ULPGC
Universidad de
Las Palmas de
Gran Canaria

ESCUELA DE
INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

Pruebas de Carga y Análisis de Rendimiento en la plataforma digital Planimatik

TITULACIÓN: Grado en ingeniería informática

AUTOR: Fernando Marcelo Alonso

TUTORIZADO POR:

Francisco José Santana Pérez
Felipe Raúl Mendoza Álamo

Junio 2024

Índice

Índice de figuras

Índice de cuadros

Resumen

Este trabajo se centra en el análisis del rendimiento de una plataforma llamada Planimatik propiedad de The Singular Factory. Este trabajo se llevará a cabo utilizando herramientas específicas con las que realizaremos pruebas de carga. Estas nos permitirán monitorizar el rendimiento de la plataforma a lo largo del tiempo, analizar los resultados obtenidos y, basándonos en este análisis, propondremos y evaluaremos diversas estrategias para mejorar su rendimiento.

Planimatik es una herramienta en línea destinada a ofrecer a sus usuarios una solución para la planificación y monitorización de cargas, entendiéndose como mercancía en grandes volúmenes a transportar entre almacenes. Planimatik opera en el contexto del transporte y la logística por carretera de dichas cargas. Específicamente, se centra en el uso de camiones para el transporte, abordando tanto la planificación logística como la monitorización en tiempo real de las cargas transportadas. En un principio el mercado donde la plataforma operaba era Norteamérica, pero recientemente se ha expandido a Europa.

La motivación principal de este trabajo surge de las crecientes expectativas de uso de la plataforma Planimatik, las cuales han superado las previsiones iniciales de la empresa que lo ha creado (The Singular Factory), debido en parte a la expansión a Europa y el interés captado en este mercado. Ante esto, es crucial conocer en profundidad el rendimiento actual de la plataforma, con el fin de garantizar su crecimiento y escalabilidad. El propósito de este trabajo es, por tanto, abordar estas necesidades y asegurar la robustez y eficacia de Planimatik frente a los nuevos desafíos que pueda plantear su mayor demanda en un futuro.

Abstract

This paper focuses on analyzing the performance of a platform called Planimatik, owned by The Singular Factory. This work will be carried out using specific tools with which we will perform load testing. These will allow us to monitor the performance of the platform over time, analyze the results obtained, and, based on this analysis, propose and evaluate various strategies to improve its performance.

Planimatik is an online tool designed to offer its users a solution for planning and monitoring loads (commodities) in large volumes to be transported between warehouses. Planimatik operates in the context of road transport and logistics of such loads. Specifically, it focuses on the use of trucks for transport, addressing both logistic planning and real-time monitoring of transported loads. Initially, the platform was supported on the North American market but is now expanding to Europe.

The main motivation for this work stems from the growing usage expectations of the Planimatik platform, which have exceeded the initial forecasts of the company that created it (The Singular Factory), due in part to the expansion into Europe and the interest captured in this market. Given this, it is crucial to thoroughly understand the current performance of the platform, in order to ensure its growth and scalability. Therefore, the purpose of this work is to address these needs and ensure the robustness and effectiveness of Planimatik against the new challenges that its increased demand may pose in the future.

1 Introducción

1.1 Motivación personal

Desde que comencé a programar, me ha apasionado mejorar el rendimiento del código. A menudo, sin un objetivo claro, simplemente por el disfrute de ver hasta dónde se puede llegar como un reto personal. Esta inclinación me ha llevado a idear estrategias para mejorar el rendimiento del código en mi empresa. Sin embargo, siempre me han faltado herramientas adecuadas para analizar el rendimiento sin invertir demasiadas horas, especialmente considerando que una mejora en un componente concreto puede perjudicar al rendimiento general del sistema debido al aumento en el consumo de otro recurso, lo cual afecta a toda la plataforma y es complicado de localizar.

Por esta razón, me he interesado en las pruebas de carga, ya que permiten poner en práctica rápidamente diversas ideas y realizar pruebas de concepto de manera ágil. Estas pruebas aceleran el desarrollo y facilitan la valoración del trabajo realizado, ya que es más adecuado presentar cifras concretas sobre las mejoras en el rendimiento, en lugar de simplemente afirmar que el sistema parece más rápido.

Además, me interesan las pruebas de carga porque abarcan no solo la ejecución de las pruebas, sino también la extracción, análisis y visualización de los datos resultantes. Estas son habilidades valiosas que siempre es útil mejorar y tener en el arsenal de uno. Las herramientas de software que he elegido para este trabajo son ampliamente adoptadas en la industria, lo que hace que los conocimientos adquiridos sean transferibles incluso fuera del contexto específico de las pruebas de carga.

En general es un trabajo cuyo desarrollo amplía las fronteras de mi conocimiento y me obliga también a planificarme y ser autodidacta.

1.2 Estado actual y objetivos iniciales

1. **Conocer en profundidad el rendimiento de la plataforma Planimatik (de The Singular Factory) y dimensionarlo para prever y localizar futuras necesidades en su explotación, mantenimiento y producción de su gestión informática.**

   Se ha cumplido en un 50 %, conocemos cuáles son las debilidades del sistema y no solo cuál puede ser el uso de usuarios concurrentes, sino cuáles serían las estrategias más efectivas a la hora de escalar el sistema. Tal vez haya faltado por falta de tiempo un dimensionamiento de cuántos usuarios podría soportar un sistema en producción actualmente cumpliendo unos requisitos mínimos de rendimiento.

2. **Detectar problemas de rendimiento en la plataforma de producción actual del software Planimatik y sugerir soluciones para los mismos.**

   En un 100 % se ha cumplido el objetivo, en el trabajo hemos profundizado en buena parte en esto, se han propuesto e implementado mejoras de rendimiento, además de dar algunas claves de puntos donde puede potenciarse aún más el rendimiento a futuro.

3. **Establecer un nuevo estándar de medida (benchmark) del estado actual de la plataforma Planimatik en cuanto a rendimiento informático que nos ayude en la localización de problemas de manera lo más inmediata posible y nos sirva como guía en la toma de decisiones. Con la ejecución continua del benchmark será fácil detectar cambios que tengan un impacto en el rendimiento mayor al previsto previamente.**

   En un 80 % se ha cumplido el objetivo, se ha creado un escenario de prueba que tiene como propósito probar la plataforma en su totalidad, dándonos un buen estándar de medida replicable. Además, se han implementado las pruebas de carga en todo el flujo de desarrollo, con lo que la retroalimentación en cuanto al estado del sistema después de introducir cambios es inmediata.

4. **Crear un cuadro de mandos (dashboard) donde poder visualizar las métricas extraídas, además de tener unos límites predefinidos para dichas métricas, los cuales nos indicarán si nos encontramos o no en los rangos esperados de rendimiento. Desde aquí también podremos comparar de forma visual distintas ejecuciones de los test para comprobar si ha habido alguna mejora o se ha degradado el rendimiento del sistema.**

   En un 80 % se ha cumplido, se han implementado varios cuadros de mandos donde monitorizar de forma sencilla los resultados. Estos cuadros de mandos incluyen métricas útiles de todos los sistemas involucrados en la ejecución de los test.

1.3 Competencias específicas y aportaciones del trabajo

- **Código CII01** "Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente."
- **Código IS01** "Capacidad para desarrollar, mantener y evaluar servicios y sistemas software que satisfagan todos los requisitos del usuario y se comporten de forma fiable y eficiente, sean asequibles de desarrollar y mantener y cumplan normas de calidad, aplicando las teorías, principios, métodos y prácticas de la ingeniería del software."
- **Código IS05** "Capacidad de identificar, evaluar y gestionar los riesgos potenciales asociados que pudieran presentarse."

Las aportaciones del proyecto son a nivel de mejorar la competitividad tecnológica de Planimatik en el sector. La aplicación de estrategias de pruebas de rendimiento da una herramienta más para el desarrollo en la empresa con la que asegurar el rendimiento del sistema, esto es clave para el crecimiento de la empresa, también puede ayudar en la retención de usuarios en última instancia lo que puede influir a nivel general en el proyecto.

1.4 Introducción a los test de carga

Las pruebas de carga son una herramienta la cual usando un software dedicado simulamos un volumen de usuarios en una API o sistemas informáticos.

1.4.1. Objetivos

Los objetivos de las pruebas de carga son múltiples, los principales son:

- Evaluar el sistema en circunstancias concretas, tales como picos de usuarios, uso intensivo y prolongado de la plataforma, o un uso moderado del sistema en prueba.
- Conocer la necesidad de requerimientos en cuanto a los sistemas informáticos, es decir, saber qué recursos son necesarios para abastecer a X número de usuarios con una serie de requisitos de rendimiento. Estos requisitos pueden ser que el 95 % de las solicitudes tarden menos de 2 segundos.
- Detectar cuellos de botella en el sistema antes de que afecten a los usuarios finales, permitiendo anticipar y gestionar posibles situaciones y escenarios críticos.
- Analizar que el rendimiento del sistema no se deteriora con el tiempo y también contabilizar el impacto que pueda tener una mejora de rendimiento.
- Evaluar distintas configuraciones de hardware y software, para determinar la combinación óptima que maximice el rendimiento y eficiencia del sistema, minimizando a su vez el consumo de recursos.

1.4.2. Conceptos/jerga

- **Escenario**: circunstancia que queremos probar, por ejemplo, queremos probar el número de usuarios máximos soportados por la plataforma, pero nos queremos centrar en los flujos relacionados con la descarga de documentos.
- **VU**: virtual users son usuarios virtuales creados por el software de pruebas de carga que usamos, cada uno de estos usuarios ejecuta de forma independiente y el objetivo es que se comporten de la forma más similar a los usuarios reales.

1.4.3. Métricas [1]

Hay muchas métricas posibles a la hora de hacer test de carga, las métricas a tener en cuenta dependen de los objetivos del test de carga y el contexto del mismo. Comentaré algunas de las más principales a continuación:

- **RPS (del inglés Request Per Second)**: Solicitudes que puede realizar el sistema por segundo. Es interesante medirlo para ver la evolución del mismo, aunque es complicado poner un número como objetivo. Cada sistema es distinto, con lo que las necesidades y patrones de comportamiento de los usuarios de cada sistema influyen en la magnitud de este número.
- **Latencia**: La latencia del sistema. En nuestro caso, la latencia entre el usuario y la API. En otros casos puede ser la latencia entre servidores o sistemas informáticos.

- **p9x**: Es una métrica basada en percentil, son utilizadas para comprender la distribución de tiempos de respuesta del sistema en cuestión. Por ejemplo, un p90 de 3 s significa que el 90 % de las solicitudes se realizan en menos de 3 s. Cuanto más recursos tiene una empresa, se pide un porcentaje mayor. Hay empresas que usan p999.
- **Métricas hardware**: Uso de CPU, uso de memoria, uso de disco, uso de red... Son complementarias y deben de usarse en conjunto con las otras métricas.

1.4.4. Tipos de pruebas de carga

Comentaremos las principales pruebas de carga usadas, aunque hay más de las nombradas y cada una de ellas tiene ramificaciones:

- **Prueba de Humo (smoke test)**: Se realiza con pocos usuarios para comprobar la lógica funcional y métricas básicas de un sistema tras cambios en el código. Es breve, dura de 30 segundos a 2-3 minutos.
- **Prueba de Carga promedio (average load test)**: Es la prueba más común y tiene el objetivo de analizar el sistema con un número de usuarios promedio. La duración es intermedia, de 5 a 60 minutos.
- **Prueba de Estrés (stress test)**: Se ejecuta con una alta cantidad de usuarios para ver cómo gestiona el sistema situaciones de carga por encima del promedio. Tiene una duración similar a la prueba de carga, de 5 a 60 minutos.
- **Prueba de Pico (spike)**: Son pruebas donde se aplica un número de usuarios elevado, son muy comunes cuando se esperan eventos estacionales o picos frecuentes de tráfico. Involucra una cantidad muy alta de usuarios durante un corto período de tiempo, usualmente de unos minutos.
- **Prueba de ruptura (breakpoint)**: Test de carga donde el número de usuarios se va incrementando hasta que el servidor no puede más y deja de responder, también funciona para probar la escalabilidad del sistema y localizar en qué punto falla.
- **Prueba de "remojo" (soak)**: Se prueba el servidor bajo una carga de usuarios media, durante un periodo largo de tiempo (horas), podemos con esto comprobar si el rendimiento se degrada con el tiempo, por ejemplo esto se podría dar si la liberación de la memoria es incorrecta o porque algún proceso no está terminando y liberando recursos correctamente. Lo de remojo es porque se deja ejecutando durante horas como dejándolo a remojo.

Podemos ver debajo en la figura 1 una gráfica con los test descritos con la duración y el volumen de usuarios empleados.

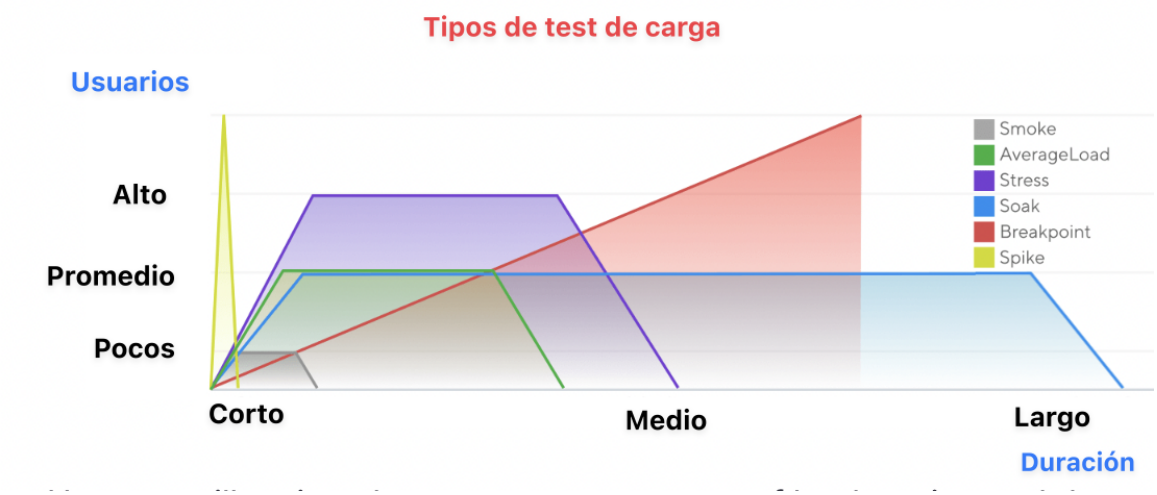


Figura 1: Gráfico de tipos test carga [2]

1.5 Estado del arte

Las pruebas llevan formando parte del desarrollo software desde los principios, tanto en formato unit testing, pruebas de integración y pruebas de carga. La importancia de las mismas ha incrementado con la profesionalización del sector, cada vez se tiene más en cuenta la mantenibilidad del código, la escalabilidad y el rendimiento. Además, se tiene más conocimiento del precio a pagar de un desarrollo deficiente, y el impacto a largo plazo de la deuda técnica.

1.5.1. Adopción en el mercado

En empresas con equipos pequeños o medianos, las pruebas de carga suelen ser dejadas de lado en algunos casos hasta el final del desarrollo, en otros casos nunca llegan a ser implementadas, lo que puede causar problemas durante el desarrollo y la entrega de software, llevando a sorpresas en el rendimiento de la plataforma en diversas circunstancias.

En cambio, en organizaciones de mayor tamaño, las pruebas de carga suelen integrarse en el proceso de desarrollo, estableciéndose métricas claras de rendimiento que deben ser cumplidas para el sistema o sistemas.

En parte, la adopción escasa en compañías pequeñas se debe a que tienen en promedio un equipo técnico con menos experiencia y las pruebas de carga en la mayoría de casos necesitan de mucho tiempo, planificación y conocimiento específico.

1.5.2. Evolución en métodos y herramientas

A lo largo de los años han ido surgiendo herramientas cada vez más sofisticadas para el desarrollo de las pruebas, la mayoría de las herramientas adoptadas por el sector han sido de código abierto. Podemos separar el desarrollo de las herramientas en 3 generaciones.

Primera generación

En un principio, algunas de las implementaciones de las pruebas de carga no usaban software dedicado para su realización, en algunos casos era puro scripting. Las primeras herramientas dedicadas específicamente a las pruebas de carga eran bastante rudimentarias. Incluso dichas herramientas específicas requerían de un conocimiento profundo de scripting para generar cargas de trabajo y analizar resultados. Algunas de estas primeras herramientas son JMeter o LoadRunner.

Segunda generación

Se empezaron a crear herramientas más simples de usar con más automatización y la capacidad de simular cargas de trabajo más realistas. Algunas de estas herramientas son Gatling y BlazeMeter. También incluían más integraciones con sistemas informáticos y mayor soporte en general por la creciente adopción de las mismas por parte de la industria.

Tercera generación

Herramientas que permiten pruebas en sistemas distribuidos y que se integran fácilmente con la nube. También se empezó a implementar más estas herramientas en el propio desarrollo continuo con la introducción de los pipelines de CI/CD. Locust y k6 son dos herramientas destacadas de esta generación.

Apunte final

Quiero destacar que las herramientas que he nombrado previamente por surgir en la primera o segunda generación no significa que no se usen a día de hoy. Todas estas herramientas se han modernizado y han adoptado las prácticas y funcionalidades que necesita la industria ahora mismo. Es más, todas las herramientas que he nombrado siguen siendo muy importantes a día de hoy en el mercado actual.

1.5.3. Método USE [3]

El método USE, abreviatura del inglés Utilization, Saturation, and Errors, es una técnica diseñada por Brendan Gregg para evaluar el rendimiento de un sistema. Aunque originalmente fue creado para el monitoreo y diagnóstico de problemas de rendimiento en sistemas operativos, este método también se ha aplicado a las pruebas de carga de aplicaciones y sistemas. Estos son los 3 pasos fundamentales del mismo:

Utilización

En este paso, se mide el grado en el que los recursos están siendo utilizados bajo una carga de trabajo específica. Esto incluye CPU, memoria, disco, ancho de banda de red, entre otros. La idea es identificar qué tan ”ocupados” están estos recursos durante el test. La alta utilización puede ser un indicador de un cuello de botella si se acompaña de otros síntomas negativos.

Saturación

La saturación se refiere al nivel en el que la demanda de un recurso excede su capacidad, provocando cuellos de botella. Por ejemplo, si la CPU está saturada, las tareas pueden empezar a ser encoladas, lo que aumenta los tiempos de respuesta y reduce el rendimiento. Medir la saturación ayuda a identificar si los recursos tienen capacidad suficiente para manejar la carga actual y futura.

Errores

Este paso implica monitorear y registrar errores relacionados con los recursos durante el test. Esto puede incluir errores de hardware, como fallos de disco, así como errores de software, como excepciones no capturadas o fallas de transacciones. La monitorización de errores es crucial, ya que incluso un sistema con buena utilización y baja saturación puede tener problemas ocultos que solo se revelan a través de los errores registrados.

1.5.4. Método RED

El método RED [4], acrónimo del inglés de Rate, Errors, and Duration, es otro enfoque útil para monitorear y evaluar el rendimiento de los sistemas, particularmente en ambientes de microservicios y arquitecturas distribuidas. Fue creado por Tom Wilkie intentando una aproximación mejor que el método USE para pruebas de carga, pues el método USE aplica más a hardware y red y no tanto a servicios y APIs. Diseñado para ser simple pero efectivo, el método RED se centra en tres métricas clave que, juntas, ofrecen una visión comprensiva de la salud y eficiencia de un servicio. Aquí te explico cómo se aplica el método RED para test de carga:

Tasa

Esta métrica mide el número de solicitudes por segundo que un servicio está manejando. En el contexto de testing de carga, observar la tasa de solicitudes ayuda a entender cómo el rendimiento del sistema cambia en respuesta a diferentes volúmenes de tráfico. Un cambio significativo en la tasa de solicitudes podría indicar problemas de escalabilidad o de manejo de la concurrencia.

Errores

Similar al método USE, el seguimiento de errores es fundamental en el método RED. Esta métrica registra la cantidad y el tipo de errores que se producen mientras el servicio está bajo carga. Esto incluye errores de servidor como fallos 500, errores de cliente como solicitudes 400, y cualquier otra excepción que interrumpa el flujo normal de operaciones. Monitorear los errores durante pruebas de carga revela la robustez del sistema y su capacidad para manejar situaciones de error bajo presión.

Duración

También conocida como latencia, esta métrica mide el tiempo que tarda un servicio en completar una solicitud. En testing de carga, es crucial medir la duración de las solicitudes para evaluar la experiencia del usuario final. Un incremento en la latencia bajo carga puede indicar cuellos de botella en el procesamiento o en el acceso a recursos como bases de datos o servicios externos.

1.5.5. Buenas prácticas

Documentación

La documentación detallada en las pruebas de carga es crucial para asegurar la replicabilidad y el mantenimiento. Debe incluir los objetivos de la prueba, la configuración del entorno de pruebas, los escenarios probados, y las herramientas utilizadas. Debe ofrecer recomendaciones basadas en los hallazgos y mantener un historial de cambios. Esta documentación facilita la colaboración, la mejora continua y el cumplimiento de normativas, asegurando que las pruebas sean coherentes y efectivas. También sirven como recordatorio de los casos de uso y patrones que se han detectado como relevante para el usuario final.

Prueba temprano, prueba con frecuencia [7]

Las pruebas de rendimiento en algunos casos comienzan una vez el proyecto de desarrollo ha finalizado. No obstante, en los últimos años, obtener retroalimentación temprana durante el ciclo de vida del desarrollo ha demostrado ser sumamente valioso para identificar y solucionar problemas de manera rápida, incluso antes de que se manifiesten en entornos de producción.

La automatización y la incorporación temprana de pruebas de rendimiento en procesos de integración continua y desarrollo reflejan la madurez creciente de la industria. Esto permite a los equipos detectar cuellos de botella y problemas de rendimiento en etapas tempranas, facilitando su resolución y asegurando que el producto final esté mejor preparado para enfrentar escenarios de alta carga en producción.

Podemos apreciar en la figura 2 cómo el costo de los errores o ”bugs” incrementa mucho si estos se encuentran en fases más tardías del ciclo de desarrollo. En algunos casos, incluso, hay errores que nunca se solucionan por completo; simplemente se aplican parches que no abordan la causa raíz.

Un claro ejemplo de esto se encuentra en los videojuegos, donde hay errores recurrentes que nunca llegan a resolverse del todo. Sin embargo, al lanzar una segunda parte del juego, esos problemas suelen desaparecer porque al comenzar de cero es mucho más sencillo abordar la causa raíz de un error.

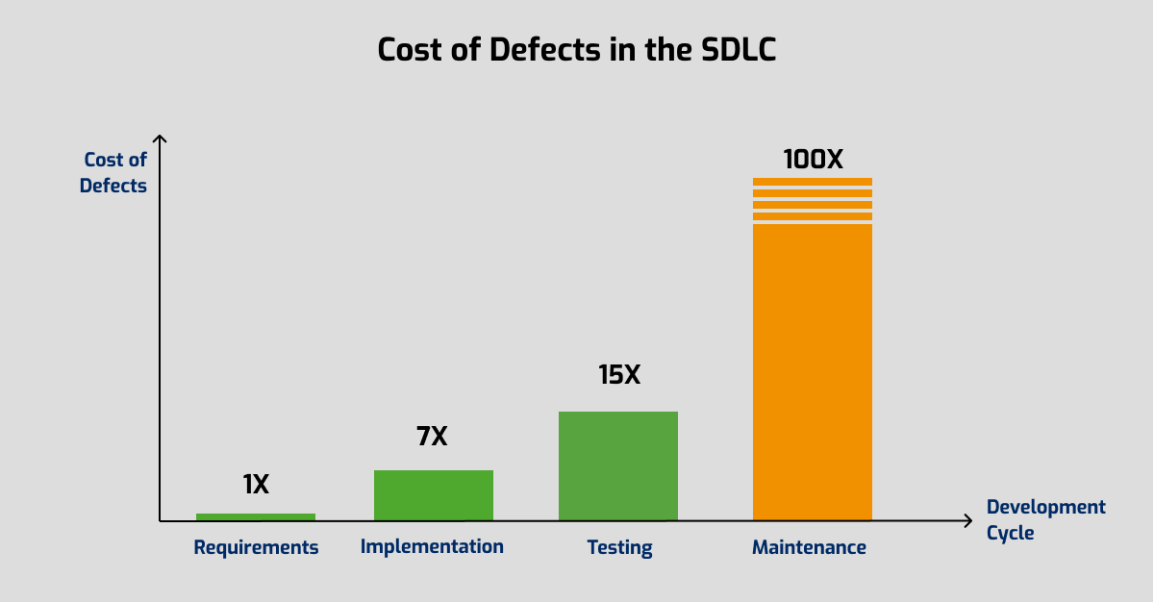


Figura 2: Costes de los defectos o errores en software [5]

Los datos importan

Lo ideal es usar datos reales para las pruebas de carga. Sin embargo, en muchos casos, esto no es factible debido a la falta de datos disponibles en el momento del desarrollo o a la inadecuación de los datos disponibles para el caso de uso a probar.

Cuando se habla de la calidad de los datos, a menudo se hace referencia al volumen de datos. El rendimiento general de la plataforma se degrada con un volumen creciente de datos, los listados se ralentizan el procesamiento de datos toma más tiempo etc. Con lo que es importante que el volumen de datos sea adecuado para cada prueba y caso de uso.

Por otro lado la calidad de datos también depende de la adecuación de los mismos para el escenario que queremos realizar, puede haber escenarios donde los datos tengan que tener unas propiedas o estructura específicas.

Tambíen, es esencial que los datos reflejen un uso completo y realista de la plataforma. Los datos generados en entornos de desarrollo no son adecuados para las pruebas de carga pues suelen ser repetitivos, ya que a menudo son creados siguiendo patrones similares debido a las preferencias de los desarrolladores. Esto puede conducir a una falta de diversidad en los escenarios de uso probados y, por ende, una falta de diversidad en los datos generados.

Por ello cada ves se usa más la generación sintética de los datos, en algunos casos combinando datos sintéticios con datos reales.

Entiende al usuario

Un test de carga que evalúe un escenario poco realista o irrelevante no tiene valor. Es fundamental conocer al usuario final, sus necesidades y sus patrones de comportamiento. Además, debemos comprender qué componentes de la plataforma son críticos para el funcionamiento general, de modo que las pruebas de carga se enfoquen en ellos.

Después de entender estas necesidades, hay un proceso de priorización donde se eligen los escenarios que desvelen más información y requieran menos trabajo, con el tiempo se debe ir incrementando el número de pruebas establecidas para cubrir más y más comportamientos y posibilidades.

El objetivo final es garantizar una experiencia de usuario buena para la mayor parte de los usuarios y en la mayor variedad de contextos posibles.

Implementación incremental

Como se ha mencionado previamente, es ideal implementar pruebas de carga de forma temprana. Sin embargo, también es importante que su implementación sea incremental, desarrollándose gradualmente a partir de las necesidades identificadas en el sistema o los puntos débiles de la plataforma.

Por lo tanto, las pruebas de carga deben crecer y evolucionar de manera natural, conforme aumenta el conocimiento sobre los patrones de comportamiento y uso del usuario final. Estas deben de reflejar nuestro entendimiento de los escenarios de uso reales que pueden afectar a nuestra plataforma.

1.5.6. Desafíos Actuales

Globalización y complejidad

A medida que la tecnología avanza y los sistemas se globalizan, se hace más evidente la necesidad de realizar pruebas de carga específicas y adaptadas a cada contexto particular.

Esta necesidad se ve agrabada con la adopción de arquitecturas de microservicios, las cuales presentan nuevos desafíos en la coordinación e interacción entre servicios. En esta coordinación se debe de conseguir el aislamiento de los servicios para su testeo de forma independiente.

Además, los usuarios acceden a los servicios desde una amplia variedad de dispositivos (móviles, tabletas, PCs) y plataformas (iOS, Android, Windows, macOS), lo que complica la simulación precisa del comportamiento del sistema y el rendimiento.

Todo esto combinado aumenta la complejidad de los escenarios a probar lo que premia una buena planificación de las pruebas y tener conocimento profundo del usuario que consume nuestros servicios.

Escalabilidad [6]

Escalar las pruebas de carga para aplicaciones desplegadas en la nube puede ser complejo, especialmente cuando se trata de servicios globales desplegados en múltiples regiones.

Volumen de datos

El volumen de datos generados por estas pruebas es cada vez mayor debiado a lo que hemos comentado de la complejidad creciente y también porque en general los sistemas cada vez generan más datos para adaptarse más al usuario.

Esta gran cantidad de datos necesita ser analizada de manera eficiente para extraer conclusiones útiles y tomar decisiones. Para ello, es imprescindible contar con herramientas avanzadas que permitan gestionar y procesar grandes volúmenes de datos. Algunas de estas herramientas incluyen sistemas de monitoreo en tiempo real, plataformas de análisis de big data y soluciones de inteligencia artificial. Esto también impulsa la necesidad de personal cualificado capaz de implementar las herramientas necesarias para realizar un análisis de los resultados.

1.5.7. Futuro de las pruebas de carga

Integración con Desarrollo continuo y CI/CD

Mirando hacia el futuro, se espera que la prueba de carga se encuentre aún más integrado con el desarrollo continuo y la entrega de software, adoptando prácticas de DevOps y CI/CD para facilitar pruebas más frecuentes y automatizados que aseguren la calidad y la escalabilidad del software desde las etapas más tempranas del desarrollo.

Código más desacoplado

Por otro lado, las pruebas de carga impulsan a la industria hacia un software más desacoplado. Esto se debe a que un software acoplado es más costoso de probar y dificulta o incluso imposibilita la prueba de módulos específicos de un servicio o plataforma. Por ello aunque los microservicios es un desafío a la hora de realizar pruebas también ayudan a realizar pruebas más independientes.

Desplazamiento a la izquierda

La implementación temprana de los test seguirá en su tendencia ascendente, hay incluso un término acuñado actualmente para ello referido como desplazamiento a la izquierda (shift left). Podemos visualizarlo en la figura 3 donde vemos que el volumen de los test es mayor en la fases de implementación y definición de requerimientos y el volumen baja al llegar a las fases de pruebas o mantenimiento, al contrario que en el modelo previo donde las pruebas están al final.

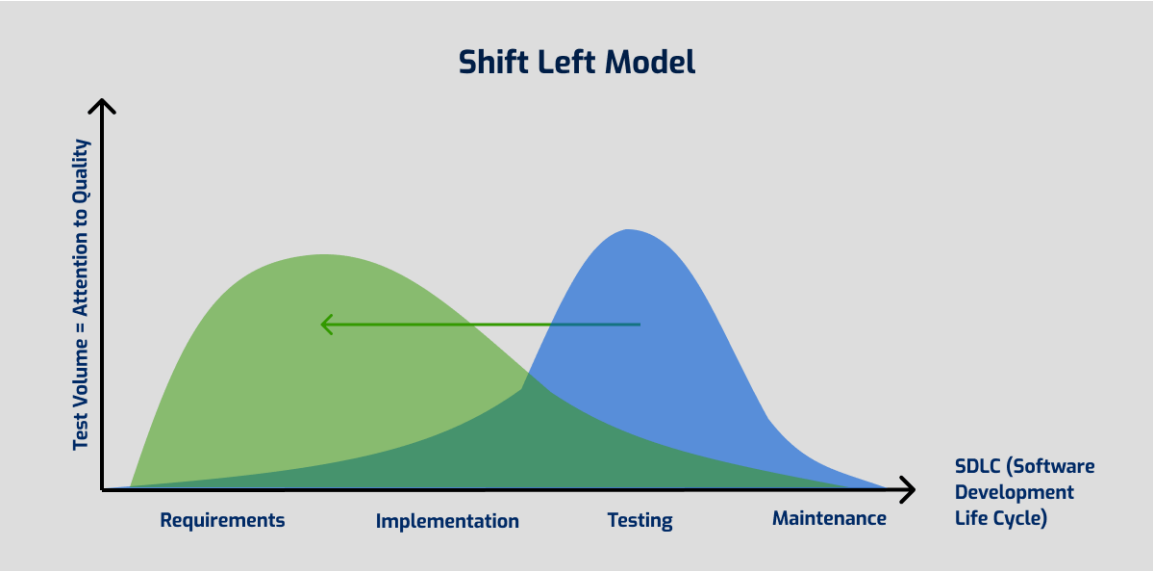


Figura 3: Modelo shift left [5]

Implementación IA

La integración de IA y machine learning ha comenzado a transformar las herramientas de pruebas de carga. Estas tecnologías ayudan a analizar patrones de tráfico, predecir puntos de fallo y proponer optimizaciones en los escenarios de prueba. También ayudan en la generación de datos y en la simulación del comportamiento del usuario.

1.6 Planificación general

En el siguiente cuadro 1 se puede ver las fases a realizar el en desarrollo completo del trabajo, con el tiempo estimado de duración de cada fase y las subtareas de la misma.

Fases	**Duración estimada**	**Tareas**
Estudio previo / Análisis	40	Tarea 1.1: Estudiar las mejores practicas para las pruebas de carga.
		Tarea 1.2: Adquirir conocimiento de las herramientas a utilizar en el proyecto (k6, Prometheus, Grafana).
		Tarea 1.3 Investigar y definir las métricas clave de rendimiento.
Diseño / Desarrollo / Implementación	170	Tarea 2.1: Diseñar un plan de pruebas de carga, definiendo los flujos a probar, las cargas de trabajo y los momentos donde se realizarán dichas pruebas.
		Tarea 2.2: Implementar las pruebas de carga utilizando la librería k6
		Tarea 2.3: Configurar Prometheus para recopilar las métricas necesarias durante las pruebas.
		Tarea 2.4: Implementar Grafana para visualizar los datos recolectados.
Evaluación / Validación / Prueba	50	Tarea 3.1: Validar y analizar los resultados de las pruebas de carga.
		Tarea 3.2: Identificar y documentar los problemas de rendimiento detectados.
		Tarea 3.3: Proponer y evaluar posibles soluciones para los problemas de rendimiento detectados.
Documentación / Presentación	40	Tarea 4.1: Documentar todo el proceso de análisis, diseño, implementación y evaluación.
		area 4.2: Preparar gráficos y tablas utilizando Grafana para ilustrar los hallazgos de las pruebas de carga.
		Tarea 4.3: Redactar una conclusión que resuma el trabajo realizado.
		Tarea 4.4: Preparar una presentación que incluya todos los puntos anteriores para la defensa del trabajo de fin de título.

Cuadro 1: Tabla fases, duración y tareas

El flujo de desarrollo del proyecto es como el que se ve en la siguiente imagen, primero hay un desarrollo base donde hay que implementar todo de forma que los sistemas funcionen para los objetivos que necesitaremos más adelante.

Después tenemos un desarrollo de cada una de las pruebas que queramos realizar para obtener información cada vez más concreta del rendimiento del sistema. Después hacemos un análisis de los resultados de las pruebas realizadas, con estos resultados decidiremos si necesitamos obtener más información o si haremos una mejora de rendimiento.

En grandes rasgos es un desarrollo iterativo donde en cada iteración iremos comprendiendo más el sistema y mejorando su rendimiento.

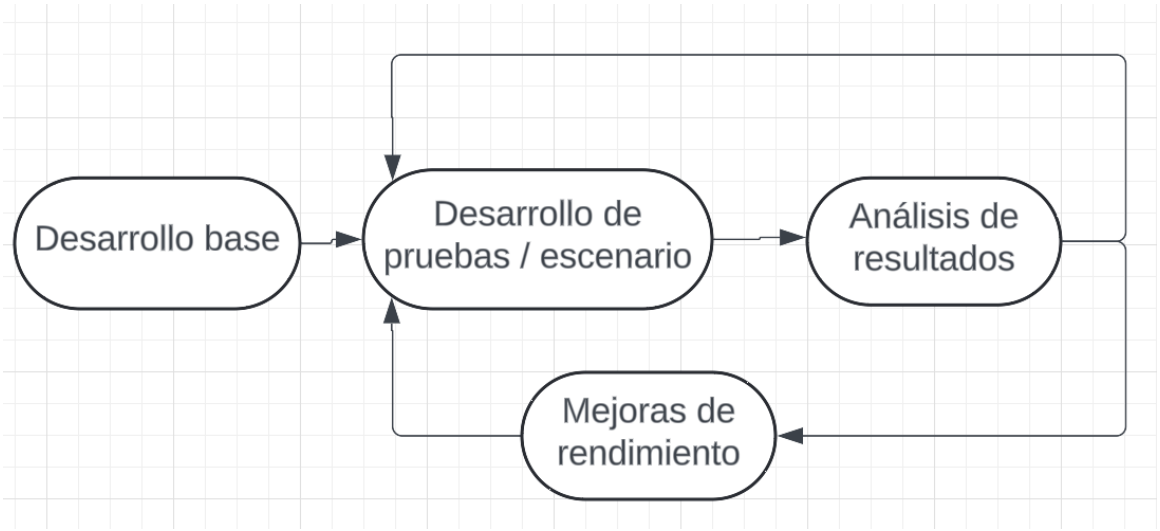


Figura 4: Flujo de desarrollo del proyecto

2 Desarrollo

2.1 Estructura del sistema y herramientas

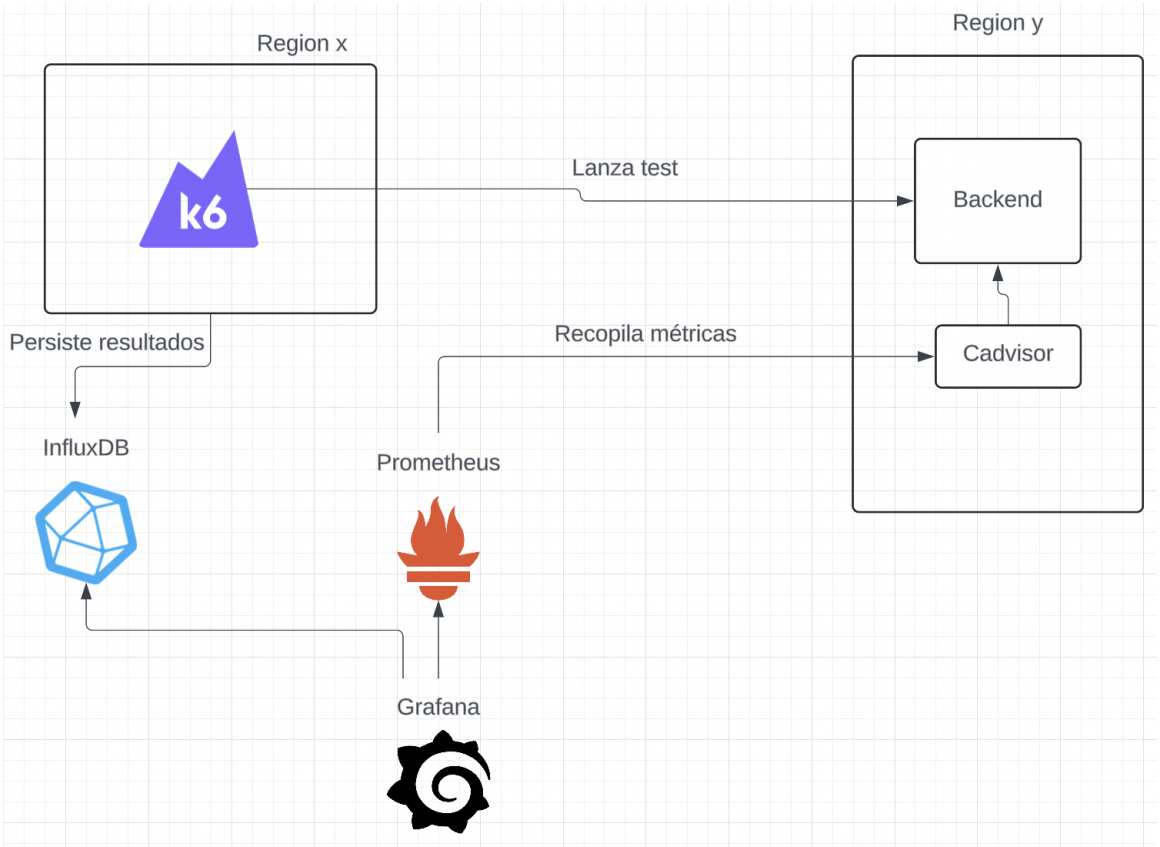


Figura 5: Esquema de la estructura del sistema

k6 [6]

Es una herramienta de código abierto diseñada para simplificar la ejecución de pruebas de carga y evaluaciones de rendimiento. Orientada a desarrolladores, esta herramienta soporta ejecuciones remotas y distribuidas, lo cual resulta ideal para la escalabilidad.

En nuestro caso k6 lanza los test contra el backend/api y hemos configurado k6 para que persista los resultados de las pruebas en InfluxDB.

Grafana [7]

Es una plataforma de visualización y análisis de datos de código abierto, ampliamente utilizada para monitorear servicios en tiempo real. Permite la creación de dashboards dinámicos que presentan gráficos y alertas a partir de múltiples fuentes de datos. Grafana destaca por su interfaz intuitiva, su capacidad de personalización y su integración con diversas fuentes de datos, facilitando así el análisis complejo y la toma de decisiones basada en datos.

En nuestro caso, Grafana lo usamos para los dashboards y usa como su fuente de datos a Prometheus e Influxdb.

Prometheus [8]

Prometheus es una herramienta de código abierto de monitoreo y alerta de código abierto diseñada para recopilar métricas en tiempo real de sistemas y aplicaciones. Además, formatea estos datos de forma que son facílmente analizados con consultas desde Grafana.

Las fuentes de datos que usa Prometheus en nuestro caso es Advisor.

cAdvisor [9]

Es un "exporter" [10] de Prometheus que extrae datos de rendimiento del hardware de una o más instancias de Docker. Prometheus usa exporters que se sitúan en el sistema en el cual queremos recuperar datos y los recuperan y formatean para después enviarlos a Prometheus.

En nuestro caso, cadvisor monitoreando y extrae información de instancias de PostgreSQL, un servicio de correo (mailer), RabbitMQ, PHP y Nginx.

InfluxDB [11]

InfluxDB es una base de datos de series temporales, optimizada para almacenar y gestionar datos que cambian con el tiempo. Se usa mucho para el monitoreo de métricas, ya sean métricas hardware sobre un servidor o sobre rendimiento. Lo cual es ideal para nuestro caso de uso con k6.

Backend

El backend/api se trata de un proyecto PHP usando el framework Symfony con base de datos relacional PostgreSQL.

Regiones

Se puede ver en el diagrama (figura 5) como k6 se encuentra en una región distinta del backend/API, esto lo realizamos con el objetivo de simular la latencia en un entorno de producción real. Si ambas instancias estuvieran en la misma región, no experimentaríamos casi latencia, lo cual es crucial para simular una interacción realista entre el usuario y la API.

2.2 Estructura del código

Lenguaje de Programación

El proyecto está desarrollado en TypeScript, aprovechando las numerosas ventajas que ofrece este lenguaje para mejorar la calidad y eficiencia del desarrollo de software. TypeScript proporciona un sistema de tipos estático que permite definir claramente los tipos de datos en el código ayudando a detectar errores antes de la ejecución, lo que resulta en un desarrollo más seguro y confiable.

El uso de TypeScript también mejora la legibilidad del código, ya que los tipos explícitos actúan como documentación viva, permitiendo a los desarrolladores entender rápidamente los datos esperados en funciones o variables. Además, TypeScript está bien soportado por los entornos de desarrollo integrados (IDEs), que proporcionan funcionalidades como autocompletado y refactorización automática, aumentando la productividad en el desarrollo.

Bases de Código

El sistema se compone de tres módulos:

- **Generador de Body:** Este módulo se encarga de la creación de los cuerpos de las solicitudes (body) para las pruebas. Se ha diseñado como un componente independiente para que pueda ser compartido entre el populador de datos y k6.
- **Populador de datos:** Este módulo tiene la función de poblar la base de datos con datos diseñados para probar los límites del sistema. Aunque su desarrollo ha sido uno de los aspectos más costosos del proyecto en términos de tiempo, es fundamental, ya que actualmente no disponemos de datos que simulan un escenario tan específico como el deseado, por ejemplo, una empresa de 1000 empleados generando datos durante seis meses. Este módulo permite configurar parámetros para ajustar cómo se generan los datos.
- **k6:** Este módulo se utiliza para lanzar las pruebas de carga.

Reutilización de Código y Escalabilidad

La utilización de k6 al ser una librería de JavaScript nos permite la reutilización del código del ”Generador de Body”, simplificando el trabajo, reduciendo la duplicidad del código y aumentando la escalabilidad del sistema.

2.3 Populador de datos

Antes que nada, el primer paso fue conseguir generar datos sintéticos, para ello fue necesaria la generación del "Generador de body". Como explicamos previamente, es un módulo que será reutilizable después para k6 en los test, pues en ambos casos, es decir, en el popular, la base de datos y las pruebas con k6, necesitamos realizar solicitudes a la API.

Podemos observar en la figura 6 que pasando unos parámetros, nos permite de forma simple crear el body que enviaremos para crear, en este caso, un vehículo.

```
export function createVehicle(companyId: number, trailerTypeWrapper: TrailerTypeWrapper): CreateVehicle {
    const faker = new FakerService();

    return new CreateVehicle(
        trailerTypeWrapper.getRandomTrailerTypeId(),
        faker.text.licensePlate(),
        companyId
    );
}
```

Figura 6: Código crear vehículo

Después con un simple JSON.stringify como se ve en la figura 7 podemos convertir la clase creada arriba CreateVehicle y transformarla en una string JSON, dicho JSON lo podemos ya enviar como body para la solicitud.

```
const vehicle = createVehicle(carrierCompany.id, trailerTypesWrapper);
const body = JSON.stringify(vehicle);

const resp = await request(
  '/vehicles',
  'POST',
  body,
  token
);
```

Figura 7: Código clase a json

El módulo "Generador de body" se ha generado principalmente con clases que encapsulan la información y permiten reusar dichas clases en puntos distintos, pues estas clases no solo contienen la información, sino que mediante métodos podemos realizar acciones con dicha información.

Por ejemplo, en la figura 8 podemos ver la clase TrailerTypeWrapper que contiene la información de los camiones (trailers). Tiene un método para conseguir un tipo de camión (trailer) aleatoriamente.

También nos aprovechamos de las ventajas de TypeScript no solo en la generación de clases e interfaces, sino también en el uso de tipos, como se observa en la figura 9 donde se define los tipos de paradas diferentes que existen. Al ser TypeScript, esto nos permite referenciar con el tipo en vez de un string.

```
class TrailerTypeWrapper{
    getRandomTrailerTypeId(): number{
        return this.trailerTypePayload[(
            Math.floor(Math.random() * this.trailerTypePayload.length))].id;
    }
```

Figura 8: Código TrailerTypeWrapper

```
export type StopType = 'PICKUP' | 'PICKUP_INITIAL' | 'DELIVERY' | 'DELIVERY_FINAL';
```

Figura 9: Código typescript type

Este componente, como hemos comentado antes, se encarga de popular la base de datos con datos sintéticos y realistas.

Usamos faker.js para la generación de datos aleatorios. faker.js es una librería de código abierto de JavaScript, que permite la generación de datos sintéticos. Tiene muchos módulos para cualquier necesidad y cada uno de ellos es extensamente configurable para obtener la información que deseemos.

Podemos ver a continuación cómo se usa el faker.js [12]. Tenemos un servicio FakerService que encapsula dentro la lógica de creación de datos aleatorios y se apoya a su vez de la librería de código abierto faker.js. Podemos ver en la figura 10 la obtención de la distinta información de localización. También apreciamos en la figura 11 el "commodity" o mercancía donde obtenemos el peso y el número de pallets, se puede ver que el peso se obtiene pidiendo un rango de números que es común para nuestro caso de uso.

```
class Location {
    public city(): string {
        return faker.location.city();
    }

    public lat(): number {
        return faker.location.latitude();
    }

    public long(): number {
        return faker.location.longitude();
    }

    public zipCode(): string {
        return faker.location.zipCode();
    }

    public address(): string {
        return faker.location.streetAddress();
    }
}
```

Figura 10: Código clase location

```
class Commodity {
    public name(): string {
        return faker.commerce.productName();
    }

    public weight(): number {
        return faker.number.int({min: 200, max: 1500});
    }

    public pallets(): number {
        return faker.number.int({min: 1, max: 40});
    }
}
```

Figura 11: Código clase commodity

El orden de la creación de datos es el siguiente:

1. Creación de usuarios
2. Creación de compañías con los documentos legales de las mismas
3. Creación de conductores y vehículos de las compañías
4. Creación de cargas, cada carga estará en un estado distinto del flujo de la misma, esto es deseable, pues a la hora de solicitar cargas en un cierto estado queremos que haya un número realista.

En la figura 12 se aprecia otro ejemplo de la aleatoriedad, vemos cómo en la primera línea se coge un número aleatorio entre 0 y 4 que se usa para el número de paradas adicionales que tendrá un recorrido.

```
let additionalStopsNumber = randomRange(0, 4);
let loadWrapper = await createLoad(tokenShipper, additionalStopsNumber, states, trailerTypes);
```

Figura 12: Código número de paradas aleatorio

Todo es configurable. Podemos cambiar el número de paradas promedio, el número de chats realizados por carga. Esto implica que este código no es estático, sino que evolucionará en función de las necesidades del momento y de la nueva información que obtengamos de los patrones de uso de la plataforma. También podemos tener dos generaciones de datos distintas para distintos test si fuera necesario.

Podemos observar en en anexo de código figura 48 un ejemplo completo con la creación del body el envio de la solicitud y control de errores.

2.4 Implementación k6

2.4.1. Configuración test k6

Configuración usuarios

En primer lugar, en un test, se define la cantidad de usuarios que queremos en cada fase del mismo.

Para entender la configuración debemos entender que todos los test empiezan con cero usuarios, después como se ve en figura 13 en cada línea definimos, por un lado, la duración de dicha fase y los usuarios finales al terminar dicha fase. El número de usuarios si cambia entre fases lo hace de forma lineal, es decir, si empieza en 0 le indicamos que llegue a 10 en 3 segundos, se incrementará el número de usuarios de 0 a 10 forma lineal durante esos 3 segundos.

Por ejemplo, en la figura 13 observamos inicialmente la instrucción `duration 150 s` y `target 50`. Como empiezan siempre con cero usuarios, el número de usuarios aumentará de 0 a 50 linealmente a lo largo de 150 segundos. La línea siguiente especifica que se mantendrá un total de 50 usuarios constantes durante 600 segundos, esto se debe a que no hay un cambio de usuarios entre fases. Finalmente, la cantidad de usuarios se reduce de nuevo a 0 en un periodo de 150 segundos. Al terminar, se espera a que los usuarios terminen su ejecución actual del escenario, aunque si tardan más de 30 s se para la ejecución de los mismos.

```
export let options = {
    stages: [
        { duration: "150s", target: 50 },
        { duration: "600s", target: 50 },
        { duration: "150", target: 0 },
    ]
};
```

Figura 13: Código configuración usuarios

En la figura 14 se observa una representación visual de lo descrito previamente, como incrementa el número de usuarios linealmente, después se establece en una “meseta” y finalmente se reduce de manera lineal hasta cero.

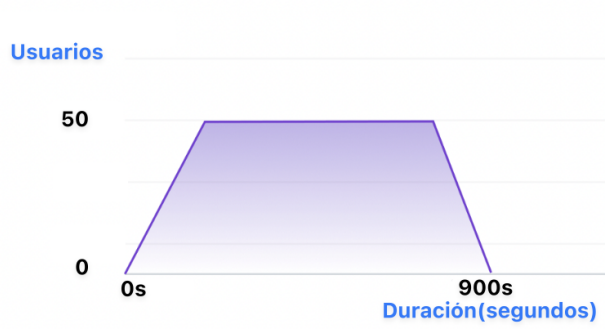


Figura 14: Gráfico ejemplo usuarios

Estructura código

Los usuarios virtuales a los que nos referimos ejecutan repetidamente una función o rutina como podemos ver en la figura 15 sería la función default, dicha rutina también es referida como el escenario. La ejecución de los usuarios es en paralelo y de forma independiente.

También observamos la función setup esta se ejecuta previa a empezar el test y se suele usar para importar archivos, ejecutar código compartido, etc. El objetivo de esta función es cargar información que será compartida por todos los usuarios en ejecución.

```
export function setup() {
    // Código compartido

    return { };// Data compartida
}

export default function (data: { }) {
    // Código a ejecutar en bucle
}
```

Figura 15: Código básico k6

Aleatoriedad

Hemos introducido un tiempo de espera aleatorio al final de cada escenario por dos razones. Primero, esto simula el comportamiento real de un usuario, quien se toma su tiempo entre acciones. Segundo, la variabilidad de estos tiempos de espera evita que todas las solicitudes se envíen simultáneamente, lo cual no refleja un escenario realista. Si el tiempo de espera no fuera aleatorio, podría darse el caso de que recibamos todas las solicitudes al mismo tiempo.

Si un escenario ejecuta múltiples acciones consecutivamente, es necesario insertar estos intervalos de espera entre cada acción. Por ejemplo, si se simula a un usuario que crea y luego modifica un recurso, habrá un lapso natural entre estas acciones que también debe ser aleatorio.

En escenarios más complejos, es útil introducir un grado mayor de aleatoriedad. Por ejemplo, en uno de nuestros escenarios se solicita una puja a un número variable de usuarios; este número puede ser tan pequeño como 3 o tan grande como 20. También podemos variar el flujo de acciones realizadas por el usuario para simular de forma más realista el comportamiento del usuario.

Podemos ver un escenario simple pero más completo en el anexo de código figura 47, con las disintas fases del k6 implementadas y aleatoriedad introducida.

2.4.2. Desarrollo de las pruebas en k6

Primero en el desarrollo hemos implementado un escenario principal donde se pruebe la plataforma entera.

Como ya hemos comentado antes, la plataforma a probar, Planimatik, su función principal es la creación de cargas (mercancía a transportar) y el control y monitoreo de las mismas desde su recogida a la entrega de la mercancía.

Como escenario principal haremos el flujo completo de creación de carga y las ramificaciones principales del mismo. La razón de empezar con este es que nos dará una idea general del rendimiento actual, además podemos extraer de él información de puntos donde pueda haber un cuello de botella y donde profundizaremos más adelante. También será usado este test como el estándar de rendimiento (benchmark) de la plataforma, pues prueba todos los componentes de la plataforma.

En este escenario haremos una prueba con un tiempo de ejecución promedio, de unos 35 mins. Empezaremos con 0 usuarios, se incrementarán durante 5 minutos hasta 50, después se mantendrá en 50 VUs durante 25 mins, por último, se reducirá durante 5 mins hasta 0. Esta prueba se llama prueba de carga promedio. Para nuestro caso es ideal, pues sin ser compleja es bastante informativa, con lo que es el tipo que usaremos para nuestras pruebas.

En la figura 16 se pueden ver los resultados en k6 de estas pruebas, pero no es suficiente, necesitamos tener una interfaz gráfica con Grafana, donde podamos visualizar todos los test realizados y además poder tratar los datos de forma correcta.

En los resultados podemos ver si las solicitudes han retornado el estatus que se espera de ellas, ya sea 200 o 201. También podemos observar los tiempos promedios, min, max, p(50) etc.

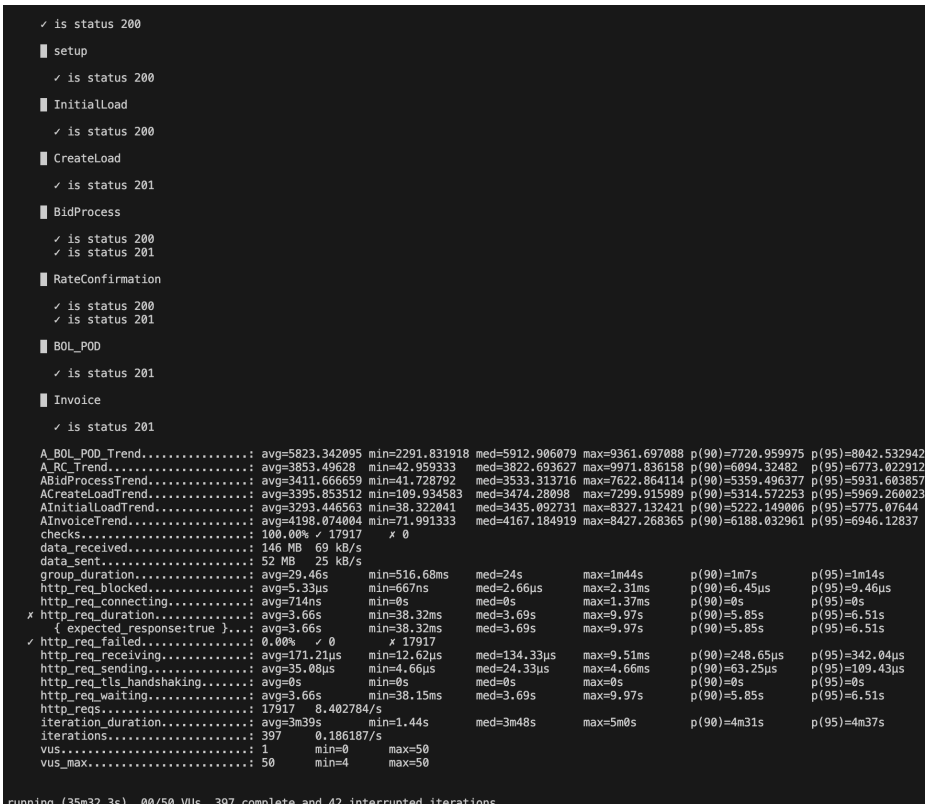


Figura 16: Output de k6

El output generado por k6 es interesante, pero incompleto si no tenemos además la información del uso de los recursos a nivel de hardware (CPU, memoria, red...). k6 también permite exportar a distintos formatos los datos, por lo que se puede analizar con diversas herramientas, no estamos limitados a Grafana.

2.5 Grafana dashboards

2.5.1. Configuración Prometheus - Grafana

La configuración de Prometheus para obtener la información es simple, en la figura 17 se observa de dónde obtenemos los datos. cAdvisor es un exporter de Prometheus que extrae la información del hardware de los dockers.

```yaml
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: "cadvisor"
    static_configs:
      - targets: ["cadvisor:8080"]
```

Figura 17: Prometheus configuración obtener datos

En Grafana, para configurar que use Prometheus, primero tenemos que definir un "data source".

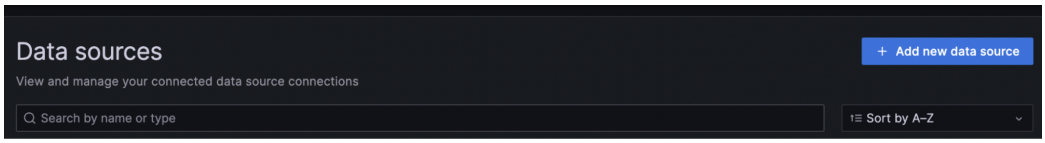


Figura 18: Grafana pantalla data sources

Después seleccionamos el tipo, en este caso Prometheus.

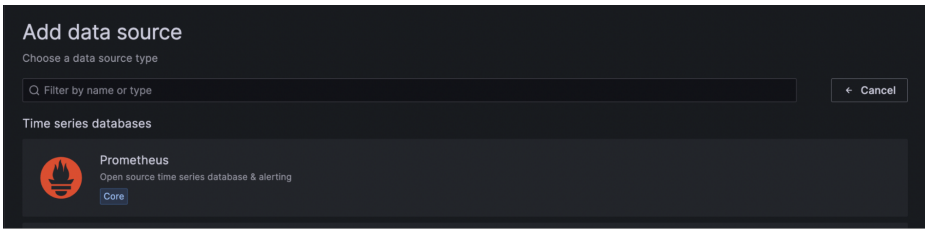


Figura 19: Grafana conectar prometheus

Por último, definimos un nombre y la conexión de Prometheus y ya podemos empezar a crear dashboards.

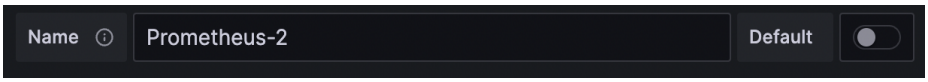


Figura 20: Grafana nombre conección

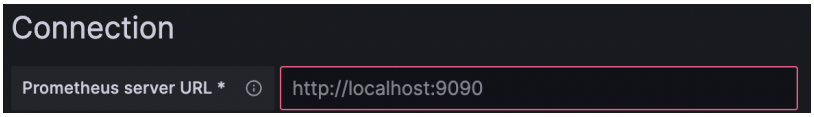


Figura 21: Grafana configurar conección

2.5.2. Grafana creación dashboard

Seleccionamos crear un dashboard y tenemos varias opciones, a continuación nos permite crear un dashboard desde cero o importar y modificar, en nuestro caso vamos a crear dashboards desde cero.

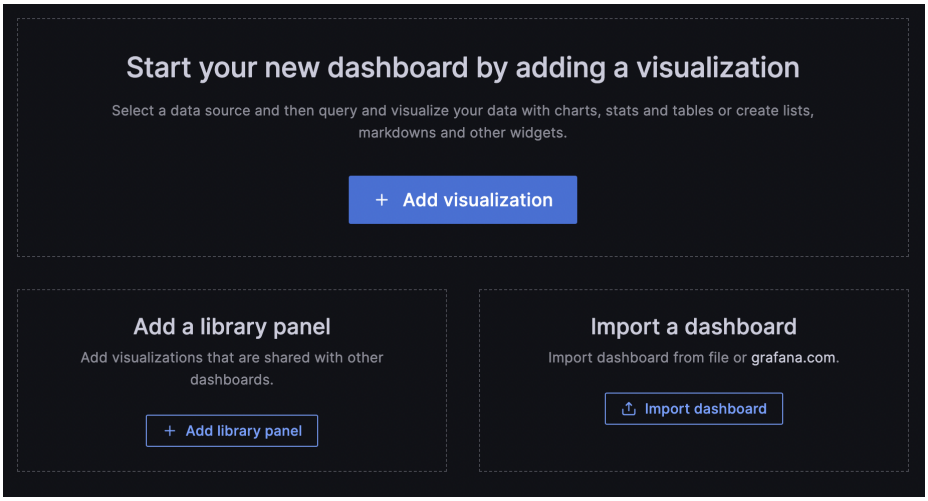


Figura 22: Grafana vista creación dashboard

Elegimos la fuente de los datos.

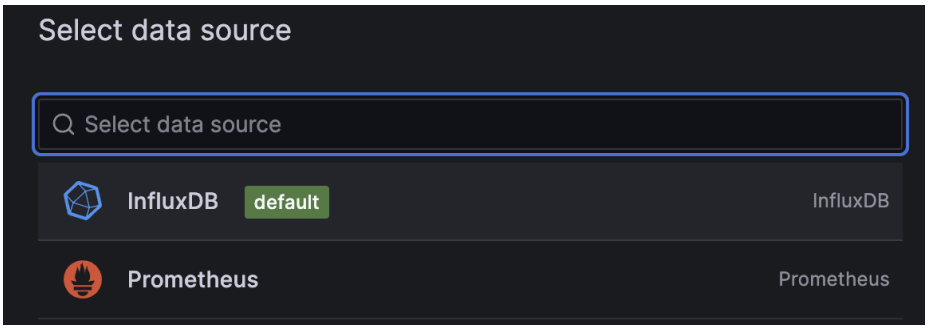


Figura 23: Grafana seleccionar fuente de datos creación dashboard

Después creamos el primer panel del dashboard seleccionando el tipo de visualización (figura 24) que queremos usar y añadimos una query que refleje los datos que queremos tomar (figura 25) y el tratamiento de los mismos.

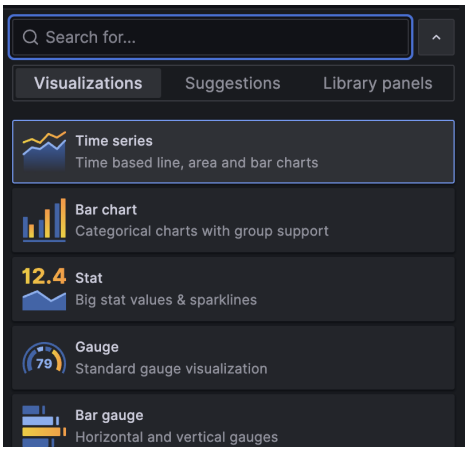


Figura 24: Grafana seleccionar panel

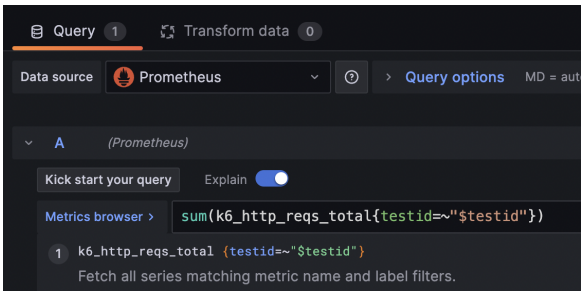


Figura 25: Grafana añadir query

Esto es una explicación superficial, Grafana permite mucha más configuración y detalles que no comentaremos en esta memoria.

2.5.3. k6 Grafana dashboard

Dashboard donde se muestra la información más importante generada por las pruebas de k6. Esta información ha sido tratada y presentada de forma que sea facilite la interpretación de los resultados.

Tener en cuenta que en esta sección explicaremos los paneles del dashboard con ejemplos, más adelante los usaremos para analizar el rendimiento en distintos escenarios.

Sección visión general

En la parte superior a la derecha (figura 26) tenemos un filtro de fechas (número 1) que afecta a todos los paneles del dashboard simultáneamente, pues todos los datos son temporales. A la izquierda (número 2) tenemos un filtro por test id donde podemos filtrar por id de la prueba para encontrar un test en particular rápido. Cada prueba tiene un id único, con lo que podemos usarlo para encontrar estos test más rápidamente.

También se puede apreciar la primera sección del dashboard (número 3), en este caso se llama visión general. Tenemos 3 líneas gráficas en dicho panel: RPS (solicitudes por segundo), VUs (usuarios virtuales), p90 (tiempo del percentil 90 % de las solicitudes en cuanto a tiempo). Con esto conseguimos un entendimiento inicial de la prueba, viendo de forma sencilla cómo escaló el número de solicitudes por segundo en relación con el aumento de usuarios, también se puede observar si se degrada el rendimiento con el tiempo, a su vez podemos ver si el tiempo promedio del 90 % de las solicitudes se degrada mucho a lo largo del tiempo.

Figura 26: Grafana ejemplo panel visión general panel

En el ejemplo de la figura 26 podemos ver que el número de RPS se mantuvo constante a lo largo de la prueba y no bajó con el tiempo, también se observa que el RPS creció junto con los usuarios lo que es positivo, vemos también que el p90 creció mucho con la subida de usuarios.

Sección rendimiento general

En el segundo panel (figura 27) vemos el número de solicitudes totales, el número de solicitudes que han fallado, las RPS promedias y el tiempo de respuesta promedio. Lo ideal es que el número de fallos sea cero, aunque con pruebas muy exigentes el sistema puede verse sobrepasado y empezar a fallar.

El panel del número de solicitudes que fallan tiene aplicado un color condicional, en caso de que falle alguna solicitud cambiará a rojo, el verde se muestra por defecto si no falla ninguna solicitud.

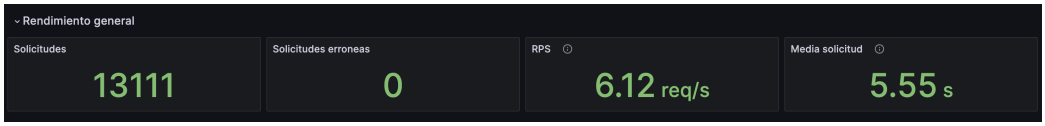


Figura 27: Grafana ejemplo sección rendimiento

Sección HTTP

Tenemos una sección llamada HTTP (figura 28) donde a la izquierda (número 1) hay un panel en formato tabla con las solicitudes e información sobre las mismas como tiempo promedio, tiempo mínimo, máximo, etc. Podemos clicar en las columnas de la tabla para ordenarlas por los distintos campos. Se muestra también el nombre de las mismas, hemos usado un alias en vez de la URL para poder saber de forma rápida de qué solicitud estamos hablando, pues una URL en muchos casos puede ser larga o difícil de comprender.

Tenemos un panel llamado Tiempos Latencia (número 2) donde se ven las distintas fases de las solicitudes HTTP por si queremos profundizar más en ese aspecto. Puede ser de utilidad si sospechamos que el tiempo de conexión o descarga puede ser el limitante. También tenemos otro gráfico (número 3) donde podemos ver por separado las solicitudes por segundo y los fallos por segundo a lo largo del tiempo, esto nos facilita en el caso de que hubiera errores saber en qué momento de la ejecución ocurrieron estos errores.

En la primera versión del dashboard el panel de las solicitudes no estaba, pero después de realizar algunas pruebas nos dimos cuenta de que era muy útil tener un panel de este tipo donde podamos ver en detalle el rendimiento por solicitud. Aunque hay que tener en cuenta que como las solicitudes se realizan en una misma ejecución, unas afectarán a otras, con lo que siempre hay que tomar con cautela estos resultados. Por ello más adelante profundizaremos con escenarios más concretos.

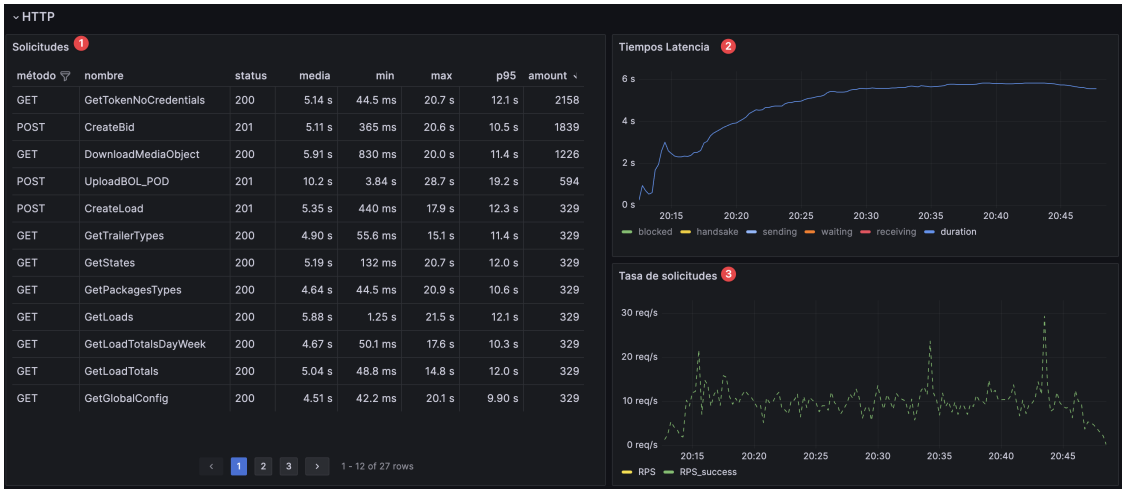


Figura 28: Grafana ejemplo HTTP panel

2.5.4. cAdvisor Grafana dashboard

El segundo dashboard generado es el que nos da información con respecto al uso del hardware de las distintas instancias de docker. Este dashboard en conjunción con el de k6 son necesarios para tener una visión completa. Uno puede responder si el cuello de botella es de memoria, CPU, etc. el otro puede responder qué solicitudes o flujos del sistema afectan más a esos recursos. Después con esta información podemos tomar decisiones en cuanto a mejora del hardware en algunos casos o mejora de la eficiencia de la API en otros casos, cada caso es distinto con lo que las decisiones de rendimiento deben de tenerse en cuenta en un contexto específico.

Tener en cuenta que en esta sección explicaremos los paneles del dashboard con ejemplos, más adelante los usaremos para analizar el rendimiento en distintos escenarios.

Sección CPU

En la parte superior del dashboard (figura 29) se encuentran los filtros (número 1) por fecha como en el dashboard previo, también podemos ver el filtro de host (número 2) donde podemos cambiar la instancia de cAdvisor, esto tiene utilidad en el caso de que introduzcamos distintos entornos de test. También tenemos un filtro en la cabecera para filtrar por el container de docker, aislando de esta forma la información de un container.

Primero tenemos el uso de CPU en el primer panel (número 3), vemos que en este ejemplo hay valores por encima de 100 % esto se debe a que tenemos 4 CPUs, por lo cual el máximo teórico es 400 %. A la derecha en el panel (número 4) podemos ver los picos máximos de CPU de cada container de docker y su promedio. Importante mantener bajo control el uso de CPU, pues es el más complicado y costoso de escalar, más que la memoria o el ancho de red.

Figura 29: Grafana ejemplo uso CPU panel

Sección memoria

A continuación en la figura 30, tenemos el uso de memoria y la memoria caché. Como en el anterior, a la derecha de cada panel tenemos un desglose por contenedor. No solo puede ser relevante si el sistema consume muchos recursos de memoria con un volumen alto de usuarios, también puede ser relevante en las pruebas con ejecuciones largas, pues podemos ver si el uso sube continuamente puede ser que el sistema no esté liberando la memoria correctamente, o se esté dejando los archivos temporales abiertos.

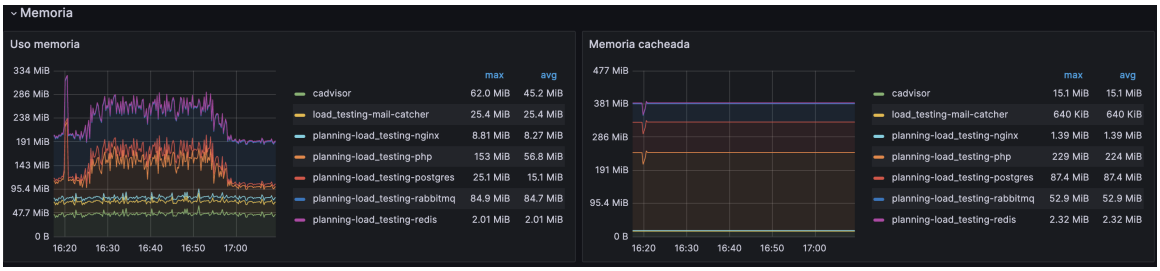


Figura 30: Grafana ejemplo uso Memoria panel

Sección tráfico de red

En el siguiente panel (figura 31) tenemos el tráfico de red recibido y el tráfico de red enviado. En algunos casos esto puede ser el limitante del sistema si hay una compartición de datos muy grande. Esto pasa en sistemas donde hay pocas solicitudes, pero que cada una de ellas traen mucha información, es mejor en la mayoría de casos tener más solicitudes y en cada caso conseguir la información que sea necesaria para la vista en la que nos encontremos. Es raro que el tráfico de red sea el limitante pues a día de hoy los tráficos de red proporcionados por los proveedores cloud son muy altos, pero estos paneles nos pueden informar de una transferencia excesiva de datos, no nos interesa que esto pase pues una tranferencia alta de datos afectará al tiempo final de la solictiud y afectará de manera desproporcionada a los usuarios con poca velocidad de internet.

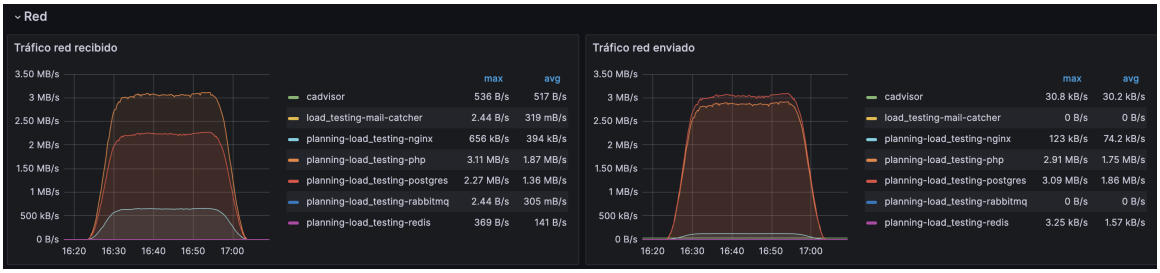


Figura 31: Grafana ejemplo uso red panel

2.6 Grafana anotaciones

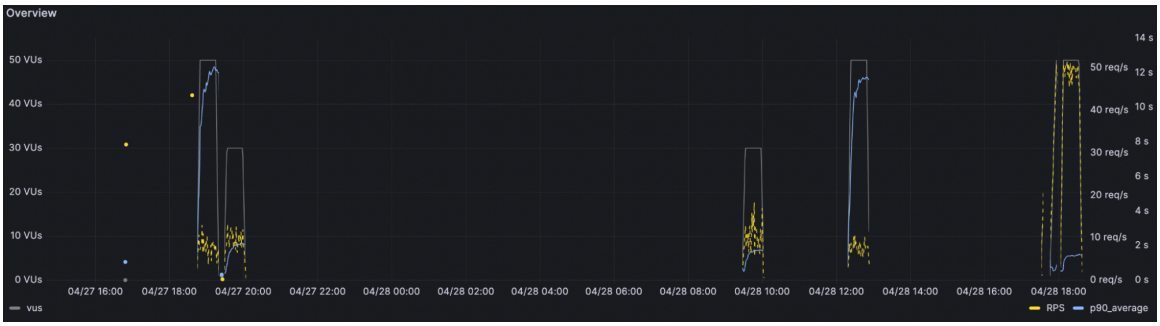


Figura 32: Grafana panel sin anotaciones

Cuando se han realizado muchos test como se observar en el dashboard (figura 32) y ponemos un periodo largo de tiempo es complicado entender con cuál prueba debemos comparar los resultados de la última realizada. Esto se ve agravado por múltiples factores, primero las pruebas pueden cambiar a lo largo del tiempo, segundo no todas las pruebas ejecutan los mismos escenarios, es decir, no prueban lo mismo o lo prueban con un volumen diferente y tercero lo ideal es comparar con el primer test que se realizo para el escenario que hemos ejecutado. Esto último se debe a que queremos ver si el rendimiento no ha bajado desde esa primera ejecución, pues puede pasar que el rendimiento se degrade lentamente y no nos demos cuenta si comparamos con la ejecución previa de dicho escenario.

Por ello, lo ideal es poder marcar las pruebas como se observa en la figura 33 hay una línea vertical azul punteada señalada con la flecha que indica información de cuál escenario se ejecutó en dicha prueba.

Figura 33: Grafana panel con anotaciones

A dichas anotaciones se les puede añadir etiquetas y después podemos filtrar por las mismas. Esto nos permite agrupar los distintos escenarios e incluso versionar los escenarios a medida que estos evolucionen con el tiempo.

También son interesantes para marcar cambios grandes en el sistema que puedan afectar al rendimiento o cambios en los propios test para no considerar los datos de ejecución previos.

2.7 Hardware usado

Para las pruebas usaremos un servidor de AWS EC2 [13]. AWS EC2 es el servicio de Amazon que permite contratar servidores de forma dinámica y escalarlos de forma sencilla en función del uso y de las necesidades. El servidor seleccionado tiene 2 GB de memoria, 4 CPUs y un tráfico de red máximo de 5 GB/s. El servidor es poco potente para los estándares actuales y no se asemeja a lo que tenemos en entornos de producción.

2.8 Pasos ejecución de las pruebas

El primer paso en las pruebas ejecuta el popular de datos para popular la base de datos con los datos sintéticos.

En el segundo paso se realiza una copia de seguridad de estos datos, la cual será utilizada más adelante.

Posteriormente, se ejecuta el escenario de prueba seleccionado y se espera el tiempo necesario para su finalización.

Una vez finalizada la prueba, es crucial restaurar la base de datos a su estado original utilizando la copia de seguridad previamente creada. Este paso de reversión es fundamental para mantener constante el punto de partida de las pruebas, asegurando así la replicabilidad y comparabilidad de los resultados. Si el estado o punto de partida de la base de datos es diferente en cada ejecución como es de esperar, la ejecución de las mismas se vería afectada y serían inconsistentes.

2.9 Resultados primeras pruebas

Esta primera prueba es el escenario que desarrollamos y comentamos antes en k6, ahora tendremos una visualización más completa de la ejecución de los test con ambos dashboards de Grafana.

Al ver la ejecución de la prueba (figura 34) podemos ver un incremento en el uso del CPU considerable desde el comienzo del test, aunque se mantuvo constante después a lo largo del test. Por otro lado, el incremento en el uso de memoria es leve, cabe recordar que el sistema tiene 2 GB de memoria RAM.

Figura 34: Grafana panel con cpu y memoria primera prueba

Estos paneles nos indican que aparentemente el limitante en nuestro sistema en cuanto a recursos hardware es el uso de CPU.

Observando el dashboard de k6 (figura 35), vemos que el tiempo promedio de las solicitudes es de 5.55 s y con un promedio de 6.13 solicitudes por segundo. También podemos observar que las solicitudes por segundo se mantienen estables durante la ejecución, con lo que el sistema no degrada su rendimiento en el transcurso del test. Aunque también podemos observar que el escalado de las RPS no ha sido completamente lineal con el incremento de usuarios, esto nos indica que el sistema está siendo limitado por algún recurso, como indicamos antes posiblemente la CPU.

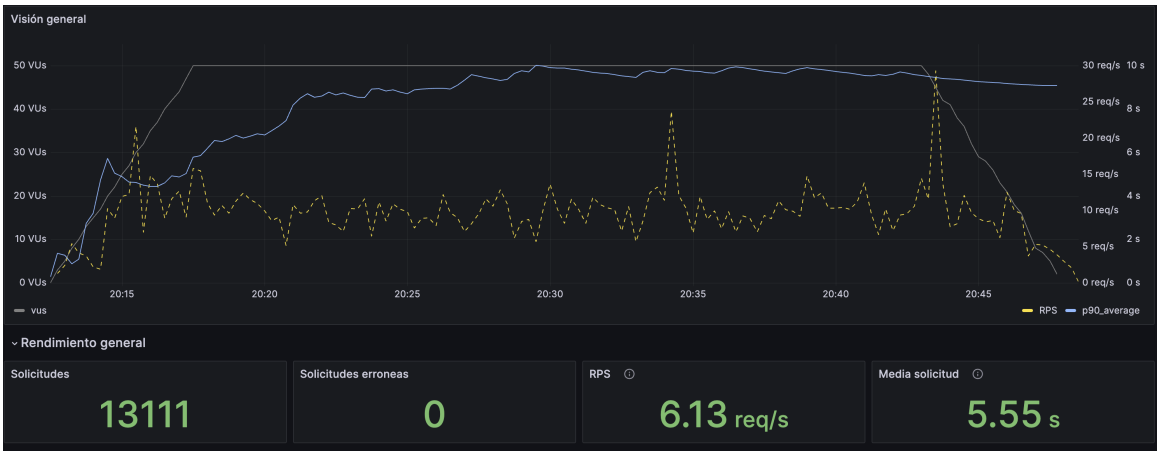


Figura 35: Grafana panel con visión general primera prueba

En la sección de solicitudes (figura 36), ordenamos por solicitudes más lentas en promedio, se observa que de las 4 más lentas en tiempo promedio son la subida de documentos. La quinta más lenta es la descarga de documentos.

Era de esperar que las solicitudes más lentas tengan que ver con los documentos, pues los servers no suelen ser muy eficientes en el manejo de archivos al tener que almacenar temporalmente el archivo en lo que se ejecuta la solicitud para después enviar el archivo a un almacenamiento en nuestro caso AWS S3 [14] y después tener que esperar por una respuesta, todo esto es poco eficiente en general. Igual pasa con la obtención de documentos, pues se descarga el documento desde S3 y se devuelve al usuario por la API, pero antes de devolverlo se tiene que hacer una copia temporal y esto es ineficiente también. Es decir, el caso de que nuestra API sirva de intermediario entre los recursos de archivos y el usuario final es en general poco eficiente.

Solicitudes

método	nombre	status	media ↓	min	max	p95	amount
POST	UploadBOL_POD	201	10.2 s	3.84 s	28.7 s	19.2 s	594
POST	UploadInvoiceDoc	201	9.58 s	3.14 s	25.4 s	17.8 s	290
POST	SingRC	201	8.24 s	3.10 s	18.6 s	15.0 s	304
PUT	FillConsignee	200	6.74 s	734 ms	21.1 s	12.7 s	310
GET	DownloadMediaObject	200	5.91 s	830 ms	20.0 s	11.4 s	1226
GET	GetLoads	200	5.88 s	1.25 s	21.5 s	12.1 s	329
POST	AcceptBid	201	5.73 s	663 ms	16.1 s	12.3 s	317
PUT	AcceptService	200	5.41 s	400 ms	18.4 s	10.9 s	311
POST	CreateLoad	201	5.35 s	440 ms	17.9 s	12.3 s	329
GET	GetStates	200	5.19 s	132 ms	20.7 s	12.0 s	329
GET	GetCompany	200	5.15 s	98.1 ms	15.2 s	9.28 s	310
GET	GetTokenNoCredentials	200	5.14 s	44.5 ms	20.7 s	12.1 s	2158

Figura 36: Grafana panel solicitudes primera prueba

2.10 Mejoras de rendimiento

2.10.1. Prueba datos estáticos

Análisis previo

Hemos sacado otra estadística que nos puede dar más información (figura 37), esta muestra la distribución de solicitudes por método HTTP y según una agrupación de solicitudes. Podemos ver que la mayoría de solicitudes son GET y que un 33 % de las solicitudes es la obtención de datos estáticos. Es importante apuntar que el listado de loads con un 15 % es relevante no solo porque su uso sea alto, sino porque es la primera pantalla de la plataforma. Todo gira alrededor de ella, el usuario se pasará un porcentaje importante del tiempo en esta pantalla.

Se observa también que la descarga de documentos es también bastante usada.

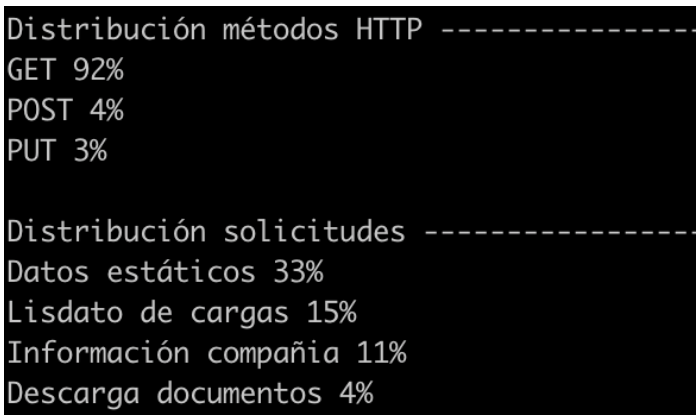


Figura 37: Estadística HTTP métodos y solicitudes distribución

Es importante considerar otro aspecto, los usuarios suelen entender y aceptar que las acciones relacionadas con la creación o actualización de recursos tomen más tiempo. Sin embargo, tienden a ser menos pacientes con las tareas que implican la recuperación de datos.

Estrategia

Tomando el análisis previo (figura 37), hemos decidido optimizar la obtención de datos estáticos. Debido a que los datos estáticos no cambian en valor de forma frecuente, estos son los más susceptibles a ser cacheados. También estos datos en nuestro caso solo tienen una vía para ser cambiados, con lo que es sencillo detectar cuándo cambian y, por tanto, quitarlos de la caché.

Hemos optimizado los datos estáticos mediante el uso de Redis [15] como sistema de caché.

Caché uso e implementación

Redis es una base de datos en memoria que frecuentemente se utiliza como caché. Esta base de datos almacena información en pares clave-valor, lo cual la hace extremadamente rápida. Gracias a su arquitectura en memoria, Redis puede retornar datos en tiempos inferiores al milisegundo, un tiempo significativamente menor al de las bases de datos convencionales, mejorando así el rendimiento de las aplicaciones de manera considerable. Además, Redis no utiliza SQL para recuperar los datos; en su lugar, se obtiene la información directamente con la clave que apunta al valor, lo que resulta en un uso bajo de los recursos de CPU.

Para ilustrar el funcionamiento del sistema de clave-valor mencionado anteriormente, consideremos un ejemplo. Supongamos que deseamos almacenar en caché los usuarios de cada compañía dado que estos no cambian con frecuencia. Para lograrlo, al acceder a estos datos por primera vez, los almacenamos en caché utilizando una clave, como 'users-idCompañia', esta clave apunta a los datos o valor en formato JSON. Vemos como no hay SQL simplemente solicitamos el dato directamente, lo que es mucho más rápido.

Obviamente, esto no quiere decir que el uso de bases de datos con SQL no tenga sentido, por lo general el uso de caché es menos flexible y limitado y nos sirve para casos con datos donde no haya muchas modificaciones.

Como es de esperar de una base de datos destinada a la caché, Redis ofrece herramientas para definir políticas de invalidación de datos. La invalidación de datos en caché es simplemente borrar los datos de la caché, estos volverán a ser cacheados en la siguiente solicitud que los requiera. Estas políticas pueden estar basadas en el uso de los datos o en el tiempo transcurrido desde que se almacenan en caché. Obviamente, si los datos subyacentes cambian podemos y debemos invalidar la caché que guarda su valor.

Podemos ver a continuación en la figura 38 que sencillo puede ser cachear datos. En el siguiente código lo que se puede ver es que se hace una solicitud a la caché requiriendo los `states`, si esta información estuviera cacheada, pues se asignan a la variable `data` y no habría que hacer nada más. En cambio, si no se encuentra en la caché se ejecuta el código de dentro, este código busca los `states` sin usar la caché y los retorna, una vez retornados el gestor de caché, en nuestro caso Redis se encarga de guardar ese dato en la caché con lo que la siguiente vez ya estaría cacheado.

En otras palabras, buscamos primero en la caché, si no la encontramos en ella, ejecutamos un código que obtiene la información no cacheada y lo guardamos en la caché para las siguientes ocasiones.

```php
$data = $this->cacheCustomRedis->get('states', function (ItemInterface $item) use ($request): string {
    $context['groups'] = $request->attributes->get( key: '_api_operation')->getNormalizationContext()['groups'];

    return $this->serializer->serialize($this->stateRepository->findAll(), format: 'json', $context);
});
return new Response($data);
```

Figura 38: Código ejemplo uso de caché

En nuestro caso, la invalidación se realiza cuando los datos subyacentes cambian. En el propio código encargado de la actualización (figura 39) de estos datos, también manejamos la lógica de invalidación. En dicho fragmento de código, el bucle `foreach` se encarga de actualizar los tipos de trailers, mientras que la función `invalidateTags` se encarga de eliminar los datos de la caché que se han vuelto obsoletos debido a estas actualizaciones. El proceso después de invalidar datos es el que explicamos arriba, la primera vez los datos no están cacheados, por lo que se obtiene sin caché dichos datos y a partir de esa vez ya estarían cacheados.

```php
public function populateTrailerTypes(ObjectManager $manager): void   fernandomarcelo, 28/4/23, 13:51 • Create commando to populate big texts database
{
    foreach (TrailerTypeResource::TRAILER_TYPES as $data) {
        $trailerType = $this->trailerTypeFactoryFixture->createTrailerType($data['code'], $data['description'], $data['type'], $data['sequence']);
        $manager->persist($trailerType);
    }
    $this->cacheCustomRedis->invalidateTags(['TrailerTypes']);
}
```

Figura 39: Código ejemplo invalidación de caché

También usamos una política de caché pasiva la cual si la caché se rellenara borrarías datos en función de los que hayan sido accedidos menos frecuentemente, dejando en la caché los datos más accedidos, aunque en nuestro caso por ahora el límite de memoria de la caché es más que suficiente para lo que queremos cachear. Aunque nunca está mal definir estas políticas para el futuro.

Resultados

Para comprobar la mejora de rendimiento hemos realizado un nuevo escenario, el cual ejecuta las solicitudes de datos estáticos. Hemos optado por esta opción para aislar estas solicitudes y poder analizar mejor la mejora de rendimiento local.

Podemos ver un incremento sustancial del rendimiento. Podemos ver una comparativa de la mejora con los nuevos cambios en el cuadro 2. La mejora en tiempo promedio es lo que más ha mejorado en un 59 %. A pesar de que el tiempo inicial es muy rápido, 340 ms, estas solicitudes se realizan mucho en la plataforma, con lo que reducir su tiempo y uso de recursos es muy relevante, pues puede afectar al tiempo de respuesta de otras solicitudes. En este caso, liberan a PostgreSQL de tener que retornar continuamente la información, además de liberar a PHP y Nginx, pues al tardar menos en ejecutarse los hilos no se mantienen ocupados durante tanto tiempo.

El número de solicitudes por segundo no ha mejorado tanto 8.5 %, pero esto era de esperar, esto se debe a la naturaleza del propio test donde se introduce entre solicitudes un tiempo de espera de unos 8-15 s simulando el comportamiento del usuario, esto significa que hay un límite de cuantas solicitudes se pueden realizar con el número de usuarios seleccionado.

Métrica	**Previo**	**Actual**	**Mejora** (%)
Solicitudes por segundo (RPS)	43.6	47.3	8.5 %
Tiempo promedio	0.827 s	0.340 s	59 %

Cuadro 2: Comparativa resultados URL firmada vs no URL firmada

Podemos ver también que el sistema está menos cargado(figura 40 vs figura 41) en cuanto a CPU el máximo de uso de CPU que llega sin las mejoras es de 300 % en cambio, con las mejoras es de un 250 %, lo que nos indica que la optimización no solo permite tiempos de respuesta más rápidos, sino que podemos manejar más usuarios con un menor uso de CPU del sistema y mejor tiempo de respuesta.

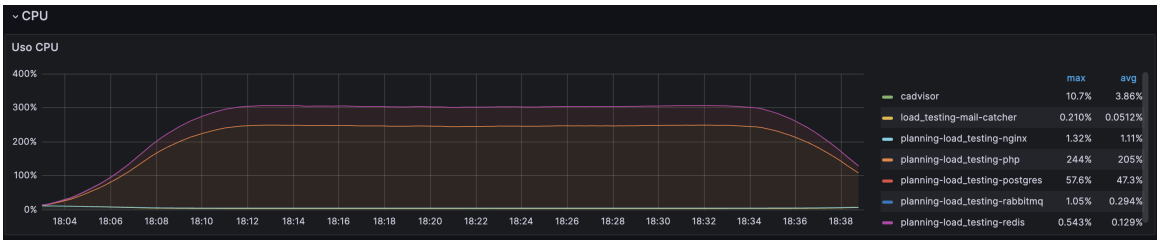


Figura 40: Grafana datos estáticos dashboard hardware sin caché

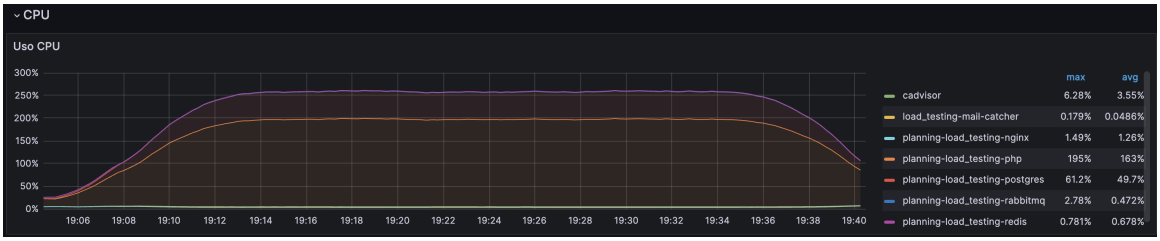


Figura 41: Grafana datos estáticos dashboard hardware con caché

2.10.2. Obtención de documentos

Análisis previo

En la gestión de documentos en nuestra plataforma, incluyendo el almacenamiento y recuperación, se emplea el servicio de Amazon S3. Hasta ahora, la recuperación de un documento se efectuaba de la siguiente manera: la solicitud de obtener el recurso se realiza a la API, la cual solicita el documento a Amazon S3 y finalmente la API lo retorna al usuario. Este método presenta inconvenientes significativos en términos de rendimiento, dado que el servidor al hacer de intermediario debe almacenar temporalmente el documento antes de su devolución al usuario, lo cual reduce la eficiencia. Tampoco ayuda el hecho de que se descargue dos veces del documento, una realizada por la API y la segunda por el usuario final.

El almacenamiento temporal en el server antes de su devolución es lo que más afecta al rendimiento de la plataforma esto se debe a que el server es de propósito general es decir no está especializado en el manejo de archivos específicamente, por ello lo ideal es delegar lo máximo posible en el server en cuanto al manejo de archivos.

Estrategia

Para superar esta limitación, se ha explorado una alternativa en la cual la API no retorna directamente el documento, sino que retorna un enlace temporal al mismo. Este enlace es un recurso que nos proporciona S3 llamado URL firmada [16] o en inglés "signed URL", permite la descarga de recursos pudiendo indicar una fecha de expiración para la URL y restringe el acceso exclusivamente al recurso solicitado, sin posibilidad de utilizarlo para acceder a otros archivos. Esta técnica, además de segura, optimiza considerablemente el proceso de transferencia de documentos.

Bajo este nuevo esquema, el proceso se inicia con la solicitud de la URL a la API. La API solicita la URL firmada a Amazon S3 y la entrega al usuario. Luego, el usuario o el frontend realiza la descarga directa del recurso utilizando la URL proporcionada. Aunque el proceso pueda parecer más complejo, es sustancialmente más rápido y directo. Como hemos comentado previamente, es mejor delegar lo máximo posible en los servers dedicados al manejo de archivos, en nuestro caso a Amazon S3.

A continuación, en la figura 42 se observa que la implementación de la obtención de la URL firmada, con el `path` del recurso obtenemos dicha URL y podemos ver que le configuramos un tiempo de expiración de 1 hora.

```php
public function getPreSignedURL(string $path): string
{
    $expiration = '+1 hour';
    $cmd = $this->client->getCommand( name: 'GetObject', [
        'Bucket' =>  $this->bucket,
        'Key' => $path,
    ]);

    $request = $this->client->createPresignedRequest($cmd, $expiration);
    return (string)$request->getUri();
}
```

Figura 42: Código URL firmada

Resultados

Hemos hecho un nuevo escenario para la descarga de documentos. En este escenario se obtienen distintos archivos del sistema con formatos variados y tamaños diversos.

A continuación en el cuadro 3 podemos ver los resultados comparativos. Se puede ver una mejora muy sustancial en el tiempo promedio por solicitud con una mejora del 95 % y el número de solicitudes por segundo con una mejora de un 100 %. En otras palabras, el sistema ha reducido en un 95 % el tiempo de solicitud y simultánemente se ha duplicado el número de solicitudes realizadas. Esto afectará a todo el sistema de forma positiva.

Métrica	**Previo**	**Actual**	**Mejora** (%)
Solicitudes por segundo (RPS)	4.32	8.61	100 %
Tiempo promedio	5.01 s	217 ms	95 %

Cuadro 3: Comparativa resultados URL firmada vs no URL firmada

El uso de CPU y memoria es un poquito mayor (cuadro 4), ambas cosas no son preocupantes, pues estamos manejando el doble de solicitudes como indicamos antes. El uso del tráfico de red es mucho menor, tanto de envío como de recogida como es de esperar. Esto afectará de forma positiva a aquellos usuarios que tengan una velocidad de red lenta, pues la transferencia de datos es menor y más rápida.

Métrica	**Previo**	**Actual**	**Mejora** (%)
Uso de CPU promedio	39.7 %	45.6 %	-15 %
Uso de memoria promedio	196 MiB	179 MiB	9 %
Memoria en caché promedio	91 MiB	157 MiB	-72 %
Tráfico de red recibido promedio	603 KB/s	102 KB/s	83 %
Tráfico de red enviado promedio	605 KB/s	143 KB/s	76 %

Cuadro 4: Comparativa resultados URL firmada vs no URL firmada (Corrección)

2.10.3. Configuración

Estrategia

Partiendo de la información obtenida en la primera prueba de que el cuello de botella no se encuentra en el uso de memoria, hemos decidido probar a ajustar algunas configuraciones que usen un uso de memoria mayor para analizar si esto repercute positivamente en el rendimiento general.

Después de diversas pruebas hemos optado por cambiar la configuración de opcache, opcache es un módulo de PHP que se encarga de cachear todo el código del proyecto. Después de varias pruebas, hemos llegado a los siguientes valores como los óptimos (figura 43): el uso de memoria de opcache lo hemos incrementado de 128 MB a 256 MB y el máximo de archivos cacheado de 10k a 20k.

```
5   session.cookie_httponly = true
6   opcache.memory_consumption = 256
7   opcache.max_accelerated_files = 20000
```

Figura 43: Código ajustes de configuración

Resultados

Realizando estos cambios y lanzando el escenario de pruebas inicial obtenemos los siguientes resultados, como se puede ver en el cuadro 5, el tiempo de solicitud promedio ha bajado desde 5.55 s a 5.27 s con un 5 % de mejora, además ha subido las solicitudes por segundo de 6.13 a 6.39 con un 4 % de mejora.

Métrica	**Previo**	**Actual**	**Mejora** (%)
Solicitudes por segundo (RPS)	6.13	6.39	4 %
Tiempo promedio	5.55 s	5.27 s	5 %

Cuadro 5: Comparativa resultados configuración con prueba base

Viendo los resultados del rendimiento del hardware(cuadro 6) observamos que el uso de CPU ha mejorado un 10 % lo que es un buen indicador teniendo en cuenta que es nuestro limitante. El uso de memoria, como era de esperar ha empeorado, en cambio, el tráfico de red se ha mantenido casi igual.

Métrica	**Previo**	**Actual**	**Mejora** (%)
Uso de CPU promedio	107.7 %	97.3 %	10 %
Uso de memoria promedio	258 MB	265 MB	-3 %
Memoria en caché promedio	296 MB	411 MB	-39 %
Tráfico de red recibido promedio	3.39 MB/s	3.5 MB/s	- 3 %
Tráfico de red enviado promedio	2.82 MB/s	2.95 MB/s	- 5 %

Cuadro 6: Comparativa resultados URL firmada vs no URL firmada (Corrección)

Los resultados son positivos, pues usando un poco más de memoria que es un recurso el cual no nos hemos visto limitados hemos conseguido incrementar la eficiencia del sistema en cuanto a tiempos de respuesta y solicitudes por segundo además de mejorar el uso de CPU.

2.11 Prueba final

Última prueba donde aplicaremos todas las mejoras de rendimiento, la configuración, la caché de Redis, y la de descarga de datos con las URLs firmadas. Es decir, todos los ajustes de rendimiento previamente probados de forma independiente.

Resultados k6

Podemos observar la ejecución de los resultados en la figura 44, en el cuadro 7 los podemos comparar con los resultados iniciales el tiempo promedio de respuesta de las solicitudes es un 25 % más rápido y la cantidad de solicitudes procesadas por segundo es un 25 % mayor. Es crucial señalar que un incremento del 25 % en ambas métricas no significa necesariamente una mejora del 25 % en el rendimiento general del sistema. De hecho, la mejora es mayor, ya que el sistema muestra un rendimiento un 25 % superior manejando un 25 % más de solicitudes/usuarios. Esto implica que el sistema mejora tanto en el tiempo de respuesta como en la cantidad de usuarios que puede manejar simultáneamente.

Figura 44: Grafana prueba final overview dashboard

Métrica	**Previo**	**Actual**	**Mejora** (%)
Solicitudes por segundo (RPS)	6.16	7.71	25 %
Tiempo promedio	5.5 s	4.14 s	25 %

Cuadro 7: Comparativa últimos resultados con primeros

Resultados Hardware

En la figura 45 podemos ver a nivel de hardware, si lo comparamos con la primera prueba (cuadro 8) vemos un uso de CPU casi igual lo cual es positivo, pues estamos manejando tiempos de respuesta menores y más usuarios con el mismo uso de CPU. El uso de memoria es igual y un 12 % superior en cuanto a la memoria cacheada, lo cual era de esperar, pues hemos usado técnicas para mejorar el rendimiento que usan más recursos en cuanto a memoria.

Figura 45: Grafana prueba final hardware dashboard

Métrica	**Previo**	**Actual**	**Mejora** (%)
Uso CPU promedio	110 %	115 %	-4 %
Uso memoria promedio	262 MB	260 MB	0 %
Memoria cacheada promedio	331 MB	290 MB	-12 %

Cuadro 8: Comparativa hardware resultados configuración con prueba base

Después de todos los ajustes de rendimiento podemos concluir que el limitante sigue siendo el uso de CPU y el uso de memoria sigue estando bastante bajo.

Esta es una plataforma relativamente grande ahora mismo, con lo que la mejora del rendimiento general de la plataforma es bastante positiva, pues tratando un número reducido de solicitudes críticas hemos mejorado el rendimiento general. Además, hay flujos de la misma, como hemos indicado antes, donde la mejora de rendimiento es aún mayor.

2.12 Test establecidos

Una vez realizados los diversos test y escenarios comentados previamente, hay que establecer qué test realizaremos de forma periódica para asegurarnos de que el rendimiento no se degrada a lo largo del tiempo con los cambios en el código.

En el proyecto hemos decidido realizar una prueba de humo en cada subida a los entornos en el propio pipeline del CI/CD. La prueba de humo es una prueba que suele tomar entre 30 segundos a 2 minutos y tiene como objetivo comprobar que todo funciona correctamente sin profundizar mucho. Esto es perfecto, pues por el poco tiempo que toma no nos retrasa en el desarrollo y nos sirve como una validación rápida. En nuestro caso la prueba durará unos 30-45 s y se usarán unos 30 usuarios.

Se observa en la figura 46 que una de las etapas del propio CI/CD es la del test de carga, es la última etapa del propio CI/CD, pues que el entorno se debe de encontrar en pie para poder realizar las pruebas.

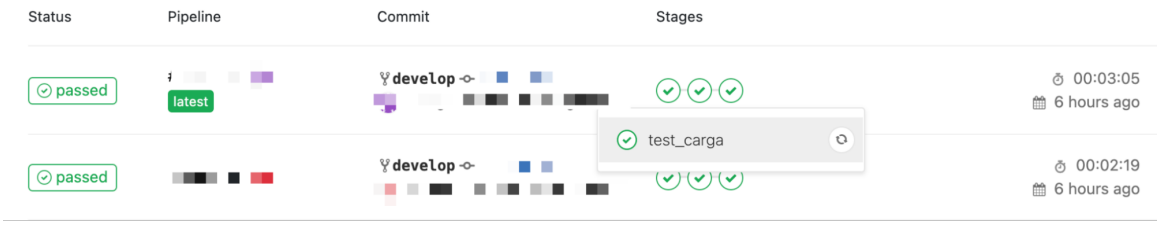


Figura 46: Imagen CI/CD

También tenemos una prueba de carga promedio automatizada para ejecutarse en horarios nocturnos, esta toma unos 45 mins y lo que nos permite es profundizar más en el rendimiento general y tener retroalimentación de que estamos en la dirección correcta. Se realiza en horarios nocturnos para que no afecte al desarrollo, teniendo en cuenta que son 45 mins en total.

2.13 Propuestas a futuro

Optimización con AWS Lambda [17]

Una función lambda es código que se ejecuta serverless, es decir, a demanda, con lo que no hay que mantener un server corriendo continuamente, lo que es más barato, además escala de forma eficiente.

En la mayoría de casos, la subida de documentos genera 2 miniaturas del archivo subido. Estas miniaturas las genera la propia API. Para agilizar la subida de documentos, se puede implementar una función Lambda que se encargue de generar las miniaturas y subirlas asíncronamente. Esto permite que el sistema solo tenga que gestionar la subida del archivo principal, mientras que las miniaturas se procesan en paralelo por un servidor a demanda, optimizando el tiempo de respuesta.

Optimización mediante AWS Signed URLs [16]

Para reducir aún más la carga en el sistema durante la subida de documentos, se podría implementar el uso de URLs firmadas de AWS. En este caso, las URLs firmadas son para subir documentos. Esta técnica, utilizada extensamente por grandes plataformas, permite que el frontend suba los documentos directamente a AWS S3 sin pasar por nuestra API, lo cual reduce significativamente el consumo de recursos. Lo único para lo que necesitaría la API es para solicitar la URL firmada que le permite a su vez subir los documentos, lo cual en cuanto a rendimiento es muy rápido.

No obstante, esta estrategia requiere precauciones para evitar abusos, como la interceptación de la URL firmada por usuarios no autorizados, quienes podrían utilizarla para subir archivos no deseados y potencialmente grandes, saturando el espacio de almacenamiento S3. Para mitigar este riesgo, se pueden aplicar restricciones a las URLs firmadas, limitando el formato y tamaño de los archivos, y estableciendo una fecha de caducidad para su validez de uso.

Adicionalmente, es recomendable implementar controles de autenticación y autorización rigurosos para asegurar que solo los usuarios autorizados puedan obtener las URLs firmadas. Esto, junto con las restricciones mencionadas, permitirá aprovechar los beneficios de esta técnica minimizando los riesgos asociados.

Más casos caché

Podemos aprovechar la implementación de la caché con Redis que tenemos para seguir cacheando más información.

Algunos de los ejemplos para los que podríamos usar la caché podría ser interesante cachear parte de la información del listado general de cargas, el cual es muy usado, también podríamos cachear usuarios y compañías, pues estos no cambian mucho.

Otra opción muy interesante que hemos estado planteando es cachear las solicitudes de localización de Google, nuestra plataforma realiza muchas de estas solicitudes y un número importante son repetidas por lo que podríamos plantear cachear dichas solicitudes. Esto sería un ahorro importante en cuando a costo, pues el coste acumulado de las solicitudes es importante ahora mismo.

Sistema de Alertas

Para garantizar que el rendimiento del sistema se mantenga dentro de los parámetros establecidos, se pueden implementar alertas que notifiquen por correo electrónico o a través de otros sistemas de mensajería cuando las pruebas excedan los límites de rendimiento acordados.

Estas alertas proporcionan una retroalimentación inmediata, permitiendo a los equipos de desarrollo y operaciones identificar rápidamente las áreas problemáticas y tomar medidas correctivas. Al integrar esta funcionalidad en el proceso de pruebas de rendimiento, se mejora la capacidad de respuesta ante incidencias, garantizando que el sistema funcione dentro de los estándares esperados, incluso bajo condiciones de alta carga.

3 Conclusiones

El desarrollo del proyecto ha resultado ser más extenso de lo inicialmente anticipado. La implementación de pruebas de carga reveló numerosos detalles críticos que necesitaban atención para garantizar su efectividad. A pesar de estos desafíos, los resultados obtenidos han satisfecho prácticamente todas nuestras necesidades previstas. Se ha logrado desarrollar un cuadro de mandos integral y eficaz, donde la información se presenta de manera clara y concisa. Además, hemos explorado más escenarios de uso en la plataforma de los que esperábamos, y hemos podido probar e implementar de forma exitosa ciertas optimizaciones basadas en la información recogida de estos escenarios.

Una parte significativa del trabajo ha sido la generación de datos ficticios, que resultó ser particularmente costosa en términos de tiempo. Lamentablemente, opté por una estrategia que buscaba maximizar la aleatoriedad y la similitud con los datos que generaría un cliente de alto volumen. Aunque esto es favorable para probar el sistema bajo condiciones extremas, el tiempo y esfuerzo invertido en esta tarea no se reflejan directamente en la memoria del proyecto, ya que no era el enfoque principal del mismo.

Estoy satisfecho con las diversas optimizaciones implementadas. Hemos adquirido conocimientos valiosos sobre la configuración de parámetros, tecnologías como Redis y características específicas de AWS, como las URL firmadas de S3, que hemos podido usar en nuestro beneficio.

Finalmente, estoy muy contento con los resultados finales y con lo aprendido durante el proyecto. Desde mi empresa, hemos aplicado algunos de los conocimientos adquiridos no solo en la plataforma sino también en otros ámbitos. Por ejemplo, hemos utilizado Redis para cachear solicitudes a Google Maps, lo cual ha representado un ahorro significativo para nuestra plataforma.

4 Bibliografia

[1] SoapUI. Taking it past response time: Key performance indicators for load testing. https://www.soapui.org/learn/load-testing/key-performance-indicators-for-load-testing/

[2] Grafana. Types of load testing. https://grafana.com/load-testing/types-of-load-testing/

[3] Brendan Gregg. USE Method: Linux Performance Checklist. https://www.brendangregg.com/USEmethod/use-linux.html

[4] Grafana, Julie Dam, 2018. The RED Method: How to Instrument Your Services. https://grafana.com/blog/2018/08/02/the-red-method-how-to-instrument-your-services/

[5] LoadNinja. Load Testing. https://loadninja.com/load-testing/

[6] OpKey, Sohaib Zaidi, 2023. Three Key Considerations For Load Testing When Migrating To The Cloud. https://www.opkey.com/blog/three-key-considerations-for-load-testing-when-migrating-to-t

[7] Element34. Embracing the 'Shift Left' Model in Software Testing. https://www.element34.com/blog/shift-left-testing

[8] k6. k6 docs. https://k6.io/docs/

[9] Grafana. Grafana docs. https://grafana.com/docs/grafana/latest/

[10] Prometheus. Prometheus overview. https://prometheus.io/docs/introduction/overview/

[11] Github. google/cAdvisor. https://github.com/google/cadvisor

[12] Prometheus. Exporters and integrations. https://prometheus.io/docs/instrumenting/exporters/

[13] Docker. Docker docs. https://docs.docker.com/

[14] InfluxDB. InfluxDB docs. https://docs.influxdata.com/

[15] Typescriptlang. TypeScript docs. https://www.typescriptlang.org/docs/

[16] Faker.js. Faker.js guide. https://fakerjs.dev/guide/

[17] k6. Running k6. https://k6.io/docs/get-started/running-k6/

[18] k6. Test lifecycle. https://k6.io/docs/using-k6/test-lifecycle/

[19] k6. Results output. https://k6.io/docs/get-started/results-output/

[20] Grafana. Grafana data sources. https://grafana.com/docs/grafana/latest/datasources/

[21] Grafana. Build your first dashboard. https://grafana.com/docs/grafana/latest/getting-started/build-first-dashboard/

[2] AWS. Amazon EC2. https://aws.amazon.com/ec2/

[22] AWS. Amazon S3 https://aws.amazon.com/s3/

[23] Redis. Redis docs: https://redis.io/docs/latest/

[24] Symfony. Redis cache adapter. https://symfony.com/doc/current/components/cache/adapters/redis_adapter.html

[25] Symfony. The Cache Component. https://symfony.com/doc/current/components/cache.html

[26] AWS. Sharing objects with presigned URLs. https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html

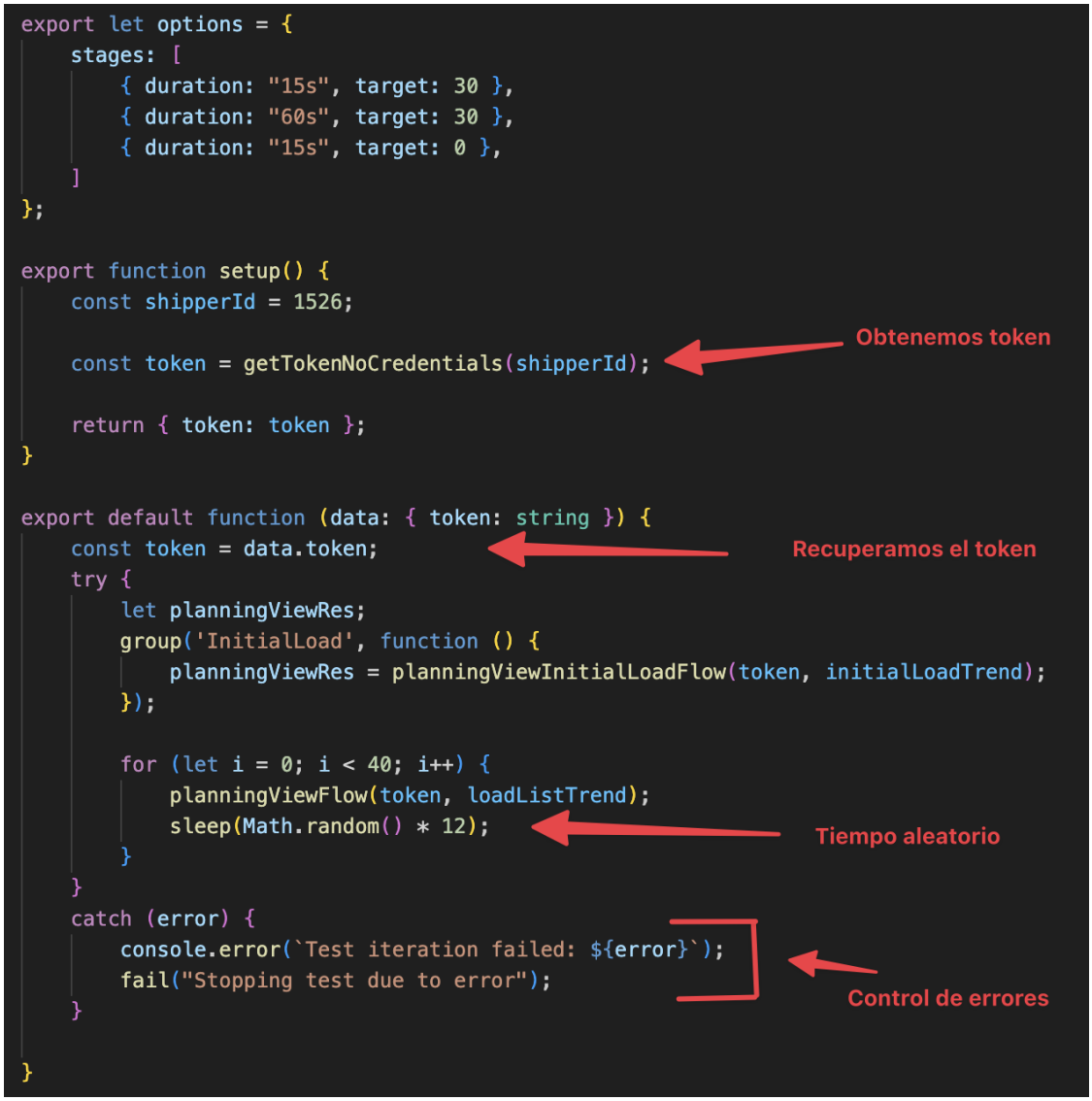
[27] PHP. Opcache docs: https://www.php.net/manual/en/book.opcache.php

[28] Gitlab. Gitlab docs. https://docs.gitlab.com/

[29] AWS. AWS Lambda docs. https://aws.amazon.com/lambda/

[30] Grafana. Grafana Alerting. https://grafana.com/docs/grafana/latest/alerting/

5 Anexo código




```
export let options = {
    stages: [
        { duration: "15s", target: 30 },
        { duration: "60s", target: 30 },
        { duration: "15s", target: 0 },
    ]
};

export function setup() {
    const shipperId = 1526;

    const token = getTokenNoCredentials(shipperId);

    return { token: token };
}

export default function (data: { token: string }) {
    const token = data.token;
    try {
        let planningViewRes;
        group('InitialLoad', function () {
            planningViewRes = planningViewInitialLoadFlow(token, initialLoadTrend);
        });

        for (let i = 0; i < 40; i++) {
            planningViewFlow(token, loadListTrend);
            sleep(Math.random() * 12);
        }
    }
    catch (error) {
        console.error(`Test iteration failed: ${error}`);
        fail("Stopping test due to error");
    }

}
```

Figura 47: Codigo ejemplo completo k6 prueba

Figura 48: Codigo ejemplo completo populador de datos