UNIVERSIDAD POPULAR AUTÓNOMA DEL ESTADO DE PUEBLA

Facultad de Tecnologías de la Información y Ciencia de Datos

Pruebas Unitarias en Infraestructura como Código

Trabajo de Investigación para
obtener el Título de

Ingeniería de Software

Presenta:

Gibran Herrera López

Puebla, Pue., México Julio de **2022**

UNIVERSIDAD POPULAR AUTÓNOMA DEL ESTADO DE PUEBLA

Facultad de Tecnologías de la Información y Ciencia de Datos

Pruebas Unitarias en Infraestructura como Código

Trabajo de Investigación para obtener el Título de

Ingeniería de Software

Presenta

Gibran Herrera López

Asesorada por

Dr. Jorge Aguilar Cisneros

Puebla, Pue. México, Julio de 2022

Abstract

Las herramientas de Infraestructura como Código, mejor conocidas por sus siglas en inglés, IaC, son parte de una evolución dentro del campo de la provisión de infraestructura y configuraciones en una industria de software con una tendencia a tecnologías de la nube. Es por ello que este código se vuelve una parte integral e importante de los procesos DevOps, específicamente dentro de sus prácticas de Entrega Continua, con siglas CD en inglés. Es por ello que es importante integrar procesos y prácticas que ayuden a su implementación efectiva en el tiempo más corto posible. Una manera de lograr dicho objetivo es a través de las pruebas unitarias, las cuales han probado agilizar el proceso de aprobación de código IaC, así como: consolidar una base de conocimiento sobre cómo hacer pruebas: y finalmente, bajar la cantidad de errores. Se concluye así que la implementación de prácticas de Ingeniería de Software, las pruebas, a código IaC aportan a las prácticas DevOps.

Índice

Introducción

Este documento a tesis registra y detalla la utilización de pruebas unitarias mediante herramientas IaC. Esta infraestructura se encuentra en la nube y se regula utilizando los principios de la filosofía DevOps y de la ingeniería de software. Dichas pruebas fueron implementadas a través de una iniciativa interna en la Empresa A y que a través de esta tesis se presenta un análisis completo sobre su desarrollo y resultados.

Debido a que se requiere de discreción y que los resultados de este estudio fueran confidenciales, la empresa en donde se desarrolló el proyecto se denominará Empresa A. La cual, ofrece servicios de minería de datos e inteligencia artificial en sus especialidades de Machine Learning (aprendizaje automático) y Computer Vision (visión computacional). La empresa cuenta con 1125 empleados y presencia mundial.

El contenido de la tesis se conforma con los siguientes capítulos:

A través del capítulo 1 se aborda la teoría y conceptos que envuelven el contexto sobre el que se desarrolló el proyecto dentro del equipo de DevOps, además de que se sustenta el planteamiento de éste al denotar las áreas de oportunidad sobre las cuales sirvieron como base. Así mismo, se explica la importancia de las pruebas como piedra angular en los ciclos de desarrollo software, el tipo de pruebas que fue elegida para su implementación inicial y la forma en que aporta mediciones para una mejora futura. Finalmente, se detallará las herramientas y frameworks que se utilizaron durante el desarrollo del proyecto.

En el capítulo 2 se explica los diferentes retos, problemas y áreas de oportunidad que se encontraban durante la creación de infraestructura en la Empresa A. Se presentará una comparación del ciclo del desarrollo de software, específicamente en la etapa de pruebas, con la de proveer infraestructura en la nube a través de las herramientas IaC empleadas. También se dará una explicación sobre como el origen y capacidades de las IaC, detalladas en el capítulo 1, difieren de un lenguaje de programación, pero que, a través de herramientas de reciente creación,

amplifican sus capacidades para poder implementar pruebas y así adaptarse al ciclo de desarrollo de software.

En el tercer apartado, el capítulo 3, se declaran las metas y objetivos que sustentaron la realización del proyecto, así como la definición de métricas basadas en la filosofía DevOps sobre los ciclos de desarrollo de software. Dichos objetivos se dividieron en 2 generales y 4 específicos.

En el cuarto capítulo se explica la implementación de las pruebas elegidas en el desarrollo de infraestructura, la forma en que se usan los diferentes frameworks y herramientas IaC que facilitan su desarrollo, ejemplos de pruebas, formas de abordar los diferentes problemas y las facilidades de hacer mejoras durante su uso y mejoramiento.

En el quinto capítulo se utilizan las métricas definidas en el capítulo 3 para definir si la implementación utilizada fue la adecuada, denotar en que áreas tuvo más impacto y si hubo una mejora en el modo de crear infraestructura en la nube.

Finalmente, en los anexos se incluyen, figuras, snipets de código, gráficas, así como tablas comparativas que ayudan a la explicación de los diversos conceptos presentados.

Capítulo 1, Marco Teórico

A través de este capítulo se presentan los conceptos y conocimientos que sirvieron como base para la realización del proyecto.

1.1 Ingeniería de Software

De acuerdo con el Instituto de Ingenieros Electrónicos y Electrónica, mejor conocido por sus siglas en inglés, IEEE, define a la ingeniería de software como la aplicación sistemática, disciplinada, con enfoque cuantificable del desarrollo, operación y mantenimiento del software. (IEEE, 1990). Sin embargo, en el campo de la ingeniería de software, se ha tendido a una evolución continua, que incluso llega a sus raíces, por lo que a través de su libro Modern Software Engineering, Farley D. define a la ingeniería de software como "la aplicación de un acercamiento empírico, científico para encontrar soluciones económico-eficientes a problemas prácticos en el software" (2021). Dando así un contraste sobre las diferentes perspectivas sobre lo que debería de incluir la ingeniería de software.

1.2 Ciclo de Vida del Software

El software es también un producto comercial, por lo que cuando éste es creado, también debe de tratarse como un producto de ingeniería. Por lo que se debe de contar con un proceso para ello, el ciclo de vida del software (Gómez, S. & Moraleda, E., 2020). Entre sus fases se encuentran las detalladas por Shylesh, S. (2017)

- Planeación y análisis de requerimientos.
- Definición de requerimientos.
- Diseño de la arquitectura de software.
- Desarrollo y construcción del producto.
- Pruebas
- Despliegue
- Mantenimiento

1.2.1 Fase de Planeación y Análisis de Requerimientos

Durante esta fase sea realizan dos actividades. La primera: El análisis de requerimientos, la cual es una completa y comprensiva descripción del comportamiento del software a desarrollar. Esto implica que los analistas de sistema y negocio definan requerimientos funcionales y no funcionales. Usualmente, los requerimientos funcionales son definidos por medio de casos de uso que describen las interacciones del usuario con el software. En contraste, los requerimientos no funcionales se refieren a varios criterios, limitaciones, restricciones y requerimientos impuestos al diseño y operación del software (Bassil, Y. 2012).

Por otro lado, la segunda actividad: La planeación, se evalúan los esfuerzos para desarrollar el software de acuerdo con los requerimientos, y así, llevar a cabo una estrategia para que el plan de desarrollo sea exitoso (Prykhodko, S. et al. 2020).

1.2.2 Fase de Definición de Requerimientos

Cuando la fase de planeación y análisis de requerimientos es finalizada, entonces se hace una definición detallada de los requerimientos; estos se documentan y a su vez se verifican con el cliente. Generalmente, el resultado de esta fase es el documento de Especificación de Requerimientos de Software, o por sus siglas en inglés, SRS, el cual contiene todos los requerimientos del producto que se va a diseñar y desarrollar (Shylesh, S. 2017).

1.2.3 Fase de Diseño de la Arquitectura de Software

Es el proceso de planear y realizar propuestas para la solución de software. Esto implica que los desarrolladores, así como los arquitectos de software, definan el plan para la solución, lo cual incluye un diseño del algoritmo, un diagrama de componentes, el esquema conceptual de la base de datos, el diseño del diagrama lógico, el diseño del concepto, el diseño de la interfaz gráfica de usuario y la definición de las estructuras de datos (Bassil, Y. 2012).

1.2.4 Fase de Desarrollo y Construcción del Producto

Es la etapa donde el código es escrito y la aplicación de software es construida. Esta fase de desarrollo empieza con la configuración del ambiente de desarrollo y pruebas. Ambos ambientes deben de estar sincronizados usando el mismo protocolo (Liviu, M. 2014).

1.2.5 Fase de Pruebas

Durante esta fase, el producto en desarrollo es probado para saber si cumple los requerimientos del proyecto, los cuales fueron detallados durante la segunda fase, definición de requerimientos. Los defectos del software son reportados, rastreados, arreglados y vueltos a probar para que el producto adquiera una calidad más alta. (Shylesh, S., 2017).

Existen diferentes tipos de pruebas que se le pueden aplicar al software, dentro de las categorías más representativas se encuentran las definidas por Hooda, I. & Singh, R. (2015) como:

- Pruebas funcionales: Pruebas cuya intención es la de verificar que el funcionamiento y comportamiento del software sean los requeridos por el proyecto.
- Pruebas no funcionales: Pruebas cuya intención es la de verificar el cumplimiento de los aspectos no funcionales del software, como lo son el rendimiento, la usabilidad, y la seguridad.

1.2.5.1 Pruebas Unitarias

Dentro del campo de las pruebas funcionales, se encuentran las pruebas unitarias. Una prueba unitaria ejecuta una “unidad” de código en aislamiento y compara los resultados esperados con los obtenidos. Dicha unidad es definida por el desarrollador o equipo de desarrollo, usualmente, en lenguajes de programación orientados a objetos, dicha unidad es una clase. Las pruebas unitarias invocan uno o más métodos de una clase para producir resultados observables que pueden ser verificados automáticamente (Olan, M., 2003).

1.2.6 Fase de Despliegue

En esta etapa es donde la aplicación es instalada en el ambiente de producción. Se realizan las configuraciones relacionadas con los términos de seguridad, equipo y recursos de software; los métodos de respaldo son definidos y probados (Liviu, M. 2014).

1.2.7 Fase de Mantenimiento

Es el proceso de la modificación de una solución de software después de su entrega y despliegue para refinar sus salidas, corregir errores, y mejorar su rendimiento y calidad. Otras actividades pueden ser realizadas durante esta fase, como la de adaptar el software a nuevos ambientes, acomodar nuevos requerimientos e incrementar la fiabilidad del software (Bassil, Y. 2012).

1.3 Modelos de Desarrollo de Software

Una metodología de desarrollo de software es un grupo de reglas y lineamientos que son utilizados en el proceso de ciclo de vida del software. Dicha metodología también incluye valores clave que son respaldados por el equipo de desarrollo, así como las herramientas utilizadas en la planificación, desarrollo e implementación del proceso (Liviu, M., 2014).

Dentro de estas metodologías se encuentran las más relevantes definidas por Sabbir M., et al. (2017):

- Modelo de Cascada
- Modelo iterativo e incremental
- Modelo de Espiral
- Modelo V
- Modelos de Procesos Ágiles

Sin embargo, debido al éxito del modelo Ágil, por su aproximación a una solución orientada al cliente, la cual mejora la eficiencia del desarrollo de software, nacen otros modelos a partir de esta, con el objetivo de aún mejorar más dicho modelo. Dentro de ellos se encuentra el modelo de:

- Desarrollo de software lean (Dash, C. 2021).

1.3.1 Metodología del Modelo de Cascada

La metodología del modelo de cascada es proceso de desarrollo de software secuencian en donde dicho proceso se considera que fluye incrementalmente hacia abajo, similarmente a una cascada, a través de una lista de fases que deben ser ejecutadas en orden para construir el software de computadora de manera exitosa (Bassil, Y. 2012)

1.3.2 Metodología del Modelo iterativo e incremental

La metodología iterativa e incremental, de acuerdo con Liviu, M. (2014) depende de la construcción de la aplicación de software, un paso a la vez en forma de un modelo expansivo. Dicho modelo no es descartado en futuras fases, sino que su propósito es el ser extendido. Después de que dicho modelo es probado, y la retroalimentación es recibida por el cliente, las especificaciones son ajustadas y nuevamente, el modelo es extendido. Dicho proceso es repetido hasta que el modelo se vuelve en una aplicación totalmente funcional y todos sus requerimientos son cumplidos.

1.3.3 Metodología del Modelo de Espiral

Este modelo es una combinación del modelo iterativo y de cascada debido a que combina la idea de iterar, pero con un proceso sistemático y controlado. Dentro de sus ventajas se encuentra su capacidad de aceptar cambios, el uso de prototipos es permitido, la recolección de requerimientos es más precisa, los usuarios pueden observar el sistema de manera temprana y el desarrollo, al ser dividido en diversas partes, tienen menor riesgo. Pero, por otra parte, sus desventajas contemplan una administración compleja, un proceso complejo, y una documentación pesada (Shylesh, S. 2017).

1.3.4 Metodología del Modelo V

Similar al modelo de cascada, el modelo V es un camino secuencial a la ejecución de procesos. Cada fase debe de ser completada para poder iniciar la siguiente fase. Sin embargo, la fase de

pruebas es enfatizada debido a que los procedimientos para las pruebas son desarrollados tempranamente, antes de incluso iniciar la codificación (Munassar, N. & Govordhan, A. 2010).

1.3.5 Metodologías Ágiles de Desarrollo de Software

De acuerdo con Williams L., las metodologías ágiles son un sub-set de los métodos iterativos y evolutivos, y están basados en la mejora iterativa y oportunista de los procesos de desarrollo. Cada iteración de la metodología ágil es un mini proyecto contenido en sí mismo, con actividades que incluyen el análisis de requerimientos, diseño, implementación, pruebas, y aceptación del cliente (2010).

1.3.6 Desarrollo de Software Lean

El desarrollo de software lean, tiene sus raíces en la manufactura lean de los años 1940, introducida por el sistema de producción de Toyota, la cual ha sido históricamente enfocada en la reducción de costos, eliminación del desperdicio y el hacer más con menos (Ohno, 1988). Sin embargo, fue Poppendieck, M. (2012) quien publicó en su libro, los fundamentos aplicados de la manufactura lean con los fundamentos de las metodologías ágiles de desarrollo de software, dando a la creación del desarrollo de software lean con base en los siguientes principios:

1. Eliminación del desperdicio
2. Amplificación del aprendizaje
3. Decidir lo más tarde posible
4. Entregar los más rápido posible
5. Empoderar al equipo
6. Construir integridad
7. Ver el todo.

1.3.6.1 Kanban

Dentro del desarrollo de software lean, se encuentra la metodología de desarrollo Kanban, la cual es descrita por Anderson, D. (2010) como “un método revolucionario que utiliza el sistema de

atracción, visualización y otras herramientas, para catalizar la introducción de ideas Lean". Es también Anderson, D. (2010) quien define los cinco principios de Kanban:

1. Visualización del flujo de trabajo: El trabajo se mueve a través de los diferentes estados (Planeado, En progreso, Hecho) al mismo tiempo en que se mueve dentro de la organización, por lo que este debe de ser visualizado en todo momento.
2. Limitar el trabajo en progreso: Deben de existir límites del trabajo en progreso para manejar la cantidad de este en cualquier fase del flujo de trabajo.
3. Medir y manejar el flujo: Utilizar herramientas como el tablero Kanban, diagramas de flujo, mapas de corriente, así como de cuatro métricas clave, el tamaño de la cola, ratio de salida, tiempo de los ciclos, y tiempo de espera.
4. Flujo: Consiste en identificar a las personas, procesos, y cultura como parte de uno mismo.
5. Hacer los procesos explícitos: Así como el trabajo se mueve a través de sus diferentes estados dentro del tablero Kanban, se deben de declarar políticas explícitas, también conocidas como criterios de entrada y salida, para determinar cuándo un objeto de trabajo puede moverse de un estado a otro.

1.4 DevOps

Construido en prácticas lean y ágiles (Ebert, C. et al. 2016) el término DevOps de acuerdo con Hüttermann, M. (2012) describe prácticas que agilizan el proceso de entrega de software, enfatizando el aprendizaje a través de la transmisión de retroalimentación desde producción al desarrollo, mejorando el tiempo del ciclo de vida del software. A través de sus cuatro principios:

1. Cultura: Personas sobre procesos. El software está hecho para y por personas.
2. Automatización: La automatización de los procesos es esencial para ganar retroalimentación rápidamente.
3. Medición: La calidad, así como de incentivos compartidos, o al menos alineados, son críticos.
4. Compartir: Crear una cultura donde las personas comparten ideas, procesos, y herramientas.

1.5 Go

Go, también conocido como Golang, es un lenguaje de programación de código abierto, compilado, de tipado estático diseñado por Google (Kolade, C. 2021) y que de acuerdo con su documentación define que Go es expresivo, conciso, limpio y eficiente, además de que su mecanismo de concurrencia hace sencillo el escribir programas que sacan provecho de máquinas multi núcleo y conectadas a través de la red (Google, n.d).

1.6 Computación en la nube

De acuerdo con Dan C. (2022), la computación en la nube es una tecnología cuyos usuarios pueden acceder a una red de servidores remotos que acogen servicios en Internet de almacenamiento, manejos y procesamiento de datos. Existen diversos tipos de computación en la nube, los cuales Dan C. (2022) define en:

- Públicas: La infraestructura es disponible para el público en general o a un gran grupo de la industria y es propiedad de una compañía que vende sus servicios.
- Privada: La infraestructura es operada solamente para la organización.
- Híbrida: La infraestructura es una composición de dos o más tipos de nube.
- Comunidad: La infraestructura es compartida por diversas organizaciones y solo da soporte a las inquietudes específicas de sus miembros

1.7 AWS

Amazon Web Services, o mejor conocido por sus siglas en inglés, AWS, es una plataforma de servicios web que ofrece soluciones de computación, almacenamiento, redes, en diferentes capas de abstracción (Witting, M. & Witting, A., 2015). De acuerdo con Richter F., en el mercado de los servicios de la computación en la nube, AWS lidera el mercado en el último cuarto del 2021 con un 33% (2022). Dando de esta manera uno de los servicios más valiosos y utilizados en el mundo.

1.8 CI/CD

El acrónimo de CI/CD se compone de dos significados; el CI se refiere a integración continua, el cual es un proceso de automatización que ayuda a la incorporación de nuevo código a la aplicación. Mientras que, por otra parte, CD, se refiere a la entrega continua, que también es un proceso de automatización, pero de despliegue de los cambios realizados en el código (Red Hat, 2022)

1.9 GitHub Actions

GitHub Actions es una plataforma de integración y entrega continuas, o mejor conocida como CI/CD, por sus siglas en inglés, que permite automatizar la construcción, las pruebas y el despliegue de una aplicación. La plataforma permite la creación de flujos de trabajo para construir o hacer pruebas al repositorio (GitHub, n.d.). Algunos de los conceptos que se utilizan dentro de la plataforma son:

- Workflow: Es un proceso automatizado configurable que puede correr uno o más jobs.
- Event: Una actividad específica dentro del repositorio que dispara la ejecución de un flujo de trabajo.
- Job: Es una serie de pasos dentro de un workflow que se ejecutan sobre la misma plataforma. Cada paso puede ser un script o una acción que debe de ejecutarse.
- Action: Es una aplicación propia de GitHub Actions que realiza complejas pero frecuentes tareas. Algunos ejemplos de estas acciones son el proceso de autenticación con un proveedor de servicios en la nube como AWS. (GitHub, n.d.)

1.10 IaC

Infraestructura como Código, IaC, es el manejo de infraestructura (redes, máquinas virtuales, balanceadores de carga, topologías de conexión) en un modelo descriptivo, usando el mismo sistema de versiones que el que se utiliza para código fuente. (Microsoft, 2021)

De acuerdo con Brikman, Y. (2022) existen cinco tipos de herramientas IaC:

- Ad hoc scripts: La herramienta más sencilla y básica para la automatización de tareas, el script, este puede ser escrito en diferentes lenguajes de scripting, como lo son Bash, Python, Ruby.
- Herramientas de administración de configuración: Diseñadas para instalar y manejar software dentro de servidores. Algunas de estas herramientas son Chef, Puppet, Ansible y SaltStack.
- Herramientas de plantillas de servidor: Estas herramientas son alternativas a las de la administración de configuración porque en vez de crear diversos servidores y configurarlos a través de correr el mismo código en cada uno, crean una imagen de un servidor para capturar el sistema operativo, software, archivos, y configuración en un solo contenedor. Algunos ejemplos de estas herramientas son Docker, Packer y Vagrant.
- Herramientas de orquestación: Manejan las tareas de desplegar, realizar actualizaciones, monitoreo, escalabilidad, distribución de tráfico y de comunicación con los diferentes recursos. Algunas de estas herramientas son Kubernetes, Docker Swarm y AWS ECS
- Herramientas de provisión: Responsables de la creación de infraestructura. Algunas de ellas son Terraform, CloudFormation y OpenStack Heat.

1.11 Terraform

Terraform es una herramienta IaC que permite la creación, cambio, y versionamiento de infraestructura de manera segura y eficiente a través de archivos de configuración que son sencillos de leer (HashiCorp, n.d. a). La herramienta está construida en una arquitectura basada en plugin que permite a los desarrolladores extenderla a través de nuevos plugin. Terraform está dividida en dos partes principales:

1. Terraform Core: Es un binario estáticamente compilado escrito en el lenguaje de programación Go que sirve como punto de entrada.
2. Terraform Plugin: Binarios ejecutables en Go para invocar el Core. Cada plugin expone la implementación de un servicio o proveedor, como por ejemplo AWS. (HashiCorp, n.d. b)

1.12 Terragrunt

Terragrunt es un envoltorio ligero que provee herramientas adicionales para reducir la repetición dentro de las configuraciones, la facilitación del uso de múltiples módulos y un estado remoto de la configuración. (Gruntwork.io, n.d. a)

1.13 Terratest

Terratest es una librería de Go que provee patrones y funciones de ayuda para realizar pruebas a la infraestructura con soporte para herramientas IaC como Terraform, Packer, Docker, Kubernetes, AWS, y GCP. (Gruntwork.io, n.d. b)

Capítulo 2, Contexto

2.1 Definición del problema

Dentro del mundo de los servicios y aplicaciones de producción, que están disponibles a una gran cantidad de clientes, existen muchos temores: temor a una caída, temor a la pérdida de datos, temor a las brechas de seguridad, temor a que el servicio se comporte de la misma manera en todos sus ambientes. Y en caso de algún problema, ¿será posible recuperarse de manera rápida y eficaz?

Una práctica que es integral a la entrega continua, CD, es la utilización de Infraestructura como Código (IaC), la cual se vuelve esencial en la implementación de pipelines de entrega. El uso de esta herramienta ha resultado en beneficios para las diferentes organizaciones, por ejemplo, la NASA redujo su proceso de parcheo de múltiples días a solo 45 minutos. Pero a pesar de los beneficios, las IaC son susceptibles a defectos, los cuales pueden causar serias consecuencias. Por ejemplo, un defecto encontrado en los repositorios de IaC en AWS causó interrupciones con un valor de 150 millones de dólares (Hasan, M. et al., 2020).

2.2 Planteamiento del problema

Si se administra la infraestructura como código, entonces debe de haber una manera para mitigar riesgos: las pruebas. El objetivo de hacer pruebas es de dar confianza al hacer cambios (Brikman, Y. 2022).

Además, Hasan M. et al. establece que existe una prevalencia de defectos en IaC, lo cual causa la necesidad de realizar pruebas al código producido por estas herramientas. Incluso detalla que, al no conocer qué prácticas implementar para adoptar el uso de pruebas de manera efectiva, se debe de recurrir a la identificación de estrategias implementadas en la industria, así como de los conocimientos de usuarios de las IaC, para poder así, consolidar una base de conocimiento que pueda derivar a nuevos caminos de investigación (2020).

2.3 Objetivo

"Aportar al aseguramiento de la calidad al usar herramientas IaC mediante la utilización de pruebas unitarias para acortar los ciclos de desarrollo de software."

Objetivos particulares:

- Integrar los nuevos procesos de pruebas a los pipelines de CI/CD a través de jobs en GitHub actions para reducir la interacción manual de los Ingenieros DevOps, mostrar los resultados de sus cambios en las pull request y ayudar a su proceso de aprobación
- Recopilar y consolidar una base de conocimiento sobre como proveer infraestructura y servicios de confianza a través de casos de pruebas unitarias en las que el equipo de Ingenieros DevOps pueda modificar, añadir y eliminar.

2.4 Justificación

Uno de los objetivos que tiene la utilización del modelo DevOps, es el acortar los ciclos de desarrollo para así obtener retroalimentación temprana. Dicho objetivo debería de ser aplicado a las herramientas IaC, las cuales, si bien no son estrictamente lenguajes de programación, deberían ser consideradas y tratadas como generadoras de código, puesto que su producto se almacena en repositorios, siguen reglas sintácticas, contienen módulos lógicos y ayudan a la solución de problemas complejos. En el caso específico de Terraform, la herramienta que utiliza la Empresa A para aprovisionar su infraestructura y servicios, es capaz de crear módulos, los cuales se pueden asemejar a los objetos de los lenguajes de programación.

El manejo previo de esta herramienta no seguía esta filosofía. El proceso de provisión seguía el siguiente flujo (Ver Figura 1):

- El ingeniero DevOps recibe un ticket, equivalente a un "user story" dentro de la metodología Kanban, la cual es la que se usa actualmente.
- El DevOps tiene la flexibilidad de dividir el ticket en diversas tareas que faciliten la administración y el estado de dicho elemento de trabajo.
- El DevOps entonces pasa las tareas al estado de "en progreso".
- Cuando esta unidad de trabajo es finalizada, pasa a un estado de "revisión de pull request".

- Cuando la unidad de trabajo es aprobada, ésta puede pasar a dos estados, listo para desplegar, o hecho. El primer caso es usado solamente cuando existe una dependencia a otro elemento de trabajo para poder ser implementado, o que su implementación implica algún inconveniente para el trabajo de otros sectores, como un periodo de caída de algún servicio, y este es generalmente programado para una ventana de mantenimiento.

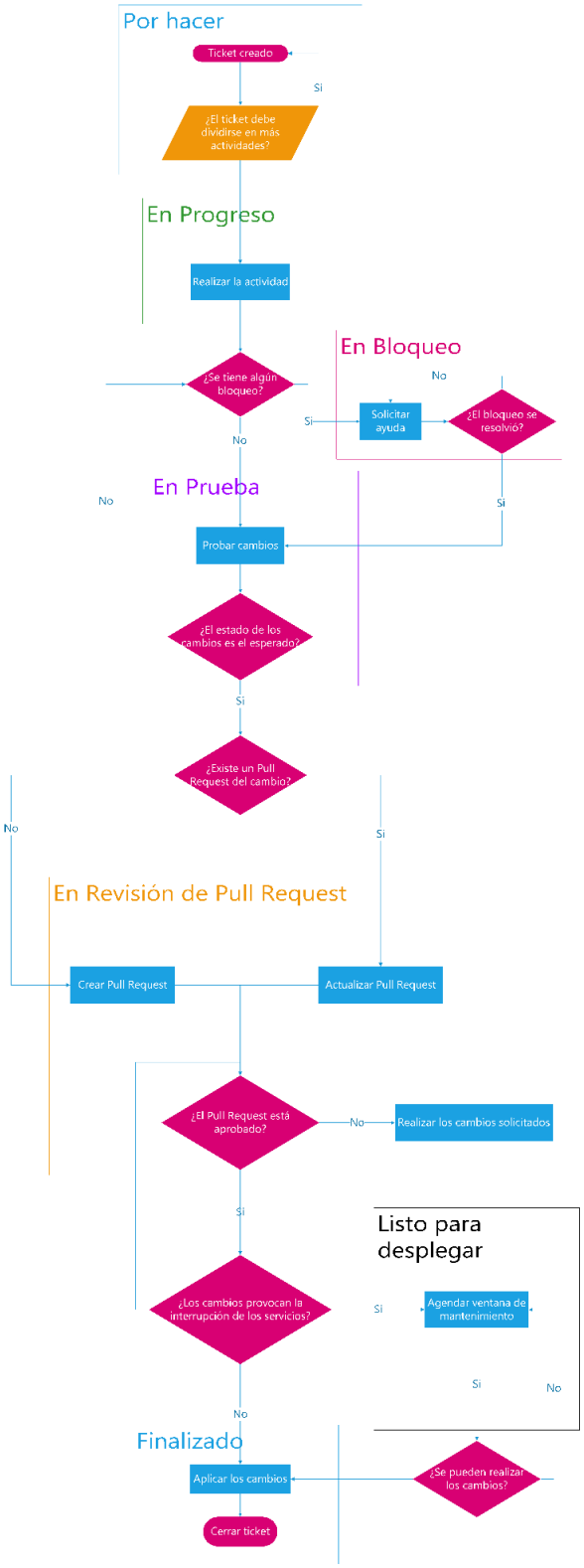

Figura 1: Diagrama del proceso de provisión de infraestructura en el equipo de DevOps en la Empresa A

Dentro de este proceso existe un gran inconveniente, el cual retrasaba el paso de las unidades de trabajo al siguiente estado, y ese es el estado de revisión de pull request.

Un pull request, como su nombre lo indica, es una petición para que un commit o un conjunto de ellos, sean colocados dentro de la rama principal del repositorio, la cual se ocupa como un reflejo actual de toda la infraestructura de la empresa A. Este es un proceso manual, ya que involucra al DevOps en la creación de dicho pull request a través de la interfaz gráfica o línea de comandos, dependiendo de su preferencia, incluyendo el enlace a la unidad del trabajo a través de nuestro tablero digital Kanban, y colocar un breve resumen de los cambios que se van a implementar, utilizando el resultado que arroja la herramienta Terragrunt al usar `terragrunt plan`. Esto se hace con la finalidad visualizar lo que va a suceder en la infraestructura, y saber si hubo cambios en algún módulo, además, esto añade el contexto de dicho cambio.

Sin embargo, es tarea del DevOps, constatar, a través de sus conocimientos, que el cambio no contiene errores, antes de enviar sus cambios al proceso de revisión. Como, previamente, se realiza este proceso de revisión, es bastante independiente del DevOps, puede ser que se realice algunas pruebas, dependiendo del cambio, como verificar que dicha infraestructura exista, o si la nueva regla del balanceador de carga se cumpla, o que los contenedores se lancen y envíen sus chequeos de salud de manera exitosa, etcétera.

Es por esto que, la independencia de dicho proceso hace que no se tenga una garantía concisa de que funcione, que se pruebe de la manera adecuada, o que el método sea el más rápido y eficiente. Además, este conocimiento no es compartido con el resto de los DevOps, causando una brecha de conocimiento en todo el equipo.

Por otro lado, la revisión de pull request, también está sujeta a errores humanos, debido a que también las revisiones son basadas en los conocimientos de otros ingenieros, por lo que se es propenso a pasar por alto algunos errores. Es por ello que a pesar de que dicho cambio sea aprobado por los demás, no se tienen garantías de que siga al pie de la letra los lineamientos de formato del código o que este funcione correctamente.

Otra de las desventajas de la forma en cómo se aprueban dichos pull request es el tiempo en el que se lleva en su aprobación. Por ejemplo, en una muestra de 158 pull request que se llevaron a cabo dentro del repositorio donde se maneja la infraestructura, se notó que, en promedio, para que dicho pull request sea cerrado, ya sea aprobado o no, toma unos 1.64 días y 20 pull requests contenían errores, haciendo que los ciclos de desarrollo de infraestructura sean largos de manera innecesaria.

Una forma que se pensó para solucionar dicho problema fue a través de la implementación de pruebas usando la librería Terratest, la cual, viene adecuada con la herramienta que se utiliza para proveer la infraestructura, Terragrunt. Terratest realiza pruebas unitarias al código IaC, y es totalmente compatible con diversas características de Terragrunt como los estados, local y remoto, dependencias, y uso de diversos módulos a la vez.

2.5 Hipótesis

- El tiempo de aprobación de pull request se reduce en un 50% con la utilización de pruebas unitarias con Terratest
- El ratio de pull request con errores pasa de un 12.6% a un 6%

2.6 Preguntas de Investigación

- ¿Cuál es la diferencia en los tiempos de aprobación entre pull requests con y sin pruebas unitarias?
- ¿Cuál es la diferencia de los ratios de error entre pull requests con y sin pruebas unitarias?

2.7 Metodología

Para recolectar la información necesaria en la validación de los objetivos, se obtendrá el tiempo de aprobación de un pull request de acuerdo con el momento en el que este sea publicado como listo para su revisión, hasta el momento en el que este sea aprobado por otras dos personas del equipo DevOps o de Desarrollo. Para la extracción de estos datos se utilizará la API de GitHub, y el tiempo será medido en minutos.

Por otro lado, para medir la cantidad de pull request que tenían errores, se utilizará la propiedad de "revert" en GitHub, así como los pull request remediales, y en caso de que algún problema persista en el pull request de solución, este será contado como otro pull request con errores.

Se utilizarán como muestra 158 pull requests para dos poblaciones distintas, pull requests que estén implementando pruebas unitarias y pull requests que no utilizaron pruebas unitarias. Ambas poblaciones deben de seguir los lineamientos de publicación y formato.

Dentro del contexto de cada una de las pull request, estas fueron sacadas de un repositorio en donde se maneja la infraestructura de la Empresa A en GitHub utilizando como IaC a Terraform-Terragrunt. El equipo DevOps que desarrolla este repositorio está constituido por 14 Ingenieros DevOps de los cuales 10 se encuentran en el nivel Senior y 4 Juniors. Todos ellos tienen conocimiento y manejo del lenguaje de programación Go.

Las fórmulas que se utilizarán para obtener el tiempo de aprobación y el ratio de error serán las siguientes:

$$Tiempo\ de\ aprobación = \ Momento\ de\ aprobación - Momento\ de\ disposición\ de\ revisón$$

$$Ratio\ de\ error\ por\ pull\ request = \frac{PRA + PRE}{PRE}$$

$$PRE = Pull\ Request\ con\ errores$$

$$PRA = Pull\ Request\ sin\ errores$$

2.9 Recolección y análisis de datos

La extracción de datos será posible a través del uso de la API de GitHub, específicamente, utilizando la librería de Octakit. Dicha librería es utilizada por una aplicación de JavaScript, utilizando el run time de Deno para su ejecución, la cual también estará a cargo del procesamiento para obtener los tiempos de aprobación y el ratio de error de ambas poblaciones para definir si se

cumplieron con los objetivos del proyecto. Este código utilizado para obtener los datos es representado en el siguiente diagrama de flujo (Ver Figura 2)

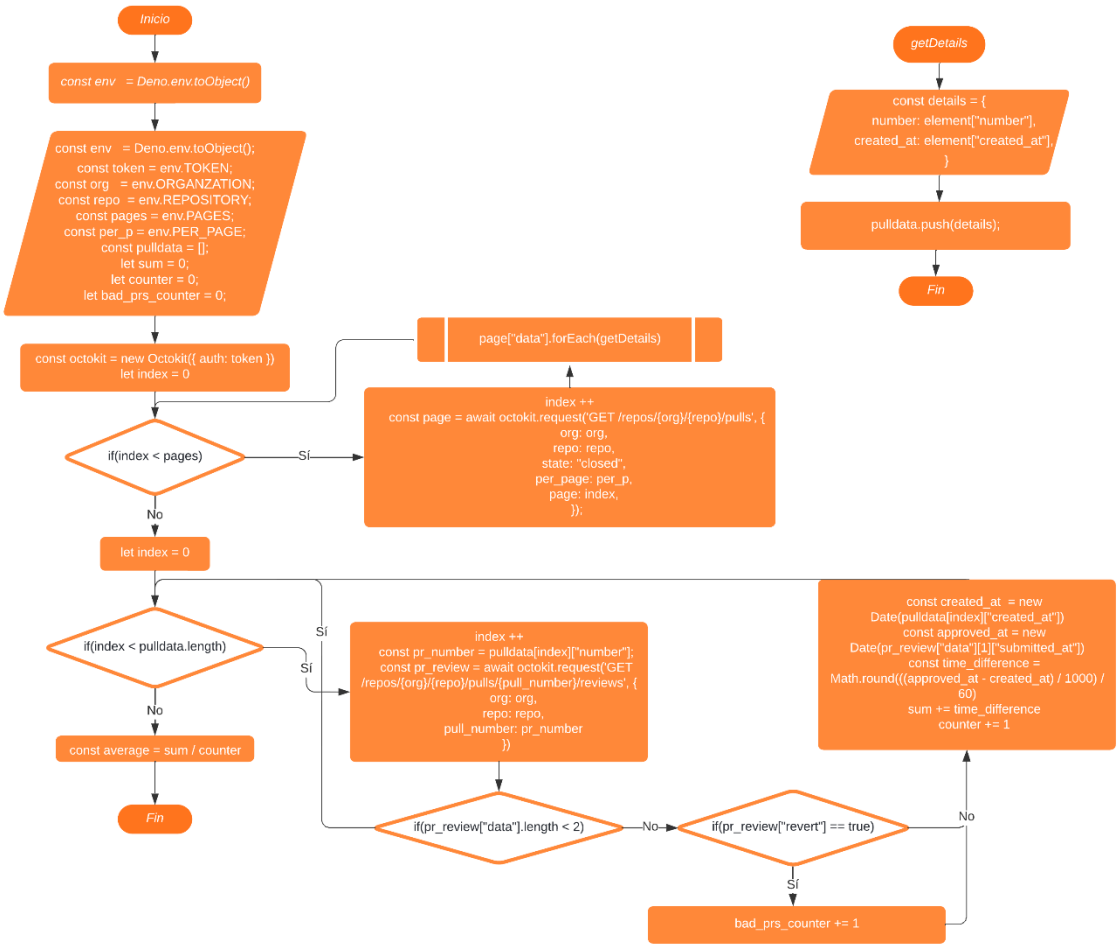

Figura 2: Diagrama de flujo para el código utilizado en la recolección del tiempo de aprobación y ratio de error.

Mientras que el código utilizado para obtener estos datos fue el siguiente (Ver Figura 3)

```
// Importa la librería para el uso de la API de GitHub
import { Octokit } from "https://cdn.skypack.dev/octokit?dts";

// Obtiene los valores de las variables de ambiente
const env   = Deno.env.toObject();
const token = env.TOKEN;
const org   = env.ORGANZATION;
const repo  = env.REPOSITORY;
```

```
const pages = env.PAGES;
const per_p = env.PER_PAGE;

// Inicializa el cliente con la API de GitHub
const octokit = new Octokit({ auth: token });

// Realiza la limpieza de los datos
const pulldata = [];
function getDetails(element) {
   const details = {
       number: element["number"],
       created_at: element["created_at"],
   }
   pulldata.push(details);
}

// Obtiene las pull requests de una sola población con o sin pruebas.
for (let index = 0; index < pages; index ++) {
   const page = await octokit.request('GET /repos/{org}/{repo}/pulls', {
       org: org,
       repo: repo,
       state: "closed",
       per_page: per_p,
       page: index,
   });

   page["data"].forEach(getDetails);
}

// Contadores para sacar el promedio de tiempo y el ratio de error.
let sum             = 0;
let counter         = 0;
let bad_prs_counter = 0;

// Obtiene los detalles de tiempo para cada una las pull requests.
for (let index = 0; index < pulldata.length; index ++) {
```

```
    // Obtiene la información de cada pull request.
    const pr_number = pulldata[index]["number"];
    const pr_review = await octokit.request('GET
/repos/{org}/{repo}/pulls/{pull_number}/reviews', {
        org: org,
        repo: repo,
        pull_number: pr_number
    })

    // Ignora las pull requests que no fueron aprobadas y cerradas
    if (pr_review["data"].length < 2) {
        continue;
    }

    // Obtiene las pull requests con errores
    if (pr_review["revert"] == true) {
        bad_prs_counter += 1;
    }

    // Obtiene los tiempos de la segunda aprobación y creación
    const created_at  = new Date(pulldata[index]["created_at"])
    const approved_at = new Date(pr_review["data"][1]["submitted_at"])

    // Obtiene el tiempo de aprobacion
    const time_difference = Math.round(((approved_at - created_at) / 1000) /
60)

    // Actualiza los contadores
    sum += time_difference
    counter += 1
}

// Obtiene el promedio del tiempo de aprobación
const average = sum / counter

// Imprime los resultados
```

```
console.log("Avarege is: " + average)
console.log("\nBad PRs: " + bad_prs_counter)
```

Figura 3: Código utilizado para la recolección del tiempo de aprobación y ratio de error.

Capítulo 3, Desarrollo

3.1 Implementación

Dentro de las herramientas que se utilizan para proveer infraestructura, Terraform fue por la que se decantó hacer la primera fase de pruebas unitarias. Esta herramienta es utilizada exclusivamente en un solo repositorio. Sin embargo, también se hace uso de otra herramienta, Terragrunt, la cual funciona como envoltura para proveer características extras que ayudan al manejo de un estado remoto y configuraciones simplificadas.

En el caso de un estado remoto, este nos ayuda a tener una copia de seguridad sobre el estado de la infraestructura y a mantener bloqueos. Estos bloqueos son utilizados cuando una persona quiere realizar un cambio y prevé que otra persona, en tiempos similares, quiera hacer cambios que puedan perjudicar a ambos.

Por otro lado, Terragrunt ayuda a mantener configuraciones simplificadas con la habilidad de crear referencias a módulos de Terraform. Un módulo de Terraform es un conjunto de configuraciones y recursos.

Para ejemplificar el módulo, debemos de partir con la forma en que se provee recursos. Si queremos crear una máquina virtual, usando el servicio de EC2 de AWS, el código de Terraform sería el siguiente (Ver Figura 4):

```
resource "aws_instance" "ec2_sample" {
  instance_type = "t3.micro"

  tags = {
    Name = "Ejemplo EC2"
  }
}
```

Figura 4: Ejemplo de provisión de una máquina virtual EC2 a través de Terraform.

Pero si necesitamos que esta máquina virtual se conecte a una nube virtual privada en concreto, necesitamos utilizar diversos recursos: una VPC, una subnet, y una interfaz de red (Ver Figura 5).

```
resource "aws_vpc" "vpc" {
  cidr_block = "10.16.0.0/16"
}

resource "aws_subnet" "subnet" {
  vpc_id            = aws_vpc.vpc.id
  cidr_block        = "10.16.10.0/24"
  availability_zone = "us-west-2a"
}

resource "aws_network_interface" "net_interface" {
  subnet_id   = aws_subnet.subnet.id
  private_ips = ["10.16.10.100"]
}

resource "aws_instance" "ec2_sample" {
  instance_type = "t2.micro"

  network_interface {
    network_interface_id = aws_network_interface.net_interface.id
    device_index         = 0
  }
}
```

Figura 5: Ejemplo de infraestructura simple para una máquina virtual EC2 a través de Terraform.

Como se puede observar, estas configuraciones pueden incrementar sus líneas de código rápidamente, así como su complejidad; y si estas son utilizadas con bastante frecuencia, pueden llegar a formar una gran cantidad de éste. Es por ello que los módulos condensan estas configuraciones y utilizan variables para aquellos elementos que puedan variar. Para crear un

módulo en Terraform solo es necesario colocar todos los archivos con la extensión ".**tf**" en un directorio, de esta manera (Ver Figura 6):

```
.
├── LICENSE
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
```

Figura 6: Ejemplo de la composición de un módulo de Terraform dentro de un directorio.

Estos módulos pueden utilizar otros módulos y crear configuraciones aún más complejas. Pero, al tener esta complejidad, se vuelve complicado su manejo, y es por ello que se utiliza Terragrunt. Una manera en la que ayuda es en la utilización de dependencias entre módulos como VPCs, bases de datos, etc. y variables comunes compartidas como la zona en donde debe de desplegarse la infraestructura.

Actualmente, la Empresa A, hace solo uso de módulos para proveer infraestructura, por lo que la definición de unidad para las pruebas será el módulo implementado en un archivo de Terragrunt. Un ejemplo de ello es el siguiente (Ver Figura 7):

```
terraform {
  # Define donde se encuentra el módulo
  source = "../../..//modules/ec2"

  # Agrega las variables comunes
  extra_arguments "custom_vars" {
    commands = get_terraform_commands_that_need_vars()
    required_var_files = [
      "${get_terragrunt_dir()}/../common/tfvars"
    ]
  }
}

dependency "vpc" {
  config_path = "../../vpc/sample"
```

```
}

# Indica los valores den entrada para cada una de las variables dentro
del módulo.
inputs = {
  ec2_name          = "Sample"
  ec2_instance_type = "t3.micro"

  cidr_block           = dependency.vpc.outputs.cidr_block
  network_interface_id = dependency.vpc.outputs.net_interface_id
}
```

Figura 7: Ejemplo de infraestructura simple para una máquina virtual EC2 a través de Terragrunt usando módulos.

Las pruebas unitarias fueron hechas a través de la librería Terratest, la cual está disponible en el lenguaje Go. Lo que hace esta librería es realizar los cambios, checar si existe algún error a la hora de proveerlos, realizar pruebas sobre lo cambiado, y restaurar la infraestructura al estado en el que estaba anteriormente. Dando finalmente un reporte de lo sucedido.

Un ejemplo de la utilización de esta librería es el siguiente (Ver Figura 8):

```
package test

import (
    "testing"
    "log"
    "net/http"

    "github.com/gruntwork-io/terratest/modules/terraform"
    "github.com/stretchr/testify/assert"
)

func TestEc2Example(t *testing.T) {
```

```
    // Construye la infraestructura definida en el módulo y maneja los
errores
    // comunes como time out. Utiliza los valores por default.
    terraformOptions := terraform.WithDefaultRetryableErrors(t,
&terraform.Options{
        // Define la locación del módulo y las opciones implementadas
        TerraformDir:
"../environments/stage/storage_service/example_project",
        TerraformBinary : "terragrunt",
    }

    // Limpia los recursos al terminar las pruebas.
    defer terraform.Destroy(t, terraformOptions)

    // Instala los proveedores y aplica la infraestructura. Falla si
hay algún error en
    // este proceso.
    terraform.InitAndApply(t, terraformOptions)

    // Obtiene los valores de salida, la url del servicio en este
caso.
    health_url := terraform.Output(t, terraformOptions, "health_url")

    // Obtiene el código HTTP de la url de salud del servicio.
    resp, err := http.Get(health_url)
    if err != nil {
        log.Fatal(err)
    }

    // Compara si el código HTTP es 200
    assert.Equal(t, 200, resp.StatusCode)
```

```
}
```

Figura 8: Ejemplo de prueba unitaria codificada en Go utilizando la librería Terratest.

Finalmente, estas pruebas se integran dentro de un Job de GitHub Actions que se encarga de ejecutar el código de las pruebas en el caso específico del módulo, así como de verificar que su formato a través de la herramienta Terragrunt HCML, el cual muestra si el archivo debe de modificarse o no. Los resultados de ambos pasos son mostrados directamente en el pull request de GitHub y bloquea que dicha pull request sea integrada a la rama principal a menos que sean positivos.

3.2 Resultados

Se implementaron pruebas unitarias en la mayoría de los módulos, priorizando aquellos que son utilizados de manera frecuente, y los casos de prueba fueron actualizados conforme al consenso y conocimiento de todos los integrantes del equipo DevOps.
Los resultados obtenidos de la implementación fueron los siguientes (Ver Figura 9):

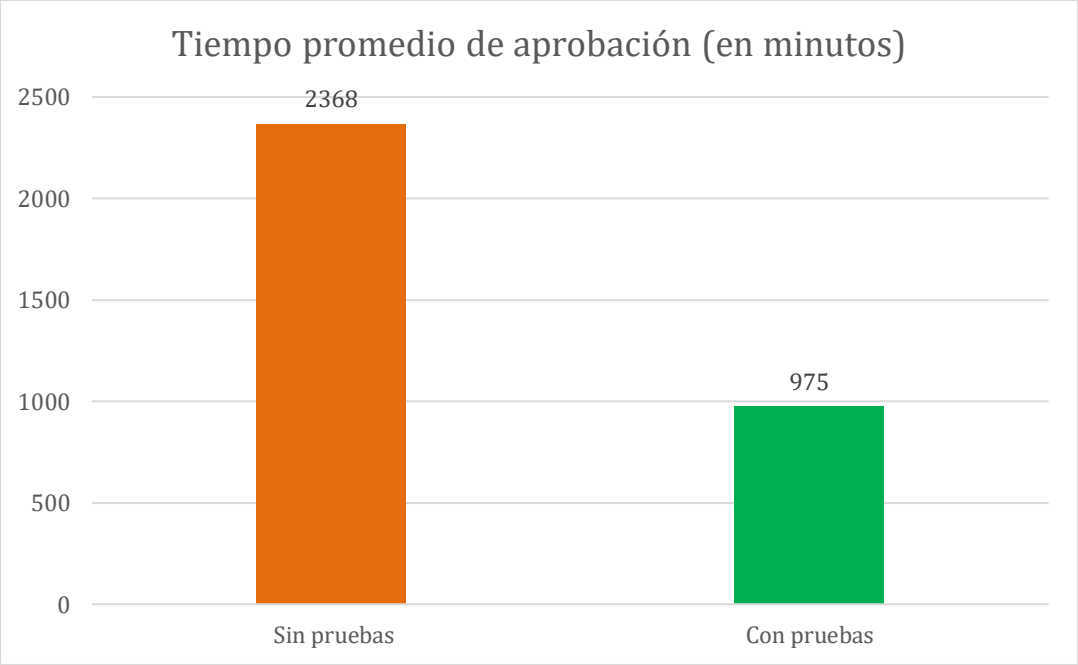

Figura 9: Gráfica comparativa del tiempo promedio de aprobación de pull request con y sin pruebas unitarias en minutos.

En el tiempo promedio de aprobación se obtuvo una mejora del 58% sobre el tiempo promedio que se tenía antes de implementar pruebas unitarias, lo cual traducía a ciclos de desarrollo más cortos.

En el caso de la cantidad de errores encontrados en estos pull request, se encontró una gran diferencia de puntos porcentuales, se pasó de un ratio de 12.66 a uno de 1.9. Esto podría atribuirse a que, en los casos de fallo, se implementaban nuevos casos de prueba que ayudaban a no realizar el mismo error (Ver Figura 10).

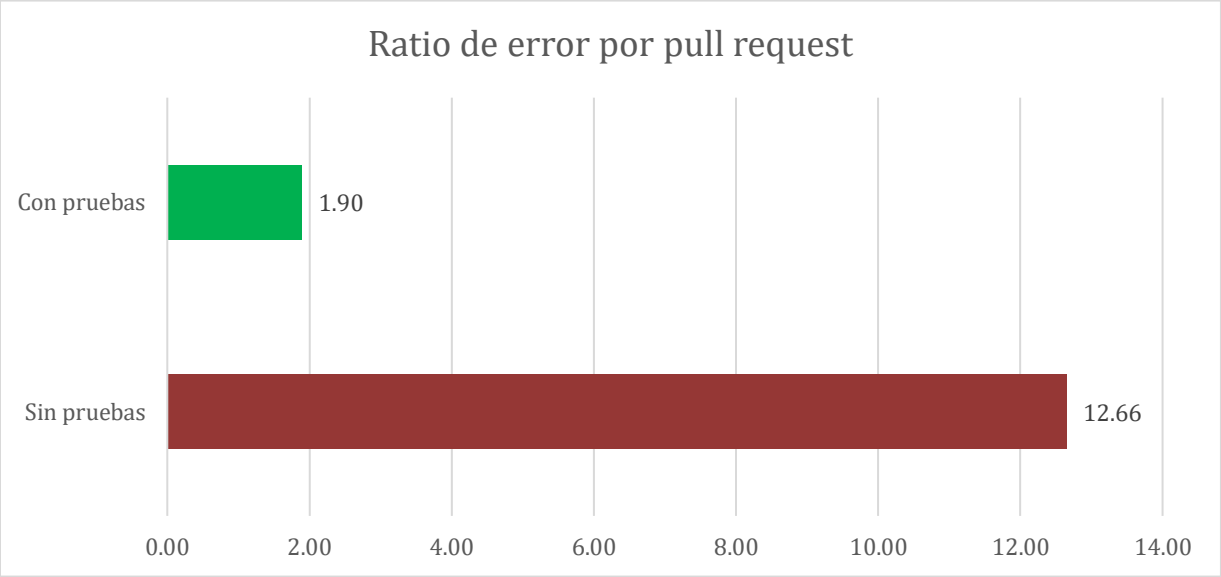

Figura 10: Comparativa del ratio de error por pull request con la implementación de pruebas unitarias y sin ellas.

3.2 Discusión

Los equipos de desarrollo DevOps son diferentes al resto de los equipos de desarrollo de software puesto que estos últimos son los clientes de las soluciones que se diseñan en DevOps, su impacto ha sido benéfico y comprobado por estudios como Accelerate State de Google (2022) en donde en una comparación entre empresas con adopción de practicas agiles y de DevOps se ha notado que existe 973 veces más despliegues de código, 6570 veces menor tiempo desde el commit hasta su despliegue, 3 veces menor el ratio de error y 6570 veces menor el tiempo de recuperación de incidentes.

Mientras que, de acuerdo con los datos recopilados en este estudio, y respondiendo a la primera pregunta de investigación, ¿cuál es la diferencia en los tiempos de aprobación entre pull requests con y sin pruebas unitarias? La diferencia de los tiempos de aprobación es de, 1393 minutos, lo cual implica en una reducción del 58%. Esto puede traducirse a una negativa de la primera hipótesis, "el tiempo de aprobación de pull request se reduce en un 50% con la utilización de pruebas unitarias con Terratest". Sin embargo, existe una mejora mayor a la esperada, por 8 puntos porcentuales.

Por otro lado, en el caso de la segunda pregunta de investigación, ¿cuál es la diferencia de los ratios de error entre pull requests con y sin pruebas unitarias? La diferencia de los ratios de error fue de 10.7 puntos. Esto indica que, nuevamente, de acuerdo con la segunda hipótesis, "el ratio de pull request con errores pasa de un 12.6% a un 6%", esta sea incorrecta. Esto debido a que el ratio de error para pull requests con pruebas unitarias fue menor de lo esperado.

Esto puede denotar la relación que tiene el adoptar prácticas de ingeniería de software, como es el del uso de pruebas unitarias con el trabajo que el mismo ingeniero DevOps realiza, por lo que abre una puerta hacia la adopción de diferentes tipos de pruebas como el de integración, así como la apertura de la discusión hacia las mejores maneras de hacer pruebas en IaC en los equipos de DevOps.

Podría decirse que el impacto que realiza este estudio es el de demostrar una mejora tangible y visible a través de datos sobre la importancia de la realización de prácticas de ingeniería de software, que a la vez son adaptadas en las prácticas de DevOps para el trabajo e implementación del mismo DevOps.

Capítulo 4, Conclusiones y trabajo futuro

Las respuestas a las preguntas de investigación reflejan el éxito de la utilización de la ingeniería de software en procesos de DevOps. Sin embargo, las pruebas unitarias utilizando la librería de Terratest tienen una limitación importante, y es en el caso de infraestructura de alto impacto. Todo este problema recae en la forma en que se realizan las pruebas con dicha librería, puesto que realiza los cambios propuestos, y, en caso de que estos sean defectuosos, pueden afectar a usuarios de los servicios, aplicaciones y plataformas. Para reducir este impacto, puede probarse la resiliencia de los módulos haciendo uso de un ambiente de pruebas. Sin embargo, el riesgo no puede eliminarse.

Por otro lado, dentro de las oportunidades, se encuentra la posibilidad de hacer otro tipo de pruebas, las de integración. Estas podrían ayudar a detectar problemas complejos con las dependencias que existen con otros módulos y por lo regular son las causantes de que los servicios se detengan, lo cual impacta de manera significativa en las ganancias de la Empresa A.

También, se puede aprovechar esta librería para poder realizar pruebas en otra de las herramientas IaC que se utilizan dentro de la Empresa A, CloudFormation.

Finalmente, otra área de oportunidad es la creación de mejores prácticas, patrones y anti-patrones a la hora de realizar pruebas en IaC, así como lo ha detallado Hasan, M. et al. en su investigación “la creación de prácticas en pruebas para IaC puede habilitar a los desarrolladores a probar sus scripts de IaC de manera efectiva y así identificar futuras ramas de investigación que podrían ser de interés para la comunidad de investigación de la Ingeniería de Software” (2020).

Bibliografía

Ahmed, N. (2021). AWS Certified Cloud Practitioner: Study Guide with Practice Questions and Labs (3ra. Edición). IP Specialist.

Anderson, D. J. (2010). Kanban: successful evolutionary change for your technology business. Blue Hole Press.

Bassil, Y. (2012). A Simulation Model for the Waterfall Software Development Life Cycle. International Journal of Engineering & Technology. http://iet-journals.org/archive/2012/may_vol_2_no_5/255895133318216.pdf

Brikman, Y. (2022). Terraform Up and Running (3ra Edición). O'Reilly.

Dan, M. (2022). Cloud Computing: Theory and Practice (3ra. Edición). Morgan Kaufmann.

Dash, C. (2021). Literature Survey on Exploratory Analysis of Waste in IT Industries During Project Development Using Agile–Lean Practices. (Volumen 3). IEMIS 2020.

Farley, D. (2021). Modern Software Engineering: Doing What Works to Build (1ra. Edición). Addison-Wesley Professional.

GitHub. (n.d.). Understanding GitHub Actions. https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

Gruntwork.io. (n.d. a) Quick Start. https://terragrunt.gruntwork.io/docs/getting-started/quick-start/

Gruntwork.io. (n.d. b) Terratest. https://terratest.gruntwork.io/

Gómez, S. & Moraleda E. (2020). Aproximación a la ingeniería de software (2da. Edición). Universitaria Ramón Areces.

Google. (n.d.). Documentation: The Go Programming Language. https://go.dev/doc/

Google. (2022). Accelerate State of DevOps 2021. https://services.google.com/fh/files/misc/report_2021_accelerate_state_of_devops.pdf

Hasan, M. et al. (2020). Testing Practices for Infrastructure as Code. Proceedings oof the 1st SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing. doi:10.1145/3416504.3424334

HashiCorp. (n.d. a) Terraform Documentation. https://www.terraform.io/docs

HashiCorp. (n.d. b) How Terraform Works with Plugins. https://www.terraform.io/plugin/how-terraform-works

Hooda, I. & Singh, R. (2015). Software Test Process, Testing Types and Techniques. International Journal of Computer Applications número 13. http:// 10.1.1.695.1299

Hüttermann, M. (2012). DevOps for Developers. Springer-Verlag New York Inc.

IEEE. (1990) Standard Glossary of Software Engineering Terminology. doi:10.1109/ieeestd.1990.101064

Kolade, C. (2021). What is Go? Golang Programming Language Meaning Explained. FreeCodeCamp. https://www.freecodecamp.org/news/what-is-go-programming-language/

Liviu, M. (2014). Comparative Study on Software Development Methodologies. http://www.dbjournal.ro/archive/17/17_4.pdf

Microsoft. (2021). What is Infrastructure as Code? https://docs.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code

Munassar, N. & Govordhan, A. (2010). A Comparison Between Five Models of Software Engineering. International Journal of Computer Science Issues. https://www.ijcsi.org/papers/7-5-94-101.pdf

Ohno, T. (1988). Toyota Production System: Beyond Large-Scale Production (1ra. Edición). https://doi.org/10.4324/9780429273018

Olan, M. (2003). Unit Testing: Test Early, Test Often. Journal of Computing Sciences in Colleges. https://www.researchgate.net/publication/255673967_Unit_testing_Test_early_test_often

Poppendieck, M. (2012). Lean Software Development: An Agile Toolkit (1ra. Edición). Addison-Wesley Professional.

Prykhodko, S. et al. (2020). Estimating the Efforts of Mobile Application Development in the Planning Phase Using Nonlinear Regression Analysis. Sciendo. https://doi.org/10.2478/acss-2020-0019

Red Hat. (2022). What is CI/CD? https://www.redhat.com/en/topics/devops/what-is-ci-cd

Richter, F. (2022). Amazon Leads $180-Billion Cloud Market. Statista. https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

Sabbir, M. et al. (2017). Comparative Study on Software Methodologies for Effective Software Development. https://www.researchgate.net/publication/316753858_Comparative_Study_on_the_Software_Methodologies_for_Effective_Software_Development

Shylesh, S. (2017). A Study of Software Development Life Cycle Process Models. http://dx.doi.org/10.2139/ssrn.2988291

Williams, L. (2010). Agile Software Development Methodologies and Practices. Advances in Computers, 1-44. doi:10.1016/s0065-2458(10)80001-4

Witting, M. & Witting, A. (2015). Amazon Web Services in Action (2da. Edición). Manning Publications.