entre sí, son muy usados como ambientes de prueba. Esto Otros usos son:

- SaaS (Software as a Service) muy performante.
- Construir una infraestructura PaaS (Platform as a Service) multi-usuario.
- Ejecutar servicios standalone y aplicaciones en múltiples ambientes, muy usado en arquitectura orientadas a servicios.
- Poder ejecutar instancias aisladas de tests, para el proceso de CI por ejemplo.
- Crear y testear aplicaciones complejas en un ambiente local antes de salir a producción.

Los distintos casos de uso mencionados son muy usados en los contextos de CD, dado que permite aislar los componentes de la aplicaciones sin depender del hardware o equipo que se este utilizando.

3.2.2 Kubernetes

Un orquestador es un sistema que realiza deploy y administra aplicaciones. Permite realizar deploy de aplicaciones y responder dinámicamente a los cambios. Por ejemplo:

- Realizar deploy de una aplicación.
- Escalar y des-escalar de acuerdo a demanda.
- Realizar auto revisiones.
- Aplicar y deshacer actualizaciones sin tiempo de espera.

Kubernetes surgió por la necesidad por parte de Google de administrar aplicaciones enormes como Gmail y Search. Docker logró hacer que los contenedores sean fáciles de usar. Por otro lado, para administrar y orquestar esas aplicaciones en contenedores, Google tenia un par de soluciones in-house. Finalmente decidieron tomar esas plataformas y generar una nueva llamada Kubernetes.[4]

Kubernetes como cluster

Como cualquier otro cluster, Kubernetes tiene muchos nodos y un panel de control. El panel de control se expone mediante una API, la cual contiene un scheduler para asignar tareas a los nodos.

Un analogía es pensar el panel de control como el cerebro y los nodos como los músculos. El panel de control es el cerebro porque implementa todas las funcionalidades importantes como auto escalado y actualizaciones automáticas. Los nodos son el músculo; porque hacen las tareas de ejecutar el código de las aplicaciones.

Kubernetes como orquestador

Un orquestador es un sistema que se encarga de realizar deploys y administrar aplicaciones. *"Una analogía sería un equipo de fútbol esta compuesto de jugadores, no hay dos jugadores iguales y cada uno tiene un rol en el equipo, algunos defienden, otros atacan, otros hacen pases, etc. Además está el entrenador, que posiciona a todos y le da un propósito al equipo. También se asegura que el equipo siga el plan, mantenga su formación, se encarga de los heridos y cualquier otra situación."* [4]

Esta analogía se puede aplicar a Kubernetes, donde existen muchos servicios especializados con diferentes funciones (como los jugadores). Kubernetes sería como el entrenador, se encarga de ordenar todo para que funcione de la mejor manera. Responde ante eventos y otros cambios. Kubernetes orquesta arquitecturas de micro-servicios cloud-native.

Docker y Kubernetes

Docker y Kubernetes son tecnologías que se complementan. Por ejemplo, se pueden desarrollar aplicaciones con Docker y luego usar Kubernetes para orquestarlas en un ambiente de producción.

De este modo se puede generar una aplicación en cualquier plataforma y luego usar Docker para generar un contenedor, probarlo y luego enviarlo a producción. Luego se puede utilizar Kubernetes para administrar al realizar deploy y ejecución de esos contenedores.

A alto nivel Kubernetes maneja los clusters con nodos que tienen la aplicación, cada nodo es un contenedor con un ambiente de Docker en ejecución. Docker es un sistema low-level que contiene las aplicaciones y Kubernetes es un sistema high-level que ofrece una mirada más abstracta y decide qué nodos se están ejecutando, tienen tareas de escalar y ejecutar actualizaciones.

3.3 Monitoreo

Las métricas son medidas numéricas de determinados atributos o características de un sistema que nos permiten tener visibilidad, lo que permite detectar y mitigar problemas más rápidamente. Las herramientas de monitoreo mediante métricas nos permiten ver el estado del sistema y controlar la experiencia del usuario. Este tipo de herramientas son importantes durante la última etapa del ciclo de entrega continua: el monitoreo. A continuación listaremos algunas de estas herramientas.

3.3.1 Prometheus

Prometheus es un conjunto de herramientas de monitoreo y alerta de sistemas de código abierto construido originalmente en SoundCloud. Funciona desde el 2012, y cuenta con una actividad de usuarios y desarrolladores activa, ya que es open source.[29].

Almacena métricas y datos con sus fechas de registro y opcionalmente una etiqueta formada por un par clave-valor.

Ofrece funcionalidades como:

- Un modelo de datos multidimensional con datos de series de tiempo identificados por el nombre de la métrica pares/clave.
- No depende del almacenamiento distribuido; los nodos de un solo servidor son autónomos.
- Utiliza PromQL un lenguaje de consulta flexible.

3.3.2 Datadog

Datadog por su lado permite buscar, filtrar y analizar rápidamente los registros para la resolución de problemas y la exploración abierta de sus datos, este recopila automáticamente registros de todos las servicios, aplicaciones y plataformas y nos permite ver los datos de registro en contexto con el etiquetado y la correlación automatizados.

Entre algunas de sus funcionalidades se encuentran:

- Permite supervisar los recorridos críticos de los usuarios capturándolos con una grabadora web fácil de usar.[72]
- Permite hacer pruebas de mantenimiento automático impulsadas por Inteligencia Artificial.[72]
- Permite detectar y alertar sobre problemas de rendimiento para los usuarios en varias ubicaciones.[72]
- Facilita la administración de acuerdos a nivel de servicio (conocido por sus siglas en inglés SLA, Service Level Agreement), se define como un acuerdo entre cliente y proveedor de parámetros cuantificables (tiempo de actividad, capacidad de respuesta del sistema, etc). También facilita la administración de los objetivos a nivel de servicio (conocido por sus siglas en inglés SLO, Service Level Objective), que establece las expectativas de los clientes basado en los SLA e indica a los equipos de desarrollo y operaciones qué objetivos alcanzar y cuantificar.[72]

3.3.3 New Relic

New Relic nos permite supervisar en vivo la red, infraestructura, aplicaciones, experiencia del usuario final y modelos de aprendizaje automático.[71]

También nos permite:

- Permite resolver problemas a través de aplicaciones multi-plataforma y asistencia de una inteligencia artificial.[71]
- Plataforma de datos segura e hiper escalable.[71]
- Solo se paga por lo que se utiliza, sin multas por exceso.[71]

3.4 Manejo de configuraciones

"Un manejador de configuración es un conjunto de practicas de ingeniería que permite administrar las siguientes entidades que involucran la entrega del software a los consumidores: Hardware, Software, Infraestructura, Personas y Procesos".[7]

Busca solucionar los problemas generados por la administración de servidores cuando involucran grandes cantidades de desarrolladores, servidores y aplicaciones. Algunos de los beneficios de los manejadores de configuración que son reflejados en el uso de entrega continua son:

- Los cambios deben ser administrados de manera sistemática para poder asegurar que el sistema está configurado de la manera correcta.
- Debe existir algún sistema automatizado que permite detectar y solucionar las fallas rápidamente, sin necesidad de intervención.

Los manejadores de configuración fueron inicialmente desarrollados por el Departamento de Defensa de Estados Unidos para configurar equipos complejos y sofisticados, por ejemplo tanques, aviones, equipos oceánicos.[59]

A medida que los sistemas se volvieron más complejos, la filosofía de los manejadores de configuración (CM por sus siglas en inglés) fue introducida en el mundo del desarrollo de software, lo cual permite a los equipos poder visualizar todos sus

sistemas virtuales y mantener un historial de los cambios. A continuación nombraremos brevemente algunos ejemplos de herramientas de manejo de configuración, y hablaremos de su impacto en los ambientes utilizados durante el ciclo de entrega continua.

3.4.1 Chef

Chef se define como: *"Una plataforma de automatización que configura y administra la infraestructura on-premise o cloud. Permite acelerar el deploy de aplicaciones, dado que permite crear un pipeline de deployment. Una de las ventajas es la posibilidad de transformar infraestructura en código".*[7]

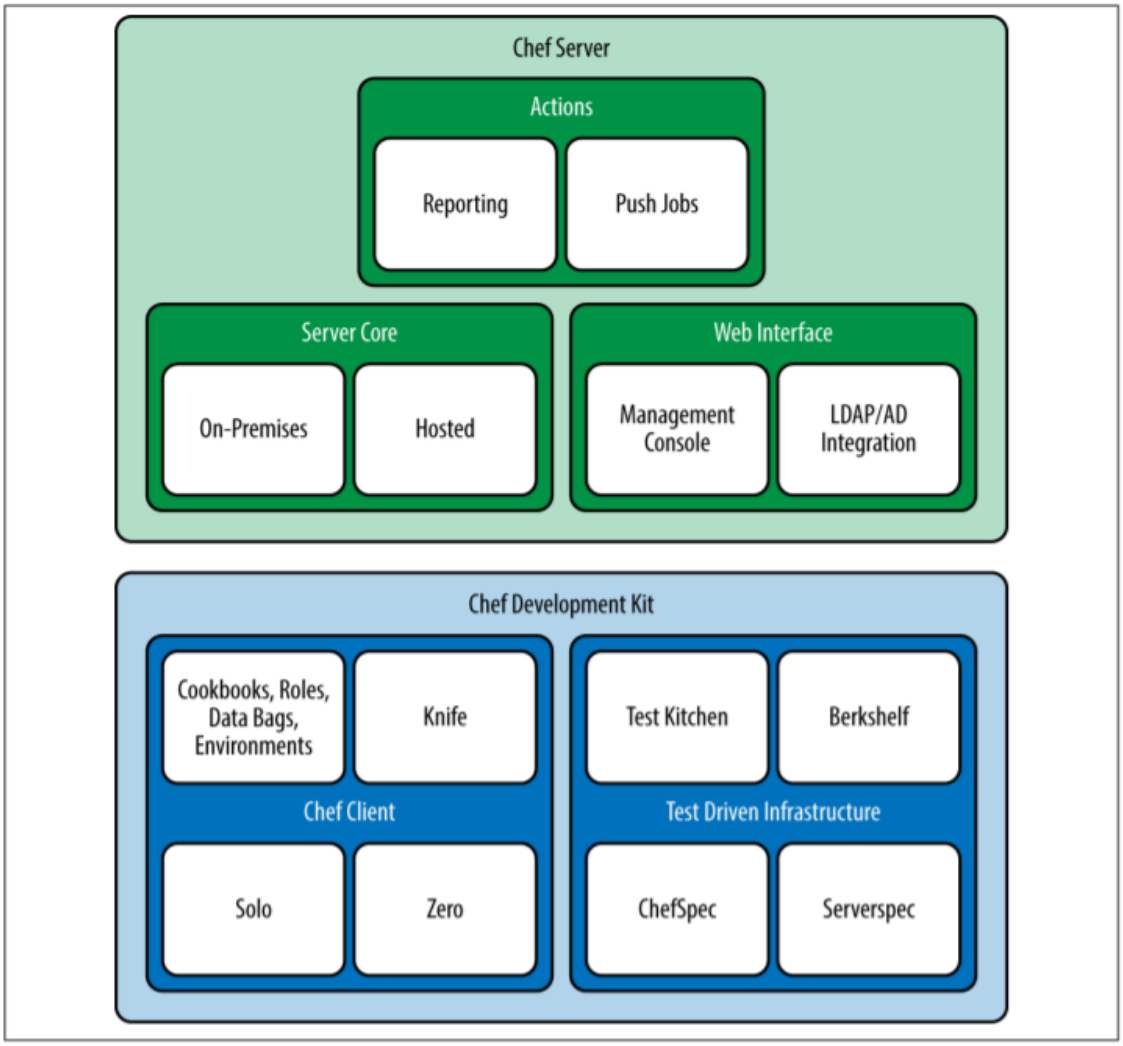

Figure 8: Arquitectura de Chef [7]

3.4.2 Puppet

"Puppet permite que los sistemas puedan cumplir con las políticas diseñadas. Puppet administra la configuración de los sistemas, usuarios, paquetes, procesos, servicios. Puppet puede administrar componentes complejos para asegurar que se cumplas las políticas".[8]

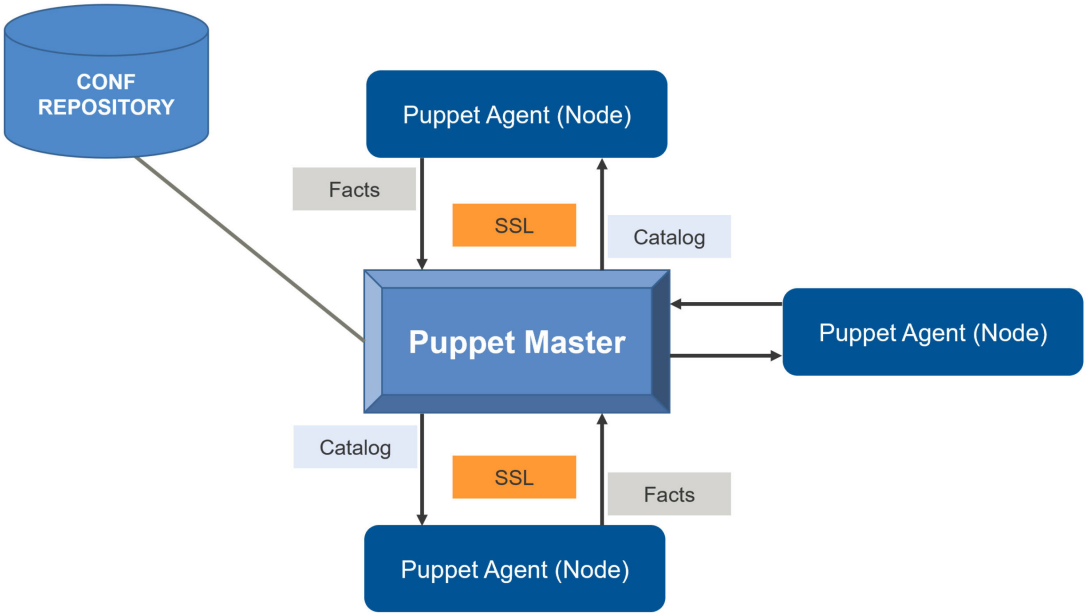

Figure 9: Arquitectura de Puppet [15]

3.4.3 Impacto en los ambientes

Durante el desarrollo de un proyecto de software se definen los ambientes, que pueden ser de desarrollo, testing, staging y de producción, queda a criterio del equipo elegir cuales va a utilizar. Esto surge debido a que durante la fase de desarrollo el software requiere la utilización de herramientas que permitan, por ejemplo, el uso de paquetes de testing, uso de generadores de código, uso de linters, entre otras.

Por otro lado en un ambiente de producción se configuran las dependencias que serán utilizadas estrictamente por la aplicación durante su ejecución, es decir las dependencias que la aplicación utiliza para funcionar.

3.5 Herramientas de calidad

Incorporar herramientas de calidad en nuestro flujo de CI nos permite controlar de forma aromatizada aspectos del sistema que de hacerlo manualmente tomarían mayor esfuerzo por parte de los equipos y no nos aseguraría que errores puedan ser pasados por alto. Estas herramientas se pueden clasificar como dinámicas o estáticas, las primeras son aquellas que ejecutan el código y las segundas no necesitan ejecutarlo. A continuación nombraremos algunos ejemplos de herramientas de calidad que pueden ser utilizadas durante el desarrollo local como en el pipeline de CI, siendo los code linters y los code formatters estáticos, security checks hay tanto dinámicos como estáticos.

3.5.1 Code linters

Un linter es un programa de que analiza y verifica código fuente, señala errores de programación, errores de formato, entre otras cosas. Es importante para reducir errores y mejorar la calidad general del código, ayuda a acelerar el desarrollo y reducir costos al encontrar errores antes [47]. Esto como consecuencia trae mejoras en la seguridad, performance y calidad interna del código.

Existen muchos linters para distintos lenguajes de programación, por ejemplo, ESLint es el más utilizado para JavaScript. Este mismo posee soporte para los estándares de codificación de Airbnb y Google.[68][69] Por ejemplo, para utilizar ESLint creamos un archivo llamado .eslintrc en el root de nuestro proyecto, donde podemos declarar el estándar a utilizar y las reglas con las que lo queremos extender.

```
{
  "extends": "google",
  "rules": {
    // Reglas adicionales al estándar que estamos usando.
  }
}
```

Figure 10: Ejemplo de un archivo .eslintrc

3.5.2 Code formatters

Los formateadores de código permiten automatizar la tarea de dar formato al código y resaltado de sintaxis para lenguajes de programación y formatos de archivo comunes. Las ventajas que trae este tipo de prácticas es hacer que todo el código sea más similar, de modo que los desarrolladores se sientan más cómodos al unir y compartir el código, porque lo hace más consistente y fácil de entender.[48]

Pero también trae sus desventajas como limitar la capacidad de realizar cualquier formateo de 'casos especiales' donde un formato alternativo haría que el código fuera más legible.[48]

3.5.3 Security checks

La seguridad en el software genera un gran impacto en la calidad, ya que cualquier vulnerabilidad afecta tanto a la empresa detrás del software como a los usuarios. Por esta razón existen herramientas encargadas de encontrar posibles puntos de falla de seguridad en el código en etapas tempranas del desarrollo.

Una herramienta muy conocida y open source es OWASP[63], de este podemos encontrar plugins que nos permitirán utilizarlo desde herramientas de integración continua.

3.6 CI

La etapa de integración continua abarca todo el ciclo de desarrollo hasta que la nueva funcionalidad se encuentra incorporada a la versión estable del sistema y lista para ir a producción. Durante esta etapa implementamos un pipeline que corre determinados checkeos para asegurarnos que la incorporación del nuevo cambio cumple con los estándares de calidad impuestos por el equipo, y que el cambio no incorporará errores a la versión estable del sistema. En esta sección hablaremos sobre algunas herramientas que nos permiten implementar este pipeline en nuestro sistema de control de versiones.

3.6.1 Pioneros

Jenkins

Es un servidor de automatización de código abierto que permite a los desarrolladores de todo el mundo crear, probar e implementar su software de manera confiable.[30] Jenkins dispone de la funcionalidad de extenderse mediante plugins, algunos de los más populares hoy en día son: [30]

- **Miler:** Permite configurar notificaciones por correo electrónico para los resultados de la compilación.
- **Sript Security:** Permite a los administradores controlar qué proceso un script puede correr.
- **Credentials:** Permite guardar credenciales en Jenkins.

CircleCI

Es una herramienta de CI que permite integrarse con sistemas de versionado de código como GitHub o Bitbucket, permitiendo que cada vez que se realice una actualización de código se cree un pipeline. Ejecuta automáticamente el pipeline en un contenedor limpio o en una máquina virtual, lo que le permite probar cada cambio realizado.[31]

Permite notificar al equipo cuando falló el pipeline, incluso tiene integración con Slack para estos casos.[31]

Travis

Permite ingresar a nuestro repositorio en la nube y prueban el proyecto antes de enviarlo. Permite también testear los cambios realizados en el código antes de ser integrados al código fuente.[32] Muchas bases de datos y servicios están pre-instalados y se pueden habilitar en su configuración de compilación.

Estas son herramientas de integración, grandes orquestadores, que se pueden ver pisados con otras herramientas brindadas por empresas como Amazon, Atlassian, GitHub y GitLab que ofrecen plataformas de guía de desarrollo con development pipeline.

3.6.2 Recientes

GitHub Actions

Ayudan a automatizar tareas dentro del ciclo de vida de desarrollo de software. Se manejan por eventos, lo cual significa que se puede ejecutar un flujo de trabajo; ejecutar una serie de comandos después de que haya ocurrido un evento especifico, por ejemplo: cada vez que alguien crea una solicitud de cambios para un

repositorio, que se ejecute automáticamente un script de prueba de software.[33]

"Un flujo de trabajo es un archivo de proceso configurable que permite automatizar uno o mas tareas. Se debe definir un archivo YAML dentro de la configuración del flujo".[33]

Los comandos más utilizados son: [33]

- **name**
  Opcional, aparecerá en el nombre del flujo de trabajo.
- **on**
  Especifica el activador de la función, por ejemplo puede ser activado por un push.
- **jobs**
  Se agrupan los trabajos a ejecutar en el workflow.
- **runs-on**
  Se configura el workflow para ejecutarse en una maquina virtual de GitHub.
- **steps**
  Se agrupan todos los pasos a ejecutar en un trabajo.
- **uses**
  Permite realizar check out del repositorio al runner, permitiendo ejecutar scripts u otras acciones sobre el código.
- **run**
  Se indica que instrucción ejecutar.

AWS Code Pipeline

Es un servicio de entrega continua que ayuda a automatizar los procesos de lanzamiento para obtener actualizaciones de infraestructura y aplicaciones rápidas y confiables.[34]

CodePipeline automatiza las fases de compilación, prueba e implementación del proceso de lanzamiento cada vez que hay un cambio de código, según el modelo de lanzamiento definido. Se puede integrar fácilmente con servicios de terceros como GitHub. Adicionalmente permite solo pagar por lo que usa.[34]

3.6.3 Conceptos

- **Job (Jenkins, CircleCI):** Cada Job es una unidad atómica de trabajo, como pueden ser los Tests Unitarios. Son responsables de ejecutar una serie de pasos que ejecutan comandos, para esto utiliza un Executor.[42]
- **Executor (Github Actions, CircleCI):** Un Executor determina el entorno de ejecución de los pasos, es decir, un contenedor o una maquina virtual corriendo una imagen especifica.
- **Step (Github Actions, CircleCI):** Los pasos con los comandos que se van a ir ejecutando en el Job.
- **Action (Github Actions):** Una acción es un programa que permite ser utilizado en los workflows. Las acciones pueden instalar software para el ambiente, generar la autenticación, o automatizar tareas complejas. Se pueden utilizar actions ya definidas en GitHub Marketplace o generar unas propias.
- **Workflow (Github Actions, CircleCI):** Es un proceso automatizado y configurable que permite ser utilizado en un repositorio para poder hacer las siguientes tareas:
  - Build
  - Test
  - Package
  - Release
  - Deploy

Los Workflows están compuestos de uno o más *jobs* los cuales son activados por un evento en GitHub.

- **Archivos de Configuración (Github Actions, CircleCI):** Para la configuración de los pipelines en GitHub Actions y CircleCI se utilizan los archivos estilo YAML.

  El archivo de configuración define el workflow de GitHub Actions. Esta escrito en formato YAML y se aloja en el repositorio en el directorio *.github/workflows*. Cada archivo de .yaml que se encuentre en esa ruta define un workflow único.

  El archivo de configuración define el workflow de CircleCI esta alojado en el repositorio en el directorio *.circleci* con el nombre *config.yml*. Dentro del archivo encontraremos los Workflows y Jobs con sus respectivos Executor y Steps.

  YAML significa Yet Another Markup Language, es un estilo de lenguaje con estilo markup human-readable que se usa comúnmente en archivos de configuración en especial para el uso de herramientas enfocadas en CI y DevOps.

- **Contexto (Github Actions, CircleCI):**

  Los contextos son una manera de acceder a la información sobre un workflow, usando una sintaxis estilo expresiones. GitHub Actions contiene una colección de variables que se llaman contextos y también una colección de de variables por defecto. Estos tienen distintos usos:

  - **Variables de ambiente por defecto:** Estas variables solo existen durante la ejecución de un job.
  - **Contextos:** Los contextos pueden ser usados en cualquier momento del workflow, incluso cuando las variables por defecto no pueden ser usadas. Por ejemplo, se pueden usar contextos con expresiones para realizar tareas antes de ejecutar un job. Esto permite usar los contextos con condicionales para determinar si un step debe ser ejecutado o no. También mientras un job se encuentra ejecutando se pueden obtener variables de los contextos.

3.7 CD

Anteriormente vimos las herramientas de CI, estas nos permiten manejar la construcción, pruebas e incorporación del código desarrollado por el equipo. Una vez el código se incorpora en una rama estable nos apoyamos en las herramientas de despliegue continuo para que este llegue a producción de forma automatizada. Para la incorporación de CD en nuestro sistema contamos con un amplio abanico de opciones, a continuación nombraremos algunas de estas.

3.7.1 Herramientas de código abierto

Argo CD

Argo CD tiene como objetivo que las implementaciones de aplicaciones sean automatizadas, versionadas y fáciles de entender. Argo CD busca cumplir su objetivo brindando una herramienta declarativa de despliegue continuo usando GitOps para Kubernetes.[49]

GitOps es una forma de implementar despliegue continuo para aplicaciones, centrándose en la experiencia del desarrollador utilizando herramientas que le son familiares como Git, donde se almacenarán archivos con descripciones declarativas necesarias para la automatización de la salida a producción del sistema.[50]

3.7.2 Proveedores de servicios en nube

Estas son buenas opciones ya que quienes utilizan uno de estos proveedores cuentan con una solución de CD integrada y optimizada para funcionar con su entorno. Los principales proveedores en la actualidad son Amazon Web Services (más conocido por sus siglas AWS), Microsoft Azure, y Google Cloud.[51]

Code Deploy

Amazon Web Services provee un servicio de CD llamado Code Deploy que permite automatizar el proceso de despliegue de sus instancias en la nube.[52]

Azure Pipelines

Azure ofrece una herramienta completa de CI/CD llamada Azure Pipelines, esta permite no solo hacer el despliegue en Azure sino que en cualquier otro proveedor, incluso permite publicar aplicaciones en las tiendas de aplicaciones.[53]

Cloud Build

Google Cloud brinda un servicio llamado Cloud Build, este permite automatizar las compilaciones y despliegues de un sistema una vez se incorporan los cambios a una rama determinada de un repositorio de Git, similar a CodeDeploy de AWS.[54]

4. Prueba de concepto de prácticas y tecnologías

Con el objetivo de complementar la investigación realizamos una prueba de concepto creando una versión básica de un pipeline de CI/CD sobre un sistema ya desarrollado. A lo largo de este capítulo iremos detallando esta prueba de concepto; los objetivos iniciales que nos plateamos, el proyecto sobre el cual agregaremos el pipeline de CI/CD, las tecnologías que utilizaremos y los criterios de evaluación que tendremos en cuenta, explicaremos el pipeline que desarrollaremos, finalizando con nuestra evaluación, resultados y reflexiones sobre el pipeline obtenido y su implementación.

4.1 Objetivos

- Probar las principales tecnologías relacionadas con el ciclo de CI/CD sometiéndolas a un flujo definido de pruebas y a un análisis de aspectos que consideramos importantes para evaluar de forma objetiva cada resultado.
- Buscamos generar un ciclo básico de CI/CD, utilizando al menos un nivel de testing como herramienta de aseguramiento de calidad.
- Evaluar el impacto de tecnologías de utilizadas en un ciclo básico de entrega continua en la calidad de software.

- Realizar una análisis del esfuerzo de implementación de las distintas tecnologías basándonos en el tiempo que nos toma implementarlas en un sistema que fue creado sin utilizar CI/CD.

4.2 Proyecto de prueba de concepto

El proyecto a utilizar consiste en una aplicación SaaS que permite gestionar un catálogo y reservas de libros de una biblioteca. Dicha aplicación SaaS esta basada en una arquitectura en micro servicios en Node.js, Go y Python. Esta misma corre en AWS Beanstalk utilizando Docker, el sistema operativo es Ubuntu. En la figura 10 podemos ver el diagrama de despliegue de la SaaS con todos sus micro servicios y sus protocolos de comunicación. Fue elegida para ser adaptada al caso de uso debido a que:

- El uso de micro servicios permite dividir los componentes en módulos o servicios independientes, que pueden ser invocados a necesidad y cada pipeline puede ser desarrollado independientemente.
- Permite poder escalar la solución independientemente, dado que cada servicio es creado independientemente del otro.
- Permite reducir el riesgo de interrupciones, cada micro servicio tiene diseñada solamente una funcionalidad, entonces las fallas son mas fáciles de aislar.
- Permite reducir el lead time, debido a que cada servicio independiente puede ser enviado a producción tan pronto como sea desarrollado.

Sin embargo el uso de esta arquitectura para el contexto de entrega continua supone algunas desventajas:

- Se requiere una gran cantidad de deployment pipelines, esto puede ser una complicación para los equipos pequeños, debido a que cada pipeline debe ser

mantenido independientemente lo cual puede generar un costo extra para el equipo, si usan una arquitectura monolítica no ocurriría.

- Al tener una arquitectura de micro servicios, se pueden utilizar muchas tecnologías diferentes lo que produce que los releases se vuelvan complicados de administrar, dado que hay que tener expertos en cada tecnología utilizada.

- De existir una dependencia entre los servicios, puede suceder que las pruebas de integración y de carga se vuelvan un desafío.

Tomando en cuenta las desventajas nombradas previamente y el tiempo disponible para el desarrollo de este proyecto, optamos por seleccionar uno de los micro servicios para hacer la prueba de concepto. El micro servicio sobre el cual nos vamos a enfocar esta desarrollado en Node.js, en la versión 12. Este micro servicio es el Session Service, en la figura 11 podemos ver su diagrama de componentes, es el encargado del manejo del registro y el manejo de la sesión del usuario en el sistema.

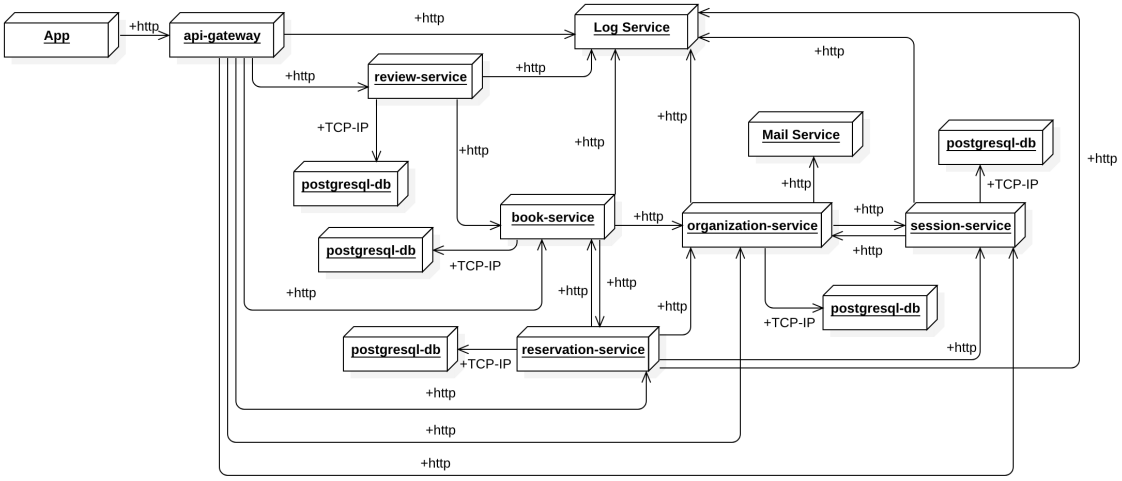

Figure 10: Diagrama de despliegue de la SaaS

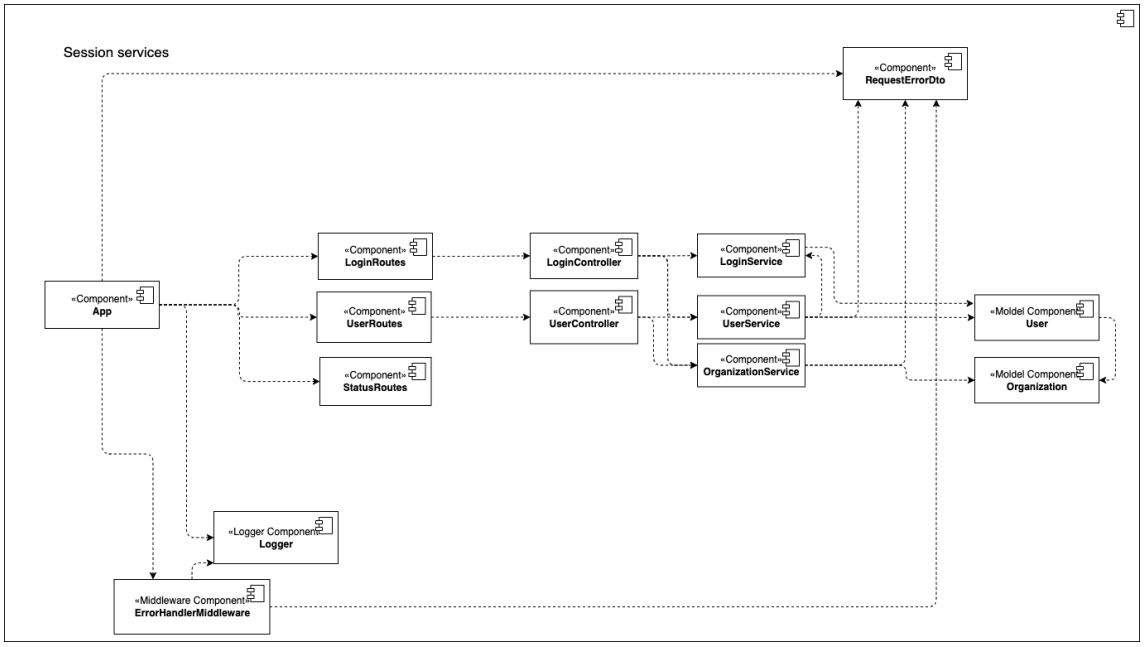

Figure 11: Diagrama de componentes del microservicio

4.3 Mapa sobre el estudio de las tecnologías

4.3.1 Criterio de evaluación de herramientas

De forma tal de ser objetivos con el uso de las herramientas, vamos a evaluarlas por los siguientes puntos:

1. Comunidad o empresas detrás del proyecto, tanto que lo soporten, como que lo usen.
2. Integración con otros servicios existentes.
3. Facilidad de uso y de aprendizaje.
4. Funcionalidades que proveen.
5. Costo y licenciamiento.

Nos importan las tendencias de las herramientas y entender hacia donde van las mismas. Intentaremos enfocarnos en la capa gratuita de estos servicios. Como mínimo en las herramientas de CI vamos a requerir que cumplan con: tener manejo de versionado y manejo de eventos que disparen pipelines, pero no vamos a enfocarnos en la rama de orquestación de estos servicios.

Tomaremos un enfoque pragmático, luego de evaluadas las herramientas por los puntos anteriormente mencionados, haremos tablas de comparación basadas en los puntos, para de esta forma llegar a las conclusiones finales.

4.3.2 Flujo del pipeline

Para ser lo más agnósticos posibles intentaremos someter a la misma prueba a todas las herramientas a probar, esta prueba es un de un flujo básico dentro de cualquier proyecto que cuente con CI/CD.

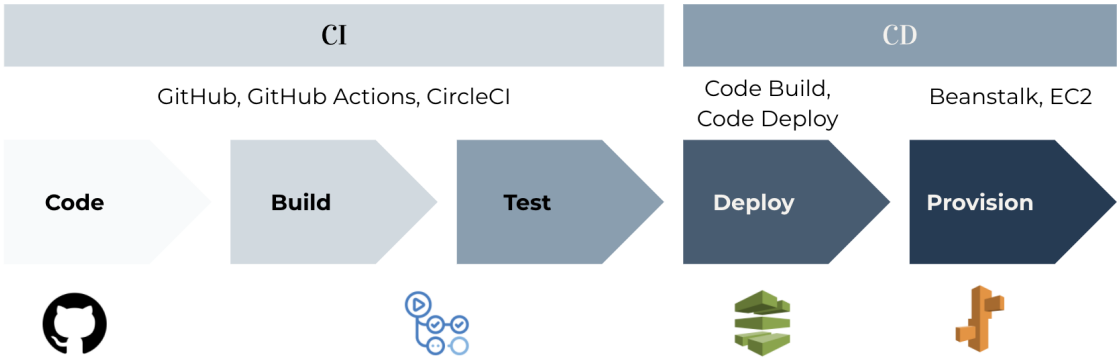

Figure 12: Pipeline básico de CI/CD

En la figura 12 vemos el flujo de CI/CD que implementaremos. Comienza con el desarrollo de un nuevo cambio que una vez terminado se sube al sistema de control de versionado en la nube, que en nuestro caso será GitHub, donde hacemos un Pull Request para incorporar los cambios a la versión estable de nuestro sistema. Al crearse el Pull Request se ejecutará la etapa de CI, donde GitHub Actions, Jenkins y Circle CI se encargarán de validar que la aplicación se pueda buildear, y de correr los

tests con el coverage correspondiente. Necesitaremos que alguien haga una revisión sobre nuestro código para validar el cambio antes de incorporarlo. De darse todos los pasos hasta el momento de forma correcta el cambio se incorporará a la rama 'master' o 'main' (rama con la versión estable del sistema) dando lugar a la etapa de CD, donde se activará de forma automática Code Deploy que se encargará de obtener el código para actualizarlo con la versión que se encuentra en el servidor.

A continuación en el diagrama 13 detallaremos más en lo que ocurre durante la etapa de CI dentro de nuestro ciclo.

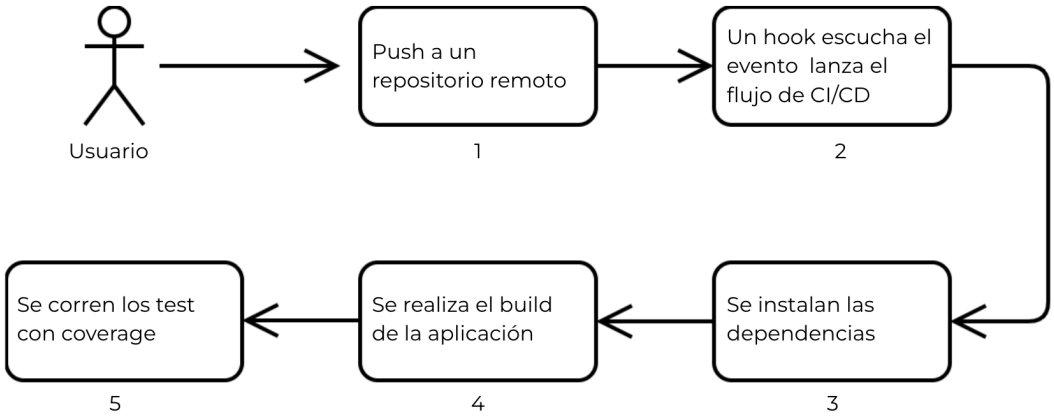

Figure 13: Pipeline a probar

Push a un repositorio remoto

Consiste en mandar los cambios realizados localmente al sistema de control de versiones distribuido. Este punto es muy común independientemente de la plataforma que utilizamos como sistema de manejo de versiones, aunque la más utilizada es GitHub como comentamos antes.

Hook que escucha el evento y lanza el flujo de CI

Un hook muy utilizado es GitHub Actions, aunque los pasos posteriores no sean corridos con el mismo, ya que este puede escuchar un evento y lanzar un Job

de Jenkins, etc. Se puede configurar que sea únicamente con pull requests a la rama "main", entre otros eventos. En nuestro caso será al crear un pull request.

Instalación de las dependencias

Para este punto es fundamental contar con un sistema de manejo de paquetes efectivo, ya que podría suceder que las dependencias a instalar en donde se corre el CI sean distintas a las que se usan en el equipo del desarrollador.

Build de la aplicación

En este paso para tecnologías como Typescript se podría correr el interprete directo sin compilar, pero al hacerlo conseguimos un entorno lo más parecido a producción posible.

Ejecución de los tests con coverage

Sobre este paso podemos definir ciertos umbrales de coverage, incluso sobre el código agregado en el nuevo pull request. En caso de que uno de estos pasos no pase satisfactoriamente el Pipeline deberá de fallar.

4.4 Desarrollo de prueba de concepto

4.4.1 Cobertura de código

Para el análisis de la cobertura de código se utilizará la herramienta Codecov, la cual es una herramienta que nos permitirá poder obtener insights del código, entre ellos se encuentran:

- **Runtime Insights:** Permite obtener información durante el runtime de la aplicación permitiendo obtener información contextual de los Pull Requests.
- **Cobertura de Código:** Permite observar la cobertura de código total y la cobertura que contiene el código fuente, pudiendo ser más rápido para detectar áreas de tests.
- **Multi Language, Multi CI/CD:** Permite usar CI/CD sin importar el lenguaje en que se encuentre el proyecto.

4.4.2 Testing

Como ya comentamos anteriormente nuestro caso práctico estará en Node.js, para esta tecnología los dos frameworks de testing son Mocha y Jest.

Mocha es un framework de prueba de JavaScript que se ejecuta en Node.js y en el navegador, lo que hace que las pruebas asincrónicas sean sencillas.

Jest es un framework de prueba de JavaScript mantenido por una empresa de la talla de Meta (anteriormente conocida como Facebook). Fue diseñado y construido con un enfoque en la simplicidad y el soporte para grandes aplicaciones. Funciona con muchas tecnologías como: TypeScript, Node.js, React, Angular, entre otras. No requiere de una gran configuración, algo que facilita su uso para usuarios nuevos.[37]

Si bien ambos sirven para poder cumplir con nuestro objetivo de calidad, Mocha es más flexible y configurable, y aunque Jest no necesita configuraciones adicionales está mayormente orientado a testear frontends web [39]. En base a esto decidimos utilizar Mocha para los tests de nuestro backend.

Por falta de tiempo no podremos implementar BDD en nuestra prueba de concepto, pero sabemos que es posible mediante la herramienta Mocha en conjunto con Chai. Esta última nos permite realizar los test con formato de BDD con una interfaz de expect/should junto con asserts tradicionales del formato de TDD. Al correrse con Mocha, los test de BDD ya estarían soportados dentro del pipeline que corre los

tests del sistema. Mocha además nos permite realizar pruebas de integración, que consiste en utilizar llamados del estilo HTTP al servicio y poder realizar las pruebas usando el framework de BDD. Este tipo de pruebas no pudimos realizarlas por falta de tiempo, pero en futuras investigaciones podrían implementarse.

4.4.3 GitHub Actions

Repositorio con la implementación del pipeline.

Empresa detrás del proyecto

GitHub Actions es una herramienta muy simple de utilizar, respaldada por una empresa del calibre de GitHub (Microsoft) que es líder en el versionado de código, que a la fecha cuenta con más de 74 millones de desarrolladores activos. [28]

Integración con otros servicios existentes

Se integra muy bien con otros servicios existentes, ya que posee extensiones o plugins los cuales agregan nuevos comportamientos al pipeline [35].

Plugins

Podemos encontrar en la tienda de plugins de GitHub Actions ESLint, un plugin de Code Linter para JavaScript.[61] También encontramos Security Checks como Nodejsscan, especifico para Node.[62]

Facilidad de uso y de aprendizaje

Al ser GitHub el mayor hosting para control de versiones, cuenta con ciertos beneficios, como por ejemplo es que resulta muy cómodo implementar nuestro

CI desde aquí, ya que no hay que subir nuestro código a ningún lugar extra, la configuración se hace mediante YAML que es un formato muy utilizado.

Funcionalidades que proveen

Permite escuchar eventos dentro de GitHub, por ejemplo, correr un pipeline al crear un nuevo Pull Request contra la rama "main", al crear un nuevo tag, etc. Respecto a integraciones es muy abierto, soporta correr contenedores de Docker, descargándolos de un Registry [36].

Costo y licenciamiento

Es gratuito hasta 500 MB y 2000 minutos de ejecución por mes, lo cual es más que suficiente para la mayoría de casos, ya que como se ve en la imagen 28 del anexo, nuestro ejemplo tomó 23 segundos en correr, eso nos daría unas 5217 ejecuciones por mes, o unas 170 ejecuciones por día.

4.4.4 Jenkins

Repositorio con el archivo obtenido de la implementación del pipeline.

Comunidad y empresas detrás del proyecto

Algunas empresas que respaldan el proyecto son Red Hat, AWS, GitHub, JFrog, entre otras. Al ser Open Source posee una gran cantidad de implementaciones desarrolladas por los millones de usuarios alrededor del mundo, tiene una gran comunidad y documentación.

Integración con otros servicios existentes

Se integra de forma muy fácil con otros servicios, ya que posee un extenso marketplace de plugins, e incluso se pueden lanzar Jobs de Jenkins desde una GitHub Action, por lo que ambos servicios se integran muy bien [30].

Plugins

Con Jenkis podemos utilizar Warnings Next Generation (un plugin de Code Linter) y OWASP Dependency-Check (para seguridad).[63][64]

Facilidad de uso y de aprendizaje

La herramienta no es muy simple de utilizar, es poco intuitiva y requiere bastante conocimiento previo para poder configurarla. A pesar de eso, como ya comentamos, posee una extensa documentación y una gran comunidad[30].

Funcionalidades que proveen

Jenkins es una herramienta muy completa y flexible, que permite correr pipelines cada cierto tiempo (utilizando cron expressions), hasta lanzar eventos cuando nuevo código se agregue al repositorio, deployar, etc. Además, Jenkins puede correr en un servidor propio, a diferencia de herramientas como GitHub Actions.

Actions vs Jobs

Una de las principales características de las tecnologías es cómo se definen los pipelines, para el caso de Jenkins se utiliza *Jobs* y GitHub maneja *Actions*. A modo de comparación de las metodologías de obtuvieron los siguientes resultados.

- **Facilidad de uso:** GitHub Actions permite ser configurado mediante un archivo YAML, el cual utiliza un sintaxis muy intuitiva y poco verbosa, en cambio los Jobs requieren un conocimiento previo para la configuración de los mismos y su archivo de configuración no utiliza un estándar conocido.
- **Sincronía:** Los jobs se ejecutan de manera sincrónica, por lo cual hace que los deploys y revisiones del pipeline se vuelvan costosas, sin embargo las actions pueden ser ejecutadas en paralelo.
- **Ambiente de ejecución:** Las actions pueden ser ejecutados en cualquier ambiente, sin embargo los jobs necesitan tener una instancia de Docker o servicio de la plataforma
- **Uso de caching:** Para el implementar el uso de caching dentro de los jobs hay plugins que lo permiten, sin embargo para las actions es necesario crear un mecanismo manual que permita poder implementar caching.

Costo y licenciamiento

Es totalmente gratuito, aunque debemos de tener un servidor donde correr los Pipelines. Jenkins es distribuido bajo la licencia MIT, lo cual nos permite utilizarlo de forma comercial [30].

4.4.5 CircleCI

Repositorio con la implementación del pipeline.

Empresa detrás del proyecto

Circle Internet Services es la empresa detrás de esta herramienta, se encuentra desde el 2011 en la industria, prometiendo calidad y efectividad a los equipos de desarrollo. Además de numerosos reconocimientos cuenta con la confianza de grandes empresas como Mercado Libre [42].

Integración con otros servicios existentes

Se integra muy bien con numerosos servicios existentes, incluso en su página permite sugerir servicios para integrarse que no estén en la lista, esta va desde Code Linters hasta servicios en la nube [42].

Facilidad de uso y de aprendizaje

Al ser una tecnología con tantos años y tanta aceptación por parte de grupos de trabajo es simple de implementar y muy clara la información que brinda. Es muy práctico que utilice el formato YAML al igual que las otras tecnologías.

Plugins

Tenemos disponible el plugin de seguridad OWASP y Linter de Code Linter.[65][66]

Funcionalidades que proveen

Permite escuchar eventos dentro de GitHub, por ejemplo, correr un Pipeline al crear un nuevo Pull Request contra la rama "main", correrla manualmente desde la web, etc. En cuanto a las integraciones tiene una variedad muy amplia y soporta correr contenedores de Docker. En caso de fallar el pipeline notifica a los responsables de repositorio. Adicionalmente ofrece la funcionalidad de hacer automáticamente el deploy del sistema. También ofrece seguridad ya sea que se utilice sus servidores o servidores propios, así como también ofrece métricas de los pipelines.[42]

Costo y licenciamiento

Es gratuito para un solo usuario, 30.000 minutos de ejecución por mes y hasta 30 jobs corriendo en simultaneo, lo cual es más que suficiente, en nuestro caso duró 28 segundos, lo que nos daría más de 78.000 ejecuciones por mes.[42]

4.4.6 Code Deploy

Code Deploy es una herramienta que provee Amazon Web Services para hacer los despliegues de forma automática. Hemos elegido de todas las opciones utilizar esta herramienta por las siguientes razones:

- Amazon Web Services es el proveedor de servicios en la nube más utilizado [55][56], esto nos permite encontrar una amplia variedad de documentación y guías sobre sus servicios.
- AWS tiene una capa gratuita amplia en los servicios básicos que se necesitan para poner a producción un sistema, incluida la herramienta de CD Code Deploy.[55]
- Code Deploy se puede integrar con GitHub Actions; la herramienta de CI que elegimos utilizar.

A continuación en la figura 14 veremos como esta compuesto el pipeline de CD con las herramientas elegidas.

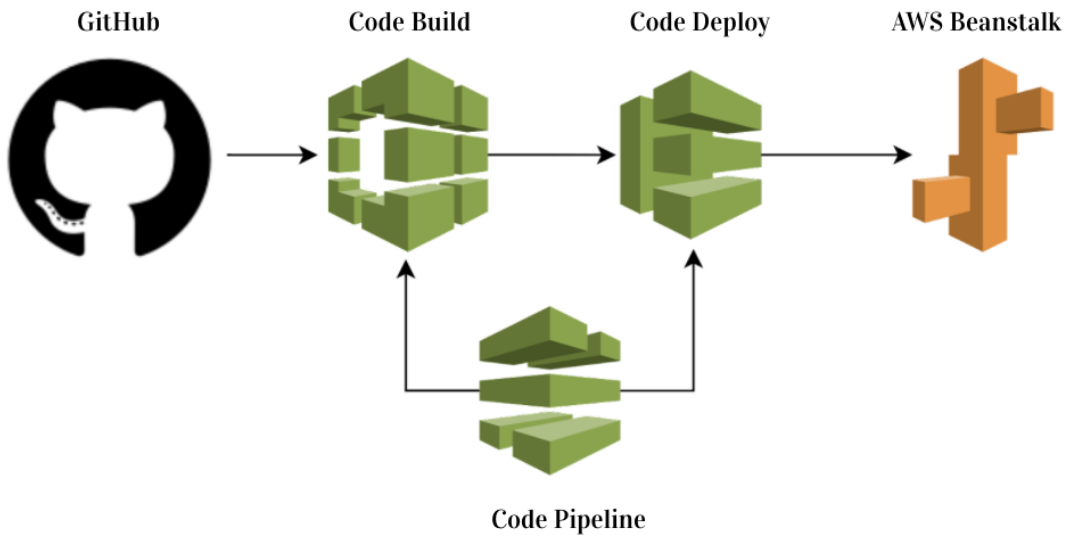

Figure 14: Pipeline de CD

El flujo completo, como comentamos anteriormente, consiste en el desarrollo de una nueva funcionalidad, esta es subida al sistema de gestión de versionado, donde se hace una solicitud para incorporar los cambios a una rama estable mediante un Pull Request. Al crearse el Pull Request se ejecutará la etapa de CI, que incluye la ejecución automática de build y tests, y la revisión manual de código. De darse todo de forma correcta el cambio se incorporará a la rama estable dando lugar a la etapa de CD, donde GitHub Actions activará Code Deploy que se encargará de obtener el código para actualizarlo el que se encuentra en el servidor de Beanstalk.

Beanstalk nos permite administrar aplicaciones en la nube rápidamente, sin necesidad de preocuparse por la infraestructura y sin limitar la capacidad de control. Esta herramienta se encargará de gestionar los detalles de aprovisionamiento de capacidad, balanceo de carga, escalado y monitorización de estado del sistema.[55]

Empresa detrás de la herramienta

Como se comentó anteriormente esta herramienta es brindada por AWS, el proveedor de servicios en la nube más utilizado.[55] Esto hace que se cuente con

mucha documentación así como una amplia comunidad.

Integración con otros servicios

Se integra con otros servicios de AWS, GitHub, también se integra con herramientas de CI como GitHub Actions, Jenkins y CircleCI.[55]

Facilidad de uso y de aprendizaje

La herramienta no es fácil de utilizar, se requiere de conocimiento previo sobre las herramientas de CI y despliegue en AWS (manejo de roles, EC2, configuración de instancias).

Costo y licenciamiento

Code Deploy es gratuito, se puede utilizar tanto con EC2 como con AWS Lambda, ambos servicios son gratuitos: lambda hasta 1 millón de solicitudes por mes y EC2 es gratis hasta 750 horas por mes (un mes tiene 744 horas, por lo que con una única instancia se puede utilizar sin generar costo).[55]

4.5 Resultados

4.5.1 Comparación entre las tecnologías de CI

Para la comparación de esfuerzo de aprendizaje utilizaremos una medida de dificultad siendo: alto, medio y bajo. Dicho criterio de comparación sera de carácter subjetivo de acuerdo a la dificultad que presentamos.

Empresa detrás del Proyecto		
Tecnología	Empresa	Observación
GitHub Actions	GitHub (Microsoft)	GitHub es una herramienta muy utilizada por lo desarrolladores con más de 74 millones de desarrolladores activos, fue adquirida por Microsoft.
Jenkins	Es un proyecto open source respaldado por Red Hat, AWS, GitHub, JFrog, entre otras	Jenkins es un proyecto open source, esto le trae una ventaja en comparación con las empresas anteriormente mencionadas, ya que recibe contribuciones de la comunidad y es respaldado por empresas muy grandes que se ven beneficiadas por el continuo crecimiento del proyecto.
CircleCI	Circle Internet Services	Circle Internet Services es una empresa que lleva muchos años en la industria en los cuales ha ganado distintos reconocimientos y se ganó la confianza de grandes empresas.

Costo y Licenciamiento		
Tecnología	Costo	Observación
GitHub Actions	Gratuito	GitHub Actions es gratuito hasta 500 MB y 2000 minutos de ejecución por mes, dado las pruebas que hicimos comprobamos que esto es suficiente para un desarrollo educativo.
Jenkins	Gratuito	Jenkins es gratuito, pero tener un servidor donde se corren los Pipelines es responsabilidad del usuario.
CircleCI	Gratuito	CircleCI al igual que GitHub Actions tiene un plan gratuito hasta determinado tiempo de ejecución, en este caso 30.000 minutos y 30 jobs corriendo en simultaneo. En un ámbito educativo esto resulta suficiente.

Conceptos similares			
	GitHub Actions	CircleCI	Jenkins
Workflow	Workflow es un concepto que abarca a todas las tecnologías de CI, es un proceso automatizado y configurable que se encarga de hacer todas las tareas necesarias hasta la llegada a producción del sistema.		
Archivo de configuración	GitHub Actions y CircleCI utilizan los archivos estilo YAML para la configuración de sus pipelines.		Jenkins se configura mediante una interfaz gráfica, por lo que no utilizamos archivos de configuración, al menos en un uso básico, suficiente para un uso educativo.

Funcionalidades que proveen		
GitHub Actions	CircleCI	Jenkins
• Escuchar eventos de GitHub • Servidor propio • Posibilidad de usar otro servidor • Permite ejecutar contenedores de Docker	• Escuchar eventos de GitHub • Ejecución manual de los Workflows • Notificación en caso de falla de evento • Hacer despliegue automático del sistema • Servidor propio • Posibilidad de usar otro servidor • Métricas en los pipelines	• Escuchar eventos de GitHub • Utilizar siempre servidor externo • Ejecutar pipelines cada cierto tiempo de forma automatizada

Integraciones con otros servicios existentes		
GitHub Actions	CircleCI	Jenkins
Permite múltiples integraciones con otros servicios existentes mediante extensiones y plugins.	En su página web lista los servicios con los cuales tiene integraciones y permite sugerir nuevos servicios para agregar.	Permite múltiples integraciones mediante plugins que tiene disponibles en su marketplace, incluso se puede integrar con GitHub Actions.

Plugins		
GitHub Actions	CircleCI	Jenkins
ESLint (Herramienta de Code Linter), Nodejsscan (Security Check)	OWASP (Security Check), Linter (Code Linter)	OWASP Dependency-Check (Security Check), Warnings Next Generation (Code Linter)

Facilidad de uso y aprendizaje		
GitHub Actions	CircleCI	Jenkins
Nivel de dificultad: **Bajo** Permite realizar todo el proceso de CI, desde el desarrollo del código hasta la integración del nuevo cambio. Al utilizar el formato YAML, como se ve en la figura 15, brinda mucha practicidad si se desea cambiar con otra tecnología como CircleCI. Tiene una documentación clara y en múltiples formatos. Resultó sencillo implementar un Workflow básico.	Nivel de dificultad: **Bajo** Al utilizar el formato YAML brinda mucha practicidad si se desea cambiar con otra tecnología como GitHub Actions, esto lo podemos apreciar comparando las figuras 15 y 16. Tiene una documentación clara y en múltiples formatos. Resultó sencillo implementar un workflow básico.	Nivel de dificultad: **Alto** No resultó simple de utilizar a pesar de su extensa documentación y su gran comunidad, la herramienta desde la cual se hacen las configuraciones no es intuitiva ya que tiene una interfaz gráfica confusa y muchos pasos a llevar a cabo para completar la configuración, como se puede ver en la figura 17.

```yaml
name: NodeJS Continious Integration

on:
  pull_request:
    branches: [ main ]

jobs:
  test_pull_request:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
      - uses: actions/setup-node@v1
        with:
          node-version: 12
      - run: npm i
        working-directory: ./app
      - run: sudo npm test
        working-directory: ./app
      - run: npm run clean
        working-directory: ./app
      - run: npm run build
        working-directory: ./app
```

Figure 15: Implementación de GitHub Actions

```yaml
version: 2.1

workflows:
  test-deploy:
    jobs:
      - test

jobs:
  test:
    docker:
      - image: cimg/node:16.13.1
    steps:
      - checkout
      - run:
          name: Install dependencies
          command: cd app && npm install
      - run:
          name: Run tests
          command: cd app && npm run test
      - run:
          name: Clean
          command: cd app && npm run clean
      - run:
          name: Build
          command: cd app && npm run build
```

Figure 16: Implementación de CircleCI

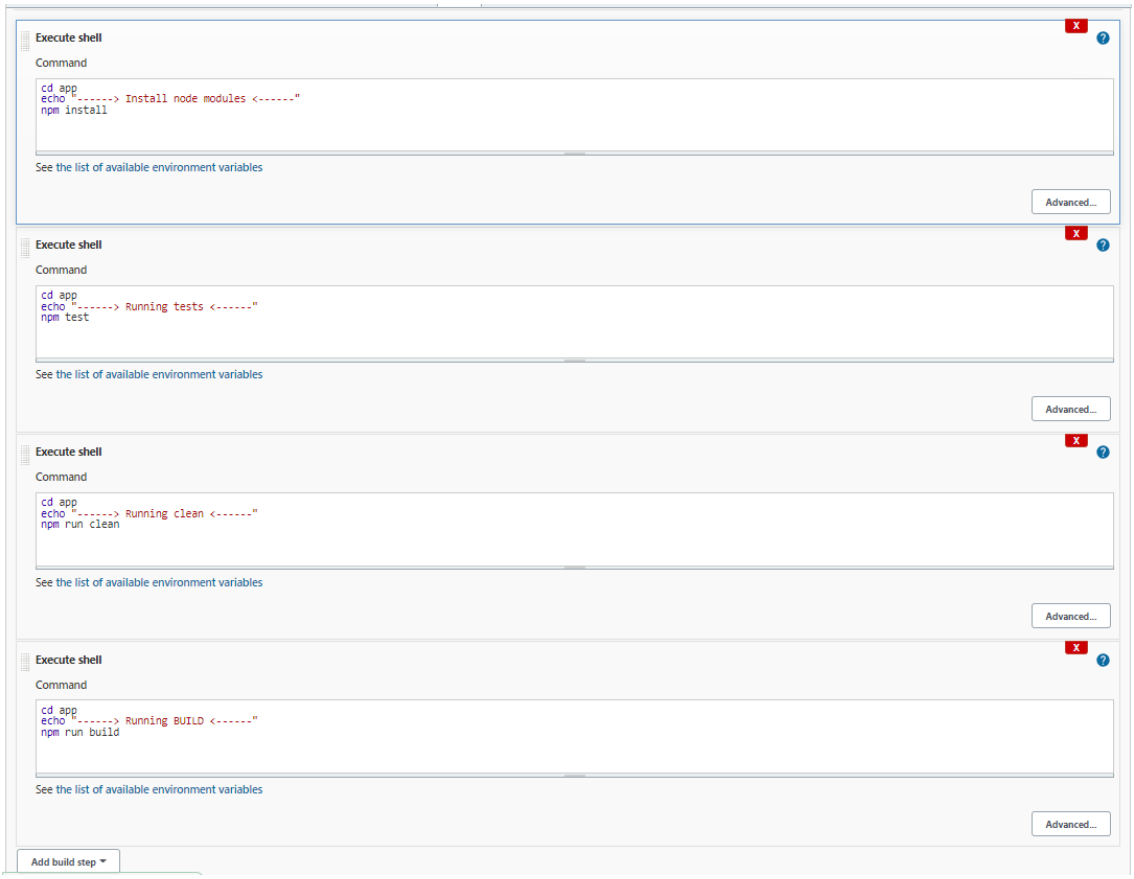

Figure 17: Implementación en Jenkins (pantalla de pasos a ejecutar)

4.5.2 Observaciones

Se investigaron seis herramientas para implementar CI, de las cuales se usaron las tres principales para la prueba de concepto. Se logró implementar exitosamente un pipeline de CI con GitHub Actions, Jenkins y CircleCI. Cada herramienta tomó un día de configuración, siendo GitHub Actions y CircleCI las más fáciles de implementar ya que eran simples de configurar y contaban con buena documentación. Luego de implementadas, con crear un PR en menos de un minuto obteníamos los resultados de los test realizados y no dejaba incorporar el cambio a menos que pase todas las pruebas cumpliendo con un coverage determinado.

Consideramos importante analizar a la empresa detrás de cada tecnología, porque las empresas grandes y las comunidades grandes nos aseguran un estándar de cal-

idad y documentación en el producto que estamos utilizando. GitHub Actions es una tecnología que provee GitHub cuya empresa madre es Microsoft, Jenkins es un proyecto open source que esta respaldado por Red Hat, AWS, GitHub, JFrog, entre otras, CircleCI es una tecnología brindada por Circle Internet Services; una empresa que lleva muchos años en la industria y ha logrado distintos reconocimientos y confianza de grandes empresas.

Desde el punto de vista del costo de cada herramienta probada, para un uso educativo este era gratuito; siendo Jenkins de código abierto, pero que requiere hosting externo, CircleCI y GitHub Actions brindando una capa gratuita de su servicio bajo ciertas condiciones (un límite de tiempo de ejecución mensual, un límite de espacio en los servidores o un límite en la cantidad de ejecuciones en simultáneo), pero comprobamos que en la práctica sería muy difícil llegar a sobrepasar los límites impuestos.

Las tres herramientas comparten el concepto de workflow ya que es común a todas las tecnologías de CI, pero se configuran de diferentes formas; CircleCI y GitHub Actions se configuran utilizando archivos YAML, Jenkins por su parte provee una interfaz gráfica.

Las tres tecnologías brindan funcionalidades básicas como: escuchar eventos en GitHub para poder ejecutar un workflow, notificar en caso de que el pipeline falle e integración con Docker. Luego cada herramienta brinda otro tipo de funcionalidades útiles como ejecutar manualmente los workflows (CircleCI), servidores propios donde ejecutar los pipelines (GitHub Actions y CircleCI), ejecutar pipelines cada cierto tiempo (Jenkins), entre otras.

Cuando tenemos un sistema ya desarrollado con un ecosistema de tecnologías funcionando correctamente y buscamos incorporar una nueva tecnología, en este caso de CI, es importante que esta sea compatible con lo que tenemos. Por esta razón es importante tomar en cuenta dentro del análisis las integraciones con los servicios existentes de las tres herramientas. Las tres herramientas brindan integraciones con múltiples servicios y plugins, además de continuamente estar agregando nuevos.

CircleCI y GitHub Actions fueron las más simples de implementar, utilizan el mismo formato de archivo de configuración que permite en tan solo unas líneas implementar un pipeline básico que corra las pruebas del sistema al escuchar un evento. Ambas tienen una documentación clara y en múltiples formatos. La ventaja de utilizar GitHub Actions es tener todo centrado en una única plataforma, sobre todo para el un uso básico, donde no se necesitan acciones muy complejas en el pipeline. Jenkins no fue simple de implementar a pesar de su extensa documentación, la interfaz no resultó ser lo suficientemente intuitiva.

Para la implementación de CD investigamos cuatro herramientas, de las cuales seleccionamos una para la implementación: Code Deploy. Elegimos esta herramienta ya que su proveedor es el más utilizado, lo que nos asegura que tendremos una amplia variedad de documentación, tiene una capa gratuita suficiente para un uso básico y permite distintas integraciones con herramientas de CI, como por ejemplo GitHub Actions, así como con otros servicios de AWS.

La implementación de esta herramienta fue más compleja comparada con las de CI, se requiere de conocer varios conceptos y manejar no solo herramientas de CI sino también de despliegue en AWS. Se requiere de más tiempo para poder aprender y entender todos los conceptos para poder estar preparado para implementar finalmente el pipeline de deploy, en nuestro caso nos tomó un esfuerzo de entre tres y cuatro días mientras que implementar las herramientas de CI nos requirió un día.

Obtuvimos como resultado un TTV de entre 3 y 4 minutos. Las métricas restantes estaban acopladas al contexto de negocio, estimaciones del equipo de desarrollo o un flujo constante de trabajo a lo largo del tiempo, datos que no pudimos obtener porque el objetivo de nuestra prueba de concepto no estaba orientado al estudio de las mismas. Estas métricas son ciertamente importantes ya que nos permiten evaluar el impacto del uso de entrega continua en nuestro sistema a lo largo del tiempo; analizar cuanto toma implementar y llevar a producción una nueva funcionalidad o solución de un error con esta práctica a lo largo de un período largo de tiempo, y compararlo con otras métricas de negocio para ver el impacto que tiene en los

usuarios.

4.5.3 Reflexión

Logros

Se logró investigar sobre las distintas tecnologías, la integración de estas con herramientas que permiten asegurar un estándar de calidad del sistema; frameworks de testing, code linters, code formatters y security checks.

Comprobamos que se puede generar un pipeline completo de CI/CD para un contexto básico de forma completamente gratuita en menos de cinco días. Estudiamos el impacto que tiene en la calidad de nuestro sistema agregando como requisito para la incorporación de un cambio que pasen los tests y una revisión de código manual, midiendo el tiempo entre que el cambio estaba listo hasta que llegaba a producción. El requerimiento de testing nos asegura que nuestro código es testeable, la revisión manual tiene en cuenta la seguridad del sistema, la calidad del código y de las pruebas unitarias realizadas, y el tiempo que demora el pipeline de CI en correr es de menos de 1 minuto, mientras que una vez incorporado el cambio el pipeline de CD demora entre 3 y 4 minutos, este tiempo nos asegura que cualquier arreglo podrá ser puesto en producción.

Concluimos que efectivamente las tecnologías y herramientas nos ayudan como equipo a lograr establecer un estándar de codificación, seguridad y testing de nuestro código para poder incorporar cambios a las versiones estables del software. Estas herramientas nos ayudan dado que se incorporan al pipeline de desarrollo siguiendo ciertas reglas si estas no son cumplidas se interrumpe el ciclo generando una cultura de trabajo en la cual el equipo se adhiere. Además nos impacta en otro atributo de calidad como la modificabilidad; obtenemos un código homogéneo, testeado y seguro, que se prueba y verifica antes de cada cambio (en la etapa de integración), antes de llegar a los usuarios (en la etapa de deploy).

Dificultades

En un comienzo, durante las investigaciones nos enfrentamos a amplio abanico de herramientas, por el alcance del proyecto no era viable investigarlas todas por lo que nuestra primer dificultad fue elegir cuáles íbamos a estudiar y cuáles iban a quedar afuera.

Durante la implementación de la prueba de concepto tuvimos dificultades para implementar Jenkins, ya que la interfaz de usuario no era clara. La implementación del pipeline de despliegue también nos presentó dificultades, porque debimos estudiar e investigar sobre tecnologías de AWS para poder implementarlo, sin estos conocimientos resulta difícil entender la documentación.

5. Conclusiones

En este proyecto realizamos un estudio de las bases conceptuales de entrega continua, las tecnologías para implementar pipelines de desarrollo de software y el impacto que tiene su uso en la calidad. El uso de entrega continua se ha vuelto clave para generar software de calidad, nos permite tener un desarrollo de software ágil, con entrega continua de software funcionando para obtener retroalimentación de los usuarios que puede rápidamente ponerse en práctica en una siguiente versión.

La entrega continua fomenta las puestas en producción de nuevas versiones del sistema en períodos cortos de tiempo; semanas o incluso días. Esto tiene un gran impacto en la calidad, ya que cualquier error es encontrado de forma temprana y reparado.

CI nos brinda la posibilidad de mantener un estándar de calidad de software para permitir que este se integre al sistema; un coverage mínimo de testing, un formato de código determinado, un estándar de seguridad definido. Una vez que el cambio cumple con los atributos de calidad definidos pasa a integrarse con el sistema de forma local, para ir a producción utilizamos CD.

En la etapa de salir a producción también nos aseguramos de mantener un estándar de calidad en nuestro sistema, con CD podemos validar previamente que nuestro sistema se cree correctamente, pase todas las pruebas y nos permite manejar el tipo de deploy adecuado para generar el mejor impacto en los usuarios.

El uso de las prácticas de CI/CD acompañadas de la cultura de DevOps en la cual

el equipo de desarrollo y de operaciones trabajan en conjunto para brindar sistemas donde la calidad comienza desde adentro del proceso, nos permite generar software testeable, mantenible, modificable, seguro y escalable.

5.1 Evaluación del impacto en la calidad

El objetivo de esta investigación fue el estudio de las bases conceptuales de la entrega continua y de las tecnologías para implementar pipelines de desarrollo de software y su impacto en la calidad.

Para esto realizamos una prueba de concepto que tenía como objetivos probar las principales tecnologías relacionadas con CI/CD, sometiéndolas a un flujo ya definido de pruebas y a un análisis de aspectos que consideramos importantes para evaluar de forma objetiva cada resultado obtenido.

Logramos completar una investigación sobre las distintas tecnologías de entrega continua, así como su integración con herramientas como frameworks de testing, linters y security checkers, que nos permiten impactar en la calidad del software desde el código a medida que llevamos a la práctica este sistema de entrega de software de forma continua.

Encontramos que con el uso de la capa gratuita de los servicios y con moderado esfuerzo podemos desarrollar un pipeline completo. Esto no solo nos permite mejorar la experiencia del usuario gracias a la obtención de feedback continuo, sino que también apoyándonos en las posibilidades de integración con otras herramientas más orientadas al impacto en la calidad podemos generar un código testeable, limpio, seguro y modificable que se va construyendo siguiendo un estándar de calidad en cada cambio desarrollado.

Para lograr nuestros objetivos nos enfrentamos a tres grandes dificultades, la primera fue elegir cual de todas las tecnologías investigadas íbamos a poner a prueba, luego durante de desarrollo tuvimos dificultades a la hora de utilizar Jenkins y así como

de implementar el pipeline de deploy. Este último requería de un estudio previo para entender y saber manejar conceptos básicos del uso de AWS, sin estos resulta más difícil entender la documentación.

5.1.1 Lecciones aprendidas

Durante el desarrollo de esta tesis hemos aprendido y reforzado el conocimiento de tecnologías y conceptos que nos encontramos en nuestro día a día como profesionales, algunos de ellos fueron: Jenkins, CircleCI, Code Deploy, Chef, Puppet y Security Checks. Aprendimos sobre el impacto que tiene el uso de herramientas de CI/CD en la calidad del software.

Nos enfrentamos a un desafío completamente diferente a lo que veníamos acostumbrados, sin tener una letra que nos dijera qué hacer tuvimos que trabajar en equipo junto con nuestro tutor para obtener una tesis que mostrará el estado del arte tecnológico relacionado con la entrega continua de una forma clara y objetiva.

Lo más difícil durante el desarrollo fue limitar; elegir qué tecnologías íbamos a utilizar, qué conceptos íbamos a estudiar y plasmar en la tesis y quitar aquello que, si bien nos pareció interesante, no aportaba verdadero valor al documento.

5.1.2 Mejoras del pipeline CI/CD

Durante el desarrollo de la tesis hemos podido investigar diferentes tecnologías pero se ha acotado el alcance para ser implementadas en el pipeline, algunas de las mejoras a implementar son:

- Ejecutar el proceso de CI dentro de contenedores.
- Ejecutar linting dentro del CI.
- Realizar revisiones de seguridad dentro del CI.

- Ejecutar una prueba de aceptación durante el proceso de CD.

5.2 Líneas futuras de investigación

Esta investigación da lugar a profundizar a futuro en las herramientas que por cuestiones de alcance tuvieron que ser dejadas afuera, tanto de CI como de CD. Siendo que estas últimas son más complejas de implementar hubo que acotar las que se llevaron a la práctica, obteniendo de esa forma la experiencia desde el punto de vista de una sola tecnología. Por lo tanto, en futuras líneas de investigación se buscaría:

- Experimentar durante el desarrollo del sistema, aplicando las tecnología de CI/CD con las herramientas de calidad investigadas y medir de forma práctica el impacto en la calidad reflejado en el producto final de software obtenido.
- Experimentar con una variedad más amplia de tecnologías de CI/CD.
- Desarrollar una prueba de concepto con un sistema que ya tenga implementado un pipeline de CI/CD y estudiar la facilidad de migrar entre tecnologías para este tipos de sistemas ya en producción.
- Desarrollar casos de mantenimiento del pipeline y evaluar el impacto de la integración de nuevo código.
- Realizar medición de métricas.. como tiempo de ciclo, en un ambiente orientado a producción.

Bibliography

[1] J. Humble and D. Farley, *Continuous delivery*, 1st ed. Upper Saddle River, NJ, NY: Addison-Wesley, 2011.

[2] G. Kim, P. Debois, J. Willis, J. Humble, and J. Allspaw, *The devops handbook: How to create world-class agility, reliability, &; Security in Technology Organizations*, 1st ed. Portland, OR, Oregon: IT Revolution Press, LLC, 2021.

[3] J. Turnbull, *The docker book*, 1st ed. Brooklyn, NY: Turnbull Press, 2014.

[4] N. Poulton, *The kubernetes book*, 1st ed. Wroclaw, Poland: Amazon, 2021.

[5] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 3rd ed. Boston: Addison-Wesley, 2022.

[6] M. Ivan, R. M. Soley, N. Ali, J. Grundy, and B. Tekinerdogan, *Software quality assurance: In large scale and complex software-intensive systems*, 1st ed. Waltham, MA: Morgan Kaufmann, 2016.

[7] M. Taylor and S. Vargo, *Learning chef: A guide to configuration management and automation*, 1st ed. Beijing: O'Reilly, 2014.

[8] J. Rhett, *Learning puppet 4: A guide to configuration management and automation*, 1st ed. Sebastopol, CA: O'Reilly, 2016.

[9] S. Patterson, *Learn AWS serverless computing: A beginner's guide to using AWS Lambda, amazon api gateway and services from Amazon Web Services*, 1st ed. Birmingham: Packt Publishing, 2019.

[10] M. Wynne, A. Hellesøy, and S. Tooke, *The cucumber book: Behaviour-driven development for testers and developers*, 1st ed. Dallas: The Pragmatic Bookshelf, 2017.

[11] B. Beyer, C. Jones, and J. Petoff, *Site Reliability Engineering: How Google Runs Production Systems*, 1st ed. Sebastopol, CA: O'Reilly Media, Inc., 2016.

[12] Atlassian. Building a devops culture, [Online]. Available: https://www.atlassian.com/team-playbook/examples/devops-culture (visited on 09/29/2021).

[13] Dewzilla. Get the fundamentals of devops right - then worry about tools, [Online]. Available: https://dewzilla.com/get-the-fundamentals-of-devops-right-then-worry-about-tools (visited on 09/29/2021).

[14] GitLab. Four continuous integration metrics you should follow, [Online]. Available: https://about.gitlab.com/topics/ci-cd/continuous-integration-metrics/ (visited on 10/10/2021).

[15] A. Jagarapu. Puppet use cases, [Online]. Available: https://www.ais.com/puppet-use-cases-how-to-troubleshoot/ (visited on 02/15/2022).

[16] P. Caroli. Continuous delivery: Lead time and cycle time, [Online]. Available: https://www.caroli.org/en/continuous-delivery-lead-time-and-cycle-time/ (visited on 10/13/2021).

[17] W. testing. La famosa pirámide de cohn y la dura realidad, [Online]. Available: https://medium.com/\@wc.testing.qa/la-famosa-pir\%C3\%A1mide-de-cohn-y-la-dura-realidad-e1250dfbe5f3/ (visited on 10/16/2021).

[18] M. Fowler. Martin fowler blog, [Online]. Available: https://martinfowler.com/ (visited on 10/16/2021).

[19] D. Northon. Bdd, [Online]. Available: http://dannorth.net/introducing-bdd/ (visited on 10/16/2021).

[20] S. F. Conservancy. Git, [Online]. Available: https://git-scm.com/ (visited on 10/16/2021).

[21] D. Northon. Sre vs devops, [Online]. Available: http://dannorth.net/introducing-bdd/ (visited on 10/20/2021).

[22] M. Fowler. Continuous integration, [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html (visited on 10/20/2021).

[23] Mike Beedle, Martin Fowler, Arie van Bennekum, Alistair Cockburn. Agile manifesto principles, [Online]. Available: https://agilemanifesto.org/principles.html (visited on 09/29/2021).

[24] M. Fowler. Continuous delivery, [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html (visited on 10/10/2021).

[25] M. Cristiá. Introducción al testing de software, [Online]. Available: https://www.fceia.unr.edu.ar/ingsoft/testing-intro-a.pdf (visited on 11/13/2021).

[26] C. Rodriguez. (Feb. 11, 2001). Automatizar pruebas de software: cuándo y por qué?, [Online]. Available: https://cl.abstracta.us/blog/automatizar-pruebas-de-software/ (visited on 11/19/2021).

[27] K. Hughes. Bitbucket vs github vs gitlab, [Online]. Available: https://stackshare.io/stackups/bitbucket-vs-github-vs-gitlab (visited on 01/27/2022).

[28] Github users, [Online]. Available: https://github.com (visited on 08/27/2021).

[29] Prometheus, [Online]. Available: https://prometheus.io/ (visited on 11/2021).

[30] Jenkins, [Online]. Available: https://www.jenkins.io/ (visited on 2021).

[31] Circleci, [Online]. Available: https://circleci.com/product/ (visited on 11/20/2021).

[32] Travis, [Online]. Available: https://docs.travis-ci.com/ (visited on 11/2021).

[33] Github actions, [Online]. Available: https://docs.github.com/es/actions/learn-github-actions/understanding-github-actions (visited on 2021).

[34] Aws code pipeline, [Online]. Available: https://aws.amazon.com/codepipeline/ (visited on 12/2021).

[35] Actions marketplace, [Online]. Available: https://github.com/marketplace?type=actions (visited on 2021).

[36] Registry container, [Online]. Available: https://docs.github.com/en/actions/using-containerized-services/about-service-containers#creating-service-containers (visited on 2021).

[37] Jest framework, [Online]. Available: https://en.wikipedia.org/wiki/Jest_(JavaScript_framework) (visited on 01/20/2022).

[38] Mocha framework, [Online]. Available: https://mochajs.org) (visited on 01/20/2022).

[39] K. Wheeler. Mocha vs. jest, [Online]. Available: https://medium.com/@wheeler.katia/jest-vs-mocha-whats-the-difference-235df75ffdf3 (visited on 01/20/2022).

[40] About continuous integration, [Online]. Available: https://docs.github.com/es/actions/automating-builds-and-tests/about-continuous-integration (visited on 2022).

[41] Github actions cheat sheet, [Online]. Available: https://resources.github.com/whitepapers/GitHub-Actions-Cheat-sheet/#:~:text=YAML\%20stands\%20for\%20\%E2\%80\%9CYet\%20Another,basis\%20of\%20its\%20configuration\%20workflows.&text=The\%20configuration\%20file\%20that\%20defines\%20your\%20GitHub\%20Actions\%20workflow. (visited on 2022).

[42] Circleci, [Online]. Available: https://circleci.com (visited on 2021).

[43] Circleci app, [Online]. Available: https://app.circleci.com (visited on 2021).

[44] Codecov, [Online]. Available: https://about.codecov.io/ (visited on 2021).

[45] Por qué son importantes las métricas dora, [Online]. Available: https://www.blueoptima.com/blog/how-to-measure-devops-success-why-dora-metrics-are-important (visited on 2021).

[46] D. G. Portman. Dora, [Online]. Available: https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance (visited on 12/2021).

[47] Beneficios del uso de linters, [Online]. Available: https://www.perforce.com/blog/qac/what-lint-code-and-why-linting-important (visited on 12/2021).

[48] Advantages and disadvantages of forced code reformat, [Online]. Available: https://softwareengineering.stackexchange.com/questions/92256/advantages-and-disadvantages-of-forced-code-reformat (visited on 12/2021).

[49] Argo cd documentation, [Online]. Available: https://argo-cd.readthedocs.io/en/stable/ (visited on 2021).

[50] Gitops documentation, [Online]. Available: https://www.gitops.tech/ (visited on 2021).

[51] A. Weissberger. Gartner: Aws, azure, and google cloud top rankings for cloud infrastructure and platform services, [Online]. Available: https://techblog.comsoc.org/2021/08/16/gartner-aws-azure-and-google-cloud-top-rankings-for-cloud-infrastructure-and-platform-services/ (visited on 12/2021).

[52] Aws codedeploy documentation, [Online]. Available: https://docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html (visited on 2022).

[53] Azure pipelines documentation, [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops (visited on 12/18/2021).

[54] Cloud build documentation, [Online]. Available: https://cloud.google.com/run/docs/continuous-deployment-with-cloud-build (visited on 01/25/2022).

[55] Amazon web services, [Online]. Available: https://aws.amazon.com/ (visited on 01/2022).

[56] Amost popular cloud platforms stack overflow poll, [Online]. Available: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-cloud-platforms (visited on 12/22/2021).

[57] Code quality metrics: Is your code any good?., [Online]. Available: https://www.sealights.io/code-quality/code-quality-metrics-is-your-code-any-good/ (visited on 2021).

[58] What is a pull request?, [Online]. Available: http://oss-watch.ac.uk/resources/pullrequest/ (visited on 2021).

[59] J. Reynolds. A brief history of configuration management., [Online]. Available: https://www.enov8.com/blog/a-brief-history-of-configuration-management/ (visited on 11/2021).

[60] G. C. Tech. What's the difference between devops and sre?, [Online]. Available: https://www.youtube.com/watch?v=uTEL8Ff1Zvk.

[61] Eslist plugin, [Online]. Available: https://github.com/marketplace/actions/run-eslint (visited on 02/02/2022).

[62] Nodejsscan plugin, [Online]. Available: https://github.com/marketplace/actions/nodejsscan (visited on 02/02/2022).

[63] Owasp dependency-check plugin, [Online]. Available: https://plugins.jenkins.io/dependency-check-jenkins-plugin/ (visited on 02/02/2022).

[64] Warnings next generation plugin, [Online]. Available: https://plugins.jenkins.io/warnings-ng/ (visited on 02/02/2022).

[65] Owasp plugin, [Online]. Available: https://circleci.com/developer/orbs/orb/entur/owasp (visited on 02/02/2022).

[66] Linter plugin, [Online]. Available: https://circleci.com/developer/orbs/orb/reactioncommerce/linter (visited on 02/02/2022).

[67] J. A. y Paul Hammond. 10+ deploys per day: Dev and ops cooperation at flickr, [Online]. Available: https://www.youtube.com/watch?v=LdOe18KhtT4 (visited on 02/12/2022).

[68] Airbnb javascript code standard, [Online]. Available: https://github.com/airbnb/javascript (visited on 02/12/2022).

[69] Google javascript code standard, [Online]. Available: https://google.github.io/styleguide/jsguide.html (visited on 02/12/2022).

[70] R. Wilsenach. Devopsculture, [Online]. Available: https://martinfowler.com/bliki/DevOpsCulture.html (visited on 02/10/2022).

[71] Newrelic, [Online]. Available: https://newrelic.com/ (visited on 02/12/2022).

[72] Data dog, [Online]. Available: https://www.datadoghq.com/ (visited on 02/13/2022).

Anexo

Tecnologías

Jenkins

Para este ejemplo intentaremos generar un Pipeline para nuestra aplicación para cada nuevo build que se haga contra la rama "jenkins-demo". Este pipeline correrá la misma versión de Node que estamos utilizando (Node.js v12) en un contenedor Docker, que también es la infraestructura utilizada en producción con AWS Beanstalk.

Lo funcionalidad esperada es:

- Instalar las dependencias.
- Verificar que correr los tests.
- Verificar que la aplicación se puede buildear para producción.

En caso de que uno de estos pasos no pase satisfactoriamente el Pipeline deberá de fallar.

Para configurar un nuevo proyecto en Jenkins hay que seguir los siguientes pasos.

Generar Proyecto

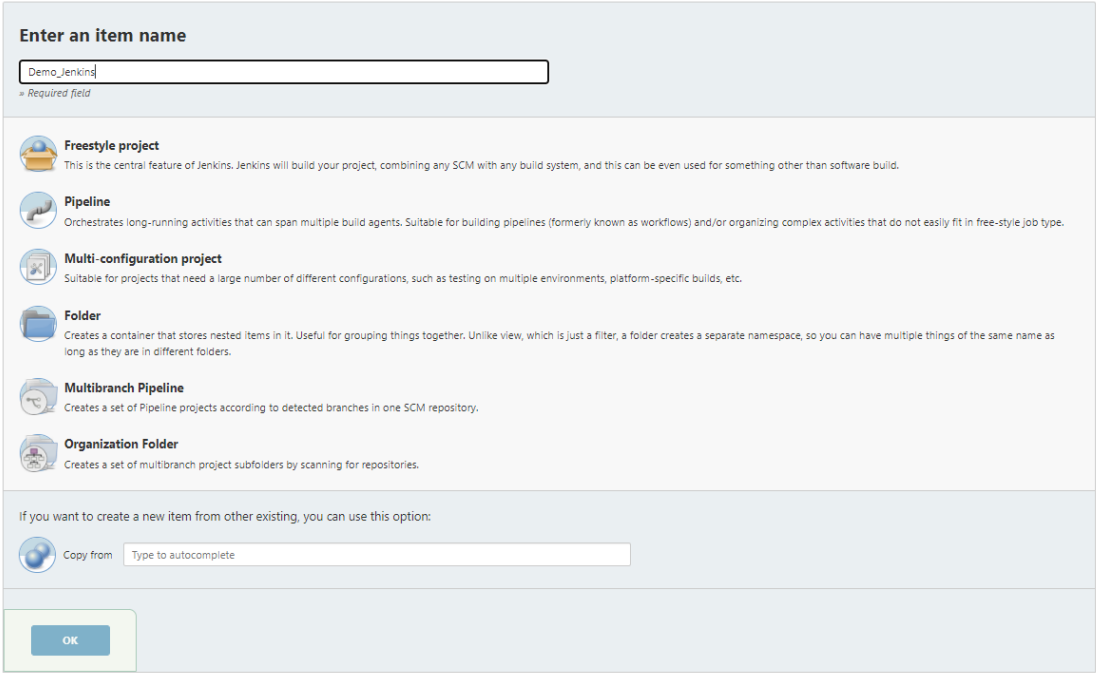

Figure 18: Proyecto de Jenkins

Luego presionar el botón crear y luego elegiremos las opciones para nuestro proyecto.

Primero crearemos la conexión con nuestro proyecto en GitHub. Para lo cual se configuran los siguientes parámetros.

- **URL**
- **Credenciales**
- **Branch**

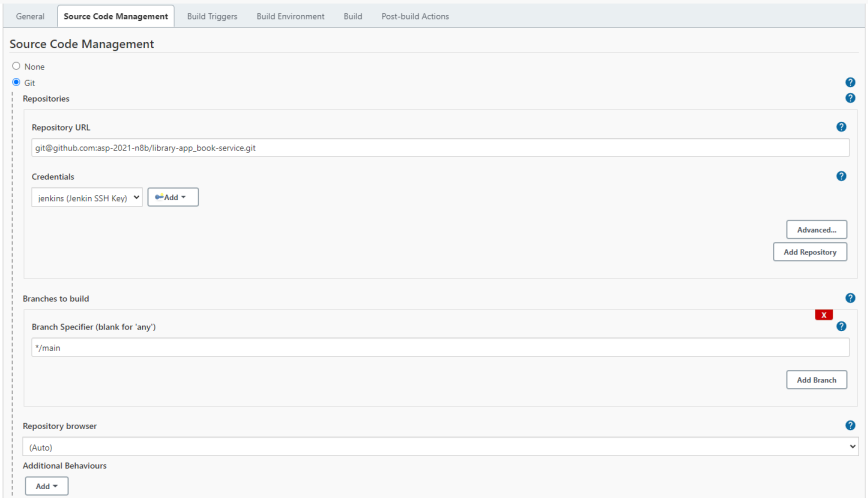

Figure 19: Configuración de SCM en Jenkins

Luego generaremos las configuración de eventos (triggers) y el ambiente de ejecución (Node.js). Las cuales se generan las siguientes configuraciones:

- **Git Hook** Esto permitirá a Jenkins poder ejecutar un workflow luego que se realice un push
- **Build Environment** Se usara un ambiente de Node.js previamente configurado

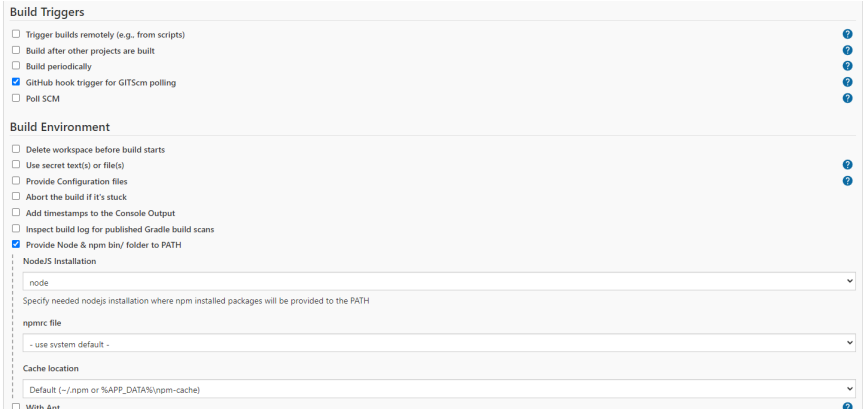

Figure 20: Build Environment

Finalmente configuraremos los pasos a ejecutar, estos son:

- **Instalar NodeJS**
- **Ejecutar los tests**
- **Limpiar el ambiente**
- **Generar un compilado de la aplicación**

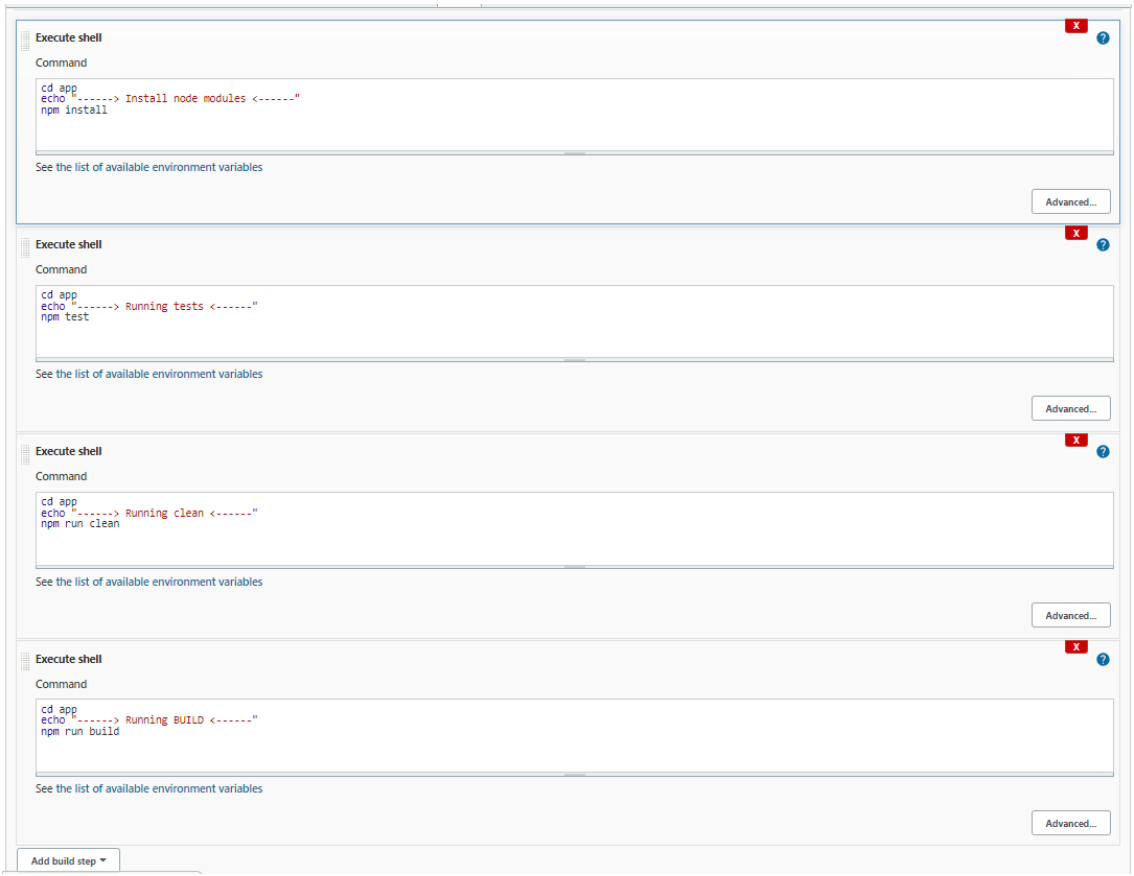

Figure 21: Pasos

El resultado obtenido de las pruebas de código

```
  10 passing (575ms)

-------------------------------------|---------|----------|---------|---------|--------------------
File                                 | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------------------------|---------|----------|---------|---------|--------------------
All files                            |   51.72 |    30.81 |   39.06 |   52.05 |
 src                                 |      72 |      100 |   55.55 |   77.27 |
  app.ts                             |      72 |      100 |   55.55 |   77.27 | 60,70-85
 src/cache                           |   58.82 |       50 |      50 |   58.82 |
  implementation.cache.ts            |     100 |      100 |     100 |     100 |
  interface.cache.ts                 |     100 |      100 |     100 |     100 |
  local.cache.ts                     |      50 |       50 |      50 |      50 | 18-31
 src/controllers                     |   31.48 |      100 |   14.28 |   31.48 |
  book.controller.ts                 |    6.66 |      100 |       0 |    6.66 | 6-65
  login.controller.ts                |     100 |      100 |     100 |     100 |
  organization.controller.ts         |   36.36 |      100 |   33.33 |   36.36 | 8-34,47
  reservation.controller.ts          |   19.04 |      100 |       0 |   19.04 | 8-39
  status.controller.ts               |   42.85 |      100 |       0 |   42.85 | 7-11
  user.controller.ts                 |   26.66 |      100 |       0 |   26.66 | 8-26
 src/database                        |    87.5 |       50 |     100 |    87.5 |
  setup.ts                           |    87.5 |       50 |     100 |    87.5 | 12
 src/database/models                 |   91.66 |        0 |   66.66 |   95.65 |
  book.model.ts                      |      75 |        0 |       0 |   85.71 | 23
  organization.model.ts              |     100 |      100 |     100 |     100 |
  reservation.model.ts               |     100 |      100 |     100 |     100 |
  user.model.ts                      |     100 |      100 |     100 |     100 |
 src/dtos                            |     100 |       50 |     100 |     100 |
  requestError.dto.ts                |     100 |       50 |     100 |     100 | 5-6
 src/email                           |      60 |      100 |       0 |      60 |
  implementation.email.ts            |     100 |      100 |     100 |     100 |
  interface.email.ts                 |     100 |      100 |     100 |     100 |
  sendGrid.email.ts                  |      50 |      100 |       0 |      50 | 13-31
 src/logger                          |   45.03 |    31.57 |   39.39 |   44.61 |
  implementation.logger.ts           |     100 |       50 |     100 |     100 | 7
  newRelic.logger.js                 |   38.46 |    31.08 |   28.57 |   38.46 | ...,219-220,231-255
  winstonTransportNewRelic.logger.ts |     100 |      100 |     100 |     100 |
 src/middlewares                     |   53.19 |    44.82 |   55.55 |   53.19 |
  errorHandler.middleware.ts         |      50 |    14.28 |     100 |      50 | 20-33
  token.middleware.ts                |   54.16 |    54.54 |      50 |   54.16 | ...3,91,109-124,133
 src/routes                          |     100 |      100 |     100 |     100 |
  book.routes.ts                     |     100 |      100 |     100 |     100 |
  login.routes.ts                    |     100 |      100 |     100 |     100 |
  organization.routes.ts             |     100 |      100 |     100 |     100 |
  status.routes.ts                   |     100 |      100 |     100 |     100 |
  user.routes.ts                     |     100 |      100 |     100 |     100 |
 src/services                        |   41.33 |    17.07 |      25 |   41.61 |
  book.service.ts                    |   21.27 |        0 |       0 |   21.73 | 22-130
  login.service.ts                   |     100 |      100 |     100 |     100 |
  organization.service.ts            |   72.97 |     37.5 |    62.5 |   72.97 | 34-35,46-66,72
  reservation.service.ts             |   15.38 |        0 |       0 |   15.38 | 10-104
  status.service.ts                  |      50 |      100 |       0 |      50 | 6-10
  user.service.ts                    |      60 |    66.66 |   33.33 |      60 | 16-26
 src/utils                           |   51.78 |    28.57 |   63.63 |   52.83 |
  environment.ts                     |   86.66 |       75 |   83.33 |   86.66 | 28,34
  roles.ts                           |   83.33 |       50 |     100 |   83.33 | 9
  uuid.ts                            |     100 |      100 |     100 |     100 |
  validators.ts                      |       4 |        0 |       0 |    4.34 | 3-29,34-38
-------------------------------------|---------|----------|---------|---------|--------------------
```

Figure 22: Resultado de Pruebas

Circle CI

Para configurar un nuevo workflow en CircleCI debemos crear un archivo con el nombre “config“ dentro de la carpeta “.circleci/“ con la extensión .yml. También vamos a especificar dentro del archivo de configuración la versión de configuración que vamos a utilizar.

En el caso previamente mencionado, el archivo sería el siguiente:

```
version: 2.1

workflows:
  test-deploy:
    jobs:
      - test

jobs:
  test:
    docker:
      - image: cimg/node:16.13.1
    steps:
      - checkout
      - run:
          name: Install dependencies
          command: cd app && npm install
      - run:
          name: Run tests
          command: cd app && npm run test
      - run:
          name: Clean
          command: cd app && npm run clean
      - run:
```

```
name: Build
command: cd app && npm run build
```

Al crear un nuevo Pull Request contra la rama “main“ o al correr manualmente desde la web (https://app.circleci.com/) se ejecutaran los jobs de los workflows.

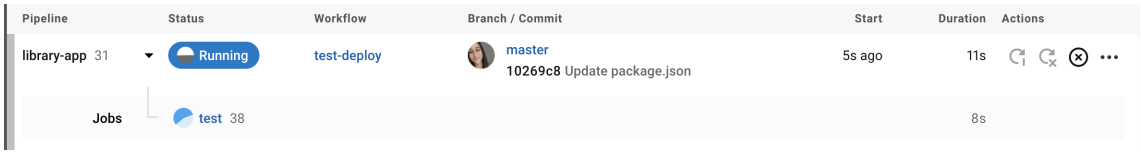

Figure 23: Workflow corriendo [43]

Si entramos dentro de Job podremos ver el Log de cada paso ejecutado, ahí podremos ver si falla cual es la razón para fácilmente arreglarlo.

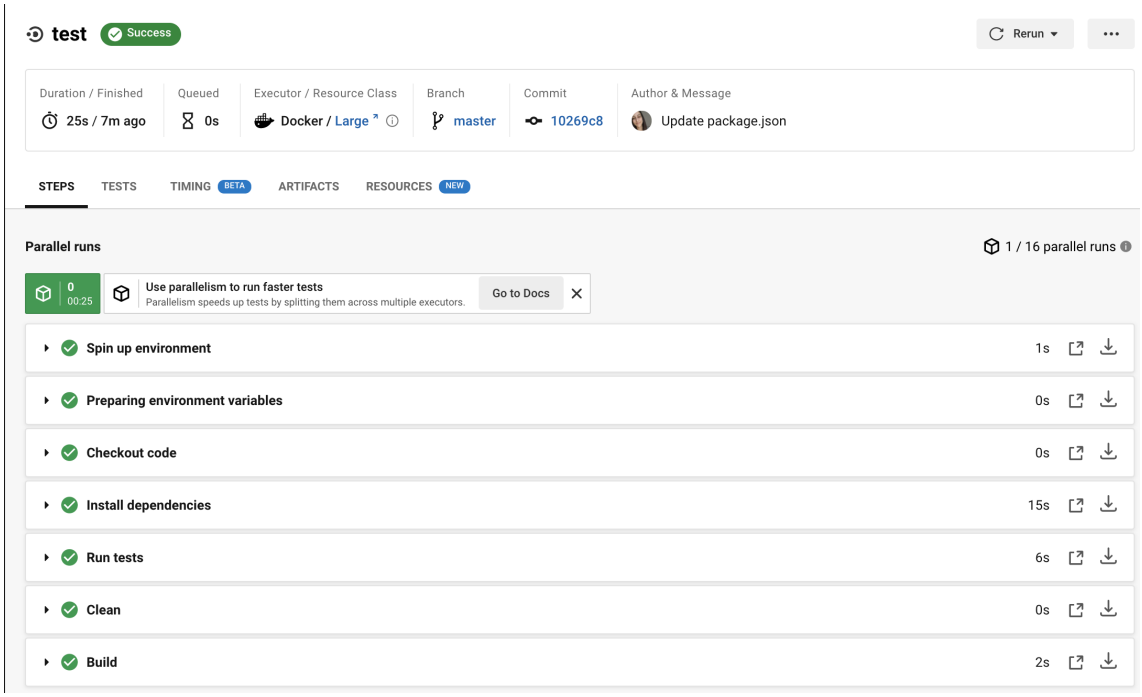

Figure 24: Job finalizado satisfactoriamente [43]

Run tests 6s

```
#!/bin/bash -eo pipefail
cd app && npm run test


> library_app@1.0.0 test
> JWT_SECRET_KEY=jwt DB_DIALECT=postgres DB_SCHEMA=memory NODE_ENV=integration-test nyc mocha --exit --requ

No New Relic Logs license key specified. Logs will not be sent.
API key does not start with "SG.".
Executing (default): CREATE TABLE IF NOT EXISTS `Organizations` (`name` VARCHAR(255) PRIMARY KEY, `APIToken


  New API Token
Executing (default): DROP TABLE IF EXISTS `Reservations`;
Executing (default): PRAGMA INDEX_LIST(`Organizations`)
Executing (default): DROP TABLE IF EXISTS `Books`;
Executing (default): PRAGMA INDEX_INFO(`sqlite_autoindex_Organizations_1`)
Executing (default): DROP TABLE IF EXISTS `Users`;
Executing (default): CREATE TABLE IF NOT EXISTS `Users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `email` VA
Executing (default): DROP TABLE IF EXISTS `Organizations`;
Executing (default): PRAGMA INDEX_LIST(`Users`)
Executing (default): CREATE TABLE IF NOT EXISTS `Organizations` (`name` VARCHAR(255) PRIMARY KEY, `APIToken
Executing (default): CREATE TABLE IF NOT EXISTS `Books` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `ISBN` VAR
Executing (default): PRAGMA INDEX_LIST(`Organizations`)
Executing (default): PRAGMA INDEX_LIST(`Books`)
Executing (default): PRAGMA INDEX_INFO(`sqlite_autoindex_Organizations_1`)
Executing (default): CREATE TABLE IF NOT EXISTS `Reservations` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `us
Executing (default): CREATE TABLE IF NOT EXISTS `Users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `email` VA
Executing (default): PRAGMA INDEX_LIST(`Reservations`)
Executing (default): PRAGMA INDEX_LIST(`Users`)
Executing (default): CREATE UNIQUE INDEX `reservations_organization_name_book_id_start_date_end_date` ON `R
Executing (default): CREATE TABLE IF NOT EXISTS `Books` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `ISBN` VAR
Executing (default): PRAGMA INDEX_LIST(`Books`)
```

Figure 25: Step ejecutado [43]

```
75    10 passing (592ms)
76
77  -----------------------------------|---------|----------|---------|---------|-------------------
78  File                               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
79  -----------------------------------|---------|----------|---------|---------|-------------------
80  All files                          |   51.72 |    30.81 |   39.06 |   52.05 |
81   src                               |      72 |      100 |   55.55 |   77.27 |
82    app.ts                           |      72 |      100 |   55.55 |   77.27 | 60,70-85
83   src/cache                         |   58.82 |       50 |      50 |   58.82 |
84    implementation.cache.ts          |     100 |      100 |     100 |     100 |
85    interface.cache.ts               |     100 |      100 |     100 |     100 |
86    local.cache.ts                   |      50 |       50 |      50 |      50 | 18-31
87   src/controllers                   |   31.48 |      100 |   14.28 |   31.48 |
88    book.controller.ts               |    6.66 |      100 |       0 |    6.66 | 6-65
89    login.controller.ts              |     100 |      100 |     100 |     100 |
90    organization.controller.ts       |   36.36 |      100 |   33.33 |   36.36 | 8-34,47
91    reservation.controller.ts        |   19.04 |      100 |       0 |   19.04 | 8-39
92    status.controller.ts             |   42.85 |      100 |       0 |   42.85 | 7-11
93    user.controller.ts               |   26.66 |      100 |       0 |   26.66 | 8-26
94   src/database                      |    87.5 |       50 |     100 |    87.5 |
95    setup.ts                         |    87.5 |       50 |     100 |    87.5 | 12
96   src/database/models               |   91.66 |        0 |   66.66 |   95.65 |
97    book.model.ts                    |      75 |        0 |       0 |   85.71 | 23
98    organization.model.ts            |     100 |      100 |     100 |     100 |
99    reservation.model.ts             |     100 |      100 |     100 |     100 |
100   user.model.ts                    |     100 |      100 |     100 |     100 |
101  src/dtos                          |     100 |       50 |     100 |     100 |
102   requestError.dto.ts              |     100 |       50 |     100 |     100 | 5-6
103  src/email                         |      60 |      100 |       0 |      60 |
104   implementation.email.ts          |     100 |      100 |     100 |     100 |
105   interface.email.ts               |     100 |      100 |     100 |     100 |
```

Figure 26: Resultado de la ejecución de tests [43]

Github Actions

Para configurar un nuevo workflow en GitHub Actions debemos crear un archivo con el mismo dentro de la carpeta ".github/workflows" con la extensión .yaml (la más utilizada dentro del ecosistema CI/CD).

En el caso previamente mencionado, el archivo sería el siguiente:

```yaml
name: NodeJS Continuous Integration

on:
  pull_request:
    branches: [ main ]

jobs:
  test_pull_request:
```

```yaml
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master
      - uses: actions/setup-node@v1
        with:
          node-version: 12
      - run: npm i
        working-directory: ./app
      - run: sudo npm test
        working-directory: ./app
      - run: npm run clean
        working-directory: ./app
      - run: npm run build
        working-directory: ./app
```

Al crear un nuevo Pull Request contra la rama “main” vemos que al final de la página nos muestra que hay un proceso corriendo, al terminar vemos lo siguiente:

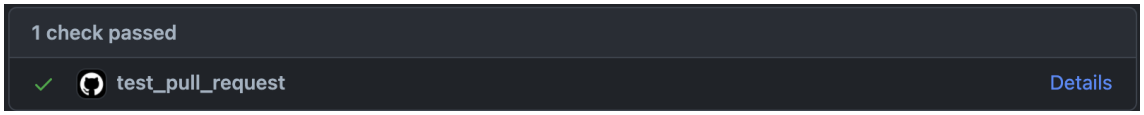

Figure 27: Workflow corrido satisfactoriamente **actions-aws**

Si entramos a los detalles podremos ver los logs dentro de cada uno de los pasos para en el caso de que falle saber donde está el error y así conseguir mayor Feedback.

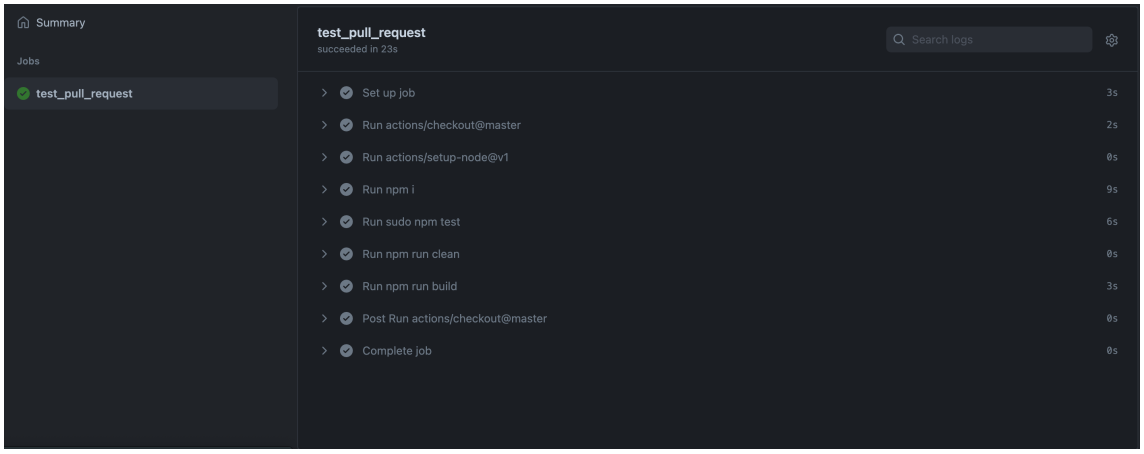

Figure 28: Detalle del workflow **actions-aws**

Prueba de concepto

El objetivo del sistema es ser una SaaS que permita crear organizaciones de bibliotecas, donde existan usuarios administradores y usuarios alumnos. Lo alumnos podrán ver el catalogo de libros, reservarlos, ver sus reservas activas y dejar reseñas, mientras que los Administradores podrán crear la organización, hacer la alta, baja y modificación de libros, y consultas relacionadas con los libros y sus reservas.

Se optó por realizar una arquitectura basada en micro servicios, cada uno de ellos se implementó con la idea de mantener una arquitectura limpia que permita una fácil mantenibilidad y modificabilidad. Para lograr estos objetivos el equipo decidió utilizar los principios y patrones de Clean Architecture.

Esta arquitectura es similar a la llamada “Cebolla”, donde las capas exteriores dependen de las inferiores, siendo las entidades el centro, ya que estas representan el negocio y por lo tanto son las que a largo plazo cambian con menor frecuencia. Esta arquitectura nos permite cumplir con métricas de diseño e impulsa el uso de los principios SOLID.

Por otro lado una arquitectura basada en micro servicios es un estilo arquitectónico para desarrollar una aplicación como una serie de pequeños servicios, cada uno

ejecutándose de forma autónoma y comunicándose entre sí. Estos cumplen determinadas características; tienen bajo acoplamiento, se lanzan de forma independiente, cada micro servicio es mantenido por un grupo pequeño de personas, son productos y pueden ser reemplazados fácilmente.

En el root de nuestra solución se encuentran varios archivos con utilidades para el desarrollo y deploy utilizando la tecnología Docker, cómo por ejemplo:

- Dockerfile: Sirve para crear una nueva imagen basada en node-alpine, es lo que se utiliza en producción, utilizamos multi-stage building para conseguir de esta forma una imagen lo más pequeña posible.
- Dockerfile.dev: Es similar al anterior mencionado, pero en lugar de estar optimizado para producción, corre un servidor que escucha los cambios en los archivos, de esta forma conseguimos la funcionalidad de Hot Reload y el feedback durante el desarrollo se acelera.
- .dockerignore: Este archivo nos indica los directorios a ignorar con el comando COPY de los Dockerfile.
- docker-compose.yml: Docker Compose sirve para evitarnos correr muchos comandos de Docker en la terminal, como sería el correr muchas imágenes, setear puertos, variables de entorno, volúmenes, redes, etc.
- Dockerrun.aws.json: Este archivo es el que usamos para subir una nueva versión a AWS Elastic Beanstalk, en nuestro caso creamos un repositorio de Docker donde subimos nuestras versiones, en este archivo especificamos la versión del repositorio queremos usar, con esto conseguimos un versionado muy granular en nuestra aplicación.

En la carpeta ".github/workflows" se encuentran las GitHub Actions usadas para implementar CI, en nuestro caso corremos los tests y buildeamos la aplicación.