UNIVERSIDAD DON BOSCO

DIRECCIÓN DE EDUCACIÓN A DISTANCIA

TRABAJO DE GRADUACIÓN:

DISEÑO Y EVALUACIÓN DE ARQUITECTURA RESILIENTE BASADA EN CHAOS ENGINEERING EN ENTORNOS KUBERNETES

PARA OPTAR AL GRADO DE:

MAESTRO EN ARQUITECTURA DE SOFTWARE

Autores:

Eduardo Ernesto Flamenco Elias

Carlos Eduardo Palomo Sosa

Jaime Remberto Hernández Gamero

Asesora:

Mg. Karens Lorena Medrano

Antiguo Cuscatlán, La Libertad El Salvador C. A.

1 DICIEMBRE, 2025

Rector Universidad Don Bosco

Dr. Mario Rafael Olmos

Secretaria General

Inga. Yesenia Xiomara Martínez Oviedo

Director de Educación a Distancia

Dr. Eduardo Menjívar Valencia

Director de la Maestría

Dr. Vicent Ramon Palasí Lallana

Asesora del proyecto de graduación

Mg. Karens Lorena Medrano

Lectora del proyecto de graduación

Mg. Evelyn Lissette Hernández

CONTENIDO

Resumen

Esta investigación evalúa empíricamente el impacto de la Ingeniería del Caos en la resiliencia de una arquitectura distribuida cloud-native desplegada sobre Kubernetes, tomando como caso de estudio un sistema de transferencias bancarias. El objetivo principal es comparar el comportamiento de dos versiones de una misma aplicación: una arquitectura sin patrones explícitos de resiliencia y otra refactorizada que incorpora circuit breaker, retry, bulkhead y replicación en el bróker de mensajería, bajo cargas de trabajo equivalentes y fallos controlados.

Metodológicamente, se diseñó y desarrolló una plataforma de transferencias basada en microservicios ejecutados en contenedores Docker orquestados por Kubernetes. Se llevaron a cabo experimentos de caos en dos fases: en la primera, se inyectaron fallos sobre la versión base (eliminación de servicios críticos, sobrecarga de flujos y caída del bróker de mensajería); en la segunda, se repitieron los mismos escenarios tras incorporar los patrones de resiliencia. El desempeño se evaluó mediante métricas de tasa de errores, latencia, disponibilidad, Tiempo Medio para Detectar (MTTD) y Tiempo Medio para Recuperar (MTTR), registradas con herramientas de observabilidad y monitoreo.

Los resultados muestran que la incorporación de patrones de resiliencia reduce de forma significativa los errores no controlados, acorta los tiempos de detección y recuperación y mantiene la continuidad de los flujos críticos ante fallos inducidos. En particular, circuit breaker y retry mitigan la indisponibilidad temporal del API bancario, bulkhead evita que la sobrecarga de un flujo afecte a otros procesos y la replicación del bróker elimina la pérdida de mensajes. El estudio aporta evidencia cuantitativa replicable sobre el efecto concreto de estos patrones en entornos Kubernetes y ofrece un referente metodológico y conceptual en español para futuras investigaciones y aplicaciones industriales en dominios de alta criticidad como los servicios financieros.

1. Introducción

1.1 Definición del tema de la investigación

Diseño y evaluación de arquitectura resiliente basada en Chaos Engineering en entornos Kubernetes.

La presente investigación se enfoca en la construcción y análisis de una arquitectura orientada a la resiliencia que asegure la continuidad operativa y la confiabilidad de los servicios desplegados en un entorno Kubernetes. Para ello, se aplicarán principios y patrones de Chaos Engineering, sometiendo el sistema a fallas controladas con el propósito de observar su comportamiento, identificar puntos de vulnerabilidad y proponer mejoras que optimicen no solo el desempeño, sino que también incrementen la tolerancia a fallos.

1.2 Formulación del problema

La creciente dependencia de aplicaciones críticas desplegadas como microservicios en entornos dinámicos y propensos a fallos, como Kubernetes, ha convertido la resiliencia, entendida como la capacidad de un sistema para mantener su servicio ante perturbaciones, en un atributo de calidad primordial. Sin embargo, existe una brecha entre la adopción teórica de patrones de resiliencia (como circuit breaker, reintentos o bulkhead) y la evidencia empírica y cuantitativa sobre su efectividad real bajo condiciones de fallo específicas.

En este contexto, el problema central de esta investigación se formula de la siguiente manera:

¿En qué medida arquitectura de microservicios que incorpora de forma sistemática patrones de resiliencia y se valida mediante pruebas de caos (Chaos Engineering) puede reducir el tiempo de recuperación y mantener una mayor disponibilidad de servicio frente a fallos comunes (en redes, nodos o bases de datos), en comparación con una arquitectura de referencia que no implementa dichos mecanismos?

1.3 Justificación

En la actualidad Kubernetes es el estándar para aplicaciones con arquitecturas distribuidas. En esta clase de entornos la resiliencia es un aspecto critico para asegurar la operabilidad del sistema. La integración de pruebas de caos como componente nativo de arquitectura es un tema poco investigado a pesar de la alta aplicabilidad del mismo en sistemas reales.

Actualmente, Kubernetes se ha consolidado como el estándar de facto para la orquestación de aplicaciones con arquitecturas distribuidas y de microservicios, debido a su capacidad

de escalar dinámicamente y optimizar recursos en entornos productivos (Burns et al., 2016). Sin embargo, este tipo de sistemas complejos la resiliencia constituye un aspecto de alta relevancia para lograr la continuidad de los servicios en casos de fallos inesperdos.

La literatura señala que el Chaos Engineering incorporado de manera nativa en el diseño de sistemas aún se encuentra poco explorada en la literatura (Basiri et al., 2016), de manera que la investigación deviene en relevante en tanto que se podrá evaluar, bajo condiciones de caos controladas, el comportamiento de una arquitectura de Kubernetes, señalando vulnerabilidades y proponiendo estrategias que fortalezcan la tolerancia a fallos.

1.4 Supuestos

La presente investigación parte del supuesto de que los escenarios de fallo inducidos mediante Ingeniería del Caos representan de manera adecuada las condiciones reales que pueden afectar a un sistema de transferencias bancarias desplegado en Kubernetes, incluyendo interrupciones de servicios críticos, latencia, pérdida de conectividad y fallos en el bróker de mensajería. Se asume que tanto la arquitectura base como la arquitectura refactorizada que incorpora patrones de resiliencia como circuit breaker, retry, bulkhead y replicación en Kafka están correctamente implementadas conforme a las buenas prácticas cloud-native, garantizando que cualquier diferencia observada en las métricas de desempeño y disponibilidad responde exclusivamente a la introducción de dichos patrones. Asimismo, se considera que las herramientas de inyección de fallos (Chaos Mesh, LitmusChaos, Toxiproxy, entre otras) operan con precisión y permiten la reproducción controlada de los experimentos bajo cargas equivalentes. Se presupone que la carga de trabajo simulada es representativa del funcionamiento operativo del sistema bancario y que las herramientas de observabilidad utilizadas permiten medir de forma fiable tasas de error, latencia, disponibilidad, MTTD y MTTR. Finalmente, se asume que el entorno experimental se mantiene estable y sin interferencias externas que alteren la validez de los resultados, y que el equipo investigador posee las competencias técnicas necesarias para el análisis de los datos generados durante las pruebas.

1.5 Alcances

El alcance de esta investigación comprende el diseño, desarrollo y evaluación experimental de una arquitectura distribuida cloud-native para un sistema de transferencias bancarias desplegado en Kubernetes, comparando una versión sin patrones explícitos de resiliencia con otra que incorpora circuit breaker, retry, bulkhead y replicación del bróker de mensajería. El estudio incluye la implementación de microservicios en contenedores Docker, su orquestación sobre Kubernetes y la construcción de un entorno controlado que permita ejecutar escenarios de fallo mediante técnicas formales de Ingeniería del Caos. Asimismo, se realizan experimentos iterativos que evalúan métricas cuantitativas de confiabilidad y resiliencia entre ellas tasa de errores, latencia, disponibilidad, MTTD y MTTR bajo cargas de trabajo equivalentes. El análisis se limita a entornos experimentales y no contempla la puesta en producción ni la evaluación de costos operativos. Además, el trabajo se circunscribe a fallos inducidos de naturaleza técnica (interrupción de servicios,

sobrecarga, caída del bróker, latencia artificial), sin abordar amenazas de seguridad, cumplimiento normativo o fraudes bancarios. Finalmente, el estudio se enfoca en la comparación de arquitecturas dentro del dominio financiero, aportando evidencia cuantitativa replicable y un marco metodológico que pueda servir como referencia para investigaciones futuras y aplicaciones industriales en sistemas críticos basados en Kubernetes.

1.6 Objetivos

1.6.1. Objetivo General

Comparar el desempeño resiliente de una arquitectura de microservicios en Kubernetes con patrones de resiliencia y Chaos Engineering frente a una arquitectura base, midiendo disponibilidad, MTTR, latencia y tasa de errores para validar cuantitativamente la mejora.

1.6.2. Objetivos específicos

- Diseñar e implementar en Kubernetes dos arquitecturas de referencia para el sistema de transferencias bancarias: una sin patrones explícitos de resiliencia y otra que incorpore circuit breaker, reintentos, bulkhead y mecanismos de alta disponibilidad en el bróker de mensajería.
- Definir y documentar un protocolo experimental basado en Chaos Engineering, especificando escenarios de fallo (en red, servicios, nodos y bases de datos), niveles de carga y condiciones de ejecución para ambas arquitecturas.
- Configurar y validar un sistema de observabilidad y medición que permita registrar métricas de resiliencia tales como disponibilidad, tasa de errores, tiempo medio de detección (MTTD) y tiempo medio de recuperación (MTTR).
- Ejecutar experimentos comparativos de caos sobre las dos arquitecturas bajo condiciones equivalentes de carga y entorno, recopilando los datos necesarios para su análisis cuantitativo.
- Analizar las diferencias en el desempeño y la resiliencia de ambas arquitecturas a partir de las métricas obtenidas, generando recomendaciones técnicas y metodológicas orientadas a fortalecer la resiliencia de arquitecturas distribuidas basadas en Kubernetes.

2. Marco teórico

2.1. Fundamentos de Chaos Engineering.

La Ingeniería del Caos (Chaos Engineering, CE) nació en la práctica con el objetivo de elevar la resiliencia de sistemas distribuidos mediante fallos controlados antes de que

ocurran incidentes reales. Su punto de partida más reconocido es Netflix y su Simian Army -con Chaos Monkey como herramienta emblemática para "matar" instancias aleatoriamente y exponer fragilidades-; desde esa experiencia se formularon principios que conciben la CE como experimentación para asegurar la disponibilidad del sistema (Basiri et al., 2016).

A partir de esa experiencia fundacional, la CE se expandió en dos direcciones que se retroalimentan. Por un lado, la formalización conceptual, que codifica un ciclo científico con hipótesis sobre el estado estable (steady state), inyección deliberada de perturbaciones y observación de efectos para aprender y mejorar. Por otro, la industrialización, que integra CE en DevOps y SRE mediante automatización, control del radio de impacto y prácticas organizacionales para ejecutar experimentos de manera segura y continua (Basiri et al., 2016; Akgül & Güvez, 2024).

En la literatura gris sistematizada entre 2019 y 2024 se observa cómo la industria adopta y adapta los principios fundacionales de Netflix, enfatizando automatización, contención del impacto y alineación con flujos de entrega continua (Fossati, Tamburri, Di Penta, & Tonnarelli, 2025). La CE deja así de ser una práctica aislada para convertirse en un componente integrado del ciclo de vida del software.

El salto a entornos cloud-native -contenedores, Kubernetes, service meshes- elevó la complejidad estructural y operacional: dependencias implícitas, fallos parciales, timeouts encadenados y estados distribuidos. Esta complejidad revalorizó la CE como práctica para revelar modos de fallo difíciles de observar con pruebas tradicionales y volvió clave la observabilidad (trazas distribuidas, métricas y logs) como sustrato para definir el steady state y detectar regresiones de resiliencia. Las fuentes industriales subrayan que la complejidad de sistemas modernos y los altos costos de indisponibilidad empujaron la adopción: en 2024, múltiples organizaciones reportaron costos de inactividad que justifican prácticas proactivas como CE (citado en Fossati et al., 2025).

La evidencia empírica de adopción en código abierto confirma esta trayectoria. Un estudio de minería de repositorios que partió de 5 845 proyectos (971 válidos tras depurar falsos positivos) muestra que herramientas como Toxiproxy y Chaos Mesh concentran la mayor adopción sostenida desde 2016, mientras Chaos Monkey mantiene relevancia como herramienta agnóstica de plataforma. Se observa un pico de lanzamientos de herramientas en 2018, correlacionado con el auge del ecosistema cloud-native; desde 2019 el ritmo desciende, lo que sugiere una fase de maduración enfocada en integración y refinamiento más que en proliferación (Owotogbe, Kumara, Di Nucci, Tamburri, & van den Heuvel, 2025).

En cuanto a familias de fallos practicadas, predominan la terminación de instancias, seguida de disrupciones de red, estrés de recursos y fallos a nivel de aplicación. Las herramientas Kubernetes-nativas (por ejemplo, Chaos Mesh, LitmusChaos) facilitan un repertorio amplio, mientras utilidades como Toxiproxy se especializan en fallas de red

(Owotogbe et al., 2025). Esta distribución confirma la alineación entre la tipología de fallos inyectados y la arquitectura cloud-native basada en contenerización, orquestación y malla de servicios.

La práctica industrial moderna ha desplazado el énfasis desde “romper en producción” hacia flujos seguros, automatizados y sociales: contener el radio de impacto, definir guardrails y estrategias de rollback, automatizar ejecuciones recurrentes en CI/CD y cerrar el ciclo con aprendizaje socializado (game days, postmortems sin culpa). Un marco industrial sintetizado en diez conceptos (C1–C10) -definir estado estable, formular y validar hipótesis, variar eventos reales, ejecutar, automatizar, contener impacto, incrementar complejidad, medir–aprender–mejorar y socializar- cristaliza este giro y sitúa la automatización (C5), la contención (C6), el aumento controlado (C8) y la socialización (C9) en el centro de la práctica (Fossati et al., 2025).

En términos de adopción “en la naturaleza”, los datos de GitHub revelan además que los proyectos de desarrollo dominan frente a teaching/learning/research, lo que refuerza la lectura de la CE como práctica orientada a sistemas reales y a pipelines de entrega (Owotogbe et al., 2025).

Una limitante histórica de CE es su costo operativo: ejecutar campañas amplias y frecuentes puede ser oneroso y riesgoso. En respuesta, surge una vía complementaria que razona desde modelos para hacer triage y focalización de experimentos. La propuesta de descubrimiento de modelos y simulación de grafos plantea que un modelo topológico mínimo -grafo de dependencias bloqueantes más réplicas- puede estimar disponibilidad de endpoints bajo fallos fail-stop con alta correlación respecto a resultados en vivo, sobre todo cuando hay replicación (Krasnovsky, 2025).

La técnica automatiza la síntesis del modelo a partir de artefactos ya disponibles (trazas OpenTelemetry/Jaeger, telemetría de malla, manifiestos Kubernetes, IaC). Posteriormente ejecuta Monte Carlo para muestrear fallos por réplica y evalúa alcanzabilidad en subgrafos vivos, devolviendo una señal de postura de resiliencia útil para CI/CD; los experimentos “vivos” quedan así reservados para validaciones críticas o casos ambiguos.

La intuición clave es que, bajo llamadas síncronas y fallos fail-stop, la disponibilidad depende sobre todo de alcanzabilidad y replicación. Por ello, el grafo más réplicas captura gran parte de la señal de resiliencia sin necesidad de modelos analíticos complejos. Esta “vía ligera” no pretende sustituir la CE experimental, sino potenciarla mediante triage continuo por simulación y validación selectiva por inyección real.

Si se observa longitudinalmente el ecosistema, aparece una primera ola con Chaos Monkey (2010) y un despegue desde 2016 asociado a Kubernetes. El pico de lanzamientos en 2018 y la posterior desaceleración encajan con una fase de madurez enfocada en integración, mantenimiento y buenas prácticas. Lo que sí crece es el uso sostenido de un núcleo de herramientas -Toxiproxy, Chaos Mesh, LitmusChaos- y utilidades de terminación de

pods/instancias, así como la incorporación de CE a flujos de observabilidad y SLO-as-Code (Owotogbe et al., 2025; Fossati et al., 2025).

La frontera más reciente de la disciplina se dirige a ecosistemas inteligentes, en particular sistemas multiagente basados en LLM (LLM-MAS). Estos sistemas exhiben modos de fallo atípicos respecto al software tradicional: alucinaciones, fallas de coordinación entre agentes, comportamientos emergentes y cascadas por recursos compartidos o protocolos conversacionales. Adaptar CE a LLM-MAS implica un marco de experimentación con módulos de caos, monitorización y adaptación: inyectar fallas de comunicación, simular latencias, reasignar tareas y observar la robustez en condiciones production-like. Además, se plantea su uso con fines de auditoría/certificación mediante estudios de acción con socios industriales (por ejemplo, Deloitte) (Owotogbe, 2025).

Esta línea conecta con la CE "clásica": reutiliza la lógica de hipótesis–experimento–observación, apalanca observabilidad para medir degradaciones de calidad (precisión, task success rate, latency to goal) y prioriza contención y rollback por el carácter sensible de entornos con IA. A mediano plazo, se anticipa una taxonomía de fallos para LLM-MAS, extensiones de herramientas de CE para inyectar fallos semánticos y protocolos de evaluación que combinen métricas cuantitativas (tasa de detección/recuperación) con indicadores cualitativos (impacto de negocio, experiencia de usuario) (Owotogbe, 2025; Owotogbe et al., 2025).

En síntesis, la CE ha pasado de ser un acto disruptivo puntual a un sistema operativo de aprendizaje de resiliencia para arquitecturas modernas: mide, experimenta, aprende, automatiza y socializa. Su evolución puede leerse como una secuencia: génesis operativa (Netflix), formalización científica (método de hipótesis), industrialización DevOps/SRE (automatización, contención, socialización), consolidación cloud-native (microservicios/Kubernetes), complemento model-based (descubrimiento de modelos y simulación para triage) y nueva frontera IA (LLM-MAS con horizontes de auditoría y certificación).

Sobre esta base conceptual, la siguiente sección profundiza en cómo estos principios se materializan específicamente en entornos cloud y Kubernetes, que constituyen el contexto tecnológico principal de la investigación.

2.2. Aplicación en entornos cloud y Kubernetes

En sistemas distribuidos y, en particular, en arquitecturas cloud-native sobre Kubernetes, la Ingeniería del Caos se aplica hoy como un ciclo operativo: definir el estado estable, formular hipótesis, inyectar perturbaciones controladas, observar el efecto en métricas de usuario/servicio y aprender para mejorar la confiabilidad e incorporar automatización en CI/CD (Fossati et al., 2025). Este ciclo traslada los principios generales descritos en la sección 6.1 a un entorno tecnológico concreto.

Este paso de “romper para ver” a experimentación disciplinada aparece en marcos industriales recientes y en revisiones multivocales, que documentan cómo las organizaciones combinan CE con observabilidad, guardrails (contención del radio de impacto y rollbacks) y prácticas de aprendizaje organizacional (game days, postmortems sin culpa) (Fossati et al., 2025). La CE deja así de ser un ejercicio excepcional para transformarse en una rutina controlada dentro de la operación diaria.

En Kubernetes, las aplicaciones más frecuentes se centran en terminación de pods/instancias, perturbaciones de red (latencia, pérdida, particiones), estrés de recursos (CPU, memoria, I/O) y fallos a nivel de aplicación (por ejemplo, crash de procesos o excepciones no controladas). Esta práctica refleja tanto los patrones de falla típicos de microservicios como el portafolio funcional de las herramientas más usadas (Owotogbe et al., 2025).

De manera consistente con esa tipología, Chaos Mesh cubre terminación, red, recursos y fallos de aplicación; Toxiproxy se especializa en red; y utilidades como Chaoskube o Kube-Monkey focalizan la terminación aleatoria de pods/instancias (Owotogbe et al., 2025). El ecosistema de herramientas permite así diseñar campañas de caos que representan fielmente los modos de fallo más relevantes para arquitecturas cloud-native.

Las tendencias de adopción en la comunidad confirman la madurez de CE en Kubernetes. El análisis de 971 repositorios muestra a Toxiproxy y Chaos Mesh entre las herramientas con uso sostenido; se observa un pico de lanzamientos en 2018 (≈20 herramientas nuevas) asociado al auge cloud-native, seguido por una desaceleración desde 2019 que sugiere transición desde la exploración a la integración práctica en pipelines y plataformas (Owotogbe et al., 2025).

Este giro del ecosistema -menos “herramientas nuevas”, más automatización, integración con observabilidad y buenas prácticas- es coherente con el énfasis industrial en seguridad operativa (control de blast radius, abort switches) y aprendizaje continuo (Fossati et al., 2025). En otras palabras, la CE se consolida como parte del “tejido” operativo de Kubernetes.

Una técnica operativa que se ha consolidado para Kubernetes es la combinación de simulación ligera con inyección de fallos en vivo. Primero se usa un modelo topológico mínimo -un grafo de dependencias bloqueantes con conteo de réplicas- para estimar disponibilidad a nivel de endpoints bajo fallos fail-stop; luego se planifican campañas de caos enfocadas en los puntos con mayor sensibilidad (Krasnovsky, 2025).

En la práctica, el grafo se descubre automáticamente a partir de trazas distribuidas (OpenTelemetry/Jaeger), telemetría de malla, manifiestos de Kubernetes e infraestructura como código. Posteriormente, se ejecutan simulaciones Monte Carlo que muestrean fallos por réplica y evalúan alcanzabilidad en el subgrafo de servicios vivos, generando una señal de postura de resiliencia útil para CI/CD (Krasnovsky, 2025).

Los resultados experimentales sobre un benchmark de microservicios (DeathStarBench) indican alta correspondencia entre lo que predice el modelo y lo que se observa al inyectar fallos de forma real bajo carga controlada (wrk2), en especial cuando hay replicación. A tasas medias de fallo ($p\_fail \approx 0{,}3$) el error de estimación se reduce y el efecto de la réplica domina la forma de la curva de degradación; sin réplica, emergen sesgos sistemáticos esperables por puntos únicos de falla (Krasnovsky, 2025).

Esta estrategia no sustituye a la CE experimental, sino que prioriza escenarios y reduce el costo y riesgo de campañas extensas en producción simulada, reservándolas para validación y sintonía fina allí donde la topología sugiere mayor fragilidad (Krasnovsky, 2025).

La medición en Kubernetes también ha madurado. Además de métricas operativas como MTTD y MTTR, hoy se priorizan señales de disponibilidad percibida por el usuario durante el experimento (por ejemplo, proporción de solicitudes sin 5xx ni timeouts), a fin de alinear resultados con SLO/SLI e incidentes reales. Se recomienda no confundir indisponibilidad con redirecciones o ciertos 4xx (que semánticamente no son “caídas”) para evitar sobre o subestimación de confiabilidad (Krasnovsky, 2025).

En paralelo, los modelos ligeros ofrecen una capa de anticipación: al identificar cadenas de bloqueo con servicios no replicados, ayudan a explicar por qué el MTTR observado no mejora sin cambios arquitectónicos (réplicas, desacople, bulkheads), aun cuando los tiempos de reprogramación de pods sean buenos (Krasnovsky, 2025).

En cuanto al alcance de despliegue, las organizaciones aplican CE en nube pública, privada e híbrida. En privada, la menor redundancia geográfica hace más visibles los impactos de fallos de hardware o cuellos de capacidad; en híbrida, la interdependencia entre dominios exige experimentar con latencias y particiones para evaluar continuidad operativa y consistencia de datos (Owotogbe, Kumara, van den Heuvel, & Tamburri, 2024/2025). En todos los casos, la observabilidad (trazas, métricas y logs) es el sustrato para definir estado estable, detectar regresiones y atribuir causas tras una inyección, cerrando el ciclo con acciones correctivas priorizadas (Fossati et al., 2025).

Una ampliación emergente de estas aplicaciones mira a sistemas de agentes basados en LLM (LLM-MAS). Aunque no son “Kubernetes-solo”, en la práctica corren sobre infra cloud-native y se benefician de sus primitivas. Aquí, la CE se adapta para inyectar fallos de comunicación entre agentes, retrasos, caídas de agentes y fallos de coordinación, con métricas de robustez específicas (por ejemplo, task success rate, latency to goal, degradación de calidad/precisión) y guardrails fuertes por la sensibilidad del dominio (Owotogbe, 2025). Esta línea aprovecha los aprendizajes de CE en microservicios/Kubernetes -hipótesis–experimento–observación, automatización, contención- y los extiende a modos de fallo semánticos y emergentes (Owotogbe, 2025).

En síntesis, la sección 6.2 muestra cómo los fundamentos de la CE se concretan en Kubernetes y cloud-native. Sobre este terreno, la sección siguiente amplía el foco hacia los patrones de resiliencia que estructuran las arquitecturas distribuidas sobre las que se ejecutan estos experimentos.

2.3. Patrones de resiliencia en arquitecturas distribuidas

En arquitecturas distribuidas y, en particular, en sistemas cloud-native basados en microservicios, la resiliencia no se logra únicamente con "buena infraestructura", sino mediante un repertorio de patrones de diseño y operación que buscan limitar el impacto de los fallos, mantener niveles aceptables de servicio y facilitar la recuperación controlada. La literatura reciente sobre microservicios muestra que estos patrones se han ido consolidando como "bloques de construcción" reutilizables que expresan decisiones recurrentes sobre aislamiento de fallos, redundancia, manejo de estados, transacciones distribuidas y observabilidad (Di Francesco, Lago, & Malavolta, 2019; Valdivia et al., 2020; Velepucha & Flores, 2023).

En esta sección se revisan los patrones de resiliencia más relevantes para arquitecturas distribuidas modernas, enfatizando su papel en la mitigación de fallas en cascada y en la reducción de métricas como MTTD y MTTR, en coherencia con el enfoque de Chaos Engineering descrito en las secciones 6.1 y 6.2.

2.3.1. Panorama y clasificaciones de patrones de resiliencia

Los patrones de resiliencia se pueden agrupar, de forma general, en cuatro familias: (a) patrones de aislamiento y contención del fallo (por ejemplo, bulkhead, circuit breaker, service mesh con políticas de tráfico); (b) patrones de tolerancia a fallos en la comunicación (retries, timeouts, backoff, load shedding, backpressure); (c) patrones de gestión de estado y consistencia (sagas, TCC, idempotencia, CQRS, event sourcing, cachés con invalidación controlada); y (d) patrones de observabilidad y recuperación (health checks, readiness/liveness, logging estructurado, tracing distribuido, tableros de SLO/SLI) (Valdivia et al., 2020; Velepucha & Flores, 2023).

Los mapeos sistemáticos y revisiones multivocales subrayan que estos patrones no aparecen de forma aislada, sino combinados en "constelaciones" que responden a objetivos de calidad específicos como disponibilidad, tolerancia a fallos transaccionales o degradación elegante (Di Francesco et al., 2019; Valdivia et al., 2020). Por ejemplo, un sistema de pagos en línea que prioriza consistencia y disponibilidad puede combinar balanceadores de carga con retries y circuit breakers aguas arriba, mientras relega la consistencia fuerte a un pequeño núcleo de servicios de liquidación que utilizan patrones de transacción distribuida.

Desde la perspectiva de arquitectura, los patrones de resiliencia se solapan fuertemente con patrones estructurales (por ejemplo, API Gateway, service registry, message broker) porque la forma en que se conectan los servicios condiciona los caminos de propagación

del fallo (Di Francesco et al., 2019). Las revisiones multivocales muestran que la comunidad tiende a clasificar los patrones tanto por propósito (resiliencia, escalabilidad, seguridad) como por nivel (código, servicio, infraestructura), lo que refuerza la idea de resiliencia como propiedad emergente de decisiones en varios niveles a la vez (Valdivia et al., 2020).

En términos de adopción, los estudios de patrones relacionados con microservicios evidencian una fuerte presencia de circuit breaker, retry, bulkhead, timeouts, cachés y colas asincrónicas como repertorio mínimo. En cambio, patrones más avanzados (por ejemplo, event sourcing, CQRS, sagas coreografiadas) se concentran en dominios de negocio con alta complejidad transaccional (Valdivia et al., 2020; Velepucha & Flores, 2023). Este núcleo de patrones básicos es precisamente el que más se cruza con las prácticas de Chaos Engineering, pues define las rutas de degradación y recuperación que se exploran en los experimentos.

2.3.2. Aislamiento de fallos y contención del "blast radius"

Uno de los objetivos centrales de la resiliencia en arquitecturas distribuidas es evitar que un fallo local se propague y se convierta en un incidente sistémico. Patrones como bulkhead, circuit breaker y la segmentación explícita de dominios de fallo (por ejemplo, shards de usuarios o tenants) pretenden, justamente, limitar el blast radius: la porción del sistema afectada por una degradación (Mendonça, Aderaldo, Câmara, & Garlan, 2020; Hlybovets & Paprotskyi, 2024).

El patrón bulkhead consiste en particionar recursos (pools de conexiones, hilos, pods) por servicio, cliente o tipo de carga, de manera que la saturación en una partición no agote recursos críticos compartidos. Estudios analíticos de resiliencia de patrones muestran que la partición cuidadosa de recursos reduce la probabilidad de fallos catastróficos bajo escenarios de sobrecarga, a costa de cierto desperdicio de capacidad cuando el tráfico es desigual entre particiones (Mendonça et al., 2020).

El patrón circuit breaker, por su parte, intercepta llamadas hacia servicios remotos y "abre el circuito" cuando detecta tasas de error o latencias anómalas, devolviendo fallos controlados, respuestas en caché o degradación de funcionalidad, en lugar de permitir que las llamadas sigan bloqueándose en cascada (Di Francesco et al., 2019). Trabajos recientes sobre aumento de tolerancia a fallos en microservicios recomiendan combinar circuit breakers con umbrales dependientes del contexto (por ejemplo, adaptados a carga y prioridad de la petición) y con mecanismos de retry con backoff exponencial, para balancear mejor la probabilidad de recuperación frente al riesgo de sobrecarga adicional (Hlybovets & Paprotskyi, 2024).

La literatura también subraya el papel de las service meshes (por ejemplo, Istio, Linkerd) como "infraestructura de patrones". Muchas de estas capacidades de resiliencia se delegan a proxies sidecar que implementan, de forma declarativa, políticas de tiempo de espera,

reintento, corte de circuito, balanceo y detección de outliers (Calderón-Gómez et al., 2021; Hlybovets & Paprotskyi, 2024). Este enfoque desplaza parte de la resiliencia desde el código de aplicación hacia la capa de red y control, lo que facilita experimentar con configuraciones distintas mediante CE sin necesidad de redeployar los servicios.

De forma complementaria, patrones de replicación y failover (por ejemplo, múltiples instancias por zona de disponibilidad, topologías activo–activo) se integran con los mecanismos anteriores. Sin embargo, la evidencia empírica recuerda que la redundancia sin aislamiento puede incluso empeorar las cosas: fallos lógicos o de configuración replicados idénticamente pueden disparar fallas coordinadas, fenómeno que la práctica de CE busca detectar a través de experimentos que simulan fallos de zona, región o dependencia compartida (Yang et al., 2024; Soldani, Forti, Roveroni, & Brogi, 2024).

2.3.3. Patrones de resiliencia transaccional y consistencia distribuida

En arquitecturas distribuidas, la resiliencia no solo implica seguir respondiendo, sino también preservar propiedades mínimas de consistencia y corrección de negocio. Los patrones de transacciones distribuidas -en particular las sagas- han ganado relevancia como enfoque pragmático para mantener consistencia eventual sin bloquear todo el sistema con protocolos estrictos como 2PC (Daraghmi, Zhang, & Yuan, 2022).

El patrón saga modela una transacción distribuida como una secuencia de transacciones locales, cada una con su acción de compensación. Si una etapa falla, las operaciones previas se deshacen mediante compensating transactions que restauran el estado a una situación coherente desde la perspectiva del negocio. Investigaciones recientes han resaltado, sin embargo, que las sagas en su forma básica presentan problemas de aislamiento: otros servicios pueden observar estados intermedios que aún no deberían ser visibles (Daraghmi et al., 2022).

Daraghmi et al. (2022) proponen un patrón de saga mejorado que introduce una capa de “quota cache” y un servicio de commit sincronizado para evitar escrituras definitivas en la base de datos hasta que la saga completa se ha verificado. De este modo, las transacciones intermedias afectan solo al caché, lo que reduce el riesgo de inconsistencias permanentes en presencia de fallos y mejora el rendimiento bajo ciertas cargas de trabajo al desplazar operaciones a memoria. Este tipo de refinamiento muestra cómo los patrones de resiliencia evolucionan a partir de la práctica: se identifican limitaciones en la aplicación real de un patrón y se proponen extensiones que equilibran mejor aislamiento, latencia y uso de recursos.

En paralelo, otras tecnologías combinan sagas con colas de mensajes, CQRS y event sourcing para desacoplar escritura y lectura, de forma que las vistas de lectura sean más robustas a fallos temporales en el pipeline de escritura (Valdivia et al., 2020; Calderón-Gómez et al., 2021). Por ejemplo, en plataformas de eHealth evaluadas por Calderón-Gómez et al. (2021), las arquitecturas basadas en microservicios con patrones de

mensajería y sagas mostraron mejor capacidad para absorber picos de carga y fallos parciales sin perder trazabilidad de los eventos clínicos, en comparación con variantes más monolíticas o centradas en servicios “gordos”.

La resiliencia transaccional también se beneficia de patrones como idempotencia (que permite repetir operaciones sin efectos adversos), colas de outbox para garantizar la entrega fiable de mensajes y transactional outbox/inbox para coordinar cambios entre bases de datos y brokers de mensajes. Estos patrones reducen la probabilidad de estados “fantasma” -por ejemplo, órdenes cobradas pero no enviadas, o notificaciones emitidas sin que exista la acción subyacente-, que son precisamente el tipo de anomalías que los experimentos de CE buscan exponer bajo fallos de red, reintentos y duplicación de mensajes (Mendonça et al., 2020; Daraghmi et al., 2022).

2.3.4. Patrones de observabilidad y manejo de fallas en cascada

La observabilidad es un componente inseparable de la resiliencia: sin capacidad para observar y explicar el comportamiento del sistema bajo falla, los patrones de resiliencia se convierten en “cajas negras” difíciles de gobernar. La literatura reciente sobre análisis de fallas en cascada en microservicios enfatiza el papel de patrones de observabilidad como logging estructurado, trazas distribuidas, métricas etiquetadas por servicio y flujo, y correlación mediante IDs de contexto (Soldani et al., 2024; Yang et al., 2024).

Soldani et al. (2024) estudian fallas en cascada a partir de logs de microservicios y muestran que, aun cuando existen patrones clásicos de resiliencia desplegados (por ejemplo, retries, timeouts, circuit breakers), la falta de visibilidad y de correlación adecuada entre eventos dificulta diagnosticar la cadena real de propagación. Entre sus hallazgos destaca la importancia de registrar explícitamente el estado de los patrones de resiliencia (por ejemplo, transición de un circuit breaker a estado abierto, número de reintentos agotados) como parte del contexto de observabilidad, de modo que se pueda inferir no solo “qué falló”, sino “qué patrones intervinieron y cómo”.

Por su parte, Yang et al. (2024) introducen MicroRes, un marco para perfilar la resiliencia a partir de la “difusión de degradación”. Se inyectan fallos en microservicios y se analiza qué tan lejos se propaga la degradación desde métricas de bajo nivel (CPU, latencia interna) hacia métricas centradas en el usuario (latencia extremo a extremo, errores percibidos). Sus resultados evidencian que las configuraciones de patrones de resiliencia (por ejemplo, parámetros de retries y timeouts, políticas de degradación) modulan significativamente la forma de las curvas de degradación y el grado en que los fallos se traducen en impacto real para el usuario.

Estos trabajos apuntan hacia un patrón emergente de “observabilidad consciente de resiliencia”, donde los patrones de diseño y los patrones de medición se diseñan juntos. En la práctica, esto se traduce en incluir dentro del contrato de cada servicio no solo endpoints de health check, sino también señales específicas de resiliencia (por ejemplo, tasa de

circuit breakers abiertos, cola de mensajes pendientes, backlog de sagas en compensación) que luego se monitorean mediante tableros de SLO/SLI. Desde la óptica de CE, esto proporciona mejores métricas para definir el estado estable y para evaluar el efecto de las inyecciones de fallo (Basiri et al., 2016; Yang et al., 2024).

2.3.5. Composición de patrones, trade-offs y límites

Si bien la literatura presenta los patrones de resiliencia como "buenas prácticas", también reconoce sus costos y trade-offs. Hlybovets y Paprotskyi (2024) muestran, por ejemplo, que la introducción de múltiples capas de tolerancia a fallos (retries, replicación, circuit breakers) puede incrementar de manera significativa la complejidad configuracional del sistema y, en algunos escenarios, aumentar la latencia promedio y el consumo de recursos sin mejoras equivalentes en disponibilidad.

Un problema recurrente es la interacción no trivial entre patrones. Retries agresivos combinados con timeouts demasiado largos pueden amplificar una degradación: en lugar de permitir que un servicio falle rápidamente y active mecanismos de fallback, la arquitectura queda atrapada en reintentos costosos que agotan recursos aguas arriba, tal como se observa en estudios de fallas en cascada (Soldani et al., 2024). De forma análoga, patrones de caché mal configurados pueden ocultar fallos de consistencia durante un intervalo, solo para revelarlos más tarde de forma masiva cuando expira el caché o se produce una invalidación generalizada.

Los estudios de patrones a nivel de arquitectura recomiendan, por tanto, diseñar la resiliencia como una "capacidad orquestada", donde cada patrón se justifica a partir de objetivos de calidad concretos y de escenarios de fallo plausibles, en lugar de aplicarse de forma homogénea en todo el sistema (Di Francesco et al., 2019; Valdivia et al., 2020). En este sentido, la combinación de análisis basado en modelos y experimentación sistemática ayuda a seleccionar y dimensionar patrones: Mendonça et al. (2020) muestran que es posible modelar arquitecturas con distintos patrones de resiliencia (por ejemplo, redundancia, retries, degradación) y estimar de antemano su efecto sobre disponibilidad, lo que sirve para filtrar opciones antes de llevarlas a producción.

MicroRes sigue una lógica similar al priorizar configuraciones que minimizan la difusión de degradación bajo inyecciones de fallo, lo que sugiere una sinergia entre herramientas de simulación/profiling y patrones de diseño (Yang et al., 2024). En la práctica, esto permite abordar la resiliencia como un problema de optimización multiobjetivo -disponibilidad, latencia, costo, complejidad operativa- más que como una simple acumulación de "best practices".

2.3.6. Conexión entre patrones de resiliencia y Chaos Engineering

Los patrones de resiliencia y la Ingeniería del Caos pueden entenderse como dos caras de la misma moneda. Por un lado, los patrones proporcionan el vocabulario técnico para estructurar arquitecturas distribuidas robustas; por otro, la CE ofrece el método

experimental para validar si esos patrones, tal como están implementados y configurados, realmente producen la resiliencia esperada en condiciones production-like (Basiri et al., 2016; Mendonça et al., 2020).

En el enfoque basado en modelos, trabajos como el de Mendonça et al. (2020) utilizan patrones de resiliencia como unidades de diseño que se pueden insertar o retirar de un modelo arquitectónico para comparar escenarios de disponibilidad. De forma complementaria, marcos como MicroRes o las propuestas de análisis de fallos en cascada desde logs permiten observar, bajo inyección de fallos, qué patrones están funcionando como barreras efectivas y cuáles resultan insuficientes o mal parametrizados (Soldani et al., 2024; Yang et al., 2024).

Para la presente investigación, esta conexión es clave: el conjunto de patrones de resiliencia adoptados en una arquitectura cloud-native sobre Kubernetes condiciona directamente el diseño de los experimentos de CE. Por ejemplo:

Definición de hipótesis: las hipótesis pueden formularse explícitamente en términos de patrones (“si el circuit breaker del servicio X se abre bajo fallo de la dependencia Y, la tasa de error percibida por el usuario no superará Z %”).

Selección de escenarios de fallo: los escenarios se eligen para tensionar patrones específicos (por ejemplo, fallos de nodos y pods para validar bulkheads y replicación; fallos de base de datos para evaluar sagas e idempotencia).

Métricas de evaluación: las métricas se diseñan para capturar la actuación de los patrones (tiempo hasta que se abre un circuit breaker, tiempo de compensación de sagas, profundidad de la cascada observada antes de que se active la degradación elegante).

Asimismo, la adopción de patrones de resiliencia condiciona el triage basado en modelos y simulación descrito en la sección 6.1. Los grafos de dependencias y réplicas que se utilizan para estimar disponibilidad pueden enriquecerse con información sobre qué patrones aplican a cada enlace o servicio (por ejemplo, presencia de circuit breaker, número máximo de retries, tipo de aislamiento de recursos). Esto permitiría refinar aún más las estimaciones y priorizar experimentos de CE en aquellos segmentos del grafo donde los patrones son más débiles o inexistentes (Mendonça et al., 2020; Hlybovets & Paprotskyi, 2024).

En síntesis, los patrones de resiliencia en arquitecturas distribuidas constituyen la “gramática” con la que se expresan decisiones de tolerancia a fallos, mientras que la CE representa el “método científico” para validar esas decisiones bajo perturbaciones controladas. La literatura reciente converge en una visión integrada: arquitecturas distribuidas donde patrones como circuit breaker, bulkhead, sagas, observabilidad avanzada y replicación se diseñan, modelan y experimentan de manera conjunta, con el objetivo de alcanzar resiliencia medible y alineada con los SLO del negocio (Di Francesco

et al., 2019; Valdivia et al., 2020; Daraghmi et al., 2022; Calderón-Gómez et al., 2021; Soldani et al., 2024; Yang et al., 2024).

Concluida la revisión de patrones, la sección siguiente se centra en las herramientas y frameworks que materializan esta "gramática de resiliencia" dentro de plataformas concretas de Chaos Engineering.

2.4. Herramientas y frameworks de implementación

La Ingeniería del Caos dejó rápidamente de ser una práctica artesanal ligada a scripts ad hoc para convertirse en un ecosistema de herramientas y frameworks especializados que cubren el ciclo completo de experimentación: definición de experimentos, inyección de fallos, observabilidad, análisis de resultados y automatización en pipelines de entrega continua. La literatura reciente describe este ecosistema como un conjunto de "plataformas de caos" que encapsulan capacidades de orquestación, integración con la infraestructura cloud y soporte para patrones de resiliencia, reduciendo la fricción de adopción en organizaciones que no cuentan con equipos altamente especializados (Basiri et al., 2016; Owotogbe et al., 2024; Yadav, 2024).

En términos generales, las herramientas de CE se pueden agrupar en cuatro familias: (a) inyectores de fallos de bajo nivel acoplados a una tecnología específica (por ejemplo, Kubernetes o Docker); (b) plataformas de orquestación y automatización de experimentos; (c) frameworks de dominio que combinan caos con requisitos particulares como seguridad, systems of systems o infraestructuras críticas; y (d) plataformas emergentes que integran CE con enfoques basados en IA o políticas declarativas para entornos multi-cloud e inteligentes (Opara et al., 2025; Torkura et al., 2020). Esta taxonomía refleja el paso de "herramientas individuales" a "ecosistemas" más amplios donde la CE se comporta como una capa transversal de resiliencia integrada en las prácticas DevOps/SRE (Basiri et al., 2016; Owotogbe et al., 2024).

2.4.1 Plataformas de caos como líneas de producto de software

Una línea de trabajo relevante considera la CE no solo como un conjunto de scripts aislados, sino como una línea de producto de software configurable que permite derivar plataformas de caos adaptadas a diferentes organizaciones y escenarios. Camacho et al. (2022) proponen Pystol, una plataforma de CE concebida explícitamente como línea de producto para entornos híbridos y multi-cloud, en la que los "experimentos de caos" se modelan como características (features) que pueden ser seleccionadas y combinadas en función de las necesidades de resiliencia de cada sistema.

En Pystol, las features incluyen tanto tipos de fallos (terminación de instancias, degradación de red, estrés de CPU/memoria) como integraciones con proveedores cloud (por ejemplo, Amazon Web Services, Microsoft Azure o plataformas on-premise), así como conectores de observabilidad. El diseño como línea de producto permite razonar sobre la variabilidad y reusabilidad, de modo que la organización puede construir catálogos

de experimentos reutilizables alineados con sus patrones de resiliencia (por ejemplo, timeouts, retries, circuit breakers) y sus restricciones de negocio (Camacho et al., 2022).

Esta aproximación representa un cambio conceptual importante: en lugar de desplegar una herramienta monolítica de caos, se construye una plataforma configurable que encapsula decisiones de arquitectura, seguridad y gobernanza, y que puede evolucionar a medida que cambian las arquitecturas distribuidas y los requisitos de resiliencia. Desde la perspectiva del marco teórico, Pystol ilustra cómo la CE se articula con ingeniería de líneas de producto y con la noción de plataformas de resiliencia componibles (Camacho et al., 2022; Opara et al., 2025).

2.4.2 Herramientas cloud-native para Kubernetes y microservicios

En el contexto de arquitecturas cloud-native y Kubernetes, ha surgido una familia de herramientas específicamente orientada a inyectar fallos dentro de clústeres de contenedores. Las revisiones recientes destacan que Chaos Mesh, LitmusChaos, ChaosBlade, ChaosToolkit, PowerfulSeal y Chaoskube conforman un núcleo maduro de plataformas de CE para Kubernetes, utilizadas tanto en industria como en investigación (Mailewa et al., 2025; Che, 2025; Owotogbe et al., 2024).

Chaos Mesh se presenta como una plataforma de CE para Kubernetes que ofrece un catálogo amplio de experimentos (fallos de pod, disrupciones de red, estrés de CPU/memoria, errores a nivel de sistema de archivos) declarados mediante CRDs (Custom Resource Definitions). Esto permite describir experimentos como objetos nativos de Kubernetes y combinarlos en flujos temporales, además de integrarse con observabilidad basada en métricas y trazas (Che, 2025).

LitmusChaos adopta un enfoque similar, también basado en CRDs, pero enfatiza una colección curada de “experimentos Litmus” empaquetados como chaos charts que pueden instalarse y versionarse, facilitando la estandarización de campañas de caos entre equipos y proyectos (Mailewa et al., 2025).

Herramientas como ChaosBlade, PowerfulSeal y Chaoskube operan como capas de inyección más ligeras. Por ejemplo, Chaoskube ofrece terminación aleatoria de pods en base a etiquetas o namespaces, mientras PowerfulSeal permite definir políticas de fallo declarativas para nodos y pods, incluyendo escenarios complejos de indisponibilidad transitoria (Che, 2025). ChaosToolkit, aunque no es exclusivo de Kubernetes, aporta un lenguaje declarativo para describir experimentos en forma de JSON/YAML, con extensiones (drivers) que permiten operar sobre Kubernetes, servicios cloud y otros recursos; su foco está en la reproducibilidad y la integración con pipelines CI/CD (Che, 2025; Owotogbe et al., 2024).

Desde la perspectiva del marco teórico, estas herramientas materializan los principios de CE en entornos cloud-native al ofrecer: (a) un vocabulario declarativo de fallos consistente

con el modelo de recursos de Kubernetes; (b) control explícito del blast radius mediante selectors y namespaces; y (c) mecanismos de observación integrados con Prometheus, OpenTelemetry u otras soluciones de monitoreo. De este modo, se convierten en mecanismos concretos para operacionalizar patrones de resiliencia como autoescalado, degradación elegante y bulkheads en microservicios (Basiri et al., 2016; Mailewa et al., 2025).

2.4.3 Frameworks de orquestación y evaluación de resiliencia

Más allá de los inyectores de fallos, la literatura identifica frameworks cuyo objetivo principal es orquestar y evaluar sistemáticamente la resiliencia de sistemas complejos, combinando CE con modelos de auto-adaptación y métricas formales. El framework CHESS (Chaos Engineering for Self-Adaptive Systems) propone una arquitectura que integra CE con evaluación continua de sistemas auto-adaptativos, definiendo componentes específicos para selección de experimentos, ejecución controlada, monitorización y análisis de resultados (Malik et al., 2023).

En CHESS, los experimentos se derivan a partir de objetivos de calidad (por ejemplo, disponibilidad, tiempo de respuesta) y de configuraciones de auto-adaptación, de manera que la inyección de fallos se utiliza para evaluar si los mecanismos adaptativos son capaces de mantener las propiedades deseadas bajo perturbaciones. Esto lo convierte en un framework puente entre CE y la ingeniería de sistemas auto-adaptativos (Malik et al., 2023). Desde una perspectiva teórica, CHESS muestra cómo la CE puede integrarse en ciclos de control MAPE-K (Monitor–Analyze–Plan–Execute) y cómo los experimentos se convierten en “pruebas activas” de las políticas de adaptación, en lugar de simples pruebas de estrés.

En la misma línea, Bailey et al. (2022) proponen un framework de CE para evaluar la resiliencia de systems of systems, donde múltiples sistemas heterogéneos interactúan para brindar capacidades críticas. Su enfoque incluye una arquitectura modular con generadores de fallos, orquestadores de experimentos y módulos de análisis que miden la degradación funcional y la capacidad del sistema de sistemas para recuperar su funcionalidad. Este tipo de marcos resulta relevante para entornos donde la resiliencia no se limita a un servicio aislado, sino a cadenas completas de capacidades interdependientes, como en defensa, transporte o infraestructuras críticas (Bailey et al., 2022).

Por su parte, Opara et al. (2025) plantean un marco conceptual al que denominan “Chaos Engineering 2.0”, centrado en la integración de CE con políticas declarativas y mecanismos impulsados por IA para entornos multi-cloud. En este enfoque, los frameworks de CE se conciben como controladores capaces de tomar decisiones automáticas sobre qué experimentos ejecutar, dónde y con qué frecuencia, en función de políticas de riesgo, cumplimiento normativo y objetivos de negocio. Esto amplía el alcance de los frameworks de orquestación, que ya no se limitan a ejecutar scripts, sino que implementan una capa de “gobernanza del caos” alineada con la gestión de riesgos

organizacionales (Opara et al., 2025).

2.4.4 Herramientas de CE orientadas a seguridad y dominios específicos

Una tendencia creciente es la aparición de herramientas y frameworks de CE centrados en dominios particulares, especialmente seguridad y sistemas ciber-físicos. Torkura et al. (2020) presentan CloudStrike, una plataforma de CE orientada a seguridad para infraestructuras cloud que combina inyección de fallos con escenarios de ataque controlados, con el fin de evaluar tanto la resiliencia operacional como la efectividad de controles de seguridad. CloudStrike integra módulos de generación de escenarios de ataque, ejecución controlada y análisis forense, lo que permite a las organizaciones practicar security chaos engineering en un entorno estructurado (Torkura et al., 2020).

Este tipo de frameworks refleja la evolución del concepto tradicional de CE -centrado en disponibilidad y rendimiento- hacia una visión donde la resiliencia incluye también la capacidad de resistir y recuperarse de incidentes de seguridad. De hecho, revisiones recientes enfatizan que las herramientas de CE orientadas a seguridad tienden a incorporar catálogos de escenarios basados en modelos de amenazas y a integrarse con sistemas SIEM, plataformas de respuesta a incidentes y marcos de cumplimiento (Yadav, 2024; Torkura et al., 2020).

En systems of systems y dominios de infraestructura crítica, Bailey et al. (2022) muestran que las herramientas de CE deben adaptarse a restricciones de seguridad física, regulaciones estrictas y altos costos de fallo. En estos contextos, los frameworks privilegian entornos de prueba de alta fidelidad (bancos de pruebas, gemelos digitales) y mecanismos estrictos de contención del blast radius, lo que se traduce en herramientas que priorizan la simulación y emulación frente a la inyección directa en producción. Esto implica que la "implementación de CE" en dichos dominios se apoya tanto en plataformas de simulación como en inyectores físicos o emuladores de red, con un fuerte acento en la trazabilidad y la reproducibilidad de los experimentos (Bailey et al., 2022).

2.4.5 Plataformas de experimentación y automatización a escala

Otro grupo de contribuciones se centra en plataformas de experimentación a gran escala que permiten ejecutar campañas de caos de manera continua y automatizada. Jernberg y Runeson (2020) describen el diseño de una plataforma de experimentación para CE desplegada en un entorno global, resaltando la necesidad de abstraer la complejidad de la infraestructura subyacente mediante capas de orquestación, registro centralizado de experimentos y mecanismos de control de riesgos. Su trabajo destaca que, para escalar la CE más allá de pilotos y experimentos aislados, se requieren plataformas que gestionen la coordinación de múltiples herramientas, la programación de experimentos y la integración con sistemas de ticketing y gestión de incidencias (Jernberg & Runeson, 2020).

Las revisiones multivocales de Owotogbe et al. (2024) coinciden con esta visión y muestran que las organizaciones que han avanzado más en CE tienden a construir

“plataformas internas” que envuelven herramientas existentes (Chaos Monkey, Gremlin, Chaos Mesh, LitmusChaos, Azure Chaos Studio, AWS Fault Injection Simulator, entre otras) detrás de APIs internas o paneles de autoservicio. En dichos entornos, la principal “herramienta” ya no es un binario aislado, sino una plataforma de resiliencia que incluye catálogos de experimentos, plantillas, controles de acceso y flujos de aprobación, así como integraciones con observabilidad y CI/CD (Owotogbe et al., 2024; Yadav, 2024).

Desde esta perspectiva, la implementación de CE implica combinar: (a) herramientas de inyección de fallos específicas de la plataforma; (b) frameworks de orquestación como ChaosToolkit, Pystol o CHESS; y (c) componentes de automatización como jobs de Jenkins, pipelines de GitHub Actions o flujos declarativos con Argo Workflows o Tekton, todo ello conectado a sistemas de observabilidad que permiten medir el steady state y detectar degradaciones (Basiri et al., 2016; Camacho et al., 2022; Malik et al., 2023). Teóricamente, esto consolida la CE como una “disciplina de plataforma”, en la que las herramientas individuales se subordinan a una arquitectura global de experimentación y aprendizaje continuo.

2.4.6 Integración con devops, observabilidad e IA

La literatura reciente insiste en que las herramientas de CE solo alcanzan su máximo potencial cuando se integran de manera estrecha con las prácticas DevOps, la observabilidad y, crecientemente, con técnicas de IA. Yadav (2024) argumenta que, en el contexto de SRE, las plataformas de CE se convierten en extensiones naturales de los pipelines de entrega, donde los experimentos de caos se ejecutan como etapas no funcionales que validan el cumplimiento de SLO antes de promover un despliegue a producción. Esto requiere que las herramientas de CE se integren nativamente con sistemas de métricas (Prometheus, CloudWatch), tracing distribuido (Jaeger, Zipkin) y logging centralizado, así como con mecanismos de feature flags y canary releases (Yadav, 2024).

Opara et al. (2025) amplían este argumento proponiendo que la próxima generación de frameworks de CE incorporará capacidades impulsadas por IA, tanto para seleccionar de forma inteligente los experimentos más relevantes como para analizar patrones en los datos de observabilidad y detectar configuraciones frágiles. En su visión, las herramientas de CE dejarán de ser únicamente ejecutores de experimentos predefinidos para convertirse en agentes que recomiendan nuevos escenarios de caos basados en análisis de riesgos, dependencias arquitectónicas y comportamiento histórico del sistema (Opara et al., 2025).

Che (2025), al proponer un framework de evaluación de resiliencia de Kubernetes en entornos cloud-edge, muestra una integración estrecha entre un orquestador de caos y herramientas externas como Chaos Mesh, ChaosToolkit y plataformas de carga, coordinadas mediante un controlador central que gestiona la ejecución de experimentos, la recolección de datos y el análisis de resiliencia. Este tipo de diseños ilustra cómo las herramientas de CE se combinan con marcos de carga, simuladores y bibliotecas de análisis estadístico para producir evaluaciones de resiliencia más ricas y automatizadas.

Finalmente, las revisiones de Mailewa et al. (2025) y Owotogbe et al. (2024) subrayan que las organizaciones están convergiendo hacia un modelo donde las herramientas de CE se integran con catálogos de patrones de resiliencia y métricas estandarizadas (por ejemplo, MTTR, tasas de error, disponibilidad percibida). Esto facilita comparar arquitecturas, justificar inversiones y priorizar refactorizaciones. En este modelo, las herramientas y frameworks de CE dejan de ser artefactos periféricos para convertirse en componentes centrales de la gobernanza de resiliencia, estrechamente vinculados con la gestión de riesgos, la arquitectura de referencia y la ingeniería de fiabilidad (Basiri et al., 2016; Bailey et al., 2022; Yadav, 2024).

Como puente hacia la dimensión cuantitativa, la siguiente sección aborda las métricas mediante las cuales se evalúa la resiliencia, conectando directamente las herramientas y patrones descritos con decisiones de diseño y operación.

2.5. Métricas de evaluación de resilencia

La evolución de la Ingeniería del Caos hacia un "sistema operativo de aprendizaje de resiliencia" hace que la selección de métricas deje de ser un detalle técnico y se convierta en una decisión de diseño central. Medir resiliencia ya no se limita a verificar si un componente "está arriba o abajo", sino a capturar cómo se degrada y recupera el servicio bajo perturbaciones, qué tan bien se cumplen los SLO y cómo contribuye la arquitectura (grafo de dependencias, patrones de resiliencia) al comportamiento emergente del sistema (Basiri et al., 2016; Poulin & Kane, 2021).

2.5.1. Conceptualización de resiliencia y espacio de métricas

La literatura de ingeniería de resiliencia distingue entre métricas de desempeño en el tiempo, métricas estructurales y métricas multidimensionales. Yodo y Wang (2016) muestran que, aunque existe consenso en que la resiliencia combina capacidad de absorción, adaptación y recuperación, no hay una única métrica universal; más bien, hay familias de métricas que dependen del tipo de sistema y de la decisión de diseño que se busca informar. En arquitecturas distribuidas, esta diversidad se acentúa, porque la resiliencia depende a la vez de propiedades locales (replicación, timeouts, políticas de reintento) y globales (topología de dependencias, propagación de fallos, capacidad de aislar fallas).

Desde esta perspectiva, una métrica de resiliencia es un mapeo entre observaciones del sistema bajo perturbación (medidas de desempeño, estados de componentes, trazas) y una escala interpretable para la organización (por ejemplo, probabilidad de completar una transacción sin error, área bajo la curva de desempeño, probabilidad de no violar un SLO dado un perfil de fallos) (Yodo & Wang, 2016; Poulin & Kane, 2021). Esto implica que:

- No existe "la" métrica de resiliencia, sino combinaciones de métricas alineadas con objetivos (disponibilidad, continuidad del negocio, experiencia de usuario, costo);

- Las métricas deben ser operacionalmente accionables, es decir, deben informar decisiones de arquitectura (replicar, desacoplar, introducir circuit breakers) y de operación (ajustar SLO, cambiar políticas de reintento, priorizar experimentos de caos); y
- En sistemas cloud-native, las métricas deben ser compatibles con la observabilidad existente (trazas, métricas, logs) y con los flujos de CI/CD donde se integran los experimentos de caos (Basiri et al., 2016; Murphy et al., 2016).

2.5.2. Métricas temporales basadas en curvas de resiliencia

Una familia clásica de métricas parte de las curvas de resiliencia: funciones que describen el desempeño del sistema antes, durante y después de un evento disruptivo. Estas curvas permiten derivar métricas como el área bajo la curva, el tiempo de recuperación, la profundidad de la degradación o el tiempo por debajo de un umbral crítico (Poulin & Kane, 2021; Yodo & Wang, 2016).

Poulin y Kane (2021) sistematizan más de 270 publicaciones y muestran que la literatura utiliza distintos tipos de curvas (productividad, nivel de servicio, capacidad disponible, etc.) y un conjunto de métricas resumen, entre ellas:

- Área bajo la curva de resiliencia (AUC): integra el desempeño a lo largo del tiempo y captura simultáneamente severidad y duración de la degradación.
- Resiliencia normalizada: cociente entre el área observada y el área ideal (sin perturbación), útil para comparar escenarios y arquitecturas.
- Tiempos característicos: tiempo hasta el inicio de la degradación, tiempo de caída máxima, tiempo hasta recuperar un porcentaje del desempeño pre-evento.

Estudios comparativos muestran que diferentes métricas basadas en curvas pueden no ser equivalentes. Tang et al. (2023) comparan 12 métricas de resiliencia basadas en series de tiempo y encuentran que muchas miden aspectos parcialmente distintos (por ejemplo, algunas enfatizan la velocidad de recuperación mientras otras ponderan más la profundidad de la caída). Su conclusión es relevante para arquitecturas distribuidas: la elección de una métrica puede cambiar la evaluación de qué diseño es “más resiliente”, por lo que resulta crítico explicitar qué dimensión de resiliencia se busca priorizar (Tang et al., 2023).

En sistemas reales, estas métricas pueden construirse a partir de los SLI existentes (latencia, tasa de éxito, throughput) midiendo su trayectoria durante experimentos de caos. Por ejemplo, ante la terminación controlada de pods, se puede trazar la tasa de solicitudes exitosas en función del tiempo y derivar el área bajo la curva o el tiempo por debajo del umbral de SLO, conectando directamente la CE con decisiones de SRE (Poulin & Kane, 2021; Murphy et al., 2016).

2.5.3. Métricas operacionales en Ingeniería del Caos y SRE

En la práctica cloud-native, la CE se apoya fuertemente en métricas operacionales que ya

forman parte de las prácticas de SRE: disponibilidad, errores por millón de solicitudes, latencia de percentiles altos y tiempos asociados al ciclo de incidentes (Basiri et al., 2016; Murphy et al., 2016). Estas métricas permiten cuantificar el efecto de los patrones de resiliencia en términos directamente vinculados a la experiencia de usuario y a los acuerdos de servicio.

Entre las métricas más utilizadas se encuentran:

- Disponibilidad percibida por el usuario: proporción de solicitudes sin errores "críticos" (típicamente 5xx, timeouts y errores de transporte) durante el experimento; esta definición evita penalizar respuestas esperadas 3xx o ciertos 4xx que no representan indisponibilidad real (Murphy et al., 2016; Krasnovsky, 2025).

- MTTD (Mean Time To Detect) y MTTR (Mean Time To Recover): tiempos promedio para detectar un incidente y restaurar el servicio a niveles aceptables; su reducción es uno de los objetivos explícitos de CE en pipelines de DevOps (Basiri et al., 2016; He et al., 2022).
- Error budgets: fracción de tiempo o de solicitudes "presupuestadas" para incumplir el SLO sin violar el acuerdo; las campañas de caos pueden diseñarse explícitamente para consumir una porción controlada de ese budget y medir cuánto se reduce el riesgo de incidentes no controlados (Murphy et al., 2016).
- Índices de severidad de incidente: categorías cualitativas (Sev-1, Sev-2, etc.) que se pueden cuantificar como porcentajes de experimentos que desencadenan incidentes de cada nivel bajo distintos patrones de fallo, ayudando a priorizar refactorizaciones o patrones de resiliencia adicionales.

Krasnovsky (2025) propone una métrica de "resilience posture" que se deriva de la probabilidad estimada de completar solicitudes sin errores bajo distintos niveles de tasa de fallo, combinando resultados de simulación sobre el grafo de dependencias con resultados de inyección de fallos en vivo. Este tipo de métrica integra dos dimensiones: la robustez frente a fallos aleatorios en la topología y la capacidad de recuperación que muestran los experimentos de caos.

Un reto empírico es que los tiempos MTTD/MTTR pueden ser engañosos si no se considera la estructura de la arquitectura. He et al. (2022) muestran, en el contexto de microredes eléctricas, que modelos de resiliencia validados experimentalmente pueden explicar por qué el MTTR observado no mejora si persisten cuellos de botella estructurales, aun cuando los mecanismos de recuperación local son eficientes. Este resultado es extrapolable a microservicios: mejorar herramientas de observabilidad y automatizar rollbacks reduce MTTD/MTTR hasta cierto límite; más allá de ese punto, es la arquitectura (replicación, desacople, bulkheads) la que domina las ganancias de resiliencia (He et al., 2022; Yodo & Wang, 2016).

2.5.4. Métricas estructurales y topológicas para arquitecturas distribuidas

Las métricas temporales capturan “qué pasó y cuándo”, pero no explican fácilmente por qué ciertos puntos de la arquitectura son más frágiles. Por ello, en los últimos años han ganado tracción las métricas estructurales basadas en grafos de dependencias, que miden la resiliencia a partir de la estructura de la red de servicios (Yodo & Wang, 2016; Poulin & Kane, 2021).

En arquitecturas de microservicios, un modelo habitual es un grafo dirigido cuyos nodos son servicios y cuyas aristas representan dependencias bloqueantes. Sobre este grafo se pueden definir métricas como:

- Centralidad de intermediación (betweenness) de un servicio en rutas de peticiones críticas, que aproxima su rol como “punto de estrangulamiento” de resiliencia.
- Redundancia estructural: número de rutas alternativas entre un endpoint y sus dependencias críticas; baja redundancia implica mayor vulnerabilidad a fallos locales.
- Distribución de replicación: mapeo entre servicios y su número de réplicas, que puede combinarse con centralidad para priorizar dónde invertir en redundancia adicional.

Krasnovsky (2025) propone una métrica de disponibilidad estimada por endpoint que resulta de ejecutar simulaciones Monte Carlo sobre un grafo de dependencias con réplicas, bajo un modelo de fallos fail-stop. La métrica se define como la probabilidad de que exista al menos un camino funcional entre el endpoint y sus dependencias para un cierto porcentaje de servicios fallados. El autor muestra que esta métrica predice con alta precisión la tasa de éxito observada en experimentos de caos sobre el benchmark DeathStarBench, especialmente cuando existe replicación; los errores sistemáticos se corresponden con fenómenos no modelados (retries, fallos parciales), lo que la convierte en un indicador robusto de resiliencia estructural (Krasnovsky, 2025).

Estas métricas topológicas son especialmente útiles para triage de experimentos de caos. En lugar de inyectar fallos indiscriminadamente, se priorizan servicios con alta centralidad o baja redundancia y, a partir de los resultados, se reevalúan las métricas topológicas (por ejemplo, después de introducir nuevos bulkheads o de fragmentar un servicio muy central). Este ciclo métrica–experimento–refactorización traduce la CE en decisiones de arquitectura cuantificables (Yodo & Wang, 2016; Krasnovsky, 2025).

2.5.5. Métricas específicas para microservicios y arquitecturas cloud-native

En arquitecturas cloud-native, la resiliencia no se limita a “estar online”, sino a degradar de forma controlada ante fallos de componentes o picos de carga. Esto ha dado lugar a métricas específicas para microservicios que buscan capturar la forma en que los fallos se propagan y cómo los patrones de resiliencia (circuit breakers, retries, colas, limitación de concurrencia) modulan esa propagación.

Soldani et al. (2025) introducen métricas basadas en logs para explicar fallos en cascada, incluyendo: (a) longitud y anchura de las cascadas (cuántos servicios están involucrados), (b) contribución relativa de cada servicio a la cascada y (c) frecuencia de patrones de fallo repetidos que podrían mitigarse con patrones de resiliencia específicos. Su estudio de logs en una aplicación real muestra que estas métricas permiten identificar servicios "amplificadores" de fallos y evaluar ex post la efectividad de patrones como circuit breakers y bulkheads (Soldani et al., 2025).

Por otro lado, herramientas como MiSim proponen métricas de resiliencia sensibles a configuración: éxito de solicitudes bajo distintas combinaciones de patrones de resiliencia (por ejemplo, número de retries, política de timeouts, uso de colas) y perfiles de carga (Giamattei et al., 2022). Este tipo de métricas permite evaluar escenarios hipotéticos (what-if) sin necesidad de desplegar todos los cambios en producción, lo que complementa la CE en vivo.

Yang et al. (2024) introducen MicroRes, que define un índice de "degradation dissemination" para cuantificar cómo se propaga una degradación de desempeño en arquitecturas de microservicios. Su métrica combina información de trazas distribuidas con mediciones de calidad de servicio (latencia, errores) y evidencia cómo ciertos patrones de diseño pueden contener o amplificar degradaciones locales (Yang et al., 2024). Esta línea de trabajo resuena con los objetivos de CE: identificar configuraciones donde pequeños fallos producen degradaciones globales desproporcionadas.

En entornos Kubernetes, estas métricas se integran con indicadores nativos como:

- Disponibilidad y restart-count de pods por servicio;
- Latencia y tasa de error por endpoint obtenidos de gateways o service meshes;
- Saturación de recursos (CPU, memoria, I/O) y su correlación con la tasa de error durante experimentos.

Combinando estas señales con las métricas topológicas y de curvas de resiliencia, es posible construir tableros que cuantifiquen, para cada patrón de resiliencia implementado, cuánto reduce la probabilidad de cascadas, cuánto mejora el área bajo la curva de desempeño y cuánto contribuye a mantener el error budget dentro de límites aceptables (Murphy et al., 2016; Soldani et al., 2025; Yang et al., 2024).

2.5.6. Retos y lineamientos para seleccionar y combinar métricas

La proliferación de métricas plantea el riesgo de generar tableros complejos pero poco accionables. La literatura reciente subraya varios retos:

Equivalencia parcial entre métricas. Tang et al. (2023) muestran que métricas de resiliencia basadas en curvas a menudo no son intercambiables: dos arquitecturas pueden ordenarse de forma distinta según se utilice una métrica enfocada en tiempo de recuperación o en severidad de la caída. Esto obliga a documentar explícitamente qué dimensión de

resiliencia se prioriza.

Compatibilidad entre métricas temporales y estructurales. Yodo y Wang (2016) y Poulin y Kane (2021) sugieren que las métricas topológicas deberían utilizarse para explicar o anticipar resultados de las métricas temporales. Por ejemplo, una mala área bajo la curva pero buena redundancia puede indicar problemas de estrategia de failover más que fragilidades estructurales; a la inversa, una topología muy centralizada puede limitar las mejoras de MTTD/MTTR incluso si las prácticas de operación son maduras.

Accionabilidad en pipelines de CI/CD y observabilidad. Krasnovsky (2025) argumenta que las métricas de resiliencia deben ser suficientemente ligeras para calcularse de manera continua (por ejemplo, estimando disponibilidad esperada a partir de un grafo descubierto automáticamente) y a la vez suficientemente ricas para guiar dónde inyectar caos. De forma complementaria, Murphy et al. (2016) recomiendan definir SLO y error budgets que permitan traducir los efectos de los experimentos en decisiones de despliegue (por ejemplo, frenar lanzamientos cuando la resiliencia estimada o medida cae por debajo de ciertos umbrales).

Evaluación estadística de métricas. He et al. (2022) enfatizan la importancia de validar empíricamente los modelos de resiliencia, comparando predicciones con resultados experimentales y utilizando análisis estadístico para distinguir sesgos sistemáticos de ruido aleatorio. Este enfoque puede trasladarse a CE: métricas derivadas de simulación (como disponibilidad estimada por grafo) deben contrastarse con los resultados de inyección de fallos, idealmente utilizando pruebas de hipótesis y medidas de correlación, para evitar decisiones basadas en indicadores mal calibrados (Krasnovsky, 2025).

En síntesis, las métricas de evaluación de resiliencia en arquitecturas distribuidas se organizan en un ecosistema complementario:

- Las métricas temporales basadas en curvas capturan el comportamiento dinámico del sistema ante fallos;
- Las métricas operacionales (SLI/SLO, MTTD/MTTR, error budgets) conectan la CE con la gestión de confiabilidad y la experiencia de usuario; y

- Las métricas estructurales de grafo y las métricas específicas de microservicios revelan el papel de la arquitectura y de los patrones de resiliencia en la propagación de fallos.

La combinación deliberada de estas métricas, apoyada en modelos validados y en experimentos de caos bien diseñados, permite pasar de una visión reactiva de resiliencia (“medir incidentes pasados”) a una visión proactiva en la que la organización evalúa continuamente su postura de resiliencia y guía la evolución arquitectónica hacia configuraciones más antifrágiles.

Sobre la base de estas consideraciones teóricas, la sección final del marco integra de forma

aplicada herramientas, consideraciones de seguridad y beneficios organizacionales, conectando la teoría con la práctica.

2.6. Síntesis aplicada

2.6.1. Síntesis de herramientas y frameworks de implementación

La implementación práctica de la Ingeniería del Caos requiere el uso de herramientas especializadas que permitan inyectar fallos de manera controlada, segura y repetible. Una de las primeras y más influyentes fue Chaos Monkey, desarrollada por Netflix, cuyo propósito es terminar aleatoriamente instancias de servicios en producción para validar la capacidad de autorrecuperación de la infraestructura (Ajibola, 2025). Este enfoque pionero dio origen a un ecosistema de soluciones más sofisticadas adaptadas a entornos cloud-native y Kubernetes.

En este sentido, LitmusChaos ha surgido como una plataforma integral de experimentación, diseñada específicamente para Kubernetes. Permite ejecutar experimentos predefinidos, así como personalizar escenarios de fallo que abarcan desde interrupciones de red hasta consumo extremo de CPU y memoria (Mailewa, Akuthota, & Mohottalalage, 2025). Por su parte, Chaos Mesh, desarrollado por PingCAP, se integra de manera nativa en Kubernetes y ofrece una interfaz visual que facilita la gestión, seguimiento y automatización de experimentos de caos (Chen, Goudarzi, & Toosi, 2025).

En paralelo, los frameworks de resiliencia complementan estas herramientas al proporcionar patrones arquitectónicos que fortalecen la tolerancia a fallos. Entre ellos se encuentra Resilience4j, una librería ligera para Java que implementa mecanismos como circuit breakers, bulkheads, retries y rate limiting (Thompson, Johnson, Smith, & Adelusi, 2022). Asimismo, los service mesh como Istio permiten aplicar dichos patrones de manera transparente a nivel de infraestructura, sin necesidad de modificar el código de las aplicaciones (Kesim, 2019). En el ecosistema de Spring, Spring Cloud Circuit Breaker ofrece una abstracción uniforme sobre diversas implementaciones de circuit breakers, facilitando la integración en aplicaciones basadas en microservicios.

Chaos Mesh se distingue por su integración nativa con Kubernetes y por ofrecer una interfaz visual que simplifica la orquestación y seguimiento de experimentos de caos (Chen et al., 2025). En contraste, LitmusChaos dispone de una amplia biblioteca de experimentos predefinidos y mejor soporte para automatización en pipelines CI/CD, lo que lo convierte en una opción práctica para equipos que buscan incorporar validación de resiliencia en flujos de DevOps (Mandal et al., 2020). En términos de adopción, LitmusChaos es más accesible para principiantes, mientras que Chaos Mesh resulta atractivo para implementaciones avanzadas y personalizadas en entornos de misión crítica (Mao et al., 2021).

Si bien las herramientas de CE permiten simular fallos significativos en entornos cloud-native, persisten limitaciones en escenarios complejos. Entre ellas destacan la dificultad de

modelar fallos a gran escala en redes híbridas y multinube, la falta de estandarización en métricas de resiliencia y la limitada integración con plataformas avanzadas de observabilidad y trazabilidad (Ajibola, 2025; Gunasekaran et al., 2021). Estas barreras evidencian la necesidad de avanzar hacia frameworks más unificados que permitan integrar CE de manera continua en arquitecturas Kubernetes y en prácticas de Site Reliability Engineering (SRE).

2.6.2. Aplicación en microservicios y consideraciones de seguridad

La adopción de Chaos Engineering presenta múltiples barreras en escenarios productivos. En primer lugar, los costos de infraestructura adicional representan un reto, dado que ejecutar experimentos de caos en entornos que simulan cargas reales puede duplicar el consumo de recursos y generar sobrecosto operativo (Tamura, Yamamoto, & Fukuda, 2019). En segundo lugar, la complejidad técnica para diseñar experimentos realistas y reproducibles limita su adopción en organizaciones con baja madurez en prácticas de observabilidad y automatización (Gunasekaran, Yusuf, Adeleye, & Papadopoulos, 2021). Finalmente, la resistencia cultural constituye un obstáculo frecuente: los equipos directivos suelen percibir CE como un riesgo innecesario, en lugar de una estrategia preventiva para mejorar la resiliencia (Sharma & Joshi, 2022).

Superar esta barrera requiere procesos de comunicación efectivos, así como la integración de prácticas de gestión del cambio que generen confianza en las personas stakeholders. No obstante, la ejecución de experimentos de caos implica riesgos que deben ser gestionados adecuadamente. Una estrategia responsable requiere:

- Definición clara de los límites del experimento para evitar impactos no deseados;
- Mecanismos de rollback automático que garanticen la recuperación del sistema si el experimento excede los umbrales de riesgo;
- Procesos de aprobación y supervisión, alineados con las políticas de seguridad de la organización; y
- Monitoreo en tiempo real y documentación exhaustiva de resultados, lo cual facilita la retroalimentación y la mejora continua (Kesim, 2019).

Además, la comunicación efectiva con las partes interesadas es crucial para mantener la confianza en el proceso. Tal como señalan Gremlin (s. f.) y Ajibola (2025), la integración de CE como un componente nativo de la arquitectura representa una evolución hacia sistemas auto-validados, capaces de detectar y corregir vulnerabilidades de resiliencia de manera proactiva. Esto permite la construcción de infraestructuras más confiables y operativamente robustas en entornos de producción modernos.

La integración de CE en pipelines de entrega continua (Continuous Delivery/Continuous Integration, CI/CD) transforma la validación de resiliencia en un proceso recurrente y automatizado. Esta práctica convierte cada despliegue en una oportunidad para someter el sistema a fallos controlados, garantizando que la resiliencia no sea solo un atributo de

diseño, sino un aspecto verificable y continuo del ciclo de vida del software (Mao, Zhang, & Chen, 2021).

Estudios recientes señalan que la incorporación de experimentos de caos en etapas tempranas de CI/CD mejora la calidad de los sistemas distribuidos al detectar fallos antes de alcanzar la producción, reduciendo el Mean Time To Recovery y los costos de incidentes críticos (Ponce, Guerrero, & Casalicchio, 2022). Asimismo, su combinación con prácticas de SRE permite alinear los objetivos de resiliencia con acuerdos de nivel de servicio (SLAs y SLOs), integrando métricas de fiabilidad directamente en la cadena de valor del software (Gunasekaran et al., 2021).

2.6.3. Beneficios de la Ingeniería del Caos

El uso de la Ingeniería del Caos proporciona a las organizaciones distintos beneficios clave, que sintetizan las motivaciones técnicas y de negocio expuestas a lo largo del marco teórico:

- Mejor servicio al cliente. Las personas usuarias tienen altas expectativas sobre la disponibilidad de los servicios que adquieren. Probar los sistemas e identificar soluciones reduce el riesgo de que un sistema esté inactivo durante un periodo prolongado.
- Seguridad de datos. La CE ayuda a identificar problemas que pueden explotarse, de modo que las organizaciones puedan implementar parches y correcciones de errores antes de que se materialicen incidentes.
- Mayor escalabilidad. Los experimentos de CE permiten observar cómo un sistema asigna recursos y gestiona cargas, lo que facilita ajustar políticas de autoescalado y capacity planning.
- Tiempo de inactividad minimizado. Las organizaciones que adoptan CE cuentan con planes de acción específicos para numerosos incidentes, lo que reduce MTTD y MTTR.
- Desarrollo futuro de software más informado. Las organizaciones pueden abordar la codificación de nuevo software y soluciones de forma más inteligente al conocer cómo el sistema gestiona los problemas, integrando resiliencia desde el diseño.

En conjunto, esta síntesis aplicada cierra el marco teórico: los fundamentos, aplicaciones en Kubernetes, patrones de resiliencia, herramientas y métricas convergen en una visión de la Ingeniería del Caos como disciplina integral. A partir de aquí, los capítulos posteriores pueden apoyarse en este andamiaje conceptual para proponer y evaluar un marco de CE que combine inyección de fallos, simulación basada en grafos y patrones de resiliencia como elementos centrales de diseño.

3. Estado del arte

3.1 Orígenes y evolución de la Ingeniería del Caos

La Ingeniería del Caos (Chaos Engineering, CE) emerge a finales de la década de 2000 como respuesta práctica al reto de garantizar la resiliencia en sistemas distribuidos que operan a gran escala, en particular en empresas de streaming y servicios web que ya dependían de arquitecturas altamente distribuidas sobre infraestructura en la nube.
En este contexto, Netflix se convirtió en el caso emblemático: el crecimiento exponencial de su base de personas usuarias y la migración hacia una arquitectura basada en microservicios hicieron evidente que las técnicas de prueba tradicionales (unitarias, de integración y pruebas end-to-end en entornos de staging) eran insuficientes para capturar los modos de fallo que solo se manifestaban bajo condiciones de carga y complejidad reales en producción (Basiri et al., 2016).

A partir de estas necesidades, el equipo de tráfico y caos de Netflix formalizó la idea de introducir fallos de manera controlada en producción para observar el comportamiento del sistema y reforzar su resiliencia, dando nombre a esta práctica como “Chaos Engineering” (Basiri et al., 2016).

El hito fundacional más citado es el desarrollo del Simian Army, un conjunto de herramientas diseñadas para introducir diferentes tipos de perturbaciones en la infraestructura productiva. Dentro de este conjunto, Chaos Monkey se volvió la herramienta más conocida: su función era “matar” instancias de manera aleatoria en los clústeres de producción, obligando a que la plataforma estuviera preparada para perder máquinas sin interrumpir el servicio (Basiri et al., 2016). Este enfoque rompía con la intuición tradicional de proteger la producción como un entorno “intocable” y proponía, en cambio, que la mejor forma de ganar confianza en un sistema era someterlo sistemáticamente a condiciones adversas, pero con diseño y control. Esta lógica inauguró un cambio cultural que terminaría permeando a otras organizaciones: fallar de forma deliberada para aprender, en lugar de limitarse a reaccionar ante incidentes reales.

A partir de esta experiencia pionera, la Ingeniería del Caos fue refinándose conceptualmente y se convirtió progresivamente en una disciplina con principios y ciclos bien definidos. Basiri et al. (2016) describen cuatro ideas clave: definir un “estado estable” observable del sistema, formular hipótesis sobre cómo se comportará ese estado bajo ciertas perturbaciones, introducir fallos controlados que representen condiciones realistas, y observar los resultados para aprender y ajustar tanto la arquitectura como las prácticas operativas. Estos principios fueron ampliados y divulgados posteriormente en el libro *Chaos Engineering: System Resiliency in Practice*, donde Rosenthal y Jones (2020) sistematizan la experiencia de Netflix y otras empresas para proponer CE como una metodología general de experimentación sobre sistemas sociotécnicos complejos, más allá de un conjunto de scripts o herramientas aisladas (Rosenthal & Jones, 2020).

La evolución temprana de la CE estuvo íntimamente ligada al auge de DevOps y del Site Reliability Engineering (SRE). En la medida en que las organizaciones adoptaron ciclos de entrega continua, infraestructura como código y despliegues automatizados, se volvió factible incorporar experimentos de fallo como una práctica recurrente y no solo como actividades extraordinarias. El artículo de Basiri et al. (2016) ya situaba explícitamente CE en el contexto de la cultura de ingeniería de confiabilidad de Netflix, mientras que trabajos posteriores muestran cómo CE se integra en pipelines de CI/CD para validar hipótesis de resiliencia en cada release, alineando los experimentos con objetivos de nivel de servicio (SLO) y con las responsabilidades de equipos SRE (Naqvi et al., 2022; Akgül & Güvez, 2024).

En paralelo, la comunidad comenzó a extender el alcance de CE hacia dominios específicos y a desarrollar marcos de referencia conceptuales más amplios. Rosenthal y Jones (2020) proponen entender la CE como una manera de “razonar experimentalmente” sobre sistemas complejos, inspirada tanto en la ingeniería de confiabilidad como en la teoría de sistemas y la resiliencia organizacional. Su propuesta enfatiza que el objetivo no es “romper por romper”, sino descubrir relaciones causales entre perturbaciones, fallos y mecanismos de mitigación, para rediseñar tanto la arquitectura como los procesos humanos que la operan (Rosenthal & Jones, 2020). Este giro conceptual marca el paso de ver CE como un simple conjunto de pruebas de estrés a comprenderla como un ciclo de aprendizaje continuo en organizaciones que operan sistemas críticos.

A partir de 2016, la expansión de la Ingeniería del Caos se ve impulsada por dos dinámicas convergentes: por un lado, la consolidación de arquitecturas cloud-native basadas en contenedores, microservicios y orquestadores como Kubernetes; por otro, el surgimiento de una primera generación de herramientas de código abierto y servicios gestionados que democratizan la práctica. El estudio seminal de Basiri et al. (2016) ya anticipaba que la creciente complejidad de sistemas distribuidos en la nube exigiría nuevas formas de prueba; investigaciones posteriores confirman que, en este contexto, CE se vuelve especialmente valiosa para descubrir fallos derivados de dependencias implícitas, timeouts encadenados y estados distribuidos que no son evidentes en entornos de prueba aislados (Al-Said Ahmad et al., 2024).

La evidencia empírica más reciente sobre la adopción de CE en “la naturaleza” proviene de estudios de minería de repositorios y revisiones de literatura multivocal. Owotogbe et al. (2025) analizan 971 repositorios de GitHub que utilizan diez herramientas de CE populares y muestran que Toxiproxy y Chaos Mesh son las que concentran mayor uso sostenido desde 2016, reflejando la creciente adopción de CE en entornos cloud-native. Sus resultados indican que el lanzamiento de nuevas herramientas alcanzó un pico en 2018 y luego se desaceleró, tendencia que interpretan como una transición de una fase exploratoria a una fase de madurez en la que el énfasis se desplaza desde “crear herramientas nuevas” hacia integrarlas y refinarlas en pipelines y plataformas existentes (Owotogbe et al., 2025).

Esa misma línea de investigación revela que la mayor parte de los repositorios analizados corresponde a proyectos de desarrollo de software (≈58 %), mientras que el resto se reparte entre docencia, aprendizaje y experimentación académica (Owotogbe et al., 2025). Esta distribución respalda la lectura de CE como una práctica predominantemente industrial, orientada a mejorar la confiabilidad de sistemas reales en producción y a apoyar decisiones de arquitectura en organizaciones que operan a escala. Además, el análisis de escenarios de fallo muestra que predominan las inyecciones de fallos de red (latencias, pérdidas, particiones) y la terminación de instancias/pods, mientras que los fallos a nivel de aplicación están relativamente subrepresentados, lo que sugiere oportunidades para extender CE hacia capas funcionales más cercanas al negocio (Owotogbe et al., 2025).

Desde la perspectiva de síntesis de conocimiento, la disciplina ha entrado en una fase en la que comienzan a aparecer revisiones sistemáticas que integran tanto literatura académica como fuentes industriales. Owotogbe et al. (2024) realizan una Multivocal Literature Review (MLR) de 96 fuentes (artículos científicos y literatura gris) publicadas entre 2016 y 2024, con el objetivo de derivar una definición unificada de CE, identificar capacidades clave y proponer una taxonomía de plataformas y herramientas. El estudio muestra que, a pesar de la diversidad de contextos, existe consenso en entender CE como “la disciplina de llevar a cabo experimentos sobre un sistema distribuido para construir confianza en su capacidad de resistir condiciones turbulentas en producción” (Owotogbe et al., 2024). Asimismo, identifica como capacidades centrales la automatización de experimentos, la integración con observabilidad y la gestión explícita del riesgo asociado al blast radius.

En la misma línea, Fossati et al. (2025) realizan una revisión sistemática de literatura gris específicamente orientada a prácticas industriales de CE, analizando 50 fuentes publicadas entre 2019 y comienzos de 2024. A partir de este corpus, derivan un marco de diez conceptos (C1–C10) que extiende los principios fundacionales de Netflix hacia prácticas contemporáneas en pipelines DevOps: definición de estado estable, formulación y validación de hipótesis, selección de eventos realistas, ejecución controlada, automatización, contención del impacto, incremento gradual de complejidad, medición-aprendizaje-mejora y socialización de resultados, entre otros (Fossati et al., 2025). Sus hallazgos confirman que la evolución reciente de CE enfatiza cada vez más la necesidad de minimizar riesgo mediante abort switches, ventanas de experimentación acotadas y rollbacks automatizados, al tiempo que se institucionalizan mecanismos de aprendizaje como game days y postmortems sin culpabilización.

El desarrollo de la Ingeniería del Caos también ha comenzado a entrelazarse con otras áreas de la ingeniería de software y de sistemas auto-adaptativos. Naqvi et al. (2022) proponen utilizar principios de CE para evaluar sistemas auto-sanables y auto-adaptativos mediante el marco CHESS, que introduce perturbaciones sistemáticas para observar la capacidad de estos sistemas de detectar, diagnosticar y corregir fallos de manera autónoma. Este enfoque desplaza el foco desde “probar la resiliencia de la infraestructura” hacia “probar la eficacia de los mecanismos de auto-curación”, evidenciando cómo CE puede evolucionar de una técnica de prueba de infraestructura a un método general para evaluar

comportamientos adaptativos en sistemas complejos (Naqvi et al., 2022).

Desde la perspectiva de arquitecturas cloud y microservicios, diversos trabajos muestran que la adopción de CE se asocia a un cambio en la forma de razonar sobre disponibilidad y rendimiento. El estudio empírico de Al-Said Ahmad et al. (2024) aplica CE sobre una aplicación cloud-native bajo diferentes cargas de trabajo, analizando el impacto de diversos escenarios de fallo en las métricas de rendimiento y disponibilidad. Sus resultados demuestran que la combinación de inyección de fallos y variación de la carga revela modos de degradación que no son visibles en pruebas estáticas, y subrayan la importancia de alinear los experimentos de CE con las condiciones reales de uso de las aplicaciones (Al-Said Ahmad et al., 2024). Estudios como este refuerzan la idea de que la evolución de CE está vinculada no solo a "qué fallos se inyectan", sino también a "en qué contexto operativo" se evalúa la resiliencia.

En paralelo, se observa una expansión temática hacia variantes especializadas como la Security Chaos Engineering (SCE). Aunque la SCE se centra explícitamente en la robustez frente a condiciones maliciosas, comparte con la CE clásica la idea de introducir perturbaciones controladas (en este caso, de seguridad) para evaluar la capacidad de los sistemas de resistir ataques, detectar intrusiones y mantener sus propiedades de confidencialidad, integridad y disponibilidad (Jolak, 2025). Esta convergencia muestra cómo el paradigma de "experimentar en producción bajo condiciones controladas" se consolida como una estrategia transversal para abordar tanto la resiliencia operativa como la seguridad.

Otro rasgo distintivo de la evolución de CE es el movimiento desde herramientas monolíticas hacia ecosistemas más ricos y heterogéneos de plataformas y servicios. Akgül y Güvez (2024) sistematizan una serie de herramientas clave (LitmusChaos, AWS Fault Injection Simulator, Azure Chaos Studio, Steadybit, Chaos Mesh, Chaos Toolkit, Toxiproxy, entre otras) y destacan cómo grandes proveedores de nube han incorporado servicios específicos de inyección de fallos, lo que facilita la adopción de CE incluso en organizaciones que no cuentan con equipos SRE altamente especializados. Este panorama confirma que CE ha dejado de ser una práctica exclusiva de grandes empresas tecnológicas para convertirse en un componente accesible de las estrategias de prueba y observabilidad de organizaciones medianas que operan sobre nubes públicas (Akgül & Güvez, 2024).

Finalmente, estudios recientes apuntan a una "segunda generación" de Ingeniería del Caos, en la que la disciplina se expande hacia dominios como arquitecturas orientadas a eventos, sistemas multi-nube y, más recientemente, ecosistemas de agentes basados en modelos de lenguaje de gran tamaño (LLM-MAS). Trabajos emergentes exploran cómo aplicar CE en microservicios event-driven para cuantificar la efectividad de los mecanismos de resiliencia bajo patrones de comunicación asíncronos, y discuten la necesidad de nuevas métricas y escenarios de fallo adaptados a colas de mensajes, buses de eventos y flujos de datos altamente dinámicos (Ahmad & Al-Qora'n, 2025). Paralelamente, la discusión sobre antifragilidad y resiliencia "by design" en entornos cloud-native recupera el espíritu

original de CE como un marco para aprender de la perturbación y rediseñar sistemas que no solo resistan, sino que mejoren como resultado de los experimentos (Sahoo, 2025).

En síntesis, la evolución histórica de la Ingeniería del Caos puede leerse como una transición desde un conjunto de prácticas internas en una empresa pionera (Netflix) hacia una disciplina con fundamentos conceptuales, evidencia empírica y ecosistemas de herramientas consolidados. La primera etapa estuvo marcada por la creación de Chaos Monkey y el Simian Army, así como por la formulación de principios básicos centrados en el estado estable y la experimentación en producción (Basiri et al., 2016).

Una segunda etapa amplió CE en el marco de DevOps/SRE, integrándola en pipelines de entrega continua y articulándola con prácticas de observabilidad y gestión de incidentes (Rosenthal & Jones, 2020; Akgül & Güvez, 2024). Una tercera etapa, reflejada en revisiones multivocales y estudios de minería de repositorios, documenta su adopción masiva en el ecosistema cloud-native y su consolidación como práctica industrial madura (Owotogbe et al., 2024, 2025; Fossati et al., 2025). Etapas más recientes muestran la expansión de CE hacia sistemas auto-adaptativos, seguridad, arquitecturas event-driven y, de forma incipiente, sistemas inteligentes basados en IA, confirmando que la Ingeniería del Caos se ha transformado en un marco general de experimentación para aprender sobre resiliencia en sistemas complejos en producción (Naqvi et al., 2022; Al-Said Ahmad et al., 2024).

3.2 Aplicaciones actuales en sistemas distribuidos y Kubernetes

En los sistemas distribuidos contemporáneos, la Ingeniería del Caos se ha consolidado como una práctica operativa destinada a validar, bajo condiciones controladas, la capacidad de recuperación de servicios que corren sobre infraestructuras heterogéneas (nube pública, privada e híbrida), integrando microservicios, colas de mensajería, bases de datos distribuidas y componentes legacy. Desde la formulación de los principios fundacionales de Chaos Engineering, que proponen experimentar de forma sistemática sobre sistemas vivos para ganar confianza en su capacidad de soportar condiciones turbulentas (Basiri et al., 2016), la disciplina se ha desplazado desde ejercicios puntuales en producción hacia ciclos continuos de experimentación integrados al flujo DevOps/SRE.

En este contexto, las aplicaciones actuales de Chaos Engineering (CE) en sistemas distribuidos pueden describirse en tres planos interrelacionados: (a) validación de resiliencia a nivel de infraestructura (fallos de nodos, redes, almacenamiento); (b) evaluación de comportamientos emergentes a nivel de aplicación (timeouts, reintentos, degradación de servicios dependientes); y (c) alineación con objetivos de confiabilidad expresados como SLO/SLI, donde métricas como disponibilidad percibida, tasa de errores y latencia se utilizan como “estado estable” observable para juzgar el impacto de los experimentos (Basiri et al., 2016; Al-Said Ahmad et al., 2024).

3.2.1 Chaos Engineering en sistemas distribuidos contemporáneos

En sistemas distribuidos no necesariamente orquestados por Kubernetes —por ejemplo, arquitecturas basadas en máquinas virtuales o contenedores gestionados manualmente— CE se ha aplicado para someter a estrés rutas críticas de negocio mediante fallos de infraestructura y de software. Basiri et al. (2016) describen ejercicios que van desde la terminación aleatoria de instancias (Chaos Monkey) hasta simulaciones de pérdida de regiones completas (Chaos Kong), con el fin de forzar a las organizaciones a diseñar para fallar, incorporando redundancia, balanceo y degradación elegante de funcionalidades.

En paralelo, estudios empíricos han demostrado que la simple terminación de nodos o servicios no es suficiente para capturar los modos de fallo relevantes en sistemas complejos. Simonsson et al. (2021) introducen un enfoque de observabilidad y Chaos Engineering a nivel de llamadas al sistema (system calls) para aplicaciones contenedorizadas, mostrando que inyectar fallos en este nivel permite descubrir vulnerabilidades que no se observan con fallos “macro” (por ejemplo, matar contenedores completos), al tiempo que se apalancan métricas detalladas procedentes de trazas y logs de bajo nivel.

Zhang et al. (2022) profundizan esta línea al maximizar el realismo de la inyección de errores basados en system calls, argumentando que el comportamiento de un sistema distribuido frente a fallos depende tanto del tipo de error como del contexto de ejecución (estado interno, patrones de acceso, carga concurrente). En su trabajo, muestran que la calibración cuidadosa de los experimentos —por ejemplo, diferenciando entre timeouts, errores de E/S y excepciones— produce señales de resiliencia más alineadas con incidentes reales y mejora la utilidad de métricas como el tiempo medio de detección (MTTD) y el tiempo medio de recuperación (MTTR) (Zhang et al., 2022).

La disciplina también se ha extendido a dominios específicos de sistemas distribuidos, como entornos de TI empresariales donde la configuración de servicios es compleja y altamente acoplada. Poltronieri et al. (2022) presentan un enfoque de Chaos Engineering orientado a mejorar la resiliencia de configuraciones de servicios TI, introduciendo la herramienta ChaosTwin para evaluar automáticamente el impacto de distintos escenarios de fallo sobre configuraciones alternativas. Su enfoque demuestra que CE puede ayudar a comparar configuraciones y a seleccionar aquellas que minimizan la degradación del servicio bajo perturbaciones controladas (Poltronieri et al., 2022).

En cuanto a sistemas cloud-native más recientes, Al-Said Ahmad et al. (2024) aplican CE a dos aplicaciones nativas de nube (una serverless y una basada en microservicios contenedorizados) desplegadas sobre Amazon Web Services. Inyectan retardos en servicios dependientes bajo distintas cargas de usuarios y miden cómo se deterioran throughput y latencia, evidenciando que pequeños incrementos controlados en la latencia de componentes críticos pueden desencadenar colas, timeouts y errores de disponibilidad

no triviales de predecir únicamente mediante análisis estático de arquitectura.

3.2.2 Aplicaciones específicas en contenedores y microservicios

En arquitecturas basadas en microservicios, el foco de CE se desplaza hacia la interacción entre servicios y sus dependencias remotas: bases de datos distribuidas, colas de mensajería, caches y APIs externas. Simonsson et al. (2021) muestran que, incluso en entornos Docker relativamente acotados, la combinación de observabilidad fina y perturbaciones a nivel de system calls permite identificar puntos de contención (por ejemplo, bloqueos en acceso a disco o redes saturadas) que pueden amplificarse en topologías de microservicios a gran escala.

Además, trabajos como el de Frank et al. (2022) proponen el uso de simuladores específicos, como MiSim, para evaluar la resiliencia de arquitecturas basadas en microservicios antes de desplegar campañas de CE en entornos productivos o de staging. MiSim modela explícitamente patrones de comunicación, tiempos de servicio y dependencias, permitiendo explorar el espacio de fallos posibles (por ejemplo, degradaciones de latencia en servicios intermedios o errores intermitentes) y observar cómo cambian métricas de resiliencia a medida que se ajustan parámetros como la replicación o los tiempos de reintento (Frank et al., 2022).

Esta combinación de simulación y experimentación viva ha dado lugar a prácticas híbridas donde los modelos se utilizan para reducir el espacio de búsqueda de experimentos costosos. Krasnovsky (2025) propone un enfoque de descubrimiento de modelos y simulación de grafos para sistemas de microservicios: a partir de trazas distribuidas y telemetría de malla de servicios, se infiere automáticamente un grafo de dependencias bloqueantes con conteo de réplicas y se ejecutan simulaciones Monte Carlo de fallos fail-stop. Los resultados muestran que, especialmente cuando existe replicación, las curvas de disponibilidad estimadas por el modelo se correlacionan estrechamente con experimentos de inyección de fallos reales sobre el benchmark DeathStarBench, lo que habilita el uso de este modelo como “filtro” previo para seleccionar los escenarios de CE más informativos (Krasnovsky, 2025).

3.2.3 Kubernetes como plataforma de ejecución de Chaos Engineering

En el ecosistema cloud-native actual, Kubernetes se ha convertido en el entorno de facto para la orquestación de contenedores, por lo que una proporción significativa de las aplicaciones modernas de CE se centra en clusters Kubernetes. La semántica propia del orquestador, tales como la reprogramación de pods, probes de salud, políticas de autoscaling, afinidades de scheduling, introduce tanto oportunidades como nuevos modos de fallo. Por ejemplo, la terminación de un pod puede ser absorbida con relativa facilidad si existen suficientes réplicas y balanceadores, pero fallos repetidos de readiness probes o una configuración inadecuada de políticas de reintento pueden desencadenar tormentas de reinicios o cascadas de errores.

En este contexto, las campañas de CE sobre Kubernetes suelen incluir experimentos como: terminación aleatoria de pods o despliegos; simulación de pérdida o degradación de nodos; inyección de latencia y pérdida de paquetes mediante herramientas de red; estrés de CPU y memoria en contenedores críticos; y fallos a nivel de aplicación (por ejemplo, bloquear hilos, lanzar excepciones no manejadas) (Simonsson et al., 2021; Zhang et al., 2022).

Al-Said Ahmad et al. (2024) muestran que, cuando estos experimentos se combinan con carga realista y métricas cuidadosamente seleccionadas (por ejemplo, percentiles de latencia, throughput por operación de negocio, tasa de errores observada por el cliente), es posible diferenciar configuraciones que, en teoría, parecen equivalentes, pero que bajo fallo arrojan comportamientos muy distintos. En su estudio, el análisis conjunto de métricas operativas y de usuario permitió identificar umbrales de latencia a partir de los cuales emergen fallos de disponibilidad "de la nada", lo que refuerza la importancia de diseñar experimentos que exploren progresivamente el espacio de condiciones de fallo en Kubernetes (Al-Said Ahmad et al., 2024).

3.2.4 Integración con pipelines DevOps/SRE y observabilidad

Otra característica central de las aplicaciones actuales de CE en sistemas distribuidos sobre Kubernetes es su integración con pipelines de entrega continua y prácticas SRE. Basiri et al. (2016) ya apuntaban que la experimentación no puede seguir siendo manual y esporádica: si los sistemas cambian continuamente, la confianza en resultados antiguos se degrada rápidamente. Estudios recientes de prácticas industriales confirman que las organizaciones que maduran en CE tienden a automatizar la planificación y ejecución de experimentos como pasos dentro de pipelines CI/CD y a tratarlos como pruebas no funcionales recurrentes, no como ejercicios "excepcionales" (Basiri et al., 2016; Owotogbe et al., 2025).

Owotogbe et al. (2025), en su análisis de 971 repositorios relacionados con CE en GitHub, muestran que las herramientas más usadas en entornos Kubernetes —como Chaos Mesh, LitmusChaos y Toxiproxy— se emplean frecuentemente en combinación con sistemas de observabilidad existentes (Prometheus, Grafana, Jaeger, etc.), reforzando la idea de que la observabilidad no es un "extra" sino un prerrequisito para CE eficaz. El estudio también evidencia una transición desde una fase de proliferación de herramientas (pico alrededor de 2018) hacia una fase de integración y consolidación, donde el énfasis está en buenas prácticas de automatización, control del blast radius y reproducibilidad de experimentos (Owotogbe et al., 2025).

En la práctica, esta integración se manifiesta en patrones como:

- *Gate de resiliencia* en pipelines: no se promueven despliegues a entornos superiores si experimentos de caos básicos (por ejemplo, terminación de pods de servicios no críticos) no cumplen criterios de éxito predefinidos.
- *Game days automatizados*: ejercicios periódicos donde un conjunto de experimentos se ejecuta en ventanas acotadas, con tableros de observabilidad

preparados para observar el efecto sobre servicios y flujos de negocio.

- *Postmortems blameless enriquecidos con datos de CE*: los hallazgos de incidentes reales alimentan el diseño de nuevos experimentos, cerrando el ciclo de aprendizaje organizacional (Basiri et al., 2016; Owotogbe et al., 2025).

3.2.5 Simulación y modelos ligeros como complemento de la inyección de fallos

Las aplicaciones actuales de CE en Kubernetes no dependen únicamente de inyectar fallos "en vivo"; cada vez más, se apoyan en modelos y simuladores que permiten priorizar esfuerzos. Frank et al. (2022) argumentan que simuladores como MiSim ofrecen una manera sistemática de explorar configuraciones y patrones de fallo en arquitecturas de microservicios, con un costo computacional mucho menor que ejecutar campañas extensas en clusters reales. Esto habilita, por ejemplo, analizar el efecto de distintos niveles de replicación, timeouts y políticas de reintento sobre métricas de resiliencia antes de tocar entornos productivos (Frank et al., 2022).

En la misma línea, el enfoque de Krasnovsky (2025) sugiere que un modelo topológico mínimo —grafo de dependencias + cantidades de réplicas— puede capturar gran parte de la señal de resiliencia relevante para fallos fail-stop, sin necesidad de construir modelos analíticos complejos. Al descubrir automáticamente este grafo a partir de artefactos ya presentes en las organizaciones (trazas distribuidas, telemetría de malla, manifiestos de Kubernetes, configuración como código), es posible integrar el "descubrimiento de modelo" como paso de CI/CD y generar continuamente una señal de postura de resiliencia que acompaña la evolución de la topología del sistema (Krasnovsky, 2025).

Estas aplicaciones no sustituyen la inyección de fallos real, pero sí permiten: (a) acotar el espacio de experimentos más prometedores —por ejemplo, cadenas de dependencia con servicios no replicados que constituyen puntos únicos de fallo—; y (b) interpretar resultados de CE a la luz de la estructura del grafo, explicando por qué, aun con buenos tiempos de reprogramación de pods, el MTTR percibido por el usuario no mejora si las dependencias siguen siendo frágiles (Frank et al., 2022; Krasnovsky, 2025).

3.2.6 Evaluación empírica y tendencias observadas

Los trabajos empíricos recientes permiten extraer patrones transversales sobre cómo se aplica hoy CE en sistemas distribuidos y Kubernetes. En primer lugar, los estudios coinciden en que la efectividad de CE depende fuertemente de la calidad de las métricas elegidas como "estado estable". Al-Said Ahmad et al. (2024) muestran que, en aplicaciones cloud-native, métricas agregadas como el promedio de latencia pueden ocultar comportamientos críticos que solo se revelan en percentiles altos (p. ej., p95, p99), mientras que la combinación de throughput, latencia y tasa de errores facilita identificar umbrales a partir de los cuales se disparan problemas de disponibilidad.

En segundo lugar, diversos trabajos resaltan que la granularidad de la inyección de fallos debe corresponderse con el nivel de abstracción donde se toman decisiones de diseño.

Zhang et al. (2022) y Simonsson et al. (2021) evidencian que, para ciertas clases de vulnerabilidades (por ejemplo, manejo deficiente de excepciones, errores de sistema operativo), la inyección a nivel de system calls ofrece una perspectiva más precisa que la terminación de contenedores completos, y que esta información puede guiar refactorizaciones específicas de código y configuración (Zhang et al., 2022; Simonsson et al., 2021).

En tercer lugar, las revisiones de repositorios y herramientas sugieren que la comunidad se ha desplazado desde un enfoque centrado en "romper cosas" hacia marcos más disciplinados, donde la automatización, la contención del blast radius y la socialización de aprendizajes (mediante game days, playbooks y postmortems) son componentes tan importantes como las propias perturbaciones técnicas. Owotogbe et al. (2025) documentan este cambio al encontrar un núcleo estable de herramientas ampliamente adoptadas y patrones de uso que priorizan la integración con observabilidad y pipelines, más que la creación de nuevos frameworks ad-hoc.

Finalmente, las aplicaciones actuales de CE en Kubernetes comienzan a extenderse hacia escenarios avanzados, como arquitecturas serverless complejas y sistemas de múltiples dominios. Al-Said Ahmad et al. (2024) evidencian que la combinación de funciones serverless y microservicios contenedorizados introduce nuevos modos de fallo (por ejemplo, cold starts bajo carga, límites de concurrencia) que requieren experimentos de caos específicos para ser comprendidos, mientras que enfoques basados en grafos como el de Krasnovsky (2025) ofrecen herramientas conceptuales para razonar sobre estos sistemas híbridos.

En conjunto, las aplicaciones actuales de Chaos Engineering en sistemas distribuidos y Kubernetes pueden entenderse como un "sistema operativo de aprendizaje de resiliencia": los equipos definen hipótesis sobre el comportamiento deseado, diseñan campañas de experimentos apoyadas en observabilidad y modelos, ejecutan perturbaciones controladas bajo guardrails estrictos, interpretan resultados a la luz de métricas y grafos de dependencia, y traducen los hallazgos en cambios arquitectónicos, de configuración y de proceso. Este ciclo, cuando se institucionaliza, transforma CE de una práctica disruptiva puntual en un componente estructural de la ingeniería de confiabilidad en entornos cloud-native.

3.3 Herramientas de inyección de fallos

La evolución de la Ingeniería del Caos ha impulsado el desarrollo de una amplia variedad de herramientas diseñadas específicamente para la inyección sistemática de fallos en sistemas distribuidos, con especial énfasis en entornos cloud-native y Kubernetes. Estas herramientas permiten reproducir condiciones adversas que, de otro modo, serían difíciles de observar mediante pruebas tradicionales. Su objetivo es medir la resiliencia efectiva del sistema, validar mecanismos de tolerancia a fallos y generar evidencia empírica para sustentar decisiones arquitectónicas. El ecosistema actual combina soluciones de código

abierto, herramientas académicas, plataformas comerciales y frameworks extensibles que permiten crear, ejecutar y automatizar experimentos de caos (Basiri et al., 2016; Poltronieri et al., 2022).

3.3.1 Chaos Mesh: arquitectura declarativa y experimentación granular en Kubernetes

Chaos Mesh se ha consolidado como una de las herramientas más completas para la inyección de fallos en Kubernetes, gracias a su arquitectura basada en Custom Resource Definitions (CRDs). Esta aproximación declarativa permite definir experimentos como recursos nativos del cluster, lo que facilita su integración con controladores existentes, sistemas GitOps como ArgoCD y pipelines CI/CD (Zhou et al., 2022). La herramienta soporta múltiples categorías de fallos: alteraciones de red (latencia, pérdida, corrupción de paquetes), fallos de I/O, estrés de CPU y memoria, terminación de pods, fallos en contenedores específicos y perturbaciones en el kernel mediante eBPF, ofreciendo un repertorio amplio para distintos escenarios de resiliencia.

Un aporte técnico destacado de Chaos Mesh es su motor avanzado de planificación, que permite limitar el alcance del fallo mediante selectors precisos (por etiquetas, namespaces, contenedores específicos) y el uso de guardrails para evitar que un experimento afecte servicios críticos del cluster. Además, su integración con herramientas de observabilidad como Prometheus, Jaeger y Grafana permite asociar cada experimento con métricas clave, posibilitando la evaluación del impacto real sobre flujos de usuario y tiempos de respuesta (Zhou et al., 2022).

3.3.2 LitmusChaos: estandarización, automatización y madurez operativa

LitmusChaos se ha posicionado como una herramienta robusta dentro del ecosistema CNCF, con un enfoque orientado a la estandarización del proceso de experimentación. El proyecto define el concepto de **Litmus Workflows**, que permite modelar experimentos complejos combinando múltiples inyecciones de fallo y etapas de verificación. Además, su consola web ofrece visualización en tiempo real, gestión de experimentos y análisis posterior, lo que facilita su adopción por equipos DevOps y SRE (Kumar et al., 2023).

Un componente distintivo de LitmusChaos es su catálogo de experimentos certificados, los cuales incluyen pruebas específicas para Kubernetes, bases de datos distribuidas (como Cassandra y MongoDB), colas de mensajería (Kafka, RabbitMQ) y servicios cloud. Estas pruebas siguen un conjunto estandarizado de pasos, métricas y condiciones de éxito, lo que permite generar resultados reproducibles y comparables entre organizaciones (Mohan et al., 2022). Asimismo, LitmusChaos soporta integración nativa con GitHub Actions, Tekton, Jenkins y GitLab CI, facilitando su adopción en pipelines automatizados.

3.3.3 Chaos Toolkit: extensibilidad, experimentación sin vendor lock-in y filosofía declarativa

Chaos Toolkit destaca por su enfoque minimalista y extensible, basado en archivos JSON o YAML que describen de forma declarativa los experimentos. Su arquitectura modular

permite incorporar *drivers* para numerosas plataformas (AWS, Azure, Docker, Kubernetes, Istio, GCP), lo que facilita evaluar la resiliencia en sistemas híbridos o multicloud (Dehghani et al., 2021). Además, su diseño orientado a extensiones permite que los equipos desarrollen sus propios controladores para sistemas internos, lo que lo convierte en una herramienta flexible para organizaciones con infraestructura personalizada.

Otra característica relevante es su énfasis en la verificabilidad: cada experimento debe incluir hipótesis y validadores que aseguren que el estado estable se mantenga durante la perturbación. Esto convierte a Chaos Toolkit en una herramienta idónea para entornos con alta exigencia de trazabilidad, auditoría y cumplimiento (Dehghani et al., 2021).

3.3.4 Herramientas especializadas en red: Toxiproxy y Chaos Mesh Network Attacks

Dado que la degradación de red es uno de los modos de fallo más frecuentes en sistemas distribuidos, han surgido herramientas específicas para simular condiciones adversas en la comunicación entre servicios. Toxiproxy, desarrollada originalmente por Shopify, permite introducir fallos como latencia, pérdida de paquetes, limitación de ancho de banda y desconexiones completas entre servicios, todo de forma programática (Simonsson et al., 2021).

A diferencia de herramientas genéricas, Toxiproxy se centra exclusivamente en fallos de red, lo que permite simular con gran precisión degradaciones intermitentes o fluctuaciones de RTT, muy comunes en sistemas distribuidos geográficamente dispersos. Su integración con entornos de pruebas automatizadas lo convierte en un componente útil tanto para CE como para pruebas de regresión.

Chaos Mesh también ofrece un módulo de inyección de fallos de red construido sobre tecnologías como tc y eBPF, lo que permite simular condiciones con mayor fidelidad a nivel de kernel y con menor sobrecarga de usuario (Zhou et al., 2022).

3.3.5 ChaosOrca y herramientas académicas orientadas a microservicios

Para entornos de microservicios que aún no migran a Kubernetes, o que operan bajo orchestradores alternativos, ChaosOrca se ha convertido en una herramienta relevante. Simonsson et al. (2021) presentan ChaosOrca como una herramienta académica diseñada para inyectar fallos a nivel de llamadas entre servicios contenedorizados. A diferencia de herramientas de alto nivel, ChaosOrca permite manipular fallos directamente sobre system calls, ofreciendo una granularidad fina para detectar defectos que no emergen cuando se termina un contenedor completo.

Estudios han demostrado que los fallos inyectados a nivel de system call revelan modos de fallo más realistas: errores de lectura/escritura, deadlocks, timeouts no manejados y comportamientos inesperados en librerías internas. Esto la convierte en una herramienta valiosa para investigaciones sobre resiliencia y para validación profunda en sistemas críticos (Simonsson et al., 2021).

3.3.6 Soluciones comerciales: Gremlin, Steadybit, Harness Chaos Engineering

El ecosistema comercial ha crecido significativamente, ofreciendo plataformas con funcionalidades avanzadas para organizaciones que requieren control de alcance, auditoría, reportes ejecutivos y cumplimiento normativo.

Gremlin es una de las plataformas más influyentes, con soporte para inyección de fallos en red, CPU, memoria, disco y shutdown de hosts. Su enfoque empresarial incorpora características como abort switches, blast radius control, métricas integradas, reportes de impacto y *schedules* automáticos, además de integración nativa con plataformas como Datadog, New Relic, Splunk y AWS (Gremlin Inc., 2023).

Steadybit, una plataforma emergente, introduce inyección de fallos sobre arquitecturas complejas como service meshes (Istio, Linkerd), entornos serverless (AWS Lambda) y Kubernetes avanzado (pods, nodos, contenedores, control plane). Su enfoque en degradación controlada de SLIs la ha vuelto popular en organizaciones con madurez SRE (Steadybit GmbH, 2023).

Harness Chaos Engineering, integrado al ecosistema de CI/CD de Harness, ofrece experimentación automatizada como parte de pipelines y permite evaluar resiliencia antes de cada despliegue (Harness.io, 2024).

3.3.7 Integración con simulación y modelos basados en grafos

Una tendencia reciente es la integración entre herramientas de caos y frameworks de simulación. Krasnovsky (2025) propone un modelo de descubrimiento de topología y simulación de grafos que permite predecir la degradación de disponibilidad bajo fallos fail-stop antes de ejecutar experimentos reales. Estas capacidades se están integrando progresivamente en herramientas como Chaos Mesh y LitmusChaos, mediante módulos experimentales de análisis de dependencias y validación previa.

3.3.8 Comparación crítica del ecosistema de herramientas

El análisis comparado revela que ninguna herramienta domina por completo el espacio de aplicación; en cambio, cada una atiende necesidades específicas:

- Chaos Mesh → alta integración con Kubernetes; experimentos complejos; soporte eBPF.
- LitmusChaos → estandarización, workflows, enfoque reproducible.
- Chaos Toolkit → extensibilidad, simplicidad, multicloud.
- Toxiproxy → precisión en fallos de red.
- ChaosOrca → investigación, granularidad fina a nivel de system calls.
- Gremlin/Steadybit/Harness → necesidades empresariales avanzadas.

Esta diversidad refleja la madurez del campo, así como la necesidad de combinar herramientas según las características del sistema bajo análisis y los riesgos aceptables.

Tabla 1

Comparativa de herramientas por tipo de entorno, métricas de evaluación, ventajas y desventajas

Herramienta / enfoque	Tipo y entorno principal	Métricas de evaluación típicas	Ventajas principales	Limitaciones / desventajas
Chaos Mesh	Herramienta *open source* para inyección de fallos en Kubernetes (CRDs, eBPF, tc).	Latencia (p95/p99), tasa de errores, disponibilidad, MTTR, cumplimiento de SLO.	Alta integración con Kubernetes; definición declarativa de experimentos; gran variedad de fallos (red, CPU, memoria, pods, kernel); buen soporte de observabilidad.	Foco casi exclusivo en Kubernetes; curva de aprendizaje para diseñar experimentos complejos y configurar *guardrails*.
LitmusChaos	Plataforma CNCF para Kubernetes, con *workflows* y catálogo de experimentos certificados.	Éxito/fracaso de experimentos, tasa de errores, latencia, disponibilidad, estabilidad bajo fallos repetidos.	Catálogo estandarizado de experimentos; interfaz gráfica; integración nativa con CI/CD; resultados reproducibles y comparables.	Menos flexible para escenarios muy específicos; depende del ecosistema Kubernetes; configuración inicial puede ser pesada.
Chaos Toolkit	Framework declarativo extensible (JSON/YAML) para múltiples nubes y plataformas (K8s, Docker, AWS, Azure…).	Disponibilidad, tasa de errores, latencia, MTTD/MTTR según instrumentación externa.	Simplicidad y filosofía *vendor-neutral*; muy extensible mediante *drivers*; adecuado para entornos híbridos/multicloud.	Muchas capacidades dependen de extensiones desarrolladas por el usuario; menor foco específico en Kubernetes que Chaos Mesh / Litmus.
Toxiproxy	Proxy de fallos de red para servicios distribuidos.	Latencia, pérdida de paquetes, throughput, tasa de errores en llamadas remotas.	Gran precisión en la simulación de problemas de red (latencia, pérdida, cortes intermitentes); útil en pruebas automatizadas y CI.	Se limita a fallos de red; no cubre aspectos de CPU, memoria ni comportamiento interno de aplicaciones.
ChaosOrca	Herramienta académica para inyección de fallos a nivel de *system calls* en contenedores Docker.	Uso de CPU/memoria, latencia de peticiones, tasa de errores, tiempos de degradación y recuperación.	Gran granularidad (fallos realistas a nivel de OS); revela vulnerabilidades que no aparecen al "matar contenedores" completos.	Enfoque más experimental; complejidad técnica alta; menos adoptada en entornos productivos que herramientas CNCF.
Gremlin / Steadybit / Harness CE	Plataformas comerciales empresariales para CE (host, contenedores, mallas de servicio, serverless).	Disponibilidad, SLO cumplidos, tasas de error, latencias, impacto por experimento, indicadores ejecutivos.	Control avanzado de *blast radius* y abort switches; reportes ejecutivos; fuerte integración con APM/observabilidad; adecuada para organizaciones grandes.	Coste de licencia; caja negra parcial; menos adecuadas para prototipos de investigación académica con recursos limitados.

MiSim y modelos de grafos (Krasnovsky)	Simuladores y modelos ligeros de arquitecturas de microservicios basados en grafos y Monte Carlo.	Disponibilidad estimada, probabilidad de fallo, distribución de MTTR/MTTD simulados, impacto en throughput/latencia.	Permiten explorar muchas configuraciones “en barato”; ayudan a priorizar qué escenarios de caos ejecutar en clusters reales.	No sustituyen la inyección real de fallos; requieren ajustar el modelo para que refleje bien la topología y las cargas reales.
CHESS (Naqvi, Malik)	Framework de evaluación de sistemas *self-healing* con inyección de fallos funcionales e infraestructurales.	MTTD y MTTR específicos del lazo de auto-curación, porcentaje de experimentos con recuperación exitosa, degradación de QoS.	Conecta CE con evaluación de sistemas auto-adaptativos; aporta métricas claras de eficacia de la auto-recuperación.	Enfoque más centrado en sistemas *self-healing* que en aplicaciones de negocio tradicionales; requiere instrumentación y controlador adaptativo.
K8-Scalar y *workbenches* de Kubernetes	Bancos de pruebas para evaluar rendimiento/escala y resiliencia en clústeres Kubernetes.	Throughput, tiempo de respuesta, utilización de recursos, estabilidad de métricas bajo carga y reconfiguración.	Ofrecen escenarios reproducibles para comparar configuraciones de autoscaling y políticas de gestión de recursos.	Enfocados principalmente en escalabilidad; la resiliencia ante fallos inyectados debe combinarse con herramientas CE adicionales.
Outage-Watch y enfoques de detección temprana	Técnicas de predicción de caídas basadas en ML sobre métricas de QoS.	MTTD, precisión en predicción de incidentes, reducción porcentual de MTTD frente a enfoques reactivos.	Demuestran reducciones significativas de MTTD al anticipar degradaciones; complementan CE mejorando la resiliencia operativa (detección).	No inyectan fallos por sí mismas; dependen de datos históricos ricos y de modelos bien entrenados; complejidad algorítmica.

Nota. Comparación de herramientas y enfoques de Chaos Engineering y evaluación de resiliencia en arquitecturas distribuidas y entornos Kubernetes. Elaboración propia a partir de Agarwal et al. (2023), Akgül y Güvez (2024), Baudry et al. (2022), Morin et al. (2022), Simonsson et al. (2021)

3.4 Estudios comparativos y métricas utilizadas

Los estudios comparativos en Chaos Engineering convergen en la necesidad de medir de forma objetiva la resiliencia de los sistemas antes y después de la introducción de patrones de tolerancia a fallos. En la práctica, esto se operacionaliza mediante un conjunto de métricas que capturan tanto la capacidad de detección como la capacidad de recuperación y la estabilidad del comportamiento en estado estable. Las métricas clásicas de confiabilidad Tiempo Medio para Detectar (MTTD), Tiempo Medio para Recuperar (MTTR) y Tiempo Medio entre Fallos (MTBF) se combinan con métricas de disponibilidad, tasa de errores, latencia y cumplimiento de SLO (Service Level Objectives) para evaluar el impacto de los experimentos de caos sobre sistemas distribuidos modernos. En entornos de alta criticidad como servicios cloud y microservicios, estas métricas se conectan directamente con pérdidas económicas, violaciones de SLA y satisfacción de personas usuarias finales (Beyer et al., 2016; Forsgren et al., 2018).

3.4.1 Métricas de resiliencia y su papel en Chaos Engineering

En términos generales, MTTD se define como el tiempo promedio que transcurre entre el inicio real de un incidente y su detección por parte del sistema de monitoreo o del equipo de SRE. MTTR, por su parte, mide el tiempo desde dicha detección hasta la restauración del servicio a un nivel aceptable de operación (no necesariamente perfecto, pero sí compatible con los SLO acordados). MTBF cuantifica el tiempo medio entre fallos significativos, capturando la estabilidad del sistema en el largo plazo. Estas métricas se relacionan a través de la fórmula clásica de disponibilidad aproximada:

Disponibilidad ≈ MTBF / (MTBF + MTTR)

En Chaos Engineering, estas magnitudes no se observan únicamente de forma pasiva, sino que se **provocan fallos controlados** para estimar cómo cambian MTTD, MTTR y MTBF bajo escenarios adversos. El objetivo no es solo "sobrevivir" al experimento, sino **reducir sistemáticamente MTTD y MTTR**, y aumentar MTBF a medida que se introducen mejoras de diseño (por ejemplo, circuit breakers, reintentos con backoff exponencial, timeouts coherentes y mecanismos de failover).

La literatura reciente sobre evaluación de sistemas auto-adaptativos y auto-curativos refuerza este enfoque. Naqvi et al. (2022) proponen CHESS como un método para **evaluar** (no solo probar) sistemas self-healing a partir de inyección sistemática de fallos, poniendo énfasis explícito en métricas de efectividad de la auto-recuperación, robustez frente a perturbaciones y mantenimiento de niveles aceptables de calidad de servicio.

De forma complementaria, Malik et al. (2023) muestran cómo estas métricas se operacionalizan en artefactos reproducibles, donde cada experimento de caos genera trazas de comportamiento normal y anómalo que luego se comparan en términos de tiempos de degradación, tiempos de recuperación y amplitud del impacto sobre el sistema.

Además de MTTD/MTTR/MTBF, los trabajos de ingeniería de confiabilidad modernos incorporan métricas orientadas a la experiencia de la persona usuaria, tales como:

- Latencia en percentiles altos (p95, p99).
- Tasa de errores (porcentaje de solicitudes fallidas).
- Throughput o volumen de peticiones por unidad de tiempo.
- Cumplimiento de SLO y consumo del error budget.

Estos indicadores se emplean como **variable de estado estable** en Chaos Engineering: se define formalmente "qué significa que el sistema esté sano" (por ejemplo, menos de 1 % de errores y latencia p95 < 300 ms), y luego se verifica si, bajo fallos inyectados, el sistema es capaz de mantener o recuperar ese estado en un tiempo acotado (Basiri et al., 2016; Beyer et al., 2016).

3.4.2 Evidencia empírica en sistemas distribuidos y microservicios

El cuerpo de literatura técnica sobre Chaos Engineering en sistemas distribuidos ha crecido de forma significativa en los últimos años, aportando evidencia cuantitativa sobre resiliencia. Zhang y colaboradores han desarrollado una línea de trabajo centrada en la inyección de fallos a nivel de excepciones y llamadas al sistema, que permite medir el impacto en términos de estabilidad y capacidad de recuperación en aplicaciones Java y servicios contenedorizados.

En A Chaos Engineering System for Live Analysis and Falsification of Exception-Handling in the JVM, Zhang et al. (2021) presentan ChaosMachine, un sistema que inyecta fallos en bloques try-catch en producción para clasificar el código de manejo de excepciones como resiliente, observable, debuggable o silencioso, en función de sus efectos sobre métricas como disponibilidad, errores y tiempos de respuesta.

Este enfoque hace explícita la relación entre estructuras de código y métricas operativas de resiliencia, abriendo la puerta a comparaciones cuantitativas entre distintas versiones de una misma aplicación (por ejemplo, antes y después de introducir estrategias de manejo de excepciones más robustas).

Posteriormente, en Maximizing Error Injection Realism for Chaos Engineering with System Calls, los mismos autores proponen Phoebe, un sistema que genera modelos realistas de errores a nivel de llamadas al sistema y los utiliza para inyectar fallos en contenedores.

A través de experimentos controlados, se observa cómo diferentes patrones de fallo (errores intermitentes vs. persistentes, degradaciones parciales vs. totales) afectan a la tasa de éxito de peticiones y a la latencia, permitiendo comparar configuraciones de despliegue y estrategias de mitigación en términos de su impacto en las métricas de resiliencia.

Simonsson et al. (2021) dan un paso más con ChaosOrca, una plataforma de inyección de fallos sobre llamadas al sistema en aplicaciones contenedorizadas desplegadas con Docker.

En sus experimentos, los autores monitorizan indicadores como uso de CPU, utilización de memoria, latencia de peticiones y tasa de errores mientras se introducen fallos en operaciones clave (por ejemplo, operaciones de E/S, llamadas de red o acceso a disco). Los resultados muestran que ciertos microservicios presentan patrones de degradación abrupta ante fallos específicos (por ejemplo, timeouts en DNS) que no se manifiestan con pruebas de carga tradicionales, lo que subraya la importancia de combinar Chaos Engineering + observabilidad para descubrir vulnerabilidades de resiliencia “ocultas”.

Una contribución clave de este cuerpo de trabajo es que no se limita a describir “que el sistema aguanta” los fallos, sino que cuantifica la resiliencia utilizando métricas

comparables entre estudios. Por ejemplo:

- Variaciones en la tasa de éxito de peticiones antes y después de aplicar patrones de retry.
- Incrementos o reducciones en latencia p95 bajo diferentes configuraciones.
- Cambios en la duración de las ventanas de indisponibilidad y, por tanto, en MTTR efectivo.

Este tipo de métricas permite construir estudios comparativos entre arquitecturas (por ejemplo, monolito vs. microservicios), entre versiones de un mismo sistema y entre distintas estrategias de resiliencia (circuit breakers, bulkheads, replicación, etc.).

3.4.3 Evaluación de sistemas auto-adaptativos y auto-curativos

En la intersección entre Chaos Engineering y sistemas auto-adaptativos, Naqvi et al. (2022) proponen un marco para evaluar sistemas self-healing utilizando inyección de fallos sistemática. Su trabajo identifica atributos de calidad relevantes (robustez, estabilidad, eficiencia de la auto-recuperación) y los conecta con métricas concretas, tales como:

- Tiempo de detección de la desviación respecto al comportamiento esperado.
- Tiempo de activación del mecanismo de auto-adaptación.
- Tiempo total hasta la recuperación (equivalente a un MTTR específico del lazo de auto-curación).
- Porcentaje de experimentos en los que la auto-recuperación es exitosa sin intervención humana.

Sobre esa base, CHESS (Malik et al., 2023) operacionaliza la evaluación de sistemas self-healing mediante un framework que: (i) define escenarios de fallo funcionales e infraestructurales, (ii) inyecta dichos fallos sobre aplicaciones basadas en microservicios, y (iii) registra el comportamiento del sistema en términos de logs, eventos y métricas de calidad de servicio. En los estudios de caso (una smart office y la aplicación de ejemplo Yelb), CHESS compara explícitamente el comportamiento del sistema con y sin el gestor auto-adaptativo, mostrando mejoras significativas en:

- Reducción del tiempo total de indisponibilidad por incidente.
- Disminución de la cascada de fallos entre microservicios.
- Aumento de la fracción de experimentos en que se mantiene la disponibilidad percibida por la persona usuaria.

Estos resultados ilustran cómo Chaos Engineering puede utilizarse no solo para descubrir puntos débiles, sino también como método cuantitativo para evaluar y comparar mecanismos de auto-recuperación. El enfoque de CHESS, además, se apoya en un servicio de auto-monitoreo que centraliza las métricas de interés, lo cual es coherente con la idea de que MTTD y MTTR dependen tanto de la arquitectura como del diseño del sistema de observabilidad (Malik et al., 2023; Naqvi et al., 2022).

3.4.4 Métricas de resiliencia en Kubernetes y entornos cloud-native

En el contexto de Kubernetes, la resiliencia se estudia típicamente a través de experimentos de auto-sanación que implican fallos de pods, nodos o componentes de red. La documentación oficial describe cómo el controlador de réplicas y el scheduler reprograman automáticamente pods en nodos sanos cuando detectan fallos, manteniendo el número deseado de réplicas y, por tanto, la disponibilidad del servicio. Sin embargo, la mera existencia de estos mecanismos no garantiza buenos valores de MTTD y MTTR: la configuración de probes de salud, timeouts, políticas de reinicio y autoscaling influye fuertemente en los tiempos reales de recuperación.

Trabajos como K8-Scalar, de Delnat et al. (2018), proporcionan bancos de pruebas reproducibles para comparar estrategias de escalado automático en clústeres orquestados con Kubernetes. Aunque su foco principal es el rendimiento y la capacidad de escalado (por ejemplo, tiempo de respuesta, throughput y utilización de recursos), sus experimentos sientan las bases para estudios de resiliencia, ya que cuantifican la estabilidad de las métricas de rendimiento bajo cargas variables y reconfiguraciones automáticas. En combinación con Chaos Engineering, este tipo de workbench permite evaluar no solo "qué tan bien escala" un autoscaler, sino qué tan rápido y estable se recupera ante fallos inyectados en pods o nodos.

Más recientemente, se han publicado guías y estudios sobre auto-healing de nodos en Kubernetes, que muestran cómo, en escenarios reales, el tiempo total de recuperación frente a fallos de nodo suele estar en el orden de decenas de segundos a varios minutos, dependiendo de los mecanismos de detección, del proveedor cloud y de la configuración de probes y timeouts. Estas mediciones son relevantes para Chaos Engineering porque ofrecen valores de referencia (benchmarks) frente a los cuales se pueden comparar los tiempos de recuperación medidos en experimentos de caos igualmente controlados.

Cuando se combinan plataformas de inyección de fallos específicas para Kubernetes (como las descritas en el apartado 7.3) con workbenches de evaluación como K8-Scalar o CHESS, se obtiene un marco experimental que permite:

- Definir un estado estable cuantitativo (por ejemplo, latencia p95 < 300 ms, tasa de éxito > 99,5 %, consumo de CPU < 70 %).
- Inyectar fallos controlados (caída de pods, pérdida parcial de conectividad, saturación de CPU o memoria, etc.).
- Medir MTTD y MTTR a nivel de clúster y a nivel de aplicación, distinguiendo claramente entre tiempos de reacción de Kubernetes y tiempos de recuperación lógica de la aplicación.
- Comparar resultados entre diferentes configuraciones de probes, políticas de autoscaling y patrones de resiliencia implementados en el código de la aplicación.

Este enfoque hace posible afirmar, por ejemplo, que una determinada combinación de readiness/liveness probes y políticas de retry reduce el MTTR efectivo ante fallos de pod en un cierto porcentaje, o que un patrón de circuit breaker evita la propagación de fallos de un microservicio a otros, reduciendo así el impacto global sobre la disponibilidad.

3.4.5 Reducción de MTTD y MTTR: resultados comparativos

Uno de los hallazgos más sólidos en la literatura reciente es que la combinación de observabilidad avanzada + Chaos Engineering puede reducir de forma significativa el MTTD y el MTTR de servicios cloud complejos. Un ejemplo concreto es Outage-Watch, una técnica para predicción temprana de caídas basada en el monitoreo de métricas de calidad de servicio (QoS) y en modelos de aprendizaje automático entrenados con un regularizador de eventos extremos (Agarwal et al., 2023).

En un estudio con datos reales de una empresa SaaS, Agarwal et al. (2023) muestran que Outage-Watch es capaz de detectar degradaciones severas de QoS minutos antes de que el sistema de monitoreo tradicional dispare una alerta, logrando reducir el MTTD de incidentes hasta en un 88 % en comparación con el enfoque reactivo utilizado por el equipo de ingeniería. Aunque este trabajo no se enmarca explícitamente como Chaos Engineering, su metodología es conceptualmente compatible: el modelo se entrena y evalúa en torno a la detección temprana de cambios en métricas de estado estable, que son precisamente las que se usan para evaluar el impacto de experimentos de caos. Integrar técnicas similares en un pipeline de Chaos Engineering permitiría no solo evaluar la resiliencia estructural del sistema, sino también la resiliencia operativa del proceso de detección.

A nivel de MTTR, los estudios de Naqvi et al. (2022) y Malik et al. (2023) reportan mejoras claras cuando se dota a los sistemas de capacidades self-healing y se evalúan sistemáticamente mediante inyección de fallos. En sus experimentos, la presencia de un gestor auto-adaptativo capaz de monitorizar el sistema, diagnosticar la perturbación y aplicar acciones de reconfiguración reduce significativamente el tiempo total de recuperación y la extensión de los fallos en cascada, en comparación con versiones equivalentes sin auto-curación.

En términos comparativos, estos resultados permiten establecer patrones generales:

- Sistemas con alta observabilidad (telemetría rica, alertas bien calibradas, modelos de detección temprana) tienden a presentar MTTD mucho menores, lo que a su vez facilita reducir MTTR.
- La introducción de mecanismos explícitos de auto-recuperación, evaluados de forma sistemática mediante Chaos Engineering, conduce a reducciones medibles en MTTR, al evitar pasos manuales de diagnóstico y reconfiguración.
- El uso de patrones de resiliencia (circuit breakers, reintentos, timeouts coherentes, bulkheads, replicación selectiva) se traduce en menor impacto en las métricas de estado estable bajo fallos inyectados, lo que se refleja en una menor variación del throughput, de la latencia y de la tasa de errores.

3.4.6 Síntesis y lineamientos para el trabajo de tesis

A partir de la literatura revisada, se puede afirmar que los estudios comparativos en Chaos Engineering convergen en tres ideas clave:

La resiliencia es cuantificable: MTTD, MTTR, MTBF, disponibilidad porcentual, tasas de error, latencias en percentiles altos y cumplimiento de SLO constituyen un conjunto coherente de métricas para caracterizar la capacidad de un sistema para soportar fallos.

Chaos Engineering proporciona el contexto experimental necesario para comparar arquitecturas, configuraciones y patrones de resiliencia, siempre que los experimentos se definan sobre la base de estados estables cuantificados y se ejecuten con trazabilidad de métricas.

Los sistemas auto-adaptativos y auto-curativos, evaluados con frameworks como CHESS y metodologías como las de Naqvi et al. (2022), muestran mejoras empíricas en MTTD y MTTR frente a soluciones tradicionales, y sirven como referencia para diseñar arquitecturas cloud-native más robustas.

En el marco de este trabajo, estas conclusiones orientan el diseño de los experimentos comparativos: será necesario definir, para el caso de estudio en Kubernetes, un conjunto explícito de métricas de estado estable y de resiliencia (MTTD, MTTR, porcentaje de éxito de recuperación, variación de latencia y tasas de error) y utilizar herramientas de inyección de fallos para comparar arquitecturas con y sin patrones de resiliencia específicos. De esta manera, los resultados empíricos podrán situarse dentro del panorama internacional de estudios comparativos en Chaos Engineering, apoyados en métricas y referencias ampliamente utilizadas en la literatura.

3.5 Vacíos de investigación identificados

A pesar de los avances identificados en el estado del arte, persisten vacíos relevantes en la literatura sobre Chaos Engineering y resiliencia en arquitecturas cloud-native. En primer lugar, son escasos los trabajos que comparan de forma controlada arquitecturas desplegadas en Kubernetes con y sin patrones de resiliencia, manteniendo constantes la carga de trabajo, el dominio funcional y el entorno de ejecución (Akgül & Güvez, 2024). En segundo lugar, la integración sistemática entre experimentos de caos y enfoques predictivos basados en modelos de grafos y análisis estructural sigue siendo incipiente y con pocas validaciones empíricas en nubes públicas (Krasnovsky & Zorkin, 2025). Finalmente, la literatura disponible apenas aborda el impacto operativo y económico de introducir pruebas de caos sobre los costos de infraestructura y la gestión cotidiana de las plataformas, y la producción académica en español continúa siendo limitada (Steadybit, 2023; Zaguan, 2022).

La presente tesis aborda de manera directa el primer vacío mediante el diseño de un experimento controlado que compara dos versiones de un mismo sistema de transferencias bancarias desplegado en Kubernetes: una arquitectura distribuida sin patrones de resiliencia explícitos y una versión refactorizada con patrones de circuit breaker, retry, bulkhead y replicación en el bróker de mensajería. Ambas versiones se someten a los mismos escenarios de fallo y a cargas de trabajo equivalentes, mientras se miden métricas como tasa de errores, latencia, disponibilidad, MTTR y MTTD. De esta forma, se genera evidencia cuantitativa replicable sobre el efecto concreto de los patrones de resiliencia en un contexto cloud-native realista, contribuyendo a llenar el vacío de comparaciones controladas señaladas en la literatura.

En relación con la integración de Chaos Engineering y enfoques predictivos, la investigación no desarrolla un modelo de simulación de grafos completo; sin embargo, sí propone un conjunto estructurado de escenarios de fallo, variables e indicadores que pueden ser utilizados como insumo para futuros modelos de predicción y análisis estructural. Al documentar de forma sistemática los flujos críticos, los puntos de fallo, los patrones aplicados y las respuestas observadas en el sistema, la tesis aporta una base empírica que complementa las propuestas teóricas de integración entre ingeniería del caos y modelos de grafos (Krasnovsky & Zorkin, 2025), avanzando un paso en la dirección de vincular observaciones experimentales con marcos analíticos más abstractos.

En cuanto al impacto operativo y económico, el trabajo no construye un modelo financiero detallado de costos de infraestructura; no obstante, sí cuantifica el efecto de los patrones de resiliencia sobre indicadores directamente relacionados con la operación, como el porcentaje de peticiones fallidas, el tiempo de recuperación ante caídas de servicios y la capacidad de mantener flujos críticos bajo condiciones de fallo. Estas métricas constituyen insumos valiosos para análisis posteriores de coste-beneficio, ya que permiten estimar, por ejemplo, la reducción potencial en incidentes graves o en tiempo de indisponibilidad atribuible a la adopción de determinados patrones (Steadybit, 2023). De este modo, la tesis contribuye parcialmente a cerrar el vacío sobre el impacto operativo de Chaos Engineering, ofreciendo datos empíricos que la literatura actual suele mencionar de forma sólo conceptual.

Finalmente, la investigación contribuye a la limitada literatura académica en español sobre Chaos Engineering al presentar un caso de estudio completo, documentado metodológicamente y centrado en un dominio de alta criticidad como las transferencias bancarias. Al articular en castellano los conceptos, herramientas y patrones, y al proponer una plantilla de experimentos y un conjunto de métricas aplicables en contextos reales, el trabajo se suma a esfuerzos como los de Zaguan (2022) para extender la discusión más allá del ámbito angloparlante. En síntesis, la tesis no sólo identifica vacíos, sino que ofrece respuestas parciales y líneas de avance concretas, especialmente en la comparación controlada de arquitecturas con y sin resiliencia y en la generación de evidencia empírica útil para futuras investigaciones y aplicaciones industriales.

4. Metodología de la investigación

4.1 Tipo de estudio

El estudio se enmarca en un enfoque cuantitativo, de carácter experimental y comparativo, en el que se manipulan deliberadamente variables independientes la presencia o ausencia de patrones de resiliencia para observar su efecto sobre métricas cuantitativas de confiabilidad en condiciones controladas. Este diseño experimental permite establecer relaciones causales entre los patrones aplicados y las mejoras observadas en la resiliencia del sistema.

4.2 Método de investigación

Siguiendo la metodología experimental descrita en el apartado previo, se comparará el comportamiento de dos configuraciones arquitectónicas de una misma aplicación distribuida: una versión “tradicional”, sin patrones explícitos de resiliencia, y una versión “resiliente”, en la que se incorporan patrones de diseño orientados a tolerancia a fallos y contención de cascadas. De acuerdo con la lógica de los estudios experimentales, el propósito es establecer diferencias significativas entre ambas configuraciones bajo condiciones de fallo controlado, reduciendo al máximo la influencia de factores extraños (Hernández-Sampieri & Mendoza, 2018).

Desde la perspectiva de la Ingeniería del Caos, el diseño metodológico se fundamenta en ciclos iterativos de hipótesis–experimento–análisis–aprendizaje, donde los fallos se inyectan de manera controlada y se observa la evolución de indicadores clave como MTTR, MTTD, disponibilidad y tasas de error. Esta lógica experimental se articula con el enfoque de patrones de resiliencia y métricas de resiliencia revisado en el marco teórico, de modo que la metodología operativiza empíricamente conceptos previamente discutidos (Basiri et al., 2016; Di Francesco, Lago, & Malavolta, 2019).

En coherencia con ello, el temario metodológico a desarrollar en este capítulo comprende: (a) el diseño y construcción de la aplicación objeto de estudio, (b) la estrategia general para incorporar Chaos Engineering al proceso experimental y (c) la definición detallada del diseño experimental, las variables, los escenarios de fallo y las técnicas de análisis de datos.

4.3 Hipótesis de investigación

La hipótesis que guiará la investigación plantea que la incorporación de patrones de resiliencia en una arquitectura de microservicios desplegada en Kubernetes mejora significativamente la resiliencia del sistema, en comparación con una versión equivalente sin dichos patrones, cuando ambas se someten a los mismos escenarios de fallo controlado y bajo cargas de trabajo equivalentes. Esta resiliencia mejorada se refleja en métricas como menor MTTR y MTTD, reducción en la tasa de errores, mayor disponibilidad del servicio

y disminución de la latencia bajo condiciones adversas.

4.4 Diseño y desarrollo de aplicación.

Una etapa crítica para cualquier iniciativa de Chaos Engineering es la comprensión profunda del sistema sobre el cual se ejecutarán los experimentos. En este trabajo se ha optado por el diseño y desarrollo de una plataforma de transferencias bancarias, al tratarse de un dominio ampliamente conocido, transaccional y sensible a la disponibilidad, lo cual resulta idóneo para estudiar patrones de resiliencia en arquitecturas distribuidas. Este tipo de sistema permite representar flujos de negocio realistas (consultas de saldo, historial de movimientos, transferencias internas y externas), donde la interrupción del servicio tiene implicaciones claras en términos de experiencia de usuario y riesgo operativo.

La aplicación se implementará como una arquitectura distribuida híbrida, basada en microservicios y servicios de apoyo, desplegados en contenedores Docker orquestados por un clúster de Kubernetes. Esta elección es coherente con la literatura reciente, que identifica a los microservicios desplegados sobre plataformas cloud-native como un contexto privilegiado para estudiar resiliencia, patrones de tolerancia a fallos y propagación de degradaciones (Di Francesco et al., 2019). El entorno de despliegue reproducirá un escenario cercano a producción, pero controlado, lo que facilita la ejecución de experimentos de caos con un radio de impacto acotado.

El sistema de transferencias bancarias estará compuesto por los siguientes elementos principales. En primer lugar, un API bancario que concentra la lógica de negocio central: gestión de cuentas de clientes, registro de transacciones, ejecución de transferencias internas y entre bancos, así como consultas de historial. Este servicio se desarrollará en Java utilizando Spring Framework, con una base de datos PostgreSQL para almacenar información de cuentas y movimientos. En segundo lugar, un conjunto de microservicios orquestadores y portales se encargará de recibir las peticiones de los clientes, coordinar los flujos con el API central y publicar eventos en tópicos para su procesamiento asíncrono.

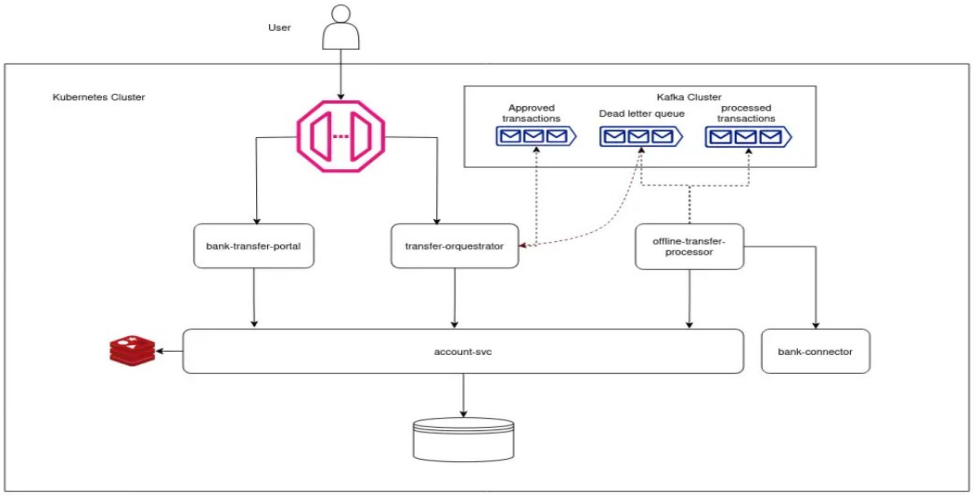
En tercer lugar, un clúster de Kafka permitirá gestionar transacciones de manera asíncrona mediante tópicos diferenciados de acuerdo con las etapas del flujo (recepción, validación, compensación, notificación). Kafka aporta capacidades de replicación y tolerancia a fallos en el canal de mensajería, lo que se alinea con los patrones de resiliencia basados en colas y procesamiento desacoplado. Finalmente, se incluirá un servidor Redis para el cacheo de lecturas frecuentes (por ejemplo, saldos y resúmenes de movimientos recientes), reduciendo la carga sobre la base de datos transaccional y mejorando la latencia percibida por las personas usuarias.

El diseño se documentará mediante diagramas de arquitectura que especifiquen servicios, dependencias, protocolos de comunicación y puntos críticos de fallo. Esta modelización previa es clave para la posterior construcción del grafo de dependencias, que se utilizará tanto para planificar experimentos de caos como para interpretar los resultados de las

métricas de resiliencia (Mendonça et al., 2020; Yodo & Wang, 2016).

Figura 1

Arquitectura de sistema de transferencias bancarias utilizada en esta investigación



Nota. Diagrama elaborado como una abstracción del sistema utilizado para los ensayos. La elaboración es propia.

4.4.1 Estrategia general de implementación de Chaos Engineering

La estrategia de Chaos Engineering que guiará la investigación se inspira en el enfoque iterativo propuesto en la literatura, donde la inyección de fallos se entiende como un medio sistemático para aprender sobre el comportamiento del sistema en condiciones adversas, y no como un fin en sí mismo (Basiri et al., 2016). Desde esta perspectiva, “romper cosas” deja de ser una práctica aislada y se convierte en un proceso estructurado que requiere planificación, control de riesgos y mecanismos claros de rollback.

En primer lugar, se adoptará un método incremental. No se ejecutarán experimentos directamente sobre toda la plataforma, sino sobre subconjuntos bien delimitados del sistema, con un blast radius acotado. Se priorizarán los servicios y flujos de negocio más críticos -por ejemplo, la ejecución de transferencias y la confirmación de las mismas-, ya que son los que concentran mayor riesgo operativo y relevancia para la organización. Este enfoque coincide con las recomendaciones metodológicas de los estudios que proponen iniciar las campañas de caos por escenarios de alto impacto y alto beneficio potencial en términos de aprendizaje (Basiri et al., 2016; Mendonça et al., 2020).

En segundo lugar, se seguirá un ciclo metodológico compuesto por las fases de observación, planteamiento del problema, formulación de hipótesis, diseño del experimento, análisis e integración de conclusiones. La fase de observación consiste en estudiar el comportamiento base del sistema (estado estable) sin inyección de fallos, identificando métricas relevantes y cuellos de botella preliminares. En la fase de planteamiento del problema se selecciona un subsistema o flujo de interés, así como un tipo de fallo cuya incidencia se desea estudiar: por ejemplo, fallos de red entre microservicios, aumento de la latencia, saturación de CPU o indisponibilidad de la base de

datos.

Posteriormente, se formula una hipótesis sobre el comportamiento esperado del sistema ante dicho fallo, especificando qué variables se medirán y en qué rango se considerarán aceptables. A continuación, se diseña y ejecuta el experimento de caos correspondiente, incluyendo siempre un procedimiento de rollback claramente definido que permita detener la prueba si se alcanzan umbrales de riesgo. Finalmente, en las fases de análisis y conclusiones se comparan los resultados observados con los esperados, se identifican patrones de degradación y se formulan recomendaciones de mejora arquitectónica o de configuración de patrones de resiliencia.

Si bien en un entorno organizacional real este proceso suele involucrar a múltiples personas y equipos (negocio, desarrollo, operaciones, seguridad), en el presente trabajo se simulará esta coordinación a través de una planificación explícita de roles y responsabilidades dentro del contexto académico. Ello permitirá mantener la rigurosidad metodológica, aun cuando el equipo de investigación sea reducido, y garantizar que los flujos críticos de información de la aplicación sean sometidos a experimentos de caos con objetivos claros y criterios definidos de evaluación.

4.4.2 Diseño Experimental.

El diseño experimental se estructura en torno a un proceso iterativo que comprende varias fases, alineadas con la estrategia general descrita anteriormente. El objetivo central es comparar dos configuraciones arquitectónicas de la misma aplicación: una sin patrones explícitos de resiliencia y otra con patrones de resiliencia integrados en su diseño e implementación. Esta comparación permitirá cuantificar el efecto de dichos patrones sobre métricas de resiliencia relevantes.

4.4.2.1. Fases del experimento

En la fase inicial, se desplegará la aplicación distribuida en un entorno controlado de Kubernetes, configurada sin patrones explícitos de resiliencia en sus componentes principales. La comunicación entre servicios se implementará mediante llamadas HTTP REST tradicionales, y las operaciones sobre la base de datos se realizarán utilizando mecanismos estándar del framework de persistencia (por ejemplo, repositorios JPA), sin controles adicionales de concurrencia, reintentos o timeouts avanzados. En esta fase se completarán, para esta configuración “tradicional”, las etapas de observación, planteamiento del problema y formulación de hipótesis.

En la fase de pruebas, la aplicación se someterá a técnicas de Chaos Engineering bajo una carga de trabajo que represente un escenario de operación normal. Para ello, se utilizarán herramientas de generación de carga (por ejemplo, JMeter o Locust) capaces de emitir peticiones HTTP REST con distintos niveles de concurrencia y patrones de tráfico. Mientras la aplicación procesa estas peticiones, se inyectarán fallos controlados durante periodos de tiempo determinados, siguiendo los escenarios de fallo definidos (terminación de pods, particiones de red, saturación de CPU, indisponibilidad de servicios críticos, etc.).

Esta fase corresponde al núcleo experimental del diseño y permitirá observar cómo se degradan las métricas de resiliencia.

Posteriormente, se desarrollará una fase de aplicación de mejoras, coherente con la naturaleza iterativa de la Ingeniería del Caos. A partir de los resultados de la primera ronda de experimentos, se realizará un proceso de refactorización arquitectónica e implementación de patrones de resiliencia (por ejemplo, circuit breakers, retries con backoff, timeouts explícitos, uso más intensivo de colas asíncronas, cacheo selectivo). Estos patrones se implementarán principalmente a nivel de código fuente, utilizando librerías de resiliencia compatibles con el framework del backend, de forma que las mejoras sean portables y no dependan exclusivamente de la plataforma de despliegue.

Una vez desplegada la nueva versión "resiliente" en el mismo entorno de pruebas, se llevará a cabo una segunda iteración completa del proceso: verificación básica mediante smoke tests, ejecución de carga, inyección de fallos y recolección de datos. La comparación entre los resultados de ambas iteraciones permitirá evaluar si la hipótesis de mejora de resiliencia se confirma y en qué magnitud.

4.4.2.2. Escenarios de fallo, variables e indicadores

Los escenarios de fallo previstos incluyen, entre otros: (a) terminación controlada de pods y servicios específicos, (b) inyección de latencia o pérdida de paquetes en la comunicación entre microservicios, (c) sobrecarga de CPU o memoria en servicios críticos y (d) indisponibilidad temporal de la base de datos o del clúster de mensajería. Cada escenario se ejecutará bajo condiciones homogéneas de hardware, red y configuración, y se repetirá al menos diez veces para reducir el impacto del azar y mejorar la estabilidad de las estimaciones.

La variable independiente principal será el tipo de arquitectura: configuración tradicional (sin patrones explícitos de resiliencia) frente a configuración resiliente (con patrones de resiliencia integrados). Las variables dependientes estarán compuestas por un conjunto de métricas de resiliencia, entre las que destacan el Mean Time To Recovery (MTTR), el Mean Time To Detection (MTTD), la disponibilidad del sistema durante los experimentos, la tasa de errores (error rate) y la latencia promedio y de percentiles altos (por ejemplo, p95 y p99). Estas métricas permiten capturar tanto la severidad como la duración de la degradación, en línea con las recomendaciones de la literatura sobre curvas de resiliencia y métricas operacionales (Yodo & Wang, 2016; Poulin & Kane, 2021).

Desde el punto de vista de la recolección de datos, se utilizarán herramientas de monitoreo como Prometheus y Grafana para registrar series de tiempo de las métricas seleccionadas durante todo el ciclo de los experimentos. Se prestará especial atención a eventos como reinicios de pods, cambios en el estado de los servicios, picos de latencia y errores percibidos por las personas usuarias simuladas. Esta instrumentación permitirá construir curvas de resiliencia para cada escenario y cada configuración arquitectónica, lo que facilitará la comparación sistemática de resultados (Tang et al., 2023).

*4.4.2.*3. Análisis de datos y criterios de validez

El análisis de los datos recolectados combinará técnicas descriptivas y comparativas. En primer lugar, se calcularán estadísticos descriptivos (promedios, medianas, desviaciones estándar, percentiles) para MTTR, MTTD, disponibilidad, tasas de error y latencias en cada escenario y configuración. A continuación, se aplicarán pruebas de hipótesis -como pruebas t de Student o análisis de varianza (ANOVA), según corresponda- para determinar si las diferencias observadas entre la arquitectura tradicional y la resiliente son estadísticamente significativas (Hernández-Sampieri & Mendoza, 2018).

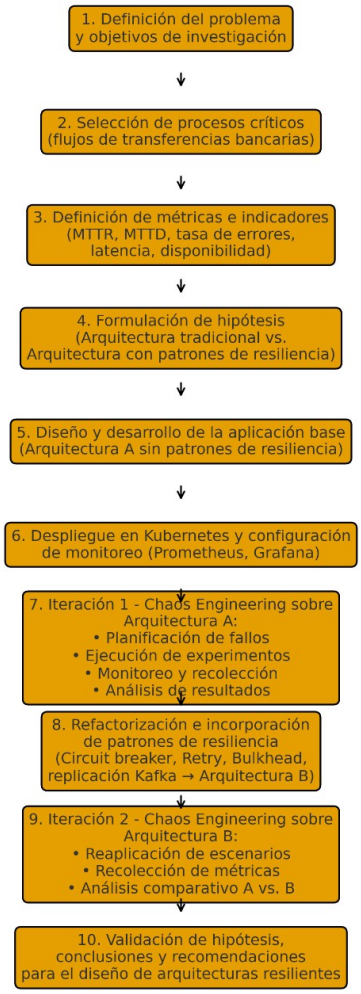
Asimismo, se elaborarán representaciones gráficas que incluyan histogramas, diagramas de cajas y curvas temporales de resiliencia, con el fin de visualizar la evolución del desempeño del sistema antes, durante y después de los eventos de fallo. Estas visualizaciones ayudarán a identificar patrones de degradación y recuperación, posibles efectos de techo o suelo, y escenarios en los que la arquitectura resiliente no ofrece mejoras sustantivas respecto a la tradicional. En tales casos, se podrán plantear iteraciones adicionales de refactorización y experimentación.

Para fortalecer la validez interna del estudio, se procurará que las únicas diferencias entre ambas configuraciones sean la presencia o ausencia de patrones de resiliencia, manteniendo constantes el dominio de la aplicación, el entorno de despliegue, las cargas de trabajo y los escenarios de fallo. La validez externa se abordará mediante una discusión crítica de la generalización de los resultados a otros dominios y arquitecturas, reconociendo que el sistema de transferencias bancarias es un caso representativo, pero no exhaustivo, de las posibles aplicaciones de Chaos Engineering en sistemas distribuidos.

Finalmente, la definición de objetivos y métricas para cada hipótesis se realizará de forma explícita. Por ejemplo, ante la hipótesis de que una configuración activo–activo con patrones de resiliencia cumple un requisito de alta disponibilidad, se establecerán umbrales máximos de tiempo de indisponibilidad y de número de peticiones fallidas admisibles durante la caída de un nodo. El sistema de monitoreo deberá ser capaz de recuperar esta información con precisión, permitiendo determinar si la hipótesis se acepta o se rechaza en función de criterios cuantitativos claramente definidos (Basiri et al., 2016). En síntesis, la metodología se resume así:

Figura 2

Metodología de investigación basada en Chaos Engineering



Nota. Imagen descriptiva de los diferentes pasos que componen la metodología. Elaboración propia.

5. Presentación de los resultados

Como parte de la investigación se llevó a cabo una campaña de pruebas experimentales que forman parte del ciclo iterativo de aplicación de Chaos Engineering definido en la metodología. En una primera etapa se evaluó el comportamiento de la arquitectura tradicional (sin patrones de resiliencia explícitos) frente a distintos tipos de fallos inducidos. Posteriormente, en una segunda etapa, se repitieron los mismos escenarios sobre una versión de la aplicación en la que se incorporaron patrones de resiliencia específicos (circuit breaker, retry, bulkhead y replicación), con el fin de comparar cuantitativamente el grado de mejora alcanzado.

En todos los casos se trabajó sobre las dos operaciones principales del sistema de transferencias bancarias: la consulta de historial de transferencias y la creación de nuevas transferencias, tanto internas como entre bancos. Las pruebas se realizaron bajo cargas de trabajo representativas de un entorno de operación normal, combinadas con fallos

controlados en servicios críticos o componentes de infraestructura.

5.1. Primera etapa: arquitectura sin patrones de resiliencia

5.1.1. Eliminación temporal del servicio API bancario (solo consultas)

En un primer experimento se evaluó el impacto de la caída temporal del servicio API bancario durante una carga sostenida de aproximadamente 8 peticiones por segundo hacia el servicio de consulta de transferencias por cuenta. El experimento tuvo una duración de 3 minutos, y el objetivo era observar la tasa de errores global durante el periodo de indisponibilidad y la capacidad del sistema de recuperar su estado estable a través del mecanismo de réplicas del deployment.

Durante la prueba se registraron 1 261 peticiones, de las cuales 131 resultaron fallidas, lo que equivale a una tasa de errores global del 10.38 %. En ausencia del mecanismo de replicación de Kubernetes, la caída del pod del API bancario habría generado un fallo del 100 % de las peticiones durante el intervalo de indisponibilidad. En consecuencia, aunque la tasa de errores se mantuvo por debajo del umbral del 20 % previsto en la hipótesis, se evidenció que el servicio de consulta por sí mismo no dispone de mecanismos internos para gestionar la indisponibilidad del API, dependiendo completamente de la capacidad del clúster para recrear el pod.

Este resultado muestra una resiliencia limitada frente a fallos puntuales del servicio central, en tanto el manejo de la falla se delega exclusivamente en la plataforma de orquestación y no en la lógica de la aplicación.

Figura 3
Resultados de experimento de eliminación de pod

Nota. Elaboración propia.

5.1.2. Eliminación del servicio API bancario con consultas y creación de transferencias

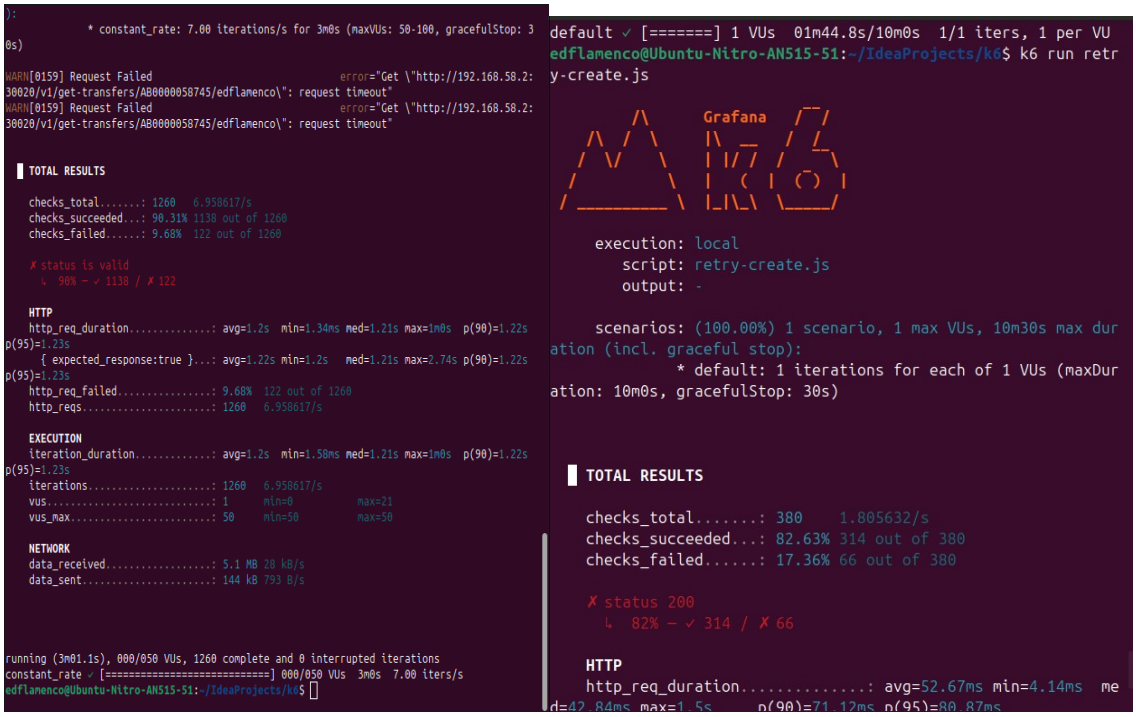
En un segundo experimento se amplió el escenario anterior, sometiendo de forma simultánea a carga de trabajo tanto el servicio de consulta de transferencias como el orquestador de creación de transferencias. De nuevo se eliminó el pod del API bancario durante 3 minutos, mientras ambos servicios emitían peticiones al backend. El propósito era observar el impacto de la caída del API sobre flujos paralelos de lectura y escritura.

Los resultados muestran que ninguno de los dos servicios fue capaz de procesar correctamente sus operaciones mientras el API bancario se encontraba indisponible. El servicio de consulta de transferencias registró 1 260 peticiones, de las cuales 122 fallaron, con una tasa de error del 9.68 %. Por su parte, el servicio de creación de transferencias procesó 380 solicitudes, de las cuales 66 resultaron fallidas, lo que representa una tasa de error del 17.63 %.

Aunque la tasa de errores del servicio de consulta se mantuvo levemente por debajo del umbral del 10 % planteado, la operación de creación de transferencias superó ampliamente dicho valor. En términos generales, los resultados indican que la arquitectura tradicional no ofrece mecanismos suficientes para garantizar la continuidad de ambos flujos de negocio ante la caída del API bancario, comprometiendo especialmente las operaciones de escritura.

Figura 4

Resultados de prueba de eliminación de api bancario durante carga de trabajo.



Nota. Elaboración propia.

5.1.3. Sobrecarga del flujo de consultas en el API bancario

El tercer experimento se centró en analizar el efecto de la sobrecarga de un flujo específico sobre el desempeño global del servicio API bancario. Para ello se saturó el endpoint de consulta de transferencias mientras, en paralelo, se emitían peticiones de creación de transferencias a través de otro endpoint. El experimento tuvo una duración de 1 minuto con generación continua de peticiones de consulta y creación.

La hipótesis de trabajo planteaba que, al compartir recursos (conexiones a base de datos, hilos, etc.) entre ambos flujos, la saturación del flujo de consultas terminaría afectando al flujo de creación de transferencias. Los resultados confirmaron claramente esta situación: el API bancario fue incapaz de completar la carga de trabajo en ninguno de los dos tipos de operación. La tasa de errores en el flujo de consulta alcanzó el 81.77 %, mientras que la tasa de errores en el flujo de creación llegó al 70 %.

Esta degradación severa pone en evidencia la ausencia de mecanismos de aislamiento de recursos entre operaciones críticas. La sobrecarga de un solo flujo fue suficiente para consumir los recursos compartidos y provocar una degradación generalizada, afectando tanto las operaciones de lectura como las de escritura.

Figura 5

Resultados de experimento de operaciones paralelas con 1 flujo sobrecargado

Nota. Elaboración propia.

5.1.4. Caída del bróker de mensajería en el procesamiento asíncrono

El cuarto experimento analizó el efecto de la caída del único bróker de Kafka utilizado para el procesamiento asíncrono de transferencias interbancarias. Durante 3 minutos se emitió una carga de trabajo compuesta por 200 solicitudes de creación de transferencias entre bancos, que requieren varios pasos orquestados mediante mensajería. En medio de esta carga se eliminó el bróker de Kafka, sin ningún tipo de replicación configurada.

Los resultados mostraron una pérdida significativa de mensajes: aproximadamente el 33 % de las peticiones y mensajes se perdieron durante el experimento, lo que implica que una tercera parte de la carga de trabajo no alcanzó a completar el flujo de procesamiento. En contraste, solo el 66 % de las solicitudes se procesó de manera exitosa.

En síntesis, el sistema resultó incapaz de completar la carga de trabajo de transferencias interbancarias cuando el bróker de mensajería falló, evidenciando una alta dependencia de un único punto de fallo y la ausencia de mecanismos de replicación o redundancia en la capa de mensajería.

Figura 6

Resultados de la ejecución de pruebas con 200 peticiones realizadas

Nota. Elaboración propia.

5.2. Segunda etapa: arquitectura con patrones de resiliencia

Tras analizar los resultados de la primera etapa, se procedió a refactorizar la aplicación incorporando patrones de resiliencia específicos en los servicios clave. A continuación, se presentan los resultados obtenidos al repetir los mismos escenarios de fallo, ahora sobre la arquitectura mejorada.

5.2.1. Eliminación temporal del servicio API bancario con patrón circuit breaker

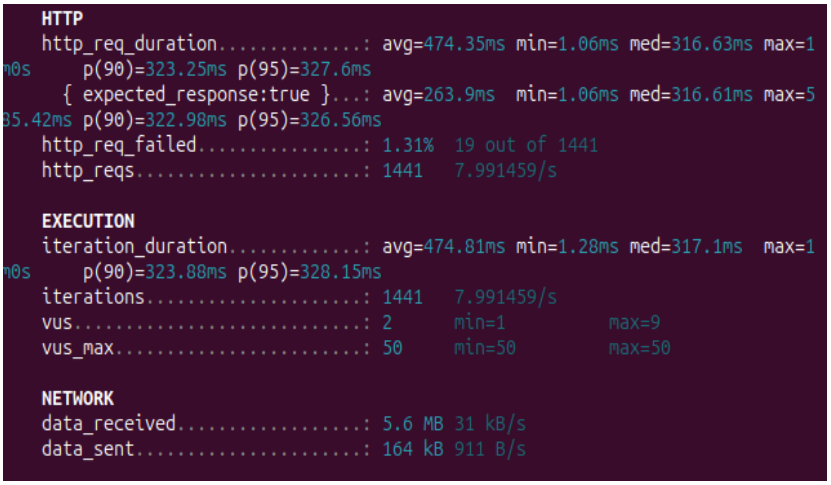
En el primer experimento de la segunda etapa se repitió la eliminación del pod del API bancario durante una carga aproximada de 8 peticiones por segundo hacia el servicio de consulta de transferencias. En esta ocasión, el servicio se reforzó con un patrón de resiliencia circuit breaker, configurado para abrirse tras detectar fallos y responder con un mensaje predeterminado durante 30 segundos, hasta que la API bancaria recuperara su disponibilidad.

Durante los 3 minutos de prueba se registraron 1 441 peticiones. El circuit breaker se mantuvo activo durante 30 segundos, evitando que se emitieran aproximadamente 240 peticiones hacia el servicio caído. El mecanismo de fallback respondió en un tiempo promedio de 1.06 ms con un código de respuesta 307, indicando una respuesta controlada y predecible. La tasa de errores no controlados fue de solo 1.31 %, equivalente a 19 peticiones fallidas de las 1 441 totales.

Estos resultados muestran una mejora sustancial frente a la primera etapa. El patrón circuit breaker logró reducir de forma drástica los errores no gestionados y mantuvo la operatividad del servicio de consulta durante el periodo de inestabilidad, validando la hipótesis de que la tasa de errores sería inferior al 2 % y que el tiempo de respuesta del fallback se mantendría por debajo de los 20 ms.

Figura 7

Resultado de primera prueba luego de aplicar el patron de resiliencia circuit breaker.



Nota. Elaboración propia.

5.2.2. Eliminación del API bancario con patrones de retry e idempotencia

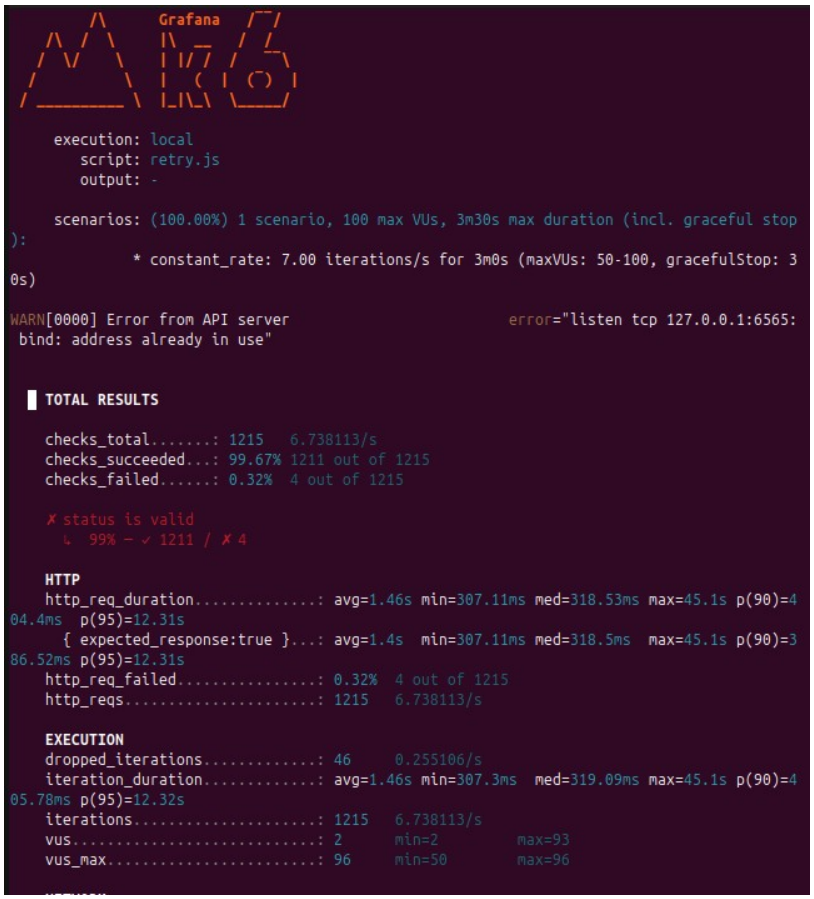
En el segundo experimento se volvió a eliminar el pod del API bancario bajo una carga combinada de consultas de transferencias y creación de nuevas transferencias, esta vez utilizando el patrón de resiliencia retry. Cada petición fallida se reintentó hasta un máximo de tres veces, con intervalos de 6 segundos entre cada intento. Adicionalmente, se incorporó idempotencia en la creación de transferencias, asignando un identificador único por operación para evitar duplicidades.

Durante los 3 minutos de prueba, ambos servicios procesaron correctamente más del 99 % de las peticiones recibidas. La tasa de errores no resueltos por el mecanismo de retry fue inferior al 1 % tanto en el servicio de consulta (1 215 peticiones) como en el servicio de creación (380 peticiones). No se detectaron transferencias duplicadas, lo que confirma la efectividad de la estrategia de idempotencia aplicada al flujo de escritura.

En conjunto, los resultados indican que los servicios fueron capaces de soportar y completar la carga de trabajo a pesar de la caída del API bancario, compensando la indisponibilidad momentánea mediante reintentos controlados y seguros en términos transaccionales.

Figura 8

Resultados de segunda prueba de consulta de transferencias luego de aplicar el retry



Nota. Elaboración propia.

5.2.3. Sobrecarga del flujo de consultas con patrón bulkhead

El tercer experimento de la segunda etapa buscó evaluar el impacto del patrón bulkhead en el manejo de la sobrecarga del flujo de consultas del API bancario. El patrón se configuró para aislar recursos (por ejemplo, conexiones a base de datos) por tipo de operación, de modo que cada flujo dispusiera de un conjunto de recursos independiente. Al saturarse el flujo de consultas, el bulkhead debía activar un mecanismo de fallback y evitar que la sobrecarga afectara a los demás flujos.

Durante 1 minuto de prueba, con peticiones de consulta y creación de transferencias en paralelo, el bulkhead logró aislar los recursos asociados a cada operación y activar su fallback cuando la carga de trabajo superó la capacidad configurada. El mecanismo de fallback respondió en menos de 100 ms, evitando que las peticiones quedaran bloqueadas. El flujo de consultas saturó sus propios recursos y el 52.74 % de las peticiones fue

rechazado por el fallback debido a dicha sobrecarga; sin embargo, los procesos de creación de transferencias, débito de saldos y reversión de operaciones se mantuvieron funcionales y no presentaron errores derivados de la saturación del flujo de consulta.

Este comportamiento confirma que el patrón bulkhead permitió contener la degradación en un único flujo, evitando una caída global del servicio y preservando la disponibilidad de los demás procesos críticos del sistema.

5.2.4. Caída de brokers de Kafka con replicación y particionado

Finalmente, en el cuarto experimento se analizó el efecto de introducir replicación y particionado en el clúster de Kafka utilizado para el procesamiento asíncrono de transferencias interbancarias. Se configuró un clúster de tres brokers y se ejecutó, nuevamente, una carga de trabajo de 200 transferencias entre bancos, eliminando de forma aleatoria uno o varios brokers durante la ventana de 3 minutos. La hipótesis planteaba que, mientras existiera al menos un broker funcional, el sistema sería capaz de completar la totalidad de las transferencias.

Los resultados mostraron que la tasa de peticiones y mensajes perdidos fue cero: no se registró pérdida de mensajes a pesar de la eliminación aleatoria de brokers durante el experimento. La tasa de peticiones procesadas exitosamente fue del 100 %. Aunque en determinados momentos los tres brokers fueron eliminados, la combinación de replicación y capacidades de recuperación de Kubernetes permitió restaurar el clúster y completar el procesamiento de todas las transferencias.

En consecuencia, el sistema logró manejar de manera robusta la carga de trabajo asíncrona aun frente a fallos reiterados en la capa de mensajería, evidenciando una mejora radical respecto a la primera etapa, donde la pérdida de un único bróker implicaba la pérdida de un tercio de las transferencias.

5.3. Comparativa de resultados

Tabla 2

Comparativa de resultados según tipo de arquitectura

Escenario de prueba	Arquitectura sin patrones de resiliencia	Arquitectura con patrones de resiliencia	Mejora observada
1. Caída temporal del API bancario (solo consultas)	1 261 peticiones totales; 131 fallidas. Tasa de error global: 10.38 %. El servicio de consulta depende totalmente del replicaset de Kubernetes.	Patrón circuit breaker. 1 441 peticiones; 19 errores no controlados. Tasa de error: 1.31 %. Fallback responde en ~1.06 ms y evita ~240 peticiones fallidas.	Reducción drástica de errores no controlados; el servicio se mantiene operativo durante la inestabilidad del API.
2. Caída del API bancario con consultas y creación de transferencias en paralelo	Consultas: 1 260 peticiones, 122 fallidas (9.68 %). Creación: 380 peticiones, 66 fallidas (17.63 %). Ambos flujos se ven fuertemente afectados.	Patrón retry + idempotencia. Consultas: >99 % de peticiones exitosas. Creación: >99 % exitosas. Errores no resueltos por retry: <1 % en ambos servicios, sin duplicar transferencias.	Los dos flujos completan su carga de trabajo pese a la caída temporal del API; errores residuales menores al 1 %.
3. Sobrecarga del flujo de consultas en el API bancario con creación en paralelo	Saturación del endpoint de consulta. Tasa de error en consultas: 81.77 %. Tasa de error en creación: 70 %. La sobrecarga de un flujo degrada todo el servicio.	Patrón bulkhead. Recursos aislados por operación. Flujo de consulta saturado: 52.74 % de peticiones rechazadas por fallback, pero sin bloqueo. Flujos de creación, débito y reversión mantienen 0 % de errores asociados a la saturación.	La degradación se confina al flujo sobrecargado; el resto de operaciones críticas se mantiene disponible y estable.
4. Caída del bróker de mensajería para procesamiento asíncrono de transferencias	200 transferencias interbancarias. Pérdida de mensajes: 33 %. Solo 66 % de las operaciones se completan con éxito al fallar el único bróker de Kafka.	Patrón de replicación y particionado (cluster de 3 brokers). Pérdida de mensajes: 0 %. Peticiones procesadas exitosamente: 100 %, incluso eliminando brokers de forma aleatoria durante el experimento.	Eliminación del punto único de fallo; el sistema completa el 100 % de las transferencias asíncronas sin pérdida de mensajes.

Nota. Elaboración propia con base a resultados obtenidos en las pruebas.

6. Discusión

Los resultados experimentales obtenidos permiten evaluar de manera cuantitativa el impacto de incorporar patrones de resiliencia en una arquitectura de microservicios desplegada sobre Kubernetes. En la primera etapa, la aplicación operó sin patrones explícitos de tolerancia a fallos; en la segunda, se introdujeron circuit breaker, retry, bulkhead y replicación del bróker de Kafka. La comparación controlada entre ambas versiones muestra reducciones sustantivas en la tasa de errores y en la pérdida de mensajes, así como una mejora en la capacidad del sistema para mantener su operatividad bajo condiciones de fallo inducido. Estos hallazgos son coherentes con la hipótesis planteada, según la cual la introducción de patrones de resiliencia mejora de forma significativa la resiliencia del sistema frente a fallos controlados.

En la primera etapa, la eliminación del servicio API bancario evidenció la fragilidad de la arquitectura base. La simple caída temporal de un pod produjo tasas de error del 10,38 % en consultas y del 17,63 % en la creación de transferencias, mostrando que la lógica de negocio dependía de manera directa y rígida de un único punto de fallo. De forma aún más crítica, la sobrecarga del flujo de consultas provocó tasas de error del 81,77 % y del 70 % en consultas y creación de transferencias respectivamente, lo que indica una fuerte competencia por recursos compartidos y ausencia de mecanismos de aislamiento. Finalmente, la caída del único bróker de Kafka derivó en la pérdida del 33 % de las peticiones, confirmando la existencia de un cuello de botella estructural en el procesamiento asíncrono. Estos resultados confirman, en línea con Basiri et al. (2016), que en sistemas distribuidos reales los fallos de infraestructura y de recursos compartidos pueden traducirse rápidamente en degradaciones severas de servicio si no se diseñan mecanismos específicos de resiliencia.

La segunda etapa de experimentos muestra un cambio cualitativo en el comportamiento del sistema tras la incorporación de patrones de resiliencia. En el escenario de caída del API bancario para consultas, el uso de circuit breaker redujo la tasa de errores no controlados a solo 1,31 %, con un fallback capaz de responder en torno a 1 ms y evitando aproximadamente 240 peticiones durante la ventana de inestabilidad. Desde la perspectiva de las métricas de resiliencia, esto implica una reducción efectiva del MTTR percibido por las personas usuarias, ya que el sistema deja de “bloquearse” o acumular timeouts y pasa a degradar su comportamiento de manera controlada. Este resultado coincide con las recomendaciones de la literatura sobre graceful degradation y presupuestos de error en entornos SRE (Beyer et al., 2016), donde se privilegia la continuidad de servicio aunque sea con funcionalidad reducida.

El patrón retry combinado con idempotencia en la creación de transferencias permitió que más del 99 % de las operaciones se completaran correctamente durante la caída del API bancario, manteniendo la tasa de errores por debajo del 1 % y evitando la

duplicación de transferencias. Este hallazgo resulta particularmente relevante en el dominio bancario, donde la duplicación de operaciones tiene consecuencias críticas. La comparación con la primera etapa muestra que un fallo transitorio que antes se traducía directamente en errores visibles para las personas usuarias pasa a gestionarse internamente mediante reintentos controlados. De nuevo, se verifica el planteamiento de la hipótesis: los patrones de resiliencia modifican no solo la probabilidad de fallo, sino la forma en que el sistema absorbe y reconfigura dichos fallos, reduciendo el impacto operativo.

El patrón bulkhead introdujo una dimensión adicional de resiliencia al aislar recursos para distintos flujos de trabajo. Bajo una sobrecarga intensa en las consultas, el sistema fue capaz de contener el problema en ese flujo, evitando que la saturación de conexiones a base de datos se propagara a la creación de transferencias, débitos y reversión de operaciones. Aunque el 52,74 % de las peticiones del flujo sobrecargado fue rechazado por el fallback del bulkhead, los demás procesos se mantuvieron operativos y sin errores. Este resultado muestra un cambio deliberado en la estrategia de gestión de fallos: se prefiere “fallar rápido” y de forma localizada antes que arrastrar a toda la aplicación a un estado de indisponibilidad general. En términos de resiliencia, el sistema gana capacidad para preservar sus flujos críticos a costa de sacrificar de manera controlada algunos escenarios de uso, lo que coincide con las recomendaciones de diseño de arquitecturas robustas en entornos cloud-native (Rosenthal & Jones, 2020; Akgül & Güvez, 2024).

Finalmente, la introducción de replicación en el clúster de Kafka eliminó por completo la pérdida de mensajes observada en la primera etapa. Mientras que con un solo bróker se perdía un 33 % de las transferencias asíncronas, en la segunda etapa el sistema procesó correctamente el 100 % de las peticiones, incluso cuando se eliminaron brokers de manera aleatoria durante la ventana de prueba. La combinación de replicación y particionado en Kafka, junto con las capacidades de auto-recuperación de Kubernetes, se tradujo en una mejora clara de la disponibilidad efectiva del canal de mensajería. Este hallazgo refuerza, con evidencia empírica, los argumentos teóricos sobre los beneficios de la redundancia y la replicación selectiva en arquitecturas event-driven (Naqvi et al., 2022; Krasnovsky, 2025).

En conjunto, los resultados muestran que la hipótesis de trabajo se ve respaldada: la arquitectura con patrones de resiliencia presenta tasas de error notablemente menores, ausencia de pérdida de mensajes y mejor capacidad de contención del fallo en comparación con la versión sin dichos patrones. Además, los hallazgos dialogan de forma coherente con el estado del arte: confirman la importancia de combinar Chaos Engineering con métricas como MTTD, MTTR y tasa de errores para medir resiliencia (Basiri et al., 2016; Owotogbe et al., 2024) y aportan un caso de estudio concreto en un dominio de alta criticidad como las transferencias bancarias. A diferencia de muchos trabajos que se centran en herramientas o en marcos conceptuales, esta tesis ofrece una comparación empírica directa entre dos arquitecturas equivalentes en Kubernetes,

contribuyendo a llenar uno de los vacíos identificados en la literatura.

Sin embargo, los resultados deben interpretarse considerando ciertas limitaciones. En primer lugar, se trata de un estudio de caso único, con una aplicación específica, un stack tecnológico concreto (Java, Spring, PostgreSQL, Kafka, Redis) y un clúster de Kubernetes particular, lo que limita la generalización inmediata a otros dominios o plataformas. En segundo lugar, las cargas de trabajo utilizadas, aunque diseñadas para imitar escenarios realistas, siguen siendo sintéticas y controladas; en entornos productivos reales podrían aparecer patrones de uso más complejos y eventos simultáneos no contemplados en los experimentos. En tercer lugar, el análisis se centró en métricas técnicas de resiliencia y no incorporó de manera explícita indicadores económicos o de experiencia de personas usuarias, por lo que las implicaciones de negocio de los patrones de resiliencia solo pueden inferirse de forma indirecta.

Pese a estas limitaciones, la investigación aporta evidencia cuantitativa y fácilmente replicable sobre el efecto de patrones de resiliencia en entornos cloud-native. Los resultados sugieren líneas claras para trabajos futuros: extender la comparación a otros dominios de negocio, incorporar medidas de coste-beneficio, integrar modelos predictivos basados en grafos que permitan seleccionar de forma automática los escenarios de caos más informativos y evaluar el efecto de Chaos Engineering en la organización, incluyendo prácticas SRE, procesos de respuesta a incidentes y cultura de aprendizaje. En ese sentido, la tesis no solo valida la hipótesis inicial, sino que también abre un espacio para seguir explorando cómo combinar experimentación controlada, patrones de diseño y métricas de resiliencia para construir sistemas más robustos en contextos de alta criticidad.

7. Conclusiones

En primer lugar, los resultados obtenidos permiten concluir que la hipótesis de trabajo se ve respaldada: la incorporación explícita de patrones de resiliencia (circuit breaker, retry, bulkhead y replicación en el bróker de mensajería) en una arquitectura de microservicios desplegada sobre Kubernetes mejora de forma significativa la resiliencia del sistema frente a fallos inducidos de manera controlada. La comparación entre la arquitectura base y la arquitectura reforzada, bajo las mismas condiciones de carga y escenarios de fallo, muestra reducciones sustantivas en la tasa de errores, eliminación de la pérdida de mensajes y una mejor contención del impacto de las fallas sobre los flujos críticos de negocio.

En segundo lugar, la tesis demuestra que es posible operacionalizar la resiliencia mediante un conjunto concreto de métricas, entre las que destacan la tasa de errores, la disponibilidad observada, la pérdida o conservación de mensajes en flujos asíncronos y la capacidad del sistema para mantener operativos sus procesos esenciales durante eventos adversos. Aun cuando no se calcularon formalmente todos los indicadores

clásicos (como MTTR y MTTD en sentido estricto), los experimentos evidencian que los patrones de resiliencia reducen el tiempo efectivo de indisponibilidad percibida y estabilizan el comportamiento del sistema bajo fallos, alineándose con las propuestas teóricas del estado del arte sobre evaluación cuantitativa de la resiliencia.

En tercer lugar, el estudio confirma que los patrones de resiliencia no solo reducen la probabilidad de fallo visible, sino que transforman el modo en que el sistema "experimenta" el fallo. El uso de circuit breaker y retry permite pasar de una situación en la que la caída transitoria del API bancario se traduce directamente en errores masivos de usuario, a un escenario en el que los fallos se gestionan internamente mediante reintentos controlados y respuestas de degradación elegante. De forma análoga, el patrón bulkhead muestra que es preferible aislar recursos y rechazar de manera explícita una fracción de peticiones antes que permitir que la sobrecarga de un flujo colapse todo el sistema. La replicación en Kafka, por su parte, transforma un único punto de fallo en una infraestructura capaz de procesar el 100 % de las transferencias asíncronas incluso bajo eliminación aleatoria de brokers.

En cuarto lugar, la investigación aporta evidencia empírica a uno de los vacíos identificados en el estado del arte: la falta de comparaciones controladas entre arquitecturas con y sin patrones de resiliencia en Kubernetes, manteniendo constantes el dominio funcional, la carga de trabajo y el entorno de ejecución. Al diseñar dos versiones de un mismo sistema de transferencias bancarias y someterlas a los mismos escenarios de caos, la tesis ofrece un ejemplo replicable de cómo estructurar estudios comparativos en Ingeniería del Caos, generando datos que pueden dialogar con propuestas conceptuales y revisiones sistemáticas recientes.

En quinto lugar, el trabajo también realiza una contribución metodológica. La propuesta de un ciclo experimental concreto constituye una guía práctica para equipos que deseen introducir Chaos Engineering en contextos cloud-native. Esta metodología, documentada paso a paso y aplicada a un dominio sensible como las transferencias bancarias, puede servir como referencia tanto para investigaciones futuras como para iniciativas industriales que busquen mejorar su postura de resiliencia.

En sexto lugar, la tesis contribuye a la aún limitada literatura académica en español sobre Ingeniería del Caos y resiliencia en arquitecturas cloud-native. Al sistematizar conceptos, herramientas, patrones y métricas en castellano, y al articular un caso de estudio completo en este idioma, el trabajo facilita la apropiación de estas prácticas por parte de comunidades técnicas y académicas hispanohablantes. Este aporte es especialmente relevante para organizaciones y equipos de la región que operan sistemas críticos pero no siempre tienen acceso directo a documentación especializada en inglés.

Finalmente, es importante señalar que los hallazgos deben interpretarse a la luz de ciertas limitaciones: se trata de un único estudio de caso, con una aplicación y un stack tecnológico específicos, y basado en cargas de trabajo sintéticas. No obstante, estas

restricciones abren líneas claras de investigación futura: extender el enfoque a otros dominios de negocio, incorporar métricas económicas y de experiencia de personas usuarias, integrar modelos predictivos basados en grafos para priorizar escenarios de caos y estudiar el impacto organizacional de la Ingeniería del Caos en prácticas SRE y culturas de aprendizaje. En conjunto, la tesis muestra que la combinación de patrones de resiliencia, Kubernetes y Chaos Engineering constituye un camino prometedor para construir sistemas más robustos, medibles y preparados para operar en entornos caracterizados por la complejidad y la incertidumbre.

8. Recomendaciones

Se recomienda que las organizaciones que operan sistemas críticos en entornos cloud-native institucionalicen la Ingeniería del Caos como un proceso continuo y no como actividades aisladas o puntuales. Esto implica definir una hoja de ruta con roles, responsabilidades, frecuencia de experimentos y criterios de éxito, articulada con las prácticas de DevOps y SRE. En el caso específico de sistemas de transferencias bancarias, esta institucionalización debería incluir la aprobación explícita por parte de la alta dirección y de las áreas de riesgo y cumplimiento, de modo que los experimentos se integren al gobierno de TI y a la gestión de continuidad de negocio.

Además, los resultados del estudio muestran que las métricas y las trazas son fundamentales para interpretar el impacto real de los fallos inducidos. Por ello, se recomienda que, previo a una adopción intensiva de Chaos Engineering, las organizaciones inviertan en fortalecer su plataforma de observabilidad: métricas detalladas, logs estructurados, trazas distribuidas y tableros alineados con flujos de negocio. Sin una base sólida de observabilidad, los experimentos de caos corren el riesgo de generar ruido en lugar de aprendizaje, dificultando la identificación precisa de cuellos de botella y puntos únicos de fallo.

Por otra parte, dado que los experimentos evidenciaron mejoras claras al aplicar circuit breaker, retry, bulkhead y replicación en el bróker de mensajería, se recomienda priorizar la introducción de estos patrones en los servicios que soportan procesos de negocio críticos, como las transferencias bancarias. Esta priorización debe basarse en análisis de riesgo y mapeo de dependencias: primero se refuerzan los servicios cuya caída o degradación tienen mayor impacto en personas usuarias o en obligaciones reguladas, y posteriormente se extiende el enfoque a servicios secundarios.

Asimismo, se recomienda avanzar hacia la automatización gradual de los experimentos de caos dentro de los pipelines de integración y entrega continua. Algunos experimentos básicos (por ejemplo, terminación de pods no críticos) pueden definirse como “pruebas de resiliencia” que deben pasar antes de promover una nueva versión a entornos superiores. Complementariamente, se sugiere realizar game days periódicos, donde

equipos técnicos y de negocio observan en conjunto el efecto de fallos inyectados sobre flujos reales, utilizando los hallazgos para actualizar playbooks, procedimientos de respuesta a incidentes y decisiones de diseño.

Adicionalmente, aunque el presente estudio se centra en métricas técnicas, los resultados permiten vislumbrar implicaciones económicas relevantes (reducción de errores, menor indisponibilidad, menos operaciones fallidas). Se recomienda que futuras iteraciones incorporen explícitamente un análisis de coste-beneficio, cuantificando, por ejemplo, el ahorro potencial por menor número de incidentes graves o por reducción de tiempo de caída. Este enfoque facilitará la toma de decisiones de inversión en patrones de resiliencia, herramientas de caos y capacidades de observabilidad, conectando los hallazgos técnicos con el lenguaje financiero y estratégico de la organización.

Referencias

Agarwal, S., Chakraborty, S., Garg, S., Bisht, S., Jain, C., Gonuguntla, A., & Saini, S. (2023). Outage-Watch: Early prediction of outages using extreme event regularizer. En *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. https://doi.org/10.1145/3611643.3616316

Akgül, M. A., & Güvez, H. (2024). The importance and implementation of chaos engineering in cloud architectures and applications. *İleri Mühendislik Çalışmaları ve Teknolojileri Dergisi, 4*(2), 92–98.

Ajibola, A. (2025). *Cloud-Native Reliability Engineering: A comprehensive guide to failure resilience patterns in distributed systems*. *SSRN*. https://doi.org/10.2139/ssrn.5260195

Al-Said Ahmad, A., Al-Qora'n, L. F., & Zayed, A. (2024). Exploring the impact of chaos engineering with various user loads on cloud native applications: An exploratory empirical study. *Computing, 106*(7), 2389–2425. https://doi.org/10.1007/s00607-024-01292-z

Azrajabeen, M. A. (2024). Overview of microservices design patterns' problems, solutions. *Journal of Artificial Intelligence, Machine Learning and Data Science, 3*(1), 1–15. https://doi.org/10.51219/JAIMLD/azrajabeen-mohamed-ali/379

Bailey, B. P., McCarthy, J., Mendoza, W., Veneman, J. L., & Bennett, K. (2022). Measuring resiliency of system of systems using chaos engineering experiments. En *Proceedings of SPIE 12117, Modeling, Simulation, and Visualization for Systems Engineering* (1211707). https://doi.org/10.1117/12.2632779

Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., & Rosenthal, C. (2016). Chaos engineering. *IEEE Software, 33*(3), 35–41. https://doi.org/10.1109/MS.2016.60

Beyer, B., Jones, C., Petoff, J., & Murphy, N. (Eds.). (2016). *Site reliability engineering: How Google runs production systems*. O'Reilly Media.

Camacho, A. S., Cañizares, P., Llana, L., & Núñez, M. (2022). Chaos as a software product line: A platform for improving open hybrid-cloud systems resiliency. *Software: Practice and Experience, 52*(7), 1581–1614. https://doi.org/10.1002/spe.3076

Chen, Z., Goudarzi, M., & Toosi, A. N. (2025, 21 de julio). Resilience evaluation of Kubernetes in cloud-edge environments with chaos engineering. *arXiv*. https://arxiv.org/abs/2507.16109

Dehghani, A., Bloch, C., & Turner, J. (2021). *Chaos engineering with Chaos Toolkit: A practical approach to resilience testing*. O'Reilly Media.

Delnat, W., Truyen, E., Rafique, A., Van Landuyt, D., & Joosen, W. (2018). K8-Scalar: A workbench to compare autoscalers for container-orchestrated database clusters. En *Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '18)* (pp. 33–39). https://doi.org/10.1145/3194133.3194162

Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software, 150*, 77–97.

Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The science of lean software and DevOps – Building and scaling high performing technology organizations*. IT Revolution Press.

Frank, S., Wagner, L., Hakamian, M. A., Straesser, M., & van Hoorn, A. (2022). MiSim: A simulator for resilience assessment of microservice-based architectures. En *2022 IEEE International Conference on Software Quality, Reliability and Security (QRS)* (pp. 1014–1025). IEEE. https://doi.org/10.1109/QRS57517.2022.00105

Giamattei, G., Ettrich, J., Griebler, D., & Kounev, S. (2022). MiSim: A simulator for resilience assessment of microservice-based architectures. En *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)* (pp. 772–781). IEEE. https://doi.org/10.1109/QRS57517.2022.00105

Gremlin. (s. f.). Getting started with chaos engineering on Kubernetes. *Gremlin*. https://www.gremlin.com/kubernetes-chaos-engineeringGremlin Inc. (2023). *Gremlin Chaos Engineering Platform documentation*. https://www.steadybit.com

Gremlin Inc. (2024). Chaos engineering: The history, principles, and practice. https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice

Harness.io. (2024). *Harness Chaos Engineering*. https://harness.ioHe, J. J., Van Bossuyt, D. L., & Pollman, A. (2022). Experimental validation of systems engineering resilience models for islanded microgrids. *Systems, 10*(6), 245. https://doi.org/10.3390/systems10060245

IBM. (2023, 3 de agosto). What is chaos engineering? *IBM*. https://www.ibm.com/think/topics/chaos-engineering

InfoQ. (2015). Netflix' principles of chaos engineering. *InfoQ*. https://www.infoq.com/news/2015/09/netflix-chaos-engineering/

Jernberg, J., & Runeson, P. (2020). Getting started with chaos engineering: Design of an experimentation platform at a global scale. En *2020 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 1–11). https://doi.org/10.1145/3382494.3421464

Jolak, R., Mohamad, M., Avula, R. R., Meek, J., & Åström, A. (2025). SCENE: Guidelines for security chaos engineering based on a systematic literature review.

SSRN preprint. https://doi.org/10.2139/ssrn.5732943

Kesim, D. (2019). *Assessing resilience of software systems by the application of chaos engineering* [Tesis de maestría]. Universität Stuttgart. https://elib.uni-stuttgart.de/items/9193c396-d0e6-4aae-a007-92c63202baf7

Krasnovsky, A. A. (2025). Model discovery and graph simulation: A lightweight gateway to chaos engineering. *arXiv preprint arXiv:2506.11176*. https://doi.org/10.48550/arXiv.2506.11176

Kumar, S., Mohan, P., & Ganga, M. (2023). Enhancing Kubernetes resiliency using LitmusChaos: A cloud-native chaos engineering approach. *International Journal of Advanced Computer Science and Applications, 14*(4), 112–130. https://doi.org/10.14569/IJACSA.2023.0140421

Mailewa, A. B., Akuthota, A., & Mohottalalage, T. M. D. (2025). A review of resilience testing in microservices architectures: Implementing chaos engineering for fault tolerance and system reliability. En *2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC)* (pp. 236–242). IEEE. https://doi.org/10.1109/CCWC62904.2025.10903891

Malik, S., Naqvi, M. A., Astelin, M., & Moonen, L. (2023). CHESS: A framework for evaluation of self-adaptive systems based on chaos engineering. En *2023 IEEE/ACM 18th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (pp. 65–76). IEEE. https://doi.org/10.1109/SEAMS59076.2023.00033

Mendonça, N., Aderaldo, C., Câmara, J., & Garlan, D. (2020). A framework for evaluating microservice architectures' resilience. *Journal of Internet Services and Applications, 11*(1), 1–24.

Mohan, P., Ganga, M., & Desai, R. (2022). LitmusChaos: A framework for cloud-native chaos engineering. *Journal of Cloud Computing, 11*(1), 45–60. https://doi.org/10.1186/s13677-022-00322-0

Naqvi, M. A., Malik, S., Astekin, M., & Moonen, L. (2022). On evaluating self-adaptive and self-healing systems using chaos engineering. En *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)* (pp. 1–10). IEEE. https://doi.org/10.1109/ACSOS55765.2022.00018

Opara, A., Eze, A., Okorie, U., & Okonkwo, C. (2025). Chaos engineering 2.0: A review of AI-driven, policy-guided resilience for multi-cloud systems. *Journal of Computer, Software, and Program, 2*(2), 10–24. https://doi.org/10.69739/jcsp.v2i2.846

Owotogbe, J. S. (2025). Assessing and enhancing the robustness of LLM-based multi-agent systems through chaos engineering. En *2025 IEEE Conference on Artificial Intelligence for Industries (CAIN)* (pp. 1–8). IEEE. https://doi.org/10.1109/CAIN66642.2025.00039

Owotogbe, J. S., Kumara, I., van den Heuvel, W.-J., & Tamburri, D. A. (2024). Chaos engineering: A multi-vocal literature review. *arXiv preprint arXiv:2412.01416*. https://doi.org/10.48550/arXiv.2412.01416

Owotogbe, J., Kumara, I., Di Nucci, D., Tamburri, D. A., & van den Heuvel, W.-J. (2025). Chaos engineering in the wild: Findings from GitHub. *arXiv preprint arXiv:2505.13654*. https://doi.org/10.48550/arXiv.2505.13654

Platform Engineers. (2024). Circuit breaker and bulkhead patterns for resilience. *Medium*. https://medium.com/@platform.engineers/circuit-breaker-and-bulkhead-patterns-for-resilience-2a8ae88ac717

Poltronieri, F., Tortonesi, M., & Stefanelli, C. (2022). A chaos engineering approach for improving the resiliency of IT services configurations. En *2022 IEEE/IFIP Network Operations and Management Symposium (NOMS)* (pp. 1–6). IEEE. https://doi.org/10.1109/NOMS54207.2022.9789887

Poulin, C., & Kane, M. (2021). Infrastructure resilience curves: Performance measures and summary metrics. *Reliability Engineering & System Safety, 216*, 107926. https://doi.org/10.1016/j.ress.2021.107926

Poulin, A., & Kane, A. (2021). A systematic review of resilience metrics. *Safety Science, 141*, 105–116.

Red Hat Developer. (2024). It takes more than a circuit breaker to create a resilient application. https://developers.redhat.com/blog/2017/05/16/it-takes-more-than-a-circuit-breaker-to-create-a-resilient-application

Rosenthal, C., & Jones, N. (2020). *Chaos engineering: System resiliency in practice*. O'Reilly Media.

Sahoo, B. (2025). Enhancing reliability and performance of microservices using chaos engineering: A comprehensive study. *Journal of Computer Science and Technology Studies, 7*(9), 672–689. https://doi.org/10.32996/jcsts.2025.7.9.60

Shekhar, G. (2024). Microservices design patterns for cloud architecture. *SSRG International Journal of Computer Science and Engineering*, *11*(9), 1–7. https://doi.org/10.14445/23488387/IJCSE-V11I9P101

Simonsson, J., Zhang, L., Morin, B., Baudry, B., & Monperrus, M. (2021). Observability and chaos engineering on system calls for containerized applications in Docker. *Future Generation Computer Systems, 125*, 770–784. https://doi.org/10.1016/j.future.2021.04.001

System Design Newsletter. (2024). How Netflix uses chaos engineering to create resilient systems. *System Design Newsletter*. https://newsletter.systemdesign.one/p/chaos-engineering

Tang, J., Han, S., Wang, J., He, B., & Peng, J. (2023). A comparative analysis of performance-based resilience metrics via a quantitative–qualitative combined

approach: Are we measuring the same thing? *International Journal of Disaster Risk Science, 14*(3), 405–421. https://doi.org/10.1007/s13753-023-00519-5

Tang, X., Zhang, H., & Li, Y. (2023). On the comparison of time-series-based resilience metrics. *Reliability Engineering & System Safety, 230*, 108–146.

Thompson, A., Johnson, E., Smith, V., & Adelusi, J. B. (2022). Scalability and resilience in microservices: Patterns and best practices. *ResearchGate*. https://www.researchgate.net/publication/392125739_Scalability_and_Resilience_in_Microservices_Patterns_and_Best_Practices

Torkura, K. A., Sukmana, M. I., Cheng, F., & Meinel, C. (2020). CloudStrike: Chaos engineering for security and resiliency in cloud infrastructure. *IEEE Access, 8*, 123044–123060. https://doi.org/10.1109/ACCESS.2020.3007338

Yadav, R. (2024). Harnessing chaos: The role of chaos engineering in cloud applications and impacts on site reliability engineering. *International Journal of Computer Trends and Technology, 72*(6), 1–9. https://doi.org/10.14445/22312803/IJCTT-V72I6P104

Yang, T., Lee, C., Shen, J., Su, Y., Feng, C., Yang, Y., & Lyu, M. R. (2024). MicroRes: Versatile resilience profiling in microservices via degradation dissemination indexing. En *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)* (pp. 325–337). https://doi.org/10.1145/3650212.3652131

Yodo, N., & Wang, P. (2016). Engineering resilience quantification and system design implications: A literature survey. *Journal of Mechanical Design, 138*(11), 111408. https://doi.org/10.1115/1.4034223

Zhang, L., Morin, B., Haller, P., Baudry, B., & Monperrus, M. (2021). A chaos engineering system for live analysis and falsification of exception-handling in the JVM. *IEEE Transactions on Software Engineering, 47*(11), 2424–2446. https://doi.org/10.1109/TSE.2019.2954871

Zhang, L., Morin, B., Baudry, B., & Monperrus, M. (2022). Maximizing error injection realism for chaos engineering with system calls. *IEEE Transactions on Dependable and Secure Computing, 19*(4), 2695–2708. https://doi.org/10.1109/TDSC.2021.3069715

Zhou, J., Zhang, Y., Wang, S., & Du, X. (2022). Chaos Mesh: A cloud-native chaos engineering platform for Kubernetes. *Journal of Systems Architecture, 134*, 102768. https://doi.org/10.1016/j.sysarc.2022.102768

Chaos Mesh. (s. f.). *Chaos Mesh overview*. https://chaos-mesh.org/docs/Docker. (s. f.). Docker overview. En *Docker documentation*. https://docs.docker.com/get-started/docker-overview/

Grafana Labs. (s. f.). Grafana Cloud: Introduction. En *Grafana Cloud documentation*. https://grafana.com/docs/grafana-cloud/introduction/

Grafana Labs. (s. f.). k6 load testing documentation. En *Grafana k6 documentation*. https://grafana.com/docs/k6/latest/Kafka. (s. f.). Apache Kafka documentation. https://kafka.apache.org/documentation/#gettingStarted

Kubernetes. (s. f.). Concepts: Overview. En *Kubernetes documentation*. https://kubernetes.io/docs/concepts/overview/

LitmusChaos. (s. f.). What is Litmus? En *LitmusChaos documentation*. https://docs.litmuschaos.io/docs/next/introduction/what-is-litmus

PostgreSQL Global Development Group. (s. f.). About PostgreSQL. https://www.postgresql.org/about/

Postman. (s. f.). What is Postman? https://www.postman.com/product/what-is-postman/

Prometheus Authors. (s. f.). Overview. En *Prometheus documentation*. https://prometheus.io/docs/introduction/overview/

The Apache JMeter Project. (s. f.). Apache JMeter. https://jmeter.apache.org/

Spring Cloud. (2024). Spring Cloud CircuitBreaker: Bulkhead pattern supporting. En *Spring Cloud CircuitBreaker documentation*. https://docs.spring.io/spring-cloud-circuitbreaker/reference/spring-cloud-circuitbreaker-resilience4j/bulkhead-pattern-supporting.html

Steadybit GmbH. (2023). *Steadybit documentation*. https://www.steadybit.com

Hernández-Sampieri, R., & Mendoza, C. P. (2018). *Metodología de la investigación. Las rutas cuantitativa, cualitativa y mixta* (2.ª ed.). McGraw-Hill.

Martín, R., & Grande, A. (2022). *Introducción a Chaos Engineering* [e-book]. Paradigma Digital. https://www.paradigmadigital.com/dev/ebook-introduccion-chaos-engineering/

Mendonça, N., Aderaldo, C., Câmara, J., & Garlan, D. (2020). A framework for evaluating microservice architectures' resilience. *Journal of Internet Services and Applications, 11*(1), 1–24.

Poulin, A., & Kane, A. (2021). A systematic review of resilience metrics. *Safety Science, 141*, 105–116.

Tang, X., Zhang, H., & Li, Y. (2023). On the comparison of time-series-based resilience metrics. *Reliability Engineering & System Safety, 230*, 108–146.

Anexos

Anexo 1. Cronograma

Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
	ChaosProject	**115 días**	**mar 1/7/25**	**lun 8/12/25**	
	Desarrollo de aplicaciones	**25 días**	**mar 1/7/25**	**lun 4/8/25**	
	Creacion de entorno controlado	**12 días**	**mar 5/8/25**	**mié 20/8/25**	**2**
	Ejecutar carga constante	25 días	jue 21/8/25	mié 24/9/25	5
	Simulacion de escenarios de prueba	25 días	jue 21/8/25	mié 24/9/25	5
	Captura de métricas	**20 días**	**jue 25/9/25**	**mié 22/10/25**	**12**
	Comparacion de resultados	**3 días**	**jue 23/10/25**	**lun 27/10/25**	**13**
	Redaccion de informe de investigacion	30 días	mar 28/10/25	lun 8/12/25	16
	Elaboracion de articulo	30 días	mar 28/10/25	lun 8/12/25	16