MiMoQ: Soporte para fallas sintéticas y despliegue automatizado

Giovanny Albarracín

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
MAESTRÍA EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
BOGOTÁ, D.C.
2025

MiMoQ: Soporte para fallas sintéticas y despliegue automatizado

Autor:

Giovanny Fermín Albarracín Riveros

MEMORIA DEL TRABAJO DE GRADO REALIZADO PARA CUMPLIR UNO DE LOS REQUISITOS PARA OPTAR AL TITULO DE

MAGÍSTER EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

Director/a

Ing. Mariela Josefina Curiel Huérfano

Comité de Evaluación del Trabajo de Grado

<Nombres y Apellidos Completos del Jurado >

<Nombres y Apellidos Completos del Jurado >

PONTIFICIA UNIVERSIDAD JAVERIANA
FACULTAD DE INGENIERIA
MAESTRÍA EN INGENIERIA DE SISTEMAS Y COMPUTACIÓN
BOGOTÁ, D.C.
Diciembre 2025

PONTIFICIA UNIVERSIDAD JAVERIANA

FACULTAD DE INGENIERIA
MAESTRÍA EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

Rector Magnífico

Luis Fernando Múnera Congote, S.J.

Decano Facultad de Ingeniería

Ing. Diego Alejandro Patiño Guevara, Ph.D.

Directora Maestría en Ingeniería de Sistemas y Computación

Ing. Mariela Josefina Curiel Huérfano, Ph.D.

Director Departamento de Ingeniería de Sistemas

Ing. César Julio Bustacara Medina, Ph.D.

Artículo 23 de la Resolución No. 1 de junio de 1946

"La Universidad no se hace responsable de los conceptos emitidos por sus alumnos en sus proyectos de grado. Sólo velará porque no se publique nada contrario al dogma y la moral católica y porque no contengan ataques o polémicas puramente personales. Antes bien, que se vean en ellos el anhelo de buscar la verdad y la Justicia"

AGRADECIMIENTOS

Agradezco profundamente a mis profesores, compañeros y demás personal de la universidad por hacer posible esta maestría que culmina con este trabajo de grado. Un cálido agradecimiento a la profesora Mariela por ayudarme en el proceso de implementación y desarrollo del trabajo, a mi familia por su inmenso apoyo en los momentos clave en este proceso de aprendizaje y muchas gracias a Franchesca por nunca dejar de creer en mí y en mis capacidades para hacer este proyecto de grado posible.

Contenido

1. LISTA DE FIGURAS

2. LISTA DE TABLAS

ABSTRACT

MiMoQ is a platform designed to evaluate quality attributes in microservice-based systems through automated experimentation. However, its initial version lacked deployment capabilities in real Kubernetes clusters and did not support fault injection. This work extends MiMoQ by refactoring its architecture, integrating Infrastructure as Code for reproducible deployments, and adding a synthetic failure injection module based on Chaos Engineering principles. The system was deployed on an on-premise Kubernetes cluster, enabling experiments that combine load testing, resilience evaluation, and metric analysis. Results demonstrate improved reproducibility, automation, and the ability to measure recovery behavior under controlled adverse conditions

RESUMEN

MiMoQ es una plataforma diseñada para experimentar atributos de calidad en sistemas basados en microservicios. Sin embargo, su versión inicial carecía de mecanismos para desplegar la aplicación en diferentes entornos además de la ausencia de la inyección de fallas en su experimento. Este trabajo extiende MiMoQ refactorizando su arquitectura, integrando infraestructura como código para su fácil despliegue y añade el módulo de inyección de fallas basado en los principios de ingeniería del caos. El sistema se desplego en un clúster on-premise (universidad Javeriana) permitiendo ejecutar experimentos que combinan las pruebas de carga, la evaluación de resiliencia y la habilidad de obtener métricas en torno al desempeño de las aplicaciones

RESUMEN EJECUTIVO

MiMoQ es una plataforma desarrollada en la Pontificia Universidad Javeriana con el propósito de proporcionar un entorno controlado para la experimentación y evaluación de atributos de calidad en arquitecturas basadas en microservicios. Estas arquitecturas, ampliamente adoptadas en la industria por su escalabilidad, flexibilidad y modularidad, presentan desafíos significativos cuando se busca evaluar su rendimiento, estabilidad, resiliencia y capacidad de recuperación. El estado del arte evidencia una falta de herramientas que integren un flujo experimental completo y automatizado que permita analizar estos atributos bajo escenarios de carga y fallo de manera sistemática y reproducible. En este contexto, el presente trabajo de grado aborda y resuelve varias de las limitaciones de la versión inicial de MiMoQ, elevando considerablemente sus capacidades y alcances investigativos.

Originalmente, MiMoQ solo podía ejecutarse de forma local mediante MicroK8s y requería configuraciones manuales extensas, lo cual dificultaba la reproducibilidad de los experimentos, limitaba la escalabilidad y reducía la validez de los resultados obtenidos. Además, la herramienta carecía de soporte para escenarios de resiliencia y no contaba con un mecanismo de inyección de fallas sintéticas. Esta carencia impedía estudiar el comportamiento de microservicios frente a fallos reales, degradaciones, interrupciones inesperadas o fluctuaciones en los recursos, los cuales son escenarios críticos en entornos distribuidos modernos. Asimismo, existía una separación significativa entre los repositorios del backend y del frontend, sin documentación completa ni integración funcional. Esto aumentaba la fricción para el usuario y complicaba el mantenimiento del sistema.

El presente trabajo se propuso resolver estas limitaciones mediante una refactorización integral y la incorporación de dos nuevas funcionalidades centrales: un módulo completo de inyección de fallas sintéticas basado en principios de Ingeniería del Caos y la capacidad de desplegar MiMoQ en un clúster real mediante infraestructura como código (IaC). Estos avances transforman la plataforma en un entorno robusto, escalable y adecuado tanto para la investigación académica como para escenarios de validación industrial.

En primera instancia, se emprendió una revisión exhaustiva de la arquitectura existente, identificando elementos redundantes, acoplamientos innecesarios y procesos manuales que dificultaban la automatización. Se optó por migrar toda la plataforma a un mono-repo con el fin de consolidar el frontend, backend, scripts, configuración de métricas, definiciones de Kubernetes y nuevos módulos en un mismo entorno versionado. Este cambio permitió garantizar coherencia, trazabilidad y facilidad de despliegue. Asimismo, se introdujo Tilt como herramienta para acelerar el desarrollo local, permitiendo a los desarrolladores observar cambios inmediatos dentro del clúster sin necesidad de reconstruir constantemente imágenes de contenedores. Esto resultó fundamental para un ciclo de desarrollo más ágil, consistente y alineado con la infraestructura objetivo.

Otro de los avances significativos fue la migración completa hacia una arquitectura totalmente contenida dentro de Kubernetes. En la versión inicial, el servidor corría fuera del clúster y se ejecutaban los servicios desde la consola, introduciendo variabilidad entre ambientes, posibles errores y demoras en los comandos. Con la nueva arquitectura, todos los componentes corren

dentro del clúster: frontend, backend, base de datos, generador de carga, Prometheus y el módulo de inyección de fallos. Esta decisión no solo simplifica el despliegue, sino que asegura que las condiciones bajo las cuales se ejecutan los experimentos sean controladas, consistentes y replicables. Además, se habilita la escalabilidad horizontal del backend y se garantiza que la infraestructura exacta se describa mediante manifiestos de Kubernetes, lo cual constituye el fundamento de IaC.

En cuanto a la funcionalidad experimental, la plataforma ahora permite configurar pruebas de carga de manera completamente flexible mediante una interfaz visual renovada. Los usuarios pueden definir endpoints, patrones de carga, número de usuarios virtuales, duración y escenarios complejos que pueden involucrar múltiples rutas o distribuciones de tráfico. Esto amplía notablemente el alcance experimental de la herramienta, pasando de pruebas simples a escenarios realistas que reflejan mejor el comportamiento de sistemas de microservicios en producción.

El módulo de métricas fue igualmente transformado. Basado en Prometheus, ahora permite recolectar series temporales asociadas a métricas de latencia, errores, uso de CPU, memoria, tráfico, fallos y más. La librería interna encargada de procesar métricas fue refactorizada para mejorar la consistencia, la trazabilidad y la capacidad de exportación de resultados en CSV. Esto facilita análisis estadísticos posteriores y permite comparar múltiples ejecuciones para efectos de validación experimental.

Una de las contribuciones más significativa del trabajo es la integración de un módulo completo de inyección de fallas sintéticas. Basado en herramientas modernas de Ingeniería del Caos (como Chaos Mesh), el sistema ahora puede provocar fallos controlados en pods, nodos, redes y componentes específicos del sistema bajo prueba. Entre las fallas soportadas se encuentran: interrupciones de pods, aumentos artificiales de latencia, pérdida de paquetes, consumo excesivo de CPU o memoria y fallos intermitentes. La ejecución de estos experimentos complementa los análisis de rendimiento con la evaluación de resiliencia, permitiendo observar cómo los microservicios se recuperan de un fallo, qué tan rápido se reestablecen las réplicas y cómo varían las métricas durante y después del fallo.

La solución fue desplegada en un clúster on-premise de la Universidad Javeriana compuesto por cuatro nodos, validando así la portabilidad y escalabilidad del sistema. Las pruebas demostraron que los manifiestos de Kubernetes permiten reproducir el entorno con precisión en diferentes máquinas, y que la arquitectura responde adecuadamente bajo escenarios de carga y caos. Se comprobó además que los experimentos son completamente reproducibles: el mismo experimento ejecutado en diferentes momentos produce resultados consistentes, lo que demuestra la validez metodológica del enfoque adoptado.

En términos de resultados, la plataforma mejorada permite ejecutar escenarios experimentales más ricos y completos. Ahora es posible analizar el comportamiento del sistema en condiciones adversas, medir la degradación, comparar tiempos de recuperación, observar patrones de resiliencia, evaluar la estabilidad del desempeño y correlacionar métricas de carga con métricas de fallo. Esto posiciona a MiMoQ como una herramienta única dentro del ecosistema académico de la Javeriana, al ofrecer un entorno de experimentación integral, automatizado y alineado con los estándares de la industria en monitoreo, IaC y Chaos Engineering.

3. INTRODUCCIÓN

Los requerimientos funcionales constituyen la base fundamental de cualquier desarrollo de software. No obstante, para lograr un funcionamiento óptimo y maximizar el valor del sistema, resulta indispensable considerar también los requerimientos no funcionales. Estos aspectos, conocidos como atributos de calidad, han cobrado cada vez más relevancia en las discusiones sobre arquitectura de software, ya que son esenciales para garantizar la eficiencia, escalabilidad y confiabilidad del sistema (Bass, 2012).

En este contexto, la industria ha reconocido la importancia de evaluar dichos atributos como parte integral del proceso de validación de diseños y arquitecturas. Una de las arquitecturas de software más adoptadas en los últimos años es la arquitectura orientada a microservicios. Esta arquitectura permite una mayor escalabilidad y elasticidad en comparación con enfoques tradicionales tales como arquitecturas monolíticas u orientadas a servicios, además de facilitar la modularización y el encapsulamiento de componentes, mejorando así la mantenibilidad y la evolución del sistema (Newman, 2015)

No obstante, evaluar atributos de calidad en sistemas de microservicios plantea desafíos significativos. Estudios recientes han identificado la falta de herramientas y entornos estandarizados para realizar experimentación controlada en arquitecturas de microservicios (Alvarado, 2017). Además, la evaluación se complica debido a los trade-offs inherentes entre ciertos atributos de calidad, como el equilibrio entre rendimiento y escalabilidad o entre rendimiento y seguridad, donde mejorar uno puede implicar sacrificar parcialmente otro (Alvarado, 2017). En particular, aspectos como el despliegue, la gestión de escalabilidad y la resiliencia frente a fallos pueden afectar considerablemente el comportamiento del sistema (Camilli, 2022). Estas limitaciones evidencian la necesidad de plataformas que permitan medir y experimentar de manera rigurosa atributos de calidad bajo distintos escenarios experimentales.

Con el objetivo de estudiar los atributos de calidad en sistemas basados en microservicios, en la universidad Javeriana se desarrolló la herramienta MiMoQ (Microservice Metrics of Quality). Un sistema de experimentación, diseñado para automatizar la ejecución de pruebas enfocadas en medir atributos de calidad clave como la elasticidad y el rendimiento (Alvarado, Sistema de experimentación para evaluar atributos de calidad en microservicios: MiMoQ, 2024). A través de MiMoQ, es posible simular distintos escenarios de experimentación y generar reportes sobre el comportamiento del sistema. Sin embargo, al momento de iniciar este proyecto la herramienta presentaba limitaciones importantes:

- La implementación no se encontraba desplegada en un clúster real, únicamente podía ejecutarse de forma local en una máquina de escritorio usando MicroK8s, mediante una configuración y despliegue manual
- No contemplaba escenarios en los que se produjeran fallos en los componentes del sistema.
- Separaron el Back y en Front en repositorios aparte sin documentación alguna, ni forma como conectarlos

- El uso de Helm y la consola para crear los proyectos y experimentos hace que se produzcan muchos errores y que sea muy lento el proceso de creación

El propósito de este trabajo de grado es mejorar MiMoQ, extendiendo su funcionalidad para permitir el despliegue, ya sea en servidores de la universidad o en la nube de algún proveedor mediante el uso de infraestructura como código (IaC) y la integración de un mecanismo de inyección de fallos sintéticos. Esto permitirá obtener métricas relacionadas con la capacidad de recuperación del sistema frente a situaciones adversas, enriqueciendo así el análisis de atributos de calidad en arquitecturas de microservicios.

El trabajo se estructura de la siguiente forma: En el capítulo 1 se describe el proyecto en forma general, identificando el problema, en el capítulo 2 se enumeran los Objetivos y la metodología de trabajo, en el capítulo 3 se describe el marco teórico, en el capítulo 4, se explica la implementación de la solución, en el capítulo 5 se describen los resultados. Finalmente en el capítulo 6 se presentan las conclusiones y Trabajo Futuro.

4. 1. DESCRIPCIÓN GENERAL

Oportunidad y problemática

Dado el contexto y la importancia de construir herramientas que permitan experimentar y tener métricas cuantificables en torno a requerimientos no funcionales de los sistemas con arquitectura basadas en microservicios (Bass, 2012), Se desarrolló MiMoQ, una plataforma que actualmente permite experimentar con pruebas de carga en entornos locales, generando métricas elegidas por el usuario asociadas a atributos definidos en el modelo ISO/IEC 25010. Además, MiMoQ ofrece capacidades para desplegar microservicios, variar factores de experimentación, generar carga, almacenar los datos obtenidos y evaluar atributos de calidad relacionados con disponibilidad, rendimiento y otros criterios relevantes (Alvarado, Sistema de experimentación para evaluar atributos de calidad en microservicios: MiMoQ, 2024). No obstante, la versión actual de la herramienta presenta limitaciones que restringen su potencial de análisis. En particular, su enfoque se centra principalmente en la ejecución de pruebas de carga para evaluar el desempeño, lo cual resulta insuficiente para obtener una visión integral del comportamiento de los sistemas ante condiciones adversas (Pahl, 2018). Esta limitación impide evaluar de manera completa la resiliencia y capacidad de recuperación de los microservicios, atributos esenciales en entornos distribuidos.

Asimismo, MiMoQ no cuenta con una infraestructura de despliegue que permita su ejecución en entornos de mayor realismo, como un clúster on-premise o ambientes en la nube, esto resulta fundamental para obtener resultados más confiables. Las condiciones operativas, los patrones de comunicación y la gestión de recursos varían significativamente entre estos entornos, lo que puede afectar el comportamiento de los microservicios y la precisión de las métricas recolectadas. Evaluar el sistema únicamente en entornos locales reduce la validez de los experimentos y limita la comprensión del impacto que tienen factores externos, como la latencia de red, la asignación dinámica de recursos o las estrategias de escalado automático (Chen, 2016)

Por último, el código existente de MiMoQ requiere mejoras en su implementación y estructura interna para facilitar su extensión y la incorporación de nuevas funcionalidades, como:

- Usar el patrón de diseño facade para administrar los diferentes recursos de Kubernetes, servicio de K6, servicio de Prometheus y servicio de Chaos, pues al crear una capa de abstracción entre los Apis de las diferentes herramientas y el código mejora la mantenibilidad, desacoplamiento y testeo de los servicios (Shvets, 2021), como lo muestra la Ilustración 1
- Usar los API de Kubernetes y los CRD (Custom Resource Definition) de K6 y Chaos para crear dinámicamente los servicios dentro del clúster

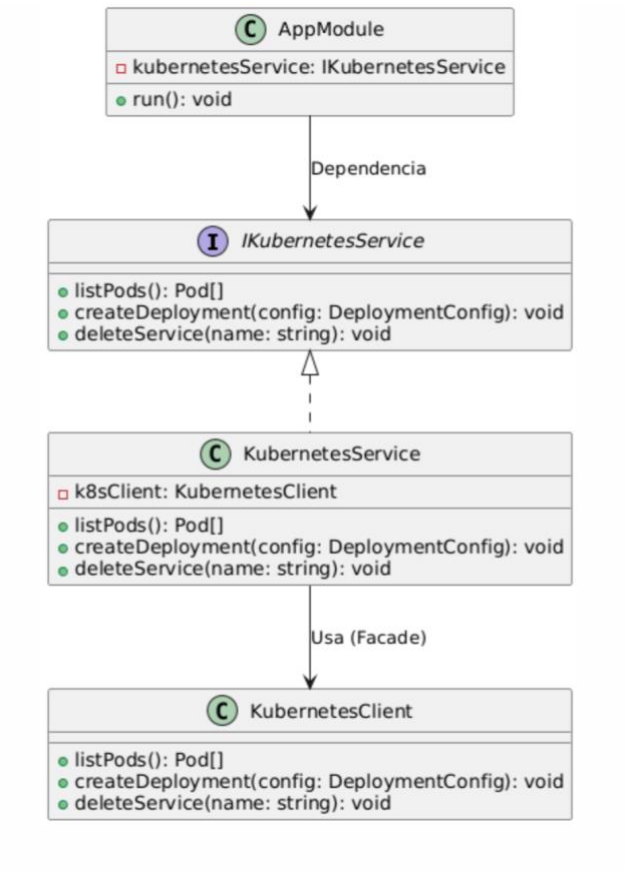

Ilustración 1. Diagrama de clases mostrando el patrón facade para el uso de API de kubernetes

Por lo anterior, resulta importante dotar de nuevas funcionalidades a la herramienta que amplíen su cobertura de análisis especialmente en métricas de resiliencia y recuperación ya que es un factor muy importante en sistemas distribuidos, además de poder llevar a MiMoQ a ambientes más realistas con más capacidad de cómputo. Así, nos permitirá estresar mucho más los sistemas y hacer pruebas mucho más fidedignas

5. DESCRIPCIÓN DEL PROYECTO

Objetivo general

- Añadir al sistema de experimentación MiMoQ funcionalidades que permitan la inyección fallas y su despliegue en entornos controlados

Objetivos específicos

- Comprender detalladamente la arquitectura y funcionalidades de MiMoQ
- Desplegar el sistema en un clúster
- Evaluar diversas herramientas y tecnologías que permitan la implementación de las nuevas funcionalidades.
- Implementar las nuevas funcionalidades definidas para MiMoQ, integrándolas de manera coherente con la arquitectura existente.
- Ejecutar un conjunto de pruebas de validación que verifiquen la funcionalidad extendida y validen los atributos de calidad en distintos entornos.

Metodología y Fases de desarrollo

El desarrollo se llevará a cabo utilizando herramientas de trabajo ágiles, específicamente SCRUM. Al finalizar cada uno de los Sprints, se realizará una entrega y demostración del avance ante la directora del trabajo de grado, que a su vez es el cliente y usuario final del proyecto, quien podrá hacer observaciones y sugerencias. Este proceso se repetirá de manera continua hasta alcanzar la mejora prevista.

Además, se harán reuniones periódicas para discutir temas relacionados con el desarrollo y avance del trabajo

Fase	**Resultados esperados**
Fase 1 - Análisis y comprensión del sistema	Documentación de la arquitectura actual de MiMoQ, sus componentes, flujos internos, dependencias, restricciones técnicas y puntos críticos.

Fase 2 - Exploración tecnológica y diseño de alternativas	Matriz comparativa de herramientas y tecnologías candidatas; análisis de viabilidad técnica; selección justificada de las tecnologías que permitirán implementar las nuevas funcionalidades.
Fase 3 - Despliegue base y diagnóstico del sistema actual	Despliegue de MiMoQ en un clúster funcional; identificación documentada de errores, limitaciones, oportunidades de mejora y brechas frente a los objetivos de calidad.
Fase 4 - Implementación de las nuevas funcionalidades	Desarrollo modular de las funcionalidades definidas; integración con la arquitectura existente; generación de documentación técnica; actualización de APIs, módulos y componentes afectados.
Fase 5 - Validación, pruebas avanzadas y despliegue final	Ejecución de pruebas controladas de resiliencia, elasticidad y rendimiento; análisis comparativo entre el sistema original y el mejorado; despliegue final del sistema extendido y reporte de validación de atributos de calidad.

6. MARCO TEÓRICO Y TRABAJOS RELACIONADOS

Marco teórico

1.1 Experimentación y Proceso Experimental

La experimentación en ingeniería de software se concibe como un proceso sistemático orientado a observar y analizar el comportamiento de un sistema cuando es sometido a condiciones controladas. Su propósito es generar evidencia empírica acerca de cómo afectan distintas variables a los atributos de calidad del sistema, tales como el rendimiento, la escalabilidad, la elasticidad o la fiabilidad, aspectos que resultan críticos en sistemas distribuidos modernos (Alvarado, 2024). A diferencia de las pruebas funcionales tradicionales, que se enfocan en la verificación del cumplimiento de requisitos, la experimentación permite comprender dinámicas internas complejas que emergen solo bajo carga o en contextos donde múltiples componentes interactúan simultáneamente.

En arquitecturas basadas en microservicios, la experimentación es indispensable debido a la naturaleza descentralizada y altamente distribuida de estos sistemas. Los microservicios se desarrollan y despliegan de manera independiente, se comunican mediante interfaces livianas y se ejecutan en entornos orquestados a través de contenedores. Esta distribución introduce efectos emergentes difíciles de predecir sin mediciones reales, como variaciones en la latencia, propagación de fallos entre componentes, saturación de recursos y patrones de desempeño no lineales (Alvarado, 2024). Por esta razón, la experimentación permite validar suposiciones arquitectónicas, identificar cuellos de botella y evaluar configuraciones de despliegue de manera objetiva.

El proceso experimental se estructura en torno a elementos clave. El sistema bajo estudio (SUT) corresponde al conjunto completo de servicios y componentes que se evalúan, generalmente desplegados en un entorno controlado para garantizar consistencia en las ejecuciones. Cuando se requiere un análisis más específico, se selecciona un componente bajo estudio (CUT), que puede corresponder a un microservicio particular o a un subconjunto del sistema. Este nivel de granularidad permite comprender comportamientos particulares que podrían quedar ocultos en análisis globales, especialmente en arquitecturas donde cada microservicio tiene patrones de carga propios y diferentes niveles de sensibilidad a la demanda (Alvarado, 2024).

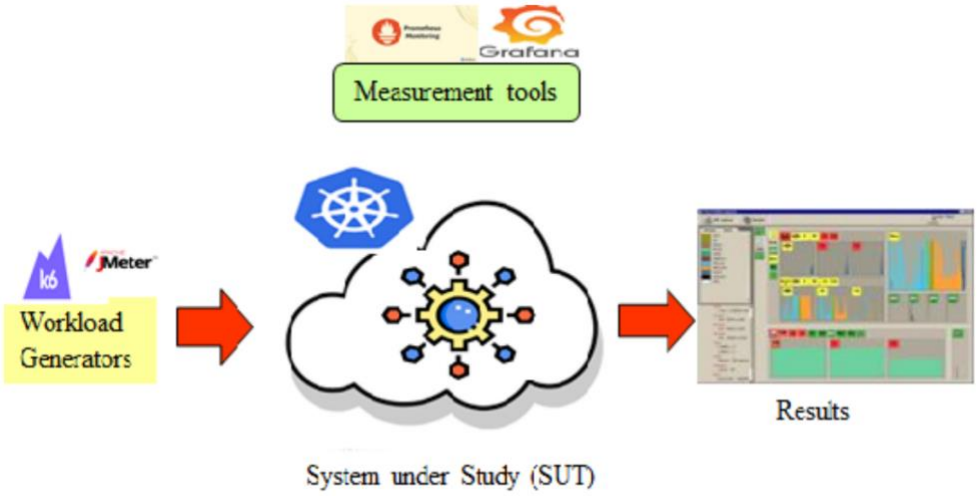

Ilustración 2. Proceso de obtención de métricas (Alvarado, 2024).

El monitoreo y la medición son indispensables para capturar datos significativos durante el experimento. A través de herramientas que registran métricas en series de tiempo, es posible observar indicadores asociados a atributos de calidad contemplados en modelos como el ISO/IEC 25010, tales como tiempo de respuesta, throughput, utilización de CPU, consumo de memoria o disponibilidad del servicio. Estas mediciones ofrecen una visión cuantitativa del comportamiento del sistema, y permiten correlacionar variaciones en la carga o en la configuración con cambios en el desempeño observado (Alvarado, 2024). Además, la persistencia y análisis posterior de los datos facilitan comparaciones rigurosas entre distintos experimentos o versiones del sistema.

El proceso experimental sigue una secuencia estructurada que inicia con la planificación del experimento. En esta etapa se definen los objetivos, las variables independientes y dependientes, el número de repeticiones, la duración de la prueba y las métricas a observar. Posteriormente se configura el entorno, el cual debe ser controlado y consistente para evitar efectos externos no deseados. La ejecución del experimento implica la aplicación de la carga definida, la recolección de métricas y el monitoreo continuo del sistema. Finalmente, los resultados se consolidan y analizan, y el entorno se restablece a su estado inicial para permitir nuevas ejecuciones sin interferencias residuales, ver Ilustración 2 sobre el proceso de obtención de métricas. Este restablecimiento resulta fundamental para asegurar la validez interna del experimento y garantizar que futuras repeticiones no estén condicionadas por estados previos del sistema (Alvarado, 2024).

La experimentación, en suma, constituye un componente esencial para comprender el comportamiento de sistemas complejos basados en microservicios. Proporciona un medio riguroso y empírico para evaluar decisiones arquitectónicas, identificar problemas ocultos, mejorar la operación del sistema y obtener información objetiva que guíe su evolución. El estudio sistemático mediante experimentación permite no solo detectar fallos o limitaciones,

sino también fundamentar mejoras que fortalezcan el desempeño, la escalabilidad y la confiabilidad del sistema en escenarios reales y simulados.

1.2 Benchmarking

Un benchmark de aplicación consiste en una versión reducida, controlada y/o escalada de un sistema real que reproduce comportamientos clave mediante la simulación de cargas, flujos de trabajo y escenarios representativos. Esta aproximación permite ejecutar mediciones de forma rápida, reproducible y con recursos acotados, generando resultados útiles sin necesidad de desplegar sistemas completos en entornos productivos (Sheldon Smith, 2023). En esencia, un benchmark funciona como un modelo experimental del sistema, lo suficientemente fiel para observar tendencias, degradaciones y límites operativos.

En el ámbito de arquitecturas basadas en microservicios, los benchmarks adquieren un papel especialmente relevante. Al emular interacciones entre servicios, latencias, fallos, dependencias externas y patrones de escalado, posibilitan la obtención de métricas clave asociadas a atributos de calidad como rendimiento, latencia extremo a extremo, resiliencia, disponibilidad, capacidad de carga y elasticidad. Este tipo de mediciones resulta fundamental para plataformas cuyo objetivo es evaluar de manera sistemática el comportamiento de un sistema distribuido bajo condiciones controladas y reproducibles (Sheldon Smith, 2023).

Dentro de este contexto, los experimentos se convierten en el mecanismo central para explotar el potencial de los benchmarks. Un experimento consiste en la ejecución planificada, repetible y medible de un escenario de prueba sobre el benchmark, con el fin de observar cómo un sistema reacciona ante condiciones específicas, variaciones de carga o la introducción de fallas. En arquitecturas de microservicios, este enfoque experimental permite analizar fenómenos complejos como fallos en cascada, cuellos de botella, velocidades de recuperación, tiempos de autoscaling y efectos de la contención entre servicios. De esta manera, los benchmarks no solo sirven como modelos reducidos, sino como entornos con la instrumentación necesaria para diseñar, ejecutar y comparar experimentos de forma sistemática (Sheldon Smith, 2023; Alvarado, 2024).

El empleo conjunto de benchmarks y experimentos dentro de MiMoQ constituye un pilar robusto para la evaluación del rendimiento y la calidad de sistemas distribuidos. Al trabajar con implementaciones simplificadas pero fieles a los patrones esenciales de aplicaciones reales, MiMoQ permite obtener métricas relevantes sobre atributos como disponibilidad, desempeño y elasticidad, reduciendo al mismo tiempo los elevados costos operativos asociados a la ejecución de sistemas completos en entornos reales de prueba (Alvarado, 2024). Además, la naturaleza modular de los benchmarks facilita la repetición, comparación y validación estadística de experimentos, permitiendo detectar regresiones, medir impactos de cambios arquitectónicos y observar comportamientos emergentes bajo condiciones controladas.

Finalmente, aunque los benchmarks constituyen una herramienta de alto valor, no sustituyen por completo la validación en entornos productivos. Sin embargo, su capacidad de ofrecer una aproximación consistente y experimental al comportamiento del sistema en un ambiente

controlado los convierte en un componente esencial para el aseguramiento de la calidad, el análisis comparativo de arquitecturas, el diseño de pruebas de carga y la optimización del rendimiento en arquitecturas de microservicios (Alvarado, 2024).

1.3 Microservicios

La arquitectura de microservicios es un estilo de diseño de software que propone construir aplicaciones como un conjunto de servicios pequeños, autónomos y altamente desacoplados, cada uno responsable de una única capacidad del negocio. Este enfoque se popularizó como una evolución de la arquitectura monolítica tradicional, en la que todos los componentes se despliegan como una sola unidad (Newman, 2015).

Uno de los principios clave de los microservicios es la independencia del despliegue. Cada servicio puede desarrollarse, actualizarse, escalarse y desplegarse sin afectar directamente al resto del sistema, lo que habilita ciclos de entrega continua más rápidos y reduce el riesgo asociado con los despliegues integrales (Fowler, 2014) Esta independencia favorece también la autonomía de los equipos, que pueden elegir las tecnologías, lenguajes de programación y bases de datos que mejor se ajusten a las necesidades de cada servicio (Newman, 2015).

La adopción de microservicios está estrechamente ligada al uso de contenedores y herramientas de orquestación como Kubernetes, que facilitan el escalado horizontal y la automatización de despliegues. Para gestionar la complejidad resultante de un sistema distribuido, se requieren prácticas como trazado distribuido, monitoreo centralizado y observabilidad avanzada (Villamizar, 2016)

En términos de atributos de calidad, la arquitectura de microservicios se destaca por mejorar la escalabilidad**,** resiliencia**,** mantenibilidad **y** disponibilidad, ya que cada servicio puede escalarse de forma independiente y aislar fallos sin comprometer el sistema completo. Sin embargo, también introduce desafíos importantes, como una mayor complejidad operativa, dependencia de redes poco fiables, sobrecarga de comunicación y mayor dificultad para garantizar consistencia en datos distribuidos (Newman, 2015).

En síntesis, los microservicios representan un enfoque arquitectónico que permite mayor flexibilidad y escalabilidad en sistemas modernos, pero requieren una infraestructura madura y prácticas especializadas para mitigar los retos inherentes a los sistemas distribuidos.

1.4 Ingeniería del Caos e Inyección de Fallos

La Ingeniería del Caos es una disciplina que busca evaluar y mejorar la resiliencia de sistemas distribuidos mediante la introducción controlada y sistemática de condiciones adversas. Su principio fundamental consiste en "experimentar" con el sistema en entornos reales o cercanos a producción para observar su comportamiento cuando enfrenta fallos inevitables, degradaciones de servicio o condiciones extremas de operación. El objetivo es descubrir debilidades ocultas, validar mecanismos de recuperación y garantizar que el sistema mantenga niveles aceptables de funcionamiento incluso bajo escenarios inesperados (Basiri et al., 2016; Netflix, 2015). En arquitecturas de microservicios esta caracterizadas por un alto grado de

distribución, comunicación remota y múltiples dependencias. Este enfoque resulta esencial debido a la complejidad emergente y a la probabilidad elevada de fallos.

Dentro del marco de calidad de software definido por ISO/IEC 25010, la ingeniería del caos permite evaluar atributos como fiabilidad**,** tolerancia a fallos**,** recuperabilidad**,** disponibilidad **y** continuidad operativa. Aunque la versión inicial de MiMoQ se centra principalmente en el atributo de desempeño, la evaluación de la fiabilidad y otros atributos no puede limitarse a recolectar métricas provenientes de los monitores; implica, además, someter al sistema a eventos de fallo reales o simulados para analizar su comportamiento y validar los mecanismos de resiliencia (ISO/IEC 25010, 2011).

En este contexto, la inyección de fallas constituye una técnica específica dentro del conjunto de prácticas de la ingeniería del caos. Consiste en provocar perturbaciones deliberadas sobre componentes del sistema, como CPU, memoria, red, almacenamiento o servicios externos con el fin de observar cómo responde ante condiciones adversas. Estas perturbaciones pueden realizarse mediante hardware, software o virtualización, aunque en los sistemas distribuidos modernos predominan las técnicas basadas en software, que permiten simular latencias, pérdida de paquetes, interrupciones de red, caídas de contenedores o fallos en dependencias externas de manera reproducible y automatizada (Bavishi et al., 2019). La inyección de fallas se relaciona directamente con la disciplina del chaos engineering, la cual propone introducir fallas de forma sistemática y continua para validar el comportamiento del sistema ante condiciones adversas reales.

Herramientas populares de esta disciplina, como Chaos Monkey**,** Chaos Mesh o Gremlin, facilitan la automatización de estos experimentos tanto en entornos de producción controlada como en preproducción, integrando mecanismos de observabilidad para medir impactos en métricas clave de desempeño y fiabilidad. En arquitecturas basadas en microservicios, estas prácticas permiten evaluar el funcionamiento de políticas de reintentos, *timeouts*, *circuit breakers*, balanceadores de carga y estrategias de auto escalado cuando el sistema enfrenta fallos parciales o totales.

Así, la ingeniería del caos (y dentro de ella, la inyección de fallas) constituye un instrumento fundamental para garantizar la robustez y resiliencia de aplicaciones distribuidas, permitiendo validar su comportamiento real ante escenarios adversos y reforzando los atributos de calidad esenciales definidos por estándares internacionales como ISO/IEC 25010. En la sección Plataformas de Ingeniería del Caos se describe y evalúan las herramientas mencionadas para la inyección de fallas

1.5 Kubernetes

Kubernetes es una plataforma de orquestación de contenedores diseñada para automatizar el despliegue, la administración, la escalabilidad y la operación de aplicaciones contenerizadas. Originalmente desarrollada por Google e inspirada en su experiencia operando sistemas distribuidos a gran escala, Kubernetes se ha convertido en el estándar de facto para la gestión de cargas de trabajo basadas en contenedores (Hightower, 2017). Su arquitectura modular y

declarativa permite administrar sistemas altamente dinámicos al abstraer la infraestructura subyacente y proporcionar mecanismos robustos de control, resiliencia y autosanación.

Desde una perspectiva arquitectónica, Kubernetes se organiza bajo un modelo control-plane / worker-nodes, en el que el control-plane toma decisiones globales sobre el estado del clúster, mientras que los nodos trabajadores ejecutan las cargas de trabajo. El Control Plane está compuesto por varios componentes centrales: api Server, encargado de exponer la API que actúa como interfaz principal del clúster, base de datos distribuida que almacena el estado deseado del sistema; scheduler, responsable de asignar Pods a nodos según restricciones y recursos; y controller-manager, que ejecuta los controladores encargados de mantener la coherencia entre el estado deseado y el estado actual del sistema (Hightower, 2017).

Por su parte, los nodos trabajadores ejecutan componentes como kubelet, que gestiona los contenedores localmente y se comunica con el Control Plane; proxy, que gestiona el enrutamiento de red para los servicios; y un *runtime* de contenedores (como containerd o CRI-O) encargado de ejecutar los contenedores en sí mismos. Esta separación de responsabilidades permite que Kubernetes mantenga una arquitectura distribuida, escalable y tolerante a fallos (Hightower, 2017).

El funcionamiento de Kubernetes se basa en un modelo declarativo mediante el cual los usuarios describen el estado deseado de sus aplicaciones utilizando recursos como *Deployments*, *Pods*, *Services*, *ConfigMaps* o *Ingresses*. Kubernetes, a través de su bucle de reconciliación, compara continuamente el estado actual del clúster con el estado deseado y ejecuta las acciones necesarias para alinearlos. Este proceso es llevado a cabo por los controladores (*controllers*), que aplican el principio de *control loops* para garantizar que el sistema se mantenga estable, coherente y autoajustado (Burns, 2016)

Entre las características clave de Kubernetes se encuentran la escalabilidad automática, la autocuración la abstracción de red y almacenamiento, y la capacidad de extender su funcionalidad mediante Custom Resource Definitions **(**CRDs**)** y Operators, permitiendo a los usuarios definir comportamientos avanzados mediante lógica declarativa o programática (Dobies, 2021). Estas capacidades hacen que Kubernetes sea especialmente adecuado para arquitecturas modernas basadas en microservicios, donde existen múltiples componentes independientes que necesitan coordinación, resiliencia y escalabilidad.

Kubernetes proporciona un marco robusto y extensible para administrar aplicaciones distribuidas, ofreciendo mecanismos avanzados de orquestación, control y automatización. Su diseño modular, declarativo y centrado en el estado deseado le permite funcionar como la plataforma principal para la ejecución de sistemas basados en contenedores en entornos tanto empresariales como de investigación

1.6 Kustomize

Kustomize es una herramienta diseñada para la gestión declarativa de configuraciones en Kubernetes, cuyo propósito es permitir la personalización de manifiestos YAML sin recurrir a plantillas ni duplicar archivos. La herramienta se ha convertido en un elemento central para la

gestión reproducible y estructurada de configuraciones en sistemas basados en Kubernetes (kubernetes, 2025)

Kustomize se basa en un enfoque 100% declarativo, en el cual las modificaciones a los recursos se realizan mediante parches estructurados o estratégicos aplicados sobre objetos ya definidos. Este diseño mantiene la filosofía de "YAML puro" promovida por Kubernetes, evitando introducir lenguajes de plantillas y asegurando que todos los cambios sean visibles, auditables y compatibles con sistemas de control de versiones.

Uno de los conceptos fundamentales de Kustomize es su modelo base–overlay, que permite separar las configuraciones comunes del sistema (la base) de las configuraciones específicas para cada entorno o variantes (Bavithran, 2025). Esta separación facilita la modularidad, promueve la reutilización y minimiza la duplicación de archivos YAML Los overlays pueden incluir parches, generadores de ConfigMaps o Secrets, transformaciones de nombres, sustitución de valores y otros mecanismos avanzados que enriquecen la personalización sin perder trazabilidad.

La siguiente figura ilustra este enfoque conceptual, mostrando cómo un conjunto de manifiestos base puede derivar en configuraciones específicas para diferentes entornos (Dev, Staging y Prod) mediante `kustomize build`, generando así recursos finales listos para ser desplegados en un clúster:

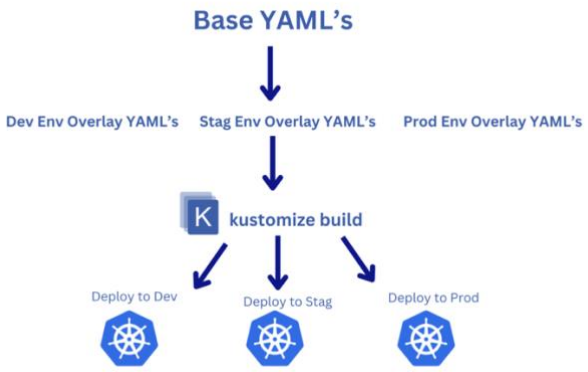

Ilustración 3. Diagrama de definición de kustomize

Este mecanismo permite que los despliegues mantengan consistencia entre entornos, preservando al mismo tiempo la capacidad de modificar parámetros específicos, como valores de recursos, endpoints, políticas de red o configuraciones de observabilidad. Además, al generar configuraciones completamente declarativas y determinísticas, Kustomize se alinea con prácticas de infraestructura como código (IaC) y despliegues reproducibles (Morris, 2020).

En resumen, Kustomize proporciona un marco conceptual que favorece la modularidad, el versionamiento, la trazabilidad y la claridad en la gestión de configuraciones para Kubernetes. Su naturaleza declarativa y su integración directa con `kubectl` lo convierten en una

herramienta especialmente apropiada para proyectos académicos y sistemas que requieren gobernanza estricta, reproducibilidad y separación explícita entre entornos de despliegue.

1.7 Infraestructura como código

La Infraestructura como Código (IaC) es un enfoque de gestión y provisión de infraestructura mediante archivos declarativos que describen el estado deseado de los recursos. En lugar de aprovisionar servidores, redes o servicios manualmente, IaC permite automatizar este proceso mediante herramientas que interpretan archivos de configuración para crear y mantener entornos de manera reproducible (Morris, 2016).

IaC se fundamenta en principios como automatización, declaratividad, control de versiones, reproducibilidad y trazabilidad**.** Estos principios permiten reducir errores humanos, mejorar la consistencia entre entornos y habilitar prácticas DevOps como integración y entrega continua (Humble & Farley, 2010). Además, IaC soporta la escalabilidad operativa al permitir que entornos enteros puedan crearse, modificarse o destruirse a través de un solo comando o pipeline.

1.8 Kubernetes como Infraestructura como Código

En el ecosistema en la nube, los manifiestos de Kubernetes (archivos YAML que definen objetos como Deployments, Services, ConfigMaps, Ingress o PersistentVolumes) se consideran una forma de IaC, ya que describen de manera declarativa el estado deseado de la infraestructura que soporta los contenedores. Estos manifiestos especifican cómo deben comportarse las aplicaciones, cómo deben escalar, qué recursos requieren y cómo deben exponerse dentro o fuera del clúster (Burns & Oppenheimer, 2016).

Kubernetes implementa el principio de IaC mediante un control loop interno: el plano de control observa continuamente el estado actual del clúster y lo compara con el estado declarado en los manifiestos. Cualquier divergencia dispara acciones automáticas para reconciliar ambos estados, garantizando que el sistema converge hacia la configuración declarada. Esto convierte a Kubernetes en una plataforma declarativa y autoajustable, altamente alineada con los principios de IaC (Hightower, Burns & Beda, 2017).

El uso de manifiestos versionables en repositorios Git permite integrar estas configuraciones dentro de prácticas como GitOps, donde la infraestructura solo puede modificarse mediante cambios revisados y aprobados en repositorio. Esto introduce trazabilidad, auditoría y despliegues consistentes a través de herramientas como ArgoCD o FluxCD (Cornell, 2020).

En síntesis, los manifiestos de Kubernetes son una manifestación directa de IaC, ya que describen de manera declarativa los recursos del clúster, permiten la automatización total del ciclo de vida de la infraestructura y garantizan reproducibilidad a través del versionamiento y la reconciliación automática.

1.9 MiMoQ

MiMoQ (Microservices Metrics and Quality, ver Ilustración) es una plataforma de experimentación desarrollada como proyecto de grado en la Pontificia Universidad Javeriana (Alvarado, MiMoQ: A System for Experimentation of Microservice-Based Applications, 2024) específicamente diseñada para evaluar atributos de calidad en aplicaciones basadas en microservicios desplegadas en entornos Kubernetes. Su contribución principal consiste en automatizar de extremo a extremo el ciclo de vida de un experimento de rendimiento y calidad: creación de proyectos, definición y despliegue de la aplicación bajo prueba, configuración y ejecución de experimentos, generación automática de carga, recolección de métricas, agregación de resultados y visualización.

Ilustración 4. MiMoQ

A diferencia de las herramientas de monitoreo comerciales (New Relic, Sysdig) o stacks genéricos (ELK, Prometheus + Grafana aislados), MiMoQ ofrece un flujo de trabajo integrado y científicamente riguroso que garantiza repetibilidad, reproducibilidad y validez estadística de los resultados con mínima intervención manual, lo cual reduce significativamente el esfuerzo y el sesgo del experimentador

MiMoQ está construido sobre una arquitectura modular, diseñada para facilitar la ejecución de experimentos de calidad en entornos basados en microservicios. La plataforma integra componentes para orquestación, despliegue, generación de carga, monitoreo y visualización, coordinados desde un backend central.

1.9.1 Tecnologías base

- Orquestador: El sistema utiliza MicroK8s, una distribución ligera de Kubernetes adecuada para entornos de investigación y ejecución local. Su simplicidad y bajo consumo permiten crear entornos reproducibles para experimentación.
- Gestión de despliegues: MiMoQ genera Helm charts personalizados de manera dinámica, lo que permite modificar parámetros del despliegue como recursos, número de réplicas, entre otros.
- Generador de carga: La carga es producida mediante k6, ejecutado como un Job, garantizando aislamiento, repetibilidad y una integración total con el ecosistema de observabilidad.
- Monitoreo: La plataforma utiliza un stack robusto compuesto por:
  – Prometheus para scraping y almacenamiento de métricas.
  – node-exporter para mediciones del sistema operativo del nodo.
- Visualización: Grafana con dashboards automáticos generados por experimento.
- Backend: NestJS (Node.js/TypeScript) que actúa como orquestador central.
- Frontend: Angular 17+.
- Persistencia: PostgreSQL.

Aunque el estándar ISO/IEC 25010 define ocho características de calidad del software, MiMoQ prioriza aquellas más relevantes para evaluar sistemas distribuidos basados en microservicios. Estas características permiten medir el comportamiento del sistema bajo diferentes condiciones de carga, estabilidad y elasticidad.

El desempeño es el atributo central evaluado por MiMoQ, ya que determina la capacidad del sistema para responder eficientemente bajo diferentes niveles de carga. En arquitecturas de microservicios, el desempeño está directamente relacionado con la experiencia del usuario, el tiempo de respuesta, el aprovechamiento de recursos. MiMoQ concentra sus experimentos en medir de manera precisa, repetible y automatizada cómo evoluciona el rendimiento del sistema bajo distintas condiciones. (Alvarado, Sistema de experimentación para evaluar atributos de calidad en microservicios: MiMoQ, 2024)

A partir de los resultados, MiMoQ establece correlaciones entre carga, latencia, errores y uso de recursos, lo que permite identificar con precisión los factores que limitan el desempeño y orientar decisiones de optimización basadas en datos.

1.9.2 Arquitectura

MiMoQ está diseñada como una plataforma modular, contenerizada y alineada con principios de microservicios y automatización de experimentación. La solución se divide en tres grandes capas funcionales, cada una implementada como un componente independiente y orquestada en un entorno Kubernetes, tal como se describe en los repositorios:

Frontend (Interfaz de usuario), permite al usuario crear proyectos de experimento, definir configuración, monitorizar la ejecución y comparar resultados. El repositorio https://github.com/murvn77/frontend-mimoq refleja esta implementación.

Backend (Lógica de negocio), alojada en https://github.com/murvn77/backend-mimoq Esta capa orquesta el ciclo del experimento: despliegue en Kubernetes, generación de carga con k6, monitoreo con Prometheus, visualización con Grafana, y persistencia de los resultados (usuario, proyecto, definición de experimento, métricas crudas y procesadas) en una base de datos PostgreSQL

El clúster Kubernetes en donde se despliegan tanto los servicios del propio MiMoQ (backend, frontend, base de datos) como los servicios de experimento (microservicios bajo prueba, generadores de carga, exportadores de métricas, etc.

A continuación se detallan los componentes clave y su interacción:

COMPONENTE	FUNCIÓN PRINCIPAL	TECNOLOGÍA
INTERFAZ DE USUARIO	Permite al usuario iniciar, configurar y visualizar experimentos.	Angular 17+ (repositorio frontend-mimoq)
API / LÓGICA DE NEGOCIO	Recibe peticiones de UI, valida configuraciones, despacha cargas de trabajo, coordina Kubernetes, recoge métricas, controla repeticiones, calcula intervalos de confianza.	NestJS (repositorio backend-mimoq)
GENERADOR DE CARGA	Ejecuta escenarios de carga definidos (por ejemplo con k6) en el sistema bajo prueba.	k6
EXPORTADORES / MONITOREO	Prometheus recoge métricas en tiempo real del clúster y de los componentes; Grafana genera paneles automáticos.	Prometheus + Grafana
PERSISTENCIA DE DATOS	Almacena usuarios, proyectos, definiciones de experimentos, resultados crudos y agregados para análisis longitudinal.	PostgreSQL
CLÚSTER DE ORQUESTACIÓN	Despliega todos los componentes anteriores, además del sistema bajo prueba, permite escalar, reiniciar, medir tiempos de escalado.	Kubernetes

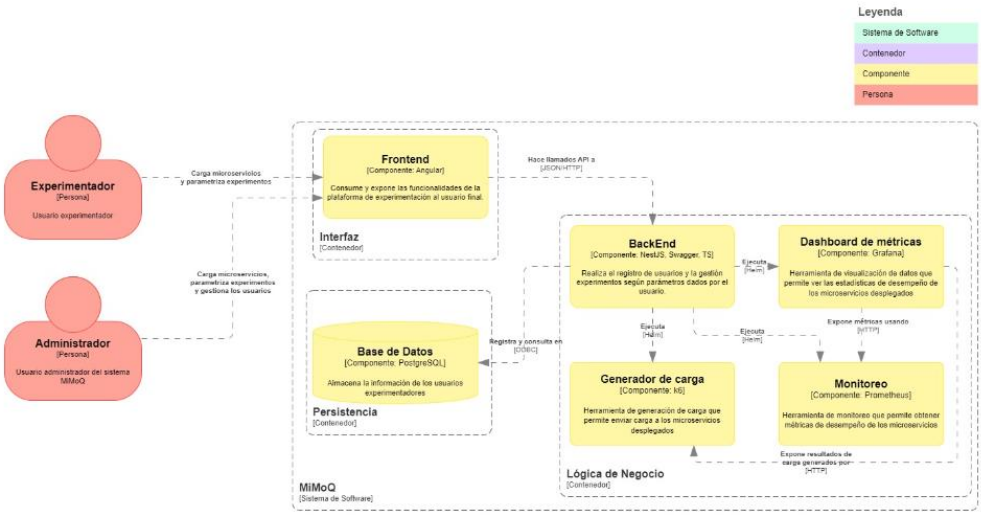

Ilustración 5. Diagrama de componentes (Alvarado, Sistema de experimentación para evaluar atributos de calidad en microservicios: MiMoQ, 2024)

1.9.2.1 Funcionalidades diferenciales de MiMoQ

Ejecución repetida con control estadístico: permite definir número de repeticiones

Tableros automáticos ricos: genera paneles Grafana preconfigurados con gráficas de latencia, utilización de recursos, escalado HPA, tasa de error, etc.

Almacenamiento persistente y versionado de resultados: facilita meta-análisis y comparación temporal.

En conclusión, MiMoQ se posiciona como una plataforma de experimentación para la evaluación de calidad en microservicios, superando las limitaciones de herramientas aisladas al ofrecer un flujo de trabajo automatizado, reproducible y alineado con estándares internacionales de calidad de software (ISO 25010) y metodologías de investigación en ingeniería de rendimiento. Constituye, por tanto, una herramienta de referencia tanto para investigación académica como para equipos de SRE y plataforma en entornos productivos.

1.9.2.2 Limitaciones

Dependencia fuerte de un entorno específico (MicroK8s)

El backend hace referencia explícita a despliegues en MicroK8s, lo que puede limitar la portabilidad hacia otros clústeres Kubernetes (kubeadm, GKE, EKS, etc.). Esta dependencia puede introducir dificultades a la hora de replicar los experimentos en entornos diferentes al de un desarrollo local.

Uso de procesos externos en lugar del SDK oficial de Kubernetes y Helm

La arquitectura actual delega la creación y gestión de objetos de Kubernetes y Helm mediante la ejecución de comandos externos a través de procesos, en lugar de utilizar los SDK oficiales. Este enfoque introduce fragilidad, aumenta la dependencia del entorno, dificulta el manejo de errores, limita la observabilidad interna y reduce la robustez del sistema ante variaciones en versiones o configuraciones de `kubectl` y `helm`. Además, al no existir una capa de abstracción tipada ni un control transaccional directo, se incrementa la complejidad para garantizar consistencia en despliegues. Esta limitación señala la necesidad de un rediseño arquitectónico que incorpore clientes nativos, con el fin de mejorar mantenibilidad, portabilidad y resiliencia de la plataforma.

Control limitado de configuración de variables del entorno

Al desplegar experimentos, es probable que haya variables de entorno (por ejemplo: tamaño del clúster, configuración de autoscaling, recursos reservados) que no están completamente parametrizadas o versionadas. Esto puede afectar la reproducibilidad absoluta de los experimentos si el entorno cambia.

Capa de visualización acoplada a herramientas específicas

El sistema está diseñado para generar automáticamente tableros con Grafana preconfigurados. Sin embargo, si el usuario quisiera otro motor de visualización, la arquitectura puede no soportarlo fácilmente. Esto puede limitar la extensibilidad o adaptación a equipos que usen otros stacks (por ejemplo, Kibana, Power BI, etc.).

Dependencia tecnológica relativamente alta / riesgo de obsolescencia

Uso de tecnologías específicas: Angular 17+, NestJS, PostgreSQL, Prometheus, Grafana, k6. Si alguno de estos cambia mucho, o si la plataforma experimenta cambios mayores (por ejemplo, nueva versión de Kubernetes), la herramienta podría requerir mantenimiento significativo. Esto implica un coste técnico y de evolución que debe considerarse en la ruta de la herramienta.

Documentación y accesibilidad del repositorio no están alineados

Dado que los repositorios son funcionales y reflejan una implementación, es posible que la documentación (README, guías de instalación, ejemplos de experimentos) no cubra todos los escenarios o tenga supuestos implícitos del entorno. Esto limita la adopción por otros investigadores o equipos de SRE. Esa barrera puede afectar la reproducibilidad externa del sistema.

Dependencia estricta de Docker Compose y de convenciones especificas

MiMoQ depende fuertemente de Docker Compose para ejecutar experimentos, lo que introduce restricciones significativas en la portabilidad y adopción de la herramienta. Para que un proyecto pueda ser utilizado dentro de MiMoQ, debe contar obligatoriamente con un archivo `docker-compose.yml`; otros nombres válidos en el estándar, como `docker-compose.yaml` o `compose.yml`, no son reconocidos por la plataforma. Además, los servicios definidos en el archivo deben incluir explícitamente el atributo `container_name`, y las imágenes utilizadas deben estar disponibles en un registry remoto (por ejemplo, Docker Hub); de lo contrario, el proceso de ejecución falla. Esta dependencia rígida de convenciones específicas reduce la flexibilidad, limita la compatibilidad con proyectos que utilizan configuraciones distintas y aumenta el esfuerzo necesario para adaptar experimentos externos a MiMoQ.

1.10 Trabajos relacionados

1.11 New Relic

New Relic es una plataforma de observabilidad diseñada para monitorear aplicaciones en tiempo real mediante métricas, trazas distribuidas, eventos y logs. Su propósito principal es proporcionar visibilidad continua del comportamiento del sistema en entornos productivos, permitiendo identificar anomalías, cuellos de botella y degradaciones en el rendimiento. La herramienta se destaca por su capacidad de integrar datos provenientes de múltiples servicios y generar paneles que facilitan el análisis operacional.

En contraste con MiMoQ, New Relic no está orientada a la ejecución controlada de experimentos, sino al monitoreo permanente. Mientras MiMoQ estructura el proceso de generación de carga, recolección de métricas y evaluación según atributos del modelo de calidad, New Relic se concentra en el seguimiento del sistema en su estado natural. Esta diferencia sitúa a MiMoQ como una herramienta más adecuada para la investigación experimental y la evaluación comparativa de escenarios diseñados por el usuario. (New Relic., 2023)

1.12 Stack ELK (Elasticsearch, Logstash, Kibana)

ELK Stack es un conjunto de herramientas enfocado en la gestión y análisis de logs. Elasticsearch permite el almacenamiento y búsqueda eficiente de grandes volúmenes de datos, Logstash posibilita el procesamiento y transformación de flujos de información, y Kibana

ofrece capacidades de visualización avanzadas. El ecosistema se utiliza ampliamente para diagnosticar comportamientos anómalos y analizar métricas en tiempo real.

Aunque ELK puede complementar pruebas de rendimiento, no incluye mecanismos nativos para automatizar la generación de carga ni para relacionar métricas con atributos de calidad específicos. En MiMoQ, estos elementos se encuentran integrados en un único flujo experimental. Esto permite que los resultados no solo se registren, sino que se interpreten dentro de un marco metodológico, reduciendo la dependencia de herramientas externas como ELK para consolidar datos. (Elastic, 2022)

1.13 WiseToolKit

WiseToolKit es un framework orientado a la automatización de pruebas funcionales en sistemas distribuidos. Facilita la ejecución de casos de prueba que verifican comportamientos esperados y validan la interacción entre componentes. Su diseño está enfocado en asegurar la correcta operación del software desde una perspectiva funcional, permitiendo a los desarrolladores incorporar pruebas en los flujos de integración continua.

En contraste, MiMoQ apunta a la evaluación de atributos dinámicos de calidad, como rendimiento, eficiencia operativa o estabilidad bajo carga. Mientras WiseToolKit se centra en lo que el sistema debe hacer, MiMoQ se enfoca en cómo se comporta cuando se somete a condiciones variables y escenarios experimentales. Esto coloca a MiMoQ como una solución más orientada al análisis cuantitativo que al aseguramiento funcional. (Rodríguez, 2019)

1.14 JMeter

Apache JMeter es una herramienta ampliamente utilizada para pruebas de carga y estrés. Permite simular usuarios concurrentes, generar distintos tipos de solicitudes y recopilar métricas como tiempos de respuesta, throughput o consumo de recursos. Su versatilidad y extensibilidad la convierten en un estándar de facto en escenarios de evaluación de rendimiento.

No obstante, JMeter no estructura el análisis dentro de un marco basado en atributos de calidad. La información generada debe ser procesada e interpretada por el usuario o integrada con sistemas externos. MiMoQ, en contraste, combina la ejecución de carga con un proceso metodológico que guía al usuario en la selección y análisis de métricas, ofreciendo una interpretación más alineada con el modelo de calidad. (Apache Software Foundation, 2023)

1.15 Locust

Locust es una herramienta de pruebas de carga basada en Python que permite describir comportamientos de usuarios mediante scripts. Su enfoque declarativo facilita la creación de escenarios complejos y altamente personalizados. También se caracteriza por su capacidad de escalar horizontalmente y distribuir la carga entre múltiples máquinas.

Sin embargo, Locust se limita a la inyección de carga y no ofrece un modelo integrado para interpretar métricas según atributos de calidad. El análisis, interpretación y vinculación de

resultados queda a cargo del usuario. MiMoQ supera esta limitación al proporcionar un entorno experimental completo donde las métricas se relacionan directamente con propiedades de calidad seleccionadas. (Heyman, 2021)

1.16 Gatling

Gatling es una herramienta de pruebas de carga de alto rendimiento escrita en Scala. Se destaca por su motor eficiente, su modelo basado en actores y la capacidad de manejar grandes volúmenes de solicitudes. Su DSL permite definir escenarios complejos de manera estructurada y reproducible.

Aunque ofrece dashboards detallados, su foco continúa siendo el rendimiento. No incorpora automáticamente modelos que conecten los resultados con atributos como fiabilidad, eficiencia energética o capacidad de recuperación. MiMoQ se diferencia al integrar estos elementos en un proceso más orientado a la investigación y evaluación multidimensional. (Gatling Corp, 2022)

1.17 Artillery

Artillery es una herramienta moderna para pruebas de carga y caos en sistemas distribuidos. Su configuración basada en archivos YAML y su arquitectura extensible permiten simular escenarios realistas, incorporando fallos controlados para observar el comportamiento del sistema bajo condiciones adversas.

A pesar de sus capacidades, Artillery sigue centrada en la generación de carga y la inyección de fallos, pero no en el análisis sistemático de calidad. MiMoQ, por su parte, permite conectar estos resultados con un modelo formal y seleccionar métricas que reflejen atributos específicos del sistema evaluado. Esto la convierte en una herramienta más completa para la experimentación académica (Hargreaves, 2021)

7. IMPLEMENTACIÓN DE LA SOLUCIÓN

La presente sección describe detalladamente el proceso de implementación de la solución propuesta a partir del estado actual de MiMoQ, abordando tanto la refactorización integral de la base existente como el desarrollo de nuevas funcionalidades que amplían significativamente las capacidades originales de la herramienta. En primer lugar, se presentan los cambios introducidos durante la refactorización y las mejoras logradas en los módulos de generación de carga, gestión de métricas y procesamiento de resultados. Finalmente, se describen las dos extensiones principales incorporadas en este trabajo: el módulo de inyección de fallas (sección 0), que habilita experimentos orientados a evaluar la resiliencia del sistema bajo condiciones adversas, y el uso de los manifiestos de Kubernetes como Infraestructura como Código (IaC) (sección 0)**,** que permite su despliegue en un clúster on-premise y la ejecución de pruebas en entornos controlados y reproducibles. Esta sección, en conjunto, evidencia la transformación de MiMoQ en una herramienta más robusta, flexible y alineada con prácticas modernas de ingeniería.

Herramientas de Desarrollo

Para mejorar la experiencia de Desarrollo e implementación de nuevas funcionalidades dentro de MiMoQ, se han incorporado varias herramientas que ayudan al desarrollo y mejora las practicas del mismo, entre ellas están:

Tilt

Es una herramienta que permite desarrollar microservicios usando Kubernetes, al automatizar la construcción de imágenes. (Tilt, 2025).

Tilt es una herramienta fundamental para mejorar el flujo de desarrollo local porque permite ejecutar todos los servicios del sistema dentro de un entorno de Kubernetes sin sacrificar la velocidad típica del desarrollo fuera de contenedores. Antes era necesario reconstruir imágenes manualmente, aplicar manifiestos a mano o mantener múltiples comandos y scripts; con Tilt, todo el ciclo se automatiza.

Las razones por las que Tilt benefició directamente la refactorización e implementación de las nuevas funcionalidades de MiMoQ incluyen:

- **Entorno local idéntico a producción:** todos los servicios corren en Kubernetes desde el principio, usando los mismos manifiestos que en despliegues reales.

- **Ciclos de desarrollo mucho más rápidos:** gracias al hot reaload, la cual es una función de desarrollo la cual permite que cualquier cambio en el código se refleja casi de inmediato sin reconstruir imágenes completas.
- **Un único comando para levantar todo el sistema:** con `tilt up` se inician frontend, backend, base de datos, módulos de caos y generadores de carga sin pasos adicionales.
- **UI centralizada para desarrollo:** Tilt muestra logs, estados de pods, errores de build y eventos del clúster en tiempo real.
- **Elimina la variabilidad del entorno local:** evita el clásico "en mi máquina sí funciona", porque todos los desarrolladores usan un único flujo de ejecución consistente.

La siguiente ilustración muestra el estado actual de los diferentes pods desplegados en el entorno de Tilt, y sus respectivos logs, esta es la vista de la UI centralizada donde podemos reiniciar, bloquear y ver las características de los diferentes componentes desplegados en el clúster de Kubernetes

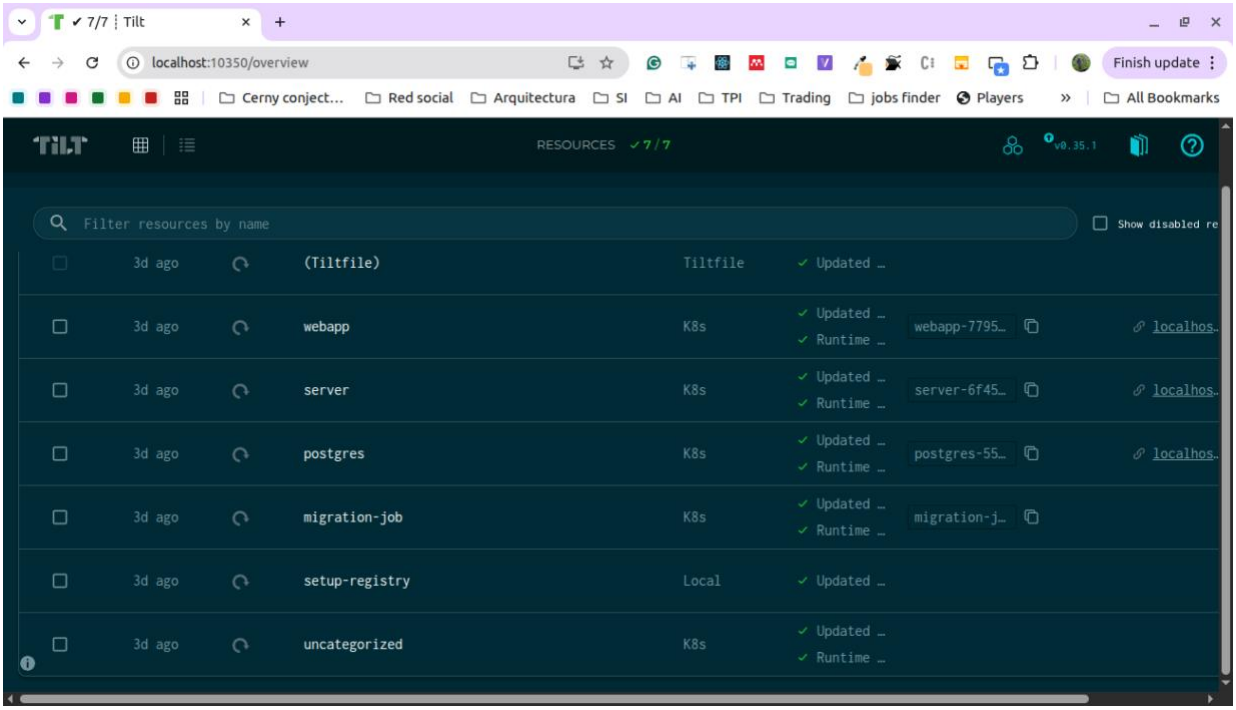

Ilustración 6. UI centralizada de Tilt

Al centralizar todo en Kubernetes y con la ayuda de Tilt para un desarrollo local rápido, la refactorización permitió:

- **Consistencia entre entornos:** lo que se ejecuta localmente es igual a lo que se ejecuta en el clúster on-premise.

- **Reproducibilidad:** ausencia de configuraciones manuales, cada servicio está definido declarativamente.
- **Escalabilidad real:** los test de carga pueden aumentar réplicas del backend sin intervención manual.
- **Resiliencia:** reinicios automáticos, probes de salud, reprogramación de pods.
- **Métricas integradas:** Prometheus puede extraer automáticamente las métricas de todos los servicios sin configuraciones externas.
- **Aislamiento:** cada componente corre en su propio pod, independiente y seguro.
- **Menos comandos por consola:** se elimina la necesidad de ejecutar múltiples comandos independientes como `npm stop`, `nest start`, `psql`, `k6 run`, etc.

Objetivos de la refactorización

El proceso de refactorización de MiMoQ tuvo como propósito transformar una implementación previa, difícil de desplegar y con componentes ejecutados manualmente en una plataforma coherente, mantenible y completamente automatizada. El objetivo principal de la refactorización fue unificar todos los servicios (frontend Angular, backend NestJS, base de datos PostgreSQL, generación de carga, módulo de métricas e inyección de fallas) dentro de un ecosistema Kubernetes que soportara despliegues reproducibles, escalables y consistentes.

La refactorización también busca:

- Mejorar la experiencia del usuario final durante la ejecución de pruebas de carga.
- Normalizar los entornos de desarrollo, pruebas y producción.
- Eliminar configuraciones manuales.
- Aumentar la capacidad para experimentar con fallas, métricas y cargas simultáneas.
- Preparar la plataforma para ser desplegada como Infraestructura como Código.

La migración hacia un mono repo permitió centralizar todos los componentes del sistema (frontend, backend, scripts, manifiestos de Kubernetes, configuración de métricas y módulos adicionales) dentro de un único repositorio. Este enfoque ofrece ventajas claras (Foster, 2025)

- **Coherencia entre módulos:** todos los cambios se versionan juntos, evitando desfases entre servicios.
- **Fácil reutilización:** configuraciones, scripts, librerías compartidas y manifiestos se almacenan en un solo lugar.
- **Mejor mantenimiento:** un único pipeline, un único repositorio, un único historial.
- **Facilita la IaC:** todo lo necesario para reconstruir el sistema está contenido en el mismo repo.
- **Ambientes replicables:** cualquier persona puede clonar el repositorio y tener la plataforma lista para correr.

Arquitectura

La nueva arquitectura de MiMoQ se cambió al implementar todos los servicios dentro del clúster de Kubernetes. Esta decisión permitió eliminar una de las principales limitaciones del

diseño anterior: la heterogeneidad de entornos y la dependencia de múltiples procesos por consola para ejecutar y coordinar cada parte del sistema.

En el enfoque previo, el backend debía ejecutarse fuera de Kubernetes, mientras que los usuarios solo desplegaban los escenarios de prueba dentro del clúster. Esto generaba una separación artificial que complicaba la reproducibilidad, aumentaba la superficie de error y dificultaba la integración con herramientas como Prometheus, k6 o los agentes de caos. Además, obligaba a que el estudiante o usuario ejecutara comandos manuales por consola para iniciar servicios, cargar scripts o levantar la aplicación, lo que introducía una variabilidad innecesaria y afectaba la consistencia de las pruebas.

La nueva arquitectura resuelve completamente estas limitaciones al adoptar Kubernetes como plataforma universal para todos los componentes del sistema. Este enfoque se basa en varias razones técnicas fundamentales:

- **Cohesión operativa**

Al estar todos los componentes dentro del mismo clúster, el backend, los generadores de carga, la base de datos, Prometheus y los agentes de caos se comunican mediante servicios internos de Kubernetes, evitando configuraciones externas y eliminando diferencias entre entornos locales y de producción.

- **Reproducibilidad total**

La arquitectura actual elimina completamente los pasos manuales. Todo —desde la aplicación web hasta los experimentos de caos— se instancia mediante manifiestos declarativos. Esto garantiza que dos usuarios pueden reproducir exactamente la misma arquitectura con solo usar `kubectl apply -k`, un beneficio esencial para entornos académicos y de investigación como MiMoQ.

- **Escalabilidad y aislamiento**

Kubernetes permite escalar de manera independiente cada componente de la arquitectura, una capacidad esencial para entornos de pruebas de carga donde los recursos deben ajustarse dinámicamente según la demanda. En este contexto, k6 puede ejecutarse como un *Job* aislado y efímero, mientras que el backend tiene la posibilidad de escalar horizontalmente para absorber picos de tráfico generados durante los experimentos. A su vez, la base de datos se despliega como un *StatefulSet* con almacenamiento persistente, garantizando estabilidad y consistencia incluso bajo condiciones de estrés. Este nivel de automatización y elasticidad habría sido complejo, manual o directamente impráctico si los servicios se ejecutaran fuera del clúster, sin las capacidades nativas de orquestación de Kubernetes.

- **Modelado explícito de la infraestructura**

Antes, muchos servicios existían solo como comandos dispersos, lo que hacía difícil entender o extender la arquitectura.

Ahora, la arquitectura es código: cada Deployment, Service, PVC, Job o ConfigMap describe explícitamente la estructura del sistema. Esto hace la plataforma más robusta y fácil de mantener.

- **Portabilidad garantizada**

La arquitectura es independiente del proveedor. Puede ejecutarse en:

- minikube
- kind
- microk8s
- clústeres on-premise con kubeadm (como el utilizado por la universidad)
- cualquier cloud provider

Sin reescribir servicios ni scripts.

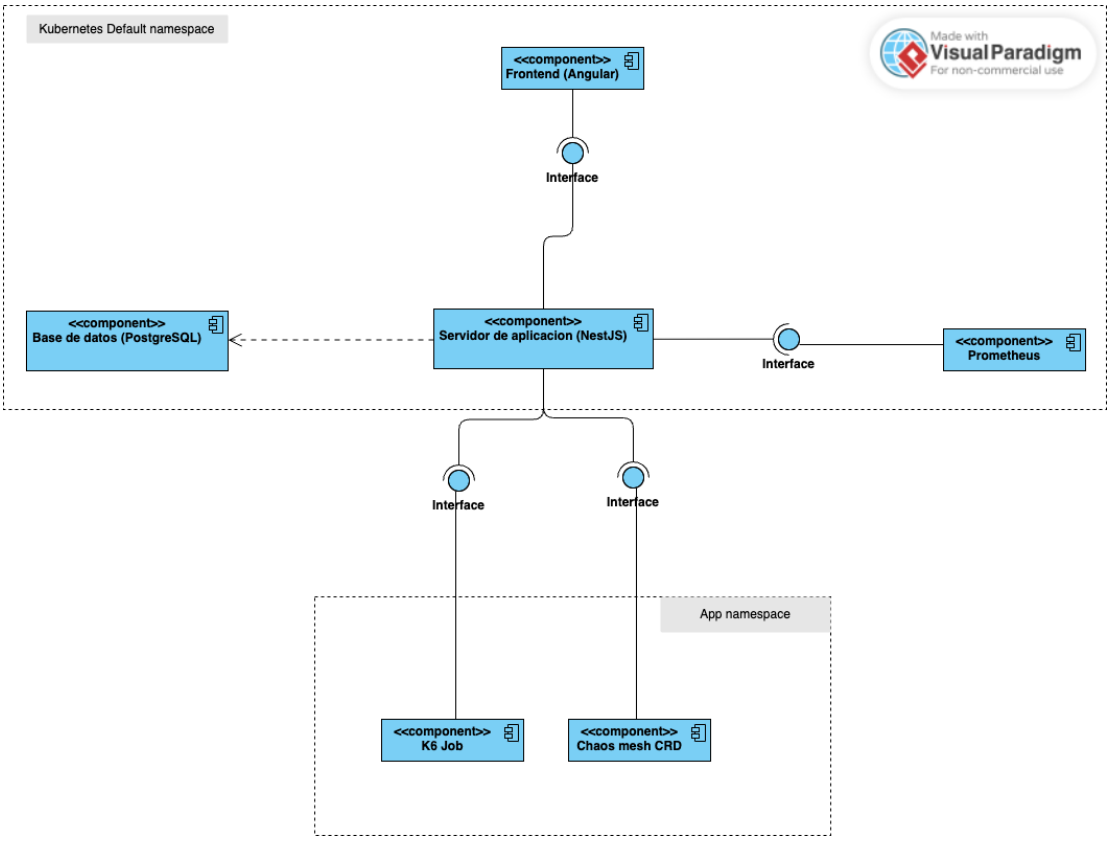

Ilustración 7. Diagrama de componentes del sistema

La nueva arquitectura convierte a MiMoQ en una plataforma moderna, portable, reproducible y alineada con prácticas actuales de Ingeniería de Software, Kubernetes, SRE y Chaos Engineering. En la Ilustración 7. Diagrama de componentes del sistema se presenta un diagrama de componentes que refleja la organización general del sistema y las interacciones entre sus módulos principales. Esta arquitectura permite a cualquier usuario ejecutar un entorno completo sin intervención manual, simplifica el despliegue, habilita escalabilidad real y garantiza que la infraestructura describa de forma precisa el funcionamiento del sistema.

Con la nueva arquitectura, los flujos internos de la aplicación presentan variaciones significativas en comparación con la primera versión de MiMoQ. A continuación, se exponen los diagramas de proceso que describen las operaciones esenciales del sistema: la creación de un proyecto, la definición y ejecución de un despliegue, y el ciclo completo de definición y

ejecución de un experimento. Estos diagramas permiten visualizar cómo interactúan el usuario con el sistema y cómo opera cada flujo dentro de la aplicación.

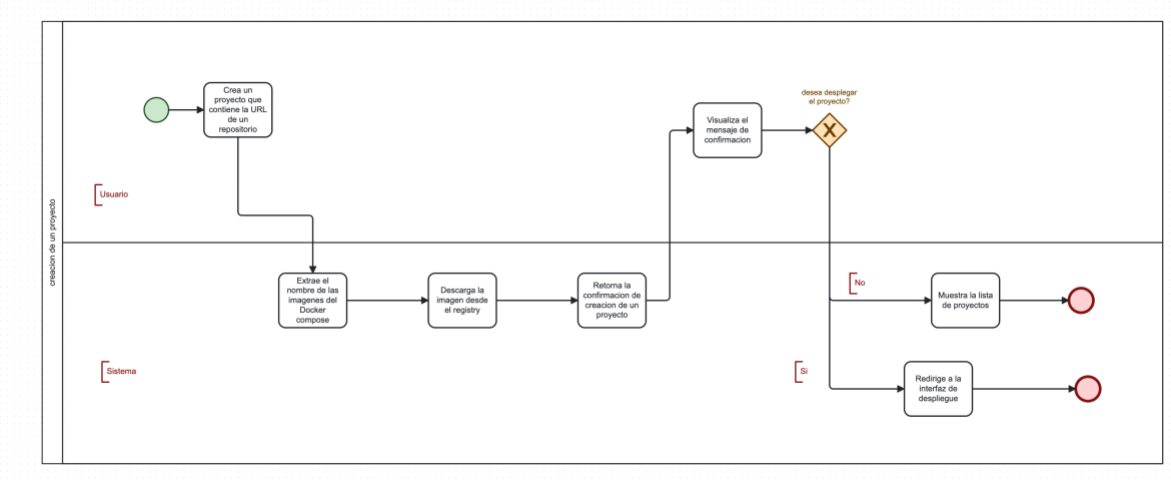

Ilustración 8. Diagrama de procesos para la creación de un proyecto

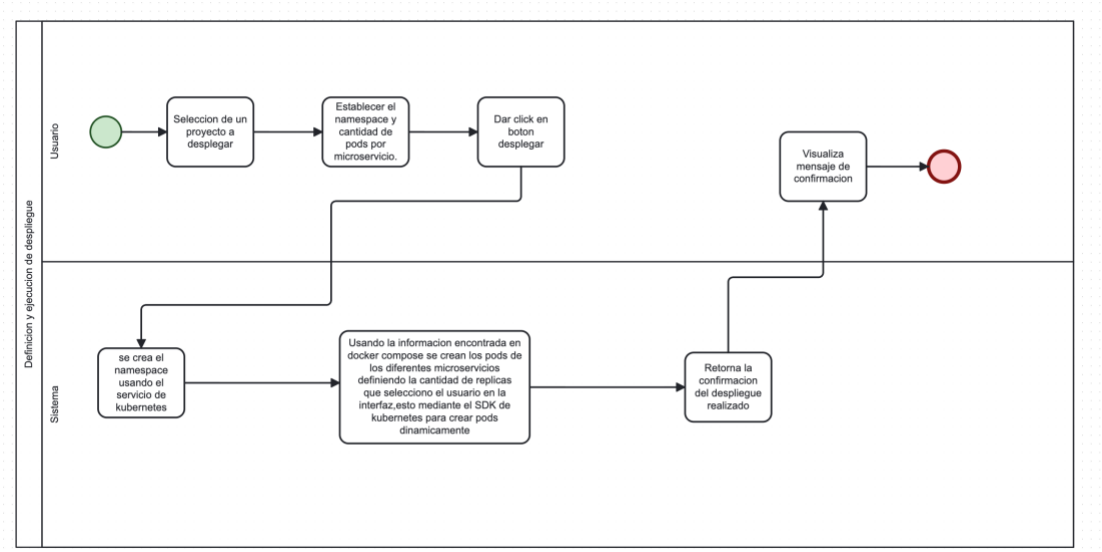

Ilustración 9 Definición y ejecución de un despliegue

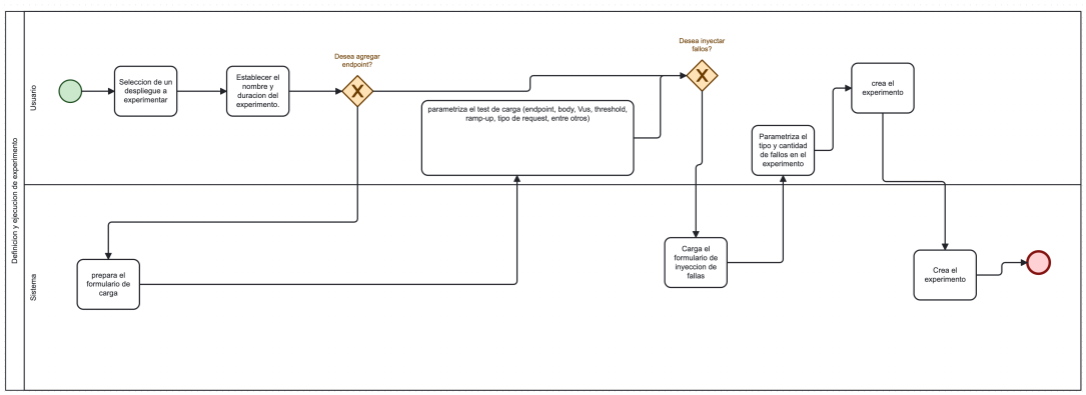

Ilustración 10 Definición y ejecución de un experimento

Generación de Carga

La generación de carga en MiMoQ constituye el centro del proceso experimental, permitiendo simular escenarios variados de uso del sistema mediante la ejecución de solicitudes concurrentes hacia uno o varios endpoints definidos por el usuario. A diferencia de la versión inicial de la herramienta, que utilizaba configuraciones estáticas y predefinidas, la versión mejorada incorpora un formulario interactivo que habilita la creación de pruebas de carga completamente personalizadas.

Este formulario permite al usuario especificar de manera precisa parámetros clave como el endpoint objetivo, el número de usuarios virtuales (VUs), la duración del experimento, el tipo de patrón de carga (rampa, constante, explosiva, etc.) y los **thresholds** que determinan los criterios de éxito o degradación del sistema. Esta capacidad no solo democratiza la construcción de experimentos, sino que habilita una mayor reproducibilidad, dado que cada configuración queda explícitamente registrada por el sistema.

Además, MiMoQ ahora soporta la definición de múltiples endpoints, lo cual amplía significativamente el alcance de los experimentos. El usuario puede subir una lista de rutas o cargar varios endpoints a través del formulario, habilitando pruebas más cercanas a un comportamiento realista donde los usuarios escogen los distintos servicios o funcionalidades del sistema que quieran probar. Cada endpoint puede incluir métodos HTTP específicos, payloads personalizados y estrategias de distribución del tráfico entre rutas (Ver Ilustración 11)

Finalmente, la definición de thresholds directamente desde la interfaz permite incorporar criterios automáticos de éxito o fallo del experimento. Estos thresholds, como máximos de latencia, tasas de error o percentiles de respuesta, guían el análisis posterior y habilitan un modelo más formal de evaluación del comportamiento del sistema bajo carga. Gracias a esta integración, MiMoQ no solo ejecuta carga, sino que también facilita experimentación sistemática y alineada con métricas de calidad del software.

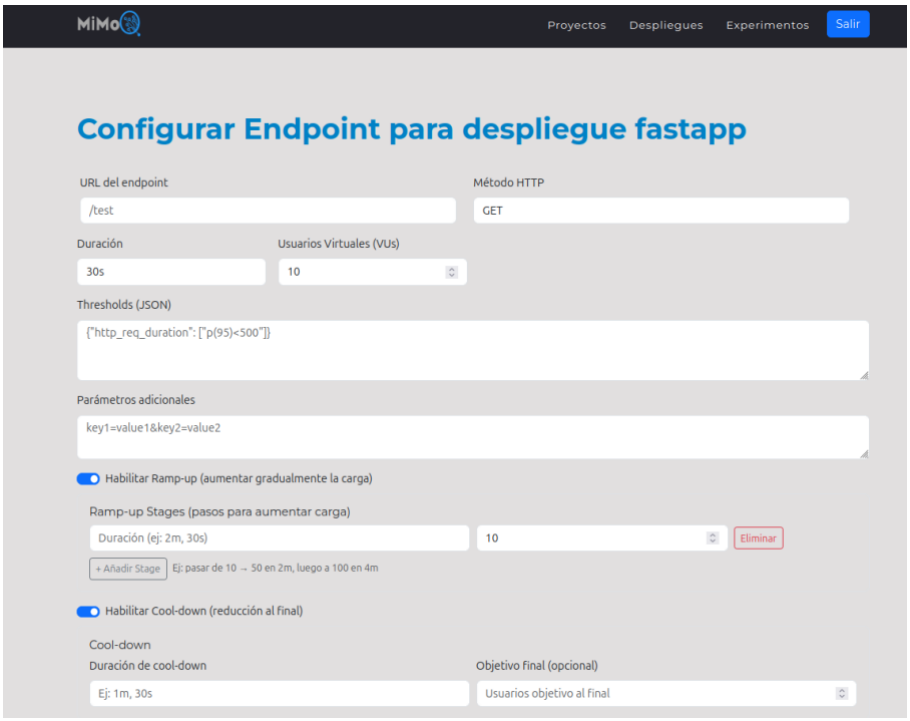

Ilustración 11. Vista para configurar carga en un endpoint

Módulo de métricas

Con la adopción de la nueva arquitectura, el módulo de métricas fue sometido a un rediseño integral que refuerza su relevancia como componente esencial de MiMoQ, al encargarse de la recolección, consulta, interpretación y exportación de información sobre el desempeño de los experimentos de carga

El módulo de métricas constituye uno de los componentes fundamentales en MiMoQ, dado que posibilita la recolección, consulta, interpretación y exportación de información cuantificable relativa al desempeño de los experimentos de carga. Este módulo está diseñado sobre la base de Prometheus, un sistema de monitorización basado en series temporales, altamente utilizado en arquitecturas nativas en la nube. La integración se lleva a cabo mediante consultas HTTP hacia la API `/api/v1/query` y `/api/v1/query_range`, permitiendo obtener muestras instantáneas y rangos temporales, respectivamente.

La implementación realizada en MiMoQ se basa en el archivo `prometheus.service.ts`, cuya responsabilidad recae en abstraer el acceso a Prometheus, construir consultas PromQL, ejecutar las peticiones y retornar los resultados en un formato homogéneo que posteriormente es procesado para generar el archivo CSV del experimento. Esta sección describe en detalle las métricas recolectadas, su significado, la semántica de las consultas PromQL construidas y el rol que desempeñan en la interpretación de los experimentos.

El servicio `getK6MetricsForExperiment()` recolecta todas las métricas vinculadas a un experimento determinado a través de la etiqueta formada por el id el experimento

Cada métrica se consulta mediante PromQL empleando la forma:

```
metricName{experiment="experimentId"}
```

Todas las consultas se ejecutan dentro de un rango temporal (`query_range`), con un `step` de 15 segundos.

Para consultar el listado de todas las consultas que se ejecutan en Prometheus para obtener el archivo de resultados, consultar el Anexo 2

Inyección de fallas

La inyección de fallas en MiMoQ introduce la capacidad de evaluar la resiliencia y robustez del sistema mediante la simulación controlada de fallos. Este enfoque se basa en principios de *Chaos Engineering*, donde se perturbaciones deliberadas para observar cómo reacciona el sistema, identificar vulnerabilidades y validar estrategias de mitigación.

MiMoQ implementa mecanismos para provocar fallos en puntos específicos del flujo de carga, como degradaciones de latencia, respuestas erróneas, caídas de servicios simuladas o interrupciones parciales. Al integrarse directamente con el generador de carga, la inyección de fallas permite combinar estrés y errores simultáneamente, produciendo escenarios más cercanos a condiciones reales de producción.

Plataformas de Ingeniería del Caos

La ingeniería del caos se fundamenta en la introducción controlada de fallas en sistemas distribuidos con el objetivo de evaluar su resiliencia, identificar puntos débiles y validar supuestos operacionales. Existen múltiples plataformas que permiten automatizar estos experimentos, entre las cuales destacan Chaos Monkey**,** Chaos Mesh **y** Gremlin, cada una con enfoques, alcances y modelos de operación distintos.

- **Chaos Monkey**

Desarrollado originalmente por Netflix, está orientado a entornos en la nube y se especializa en la terminación aleatoria de instancias (Hochstein, 2017). Su alcance es limitado para sistemas complejos orquestados mediante Kubernetes, ya que no ofrece granularidad a nivel de pod, contenedor, red o almacenamiento. Su diseño, más cercano a un mecanismo de perturbación aleatoria, lo vuelve útil en ciertos contextos, pero insuficiente para plataformas que requieren reproducibilidad, trazabilidad y experimentos compuestos.

• Gremlin

Gremlin es una plataforma comercial de ingeniería del caos que ofrece un conjunto amplio y estructurado de ataques predefinidos, tales como latencia, pérdida de paquetes, limitación de CPU, saturación de memoria o interrupciones de red (Gremlin, 2025). Además, incorpora funciones avanzadas orientadas a la gobernanza organizacional, como control de accesos, roles, auditoría centralizada y flujos de aprobación, lo que facilita su adopción en empresas con altos estándares de seguridad y cumplimiento (Gremlin, 2025)

A pesar de su robustez, su naturaleza propietaria y su modelo de licenciamiento comercial restringen su uso en contextos académicos y de investigación, donde se priorizan herramientas abiertas, auditables y con soporte a integraciones automatizadas en pipelines reproducibles. Estas limitaciones vuelven a Gremlin poco adecuado para proyectos como MiMoQ, cuyo desarrollo se basa en código abierto, reproducibilidad y trazabilidad completa del proceso experimental

• Chaos Mesh

Es una plataforma de código abierto, nativa para Kubernetes y con soporte directo para múltiples categorías de fallas: de red, de CPU/memoria, de almacenamiento, de tiempo del sistema, e incluso fallas arbitrarias mediante scripts (PingCAP, 2025). Además, posee un modelo declarativo basado en CRDs (Custom Resource Definitions), lo que permite integrar la definición de fallas directamente dentro de los manifiestos del clúster. Esta característica lo vuelve idóneo para un sistema como MiMoQ, cuyo diseño se fundamenta en IaC y despliegues reproducibles. Su arquitectura modular, combinada con su interfaz de control granular, motivó que fuera la plataforma seleccionada para el módulo de inyección de fallas.

Característica	Chaos Monkey	Chaos Mesh	Gremlin
Tipo	Script / Servicio	Operador de Kubernetes	Plataforma SaaS / Enterprise
Licencia	Open Source (Netflix)	Open Source (CNCF)	Comercial (Gremlin Inc.)
Integración principal	AWS / VMs	Kubernetes nativo	Multinube, Kubernetes, bare metal
Nivel de control	Bajo (aleatorio)	Muy alto (declarativo)	Alto (controlado vía API/UI)
Enfoque	Simulación de fallos aleatorios en instancias	Inyección de fallas en pods, redes y tiempos	Inyección controlada de fallas con métricas y reporting

Además de las capacidades técnicas previamente descritas y explicadas en la tabla, la elección de Chaos Mesh respondió tanto a criterios funcionales como metodológicos del proyecto. Dado que MiMoQ se fundamenta en principios de infraestructura como código, despliegues reproducibles y trazabilidad completa del entorno experimental, resultaba indispensable adoptar una herramienta cuyas definiciones de fallas pudieran versionarse, declararse y auditarse al igual que el resto de los artefactos del sistema. Chaos Mesh, al operar mediante CRDs y ajustarse de manera nativa al ecosistema de Kubernetes, permite mantener una alineación conceptual y operativa entre los experimentos y la arquitectura general del clúster. Estos factores, combinados con su granularidad en la definición de fallas y su madurez dentro del entorno Kubernetes, hicieron que Chaos Mesh fuera la opción más adecuada para los objetivos experimentales de MiMoQ.

Diseño

El diseño del módulo de inyección de fallas se articula alrededor de un principio central: toda perturbación debe estar sincronizada con la duración del test de carga. Para ello, el módulo coordina los experimentos de Chaos Mesh de manera que éstos comienzan simultáneamente con la ejecución de los tests k6 y concluyen cuando finaliza la carga. Esta sincronización asegura que las métricas obtenidas —latencia, throughput, código de estado, tiempos de respuesta, saturación de recursos— reflejen el impacto real de las fallas sobre el sistema bajo estrés.

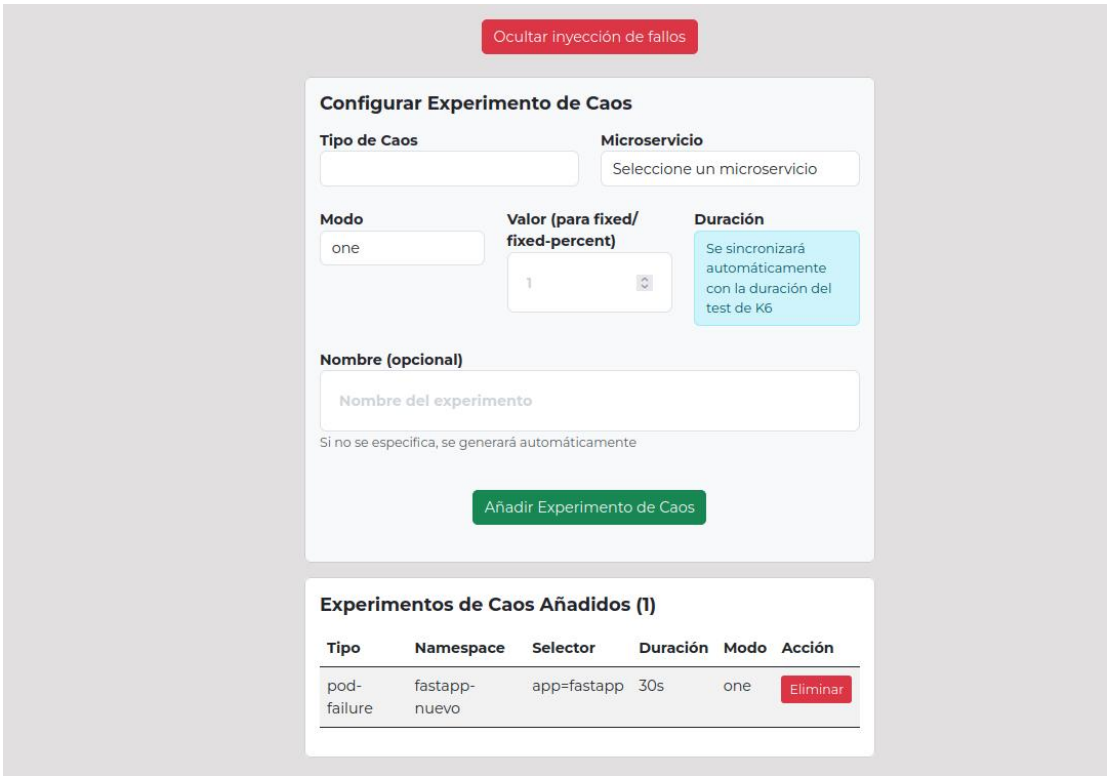

Ilustración 12. Creación de inyección de fallas

Como se puede observar en la ilustración, el módulo opera mediante una capa de orquestación que genera dinámicamente manifiestos de Chaos Mesh a partir de la configuración que define el usuario (tipo de falla, intensidad, duración). Una vez construidos estos manifiestos, se aplican al clúster Kubernetes junto con el test. Durante la ejecución, los CRDs de Chaos Mesh gobiernan su propio ciclo de vida, lo que permite un aislamiento claro entre los recursos del experimento y el resto de la plataforma.

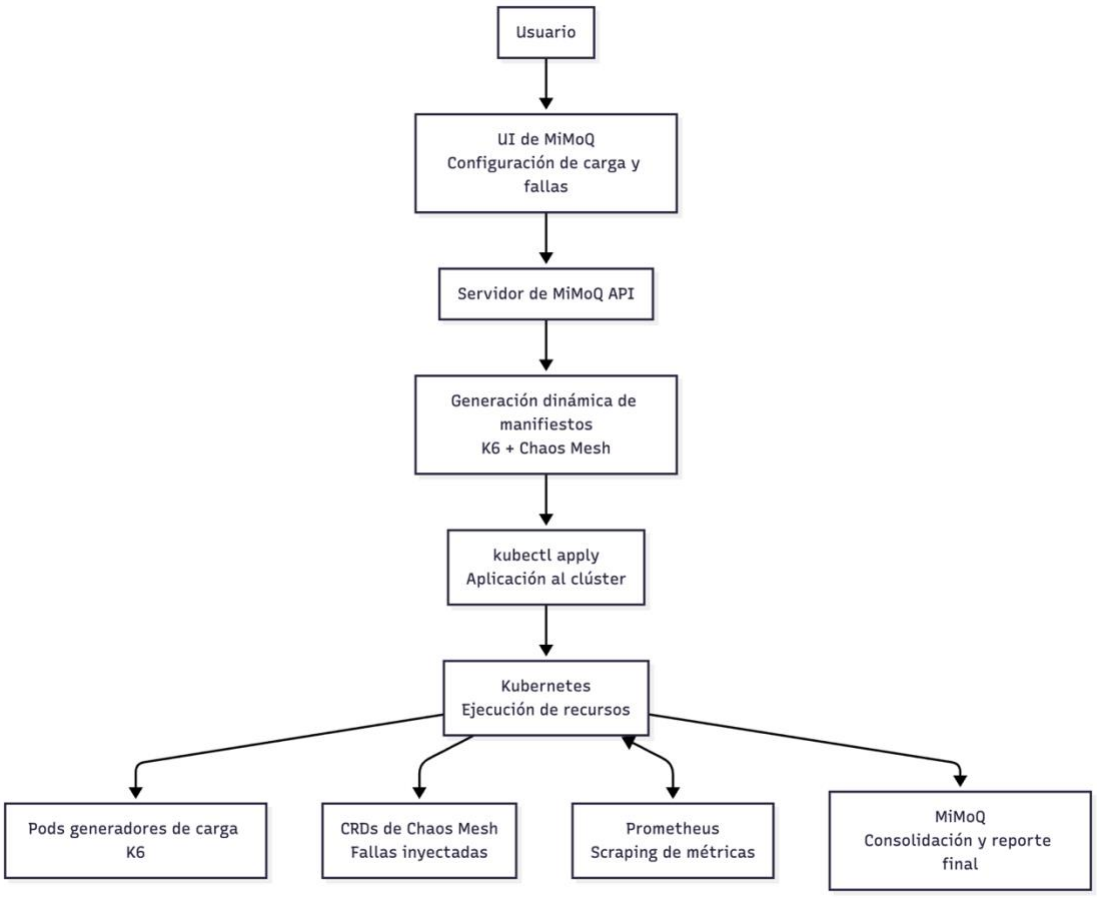

Ilustración 13. Diagrama de flujo de generación de reportes

Este diseño (ver ilustración) evita la necesidad de que el usuario ejecute comandos manuales o gestione recursos externos. Toda la infraestructura —incluyendo el generador de carga, el backend, la base de datos, el frontend y los experimentos de caos— convive dentro del mismo clúster, logrando repetibilidad, trazabilidad y una plataforma completamente autónoma

Implementación

La implementación del módulo de inyección de fallas se realizó utilizando Chaos Mesh instalado mediante Helm**,** metodología coherente con el enfoque de IaC adoptado en el proyecto. Helm brinda control de versiones, reproducibilidad de la instalación y facilidad para ajustar valores mediante archivos de configuración

El proceso de instalación se integró directamente en el entorno Kubernetes que MiMoQ utiliza para ejecutar tanto el sistema como los experimentos. El chart oficial de Chaos Mesh se desplegó en un *namespace* dedicado, garantizando aislamiento y evitando interferencias con otros componentes de la plataforma. Además, el uso de CRDs permite que las fallas puedan

representarse mediante manifiestos YAML que luego MiMoQ genera y aplica de forma programática desde su backend.

Para crear cada falla, MiMoQ construye un documento YAML que describe el tipo de falla (por ejemplo, `NetworkChaos`, `PodChaos`, `DNSChaos`), sus parámetros (`loss`, `latency`, `corruption`, `mode`, `selector`), y la duración del ataque. Posteriormente, el sistema aplica estos manifiestos en el clúster utilizando comandos equivalentes a `kubectl apply`. Kubernetes y Chaos Mesh se encargan del ciclo de vida del experimento, incluyendo su inicio, monitoreo y finalización automática.

Tipos de Inyección de fallos

Los tipos de fallos disponibles actualmente en MiMoQ, que se pueden configurar directamente desde la interfaz gráfica de la aplicación son:

POD_FAILURE

El caso de falla *Pod Failure* simula una condición en la cual uno o varios pods dejan de estar operativos debido a una falla interna que impide que sus contenedores se ejecuten correctamente. Esta situación provoca que los pods afectados permanezcan en estados como *Failed* o *NotReady*, generando interrupciones parciales o totales en el servicio que prestan. Su utilidad radica en evaluar la resiliencia de aplicaciones distribuidas frente a fallas no catastróficas pero persistentes, permitiendo analizar si los mecanismos de autorrecuperación, balanceo de carga, readiness probes y supervisión del sistema son capaces de manejar fallos prolongados sin degradar completamente la disponibilidad del sistema

POD_KILL

El caso de falla *Pod Kill* termina de forma abrupta uno o varios pods, sin permitir un proceso de apagado controlado. Esto provoca la destrucción inmediata de las instancias afectadas, obligando al orquestador a crear nuevas réplicas para mantener el estado deseado. Este tipo de falla es útil para evaluar cómo reaccionan las aplicaciones ante caídas súbitas, especialmente cuando dependen de sesiones, conexiones persistentes o transacciones en curso. Permite estudiar la efectividad de los controladores de Kubernetes para restaurar el servicio rápidamente y analizar la robustez del sistema ante fallos inesperados.

CONTAINER_KILL

El caso de falla *Container Kill* introduce un fallo súbito en uno de los contenedores dentro de un pod, sin afectar al resto de contenedores que coexistan en esa misma unidad. Esto provoca que el contenedor específico falle y se reinicie, mientras el pod continúa ejecutándose de manera parcial. Este tipo de perturbación es adecuado para estudiar arquitecturas que dependen de sidecars, contenedores auxiliares o patrones multi-contenedor, permitiendo verificar si la

aplicación puede seguir operando pese a un fallo interno y si la estrategia de reinicio configurada es suficiente para mantener la estabilidad general del servicio.

NETWORK_DELAY

El caso de falla *Network Delay* introduce latencia artificial en las comunicaciones de red asociadas a los pods objetivo. La consecuencia inmediata es el aumento en el tiempo de transmisión de los paquetes, lo que puede causar ralentizaciones perceptibles, timeouts en peticiones HTTP, degradación en el rendimiento de microservicios o interrupciones en procesos altamente dependientes de la sincronización. Su valor reside en permitir la emulación de condiciones reales de congestión o deterioro en redes distribuidas, facilitando la evaluación del comportamiento del sistema en situaciones adversas donde la calidad del enlace es pobre o variable.

NETWORK_LOSS

El caso de falla *Network Loss* provoca la pérdida aleatoria de un porcentaje de paquetes de red. Esto genera interrupciones intermitentes en la comunicación entre servicios, afectando la fiabilidad percibida y ocasionando reintentos, fallos parciales o inconsistencias temporales. Es especialmente útil para simular escenarios donde existen redes inestables, enlaces saturados o dispositivos defectuosos, permitiendo evaluar estrategias de reintento, políticas de tolerancia a fallos y la robustez general de sistemas que dependen de llamadas remotas o intercambios frecuentes de información.

NETWORK_BANDWIDTH

El caso de falla *Network Bandwidth* limita de manera artificial la cantidad de datos que puede transferirse desde o hacia los pods afectados. Esto provoca una reducción significativa en la velocidad de transferencia, generando retrasos en procesos de sincronización, carga lenta de recursos, o degradación en servicios que dependen de flujos de datos de mediana o alta capacidad. Su propósito es estudiar cómo se comporta una aplicación ante restricciones de ancho de banda, permitiendo identificar cuellos de botella y evaluar mecanismos adaptativos o estrategias de mitigación.

NETWORK_PARTITION

El caso de falla *Network Partition* crea una partición lógica en la red que impide la comunicación entre un conjunto de pods y el resto del sistema. El efecto es una separación total entre componentes, generando escenarios de tipo *split-brain*, donde distintas partes de la aplicación operan sin visibilidad mutua. Este experimento resulta fundamental en sistemas distribuidos que requieren consistencia, replicación o consenso, ya que permite analizar el funcionamiento bajo aislamiento, evaluar la corrección de los algoritmos de coordinación y determinar si las aplicaciones pueden recuperarse sin pérdida de datos o comportamientos anómalos.

CPU_STRESS

El caso de falla *CPU Stress* provoca una saturación artificial del uso de CPU en los pods objetivo. Durante este escenario, los contenedores consumen cantidades elevadas de ciclos de procesamiento, lo que causa aumentos en la latencia, bloqueos temporales y degradación en la capacidad de respuesta. Este tipo de perturbación es esencial para estudiar el comportamiento de aplicaciones bajo sobrecarga computacional, validar políticas de escalado automático basadas en CPU, y analizar cómo reaccionan los servicios cuando compiten por recursos críticos.

MEMORY_STRESS

El caso de falla *Memory Stress* genera un consumo intensivo de memoria dentro de los pods seleccionados, simulando condiciones similares a fugas de memoria o cargas inusualmente pesadas. Esto provoca que las aplicaciones operen con recursos escasos, pudiendo desencadenar eventos de *OOMKilled* (Out Of Memory) si se exceden los límites configurados. La finalidad es evaluar la tolerancia del sistema ante presiones de memoria, identificar comportamientos inesperados y determinar si los límites de recursos y las estrategias de recuperación son adecuado

IO_STRESS

El caso de falla *I/O Stress* introduce una carga artificial sobre el sistema de entrada/salida, generando operaciones intensivas de lectura y escritura. Esto provoca lentitud en el acceso a disco, retrasos en consultas a bases de datos, mayores tiempos de respuesta y posible bloqueo temporal de procesos dependientes de almacenamiento. Se trata de un experimento valioso para evaluar el rendimiento bajo condiciones de almacenamiento degradado, analizar el impacto de la latencia del disco y estudiar la capacidad del sistema para mantener la integridad y disponibilidad de los datos ante escenarios adversos.

Finalmente, la integración estrecha entre Chaos Mesh y Prometheus (que recolecta métricas durante los experimentos) permite observar cómo cada falla afecta la resiliencia del sistema bajo condiciones de carga. Esto convierte a MiMoQ en una plataforma completa para estudiar fenómenos de degradación, latencia inducida, pérdida de disponibilidad y comportamientos transitorios, todo ello de manera sistemática y reproducible.

Infraestructura como Código (IaC)

En el contexto de MiMoQ, IaC es fundamental porque el sistema requiere coordinar múltiples componentes heterogéneos: backend, frontend, base de datos, pods de pruebas k6, Prometheus para métricas y Chaos Mesh para ingeniería del caos. Administrar todo esto manualmente implicaría una alta probabilidad de inconsistencias entre entornos, especialmente cuando el sistema debe ejecutarse en clústeres de investigación académica o laboratorios on-premise. Al adoptar IaC, cada componente del proyecto se expresa mediante manifiestos YAML versionados dentro del repositorio, permitiendo su reconstrucción completa a partir del código fuente.

Diseño de Kubernetes

Kubernetes proporciona un modelo declarativo para describir el estado deseado de un sistema distribuido. Esta característica es fundamental para IaC, ya que permite definir servicios, despliegues, volúmenes, configuraciones y permisos como código. En MiMoQ, todos los elementos de la plataforma —frontend, backend, base de datos, generadores de carga, Prometheus y Chaos Mesh— se ejecutan como recursos nativos de Kubernetes, siguiendo un diseño uniforme que evita dependencias externas o ejecuciones manuales.

A diferencia del enfoque inicial de MiMoQ, donde ciertos componentes se ejecutaban fuera del clúster (por ejemplo, ejecutar k6 por consola, backend separado o base de datos ad hoc), el nuevo diseño integra absolutamente todo dentro del mismo entorno orquestado. Esto elimina discrepancias entre entornos, simplifica la ejecución de experimentos y asegura que el comportamiento bajo carga o fallas sea representativo del sistema real

Kustomize cumple un rol esencial dentro del diseño. El repositorio contiene directorios `base/` para cada aplicación y `overlays/` para ambientes específicos como `dev` o `prod`. Kustomize permite componer estos manifiestos sin duplicación, aplicando parches, overlays y configuraciones personalizadas según el entorno. Este enfoque facilita extender MiMoQ en el futuro, agregar nuevos servicios o ajustar los existentes sin modificar la base declarativa.

Este diseño también habilita a MiMoQ para operar en múltiples plataformas de manera uniforme: clústeres on-premise, clústeres académicos, o entornos en la nube como GKE, EKS o AKS, manteniendo exactamente la misma estructura declarativa.

Implementación de los manifiestos de Kubernetes

MiMoQ utiliza una variedad de objetos nativos de Kubernetes, cada uno cumpliendo un propósito específico dentro de la arquitectura del sistema:

- **Deployments**: Utilizados para el servidor (`apps/server/k8s/base`) y el frontend (`apps/frontend/k8s/base`). Aseguran que siempre exista el número deseado de réplicas, permiten actualizaciones controladas y ofrecen una alta disponibilidad básica del sistema.

- **Services**: Exponen los componentes internos para que puedan comunicarse entre sí. El backend, frontend y la base de datos cuentan con sus propios servicios que abstraen la red interna del cluster y permiten descubrimiento automático.

- **ConfigMaps**: Permiten suministrar configuraciones específicas a las aplicaciones. Por ejemplo, k6 utiliza configuraciones definidas en YAML para parametrizar los experimentos.

- **PersistentVolumeClaims (PVCs)**: Usados principalmente por la base de datos MySQL (`apps/db/k8s/base`). Aseguran que los datos persistan más allá del ciclo de vida del pod, requisito indispensable en sistemas que almacenan información de experimentos.

- **StatefulSets** (si aplica en la base de datos): Gestionan pods con identidad estable y almacenamiento consistente, características importantes para bases de datos.

- **CronJobs / Jobs** (para componentes como k6): Permiten ejecutar tareas de forma aislada, como lanzar los pods que generan carga, garantizando que cada experimento tenga un ciclo de vida independiente.

- **Namespaces**: Separan la infraestructura de MiMoQ de otros recursos del cluster, especialmente importante en entornos multiusuario como laboratorios o clusters de investigación.

- **Custom Resource Definitions (CRDs)**: Introducidos por Chaos Mesh, habilitan recursos como `NetworkChaos`, `PodChaos` y `StressChaos`, los cuales se integran directamente al modelo declarativo de Kubernetes.

La estructura del repositorio refleja una adopción completa del enfoque modular de Kubernetes: cada aplicación tiene su directorio `k8s/base` con sus manifiestos fundamentales y un `k8s/overlays` para adaptaciones específicas. Este patrón permite reproducir todo el stack con un solo comando.

Proceso de despliegue

El despliegue de la herramienta se realizó utilizando un enfoque de IaC mediante manifiestos de Kubernetes lo que permitió automatizar la puesta en marcha del sistema de forma consistente, repetible y sin intervención manual significativa. Este enfoque contrasta con los procesos utilizados en etapas previas del proyecto, donde la inicialización del entorno dependía de configuraciones manuales, archivos editados a demanda y pasos secuenciales propensos a

errores humanos.

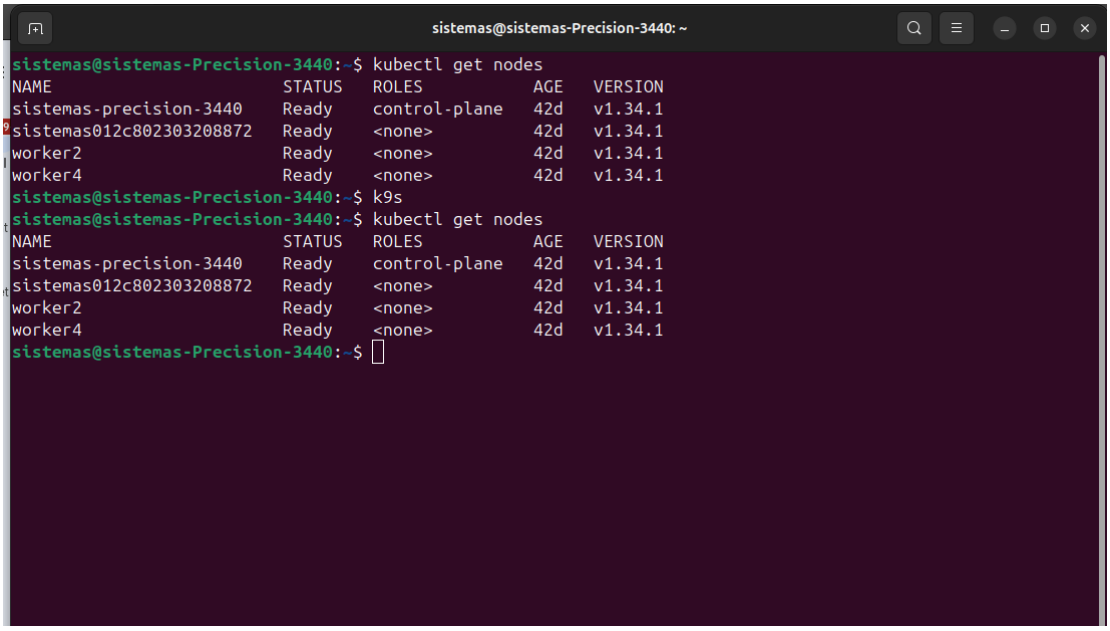

Ilustración 14. Descripción de los nodos del clúster on-premise

El proceso de despliegue en MiMoQ es deliberadamente sencillo, consecuencia directa del enfoque declarativo adoptado. Una vez configurado el clúster —un entorno on-premise construido con kubeadm y compuesto por cuatro nodos de cómputo— basta con ejecutar:

```
kubectl apply -k overlays/prod
```

Este comando aplica todos los manifiestos definidos por Kustomize, creando o actualizando: backend, frontend, base de datos, pods de k6, Prometheus, Chaos Mesh y cualquier otro recurso necesario para la ejecución de pruebas y experimentos.

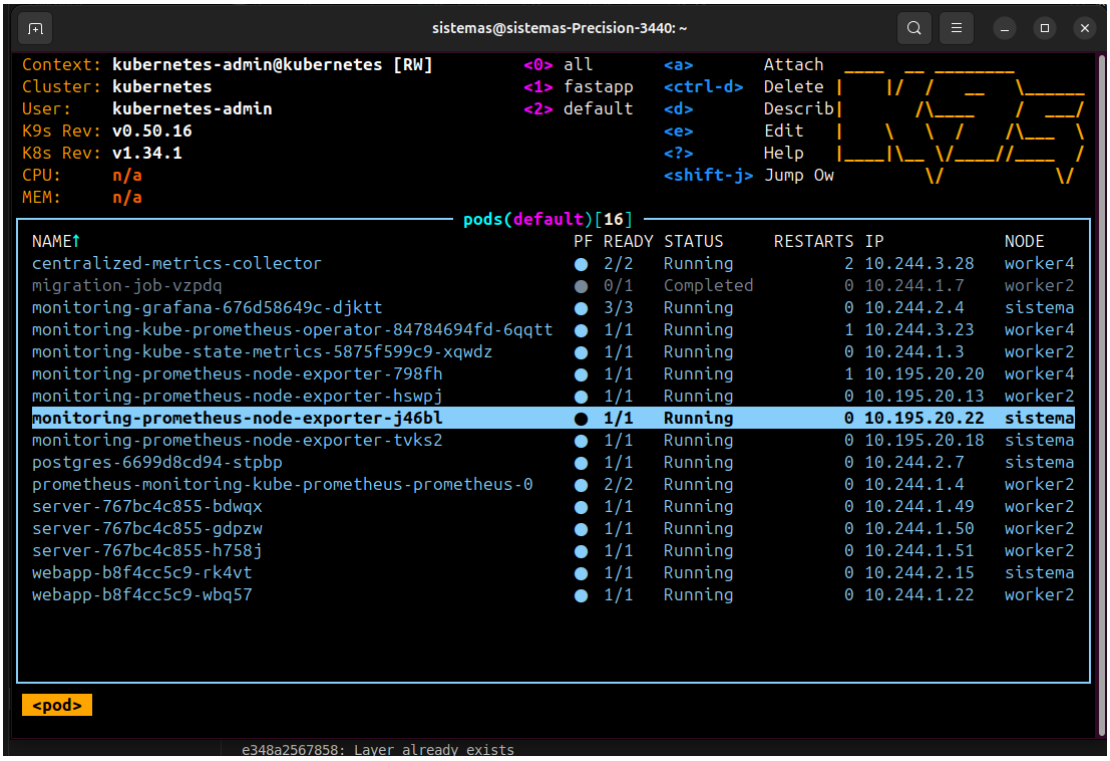

Ilustración 15. Vista de los servicios desplegados en el cluster

La simplicidad del despliegue muestra claramente la ventaja del enfoque IaC: no hay comandos manuales, no se requiere intervención adicional, y cualquier contribuidor puede reproducir el entorno completo a partir del repositorio. Además, dado que el cluster es on-premise, administrado con kubeadm y compuesto por cuatro nodos, IaC permite resolver un problema recurrente en entornos académicos: estandarizar configuraciones a través de hardware heterogéneo con mínima intervención humana.

Además, el modelo de despliegue facilita enormemente la evolución futura del sistema. Si en una versión posterior se actualiza el backend o el frontend, basta con aplicar un rollout controlado mediante:

```
kubectl rollout restart deployment/mimoq-server

kubectl rollout restart deployment/mimoq-frontend
```

o mediante actualizaciones automáticas basadas en cambios de imagen. Kubernetes gestiona la transición asegurando disponibilidad, evitando tiempos muertos y minimizando riesgos. Este enfoque contrasta con despliegues manuales, donde cada actualización requeriría reconfigurar componentes y reiniciar instancias a mano. Con IaC y Kubernetes, la plataforma puede evolucionar de manera sistemática, segura y auditada.

8. RESULTADOS

El presente capítulo expone los resultados obtenidos durante el proceso de refactorización y mejora de la herramienta MiMoQ. Estos resultados guardan relación directa con los objetivos del proyecto, la problemática planteada y el proceso de implementación descrito en los capítulos anteriores.

Validación del despliegue automatizado

La primera serie de resultados corresponde a la incorporación de mecanismos que permiten desplegar MiMoQ en entornos controlados mediante Infraestructura como Código. Se verificó la instalación automatizada del sistema utilizando manifiestos de Kubernetes y configuraciones parametrizables, lo que permitió ejecutar la herramienta tanto en un entorno local basado en Kind como en un clúster remoto. Para ello se creó un script para ejecutar una prueba del despliegue de la aplicación en entornos locales y así validar el despliegue antes de probar en un clúster real (ver Anexo 1)

Pruebas de despliegue en entornos locales

Las pruebas iniciales se realizaron sobre un entorno controlado ejecutado de manera local. Se evaluó la correcta inicialización de los componentes de MiMoQ, incluyendo el backend, frontend, módulo de generación de cargas, motor de monitoreo y base de datos. Los resultados evidenciaron una reducción significativa del tiempo de puesta en marcha y la eliminación de intervenciones manuales necesarias en versiones anteriores. Además, se verificó la consistencia del entorno posterior a múltiples ciclos de despliegue y eliminación de recursos.

Despliegue en clúster remoto y reproducibilidad

Posteriormente, se ejecutó el despliegue en un entorno remoto con mayor capacidad computacional. La herramienta se ejecutó de manera estable, permitiendo la realización de experimentos bajo cargas superiores y escenarios concurrentes. Los resultados confirmaron la reproducibilidad entre entornos, lo cual contribuye a la validez externa de los experimentos y facilita el uso de la plataforma en escenarios académicos y de investigación aplicada.

Integración del módulo de fallas sintéticas

El segundo conjunto de resultados corresponde a la incorporación del módulo de inyección de fallas dentro del sistema de experimentación, lo cual permitió extender las capacidades de MiMoQ hacia la evaluación de resiliencia y recuperación.

Ejecución de experimentos con fallos controlados

Se diseñaron y ejecutaron experimentos que introdujeron fallos controlados en servicios individuales del sistema evaluado. Entre los escenarios evaluados se incluyeron interrupciones de instancias, degradación deliberada del rendimiento y simulación de pérdida temporal de

disponibilidad. Los resultados permitieron observar tiempos de recuperación, impacto sobre la latencia y afectaciones en la propagación de errores entre microservicios.

Métricas obtenidas y análisis comparativo

Para poder probar la herramienta se realizaron experimentos a una aplicación API sencilla llamada fastapp (https://github.com/gfalbarracinr/fastapp) corresponde a una aplicación ligera diseñada para exponer un servicio HTTP con tiempos de respuesta medibles y patrones de comportamiento estables bajo carga. Su estructura simple y su ejecución en un contenedor aislado lo convierten en un candidato adecuado para evaluar atributos de calidad como desempeño, consumo de recursos, impacto de fallas y variabilidad temporal.

A continuación, se presentan los resultados obtenidos a partir de la ejecución de dos experimentos de carga definidos sobre la misma aplicación objetivo. Ambos experimentos se ejecutaron bajo condiciones equivalentes de carga: dos endpoints, 100 usuarios virtuales concurrentes y una duración total de cinco minutos sobre la aplicación desplegada con tres réplicas (pods). La diferencia entre ambos escenarios radica en que, en uno de ellos, se inyectó una falla a nivel de pod con una probabilidad fija del 90%, con el objetivo de evaluar el impacto de este tipo de perturbación sobre el comportamiento del sistema.

Detalle técnico del experimento

Se llevaron a cabo dos experimentos de carga sobre la aplicación FastApp, ambos bajo condiciones controladas y equivalentes, con el objetivo de evaluar el impacto de la inyección de fallas a nivel de pod. La configuración general de los experimentos se describe a continuación.

Tabla 1 Configuración de los experimentos

Parámetro	Experimento sin inyección de fallas	Experimento con inyección de fallas
Aplicación	FastApp	FastApp
Duración	5 minutos	5 minutos
Repeticiones	2	2
Microservicios	1	1
Réplicas	3	3
Endpoint 1	`GET /time`	`GET /time`
Usuarios virtuales (Endpoint 1)	100	100
Endpoint 2	`GET /docs`	`GET /docs`
Usuarios virtuales (Endpoint 2)	100	100

Parámetro	Experimento sin inyección de fallas	Experimento con inyección de fallas
Inyección de fallas	No aplica	Sí
Tipo de falla	—	Falla a nivel de pod
Probabilidad de falla	—	90% (falla fija)

Después de realizar el experimento, las métricas analizadas corresponden a información de uso de recursos recopilada durante la ejecución de los experimentos, específicamente consumo de CPU, uso de memoria y tráfico de red.

- **Uso de CPU**, que representa la cantidad de tiempo de procesamiento utilizada por los pods de la aplicación.
- **Uso de memoria**, expresado en megabytes (MB), que indica la memoria ocupada durante la ejecución.
- **Tráfico de red**, medido en bytes, correspondiente al volumen total de datos transmitidos y recibidos por los pods.

Dado que la aplicación se ejecutó con tres réplicas fijas, las métricas fueron consideradas a nivel agregado, es decir, se sumaron los valores correspondientes a los tres pods para representar el comportamiento global del sistema en cada instante de medición.

Para cada métrica y para cada experimento se calculó el promedio (media aritmética**)** a partir de las muestras recolectadas durante los cinco minutos de ejecución (ver anexo 3 y 4). El procedimiento seguido fue el siguiente:

- Para cada intervalo de tiempo registrado, se sumaron los valores de la métrica correspondiente a los tres pods.
- Esto produjo una serie temporal que representa el consumo total del sistema para cada métrica.
- Sobre esta serie temporal se calculó la media aritmética, definida como la suma de todos los valores dividida por el número total de muestras.

La Tabla 2 presenta los valores medios agregados obtenidos para ambos experimentos.

Tabla 2. Promedio. de métricas agregadas por experimento

Experimento	CPU media total	Memoria media total (MB)	Tráfico de red medio total (bytes)
Sin falla	1.10	448.19	1 867 238
Con falla (90% pod)	0.77	455.70	1 241 358

Estos valores reflejan el consumo promedio del sistema durante toda la ejecución de cada experimento.

A partir de los valores obtenidos se evidencia un comportamiento diferenciado entre el escenario sin falla y el escenario con inyección de falla a nivel de pod.

En el experimento sin falla, el sistema presenta un mayor uso medio de CPU y un volumen de tráfico de red considerablemente superior. Este comportamiento es indicativo de un sistema que recibe y procesa efectivamente las solicitudes generadas por los usuarios virtuales, manteniendo una ejecución sostenida y estable durante toda la prueba.

Por el contrario, cuando se inyecta la falla, se observa una reducción significativa en el uso medio de CPU (Ilustración 17), así como una disminución notable del tráfico de red, como se puede apreciar en la Ilustración 16. Este resultado no debe interpretarse como una mejora en el desempeño del sistema, sino como una consecuencia directa de la pérdida de capacidad para procesar solicitudes. Al verse afectados los pods, parte de las peticiones no llegan a ser procesadas o no se completan correctamente, lo que reduce el volumen de trabajo efectivo realizado por el sistema y, en consecuencia, el consumo total de recursos.

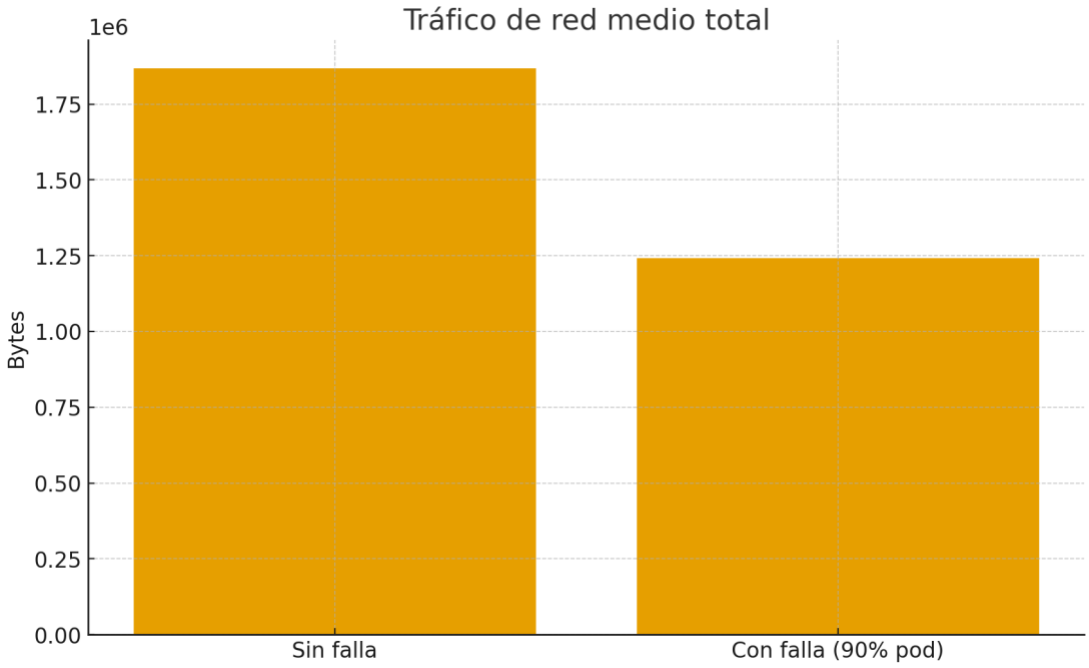

Ilustración 16. Trafico de red

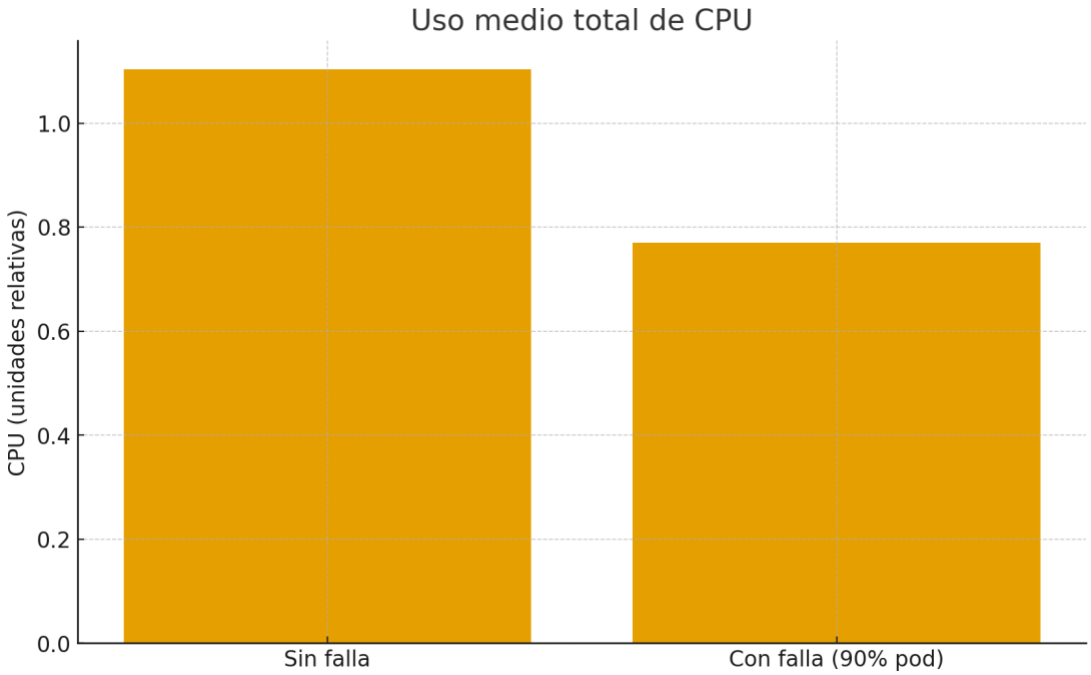

Ilustración 17. Uso medio de CPU

El uso de memoria, en cambio, se mantiene relativamente estable entre ambos escenarios. Esto es coherente con el comportamiento típico de las aplicaciones, donde la memoria asignada no se libera inmediatamente ante fallas transitorias o reinicios parciales, y suele ser menos sensible a corto plazo frente a interrupciones en la carga de trabajo.

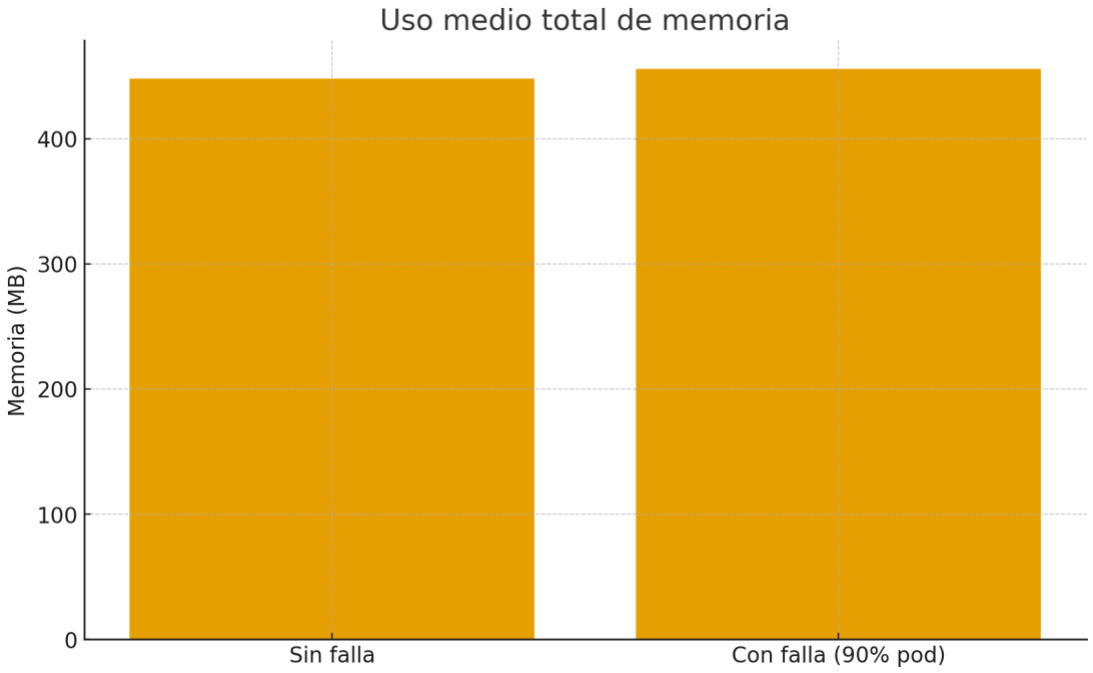

Ilustración 18. Promedio de uso de la memoria

Las gráficas comparativas refuerzan los resultados observados en la tabla. En particular, la disminución del uso de CPU y del tráfico de red en el escenario con falla muestra de manera visual la degradación de la capacidad operativa del sistema. Estas diferencias permiten identificar claramente el impacto de la inyección de fallas, aun cuando las condiciones de carga externas permanecen constantes.

los resultados obtenidos demuestran que la inyección de fallas a nivel de pod genera una afectación observable en el comportamiento global del sistema. La reducción en el uso de CPU y en el tráfico de red evidencia una disminución en la cantidad de trabajo efectivamente procesado, confirmando que el sistema no es capaz de sostener el mismo nivel de servicio bajo condiciones adversas. Estos hallazgos validan la utilidad de MiMoQ para la experimentación controlada y el análisis del impacto de fallas en aplicaciones desplegadas sobre Kubernetes.

9. CONCLUSIONES Y TRABAJOS FUTUROS

El presente capítulo ofrece una síntesis de los hallazgos principales del proyecto y establece líneas de trabajo que pueden fortalecer y ampliar los resultados obtenidos. Con ello, se mantiene la coherencia narrativa con los capítulos anteriores, cerrando el ciclo que inició con la identificación del problema, continuó con el diseño e implementación de la solución y culminó con la evaluación experimental.

La ampliación de MiMoQ permitió superar las limitaciones de su versión inicial, ofreciendo ahora soporte para experimentación en entornos más realistas y con mayor control sobre variables involucradas en el comportamiento de sistemas basados en microservicios. La incorporación de despliegue automatizado mediante IaC incrementó la reproducibilidad, redujo la intervención manual y mejoró la estabilidad del entorno experimental.

El módulo de inyección de fallas sintéticas amplió el alcance de la herramienta hacia la evaluación de atributos como resiliencia y capacidad de recuperación, complementando las métricas previamente centradas en desempeño y disponibilidad.
Los experimentos realizados demostraron que las nuevas funcionalidades permiten caracterizar de manera más precisa el comportamiento de sistemas distribuidos bajo condiciones adversas, contribuyendo así al estudio de arquitecturas de microservicios en contextos académicos y de investigación aplicada.

Así mismo se evidencia una alta dependencia de forma estricta del uso de Docker compose como mecanismo base para definir y ejecutar los microservicios bajo evaluación. Esta dependencia introduce restricciones importantes, dado que la herramienta exige que el repositorio seleccionado para crear un proyecto contenga un archivo denominado exactamente `docker-compose.yml`, sin permitir variantes como `compose.yaml`, `docker-compose.yaml` o estructuras declarativas alternativas. Este requisito limita la adopción por parte de proyectos reales, repositorios académicos y arquitecturas modernas que utilizan Helm charts, Kustomize, operadores o despliegues nativos de Kubernetes.

De manera adicional, se identificó que MiMoQ no puede procesar correctamente experimentos si los servicios definidos en el archivo docker-compose carecen del atributo `container_name`. Esta condición obliga a los usuarios a modificar manualmente sus repositorios para ajustarse a la herramienta

Por último, se determinó que MiMoQ no puede ejecutar proyectos cuyos servicios definan sus contenedores mediante instrucciones `build` basadas en Dockerfiles locales, ya que el sistema requiere que las imágenes estén previamente publicadas en un registry externo, como Docker Hub u otro repositorio de contenedores en la nube. Esta condición implica que el usuario debe contar con:

- imágenes preconstruidas,
- etiquetadas,
- versionadas,

- accesibles públicamente o bajo autenticación,

lo cual representa una barrera tanto para proyectos académicos como para prototipos experimentales que aún no cuentan con pipelines de construcción o distribución de imágenes.

las limitaciones identificadas fueron abordadas satisfactoriamente a lo largo del proyecto. La migración hacia un entorno basado en Kubernetes, la incorporación de manifiestos de Kubernetes como mecanismo de despliegue, la estandarización de variables del entorno, la desacoplación de la capa de visualización y la ampliación del modelo de calidad permitieron superar los obstáculos detectados en la versión previa del sistema. Estas mejoras incrementaron la portabilidad, flexibilidad y capacidad de extensión de MiMoQ, consolidándolo como una herramienta más robusta, reproducible.

Trabajos futuros

A partir de las limitaciones identificadas durante la ampliación y análisis de MiMoQ, se establecen diversas líneas de trabajo futuro orientadas a fortalecer su flexibilidad, portabilidad, capacidad experimental y aplicabilidad en entornos reales de microservicios. Estas líneas buscan desacoplar la herramienta de dependencias rígidas, ampliar su compatibilidad con diferentes modelos de despliegue y mejorar el alcance de los experimentos que pueden realizarse mediante la plataforma.

En primera instancia, se considera prioritario eliminar la dependencia estricta de Docker Compose como única forma válida de definir microservicios, habilitando compatibilidad con manifiestos nativos de Kubernetes, Helm charts, Kustomize y operadores. Este avance permitiría que MiMoQ pueda ser utilizado en arquitecturas modernas y representativas de entornos productivos.

De manera complementaria, se propone desarrollar un mecanismo flexible de descubrimiento de servicios, que no dependa del atributo `container_name` ni de estructuras rígidas del archivo `docker-compose.yml`. Con ello se evitaría que los usuarios deban modificar sus repositorios para poder ejecutar experimentos, mejorando la reproducibilidad y neutralidad metodológica.

Otra línea prioritaria consiste en permitir que MiMoQ ejecute proyectos cuyos servicios se construyen mediante Dockerfiles locales, eliminando la restricción actual que obliga a utilizar imágenes publicadas previamente en Docker Hub u otros registries externos. La integración con pipelines CI/CD, registries privados y flujos automatizados de empaquetamiento permitiría adecuar la herramienta a prácticas reales de desarrollo.

Adicionalmente, se identifica como evolución estratégica la incorporación de soporte para despliegue y operación de MiMoQ en proveedores de nube, incluyendo plataformas administradas de Kubernetes como Amazon EKS, Google GKE, Azure AKS u OpenShift, entre otras. Esta mejora permitiría:

- ejecutar experimentos en infraestructura escalable,

- medir atributos de calidad bajo condiciones reales de carga elástica,
- acceder a telemetría avanzada de nube,
- utilizar registries privados administrados,
- evaluar costos asociados al rendimiento,
- reproducir escenarios propios de entornos industriales.

Con ello, MiMoQ podría trascender el ámbito local y transformarse en una plataforma capaz de realizar experimentación distribuida, multi–región y con condiciones heterogéneas de red, lo cual enriquecería significativamente la validez externa de los resultados.

Finalmente, se plantea desarrollar capacidades orientadas a docencia e investigación aplicada, tales como plantillas de experimentos, laboratorios guiados, datasets comparativos y módulos pedagógicos para cursos de arquitectura, calidad de software y sistemas distribuidos.

Estas líneas de trabajo representan el camino natural para consolidar a MiMoQ como una plataforma robusta, versátil y alineada con la realidad tecnológica actual, habilitando investigación avanzada, transferencia tecnológica y potencial adopción por parte de organizaciones que operan microservicios en infraestructura local, híbrida o en la nube.

10. ANEXOS

Anexo 1. Script para validar despliegue local

A continuación, se presenta el script para validar el despliegue de manera local

```
#!/bin/bash

echo "🧪 Probando despliegue de producción localmente con Kind..."

if ! command -v kind &> /dev/null; then

    echo "❌ Kind no está instalado. Instalando..."

    curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-
amd64

    chmod +x ./kind

    sudo mv ./kind /usr/local/bin/kind

fi

echo "🚀 Creando cluster Kind para testing..."

kind create cluster --name mimoq-prod-local --config kind.yaml

kind get kubeconfig --name mimoq-prod-local > ~/.kube/config-kind-
mimoq-prod-local

export KUBECONFIG=~/.kube/config-kind-mimoq-prod-local

kubectl cluster-info

echo "📦 Instalando K6 operator..."

make setup-k6-operator

echo "📊 Instalando monitoring stack..."

helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts

helm repo update
```

```
helm install prometheus-operator prometheus-community/kube-
prometheus-stack \

  --namespace monitoring \

  --create-namespace \

  --set
prometheus.prometheusSpec.serviceMonitorSelectorNilUsesHelmValues=fa
lse \

  --set
prometheus.prometheusSpec.podMonitorSelectorNilUsesHelmValues=false
\

  --set
prometheus.prometheusSpec.ruleSelectorNilUsesHelmValues=false

kubectl wait --for=condition=ready pod -l
app.kubernetes.io/name=prometheus-operator --timeout=300s -n
monitoring

if [ -f "apps/server/k8s/kustomize/base/values-monitoring.yml" ];
then

    echo "📊 Aplicando configuraciones específicas de monitoring..."

    helm upgrade monitoring prometheus-community/kube-prometheus-
stack \

      --namespace monitoring \

      -f apps/server/k8s/kustomize/base/values-monitoring.yml

else

    echo "ℹ️ No se encontraron configuraciones específicas de
monitoring"

fi

echo "📦 Desplegando configuración de producción..."

kubectl apply -k apps/server/k8s/kustomize/overlays/production/

kubectl apply -k apps/webapp/k8s/kustomize/overlays/production/
```

```
echo "⏳ Esperando a que los pods estén listos..."

kubectl wait --for=condition=ready pod -l app=server --timeout=300s

kubectl wait --for=condition=ready pod -l app=webapp --timeout=300s

echo "📊 Estado de los recursos:"

kubectl get all

echo "🌐 Configurando port-forward para testing..."

echo "   - Server: http://localhost:3000"

echo "   - Webapp: http://localhost:4200"

echo ""

echo "💡 Para probar:"

echo "   - Abre http://localhost:4200 en tu navegador"

echo "   - Verifica que la API responda en
http://localhost:3000"

echo ""

echo "🛑 Para limpiar: kind delete cluster --name mimoq-prod-
local && rm ~/.kube/config-kind-mimoq-prod-local"

kubectl port-forward service/server 3000:3000 &

kubectl port-forward service/webapp 4200:80 &

echo "✅ Cluster de testing listo! Presiona Ctrl+C para
detener port-forward y limpiar."
```

Anexo 2. Métricas generadas por K6 para los experimentos

- **k6_http_requests_total**

Métrica acumulativa: representa el número total de solicitudes HTTP emitidas por el script de K6 durante la ejecución.

PromQL: `k6_http_requests_total{experiment="X"}`

Permite determinar la carga generada sobre los endpoints probados; útil para verificar que la tasa de peticiones coincida con la configuración del experimento.

CSV: incluye `timestamp` y `value` (cantidad acumulada en ese instante).

- **k6_http_request_duration_seconds**

Distribución de la duración de las solicitudes HTTP en segundos; puede representar buckets si está instrumentado como histograma.

PromQL: `k6_http_request_duration_seconds{experiment="X"}`

Describe la latencia de las peticiones, siendo esencial para el análisis del rendimiento.

CSV: percentiles (si se calculan) o valores por bucket.

- **k6_http_request_duration_seconds_p90, p95, p99**

Percentiles 90, 95 y 99 obtenidos a partir del histograma de latencias de K6.

PromQL (ejemplo para p95): `k6_http_request_duration_seconds_p95{experiment="X"}`

Permiten evaluar cuellos de latencia, especialmente en escenarios de alta concurrencia donde algunas peticiones pueden degradarse significativamente.

- **k6_http_request_failures_total**

Número acumulado de fallos de solicitudes HTTP durante el experimento.

PromQL: `k6_http_request_failures_total{experiment="X"}`

Tipo: *Counter*.

un incremento constante es indicativo de errores en los endpoints, problemas de infraestructura o fallas en la lógica del test.

- **k6_virtual_users**

Cantidad de usuarios virtuales activos en cada momento del experimento.

PromQL: `k6_virtual_users{experiment="X"}`

Tipo: *Gauge*.

correlaciona el nivel de concurrencia con el comportamiento del sistema y su degradación bajo carga.

- **k6_data_sent_bytes_total / k6_data_received_bytes_total**

Volumen total de datos enviados y recibidos por los clientes K6 durante la prueba.

PromQL:

```
k6_data_sent_bytes_total{experiment="X"}

k6_data_received_bytes_total{experiment="X"}
```

Tipo: *Counter*.

permite analizar consumo de ancho de banda, tráfico de red y patrones de datos (por ejemplo, cuándo se consumen más datos).

- **k6_iterations_total**

Número total de iteraciones completas del script de K6.

PromQL: `k6_iterations_total{experiment="X"}`

Tipo: *Counter*.

cuantifica el "trabajo real" ejecutado por el script más allá de solo peticiones HTTP, especialmente relevante si cada iteración realiza múltiples acciones o API calls.

- **k6_experiment_requests_total**

Métrica personalizada que representa todas las solicitudes asociadas al experimento específico.

PromQL: `k6_experiment_requests_total{experiment="X"}`

sirve como verificación interna para validar que el número de peticiones esperadas en el experimento coincida con la cantidad realmente ejecutada.

- **k6_experiment_duration_seconds**

Duración total de la ejecución del experimento en segundos.

PromQL: `k6_experiment_duration_seconds{experiment="X"}`

Tipo: *Gauge*.

esencial para calcular el throughput real del experimento, usando la fórmula:

requests_per_second `= total_requests / duration`

1.17.1.1 Métricas de CPU de los Pods

- **container_cpu_usage_seconds_total**

Semántica: contador acumulativo que mide el tiempo de CPU utilizado por cada contenedor.

PromQL:

```
rate(
  container_cpu_usage_seconds_total{
    pod="P",
    namespace="N",
    container!="POD"
  }[5m]
)
```

la función `rate()` calcula el cambio por segundo, lo que permite estimar la cantidad de CPU (en "cores equivalentes") que consume un pod en promedio.

CSV: incluye el `timestamp` y el valor de CPU utilizado (por ejemplo, 0.15= 15% de un núcleo).

Métricas de Memoria de los Pods

- **container_memory_working_set_bytes**

Semántica: cantidad de memoria (bytes) realmente usada y "trabajando" por el contenedor, excluyendo caché descartable.

PromQL:

```
container_memory_working_set_bytes{
  pod="P",
  namespace="N",
  container!="POD"
}
```

esta métrica es útil para detectar fugas de memoria, saturación o patrones de uso anómalos bajo carga intensa.

CSV: los valores se reportan en bytes, aunque en análisis posteriores se suelen convertir a MiB o GiB.

Métricas de Red de los Pods

- **Tasa de recepción de bytes**

PromQL:

```
sum(
  rate(
    container_network_receive_bytes_total{
      pod="P", namespace="N"
    }[5m]
  )
) by (pod)
```

representa los bytes por segundo que un pod recibe por red, lo cual permite detectar saturaciones o picos de ingreso de datos.

- **Tasa de transmisión de bytes**

PromQL:

```
sum(
  rate(
    container_network_transmit_bytes_total{
      pod="P", namespace="N"
    }[5m]
  )
) by (pod)
```

mide los bytes por segundo transmitidos por el pod, útil para analizar la salida de datos, cuellos de red o comunicación entre servicios.

Métricas HTTP de Servicios Internos

- Explora diferentes posibles métricas según el exportador disponible:

```
http_requests_total

nginx_http_requests_total

istio_requests_total

envoy_http_downstream_rq_total
```

Para cada servicio (`serviceName`), MiMoQ construye y prueba varias consultas PromQL hasta encontrar resultados válidos:

```
metricName{service="S", namespace="N"}

metricName{pod=~".*S.*", namespace="N"}

metricName{job="S"}
```

Estas métricas proporcionan visibilidad del tráfico HTTP interno en la infraestructura, no solo el generado por los tests de K6.

1.17.1.2 Métricas de Latencia HTTP en Servicios Internos

Se consultan posibles métricas de latencia:

```
http_request_duration_seconds

nginx_http_request_duration_seconds

istio_request_duration_milliseconds

envoy_http_downstream_rq_time
```

Ejemplo de PromQL:

```
avg(envoy_http_downstream_rq_time{job="S"})
```

Esta métrica permite identificar degradaciones internas en la latencia, lo que es especialmente relevante dado que un servicio puede responder rápido externamente (K6) pero sufrir lentitud en su comunicación interna o con otros componentes.

11. REFERENCIAS

Bass, C. P. (2012). *Software Architecture in Practice.*

Newman, S. (2015). *Building Microservices: Designing Fine-Grained System.* O'Reilly Media.

Camilli, M. (2022). Modeling performance of microservices systems with growth theor. *Empirical Software Engineering*.

Alvarado, A. J. (2024). Sistema de experimentación para evaluar atributos de calidad en microservicios: MiMoQ. *Pontificia Universidad Javeriana, Facultad de Ingeniería, Programa de Ingeniería de Sistemas*.

Pahl, C. (2018). Performance and Reliability Testing of Microservices: A Survey. *Journal of Systems and Software*.

Chen, L. A. (2016). Cloud-based testing: Challenges and opportunities. *IEEE Software*.

Sheldon Smith, E. R. (2023). Benchmarks for End-to-End Microservices Testing. *IEEE SOSE*.

Dragoni, N. G. (2017). Microservices: Yesterday, Today, and Tomorrow. *Present and Ulterior Software Engineering*.

Fowler, M. &. (2014). *Microservices: a definition of this new architectural term*. Retrieved from Martin Fowler: https://martinfowler.com/articles/microservices.html

Villamizar, M. G. (2016). Evaluating the monolithic and the microservice architecture pattern using Docker containers. *Latin American Computing Conference*.

New Relic. (2023). *New Relic Observability Documentation*. Retrieved from New Relic: https://docs.newrelic.com/

Elastic. (2022). *Elastic Stack Documentation*. Retrieved from Elastic: https://www.elastic.co/guide

Apache Software Foundation. (2023). *Apache JMeter Documentation*. Retrieved from JMeter: https://jmeter.apache.org/

Heyman, J. (2021). *Locust Documentation*. Retrieved from locust: https://docs.locust.io/en/stable/

Gatling Corp. (2022). *Gatling Load Testing Tool Documentation*. Retrieved from Gatling: https://docs.gatling.io/

Hargreaves, P. (2021). *Artillery Documentation*. Retrieved from Artillery: https://www.artillery.io/docs

Rodríguez, M. P. (2019). *WiseToolKit: Framework para automatización de pruebas*. Retrieved from Digital Library: https://dl.acm.org/doi/10.1007/978-3-030-59635-4_11

Tilt. (2025). *Tilt documentation*. Retrieved from Tilt: https://docs.tilt.dev/

ISO/IEC. (2011). ISO/IEC 25010:2011 — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE).

Hightower, K. B. (2017). *Kubernetes: Up and Running – Dive into the Future of Infrastructure.* O'Reilly Media.

Burns, B. G. (2016). Borg, Omega, and Kubernetes. *Communications of the ACM*.

Dobies, C. &. (2021). *Kubernetes Operators: Automating the Container Orchestration Platform.* O'Reilly Media.

Alvarado, A. J. (2017, Mayo). *Benchmark requirements for microservices architecture research*. Retrieved from IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering: https://doi.org/10.1109/ECASE.2017.1

Alvarado, A. J. (2024). MiMoQ: A System for Experimentation of Microservice-Based Applications. *Colombian Conference on Computing.*

Foster, G. (2025, Junio). *The benefits of a monorepo*. Retrieved from graphite: https://graphite.com/guides/benefits-of-a-monorepo

Hochstein, L. (2017). *Chaos Monkey*. Retrieved from Github Netflix Simian Army: https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey

Gremlin. (2025). *About Gremlin*. Retrieved from Gremlin: https://www.gremlin.com/about

PingCAP. (2025). *Chaos Mesh*. Retrieved from Chaos Mesh: https://chaos-mesh.org/

kubernetes. (2025). *Declarative Management of Kubernetes Objects Using Kustomize*. Retrieved from kubernetes: https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/

Morris, K. (2020). *Infrastructure as Code.* O'Reilly.

Bavithran. (2025, Abril 8). *Kustomize — Managing Kubernetes Configurations*. Retrieved from Medium: https://medium.com/@bavicnative/kustomize-managing-kubernetes-configurations-4f331ba28e61

Shvets, A. (2021). *Dive into Design Patterns.*