Desarrollo de un prototipo de servicio para comparar respuestas de diferentes ambientes en arquitecturas REST

Andres Felipe Gaviria Ocampo

Nota de Aceptación

Certificamos que el presente Trabajo de Grado Satisface, en alcances y calidad, todos los requisitos que demanda un Trabajo de Grado de Maestría.

Mario Julián Mora Cardona, Msc.

Director

Luisa Fernanda Rincon Pérez
Jurado 1

Juan Camilo Parra Martínez
Jurado 2

Aprobado en cumplimiento de los requisitos exigidos por la Pontificia Universidad Javeriana Cali, para optar el título de Magister en Ingeniería de software.

HERNÁN CAMILO ROCHA NIÑO Ph. D.
Decano Facultad de Ingeniería y Ciencias

JUAN CARLOS MARTÍNEZ ARIAS
Director Posgrados de Ingeniería y Ciencias

Cali 04/03/2024

Acta de Correcciones al Documento de Trabajo de Grado

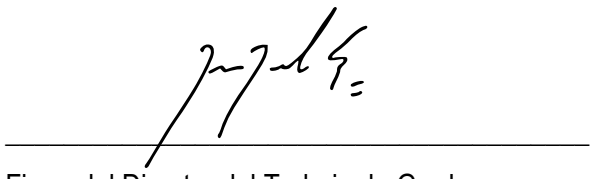
Santiago de Cali, 4/03/2024

Autor:

Título del Trabajo de Grado: "Desarrollo de un prototipo de servicio para comparar respuestas de diferentes ambientes en arquitecturas REST"

Director: Mario Julián Mora Cardona

Como indica el artículo 2.13 de las Directrices para Trabajo de Grado de Maestría, he verificado que el estudiante indicado arriba ha implementado todas las correcciones que los Jurados del Proyecto de Trabajo de Grado definieron que se efectuaran, como consta en el Acta de Evaluación correspondiente.



Firma del Director del Trabajo de Grado

Santiago de Cali, 21 de 11 del 2023

Doctor:
Juan Carlos Martínez Arias
Director Posgrados de Ingeniería y Ciencias
Facultad de Ingeniería
Pontificia Universidad Javeriana Cali

Cumplido los requisitos establecidos en los artículos 5.6 y 5.7 de las Directrices para Trabajo de Grado de Maestría, solicitamos se autorice la sustentación del Trabajo de Grado denominado "Desarrollo de un prototipo de servicio para comparar respuestas de diferentes ambientes en arquitecturas REST", realizado por el (la) estudiante Andrés Felipe Gaviria Ocampo con código 8973244 perteneciente al énfasis en Ingeniería de software, bajo la dirección del profesor Mario Julián Mora Cardona.

El suscrito director del Trabajo de Grado autoriza para que se proceda a hacer su sustentación ante el Tribunal que para el efecto se designe, toda vez que ha revisado meticulosamente el documento y avala que el Trabajo de Grado ya se encuentra listo para ser evaluado oficialmente.

Atentamente,

Andrés Felipe Gaviria Ocampo

C.C. 1144096389 de Cali

Mario Julián Mora Cardona

C.C. 94.400.220 de Cali

Documentación anexa:

Dos copias anilladas del documento de Trabajo de Grado, con impresión por lado y lado y paginación completa.
El resumen del Trabajo de Grado en formato electrónico (máximo 1 página).

Ficha Resumen
Proyecto de Trabajo de Grado

Desarrollo de un prototipo de servicio para probar respuestas de diferentes ambientes en arquitecturas REST.

1. Área de trabajo: Detección de errores (Prueba de calidad).
2. Tipo de proyecto: Aplicado
3. Estudiante: Andrés Felipe Gaviria Ocampo
4. Correo electrónico: fgaviriaocam@javerianacali.edu.co
5. Dirección y teléfono: Tulipanes del castillo casa 204, 3164394974
6. Director: Mario Julian Mora Cardona
7. Vinculación del director: Magister en ingeniería
8. Correo electrónico del director: mariomora@javerianacali.edu.co
9. Co-Director (Si aplica):
10. Grupo o empresa que lo avala (Si aplica):
11. Otros grupos o empresas:
12. Palabras clave(al menos 5): Pruebas, Calidad, Caja negra, regresión, comparador de respuestas.
13. Fecha de inicio: 24 de Julio de 2023
14. Duración estimada: 6 meses
15. Resumen: Desarrollo de un prototipo de servicio web que permita comparar las respuestas de diferentes ambientes antes de los despliegues productivos. El resultado esperado es reducir los errores inyectados en producción.

Desarrollo de un prototipo de servicio para comparar respuestas de diferentes ambientes en arquitecturas REST

Andrés Felipe Gaviria Ocampo

Proyecto presentada(o) como requisito parcial para optar al título de:
Magister en Ingeniería de Software

Director:
MSc. Mario Julián Mora Cardona

Pontificia Universidad Javeriana Cali
Facultad de Ingeniería
Departamento de Electrónica y Ciencias de la Computación
Cali, Colombia
19 de abril de 2024

Índice

Índice de figuras

Agradecimientos

Con profundo agradecimiento, inicio expresando mi reconocimiento a Dios, fuente de fortaleza y guía en este viaje académico, quien ha sido faro y sostén a lo largo de este exigente proceso. Agradezco también a las personas que han sido pilares fundamentales durante esta travesía académica. Quiero mencionar especialmente a mi amada esposa, Daniela Moreno, a quien agradezco enormemente por su inquebrantable apoyo, su paciencia inagotable y su comprensión constante. Su aliento y amor han sido mi mayor motivación, permitiéndome superar los desafíos con determinación y optimismo. También, quiero expresar mi agradecimiento a mi familia, cuyo respaldo incondicional ha sido un apoyo constante en cada paso de mi trayectoria académica. Sus palabras de aliento han sido un recordatorio constante sobre la importancia de perseverar en la búsqueda de mis sueños.

Mi reconocimiento y gratitud se extienden al MSc. Mario Julian Mora Cardona, mi asesor, cuya orientación experta, conocimiento profundo y dedicación incansable han enriquecido de manera significativa este trabajo de investigación. Sus valiosos consejos y sugerencias han sido la brújula que ha guiado mis esfuerzos, brindándome perspectivas fundamentales para el desarrollo de este proyecto. Además, quiero expresar mi agradecimiento a los profesores de las materias de calidad de software, patrones de diseño y arquitectura de software en la Pontificia Universidad Javeriana Cali. Sus enseñanzas y las herramientas proporcionadas durante mis estudios fueron fundamentales para abordar los desafíos específicos de este proyecto, constituyendo una base sólida que ha contribuido a mi formación integral.

No puedo dejar de mencionar mi agradecimiento a la Pontificia Universidad Javeriana Cali por brindar el entorno académico propicio para mi formación. La calidad excepcional de la educación recibida ha sido un factor determinante en mi desarrollo tanto profesional como personal. A través de esta experiencia, he podido cultivar no solo conocimientos técnicos, sino también habilidades y valores que me han preparado para enfrentar los retos del mundo laboral y contribuir de manera significativa a la sociedad.

Resumen

Los servicios web basados en la arquitectura REST (Representational State Transfer) están ganando popularidad en el ámbito empresarial. Las empresas del sector tecnológico tienen un núcleo de negocio basado en los servicios web, los cuales son vitales para la ejecución de las operaciones comerciales. La descomposición del negocio en microservicios ha traído la dificultad para evaluarlos. Es crucial garantizar la calidad del código y proporcionar un nivel de asegurabilidad que proteja al negocio de posibles errores.

En este contexto, la propuesta de investigación de este proyecto tiene como propósito desarrollar un prototipo de servicio que mejore la observabilidad de posibles errores en las respuestas antes de su implementación en un entorno productivo. Este servicio no reemplazará otras pruebas existentes en la organización, sino que se integra como un paso adicional en el proceso de aseguramiento de calidad.

Palabras Clave Pruebas de caja negra, comparador, calidad de código, Rest, deployment, protocolo, unit test, pruebas de regresión.

Abstract

Web services based on the REST (Representational State Transfer) architecture are gaining popularity in the business environment. Each company has a core business based on web services, which are vital for carrying out business operations. The decomposition of the business into microservices has brought the challenge of evaluating them. It is crucial to ensure code quality and provide a level of assurance that protects the business from potential errors.

In this context, the research proposal of this project aims to develop a prototype service that enhances the observability of potential errors in code behavior before its implementation in a production environment. This service will not replace other existing tests in the organization but will be integrated as an additional step in the quality assurance process.

Keywords Black box testing, comparator, code quality, REST, deployment, protocol, unit test, regression testing.

1. Introducción

En el ámbito del desarrollo de software, la gestión de múltiples entornos, como el productivo y de pruebas, presenta desafíos significativos. Los errores que pasan desapercibidos en etapas tempranas pueden manifestarse en el entorno productivo, impactando negativamente en la calidad del software y generando consecuencias adversas tanto para los desarrolladores como para los usuarios finales. En este contexto, surge la necesidad de una herramienta que permita identificar estos errores antes de su despliegue en producción.

Este proyecto se benefició de los fundamentos sólidos proporcionados por las asignaturas de calidad de software, patrones de diseño y arquitectura de software en la Universidad Javeriana. Estos cursos proporcionaron las herramientas conceptuales y prácticas esenciales para abordar de manera efectiva los desafíos del desarrollo de este proyecto.

El proyecto se centra en el desarrollo de un prototipo de servicio destinado a comparar respuestas en entornos REST, ofreciendo una solución integral para la detección temprana de discrepancias y la mejora continua de la calidad del software. La relevancia de este proyecto radica en su capacidad para prevenir errores en el código y optimizar la experiencia del usuario final, contribuyendo a la consolidación de prácticas de desarrollo más seguras y eficientes.

En la presente tesis, abordaremos detalladamente la problemática asociada con la gestión de entornos de desarrollo, destacando las consecuencias de errores y la importancia de estrategias proactivas. La justificación de este trabajo se fundamenta en la necesidad de contar con herramientas que permitan anticipar posibles fallas, reduciendo así el impacto negativo en la operatividad y confiabilidad del sistema. La metodología empleada se adapta a las características específicas del proyecto, optando por enfoques por objetivos seccionados por etapas. Este enfoque facilita una comprensión clara de los desafíos y metas, permitiendo un abordaje efectivo a lo largo del desarrollo del prototipo.

2. Objetivos del proyecto

2.1. Objetivo General

Diseñar e implementar un prototipo de un servicio web que permita realizar comparaciones de comportamientos basado en respuestas REST entre diferentes ambientes, para identificar y mostrar discrepancias.

2.2. Objetivos específicos

- Identificar las consecuencias y repercusiones que los errores puede tener en el funcionamiento del sistema, la calidad del software y la experiencia del usuario.
- Desarrollar un servicio web que permita comparar las respuestas, con el fin de identificar y mostrar las discrepancias en los servicios REST.
- Elaborar una caracterización de las posibles discrepancias a encontrar.
- Realizar pruebas de validación del servicio desarrollado para asegurar su funcionamiento.

2.3. Resultados esperados

- Un prototipo de un servicio web que permita realizar informes de las discrepancias encontradas en las respuestas de dos servicios Rest.
- Análisis de la implementación del prototipo en un contexto real.

3. Marco teórico de referencia y antecedentes

3.1. Bases Teóricas

3.1.1. Pruebas de software

3.1.1.1. ¿Qué son las pruebas de software?

En la investigación titulada "Software Testing Techniques: A Literature Review" Jamil et al. (2017) ofrece una valiosa perspectiva sobre el concepto y la importancia de las pruebas de software en el desarrollo de sistemas. Las pruebas de software se definen como un proceso esencial de evaluación que tiene como objetivo determinar si un sistema cumple con los requisitos previamente especificados. Estas pruebas permiten comparar el resultado real obtenido durante la ejecución del software con el resultado esperado, lo que revela la presencia de posibles fallas o errores en el software desarrollado y proporciona información clave sobre el estado de la calidad del producto.

Las pruebas de software se llevan a cabo mediante la ejecución de diferentes técnicas y enfoques. En su investigación sobre técnicas de pruebas, Rani and Gupta (2018) mencionan que existen diversas técnicas, tales como pruebas unitarias, pruebas de integración, pruebas funcionales y pruebas de regresión, que se utilizan para evaluar diferentes aspectos del software durante su desarrollo y mantenimiento.

El proceso de pruebas de software se basa en la creación de casos de prueba que cubran diferentes escenarios y condiciones del sistema, incluyendo datos de entrada específicos y situaciones de uso realistas. Estos casos de prueba se ejecutan sistemáticamente y se comparan los resultados obtenidos con los resultados esperados, lo que permite detectar cualquier desviación y asegurar la corrección y la confiabilidad del software.

3.1.1.2. Importancia de las pruebas en el desarrollo de software

Las pruebas desempeñan un papel importante en la mejora de la confiabilidad y estabilidad del software. En el libro "The Art of Software Testing" Myers et al. (2011), destacan que las pruebas rigurosas permiten descubrir y solucionar errores antes de que el software sea utilizado por los usuarios finales, lo que reduce la posibilidad de fallos inesperados y aumenta la satisfacción del cliente. La detección temprana y la corrección de errores también disminuyen los riesgos asociados con el uso de aplicaciones defectuosas, como pérdida de datos o interrupción de servicios críticos.

Además de la detección de errores, las pruebas proporcionan información valiosa sobre la calidad del producto. Los autores Jamil et al. (2017) señalan que las pruebas permiten comparar los resultados reales con los esperados, identificando cualquier discrepancia y proporcionando una evaluación objetiva del estado del software. Este conocimiento de la calidad del producto ayuda a tomar decisiones informadas sobre su lanzamiento y proporciona confianza tanto a los desarrolladores como a los usuarios finales.

3.1.1.3. Tipos de pruebas de software

Existen diversos tipos de pruebas que se aplican con el objetivo de evaluar diferentes aspectos del sistema y garantizar su calidad. A continuación, se describen algunos de los principales tipos de pruebas utilizados en la industria:

- **Pruebas funcionales:** se centran en verificar si el software cumple con los requisitos funcionales especificados. Según Sommerville (2016), estas pruebas se basan en diseñar casos de prueba que evalúen el comportamiento del sistema ante diferentes entradas y condiciones. Su objetivo principal es determinar si el software realiza correctamente las funciones esperadas, sin errores o resultados inesperados, como se evidencia en la figura 1.

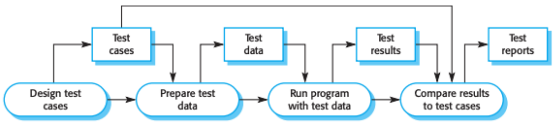


Figura 1: A model of the software testing process
Sommerville (2016)

- **Pruebas de rendimiento:** las pruebas de rendimiento se utilizan para evaluar el desempeño y la capacidad de respuesta del software bajo condiciones específicas de carga y estrés. De acuerdo con Almeida et al. (2004), estas pruebas se enfocan en medir aspectos como el tiempo de respuesta, la velocidad de procesamiento y el consumo de recursos del sistema. Su objetivo es identificar posibles cuellos de botella y optimizar el rendimiento del software.

- **Pruebas de usabilidad:** las pruebas de usabilidad se centran en evaluar la facilidad de uso y la experiencia del usuario al interactuar con el software. Según Nielsen (1993), estas pruebas se realizan con usuarios reales, quienes proporcionan retroalimentación sobre la interfaz, la navegación y la comprensión de las funcionalidades. El objetivo es identificar problemas de usabilidad y realizar mejoras para aumentar la satisfacción del usuario.

- **Pruebas de seguridad:** las pruebas de seguridad se realizan para evaluar la resistencia del software ante ataques maliciosos y garantizar la protección de los datos y la integridad del sistema. Según Howard and Lipner (2006), estas pruebas implican la identificación de vulnerabilidades y la aplicación de técnicas de penetración para verificar la robustez de las medidas de seguridad implementadas.

3.1.2. Estrategias de pruebas

Las estrategias de pruebas son enfoques sistemáticos utilizados para diseñar y ejecutar pruebas de software con el objetivo de lograr una cobertura adecuada y garantizar la calidad del producto final. A continuación, se describen algunas de las

estrategias de pruebas más comunes y su importancia en el proceso de desarrollo de software.

3.1.3. Concepto de estrategias de pruebas

El concepto de estrategias de pruebas hace referencia a enfoques sistemáticos y planificados para realizar pruebas de software con el objetivo de garantizar la calidad y fiabilidad del producto. Estas estrategias implican la selección y combinación adecuada de técnicas, herramientas y recursos disponibles para lograr una cobertura exhaustiva de las funcionalidades del software y detectar posibles fallas o errores. A continuación, se presenta una descripción del concepto de estrategias de pruebas.

Según Bertolino (2007), las estrategias de pruebas son "planes y enfoques organizados que guían el proceso de pruebas y proporcionan una estructura para garantizar la efectividad y eficiencia de las actividades de prueba". Estas estrategias ayudan a identificar los tipos de pruebas adecuados, las técnicas y herramientas a utilizar, así como la asignación de recursos y la definición de objetivos de prueba.

3.1.3.1. Enfoques comunes en estrategias de pruebas

- **Pruebas de caja blanca (white-box):** se basa en el conocimiento interno de la estructura y el diseño del software. Se analiza el código fuente y se crean casos de prueba para cubrir diferentes caminos de ejecución, condiciones lógicas y ramas del código. Las pruebas de caja blanca son útiles para verificar la lógica interna del software y garantizar la cobertura de todas las sentencias y condiciones. Según Myers et al. (2011), las pruebas de caja blanca son especialmente efectivas para encontrar errores de lógica y de flujo de control.
- **Pruebas de caja negra (black-box):** se centra en probar el software sin conocer su estructura interna. Las pruebas se basan únicamente en los requisitos y la especificación del software. Se diseñan casos de prueba para cubrir

diferentes escenarios y se evalúa la respuesta del software frente a diferentes entradas. Las pruebas de caja negra son útiles para validar la funcionalidad y la usabilidad del software desde la perspectiva del usuario. Según Pressman (2010), las pruebas de caja negra se centran en verificar si el software cumple con los requisitos funcionales y no funcionales.

- **Pruebas unitarias:** este enfoque se centra en probar las unidades individuales de código fuente, como funciones, métodos o clases. Se busca verificar la corrección de cada unidad y su interacción con otras unidades. Según Binder (2000), las pruebas unitarias se realizan durante el desarrollo para detectar y corregir errores tempranos.

- **Pruebas de marcha blanca controlada:** este enfoque se centra en probar el software en un entorno similar al de producción, utilizando datos reales. Se busca validar el comportamiento del sistema en condiciones reales antes de su implementación. Según Kaner (1999), las pruebas de marcha blanca controlada permiten identificar problemas relacionados con la configuración, la interoperabilidad y el desempeño del software.

3.1.3.2. Definición de casos de prueba

Los casos de prueba son un componente esencial en el proceso de pruebas de software. Los casos de prueba son un conjunto de condiciones o acciones diseñadas para evaluar el comportamiento de un sistema o componente específico. Estos casos representan situaciones reales o escenarios que se utilizan para verificar si el software cumple con los requisitos establecidos.

La definición de casos de prueba varía dependiendo de la metodología de pruebas utilizada. Según Pressman (2010), los casos de prueba pueden estar basados en requisitos, en los que cada caso se enfoca en verificar un requisito específico, o pueden ser escenarios realistas que simulan la interacción del usuario con el sistema. En ambos casos, los casos de prueba deben ser claros, comprensibles y medibles, de manera que

se pueda determinar si el software pasa o falla cada caso de prueba.

3.1.3.3. Componentes de un caso de prueba

Identificador único: cada caso de prueba debe tener un identificador único que lo distinga de otros casos. Esto facilita la referencia y seguimiento durante el proceso de pruebas. Según L. Beizer, "el uso de identificadores únicos es fundamental para la gestión efectiva de los casos de prueba" Beizer (1990).

Descripción: se debe proporcionar una descripción clara y concisa del objetivo del caso de prueba. Esto incluye los requisitos o funcionalidades específicas que se están probando. Según J. Fewster y D. Graham, "la descripción debe ser comprensible y brindar una visión clara de lo que se espera probar" Fewster and Graham (1999).

Entradas: especifica los datos o condiciones de entrada que se utilizan para ejecutar el caso de prueba. Estas entradas pueden incluir valores específicos, archivos de prueba, configuraciones o cualquier otra información relevante. S. Kaner et al. afirman que "las entradas deben ser lo suficientemente variadas como para cubrir diferentes escenarios y casos límite" Kaner (1999).

Pasos de prueba: son las instrucciones detalladas que describen cómo ejecutar el caso de prueba. Estos pasos deben ser claros, precisos y secuenciales, de manera que cualquier persona pueda seguirlos fácilmente. Según R. Patton, "los pasos de prueba deben ser lo suficientemente detallados como para proporcionar una guía precisa de lo que se debe hacer" Patton (2005).

Resultados esperados: indica los resultados o comportamientos esperados del sistema bajo prueba después de ejecutar el caso. Esto permite comparar los resultados reales con los esperados y determinar si el software se comporta según lo previsto. K. Wiegers señala que "los resultados esperados deben ser específicos y medibles" Wiegers (2003).

Criterios de aceptación: establece los criterios que deben cumplirse para considerar que el caso de prueba es exitoso. Estos criterios pueden incluir valores específicos, mensajes de error esperados o cualquier otra condición relevante. Según D. Graham y M. Perry, "los criterios de aceptación deben ser claros y objetivos" Fewster and Graham (1999).

3.1.3.4. Técnicas de diseño de casos de prueba

Técnica del caso de prueba basado en especificaciones: la técnica del caso de prueba basado en especificaciones se centra en la revisión de los requisitos y especificaciones del software para identificar condiciones de prueba. Esta técnica se basa en comprender y analizar los documentos de requisitos y utilizar esa información para diseñar casos de prueba relevantes.

Según Beizer (1990), esta técnica ayuda a garantizar que los casos de prueba estén alineados con los requisitos del sistema y cubran adecuadamente los escenarios especificados.

Técnica del caso de prueba basado en estructura: la técnica del caso de prueba basado en estructura se enfoca en la estructura interna del software, como el código fuente y la arquitectura del sistema. Esta técnica utiliza conocimientos detallados sobre el diseño interno del software para identificar rutas de ejecución y puntos críticos que deben ser probados.

Según Fewster and Graham (1999), esta técnica permite diseñar casos de prueba que se centren en la lógica interna del software y ayuden a descubrir errores relacionados con la implementación.

Técnica del caso de prueba basado en experiencia: la técnica del caso de prueba basado en experiencia se basa en el conocimiento y la experiencia acumulada de los profesionales en pruebas de software. Estos profesionales utilizan su experiencia previa en proyectos similares para identificar áreas críticas, situaciones inusuales y escenarios relevantes que deben ser probados.

Técnica del caso de prueba basado en errores conocidos:la técnica del caso de prueba basado en errores conocidos se basa en el análisis de errores y fallas anteriores en el software para diseñar casos de prueba que aborden específicamente esas áreas problemáticas. Esta técnica se centra en aprender de los errores del pasado y diseñar pruebas que ayuden a prevenir la aparición de errores similares en el futuro.

3.1.4. Automatización de pruebas

3.1.4.1. Concepto de automatización de pruebas

La automatización de pruebas es un proceso que consiste en utilizar herramientas y software especializado para ejecutar pruebas de manera automatizada, en lugar de realizarlas manualmente. Esta técnica permite mejorar la eficiencia y la precisión de las pruebas, al reducir la intervención humana y proporcionar resultados rápidos y confiables. A continuación, se presentan algunas definiciones y perspectivas sobre el concepto de automatización de pruebas.

Según Beizer (1990), la automatización de pruebas se refiere al uso de herramientas y técnicas para ejecutar pruebas repetitivas, secuenciales y predecibles de forma automatizada. Estas pruebas automatizadas pueden abarcar desde pruebas unitarias hasta pruebas de sistema, permitiendo una mayor cobertura y una detección más temprana de errores.

Binder (2000) define la automatización de pruebas como el proceso de escribir y ejecutar scripts o programas que realizan pruebas automáticamente, sin la intervención manual del tester. Esta automatización puede incluir la interacción con la interfaz de usuario, la manipulación de datos y la verificación de resultados esperados.

3.1.4.2. Ventajas y desafíos de la automatización de pruebas

Ventajas

La automatización permite ejecutar pruebas de manera rápida y eficiente, reduciendo el tiempo y los recursos necesarios en comparación con las pruebas manuales Sommerville (2016). A continuación se presentan las ventajas de la automatización

Mayor cobertura de pruebas: La automatización facilita la ejecución de un gran número de pruebas en un período de tiempo limitado, lo que aumenta la cobertura de pruebas y permite detectar más errores potenciales Kaczmarek et al. (2003).

Reutilización de casos de prueba: Los casos de prueba automatizados pueden ser reutilizados en diferentes versiones del software, lo que ahorra tiempo en la creación de nuevos casos de prueba y garantiza una cobertura consistente Salam and Mohd (2013).

Mejora de la precisión y confiabilidad: Al eliminar la intervención manual, se reducen los errores humanos y se obtienen resultados más consistentes y confiables en las pruebas Papadakis et al. (2018).

Desafíos

La automatización de pruebas requiere una inversión inicial en herramientas y recursos de desarrollo, así como tiempo para desarrollar los scripts y mantener las pruebas actualizadas Ammann and Offutt (2008). A continuación se muestran los principales desafíos

Selección adecuada de pruebas para automatizar: No todas las pruebas son adecuadas para la automatización. Es importante identificar qué pruebas se beneficiarán más de la automatización y cuáles deben seguir siendo ejecutadas manualmente Kaner et al. (2004).

Mantenimiento y actualización de las pruebas: A medida que el software evoluciona, las pruebas automatizadas deben ser actualizadas y mantenidas para garantizar su relevancia y efectividad continua Jorgensen and Shepperd (2007).

Necesidad de habilidades y conocimientos técnicos: La automatización de pruebas requiere personal con habilidades técnicas y conocimientos en el uso de herramientas y lenguajes de programación específicos Sommerville (2016).

3.1.4.3. Herramientas y tecnologías para la automatización de pruebas

La automatización de pruebas en el desarrollo de software se ha vuelto cada vez más importante debido a su capacidad para mejorar la eficiencia y la calidad de las pruebas. Existen diversas herramientas y tecnologías disponibles para facilitar este proceso. A continuación, se presentan algunas de las herramientas más utilizadas.

Selenium: selenium es una popular suite de herramientas de automatización de pruebas para aplicaciones web. Permite la automatización de acciones en navegadores web, como hacer clic en botones, llenar formularios y navegar por páginas. Esta herramienta es ampliamente adoptada y cuenta con una gran comunidad de usuarios y desarrolladores.

Appium: Appium es una herramienta de automatización de pruebas específicamente diseñada para aplicaciones móviles. Permite realizar pruebas en dispositivos iOS, Android y Windows utilizando un único conjunto de API. Appium es conocido por su compatibilidad con múltiples plataformas y por ofrecer un enfoque basado en estándares para la automatización de pruebas móviles.

JUnit: JUnit es un popular framework de pruebas unitarias para aplicaciones Java. Proporciona una estructura para escribir y ejecutar pruebas unitarias de manera eficiente. JUnit ofrece funciones para realizar aserciones, configurar y limpiar el entorno de prueba, y organizar las pruebas en suites. Es ampliamente utilizado en el desarrollo de software basado en Java.

TestNG: TestNG es otro framework de pruebas unitarias para aplicaciones Java. Ofrece características avanzadas en comparación con JUnit, como la capacidad de realizar pruebas en paralelo, la definición de dependencias entre pruebas y la ge-

neración de informes detallados. TestNG se ha vuelto popular en la comunidad de desarrollo de Java debido a su flexibilidad y funcionalidad mejorada.

Cucumber: Cucumber es una herramienta de automatización de pruebas basada en el lenguaje de dominio específico Gherkin. Permite escribir pruebas en un formato legible por humanos utilizando el enfoque de desarrollo dirigido por comportamiento (BDD). Cucumber se utiliza ampliamente para la colaboración entre los equipos de desarrollo y de pruebas, ya que facilita la comprensión y la comunicación de los requisitos y escenarios de prueba.

3.1.5. Comparación de respuestas en ambientes productivos y de pruebas

3.1.5.1. Importancia de comparar las respuestas en diferentes ambientes

En el desarrollo de software, es crucial comparar las respuestas y el comportamiento del sistema en diferentes ambientes, como el ambiente productivo y el ambiente de pruebas. Esta comparación proporciona información valiosa sobre la estabilidad, la eficiencia y la calidad del software.

En el estudio titulado "A Comparative Analysis of Production and Test Failures" realizado por Dallmeier et al. (2015), se demostró que la comparación de las respuestas en diferentes ambientes revela patrones y tendencias en los errores y fallas del software. Esto ayuda a los equipos de desarrollo a comprender mejor los puntos débiles del sistema y tomar medidas correctivas de manera más efectiva.

En el artículo "Comparing Production and Test Coverage" escrito por Andrews et al. (2006), se destaca que la comparación de las respuestas en diferentes ambientes permite evaluar la efectividad de las pruebas realizadas durante el ciclo de desarrollo. Ayuda a identificar las áreas del sistema que no están adecuadamente cubiertas por las pruebas y que pueden ser fuentes potenciales de errores.

La comparación de las respuestas en diferentes ambientes proporciona una visión integral del software y ayuda a identificar posibles problemas y deficiencias. Permite realizar ajustes y mejoras antes de que el software se despliegue en el ambiente productivo, lo que a su vez contribuye a la entrega de un producto de alta calidad.

3.1.5.2. Diferencias comunes entre el ambiente productivo y el ambiente de pruebas

En el desarrollo de software, existen diferencias significativas entre el ambiente productivo y el ambiente de pruebas. Estas diferencias pueden afectar el comportamiento y rendimiento del sistema, así como la detección de errores y la validación de su correcto funcionamiento.

Según la investigación realizada por Andrews et al. (2006) sobre la comparación de la cobertura de producción y pruebas, se observó que los errores y fallas en el ambiente productivo suelen ser diferentes a los encontrados durante las pruebas. Esto se debe a que el ambiente de pruebas generalmente no puede reproducir exactamente las condiciones del ambiente productivo, lo que resulta en la aparición de fallas que no fueron detectadas durante las pruebas.

Además, Dallmeier et al. (2015) realizaron un análisis comparativo de las fallas en producción y pruebas, y encontraron que las fallas en el ambiente productivo tienden a ser más críticas y tener un impacto más significativo en los usuarios finales. Esto se debe a que el ambiente productivo está expuesto a una mayor variedad de configuraciones, datos reales y cargas de trabajo, lo que puede revelar problemas que no fueron detectados en el ambiente de pruebas controlado.

Otra diferencia importante es la presencia de datos reales en el ambiente productivo, que pueden tener características y patrones que no se encuentran en los datos de prueba. Jones and Bonsignour (2012) enfatizan en su libro "The Economics of Software Quality" que la variabilidad en los datos de producción puede exponer vulnerabilidades y errores que no fueron considerados en las pruebas.

Las diferencias comunes entre el ambiente productivo y el ambiente de pruebas incluyen la presencia de errores no detectados durante las pruebas, la aparición de fallas más críticas en producción y la variabilidad de los datos reales. Estas diferencias resaltan la importancia de comparar las respuestas entre ambos ambientes para identificar y solucionar posibles problemas que pueden surgir en la implementación del software.

3.2. Contexto en el que se desarrolló el proyecto aplicado

3.2.1. 99Minutos

En la era digital en la que vivimos, la logística y la entrega de productos se han convertido en pilares fundamentales para el éxito de cualquier negocio. Empresas de comercio electrónico, restaurantes, tiendas minoristas y muchos otros sectores dependen de la eficiencia en la distribución de sus productos. En América Latina, una empresa ha emergido como un líder en este campo: 99minutos.

América Latina es una región diversa y vasta, que abarca desde México hasta Chile y Argentina. A pesar de su diversidad, comparte desafíos logísticos comunes. Las distancias geográficas, la infraestructura variable y la congestión del tráfico en las ciudades principales son solo algunos de los obstáculos que enfrentan las empresas que desean entregar productos de manera rápida y confiable.

En Golan (2022) menciona que la empresa 99minutos fue fundada en México en 2014, surgió como una solución a estos desafíos logísticos. La empresa comenzó como un servicio de entrega exprés en la Ciudad de México y rápidamente se expandió para atender a otras ciudades importantes de México. Su enfoque en la entrega rápida y eficiente capturó la atención de comercios electrónicos, restaurantes y empresas de diversos sectores.

Una de las claves del éxito de 99minutos ha sido su enfoque en la tecnología. La empresa ha desarrollado una plataforma de software robusta que optimiza las rutas

de entrega y permite a los clientes rastrear en tiempo real el progreso de sus envíos. Esta transparencia y eficiencia han sido fundamentales para ganar la confianza de sus clientes y la visión de 99minutos no se detuvo en las fronteras de México. La empresa se ha expandido a lo largo de América Latina, llegando a países como Colombia, Chile y Perú. Esta expansión ha sido estratégica, ya que ha permitido a 99minutos ofrecer sus servicios en toda la región, aprovechando su experiencia en la resolución de desafíos logísticos complejos.

Para respaldar su crecimiento, según Parra (2023) menciona que 99minutos ha establecido alianzas estratégicas con empresas líderes en la industria. Colabora estrechamente con comercios electrónicos, restaurantes de comida rápida y minoristas, lo que ha contribuido a su rápido ascenso en el mundo de la logística.Además de su éxito comercial, 99minutos se ha comprometido con la sostenibilidad y la responsabilidad social. La empresa ha implementado medidas para reducir su huella ambiental y ha promovido la inclusión laboral y el desarrollo profesional de sus empleados.

En un mundo donde la entrega rápida es esencial, 99minutos se ha convertido en un actor fundamental en la transformación de la logística en América Latina. Su historia de éxito muestra cómo la innovación tecnológica y el compromiso con la eficiencia pueden superar los desafíos logísticos en una región diversa y en constante evolución. A medida que la empresa continúa creciendo y expandiéndose, es probable que siga desempeñando un papel crucial en la forma en que las empresas en América Latina entregan sus productos y servicios. Con su enfoque en la tecnología y la mejora continua, 99minutos está marcando el rumbo para el futuro de la logística en la región.

3.2.2. Coverage Microservice

El servicio Coverage Microservice está diseñado como un componente esencial en el ecosistema de 99minutos. Se desarrolló en el lenguaje de programación Go y ofrece dos protocolos de conexión distintos, GRPC y REST, a través del mismo puerto.

La razón detrás de esta dualidad es proporcionar un estándar de respuesta rápido a los servicios internos de la organización mediante GRPC, mientras que el protocolo REST está destinado a clientes externos que no cuentan con la tecnología necesaria para interactuar mediante GRPC.

El objetivo principal de Coverage Microservice es determinar la disponibilidad de entrega de 99minutos en ubicaciones específicas. Esto se logra mediante un endpoint llamado GetRoute que implementa diversas estrategias lógicas diseñadas para evaluar si 99minutos puede realizar una entrega desde un punto de origen (A) hasta un punto de destino (B). Cada estrategia se adapta a condiciones específicas y, según estas condiciones, se ejecutan consultas particulares para evaluar la viabilidad de la entrega. La importancia de este servicio radica en proporcionar información precisa sobre la cobertura de entrega de 99minutos en diferentes áreas geográficas. Cabe destacar que si el servicio falla, 99minutos se considerará fuera de cobertura, lo que afectará la capacidad de ofrecer cualquier servicio logístico. Además, otros servicios dependen directamente de Coverage Microservice, por lo que su correcto funcionamiento es crucial.

Para llevar a cabo su función, Coverage Microservice utiliza una base de datos MongoDB en la que almacena y busca datos GeoJSON para ubicar áreas mediante coordenadas de latitud y longitud. Esto permite identificar estaciones, las cuales deben estar interconectadas. En otras palabras, para que una estación A (Origen) y una estación B (Destino) estén dentro de la cobertura, deben estar conectadas en un grafo como se muestra en la figura 2. Cada país posee su propio grafo, ya que las actualizaciones de cobertura se gestionan de manera independiente para cada país en el que opera 99minutos.

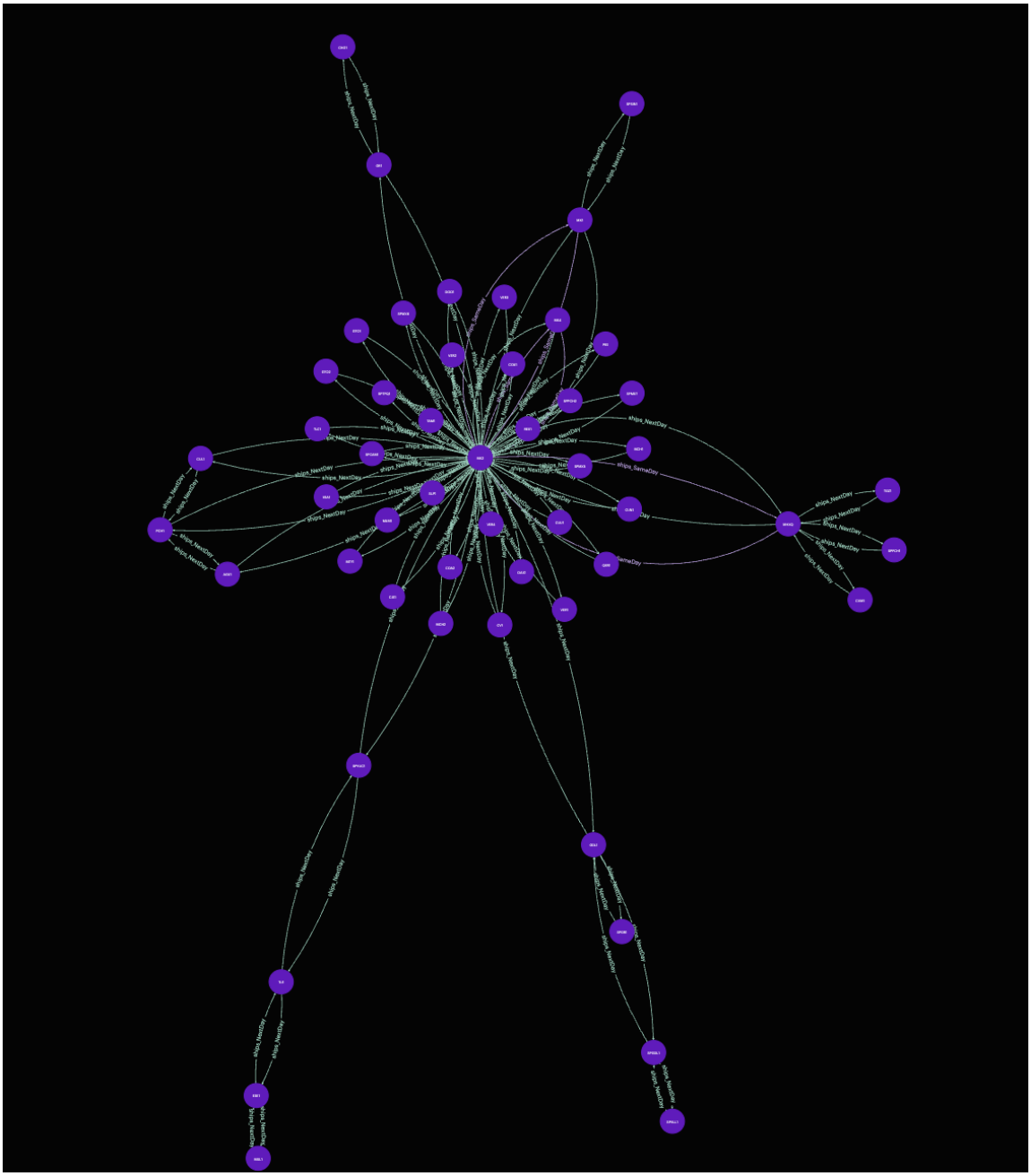


Figura 2: Grafo de los nodos de cobertura.

El servicio ofrece aproximadamente 20 endpoints, pero en el contexto de este pro-

yecto de grado, nos centraremos en el endpoint más crítico: GetRoute. Este endpoint es especialmente relevante, ya que maneja un alto volumen de peticiones diarias, que oscila entre 45.000 y 97.000, como se muestra en la figura 3.

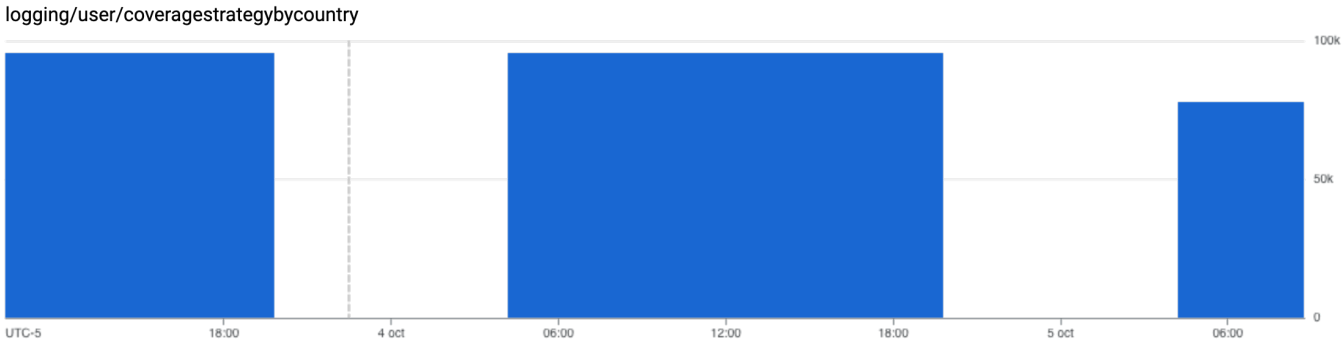


Figura 3: Numero de peticiones para el endpoint GetRoute.

3.3. Estado del Arte

3.3.1. Antecedentes de aplicaciones que permita comparar respuestas de diferentes ambientes

A continuación se presenta un analisis de las herramientas disponibles para realizar pruebas de comparación de respuestas rest api en diferentes ambientes. El análisis se realiza a partir de la documentación disponible en cada herramienta.

3.3.1.1. Postman

Postman se utiliza principalmente para realizar pruebas de API. Permite enviar solicitudes HTTP a un servidor o servicio web específico y recibir respuestas en diferentes formatos, como JSON o XML. Algunas de las funcionalidades y características clave son:

Creación y envío de solicitudes: proporciona una interfaz intuitiva para construir y enviar solicitudes HTTP. Permite especificar la URL, los parámetros, los

encabezados y el cuerpo de la solicitud, y realizar solicitudes GET, POST, PUT, DELETE, entre otras.

Gestión de entornos y variables: permite definir entornos y variables que facilitan la configuración y personalización de las pruebas. Esto es especialmente útil cuando se trabaja con diferentes ambientes, como desarrollo, prueba y producción.

Automatización de pruebas: permite la automatización de pruebas mediante la creación de scripts utilizando JavaScript. Estos scripts pueden realizar pruebas de regresión, pruebas de integración y otras pruebas repetibles, lo que agiliza el proceso de pruebas.

Colecciones de solicitudes: permite organizar las solicitudes en colecciones, lo que facilita la reutilización y el mantenimiento de las pruebas. Las colecciones pueden contener diferentes solicitudes, así como scripts y variables asociadas.

Generación de documentación: permite generar documentación de API automáticamente a partir de las solicitudes y respuestas configuradas. Esto es útil para documentar y compartir la información sobre la API con otros miembros del equipo.

3.3.1.2. API Fortress

API Fortress es una plataforma de pruebas y monitorización de API (Application Programming Interface) que se utiliza para garantizar la calidad, rendimiento y confiabilidad de las API. Proporciona una serie de funcionalidades y características que facilitan el proceso de pruebas y monitoreo de las API.

API Fortress se utiliza para realizar pruebas funcionales y de rendimiento en API. Algunas de sus características principales son:

Creación de pruebas: crear pruebas funcionales para verificar que las API se comporten según lo esperado. Se pueden configurar diferentes casos de prueba para validar el funcionamiento de las rutas, los parámetros, las respuestas y las validaciones

de datos.

Automatización de pruebas: permite la automatización de pruebas mediante la creación de flujos de trabajo. Estos flujos de trabajo pueden incluir diferentes pruebas, así como acciones adicionales como la generación de datos de prueba, el manejo de variables y la ejecución programada de las pruebas.

Monitorización de API: proporciona capacidades de monitorización en tiempo real de las API. Permite definir umbrales y alertas para detectar cualquier anomalía o error en el rendimiento de las API. Esto ayuda a identificar problemas y realizar un seguimiento proactivo de la salud de las API.

Generación de informes: ofrece la generación de informes detallados sobre los resultados de las pruebas y el rendimiento de las API. Estos informes pueden incluir métricas como el tiempo de respuesta, la tasa de éxito, los errores detectados y otras estadísticas relevantes.

Integraciones y colaboración: se integra con otras herramientas y sistemas, como sistemas de gestión de incidencias, sistemas de control de versiones y herramientas de CI/CD (Continuous Integration/Continuous Deployment). También permite la colaboración en equipo, lo que facilita la comunicación y el intercambio de información entre los miembros del equipo de desarrollo y pruebas.

3.3.1.3. Assertible

Assertible es una herramienta de pruebas de API y monitoreo de servicios web. Se utiliza para automatizar pruebas funcionales, realizar pruebas de integración y asegurar la calidad de las API.

La funcionalidad principal de Assertible se centra en la ejecución de pruebas automatizadas para verificar que las API se comporten correctamente. Permite definir

casos de prueba con diferentes escenarios y realizar solicitudes HTTP para validar el comportamiento de las API, algunas de las características y usos clave de Assertible son:

Pruebas de respuesta: verificar y validar las respuestas de las API, asegurándose de que los datos devueltos sean los esperados. Puede realizar comparaciones de cuerpos de respuesta, encabezados, códigos de estado y otros atributos relevantes.

Pruebas de integración: permite realizar pruebas de integración entre diferentes servicios web y API. Esto es útil para garantizar que las integraciones entre sistemas funcionen correctamente y cumplan con los requisitos establecidos.

Monitoreo de servicios: ofrece monitoreo continuo de servicios web y API. Puede configurar reglas y alertas para detectar cualquier anomalía en el rendimiento de las API, como tiempos de respuesta lentos o errores frecuentes.

Integraciones y automatización: Assertible se integra con herramientas de CI/CD y sistemas de integración continua, lo que permite ejecutar pruebas automáticamente como parte del proceso de desarrollo. También se puede utilizar en entornos de integración y entrega continua para validar la calidad de las API en cada despliegue.

Generación de informes: genera informes detallados sobre los resultados de las pruebas, incluyendo métricas de rendimiento, éxitos y fallas, y otros datos relevantes. Estos informes son útiles para el análisis y seguimiento de la calidad de las API a lo largo del tiempo.

4. Metodología de la investigación

En la ejecución del proyecto, se tomó la decisión de no emplear la metodología Scrum, a pesar de su reconocida eficacia en la gestión de proyectos. Esta elección se basó en consideraciones específicas relacionadas con la naturaleza del proyecto.

En primer lugar, Scrum se centra en la colaboración en equipo, con roles definidos como Scrum Master, Product Owner y miembros del equipo de desarrollo. En este caso, al tratarse de un proyecto individual, no existía un equipo con el que colaborar ni roles que asignar, lo que hizo innecesario el uso de Scrum.

Además, Scrum se adapta mejor a proyectos de mayor complejidad, con múltiples entregas iterativas y un flujo constante de funcionalidades. En cambio, este proyecto tenía una naturaleza más simple y no requería entregas iterativas regulares. El enfoque en sprints y entregas continuas no se ajustaba a las necesidades específicas del proyecto, la cual se centraba más en la definición y el logro de objetivos concretos, por lo tanto se llevo a cabo una metodología por objetivos seccionadas por etapas la cual se ajustaba al proyecto.

Con el fin de dar cumplimiento al objetivo de diseñar e implementar un prototipo de un servicio web que permita realizar comparaciones de comportamientos basados en respuestas REST entre diferentes ambientes, para identificar y mostrar discrepancias, se están llevando a cabo las siguientes etapas:

1. Se llevó a cabo un análisis para documentar las consecuencias y repercusiones que los errores pueden tener en el contexto de un servicio REST. El propósito principal de esta etapa fue profundizar en la comprensión de cómo los errores pueden impactar significativamente el rendimiento del sistema, la calidad del software y la experiencia global del usuario.

    - Investigar diferentes fuentes.
    - Realizar un escrito de las consecuencias y repercusiones 5.2.

2. Diseñar y desarrollar un servicio web. Este servicio tiene la capacidad de comparar las respuestas generadas por dos servicios REST distintos. El objetivo principal aquí fue proporcionar una herramienta que identificará y presentará las diferencias entre las respuestas de estos servicios, lo que resultaría fundamental para la detección temprana de posibles errores lógicos del negocio o errores del sistema.

    - Diseñar los componentes del prototipo: para el diseño de componentes del prototipo se realizó el uso de la práctica attribute domain design 7.2.2.
    - Desarrollar con lenguaje Golang el servicio web 7.2.2.

3. En la tercera etapa, se trabajó en la mejora y expansión del servicio web desarrollado previamente. Se agregó una funcionalidad crucial de generación de informes que permitiría registrar y reportar las diferencias encontradas en las respuestas de los servicios REST. Este paso fue esencial para facilitar la monitorización y la gestión efectiva de problemas.

    - Diseñar el modelo del informe mediante gráficas.
    - Desarrollar el aprovisionamiento de la información en New Relic 5.6.

4. Implementación del comparador en uno de los servicios críticos de la organización 99minutos, específicamente el servicio de coberturas, conocido como Coverage Microservice. Este análisis detallado contribuyó a una comprensión más profunda de su funcionamiento interno y permitió identificar áreas específicas que podrían beneficiarse de mejoras y optimizaciones.

    - Implementar el comparador 5.8.
    - Recopilar resultados. 5.9

Esta estructura por etapas permitirá que cada fase del proyecto se enfocara en tareas y metas específicas, proporcionando un enfoque más claro y efectivo para abordar los desafíos y alcanzar los resultados deseados.

5. Desarrollo del proyecto

Para ofrecer una visión más detallada de la arquitectura previa a la implementación del nuevo componente, se presenta una representación visual en la figura. Esta representación gráfica destaca los diferentes módulos y su interconexión, permitiendo una comprensión más completa de la infraestructura subyacente 4.

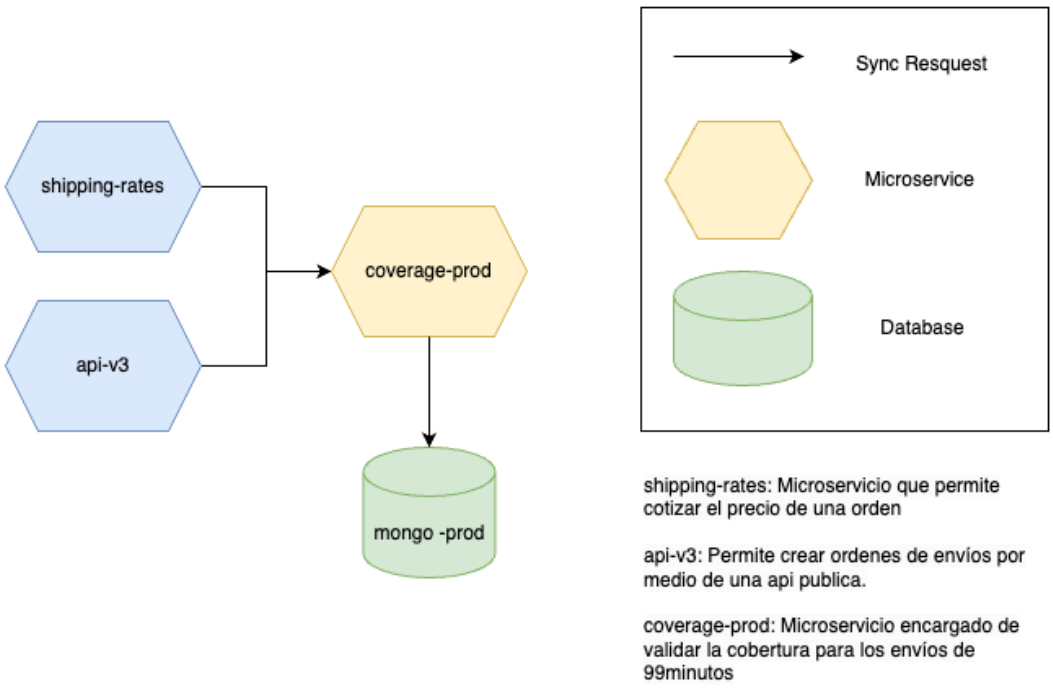


Figura 4: Arquitectura actual antes de la implementación.

En la figura 5, se destaca cómo el nuevo componente interactúa con los módulos preexistentes, aportando una capa adicional de funcionalidad de forma asíncrona. Su integración se ha diseñado para ser coherente con la arquitectura general, minimizando las interrupciones.

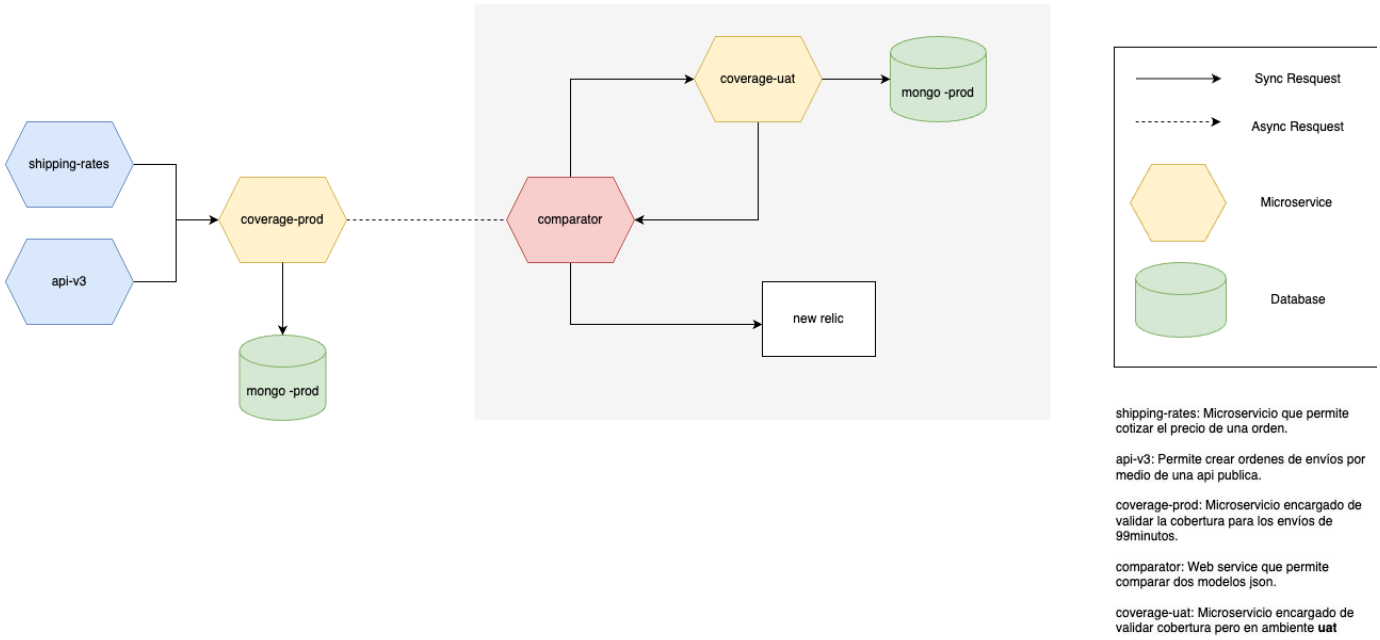


Figura 5: Arquitectura con la implementación del comparador.

5.1. Attribute Domain Design

Attribute Domain Design (ADD) es una metodología de diseño de software que se centra en la definición y estructuración de los dominios de atributos de un sistema Bass et al. (2012) . En esencia, ADD se enfoca en cómo los atributos de un sistema, como las variables, propiedades y características, se organizan y relacionan entre sí para lograr los objetivos del sistema. A medida que los sistemas de software se vuelven cada vez más complejos y extensos, la gestión y el diseño efectivos de los atributos se vuelven cruciales para garantizar la calidad y el rendimiento del software.

ADD se basa en varios principios fundamentales que guían su enfoque:

Organización de atributos: el primer paso en ADD implica identificar y organizar todos los atributos relevantes de un sistema. Esto incluye tanto atributos funcio-

nales como no funcionales. Los atributos se agrupan y se definen claramente para comprender su papel en el sistema.

Jerarquía de atributos: los atributos se organizan en una jerarquía que refleja su importancia y relaciones Pfleeger and Atlee (2016). Esto ayuda a comprender cómo los atributos de nivel superior afectan a los atributos de nivel inferior y cómo se propagan los cambios.

Visualización de atributos: la visualización juega un papel crucial en ADD. Los diagramas y representaciones visuales ayudan a los diseñadores y desarrolladores a comprender la estructura de atributos de un sistema y cómo se interconectan.

Modelado de dominios de atributos: ADD utiliza técnicas de modelado para representar los dominios de atributos. Esto incluye diagramas de clases, diagramas de flujo de datos y otros modelos que describen la relación entre los atributos Society (2004).

Refinamiento de atributos: a medida que se avanza en el diseño del software, los atributos se refinan y se detallan más. Se agregan restricciones y especificaciones adicionales para garantizar que los atributos cumplan con los requisitos del sistema.

Gestión de cambios de atributos: ADD también se ocupa de cómo los cambios en los atributos afectan al sistema en su conjunto. Los diseñadores deben considerar cómo los cambios en un atributo pueden tener un impacto en otros atributos y en el sistema en general.

La importancia de ADD radica en su capacidad para abordar la complejidad en el diseño de software Bass et al. (2012). A medida que los sistemas crecen, la gestión de atributos se vuelve crucial para garantizar la modularidad, la escalabilidad y la mantenibilidad del software. Además, al organizar y estructurar los atributos de manera efectiva, los equipos de desarrollo pueden tomar decisiones informadas sobre la priorización y asignación de recursos.

Un ejemplo de la aplicación de ADD podría ser en el diseño de un sistema de gestión de pedidos en línea. Los atributos en este contexto podrían incluir la velocidad de procesamiento de pedidos, la seguridad de los datos del cliente, la escalabilidad del sistema y la facilidad de uso. Aplicar ADD permitiría definir y organizar estos atributos de manera que se puedan abordar adecuadamente durante el desarrollo del sistema.

ADD es una metodología valiosa para el diseño de software en sistemas complejos. Al proporcionar una estructura y un enfoque para la gestión de atributos, ayuda a garantizar que el software cumpla con los requisitos y expectativas del cliente, al tiempo que se mantiene una alta calidad y un rendimiento óptimo. La aplicación de ADD requiere experiencia en diseño de software y un enfoque metódico para la identificación y organización de atributos.

En este proyecto, se aplicó la metodología mencionada anteriormente, la cual se detalla minuciosamente en 7.2.2

5.2. Identificación de Consecuencias y Repercusiones de Errores

El desarrollo de software es un proceso complejo que implica una serie de desafíos técnicos y de gestión McConnell (2004); Pressman (2010). Uno de los aspectos más críticos en este proceso es la identificación de errores y su posterior manejo, ya que los errores en el software son inevitables, pero su detección y corrección efectivas son fundamentales para garantizar la calidad y confiabilidad del producto final IEEE Computer Society (2014).

Uno de los impactos más inmediatos de los errores en el desarrollo de software son los retrasos en el proyecto, los cuales pueden surgir en cualquier etapa del ciclo de vida del desarrollo, desde la definición de requisitos hasta la implementación y las pruebas finales Sommerville (2016). La identificación y corrección de errores pueden requerir tiempo adicional, lo que a menudo resulta en la postergación de los plazos previstos.

La corrección de errores suele ser costosa, implicando dedicar recursos adicionales, tanto en términos de horas de trabajo como de recursos técnicos. Además, si los errores no se detectan a tiempo y llegan a producción, los costos asociados con su corrección aumentan significativamente Pressman (2010).

Los errores en el software también pueden tener un impacto directo en la experiencia del usuario final. Los usuarios esperan que las aplicaciones sean confiables y funcionen como se espera, por lo que problemas recurrentes pueden disminuir su satisfacción y llevar a la pérdida de clientes McConnell (2004). La reputación de una empresa, siendo uno de sus activos más valiosos, puede socavarse con errores recurrentes, llevando a los clientes a buscar alternativas si perciben que la empresa no puede ofrecer productos o servicios de alta calidad Sommerville (2016).

Algunos errores en el software también pueden exponer vulnerabilidades de seguridad, lo que tiene graves implicaciones legales y financieras. La seguridad del software es una preocupación crítica, especialmente en aplicaciones que manejan información sensible. Los errores en la producción pueden resultar en la pérdida de datos críticos, con consecuencias graves para las empresas y los usuarios. La recuperación de datos perdidos puede ser costosa y, en algunos casos, imposible Pressman (2010).

Los errores en sistemas en producción pueden causar tiempo de inactividad no planificado, afectando la productividad de las empresas y resultando en pérdida de ingresos. Los clientes que no pueden acceder a servicios o aplicaciones pueden buscar soluciones alternativas.

La reputación de una empresa puede verse afectada negativamente por errores en producción, ya que los clientes que experimentan problemas graves pueden compartir sus experiencias en línea y a través de las redes sociales, amplificando el impacto negativo en la marca. La corrección de errores en producción puede ser costosa y urgente, requiriendo que los equipos de desarrollo y operaciones trabajen juntos para solucionar los problemas lo más rápido posible. Esto puede implicar el desarrollo e implementación de parches y actualizaciones de emergencia.

Para abordar eficazmente la identificación y mitigación de consecuencias y repercusiones de errores en el desarrollo de software, es fundamental contar con procesos sólidos de pruebas, revisiones de código y monitoreo constante en producción IEEE Computer Society (2014).

5.3. Casos de pruebas

La funcionalidad a probar es la de comparar dos cuerpos JSON, es decir uno corresponde al cuerpo de respuesta de un servicio de producción, mientras que el otro es la respuesta del servicio que se está probando. El comparador tiene la tarea de analizar y comparar ambas respuestas JSON. Se debe verificar que sea capaz de detectar diferencias con distintos tipos de datos, como enteros, decimales, cadenas de texto, booleanos y estructuras dinámicas, incluyendo la eliminación de atributos y la incorporación de nuevos atributos. La estrategia general implementada en las pruebas del proyecto se puede detallar en 7.2.2.

La cobertura de sentencia se refiere a asegurarse de que todas las sentencias del código hayan sido ejecutadas al menos una vez durante las pruebas. El código que permitirá realizar las comparaciones entre dos objetos contiene las siguientes sentencias:

1. `differ := diff.New()`
2. `diff := differ.CompareObjects(comparator.ResponseC, result)`
3. `f := format.NewAsciiFormatter(result,format.AsciiFormatterDefaultConfig)`
4. `deltaJSON, _ := f.Format(diff)`
5. `return diff, deltaJSON`

Para lograr una cobertura de sentencia completa, necesitaríamos ejecutar una prueba que pase por todas las sentencias anteriores 6. Dado que todas las sentencias

son líneas consecutivas y no hay bifurcaciones ni bucles, una sola ejecución de esta función debería cubrir todas las sentencias.

La cobertura de decisión se refiere a asegurarse de que todas las decisiones (ramas de código) hayan sido evaluadas tanto como verdaderas como falsas durante las pruebas.

En este código, no hay decisiones lógicas directas que se evalúen (como condicionales if o bucles for). Las decisiones podrían encontrarse en las implementaciones de las funciones diff.New() y f.Format(diff).

```go
func TestObjects(t *testing.T) {

    tests := []struct {
        name       string
        argsResult map[string]interface{}
        argsIn     dto.ComparatorIn
        want       string
    }{
        {
            name:       "Identical",
            argsResult: DataInJson(),
            argsIn:     dto.ComparatorIn{ResponseC: DataInJson()},
            want:       diff_string_answer,
        },
        {
            name:       "Diff string",
            argsResult: DataInJson(),
            argsIn:     dto.ComparatorIn{ResponseC: DataStringJson()},
            want:       identical_answer,
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            _, deltaJson := DiffJson(tt.argsResult, tt.argsIn)
            assert.Equal(t, deltaJson, tt.want)
        })
    }
}

func DataInJson() map[string]interface{} {
    var data map[string]interface{}

    if err := json.Unmarshal(jsonData, &data); err != nil {
        fmt.Println("Error al convertir JSON:", err)
        return nil
    }

    return data
```

Figura 6: Test.

Resultado de la prueba unitaria, dando una cobertura del 100 en 0.6s para una de las fuciones más críticas

```
ok      ws_comparator/infrastructure/diff_json  0.691s  coverage: 100.0% of statements
```

Figura 7: Resultado Test.

Para las pruebas de caja negra se implementa la técnica de tabla de decisión, se diseña los siguientes escenarios:

Objeto A	Objeto B	Resultado Esperado
Entero	Entero	No hay diferencias
Entero -33	Entero -30	Diferencia en la comparación
Entero -33	Cadena “Test”	Diferencia en la comparación
Entero -33	Booleano true	Diferencia en la comparación
Cadena	Cadena	No hay diferencias

Reporte de los resultados de la ejecución de los casos de prueba

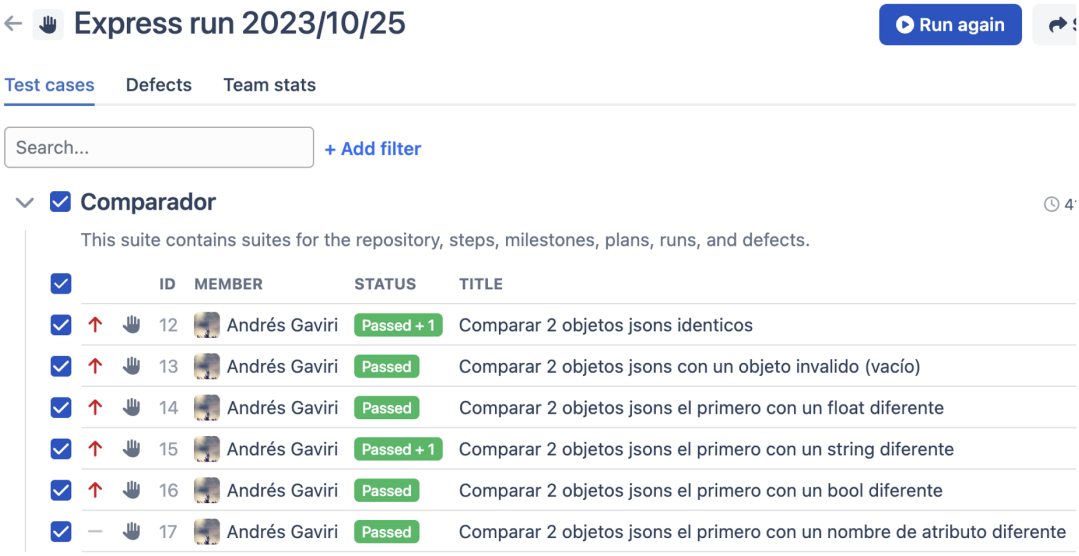


Figura 8: Ejecución de Test.

A continuación, se diseñan estrategias de pruebas para este desarrollo, definiendo las tecnologías a utilizar para cada tipo de prueba y los diversos tipos de pruebas seleccionados específicamente para este proyecto.

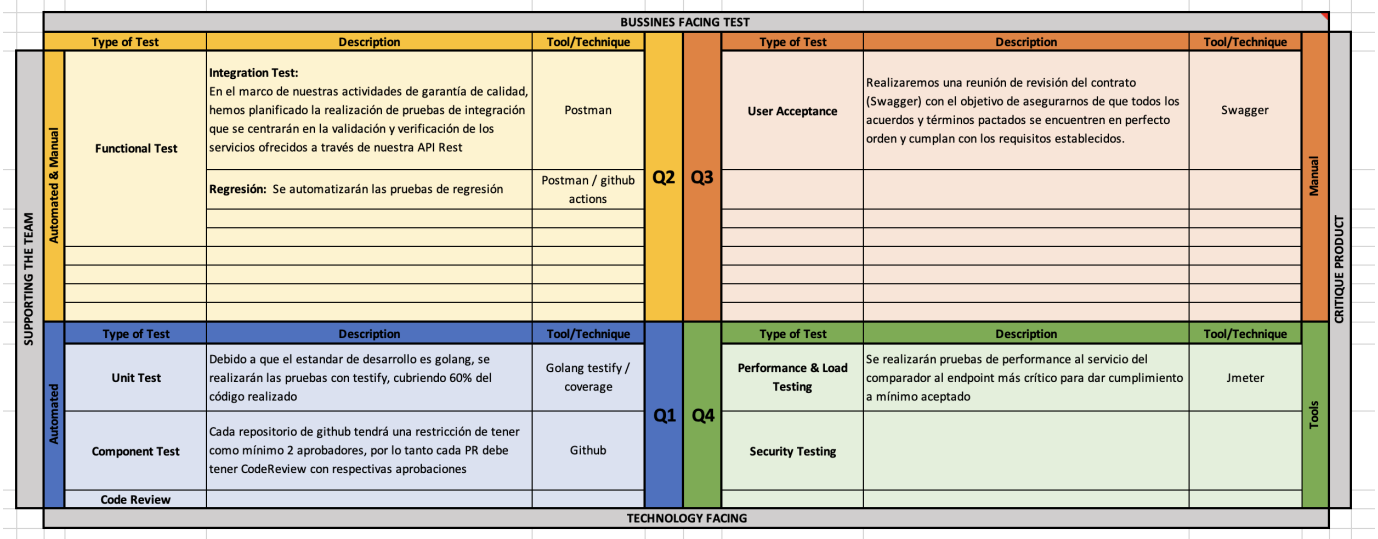
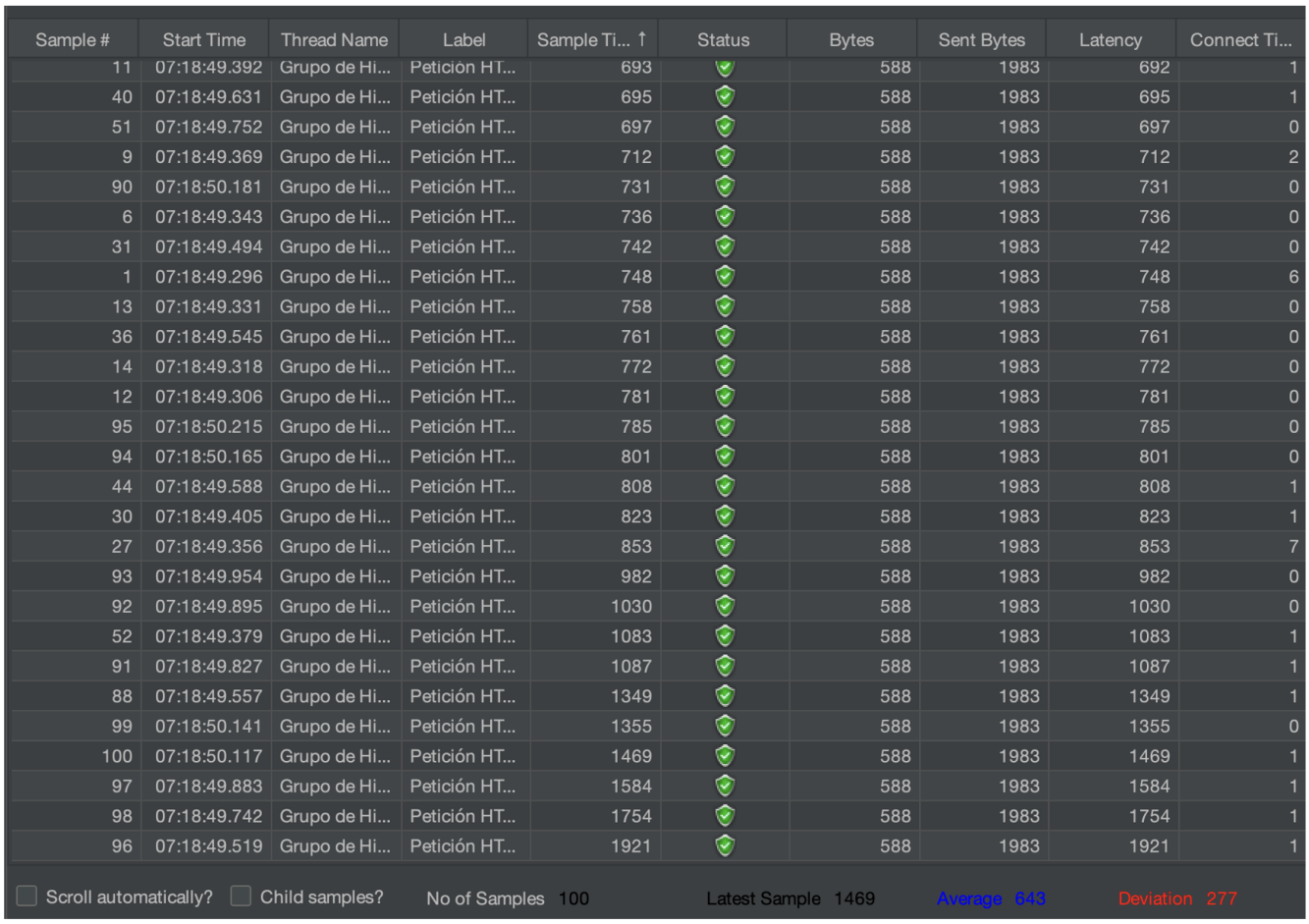


Figura 9: Test.

Se llevaron a cabo pruebas de rendimiento, comenzando con la simulación de 100 usuarios concurrentes mediante JMeter, seguido por una prueba con 1,000 usuarios. A continuación, se presentan los resultados, teniendo en cuenta métricas como la CPU, la memoria y los tiempos de respuesta.

N	Endpoint	Number of Threads (users)	Ramp-up period (seconds)	Loop Count	Method	CPU (%) Initial	Memory (MB) Initial	CPU (%) Process	Memory (MB) Process	Result Tree Average (ms)
1	/v1/request/comparation	100	1	1	POST	0	0	5,9	48	643
2	/v1/request/comparation	1000	4	1	POST	0	0	17,8	153,8	599
							Maximum	17,8	153,8	643
							Minimum	5,9	48	599
							Average	11,9	100,9	621

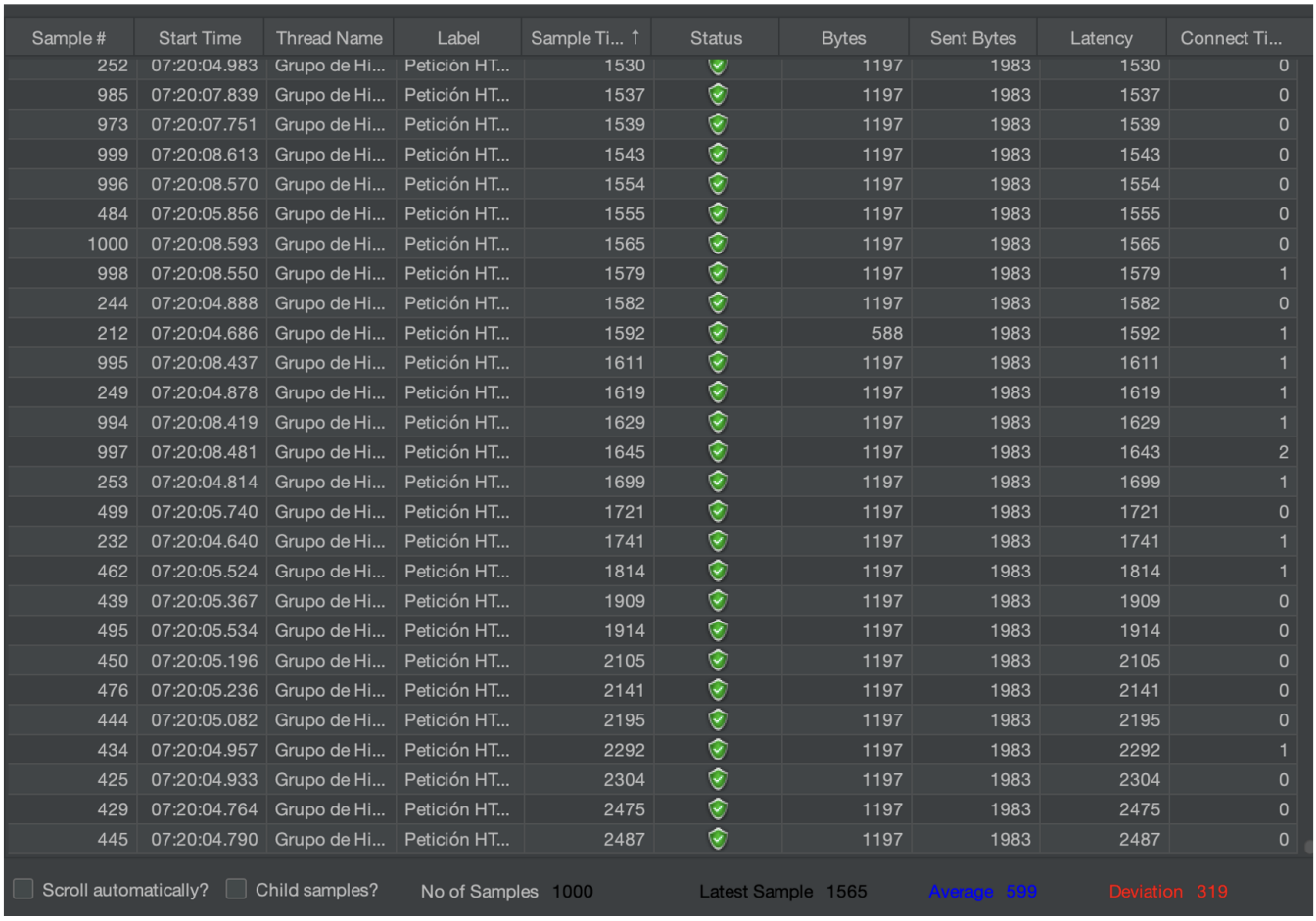
Figura 10: Resultado test de rendimiento.



Sample #	Start Time	Thread Name	Label	Sample Ti... ↑	Status	Bytes	Sent Bytes	Latency	Connect Ti...
11	07:18:49.392	Grupo de Hi...	Petición HT...	693		588	1983	692	1
40	07:18:49.631	Grupo de Hi...	Petición HT...	695		588	1983	695	1
51	07:18:49.752	Grupo de Hi...	Petición HT...	697		588	1983	697	0
9	07:18:49.369	Grupo de Hi...	Petición HT...	712		588	1983	712	2
90	07:18:50.181	Grupo de Hi...	Petición HT...	731		588	1983	731	0
6	07:18:49.343	Grupo de Hi...	Petición HT...	736		588	1983	736	0
31	07:18:49.494	Grupo de Hi...	Petición HT...	742		588	1983	742	0
1	07:18:49.296	Grupo de Hi...	Petición HT...	748		588	1983	748	6
13	07:18:49.331	Grupo de Hi...	Petición HT...	758		588	1983	758	0
36	07:18:49.545	Grupo de Hi...	Petición HT...	761		588	1983	761	0
14	07:18:49.318	Grupo de Hi...	Petición HT...	772		588	1983	772	0
12	07:18:49.306	Grupo de Hi...	Petición HT...	781		588	1983	781	0
95	07:18:50.215	Grupo de Hi...	Petición HT...	785		588	1983	785	0
94	07:18:50.165	Grupo de Hi...	Petición HT...	801		588	1983	801	0
44	07:18:49.588	Grupo de Hi...	Petición HT...	808		588	1983	808	1
30	07:18:49.405	Grupo de Hi...	Petición HT...	823		588	1983	823	1
27	07:18:49.356	Grupo de Hi...	Petición HT...	853		588	1983	853	7
93	07:18:49.954	Grupo de Hi...	Petición HT...	982		588	1983	982	0
92	07:18:49.895	Grupo de Hi...	Petición HT...	1030		588	1983	1030	0
52	07:18:49.379	Grupo de Hi...	Petición HT...	1083		588	1983	1083	1
91	07:18:49.827	Grupo de Hi...	Petición HT...	1087		588	1983	1087	1
88	07:18:49.557	Grupo de Hi...	Petición HT...	1349		588	1983	1349	1
99	07:18:50.141	Grupo de Hi...	Petición HT...	1355		588	1983	1355	0
100	07:18:50.117	Grupo de Hi...	Petición HT...	1469		588	1983	1469	1
97	07:18:49.883	Grupo de Hi...	Petición HT...	1584		588	1983	1584	1
98	07:18:49.742	Grupo de Hi...	Petición HT...	1754		588	1983	1754	1
96	07:18:49.519	Grupo de Hi...	Petición HT...	1921		588	1983	1921	1

Scroll automatically? Child samples? No of Samples 100 Latest Sample 1469 Average 643 Deviation 277

Figura 11: 100 peticiones concurrentes.



Sample #	Start Time	Thread Name	Label	Sample Ti... ↑	Status	Bytes	Sent Bytes	Latency	Connect Ti...
252	07:20:04.983	Grupo de Hi...	Petición HT...	1530		1197	1983	1530	0
985	07:20:07.839	Grupo de Hi...	Petición HT...	1537		1197	1983	1537	0
973	07:20:07.751	Grupo de Hi...	Petición HT...	1539		1197	1983	1539	0
999	07:20:08.613	Grupo de Hi...	Petición HT...	1543		1197	1983	1543	0
996	07:20:08.570	Grupo de Hi...	Petición HT...	1554		1197	1983	1554	0
484	07:20:05.856	Grupo de Hi...	Petición HT...	1555		1197	1983	1555	0
1000	07:20:08.593	Grupo de Hi...	Petición HT...	1565		1197	1983	1565	0
998	07:20:08.550	Grupo de Hi...	Petición HT...	1579		1197	1983	1579	1
244	07:20:04.888	Grupo de Hi...	Petición HT...	1582		1197	1983	1582	0
212	07:20:04.686	Grupo de Hi...	Petición HT...	1592		588	1983	1592	1
995	07:20:08.437	Grupo de Hi...	Petición HT...	1611		1197	1983	1611	1
249	07:20:04.878	Grupo de Hi...	Petición HT...	1619		1197	1983	1619	1
994	07:20:08.419	Grupo de Hi...	Petición HT...	1629		1197	1983	1629	1
997	07:20:08.481	Grupo de Hi...	Petición HT...	1645		1197	1983	1643	2
253	07:20:04.814	Grupo de Hi...	Petición HT...	1699		1197	1983	1699	1
499	07:20:05.740	Grupo de Hi...	Petición HT...	1721		1197	1983	1721	0
232	07:20:04.640	Grupo de Hi...	Petición HT...	1741		1197	1983	1741	1
462	07:20:05.524	Grupo de Hi...	Petición HT...	1814		1197	1983	1814	1
439	07:20:05.367	Grupo de Hi...	Petición HT...	1909		1197	1983	1909	0
495	07:20:05.534	Grupo de Hi...	Petición HT...	1914		1197	1983	1914	0
450	07:20:05.196	Grupo de Hi...	Petición HT...	2105		1197	1983	2105	0
476	07:20:05.236	Grupo de Hi...	Petición HT...	2141		1197	1983	2141	0
444	07:20:05.082	Grupo de Hi...	Petición HT...	2195		1197	1983	2195	0
434	07:20:04.957	Grupo de Hi...	Petición HT...	2292		1197	1983	2292	1
425	07:20:04.933	Grupo de Hi...	Petición HT...	2304		1197	1983	2304	0
429	07:20:04.764	Grupo de Hi...	Petición HT...	2475		1197	1983	2475	0
445	07:20:04.790	Grupo de Hi...	Petición HT...	2487		1197	1983	2487	0

Scroll automatically? Child samples? No of Samples 1000 Latest Sample 1565 Average 599 Deviation 319

Figura 12: 1.000 peticiones concurrentes.

5.4. Pruebas de rendimiento

JMeter es una aplicación de código abierto diseñada para realizar pruebas de carga y medir el rendimiento de aplicaciones web y otras funciones de prueba. JMeter permite simular cargas pesadas en servidores, probar su resistencia o analizar su rendimiento bajo diferentes tipos de carga. Sus características incluyen la capacidad de probar diversas aplicaciones y protocolos, un entorno de desarrollo integrado (IDE) para grabar y depurar pruebas, un modo de línea de comandos (CLI) para realizar

pruebas desde cualquier sistema operativo compatible con Java, extracción de datos de formatos de respuesta populares y complementos para el análisis y visualización de datos Halili (2000).

Para llevar a cabo las pruebas de rendimiento, se empleó Apache JMeter, una herramienta ampliamente utilizada en la industria para evaluar el rendimiento de aplicaciones web.

En relación a las necesidades específicas planteadas, se establecieron los siguientes requerimientos:

Se requirió que el endpoint objetivo fuera capaz de manejar un mínimo de 100 peticiones concurrentes en un lapso de 1 segundo. Esta métrica es fundamental para evaluar la capacidad de respuesta del sistema bajo una carga considerable.

Además, se solicitó que el endpoint pudiera soportar 1.000 peticiones concurrentes en un periodo de 4 segundos. Este escenario permite simular una carga más intensa y poner a prueba la estabilidad y eficiencia del sistema en situaciones de alto tráfico.

Gracias a la versatilidad y facilidad de uso de Apache JMeter, fue posible llevar a cabo estas pruebas de rendimiento de manera efectiva, obteniendo valiosa información 7.2.2 sobre el comportamiento y rendimiento del endpoint en cuestión.

5.5. Diagrama de flujo de trabajo

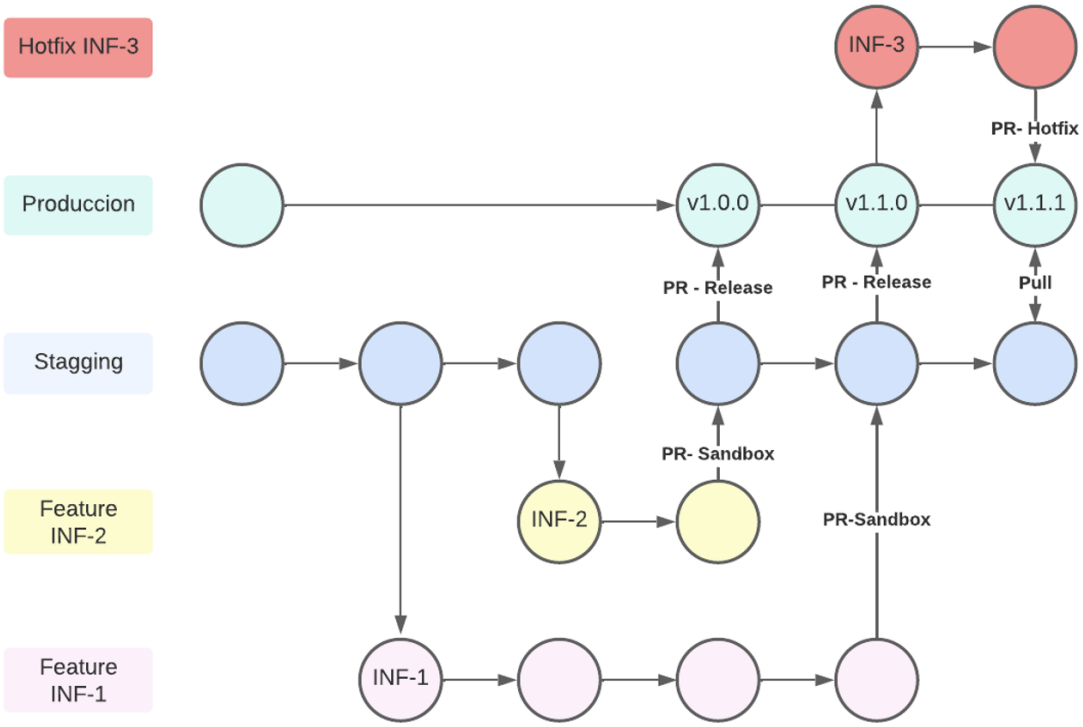


Figura 13: Flujo de trabajo.

En la figura 13, se presenta el flujo de desarrollo, estructurado en dos ramas principales: "producciónz "staging". Estas ramas facilitan el despliegue en la nube de las respectivas versiones. Cada vez que se trabaja en el desarrollo de una nueva funcionalidad en el proyecto del comparador, se crea una rama nueva identificada por ÏNF"seguido del número identificador del feature.

Una vez que el desarrollo de la funcionalidad se completa, se abre un "pull request"(PR) hacia la rama "staging". Después de que el PR es fusionado con éxito en "stagingz se ha probado satisfactoriamente, se abre un nuevo PR hacia la rama "producción"para lanzar la versión productiva correspondiente.

En caso de identificar un bug, se procede a abrir una rama "hotfix"para abordar y corregir el error. Posteriormente, se fusiona la rama de hotfix con la rama de "producción", asegurando la estabilidad y correcto funcionamiento de la versión en producción. Este flujo establece un proceso estructurado que permite el desarrollo progresivo y controlado, minimizando riesgos y asegurando la calidad del software.

5.6. Gráficas de informe

Con el fin de evidenciar y proporcionar una visión más detallada de las discrepancias, se realizó la implementación de gráficas en New Relic. Estas gráficas están diseñadas para proyectar la categorización de las diferencias en las peticiones.

Las gráficas ofrecen una perspectiva integral, incluyendo:

1. **Número total de peticiones:** Para identificar el volumen total de solicitudes procesadas.
2. **Peticiones con discrepancias:** Destacamos aquellas peticiones que presentan diferencias discernibles.
3. **Peticiones sin discrepancias:** Visualizamos la cantidad de peticiones que se ejecutan sin encontrar discrepancias.

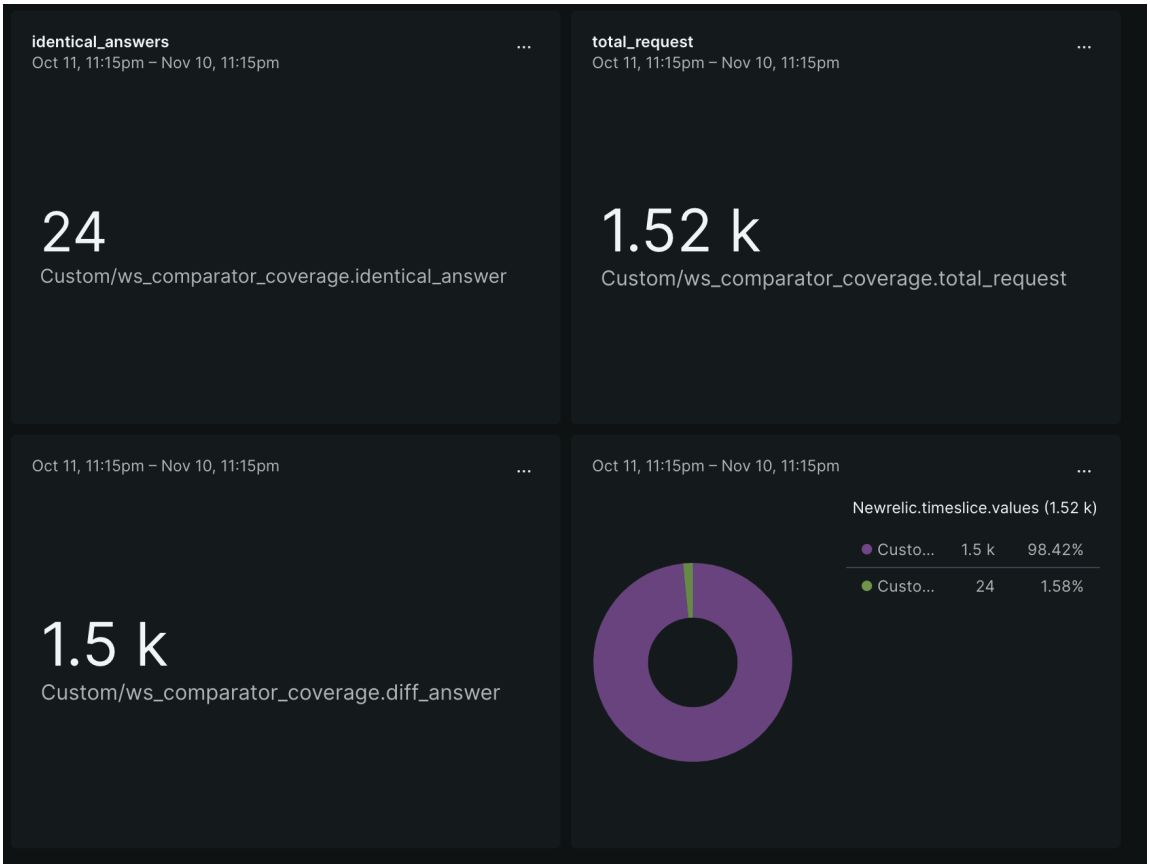


Figura 14: Gráficas.

Esta visualización enriquecida de new relic nos permite analizar rápidamente el estado general de las operaciones y entender la prevalencia de discrepancias en comparación con el total de peticiones.

Se pueden observar el total de peticiones comparadas, la cantidad de respuestas idénticas y el número de peticiones con diferencias.

Además, se incorporó la capacidad de acceder directamente al JSON asociado a las diferencias detectadas. Esto facilita la identificación y resolución de discrepancias específicas al proporcionar una visión detallada de los datos en cuestión.

Estas gráficas simplificarán el proceso de análisis y toma de decisiones.

5.7. Estrategia de goteo

La estrategia de goteo o también conocido como sampling, también conocida como çanary deployment.º rolling release", es una metodología de despliegue de software que busca minimizar los riesgos asociados con cambios significativos en un sistema. Esta estrategia se ha vuelto fundamental en el desarrollo de software moderno, permitiendo una implementación gradual y controlada de nuevas características o actualizaciones.

Principios Fundamentales de la Estrategia de Goteo:

1. **Implementación gradual:** La estrategia de goteo implica liberar cambios en pequeñas porciones del entorno de producción en lugar de realizar una implementación masiva.
2. **Monitoreo constante:** Se centra en el monitoreo constante del rendimiento, estabilidad y comportamiento del sistema a medida que se introduce progresivamente el cambio.
3. **Reversión segura:** Permite revertir rápidamente los cambios si se detectan problemas, minimizando el impacto en los usuarios finales.

Implementación en el Comparador:

En la implementación del comparador, se adoptó una estrategia de goteo adaptada para procesar un porcentaje específico del tráfico entrante. Este enfoque permitió:

- **Control Gradual del Cambio:** Al procesar solo un porcentaje del tráfico, se controló gradualmente la exposición a la nueva lógica implementada en el comparador.

- **Identificación de problemas específicos:** Al procesar un subconjunto del tráfico, se facilitó la identificación de problemas específicos que podrían surgir con la nueva implementación.
- **Menor uso de la base de datos productiva:** Al procesar solo una fracción del tráfico, se redujo la dependencia directa de la base de datos productiva, disminuyendo la carga y permitiendo escalar de manera más eficiente y hasta sin necesidad de escalarla.

Esta estrategia de goteo en el comparador refleja una cuidadosa consideración para garantizar una transición suave y minimizar los riesgos asociados con cambios sustanciales en el servicio y así como estrategia de reducción de afectación del rendimiento de la base de datos productiva.

5.8. Despliegue del comparador en ambiente staging y producción

La implementación y gestión eficiente del servicio comparador se llevó a cabo mediante las herramientas y servicios proporcionados por Google Cloud Platform (GCP). A continuación, se describen detalladamente los pasos realizados de la implementación continua, la integración con SonarCloud para el análisis estático del código, y la implementación en los entornos de staging y producción.

Google Cloud Run se utilizó para la implementación sin problemas y escalable del servicio comparador. Este servicio basado en contenedores permitió la ejecución rápida y eficiente de la aplicación en un entorno completamente gestionado.

Se implementó un flujo de trabajo utilizando GitHub Actions para permitir la implementación automática con cada actualización del repositorio de código. Esta automatización garantizó eficiencia en el ciclo de vida del desarrollo y una implementación consistente y libre de errores.

SonarCloud se integró en el repositorio para realizar un análisis estático del código. Esta herramienta facilitó la identificación y resolución de posibles problemas de calidad de código, mejorando así la solidez y mantenibilidad de la aplicación.

Las implementaciones se llevaron a cabo en dos entornos distintos: staging y producción. El entorno de staging proporcionó una réplica controlada de la aplicación, permitiendo pruebas extensas antes de cualquier implementación en producción. Esto aseguró que el servicio estuviera libre de errores y cumpliera con los requisitos establecidos antes de implementar en ambiente productivo.

Después de una validación exitosa en el entorno de staging, la implementación continuó en producción. Esta fase se llevó a cabo para garantizar la disponibilidad y confiabilidad del servicio en un entorno de producción.

Este proceso completo de implementación en GCP, junto con la integración de herramientas como GitHub Actions y SonarCloud, fue fundamental para lograr una implementación continua y eficiente del servicio comparador. Garantizó calidad y confiabilidad en cada etapa del desarrollo.

5.9. Resultados de implementación

Se llevó a cabo una refactorización del servicio de coverage con el objetivo de aplicar buenas prácticas a nivel de código y asegurar la mantenibilidad del servicio a lo largo del tiempo. Para implementar estos cambios, se desplegó una nueva versión de coverage en la base de datos de producción utilizando el flujo de trabajo de GitActions, como se puede observar en la Figura 15.

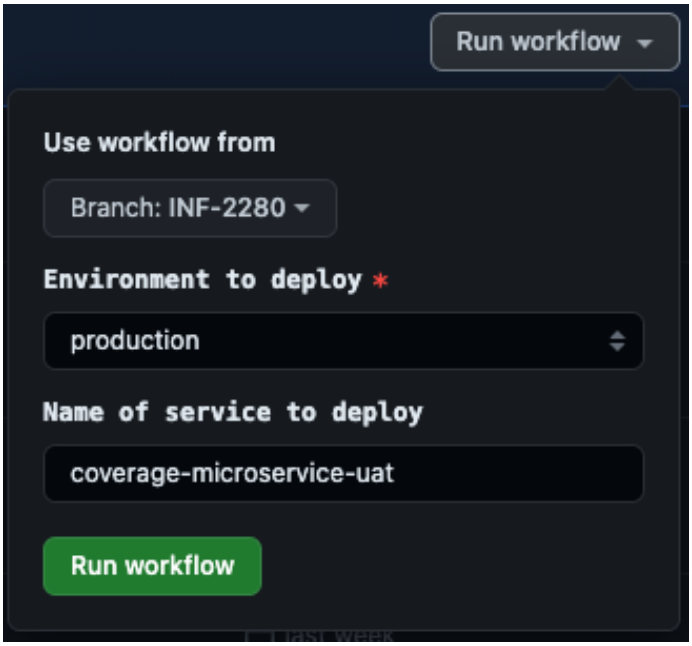


Figura 15: GitActions en acción.

Con el propósito de garantizar que la refactorización no afectara la lógica de negocio existente en coverage, se activó un comparador. Este comparador fue utilizado para identificar posibles diferencias entre la versión anterior y la versión refactorizada del servicio. Para realizar esta evaluación de manera efectiva, se escaló la base de datos de MongoDB de M30 a M40 para gestionar la nueva carga sin afectar el tráfico del producto.

Posteriormente, se implementó el comparador en producción, capturando los datos de entrada y salida en formato JSON. Estos datos se enviaron a una goroutine en Go, la cual ejecutó el comparador para identificar y reportar cualquier diferencia. Este proceso se registró en New Relic, como se muestra en la 17. El comparador estuvo activo durante 1 hora y 40 minutos, gestionando alrededor de 5.34k solicitudes.

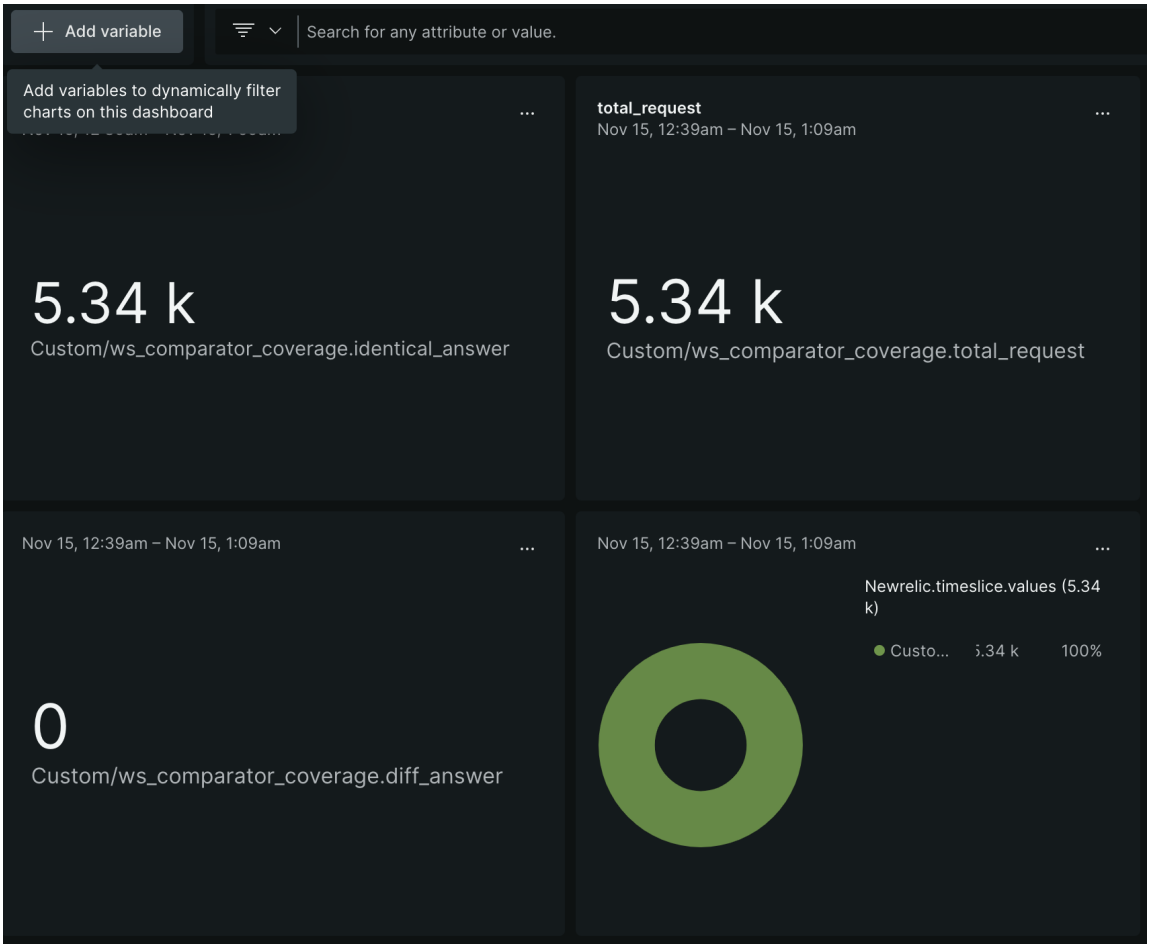


Figura 16: Resultados NewRelic.

El resultado de esta implementación fue satisfactorio, proporcionando tranquilidad para el despliegue en producción. Además, permitió identificar que no se introdujo ningún error, asegurando que los usuarios finales no se vieran afectados.

Es importante destacar que se llevaron a cabo pruebas para la detección de diferencias exitosamente. Este procedimiento se realizó de manera sistemática y minuciosa, permitiéndonos identificar con precisión los casos en los que surgieron discrepancias. Este enfoque meticuloso contribuyó significativamente a nuestro entendimiento de las diferencias y su origen, brindando una perspectiva clara sobre los casos en los

que se presentaron variaciones.

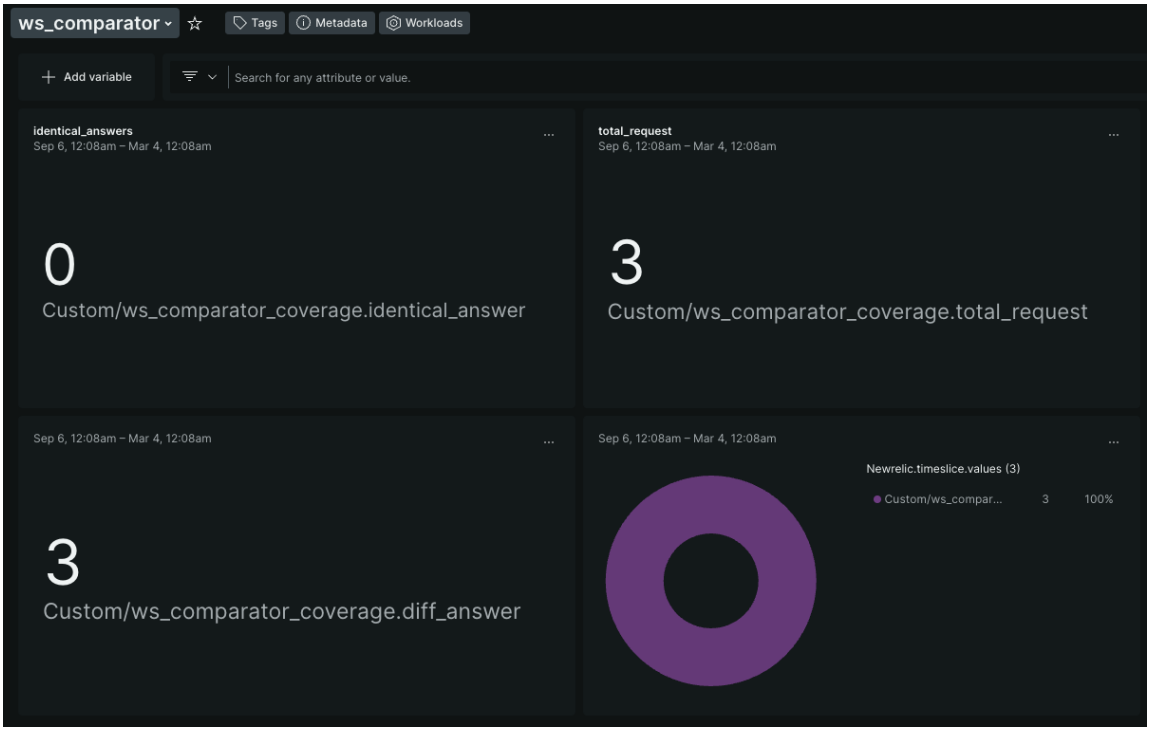


Figura 17: Resultados Diff NewRelic.

5.10. Uso del comparador

Para utilizar eficazmente este comparador, el primer paso implica desplegarlo en un servicio como Cloud Run. Es importante destacar que, aunque se puede aprovechar otro servicio distinto a Cloud Run, el proyecto cuenta con un archivo Docker que posibilita el despliegue de una imagen en diversas tecnologías, no exclusivamente en GCP. Puedes encontrar el repositorio público correspondiente 7.2.2.

Una vez que el comparador se ha desplegado en un servicio, es necesario obtener la URL asociada. A continuación, se procede a ajustar el servicio de producción. En este escenario particular, empleamos goroutines en Go para enviar la solicitud al

comparador a través de la URL-CLOUD-RUN/v1/request/comparation. La solicitud sigue el siguiente modelo:

Figura 18: Request body

Es vital destacar que el campo "body"debe coincidir con el cuerpo que se recibe en el servicio de producción, y el campo response"debe ser el mismo cuerpo de respuesta que se obtiene en producción. Además, la URL representa el entorno UAT o Sandbox al que se desea realizar cambios.

Es necesario desplegar el servicio que aloja el entorno UAT o Sandbox apuntando a la base de datos productiva. Es importante tener en cuenta que esto se realiza específicamente para pruebas de endpoints idempotentes. Además, se debe escalar la base de datos, ya que se duplicará el tráfico. En nuestro caso, realizamos este escalado de una base de datos Mongo M30 a M40.

Para concluir el proceso, se integra con New Relic al configurar la clave correspondiente en el archivo .env del comparador.

6. Conclusiones

La implementación y uso del comparador en el proceso de despliegue del servicio de coverage ha demostrado ser una estrategia valiosa para garantizar la consistencia y la integridad del sistema. Al incorporar el comparador como parte integral del flujo de trabajo, se logró una mayor confiabilidad en la detección de posibles discrepancias entre la versión anterior y la refactorizada del servicio.

La activación exitosa del comparador en producción, respaldada por la escalabilidad de la base de datos, permitió una validación exhaustiva de la lógica de negocio sin afectar negativamente la experiencia del usuario final. La capacidad de escalar la base de datos a M40 fue esencial para manejar la carga adicional generada por el comparador, demostrando así que la inversión en la escalabilidad puede ser crucial para asegurar pruebas significativas.

En este contexto, la implementación del comparador no solo ha fortalecido la confianza en la estabilidad del servicio, sino que también ha proporcionado una herramienta eficaz para la identificación temprana de posibles problemas. Este enfoque proactivo contribuye a minimizar el riesgo de errores en producción y facilita la corrección oportuna de cualquier discrepancia.

La integración del comparador como parte esencial del proceso de despliegue ha demostrado ser una práctica valiosa para mantener la coherencia y la calidad del servicio de coverage. La experiencia positiva obtenida respalda la recomendación de continuar utilizando y mejorando esta herramienta en futuros despliegues, consolidándola como una medida efectiva para garantizar la estabilidad y confiabilidad del sistema.

Es importante mencionar que este enfoque se presenta como un prototipo, ya que su implementación inicial ha arrojado resultados prometedores. Sin embargo, la consideración de un prototipo implica una fase de prueba y refinamiento continuo. La adaptabilidad y mejora constante son esenciales para asegurar que el comparador

evolucione en respuesta a los cambios en el servicio y las necesidades del sistema. Este enfoque prototipo proporciona una base sólida para futuras iteraciones y mejoras en la estrategia de comparación.

A continuación, se detalla cómo se logró cada objetivo, con el propósito de proporcionar una conclusión más comprensiva.

6.0.1. Identificación de consecuencias y repercusiones (Primer objetivo)

Con el propósito de profundizar en la comprensión de los errores en el desarrollo de software, llevé a cabo una investigación que culminó en la creación de un documento, disponible en la sección 5.2. Este documento sirvió como base para identificar las consecuencias y repercusiones de los errores, proporcionando un marco sólido para abordar el primer objetivo.

6.0.2. Desarrollo de un servicio web comparador (Segundo objetivo)

Con el objetivo de alcanzar el segundo hito, se concibió y materializó un servicio web comparador. Este servicio posee la habilidad de analizar y contrastar respuestas, facilitando la detección efectiva de discrepancias en los servicios REST evaluados. Con el compromiso de fomentar la colaboración y mejora continua, esta herramienta se ha compartido en GitHub 7.2.2, brindando a la comunidad la oportunidad de contribuir y a cualquier empresa la posibilidad de utilizarla.

6.0.3. Elaboración de caracterización de discrepancias (Tercer objetivo)

Dentro del alcance del tercer objetivo, se llevó a cabo una caracterización de las posibles discrepancias que podrían manifestarse durante el proceso de comparación de respuestas. Esta fase resulta esencial para obtener una comprensión profunda de las diversas formas en que los servicios REST pueden desviarse. Como parte de esta iniciativa, se desarrolló un dashboard en New Relic, proporcionando una

presentación más condensada y visual de las discrepancias, contribuyendo así a una mejor comprensión de la variabilidad en los servicios evaluados.

6.0.4. Pruebas de validación en ambiente productivo (Cuarto objetivo)

Con el propósito de garantizar la operatividad adecuada del servicio desarrollado, se procedió a su implementación en una arquitectura real dentro de un entorno productivo. Estas pruebas de validación desempeñaron un papel fundamental al asegurar la eficacia del servicio, detectando discrepancias incluso en un servicio que había sido refactorizado. Este enfoque no solo posibilitó la identificación proactiva de posibles inyecciones de errores, sino que también contribuyó a consolidar la fiabilidad y robustez del sistema. Es importante destacar que este proceso fue exitosamente implementado en el entorno empresarial de 99minutos.

7. Recomendaciones y Trabajos Futuros

7.1. Estrategia de Goteo Mejorada

7.1.1. Optimización del Random

7.1.1.1. Algoritmo de Selección: Considerar la implementación de un algoritmo de selección más sofisticado para la estrategia de goteo. Pueden explorarse técnicas basadas en aprendizaje automático o análisis predictivo para una selección más inteligente.

7.2. Implementación de Tags Dinámicos

7.2.1. Configuración Dinámica:

Desarrollar una funcionalidad que permita la configuración dinámica de tags en New Relic para alimentar diferentes dashboards. Esto facilitará el uso del comparador por parte de diversos equipos simultáneamente, brindando una visión personalizada para cada uno.

7.2.2. Contextualización de Datos:

Utilizar tags para contextualizar los datos en función de parámetros específicos, como equipos, entornos o proyectos. Esto mejorará la capacidad de análisis y facilitará la identificación de tendencias y patrones.

Referencias

Almeida, V., Dowdy, L., Dowdy, L., Almeida, V. A. F., Menas, for Higher Education (Firm), O., and an O'Reilly Media Company. Safari (2004). *Performance by Design: Computer Capacity Planning by Example*. Publisher.

Ammann, P. and Offutt, J. (2008). An introduction to software testing. In *International Conference on Software Testing, Verification and Validation (ICST 2008)*, pages 498–499. IEEE.

Andrews, J., Briand, L., Labiche, Y., and Namin, A. (2006). Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624.

Bass, L., Clements, P., and Kazman, R. (2012). *Software Architecture in Practice*. Addison-Wesley.

Beizer, B. (1990). *Software Testing Techniques*. International Thomson Computer Press.

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams.

Binder, R. (2000). *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley object technology series. Addison-Wesley.

Dallmeier, V., Zimmermann, T., Zeller, A., and Bird, C. (2015). A comparative analysis of production and test failures. *Empirical Software Engineering*, 20(2):478–511.

Fewster, M. and Graham, D. (1999). *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press Series. Addison-Wesley.

Golan, P. (2022). 99 minutos: la startup de envíos express que se expande por latinoamérica.

Halili, E. H. (2000). *Jmeter*.

Howard, M. and Lipner, S. (2006). *The security development lifecycle : SDL, a process for developing demonstrably more secure software*. Microsoft Press.

IEEE Computer Society (2014). Ieee guide for software verification and validation plans.

Jamil, M. A., Arif, M., Abubakar, N. S. A., and Ahmad, A. (2017). Software testing techniques: A literature review. *Proceedings - 6th International Conference on Information and Communication Technology for the Muslim World, ICT4M 2016*, pages 177–182.

Jones, C. and Bonsignour, O. (2012). *The Economics of Software Quality*. Addison-Wesley Professional.

Jorgensen, M. and Shepperd, M. (2007). Experience-based guidelines for test case generation. In *Proceedings of the 2007 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 225–234. IEEE.

Kaczmarek, J., Persson, P., and Engström, E. (2003). Automated black-box test case generation for path coverage. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE 2003)*, pages 269–280. IEEE.

Kaner, C., Falk, J., and Nguyen, H. Q. (2004). Automated software testing. *Software Testing & Quality Engineering Magazine*, 6(3):20–27.

Kaner, C. Falk, J. N. H. Q. (1999). *Testing Computer Software*. Wiley, New York, 2nd edition edition.

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press.

Myers, G., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*. ITPro collection. Wiley.

Nielsen, J. (1993). *Usability Engineering*. Academic Press.

Papadakis, M., Daka, E., and Malevris, N. (2018). Automated software testing: a systematic literature review. *Software Quality Journal*, 26(3):621–669.

Parra, R. (2023). 99minutos acelera su estrategia de crecimiento en américa latina para 2023.

Patton, R. (2005). *Software Testing*. Sams Publishing.

Pfleeger, S. L. and Atlee, J. M. (2016). *Software Engineering: Theory and Practice*. Pearson.

Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill higher education. McGraw-Hill Education.

Rani, S. and Gupta, D. (2018). A comparative study of different software testing techniques: a review. *J. Adv. Shell Program*, 5(1):1–8.

Salam, A. and Mohd, M. R. (2013). Automated test case generation from uml activity diagrams using backtracking search algorithm. *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on*, pages 717–722.

Society, I. C. (2004). Guide to the software engineering body of knowledge (swebok).

Sommerville, I. (2016). *Software engineering*.

Wiegers, K. E. (2003). *Software Requirements*. Microsoft Press.

Anexos

Anexo 1: Diseño por Attribute Domain Design

ASR1 RENDIMIENTO

Rendimiento	
Unidad:	% de tiempo de respuesta luego de procesar la comparación

Actor:	Comparador	Estimulo:	
Ambiente:	Operación normal	Artefacto:	Servicio de comparador
Respuesta Esperada:	Disminución en los tiempos al realizar la comparación.		

Interoperabilidad	
Prioridad:	Alta
Impacto:	Al realizar mejoras de rendimiento se pueden ver afectados sistemas externos los cuales no cumplen con los tiempos esperados.

ADD

1. Motivadores de la Iteración.

Motivador	Descripción
Procesar la comparación	El servicio de comparación puede ser altamente concurrente por el número de solicitudes, por ende, el servicio debe tener un buen rendimiento cuando realice la comparación.

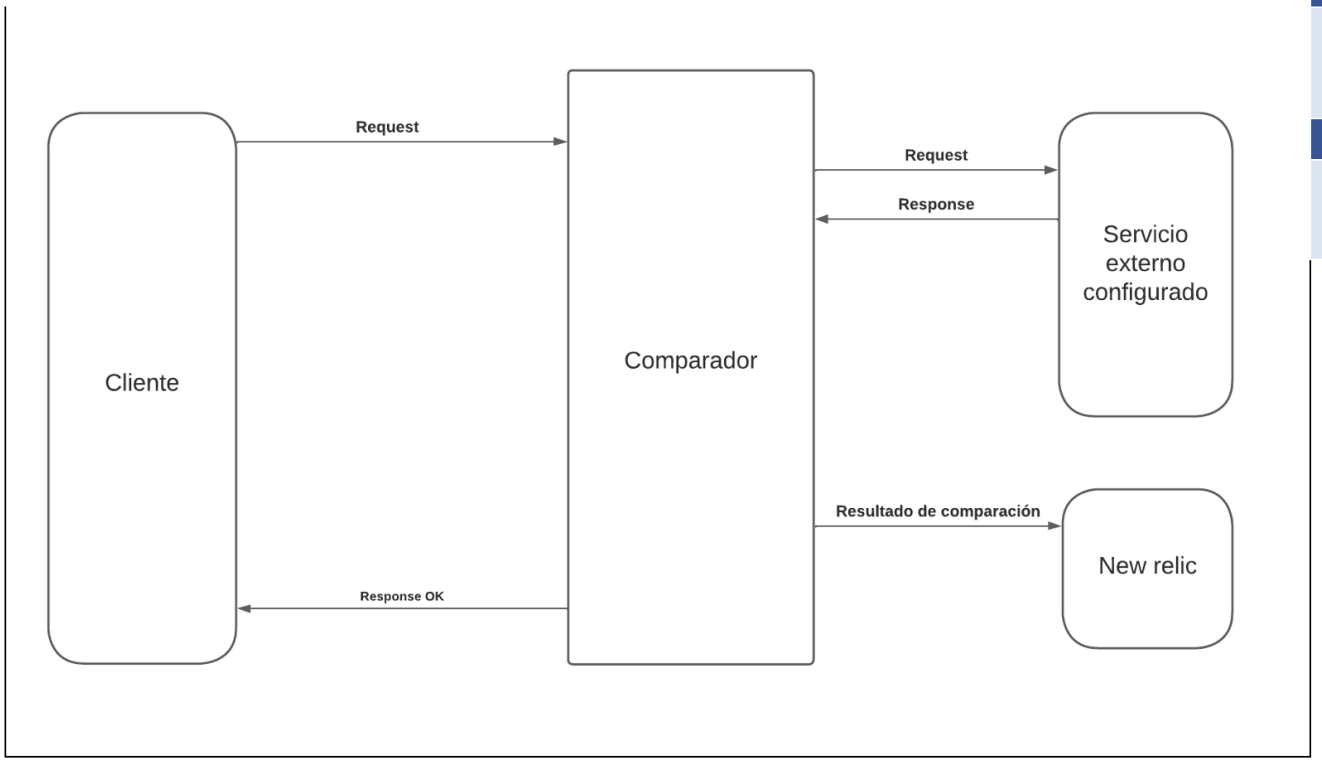
2. Conceptos que satisfacen los Motivadores.

Estilo Arquitectura	Justificación
Arquitectura de microservicios	Este estilo de arquitectura crear un componente especializado para las comparaciones e identificar las diferencias en diferentes estructuras.

3. Bosquejos iniciales de Vista y Conocimiento.

Cliente:	N/A	Proyecto:	Comparador	Grupo:		Fecha:	
Vista:	Contexto	Titulo:	ASR1 Rendimiento	ID:	1	Versión:	1.0

Estilo:
Arquitectura Microservicios
Notación:

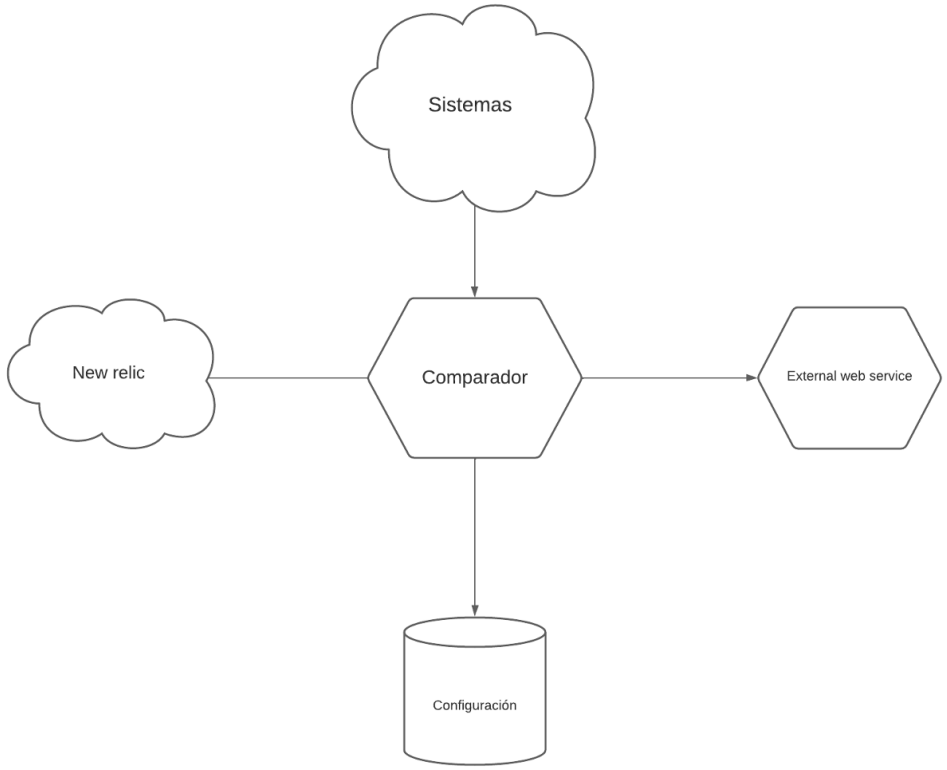


Cliente:	N/A	Proyecto:	Comparador	Grupo:		Fecha:	
Vista:	Funcional	Titulo:	ASR 1 Rendimiento	ID:	2	Versión:	1.0

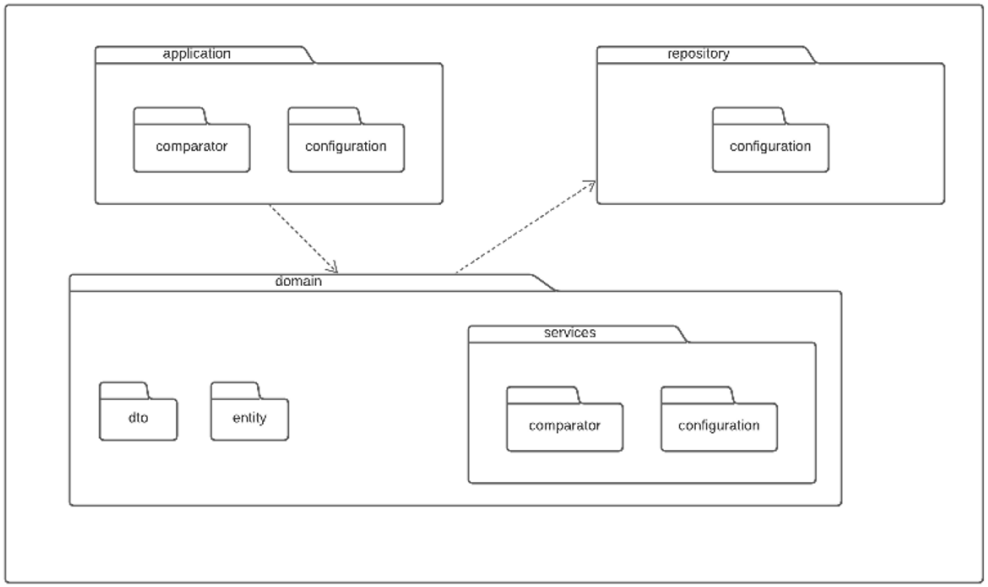
Estilo:

Arquitectura Microservicios

Notación:



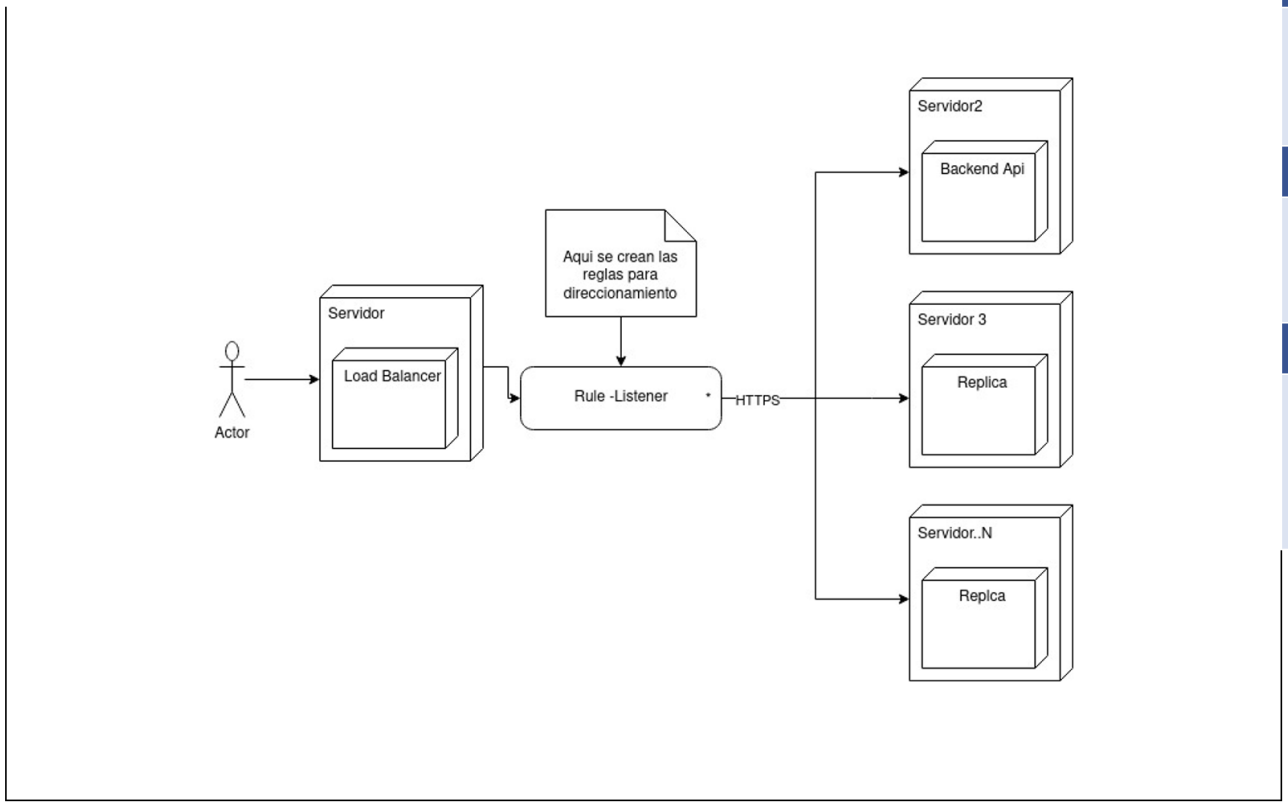
Cliente:	N/A	Proyecto:	Comparador	Grupo:		Fecha:	
Vista:	Desarrollo	Titulo:	ASR1 Rendimiento	ID:	3	Versión:	1.0



Estilo:
Arquitectura Microservicios
Tácticas:
Notación:
UML

Cliente:	N/A	Proyecto:	Comparador	Grupo:		Fecha:	
Vista:	Despliegue	Titulo:	ASR1 Rendimiento	ID:	4	Versión:	1.0

Estilo:
Arquitectura Microservicios
Tácticas:
Notación:
UML



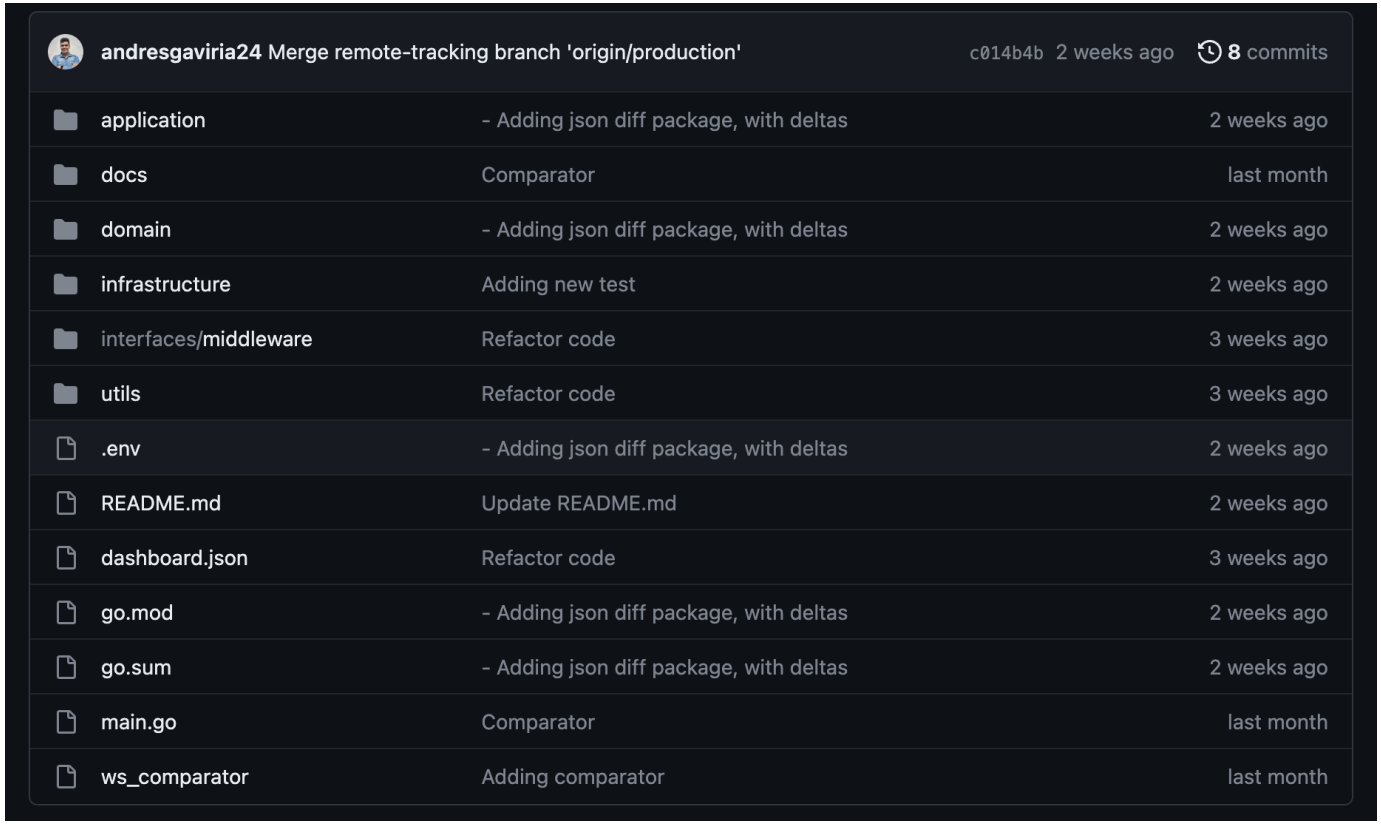
4. Endpoints del comparador.

/v1/request/comparation

```
{
  "method": "POST",
  "body": {
    "generic_body": "body_request"
  },
  "query_params": {
    "param1": "valor1",
    "param2": "valor2"
  },
  "response": {
    "generic_response": "response_body"
  },
  "url": "https://api.example.com/resource"
}
```


Anexo 2: Repositorio de GitHub

Para más información se encuentra el repositorio de GitHub: `https://github.com/andresgaviria24/ws_comparator`


andresgaviria24 Merge remote-tracking branch 'origin/production'
c014b4b 2 weeks ago
8 commits
application - Adding json diff package, with deltas 2 weeks ago
docs Comparator last month
domain - Adding json diff package, with deltas 2 weeks ago
infrastructure Adding new test 2 weeks ago
interfaces/middleware Refactor code 3 weeks ago
utils Refactor code 3 weeks ago
.env - Adding json diff package, with deltas 2 weeks ago
README.md Update README.md 2 weeks ago
dashboard.json Refactor code 3 weeks ago
go.mod - Adding json diff package, with deltas 2 weeks ago
go.sum - Adding json diff package, with deltas 2 weeks ago
main.go Comparator last month
ws_comparator Adding comparator last month

Anexo 3: Código de comparación

```go
func (r *ComparatorServiceImpl) Comparator(comparator dto.ComparatorIn, app *newrelic.Application) dto.Response {
    var response dto.Response

    payload := make(map[string]interface{})
    for k, b := range comparator.Body {
        payload[k] = b
    }

    client := resty.New()
    resp, err := client.R().
        SetHeader("Content-Type", "application/json").
        SetBody(payload).
        Post(comparator.Url)

    if err != nil {
        fmt.Println("Error realizando la solicitud:", err)
        return responseWithError(response, app)
    }

    result := make(map[string]interface{})
    if err := json.Unmarshal(resp.Body(), &result); err != nil {
        fmt.Println("Error al decodificar JSON:", err)
        return responseWithError(response, app)
    }

    diff, deltaJSON := diffjson.DiffJson(result, comparator)

    newrelic_metrics.SendMetric(diff, app, deltaJSON)

    response.Status = http.StatusOK
    response.Description = diff.Deltas()
    return response
}
```

```go
func DiffJson(result map[string]interface{}, comparator dto.ComparatorIn) (diff.Diff, string) {
    differ := diff.New()
    diff := differ.CompareObjects(comparator.ResponseC, result)

    f := format.NewAsciiFormatter(result, format.AsciiFormatterDefaultConfig)
    deltaJSON, _ := f.Format(diff)
    return diff, deltaJSON
}
```

```go
func (differ *Differ) CompareObjects(
    left map[string]interface{},
    right map[string]interface{},
) Diff {
    deltas := differ.compareMaps(left, right)
    return &diff{deltas: deltas}
}
```

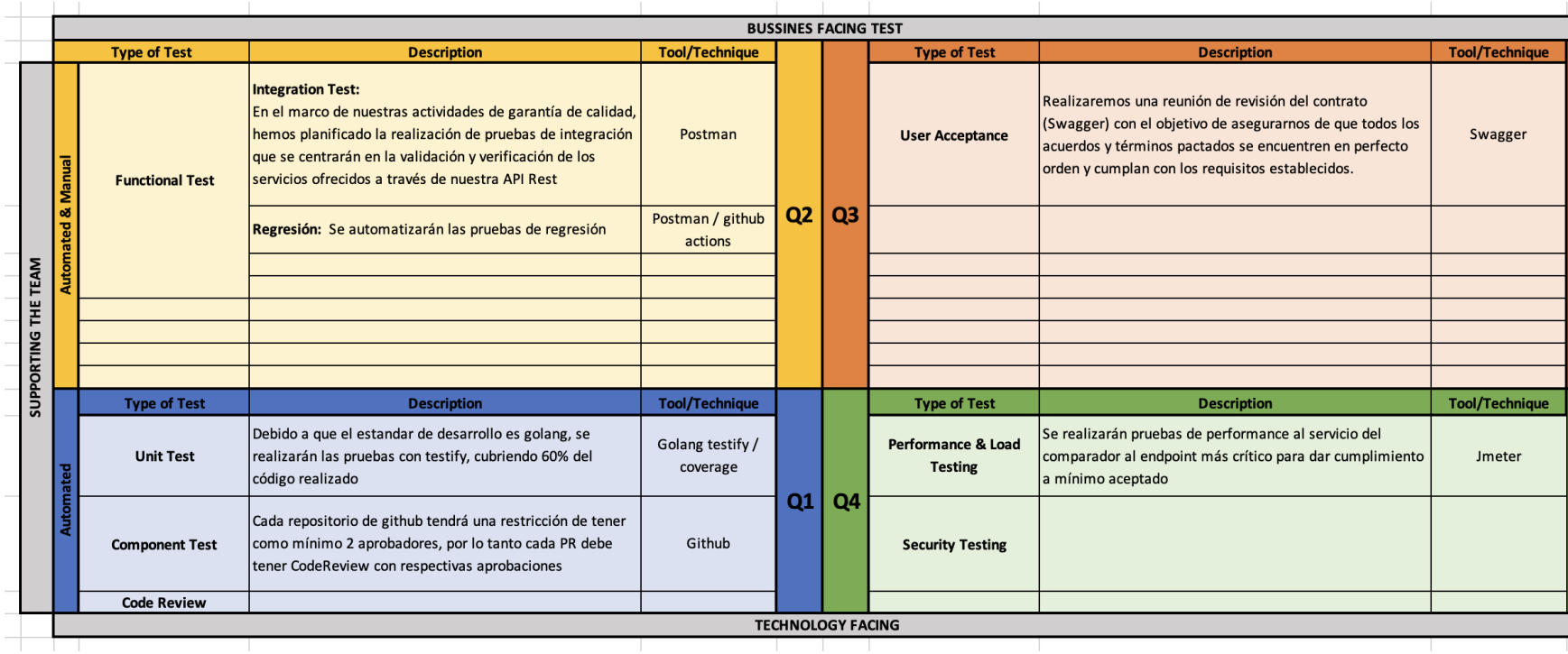
```go
func (differ *Differ) compareMaps(
    left map[string]interface{},
    right map[string]interface{},
) (deltas []Delta) {
    deltas = make([]Delta, 0)

    names := sortedKeys(left)
    for _, name := range names {
        if rightValue, ok := right[name]; ok {
            same, delta := differ.compareValues(Name(name), left[name], rightValue)
            if !same {
                deltas = append(deltas, delta)
            }
        } else {
            deltas = append(deltas, NewDeleted(Name(name), left[name]))
        }
    }

    names = sortedKeys(right)
    for _, name := range names {
        if _, ok := left[name]; !ok {
            deltas = append(deltas, NewAdded(Name(name), right[name]))
        }
    }

    return deltas
}
```

```go
func (differ *Differ) compareArrays(
    left []interface{},
    right []interface{},
) (deltas []Delta) {
    deltas = make([]Delta, 0)
    lcsPairs := lcs.New(left, right).IndexPairs()

    maybeDeleted := list.New()
    lcsI := 0
    for i, leftValue := range left {
        if lcsI < len(lcsPairs) && lcsPairs[lcsI].Left == i {
            lcsI++
        } else {
            maybeDeleted.PushBack(maybe{index: i, lcsIndex: lcsI, item: leftValue})
        }
    }

    maybeAdded := list.New()
    lcsI = 0
    for i, rightValue := range right {
        if lcsI < len(lcsPairs) && lcsPairs[lcsI].Right == i {
            lcsI++
        } else {
            maybeAdded.PushBack(maybe{index: i, lcsIndex: lcsI, item: rightValue})
        }
    }

    var delNext *list.Element
    for delCandidate := maybeDeleted.Front(); delCandidate != nil; delCandidate = delNext {
        delCan := delCandidate.Value.(maybe)
        delNext = delCandidate.Next()

        for addCandidate := maybeAdded.Front(); addCandidate != nil; addCandidate = addCandidate.Next() {
            addCan := addCandidate.Value.(maybe)
            if reflect.DeepEqual(delCan.item, addCan.item) {
                deltas = append(deltas, NewMoved(Index(delCan.index), Index(addCan.index), delCan.item, nil))
                maybeAdded.Remove(addCandidate)
```

```go
func isDropActive(envGetter utils.EnvGetter) bool {
    return envGetter.GetBoolEnv("DRIP_STRATEGY_ACTIVE")
}

func Drip(c *gin.Context, envGetter utils.EnvGetter) bool {

    if !isDropActive(envGetter) {
        return true
    }

    rand.Seed(time.Now().UnixNano())

    threshold := envGetter.GetDoubleEnv("DRIP_STRATEGY_PORCENTAJE") / 100

    random := rand.Float64()

    if random < threshold {
        return true
    }

    c.JSON(http.StatusOK, dto.Response{})
    return false
}
```

```go
func (r *ComparatorController) ComparatorHandler(c *gin.Context) {
    var (
        comparator dto.ComparatorIn
        response   = dto.Response{
            Status: http.StatusOK,
        }
    )

    if !drip.Drip(c, &utils.RealUtils{}) {
        return
    }

    if err := c.ShouldBindJSON(&comparator); err != nil {
        c.JSON(http.StatusUnprocessableEntity, dto.Response{})
        return
    }

    if response = r.comparatorService.Comparator(comparator, r.newrelicClient); response.Status != http.StatusOK {
        c.JSON(response.Status, response)
    }

    c.JSON(http.StatusOK, response)

}
```

Anexo 4: Estrategia de pruebas



		BUSSINES FACING TEST							
		Type of Test	Description	Tool/Technique			Type of Test	Description	Tool/Technique
SUPPORTING THE TEAM	Automated & Manual	Functional Test	**Integration Test:** En el marco de nuestras actividades de garantía de calidad, hemos planificado la realización de pruebas de integración que se centrarán en la validación y verificación de los servicios ofrecidos a través de nuestra API Rest	Postman	Q2	Q3	User Acceptance	Realizaremos una reunión de revisión del contrato (Swagger) con el objetivo de asegurarnos de que todos los acuerdos y términos pactados se encuentren en perfecto orden y cumplan con los requisitos establecidos.	Swagger
			Regresión: Se automatizarán las pruebas de regresión	Postman / github actions					
	Automated	Type of Test	Description	Tool/Technique			Type of Test	Description	Tool/Technique
		Unit Test	Debido a que el estandar de desarrollo es golang, se realizarán las pruebas con testify, cubriendo 60% del código realizado	Golang testify / coverage	Q1	Q4	Performance & Load Testing	Se realizarán pruebas de performance al servicio del comparador al endpoint más crítico para dar cumplimiento a mínimo aceptado	Jmeter
		Component Test	Cada repositorio de github tendrá una restricción de tener como mínimo 2 aprobadores, por lo tanto cada PR debe tener CodeReview con respectivas aprobaciones	Github			Security Testing		
		Code Review							
		TECHNOLOGY FACING							

Figura 19: Test.

Anexo 5: Pruebas de rendimiento

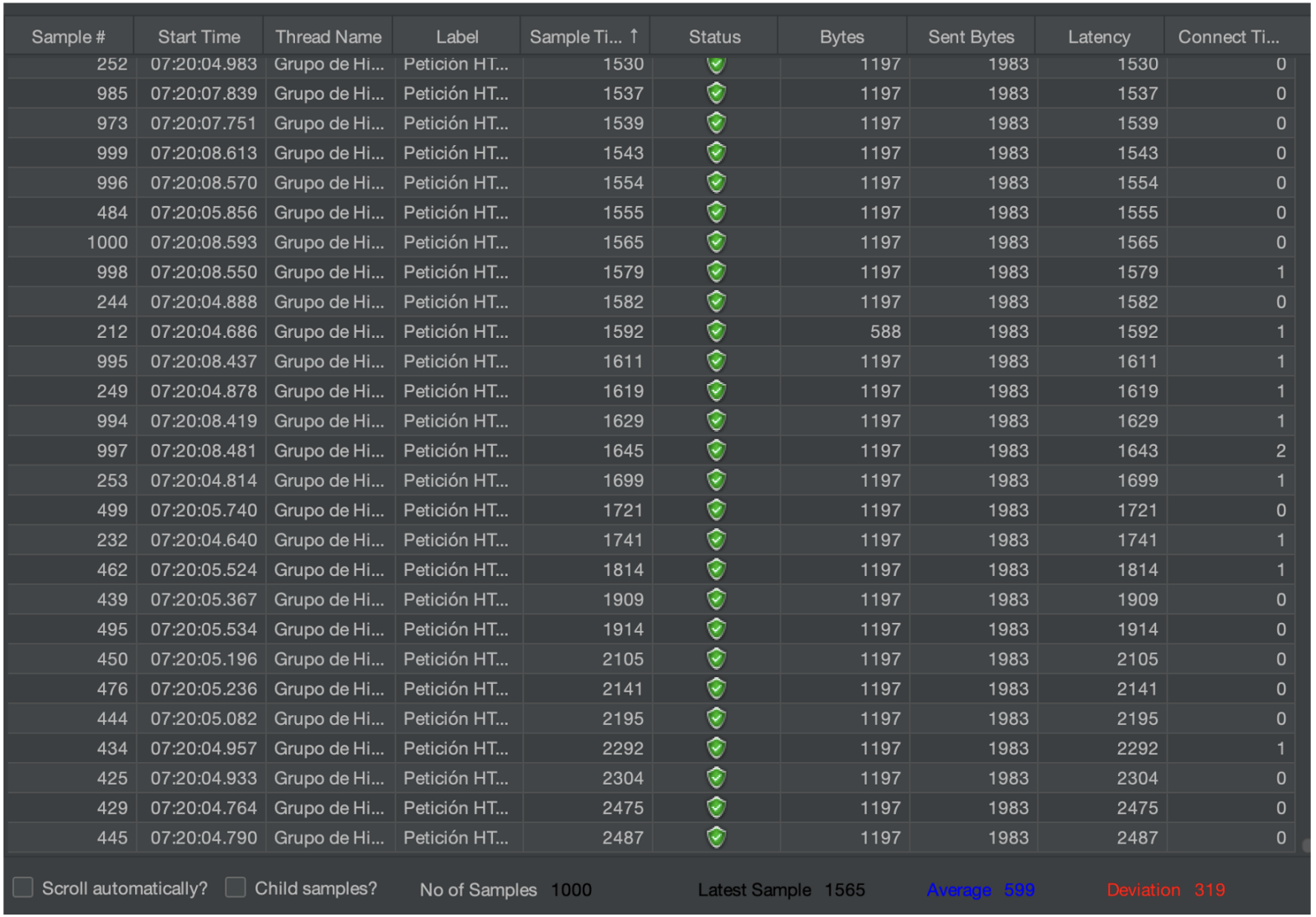
N	Endpoint	Number of Threads (users)	Ramp-up period (seconds)	Loop Count	Method	CPU (%) Initial	Memory (MB) Initial	CPU (%) Process	Memory (MB) Process	Result Tree Average (ms)
1	/v1/request/comparation	100	1	1	POST	0	0	5,9	48	643
2	/v1/request/comparation	1000	4	1	POST	0	0	17,8	153,8	599
							Maximum	17,8	153,8	643
							Minimum	5,9	48	599
							Average	11,9	100,9	621

Figura 20: Resultado test de rendimiento.

Sample #	Start Time	Thread Name	Label	Sample Ti... ↑	Status	Bytes	Sent Bytes	Latency	Connect Ti...
11	07:18:49.392	Grupo de Hi...	Petición HT...	693		588	1983	692	1
40	07:18:49.631	Grupo de Hi...	Petición HT...	695		588	1983	695	1
51	07:18:49.752	Grupo de Hi...	Petición HT...	697		588	1983	697	0
9	07:18:49.369	Grupo de Hi...	Petición HT...	712		588	1983	712	2
90	07:18:50.181	Grupo de Hi...	Petición HT...	731		588	1983	731	0
6	07:18:49.343	Grupo de Hi...	Petición HT...	736		588	1983	736	0
31	07:18:49.494	Grupo de Hi...	Petición HT...	742		588	1983	742	0
1	07:18:49.296	Grupo de Hi...	Petición HT...	748		588	1983	748	6
13	07:18:49.331	Grupo de Hi...	Petición HT...	758		588	1983	758	0
36	07:18:49.545	Grupo de Hi...	Petición HT...	761		588	1983	761	0
14	07:18:49.318	Grupo de Hi...	Petición HT...	772		588	1983	772	0
12	07:18:49.306	Grupo de Hi...	Petición HT...	781		588	1983	781	0
95	07:18:50.215	Grupo de Hi...	Petición HT...	785		588	1983	785	0
94	07:18:50.165	Grupo de Hi...	Petición HT...	801		588	1983	801	0
44	07:18:49.588	Grupo de Hi...	Petición HT...	808		588	1983	808	1
30	07:18:49.405	Grupo de Hi...	Petición HT...	823		588	1983	823	1
27	07:18:49.356	Grupo de Hi...	Petición HT...	853		588	1983	853	7
93	07:18:49.954	Grupo de Hi...	Petición HT...	982		588	1983	982	0
92	07:18:49.895	Grupo de Hi...	Petición HT...	1030		588	1983	1030	0
52	07:18:49.379	Grupo de Hi...	Petición HT...	1083		588	1983	1083	1
91	07:18:49.827	Grupo de Hi...	Petición HT...	1087		588	1983	1087	1
88	07:18:49.557	Grupo de Hi...	Petición HT...	1349		588	1983	1349	1
99	07:18:50.141	Grupo de Hi...	Petición HT...	1355		588	1983	1355	0
100	07:18:50.117	Grupo de Hi...	Petición HT...	1469		588	1983	1469	1
97	07:18:49.883	Grupo de Hi...	Petición HT...	1584		588	1983	1584	1
98	07:18:49.742	Grupo de Hi...	Petición HT...	1754		588	1983	1754	1
96	07:18:49.519	Grupo de Hi...	Petición HT...	1921		588	1983	1921	1

Scroll automatically? Child samples? No of Samples 100 Latest Sample 1469 Average 643 Deviation 277

Figura 21: 100 peticiones concurrentes.



Sample #	Start Time	Thread Name	Label	Sample Ti... ↑	Status	Bytes	Sent Bytes	Latency	Connect Ti...
252	07:20:04.983	Grupo de Hi...	Petición HT...	1530		1197	1983	1530	0
985	07:20:07.839	Grupo de Hi...	Petición HT...	1537		1197	1983	1537	0
973	07:20:07.751	Grupo de Hi...	Petición HT...	1539		1197	1983	1539	0
999	07:20:08.613	Grupo de Hi...	Petición HT...	1543		1197	1983	1543	0
996	07:20:08.570	Grupo de Hi...	Petición HT...	1554		1197	1983	1554	0
484	07:20:05.856	Grupo de Hi...	Petición HT...	1555		1197	1983	1555	0
1000	07:20:08.593	Grupo de Hi...	Petición HT...	1565		1197	1983	1565	0
998	07:20:08.550	Grupo de Hi...	Petición HT...	1579		1197	1983	1579	1
244	07:20:04.888	Grupo de Hi...	Petición HT...	1582		1197	1983	1582	0
212	07:20:04.686	Grupo de Hi...	Petición HT...	1592		588	1983	1592	1
995	07:20:08.437	Grupo de Hi...	Petición HT...	1611		1197	1983	1611	1
249	07:20:04.878	Grupo de Hi...	Petición HT...	1619		1197	1983	1619	1
994	07:20:08.419	Grupo de Hi...	Petición HT...	1629		1197	1983	1629	1
997	07:20:08.481	Grupo de Hi...	Petición HT...	1645		1197	1983	1643	2
253	07:20:04.814	Grupo de Hi...	Petición HT...	1699		1197	1983	1699	1
499	07:20:05.740	Grupo de Hi...	Petición HT...	1721		1197	1983	1721	0
232	07:20:04.640	Grupo de Hi...	Petición HT...	1741		1197	1983	1741	1
462	07:20:05.524	Grupo de Hi...	Petición HT...	1814		1197	1983	1814	1
439	07:20:05.367	Grupo de Hi...	Petición HT...	1909		1197	1983	1909	0
495	07:20:05.534	Grupo de Hi...	Petición HT...	1914		1197	1983	1914	0
450	07:20:05.196	Grupo de Hi...	Petición HT...	2105		1197	1983	2105	0
476	07:20:05.236	Grupo de Hi...	Petición HT...	2141		1197	1983	2141	0
444	07:20:05.082	Grupo de Hi...	Petición HT...	2195		1197	1983	2195	0
434	07:20:04.957	Grupo de Hi...	Petición HT...	2292		1197	1983	2292	1
425	07:20:04.933	Grupo de Hi...	Petición HT...	2304		1197	1983	2304	0
429	07:20:04.764	Grupo de Hi...	Petición HT...	2475		1197	1983	2475	0
445	07:20:04.790	Grupo de Hi...	Petición HT...	2487		1197	1983	2487	0

Scroll automatically? Child samples? No of Samples 1000 Latest Sample 1565 Average 599 Deviation 319

Figura 22: 1.000 peticiones concurrentes.