POSGRADOS

Maestría en

Software con mención en Diseño de Arquitectura de Sistemas

RPC-SO-34-NO.778-2021

Opción de Titulación:

Proyecto de titulación con componentes de investigación aplicada y/o de desarrollo

Tema:

Implementación del patrón Circuit Breaker en microservicios con Spring Boot para evitar fallos en cascada a través de una simulación de consumo de servicios externos

Autor(es)

Verónica Elizabeth Vicente Cueva

Director:

Rodrigo Tufiño Cárdenas

QUITO – Ecuador
2025

Autor(es):

Verónica Elizabeth Vicente Cueva
Ingeniera en Computación Gráfica
Candidata a Magíster en Software por la Universidad Politécnica Salesiana – Sede Quito.
vvicente@est.ups.edu.ec

Dirigido por:

Rodrigo Efraín Tufiño Cárdenas
Ingeniero en Sistemas
Master Universitario en Ciencias y Tecnologías de la Computación
Master Universitario en Software Libre
rtufino@ups.edu.ec

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra para fines comerciales, sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual. Se permite la libre difusión de este texto con fines académicos investigativos por cualquier medio, con la debida notificación a los autores.

DERECHOS RESERVADOS
2025 © Universidad Politécnica Salesiana.
QUITO– ECUADOR – SUDAMÉRICA
Verónica Elizabeth Vicente Cueva
Implementación del Patrón Circuit Breaker en Microservicios con Spring Boot para evitar fallos en cascada a través de una simulación de consumo de servicios externos

DEDICATORIA

Dedico este trabajo a todas las personas que, a pesar de las dificultades cotidianas, siguen luchando con valentía por sus sueños y metas.

El camino puede parecer largo y lleno de obstáculos, pero cuando creemos en nosotros mismos y perseveramos con determinación, todo es posible.

Como bien nos enseña Edith Eger:

“Puedes vivir en la prisión del pasado o puedes dejar que el pasado sea el trampolín que te ayude a alcanzar la vida que deseas.”

AGRADECIMIENTO

Doy gracias a Dios, por concederme la salud, la fortaleza y las oportunidades necesarias para avanzar en mi formación académica y personal.

A mis padres y hermanos, por su amor, paciencia y apoyo incondicional, que han sido fundamentales en cada etapa de este camino. Su confianza en mí ha sido una fuente constante de motivación.

Extiendo mi agradecimiento a mis profesores, quienes, con dedicación y compromiso, contribuyeron significativamente a mi proceso de aprendizaje.

Finalmente, agradezco a mi tutor de tesis, por su valiosa guía y acompañamiento durante el desarrollo de este trabajo. Su experiencia y consejos fueron clave para la culminación de este proyecto académico.

Tabla de Contenido

Resumen 10

Abstract 11

1 Introducción 12

2 Determinación del Problema 14

3 Marco teórico referencial 15

3.1. Interdependencia entre microservicios y fallos en cascada 15

3.1.1 Fallos en Cascada 17

3.2. Resiliencia y Tolerancia de fallos 19

3.2.1 Patrones de resiliencia 21

3.3. Patrón Circuit Breaker 23

3.4. Integración de resilience4j en aplicaciones Spring Boot 25

3.4.1 Spring Boot 26

3.4.2 Resilience4j 26

3.5. Simulación de condiciones de red 28

3.6. Pruebas de rendimiento 30

3.6.1. Pruebas de carga 30

3.6.2. Pruebas de estrés 31

3.6.3. Pruebas de pico 32

3.7. Monitoreo 33

3.7.1. Métricas de rendimiento de la JVM 33

3.7.2. Prometheus 36

3.7.3. Grafana 36

4. Materiales y metodología 37

4.1. Definición de métricas 37

4.2. Caso de estudio 38

4.3. Entorno experimental 42

4.4. Implementación del Patrón Circuit Breaker 45

4.5. Escenarios de prueba 50

4.5.1. E1: Línea Base 51

4.5.2. E2: Sin Circuit Breaker con latencia 53

4.5.3. E3: Circuit Breaker con latencia 54

4.5.4. E4: Circuit Breaker ante errores HTTP 5xx 55

4.5.5. E5: Circuit Breaker con ajuste de parámetros 56

5. Resultados y discusión 59

5.1. Resultados E1: Línea Base 59

5.2. Resultados E2: Sin Circuit Breaker con latencia 64

5.3. Resultados E3: Circuit Breaker con latencia 67

5.4. Resultados E4: Circuit Breaker ante errores HTTP 5xx 70

5.5. Resultados E5: Circuit Breaker con ajuste de parámetros 73

5.6. Discusión 78

6. Conclusiones 80

Referencias 82

Índice de tablas

Tabla 1 Las cuatro señales doradas para el monitoreo 20
Tabla 2 Parámetros de Configuración de Resilience4j para el Estado CERRADO 27
Tabla 3 Parámetros de Configuración de Resilience4j para el Estado ABIERTO 27
Tabla 4 Parámetros de Configuración de Resilience4j para el Estado SEMIABIERTO 27
Tabla 5 Propiedades configurables de Toxiproxy 29
Tabla 6 Estados de los hilos en Java 35
Tabla 7 Métricas e Indicadores de Resiliencia 37
Tabla 8 Stack Tecnológico Usado en la Implementación del Circuit Breaker 43
Tabla 9 Características de la máquina para ejecución de pruebas 44
Tabla 10 Parámetros de Configuración del Circuit Breaker 49
Tabla 11 Resumen de los Escenarios de Prueba del Circuit Breaker 50
Tabla 12 Umbrales para detección de llamada lentas 54
Tabla 13 Umbrales para detección de llamada lentas 56
Tabla 14 Variación de parámetros del Circuit Breaker 57
Tabla 15 Resultado de las métricas del Escenario 1 59
Tabla 16 Resultado de las métricas del Escenario 2 64
Tabla 17 Resultado de las métricas del Escenario 3 67
Tabla 18 Resultado de las métricas del Escenario 4 70
Tabla 19 Resultado de las métricas del Escenario 5 73

Índice de Figuras

Figura 1 Arquitectura de referencia basada en microservicios.................................... 16
Figura 2 Esquema de interdependencia entre microservicios..................................... 17
Figura 3 Fallos en cascada causado por alta latencia.. 18
Figura 4 Dependencias internas y externas de un microservicio................................. 21
Figura 5 Esquema del funcionamiento del Patrón Circuit Breaker 25
Figura 6 Arquitectura de ToxiProxy.. 29
Figura 7 Pruebas de carga vs pruebas de estrés 32
Figura 8 Pruebas de pico 33
Figura 9 Ciclo de vida de un hilo en Java 35
Figura 10 Arquitectura General del Sistema 40
Figura 11 Diagrama de secuencia del funcionamiento del sistema............................. 41
Figura 12 Diagrama de estudio del patrón Circuit Breaker utilizando monitoreo......... 42
Figura 13 Diagrama de clases del microservicio eletric-bill-service.............................. 46
Figura 14 Configuración de dependencias en el archivo build.gradle 47
Figura 15 Transacciones por segundo del microservicio bill-management-service en el Escenario 1.. 60
Figura 16 Uso de CPU del Escenario 1 61
Figura 17 JVM Heap en el Escenario 1.. 62
Figura 18 Estados del Hilo en el Escenario 1 63
Figura 19 Transacciones por segundo del microservicio bill-management-service en el Escenario 2.. 65
Figura 20 Estados del Hilo en el Escenario 2 66
Figura 21 Estados del Circuit Breaker en el Escenario 3.. 68
Figura 22 Estados del Hilo en el Escenario 3 69
Figura 23 Estados del Circuit Breaker en el Escenario 4.. 71
Figura 24 Estados de hilos en ejecución... 71
Figura 25 Estados del Circuit Breaker del Escenario 5 – umbral 20 % 74
Figura 26 Estado del Hilo en el Escenario 5 - Umbral 20%.. 75
Figura 27 Estados del Circuit Breaker del Escenario 5 – 50 %....................................... 75
Figura 28 Estado del Hilo en el Escenario 6 - Umbral 50%.. 76
Figura 29 Estados del Circuit Breaker del Escenario 5 - 80% 77
Figura 30 Estado del Hilo en el Escenario 5 - Umbral 80%.. 77

Implementación del patrón Circuit Breaker en microservicios con Spring Boot para evitar fallos en cascada a través de una simulación de consumo de servicios externos

Autor(es):

Verónica Elizabeth Vicente Cueva

Resumen

El presente trabajo aborda la problemática de los fallos en cascada en arquitecturas basadas en microservicios, ocasionados por dependencias externas. Esta anomalía puede provocar degradaciones en la disponibilidad y el rendimiento de los sistemas distribuidos, afectando la continuidad operativa y la experiencia de los usuarios finales.

Como solución se implementa el patrón de resiliencia Circuit Breaker utilizando el framework *Spring Boot* y la librería *Resilience4j*. El estudio se desarrolla sobre un caso del sector financiero, que consiste en la consulta del valor de facturas del servicio eléctrico a través de una API externa. Para analizar el comportamiento del sistema, se definen tres escenarios experimentales: operación normal, degradación sin mecanismos de resiliencia y aplicación del patrón Circuit Breaker.

La metodología combina pruebas de carga con inyección controlada de fallos mediante k6 y *Toxiproxy*, junto con un monitoreo detallado de métricas en tiempo real utilizando *Prometheus* y *Grafana*.

Este trabajo aporta evidencia técnica y metodológica que respalda la adopción de patrones de resiliencia en arquitecturas de microservicios, y constituye una guía práctica para arquitectos de software interesados en diseñar sistemas tolerantes a fallos.

Palabras clave:

Arquitectura de microservicios, resiliencia, tolerancia a fallos, Circuit Breaker, Spring Boot, Resilience4j, pruebas de carga, observabilidad.

ABSTRACT

This paper explores the issue of cascading failures in microservices based architectures, particularly those originating from external dependencies. This anomaly can lead to significant degradations in the availability and performance of distributed systems, consequently impacting operational continuity and end-user experience.

As a solution, the Circuit Breaker resilience pattern is implemented utilizing the Spring Boot framework and the Resilience4j library. The study is conducted within a financial sector use case, involving the retrieval of electricity bill values via an external API. To analyze system behavior, three experimental scenarios are defined: normal operation, degradation without resilience mechanisms, and the application of the Circuit Breaker pattern.

The methodology integrates load testing with controlled fault injection using k6 and Toxiproxy, coupled with detailed real-time metric monitoring employing Prometheus and Grafana. This work provides technical and methodological evidence supporting the adoption of resilience patterns in microservices architectures and serves as a practical guide for software architects interested in designing fault-tolerant systems.

Keywords:

Microservices architecture, resilience, fault tolerance, Circuit Breaker, Spring Boot, Resilience4j, load testing, observability.

1 Introducción

La aceleración de la transformación digital en el sector financiero ha impulsado la migración de sistemas monolíticos hacia arquitecturas basadas en microservicios, con el objetivo de lograr plataformas más escalables, flexibles y mantenibles. Este cambio ha permitido ofrecer a sus clientes servicios en línea como banca móvil, pagos digitales, transferencias interbancarias, consultas de saldos y pagos de facturas a través de múltiples canales digitales consolidando un ecosistema financiero cada vez más interconectado. (Finnovista and Banco Interamericano de Desarrollo and BID Invest, 2024)

Sin embargo, esta evolución tecnológica ha traído consigo nuevos desafíos relacionados con la resiliencia, entendida como la capacidad de un sistema para recuperarse frente a fallos manteniendo su funcionamiento o restaurándolo de forma autónoma (Newman, Building Microservices, Designing Fine-Grained Systems, 2021).

En arquitecturas basadas en microservicios, las aplicaciones dependen frecuentemente de APIs y servicios de terceros como pasarelas de pago, proveedores de facturación, plataformas de autenticación sobre los cuales las entidades no tienen control directo. La degradación o caída de estas dependencias puede provocar fallos cascada afectando gravemente la operación general. Como señala (Nygard, 2018), este tipo de fallos pueden ocasionar un mayor latencia o errores inesperados como también agotar recursos críticos del sistema, como la sobrecarga de bases de datos, provocando potencialmente a una indisponibilidad total de la aplicación.

Ante esa problemática, Michael T. Nyrad popularizó el patrón Circuit Breaker, un mecanismo de protección para detectar fallos persistentes, aislar los componentes problemáticos y prevenir que los errores individuales se propaguen a todo el sistema (Nygard, 2018) .Este patrón actúa como una barrera de contención que permite

mantener la estabilidad operativa incluso cuando una dependencia externa se encuentra en estado degradado.

Este proyecto constituye un aporte práctico para arquitectos de software y equipos técnicos de instituciones financieras que enfrentan retos similares, ofreciendo una validación técnica documentada de una estrategia para mitigar fallos en cascada y fortalecer la resiliencia de sistemas críticos. Además, se integra un entorno de monitoreo con Prometheus y Grafana, y se aplican pruebas de carga e inyección controlada de fallos para analizar métricas como latencia, errores y uso de recursos.

2 Determinación del Problema

La resiliencia tecnológica es un factor clave para la continuidad operativa en sistemas financieros. Las infraestructuras digitales deben diseñarse con tolerancia a fallos, aislamiento y mecanismos de recuperación para evitar fallas en cascada y garantizar la disponibilidad de los servicios (Dasari, 2025).

El creciente uso de aplicaciones móviles y plataformas digitales en Ecuador ha aumentado 25 veces desde 2019, evidencia que las operaciones financieras dependen cada vez más de servicios digitales interconectados (IT ahora, 2024) . Esta tendencia genera una alta demanda de disponibilidad, baja latencia y estabilidad en los sistemas, requisitos que solo pueden cumplirse mediante arquitecturas resilientes y bien diseñadas.

En arquitecturas basadas en microservicios, esta interdependencia introduce riesgos específicos. Cada microservicio suele depender de otros para completar operaciones críticas, como autenticación, validación de pagos o consulta de información mediante APIs externas. Si alguno de estos servicios falla, está en mantenimiento o responde con lentitud, la interrupción puede propagarse a otros servicios dependientes, provocando lo que se conoce como fallas en cascada (Richardson, 2018).

A pesar de la relevancia del problema, muchas implementaciones de microservicios no incorporan desde el inicio mecanismos de tolerancia a fallos, provocando arquitecturas vulnerables. En ese contexto, surge la pregunta de investigación: ¿Cómo prevenir la propagación de fallos en cascada en una arquitectura de microservicios ante la degradación o caída de servicios críticos?

El patrón Circuit Breaker, surge como una estrategia para abordar este problema, ya que detecta detectar fallos persistentes en dependencias externas y evita que desestabilicen todo el sistema.

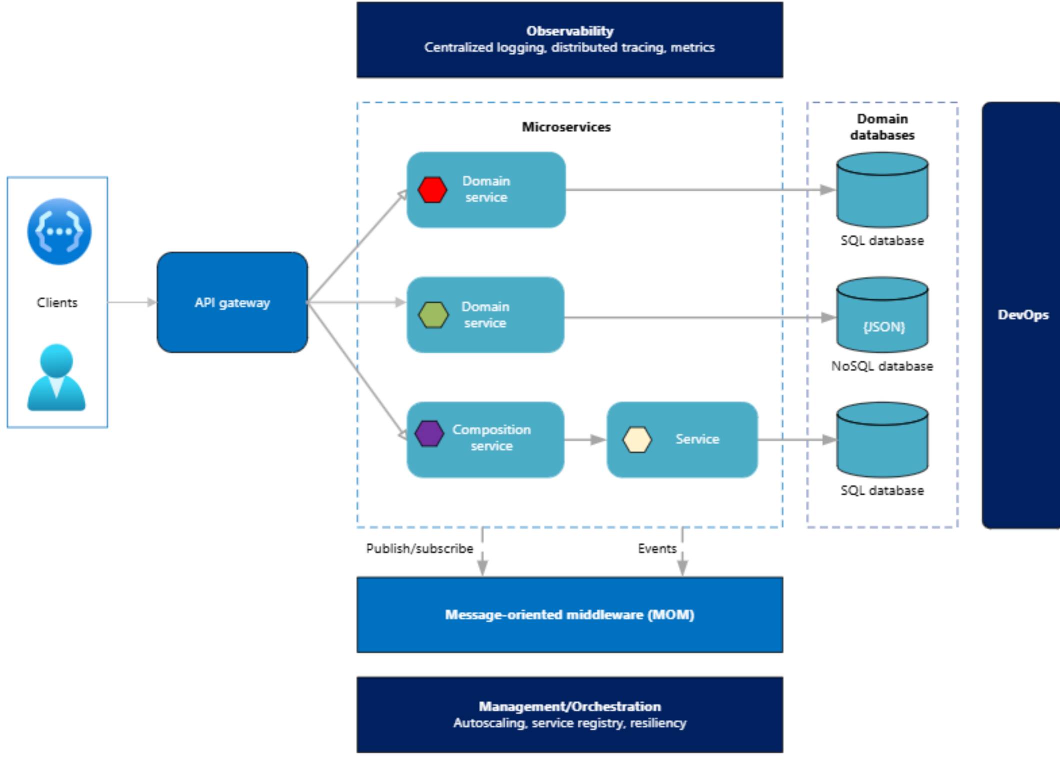
3 Marco teórico referencial

En este capítulo, se presentan los conceptos fundamentales que respaldan la implementación del patrón Circuit Breaker en microservicios desarrollados con *Spring Boot*. Se analizan los principios de interdependencia entre servicios, los riesgos asociados a fallos en cascada y la importancia de la resiliencia y tolerancia a fallos en sistemas distribuidos. Finalmente, se examina el funcionamiento del patrón Circuit Breaker y su relación con la librería *Resilience4j* como mecanismo práctico para evitar degradaciones en sistemas críticos.

3.1. Interdependencia entre microservicios y fallos en cascada

Las arquitecturas basadas en microservicios han ganado relevancia debido a su capacidad para ofrecer escalabilidad, flexibilidad y mantenibilidad. En este enfoque, el sistema se compone de varios servicios pequeños, cada uno encargado de una función específica. Esta distribución permite desplegar, escalar y actualizar componentes sin afectar al resto del sistema. (Sithec, 2024). En la Figura 1, se muestra un diagrama de una arquitectura de microservicios.

Figura 1 Arquitectura de referencia basada en microservicios

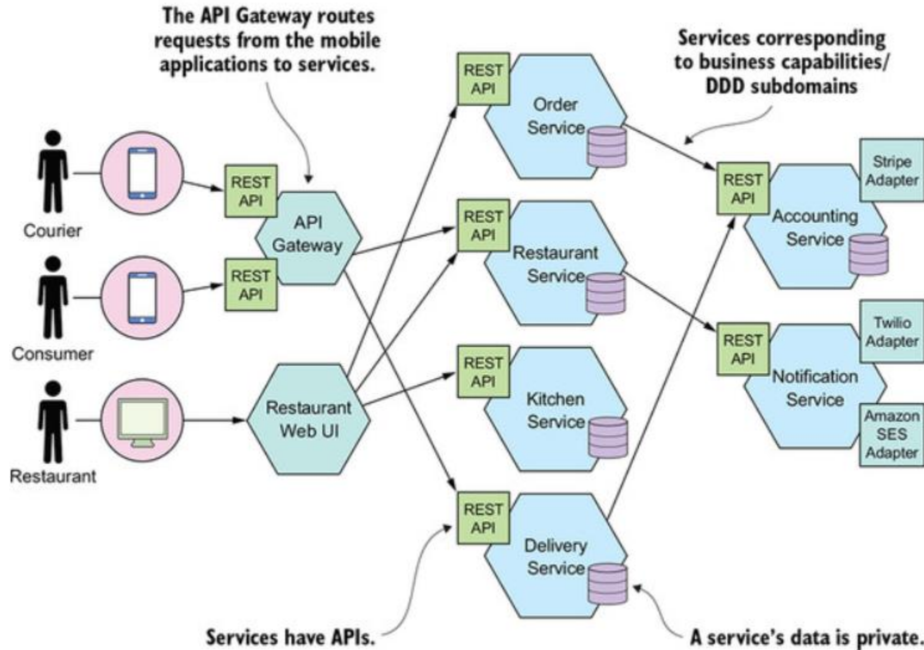


Nota. En esta arquitectura de microservicios, los clientes acceden a los servicios a través de un API Gateway, que enruta las solicitudes hacia distintos servicios, mientras que cada microservicio gestiona su propia base de datos, garantizando la independencia y el aislamiento de los datos. Tomada de (Microsoft, 2025)

No obstante, a pesar de su independencia funcional, los microservicios necesitan interactuar entre sí o con APIs externas para obtener información o coordinar procesos. Esta comunicación es esencial para el correcto funcionamiento del sistema, ya que permite que los diferentes componentes trabajen de manera integrada.

En algunos casos, como se observa en la Figura 2, esta interacción se realiza de forma síncrona, lo que significa que cuando un microservicio realiza una solicitud a otro debe esperar su respuesta para continuar con su ejecución. Este enfoque requiere que ambos microservicios estén disponibles al mismo tiempo para que la comunicación sea exitosa. Esta dependencia puede implicar que, si uno de los servicios no responde o presenta fallos, se propague los fallos a todo el sistema. (Indrasiri & Siriwardena, 2018).

Figura 2 Esquema de interdependencia entre microservicios



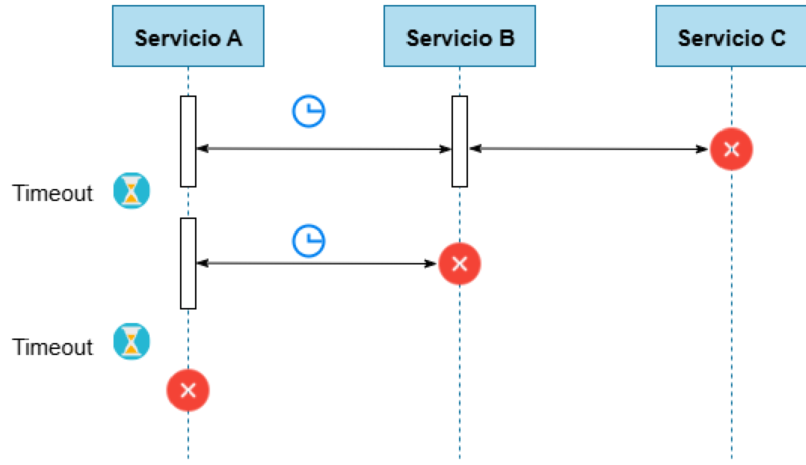
Nota. En este diagrama, se evidencia cómo algunos servicios dependen de otros para procesar pedidos, preparar comida y realizar entregas. Estas dependencias evidencian que, en comunicaciones síncronas, la falla de un servicio puede afectar el funcionamiento de los demás. Tomado de (Newman, Building Microservices, Designing Fine-Grained Systems, 2021)

3.1.1 Fallos en Cascada

Un fallo en cascada se define como un error que crece exponencialmente a través del sistema como resultado de un bucle de retroalimentación positiva. Desde la perspectiva de la Ingeniería de Fiabilidad de Sitios (SRE), la causa raíz de muchas interrupciones a gran escala es una sobrecarga que se propaga en cascada (Beyer, Jones, Petoff, & Niall, 2016).

Cuando un servicio falla, su carga es redirigida a las réplicas restantes, sin embargo, sino están aprovisionadas para soportar tráfico adicional, se sobrecargan y también fallarán, ocasionando un colapso, tal como se muestra en la Figura 3.

Figura 3 Fallos en cascada causado por alta latencia



Para analizar esta propagación es importante hacer la distinción:

"Un fallo (fault) es un componente que se desvía de su especificación, mientras que una falla (failure) ocurre cuando el sistema en conjunto deja de proveer el servicio." (Kleppmann, 2017)

La afectación de los fallos se puede evidenciar en la infraestructura a través del agotamiento de recursos. Autores cómo (Nygard, 2018) y (Beyer, Jones, Petoff, & Niall, 2016) han documentado cómo la falta de CPU, memoria o hilos de ejecución puede desencadenar la cascada:

- **Agotamiento de CPU:** Ralentiza todas las solicitudes, aumenta la latencia y satura las colas de peticiones. A menudo, una alta utilización de la CPU es un síntoma de problemas de memoria, donde el sistema entra en un ciclo de recolección de basura constante, consumiendo recursos.
- **Agotamiento de Memoria:** Un mayor número de solicitudes en curso consume más RAM. Esto puede llevar a que los procesos sean terminados por el sistema operativo o, en entornos como Java, a una "espiral de la

muerte de la recolección de basura (Garbage Collector GC)", donde el sistema gasta más tiempo liberando memoria que procesando trabajo útil

- **Agotamiento de Hilos:** Ocurre cuando todos los hilos de un pool de recursos (como un pool de conexiones) están bloqueados, esperando respuestas de sistemas externos que nunca llegan. Un servicio sin hilos disponibles no puede aceptar nuevas conexiones, lo que resulta en fallos inmediatos para las nuevas peticiones.

Frente a este escenario, el patrón Circuit Breaker aparece como una solución de estabilidad, diseñada para detectar fallos persistentes, aislar el componente problemático y prevenir activamente la propagación de fallos en cascada.

3.2. Resiliencia y Tolerancia de fallos

La resiliencia es un principio fundamental de los microservicios, y se refiere a la capacidad que cada componente para recuperarse ante fallos, sin afectar el rendimiento ni la disponibilidad general (Kumar, 2024).

En un entorno productivo, los microservicios deben ser tolerante a fallos y estar preparados para posibles catástrofes. Dado que los fallos son inevitables, es importante garantizar la disponibilidad en todo el ecosistema de microservicios mediante una planificación cuidadosa y la preparación para gestionar fallos de manera efectiva (Fowler, 2016).

Para crear aplicaciones resilientes los componentes deben consideras las siguientes áreas (Badman & Kosinski, 2025):

- **Tolerancia a fallos:** Permite que el sistema siga funcionando cuando ocurre un fallo mediante sistemas de respaldo o reparando los problemas sin que el usuario note el problema.

- **Degradación elegante:** Cuando ciertos componentes fallan, el sistema debe seguir funcionando de forma parcial o con la funcionalidad básica en lugar de colapsar por completo.
- **Autorrecuperación:** El sistema puede recuperarse automáticamente, reiniciando servicios o reintentando operaciones.
- **Aislamiento y contención:** Se evita la propagación de fallos mediante el aislamiento de servicios.
- **Redundancia y conmutación por error:** Se duplican servicios para garantizar de que otro servicio pueda tomar el control si falla una instancia, asegurando la continuidad del servicio.
- **Monitoreo y observabilidad:** Proporcionan una visibilidad del estado y el rendimiento de los microservicios, lo que facilita la identificación de problemas de forma temprana y su solución antes de que vuelvan a suceder.

Existen cuatro métricas clave para monitorear sistemas orientados al usuario conocidas como "The Four Golden Signals", que son esenciales para detectar problemas y mantener la salud del sistema. (Beyer, Jones, Petoff, & Niall, 2016)

Tabla 1 Las cuatro señales doradas para el monitoreo

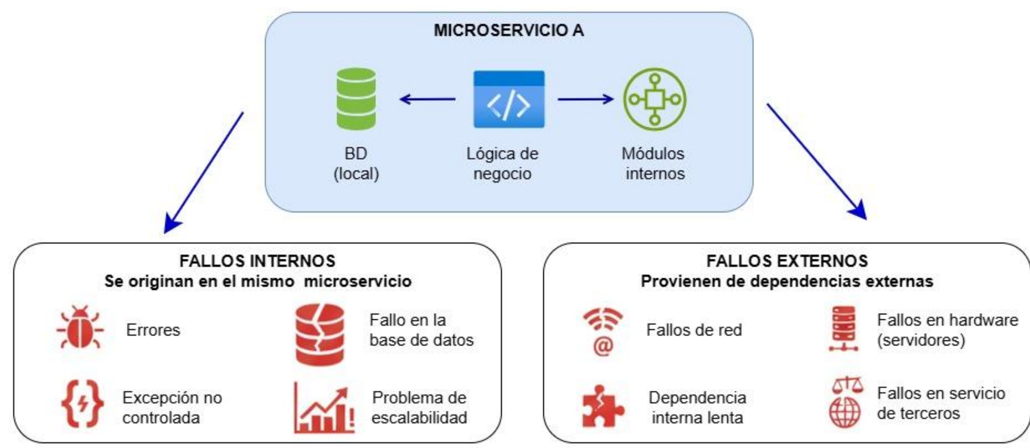
Métrica	**Descripción**
Latencia	Tiempo que tarda una solicitud en ser atendida. Se debe diferenciar entre solicitudes exitosas y fallidas, ya que los errores rápidos pueden distorsionar el promedio.
Tráfico	Volumen de actividad que recibe el sistema, como solicitudes HTTP por segundo o sesiones activas, dependiendo del tipo de servicio.
Errores	Tasa de solicitudes fallidas, ya sea por códigos de error explícitos como HTTP 500, respuestas incorrectas o tiempos de respuesta demasiado altos.
Saturación	Se enfoca en los recursos más críticos CPU, memoria, etc. y en señales tempranas como aumentos en la latencia. También considera predicciones, como el agotamiento de almacenamiento

El diseño de microservicios tolerantes a fallos requiere la identificación de fallos tanto internos como externos, tal como se muestra en la Figura 4. Los fallos internos

se originan en el mismo microservicio afectando al código, configuración o recursos locales. Por otra parte, los fallos externos provienen de dependencias de infraestructura (red, servidores) o de otros de otros sistemas (servicios de terceros, bases de datos compartidas) con los que interactúa el microservicio.

Una estrategia de resiliencia requiere la realización de pruebas incluyendo pruebas unitarias, de integración, de extremo a extremo, pruebas de carga para evaluar el rendimiento bajo estrés, y pruebas de caos para verificar la capacidad de recuperación del sistema ante fallos inesperados y deliberadamente inducidos. (Fowler, 2016).

Figura 4 Dependencias internas y externas de un microservicio



Particularmente, en el caso del patrón Circuit Breaker, la latencia y la tasa de errores son métricas clave para evaluar la salud del sistema. Estas métricas permiten detectar a tiempo cuando un componente empiece a fallar, con la finalidad de prevenir fallos en cascada y mantener la estabilidad general del sistema.

3.2.1 Patrones de resiliencia

Los patrones de resiliencia son estrategias que ayudan a fortalecer los sistemas manteniendo su disponibilidad, y la estabilidad ante fallos, comportamientos inesperados o alta demanda. A continuación, se detallan algunos de los principales:

- **Retry:** Este patrón se utiliza para reintentar automáticamente una operación que ha fallado temporalmente debido a errores transitorios, como los códigos 503 (Servicio no disponible) o 504 (Tiempo de espera agotado). En lugar de abortar la operación de inmediato, el sistema vuelve a intentarla tras un intervalo definido. Este patrón es especialmente útil cuando los fallos se deben a condiciones momentáneas que pueden resolverse con un nuevo intento (Newman, Building Microservices, Designing Fine-Grained Systems, 2021).
- **Timeout:** Permite definir un tiempo máximo de espera para recibir una respuesta de un servicio. Esta configuración evita que el sistema quede bloqueado si un servicio no responde, ya que impide que una solicitud espere indefinidamente (Nygard, 2018).

  Los timeouts se aplican a las solicitudes salientes y deben configurarse tomando como referencia los tiempos de respuesta esperados de los servicios dependientes.

- **Bulkhead:** Consiste en aislar recursos de concurrencia —como hilos o conexiones— para cada servicio mediante pools fijos o semáforos. Esto evita la saturación total del sistema y permite que servicios críticos continúen operando incluso si otros presentan fallos o comportamientos lentos, mejorando así la resiliencia global (Newman, Building Microservices, Designing Fine-Grained Systems, 2021).
- **Circuit Breaker:** Un mecanismo que intercepta las llamadas a un servicio remoto y, tras detectar que dicho servicio falla repetidamente, "abre el circuito" para dejar de enviarle peticiones y proteger el sistema. Luego analiza el estado del servicio y, cuando éste se recupera, "cierra el circuito" nuevamente para reanudar las peticiones (Blancarte, 2018).

Aunque estos patrones pueden combinarse, el presente trabajo se enfoca específicamente en el patrón Circuit Breaker, dada su eficacia ante latencias elevadas y fallas repetitivas en dependencias remotas.

3.3. Patrón Circuit Breaker

El patrón Circuit Breaker fue popularizado por Michael Nygard en su libro Release It, es un patrón que actúa como un proxy para monitorear las llamadas a servicios dependientes. Cuando detecta un número excesivo de fallos, "abre el circuito" y evita nuevas llamadas hasta que el servicio problemático se recupere. Durante este periodo, el sistema puede responder con mensajes predeterminados o errores controlados, lo que ayuda a prevenir fallos en cascada (Nygard, 2018) .

Una vez que el servicio se recupera, el patrón permite enviar solicitudes de forma gradual, de esta forma, se mantiene la disponibilidad del sistema y se mejora la experiencia de usuario.

El objetivo principal del patrón Circuit Breaker es gestionar las fallas de los componentes con los que interactúa, como otro microservicio, una base de datos o una API externa, y evitar que estos errores provoquen un colapso total del sistema (Richards, 2022)

La forma más sencilla de entender este patrón es pensar en un interruptor automático (breaker) en el tablero eléctrico de una casa.

- **Funcionamiento normal:** La electricidad fluye sin problemas, y los electrodomésticos funcionan.
- **Sobrecarga:** Si el flujo de electricidad es mayor que un cierto valor umbral, el breaker interrumpirá el flujo y protegerá los aparatos detrás de él de daños
- **Reactivación:** Una vez que el problema se resuelve (desconectas algunos aparatos), puedes volver a subir el interruptor para restaurar la energía.

De manera análoga, el patrón Circuit Breaker actúa como un mecanismo de protección es sistemas de microservicios, gestionando el flujo de solicitudes y

evitando que los fallos se propaguen a otras partes del sistema (Indrasiri & Siriwardena, 2018).

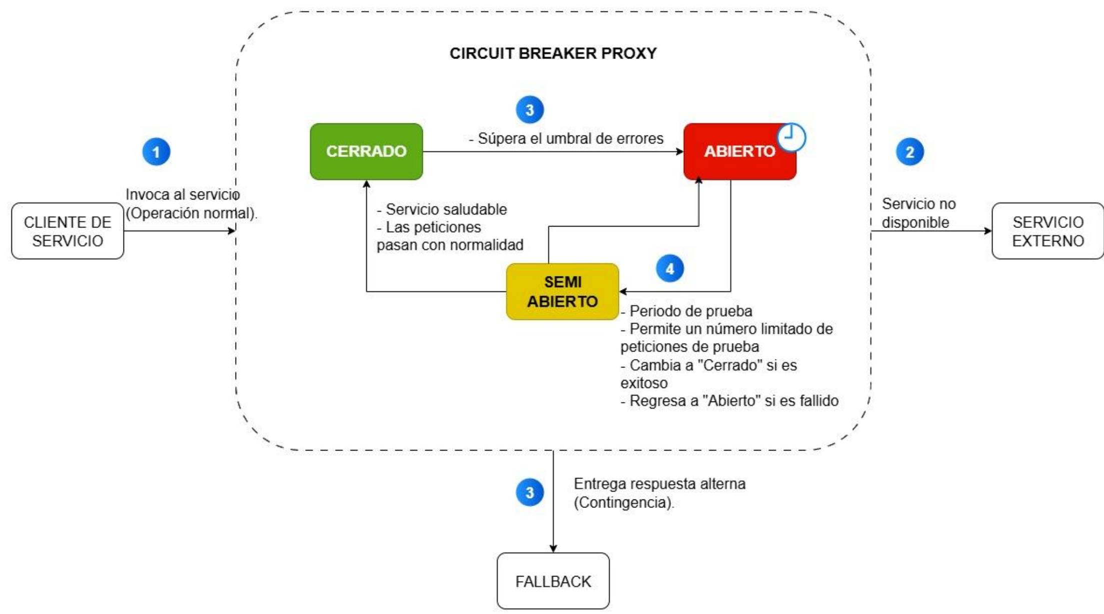
Estados del patrón Circuit Breaker

El patrón opera en tres estados distintos, que gestionan el flujo de peticiones de manera inteligente. Su ciclo de vida se ilustra en la Figura 5.

- **Cerrado (Closed)**: La aplicación se comunica correctamente con otros servicios, como bases de datos o APIs externas. Las solicitudes se realizan y responden sin problemas.
- **Abierto (Open)**: Si un servicio externo comienza a fallar de manera repetida debido a caídas o presenta comportamientos anormales, como tiempos de respuesta lentos o errores constantes, el patrón Circuit Breaker entra en acción. Para evitar que la aplicación sobrecargue aún más el servicio defectuoso, el Circuit Breaker cambia al estado “Abierto”. En este estado, el microservicio deja de contactar al servicio problemático y, en su lugar, devuelve inmediatamente un error o una respuesta alternativa, lo que ayuda a evitar el desperdicio innecesario de recursos y tiempo.
- **Semi-Abierto (Half Open):** En esta etapa, el Circuit Breaker permite que nuevamente unas pocas solicitudes intenten contactar al servicio que estaba fallando con la finalidad de verificar si ya se ha recuperado:

    - Si el servicio responde correctamente, el Circuit Breaker vuelve al estado "Cerrado", restaurando el flujo normal de solicitudes.
    - Si el servicio sigue fallando, el Circuit Breaker vuelve al estado "Abierto" y espera un tiempo determinado antes de intentar nuevamente.

Figura 5 Esquema del funcionamiento del Patrón Circuit Breaker



3.4. Integración de resilience4j en aplicaciones Spring Boot

La combinación de *Spring Boot* y *Resilience4j* ha ganado popularidad en la construcción de sistemas resilientes, especialmente en entornos basados en microservicios dónde las fallas parciales, la latencia y la inestabilidad de servicios externos son escenarios comunes (Larsson, Hands-On Microservices with Spring Boot and Spring Cloud, 2019).

Spring Boot proporciona una plataforma madura para el desarrollo de aplicaciones *Java*, mientras que *Resilience4j* añade una capa especializada de tolerancia a fallos diseñada que incorpora patrones como *Circuit Breaker, Retry, Bulkhead, Rate Limiter, Time Limiter* de forma declarativa, desacoplada y altamente configurable (Baeldung, 2024).

3.4.1 Spring Boot

Spring Boot es un framework de código abierto Java, basado en Spring Framework. Está diseñado para simplificar y reducir la configuración repetitiva en la creación de aplicaciones backend, como los microservicios y las aplicaciones web. (Spring, 2025)

Las principales características de *Spring Boot* son (Musib, 2022):

- Permite generar una aplicación especificando solo las dependencias necesarias, evitando configuraciones manuales extensas.
- Configura automáticamente componentes según las dependencias y propiedades detectadas, como, por ejemplo, una base de datos.
- Utiliza "starters" que agrupan dependencias necesarias para un tipo específico de aplicación (como spring-boot-starter-web para apps web).
- Las aplicaciones se pueden ejecutar sin necesidad de servidores externos, ya que incluyen un servidor web embebido como Tomcat.
- Ofrece herramientas de monitoreo y gestión como métricas, health checks y más, facilitando la administración en entornos productivos.

Si bien *Spring Boot* ya presenta varias ventajas al construir microservicios, es importante fortalecer la resiliencia. Para ello, permite la integración de librerías como *Resilience4j*, que proporciona capacidades de resiliencia, como circuit breakers, control de fallos y reintentos. Estas capacidades ayudan a gestionar los fallos de forma eficiente, evitando la propagación de errores, mejorando la disponibilidad de los sistemas.

3.4.2 Resilience4j

Es una librería ligera, diseñada para crear aplicaciones tolerantes a fallos. Se integra fácilmente con aplicaciones *Spring Boot* mediante propiedades y anotaciones. *Resilience4j* mejora la robustez de las aplicaciones al prevenir fallos en cascada, gestionar picos de tráfico y asegurar una degradación controlada del servicio.

La configuración de las propiedades de la librería *Resilience4j* se realiza principalmente a través del archivo `application.yml` y mediante anotaciones en

el código fuente. A continuación, se detallan las propiedades clave que permiten controlar el comportamiento del patrón Circuit Breaker, las cuales se han agrupado mediante los estados del circuito (resilience4j, 2025).

Tabla 2 Parámetros de Configuración de Resilience4j para el Estado CERRADO

Propiedad	Descripción	Valor predeterminado
`sliding-window-size`	Tamaño de la ventana deslizante para calcular el número de llamadas recientes.	100
`sliding-window-type`	Tipo de ventana deslizante. Puede ser COUNT_BASED (por número de llamadas) o TIME_BASED (por tiempo).	COUNT_BASED
`minimum-number-of-calls`	Número mínimo de llamadas requeridas antes de que se evalúe la tasa de fallos.	100
`failure-rate-threshold`	Umbral de tasa de fallos en porcentaje. Si se supera este porcentaje, el circuito se abre.	50
`slow-call-duration-threshold`	El umbral de tiempo en milisegundos que define una llamada como lenta.	500ms
`slow-call-rate-threshold`	Umbral de porcentaje de llamadas lentas que activa la apertura del circuito si se supera.	100%

Tabla 3 Parámetros de Configuración de Resilience4j para el Estado ABIERTO

Propiedad	Descripción	Valor predeterminado
`wait-duration-in-open-state`	Duración (en milisegundos) que el Circuit Breaker permanece en estado abierto antes de probar nuevamente.	1000ms

Tabla 4 Parámetros de Configuración de Resilience4j para el Estado SEMIABIERTO

Propiedad	Descripción	Valor predeterminado
`permitted-number-of-calls-in-half-open-state`	Número máximo de llamadas permitidas en estado semi-abierto para probar si el servicio puede recuperarse.	10
`automatic-transition-from-open-to-half-open-enabled`	Habilita la transición automática del estado abierto a semi-abierto después de que haya pasado waitDurationInOpenState .	false

Además de la configuración de las propiedades, en el código se debe agregar la anotación `@CircuitBreaker` (Larsson, 2023), y colocar los siguientes parámetros.

- `name`: es un parámetro obligatorio, es el nombre que se le asigna al Circuit Breaker, debe coincidir con el nombre de la instancia definida en el archivo `properties.yml`.
- `fallbackMethod`: se ejecuta cuando el patrón está abierto.

  El método de respaldo debe cumplir varios criterios:

  - Debe devolver el mismo tipo de datos que el método original.
  - Debe aceptar una excepción como parámetro y también puede aceptar los mismos parámetros que el método original (opcional), pero no más que el método original.

3.5. Simulación de condiciones de red

Para validar la resiliencia de una aplicación, es necesario simular condiciones de red adversas y fallos controlados. Herramientas como Chaos Monkey, Gremlin y ToxiProxy han sido diseñadas específicamente para facilitar estas prácticas de Ingeniería del caos (Chaos Engineering).

El término Chaos Engineering fue acuñado por Netflix, particularmente por su equipo de ingeniería encargado de operar sistemas distribuidos a gran escala. Aunque Netflix ya utilizaba Chaos Monkey desde 2010, el concepto fue formalizado públicamente en 2016 con la publicación del documento "Principles of Chaos Engineering" (Rosental & Jones, 2020).

Chaos Monkey es una herramienta de código abierto de Netflix, su objetivo es probar de forma proactiva la resiliencia de un sistema mediante la simulación de fallos e interrupciones reales. (Netflix, 2018)

Gremlin, por su parte, es una plataforma comercial orientada a la ejecución controlada de ataques de caos, permitiendo reproducir fallos de CPU, memoria, latencia, pérdida de paquetes o apagado de servicios con el fin de medir la tolerancia del sistema (Gremlin, 2023).

ToxiProxy es una herramienta de código abierto que permite simular condiciones de red, así como la introducción de fallos aleatorios. Actúa como un proxy intermedio entre un microservicio consumidor y su dependencia, facilitando las pruebas de resiliencia en entornos de desarrollo (Shopify, 2025)

En la Figura 6 se muestran los componentes principales de ToxiProxy: un cliente, que permite configurar las propiedades de red deseadas, detalladas en la Tabla 5, y un servidor proxy, que intercepta el tráfico y aplica las condiciones simuladas.

Figura 6 Arquitectura de ToxiProxy

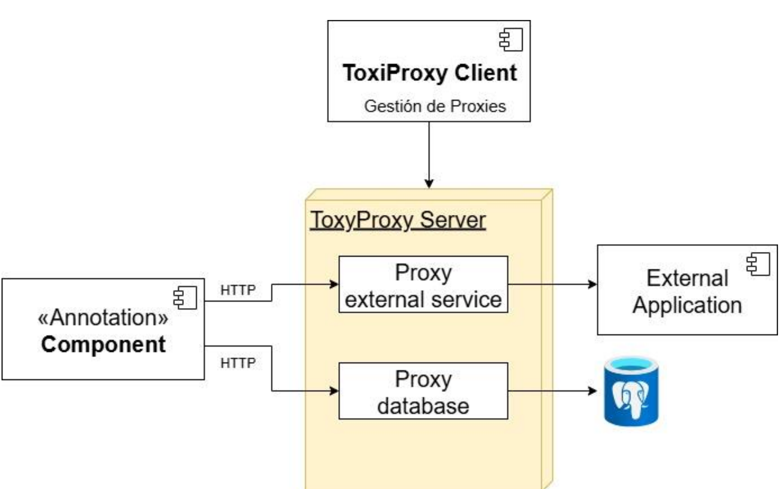


Tabla 5 Propiedades configurables de Toxiproxy

Propiedad	Descripción
Latencia (latency)	Permite simular servicios con alta demora en la respuesta.
Tiempos de Espera Agotados (timeout)	Permite detener el flujo de datos después de un cierto período, simulando timeouts en la conexión o en la respuesta del servicio externo.
Corte de Conexión (slicer):	Permite cortar los flujos de datos en fragmentos pequeños o introducir pausas, simulando conexiones inestables.
Ancho de Banda Limitado (bandwidth)	Limita una conexión a un número máximo de datos que pueden pasar por unidad de tiempo, simulando redes congestionadas.

3.6. Pruebas de rendimiento

Las pruebas de rendimiento forman parte fundamental del ciclo de vida de desarrollo y aseguramiento de calidad. Su objetivo es evaluar cómo se comporta un sistema bajo diferentes cargas de trabajo, midiendo aspectos como tiempo de respuesta, consumo de recursos, capacidad de procesamiento, estabilidad y escalabilidad (Parasoft, 2025).

Este tipo de pruebas permite identificar cuellos de botella, validar la capacidad del sistema para soportar el uso previsto y asegurar una buena experiencia de usuario en todo tipo de aplicaciones, tanto en entornos de escritorio como móviles, incluso bajo condiciones de alta demanda (Yorkston, 2021).

En arquitecturas modernas basadas en microservicios, las pruebas de rendimiento son esenciales para garantizar la resiliencia del sistema, ya que permiten anticipar fallos, degradaciones o comportamientos inesperados antes de llegar a producción. La combinación de pruebas de carga, estrés y resistencia facilita la detección temprana de comportamientos anómalos y asegura que los servicios puedan escalar y recuperarse adecuadamente.

3.6.1. Pruebas de carga

Las pruebas de carga (Load Testing) evalúan el comportamiento del sistema bajo un volumen específico de usuarios o transacciones, generalmente equivalente al uso normal o máximo esperado en operación. Su propósito es comprobar que la aplicación cumple los tiempos de respuesta requeridos, se mantiene estable y utiliza los recursos de manera eficiente durante una carga establecida.

Estas pruebas permiten responder preguntas como:

- ¿Cuántos usuarios simultáneos puede manejar el sistema sin degradarse?
- ¿Cómo varía el tiempo de respuesta a medida que aumenta la carga?
- ¿Existen componentes que se saturan antes que otros?

Además, los resultados obtenidos sirven para ajustar parámetros de infraestructura, optimizar consultas, mejorar la configuración de servicios externos o validar estrategias de escalabilidad horizontal y vertical.

3.6.2. Pruebas de estrés

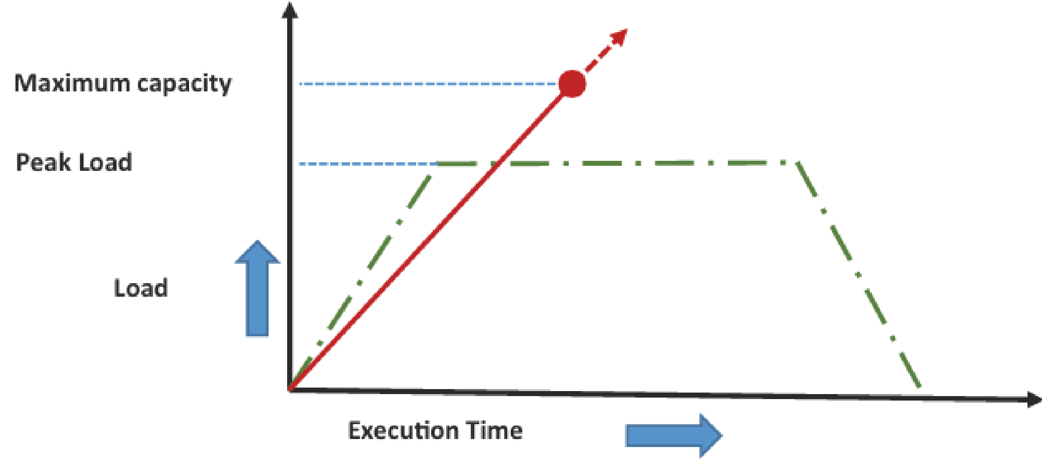
Las pruebas de estrés (Stress Testing) someten al sistema a cargas superiores a las esperadas en escenarios normales de operación, con el propósito de determinar su punto de ruptura, observar cómo falla y analizar su capacidad de recuperación. A diferencia de las pruebas de carga, que trabajan dentro de límites razonables, las pruebas de estrés llevan el sistema al extremo: incrementan progresivamente el número de usuarios, el volumen de peticiones o la cantidad de datos hasta que el sistema deja de responder adecuadamente.

Este tipo de prueba permite identificar:

- El límite máximo de capacidad del sistema.
- Qué componentes se degradan primero.
- Si el sistema puede recuperarse una vez que cesa la sobrecarga.
- La solidez de los mecanismos de resiliencia, como timeouts, reintentos, circuit breakers y políticas de escalado.

En sistemas críticos, las pruebas de estrés ayudan a garantizar que la aplicación actúe de forma controlada bajo situaciones inesperadas, evitando fallos catastróficos o comportamientos no deseados.

Figura 7 Pruebas de carga vs pruebas de estrés



Nota. La imagen fue tomada del libro (Yorkston, 2021)

3.6.3. Pruebas de pico

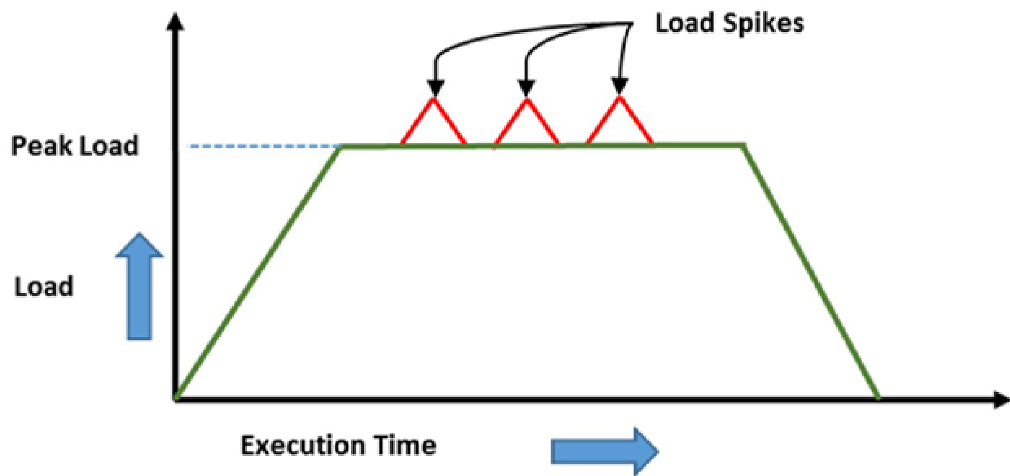
Las pruebas de pico (spike testing) miden la capacidad del sistema para responder ante incrementos abruptos e inesperados en la carga, usualmente generados en cuestión de segundos o minutos. A diferencia de las pruebas de estrés, donde la carga incrementa gradualmente, en las pruebas de pico el objetivo es observar cómo reacciona el sistema ante un cambio repentino.

Permiten evaluar:

- Cómo responde el sistema ante picos súbitos de tráfico (por ejemplo, promociones, ventas especiales o eventos virales).
- La rapidez con que los mecanismos de escalado automático pueden activarse.
- La estabilidad del sistema durante la caída posterior al pico.
- Comportamientos como bloqueos, timeout masivos o saturación de bases de datos.

Estas pruebas son relevantes para sistemas de alta demanda donde los patrones de tráfico pueden variar rápidamente, como aplicaciones de comercio electrónico, redes sociales y servicios de streaming.

Figura 8 Pruebas de pico



Nota. La imagen fue tomada del libro (Yorkston, 2021)

3.7. Monitoreo

El monitoreo es un componente esencial para garantizar la estabilidad y la resiliencia de los sistemas modernos. Su objetivo es identificar las causas del comportamiento anómalo que permitan comprender el estado de las aplicaciones, servicios e infraestructura. (Chakraborty & Pratap Kundan, 2021).

En arquitecturas basadas en microservicios, el monitoreo se integra dentro del modelo de observabilidad, que combina métricas, logs y trazas distribuidas para ofrecer una visión completa del sistema. Herramientas como Prometheus y Grafana son ampliamente adoptadas gracias a su integración, extensibilidad (Mentores Tech, 2025).

3.7.1. Métricas de rendimiento de la JVM

Para evaluar el rendimiento de una aplicación Java, es necesario monitorear las métricas a nivel de JVM (Java Virtual Machine). Estas métricas permiten observar cómo se utilizan los recursos, dónde pueden surgir cuellos de botella. Algunas de

las métricas más relevantes son CPU, Heap Memory, y Thread States (runnable, time waiting, blocked,etc)

a) Uso de CPU

Medir el CPU consumido por la JVM es importante ya que indica cuánta carga computacional genera la aplicación y qué porcentaje del procesador está involucrado en su ejecución.

Investigaciones recientes han demostrado que los diferentes recolectores de basura y sus hilos concurrentes pueden competir por recursos de CPU, afectando directamente el rendimiento, la latencia y los fallos de caché en cargas modernas (Narra, 2025).

Esto refuerza la necesidad de monitorear esta métrica de forma continua para anticipar cuellos de botella y optimizar la configuración de la JVM.

b) JVM Heap

El heap es la región de memoria dónde se almacena los objetos creados por la aplicación, y su administración por parte del Garbage Collector(GC) tiene un impacto directo en la latencia y en el rendimiento general del sistema (Obregon, 2023).

Es particularmente relevante monitorear:

- HeapCommitted: la cantidad de memoria asignada (“comprometida”) para el heap, disponible para la JVM.
- HeapUsed: la parte actualmente usada del heap, que siempre es menor o igual a la comprometida.

Estos datos proporcionan una visión clara del “coste” que la recolección de basura impone sobre el rendimiento de la aplicación.

c) Thread States

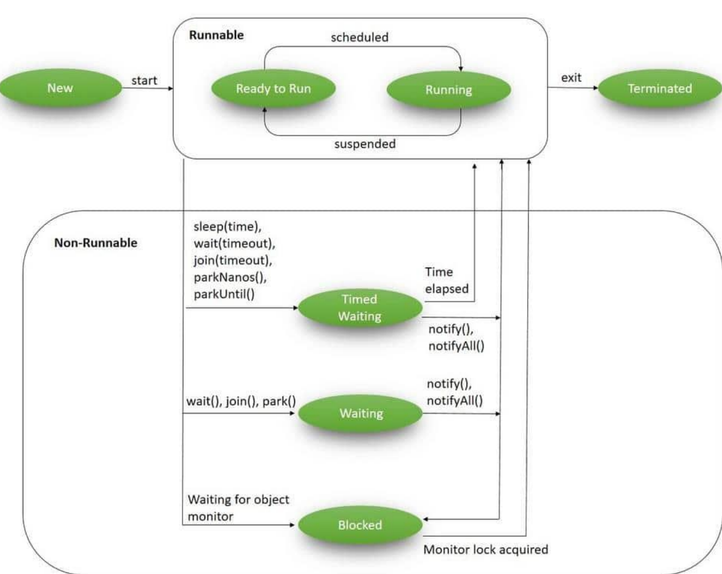
Los hilos (threads) de la JVM permiten comprender cómo la aplicación maneja la concurrencia. Conocer cuántos hilos existen, en qué estado están y cómo evolucionan esos estados ayuda a diagnosticar problemas de bloqueo, saturación o mala configuración. Algunas métricas importantes se describen en la Tabla 6 y en la Figura 9 se puede observar el ciclo de vida de un hilo.

Tabla 6 Estados de los hilos en Java

Estado	Descripción
Runnable	Hilos activos listos para ejecutarse y esperando a que el CPU esté disponible.
Time Waiting	Hilos que están esperando por un recurso o por un tiempo determinado (por ejemplo, sleep(), wait(timeout)). Este estado es común cuando se usan timeouts, o cuando se llaman métodos bloqueantes con límite de tiempo.
Waiting	Hilos que están esperando indefinidamente (wait(), sin timeout). Esto puede sugerir bloqueo o sincronización intensa.
Blocked	Cuando un hilo intenta acceder a un bloque/método sincronizado que actualmente está en poder de otro hilo.

Nota. La información fue obtenida de (Bhushan, 2025)

Figura 9 Ciclo de vida de un hilo en Java



Nota. La imagen fue obtenida de (Baeldung, 2024)

3.7.2. Prometheus

Prometheus es un sistema de monitoreo y recolección de métricas de código abierto orientado a aplicaciones modernas distribuidas. Fue creado por SoundCloud en 2012 y más tarde adoptado como proyecto oficial de la Cloud Native Computing Foundation CNCF, convirtiéndose en uno de los estándares de la industria para monitoreo cloud-native (Prometheus, 2025) .

Prometheus utiliza un modelo de extracción, en el cual consulta endpoints como /metrics para recolectar datos que almacena en una base de datos de series temporales. Su lenguaje de consulta, PromQL, facilita el análisis detallado de métricas y la creación de reglas de alertas. Además, ofrece una amplia variedad de exporters que permiten monitorear sistemas operativos, bases de datos y servicios externos (Pivotto & Brazil, 2023).

Gracias a su arquitectura modular y eficiencia operativa, Prometheus es una de las herramientas más utilizadas para monitoreo en Kubernetes y sistemas distribuidos.

3.7.3. Grafana

Grafana es una plataforma de visualización y análisis de datos que permite crear paneles interactivos basados en diversas fuentes de datos como Prometheus, InfluxDB, Elasticsearch, Loki o SQL. Aunque no almacena métricas por sí misma, tiene la capacidad de transformar series temporales en gráficos avanzados y paneles dinámicos. (Grafana Labs., 2025)

Grafana ofrece funcionalidades como generación de alertas, plantillas de dashboards, integración con herramientas DevOps y un ecosistema de plugins ampliamente utilizado. Cuando se integra con Prometheus, se obtiene una solución completa de monitoreo, donde Prometheus recolecta y procesa métricas, y Grafana las visualiza de forma clara e intuitiva (Chakraborty & Pratap Kundan, 2021).

4. Materiales y metodología

Este capítulo se estructura en cinco etapas. En primer lugar, se presenta el caso de estudio, describiendo sus principales características y contexto de aplicación. Posteriormente, se detalla el entorno experimental y la implementación del patrón Circuit Breaker en el caso seleccionado. A continuación, se plantean los escenarios de prueba diseñados para evaluar el comportamiento del patrón implementado. Finalmente, se describen las métricas utilizadas para el análisis del desempeño y las condiciones del entorno en el que se ejecutaron las pruebas.

4.1. Definición de métricas

Para evaluar objetivamente el comportamiento del sistema, se han definido los siguientes Indicadores Clave. Estas métricas permiten contrastar el estado de salud del sistema con y sin mecanismos de protección.

Tabla 7 Métricas e Indicadores de Resiliencia

Métrica	Indicador Técnico	Descripción
Latencia del Servicio	p90 y p95 (ms)	Los percentiles 90 y 95 permiten medir la degradación experimentada por la mayoría de los usuarios durante picos de carga.
Disponibilidad y Robustez	Tasa de Error (%)	Proporción de solicitudes fallidas (HTTP 5xx) frente a solicitudes exitosas (HTTP 2xx). Se distingue entre fallos no controlados y respuestas de contingencia (fallback).
Capacidad de Procesamiento	Throughput (TPS)	Transacciones por segundo que el sistema logra completar. Permite identificar si la capacidad se degrada drásticamente ante fallos externos.
Saturación de Recursos	Estado de Hilos (JVM Thread States)	Métrica Crítica. Se monitorea el número de hilos en estado TIMED_WAITING y BLOCKED. Un aumento descontrolado en esta métrica es el precursor técnico de un fallo en cascada (agotamiento del pool).

Consumo de Memoria	JVM Heap	Evalúa si la acumulación de peticiones en espera provoca fugas de memoria o saturación del Heap, llevando a errores OutOfMemoryError.
Estado del Circuito	State Transition	(Closed, Open, Half-Open). Se correlaciona con el tráfico para verificar si el patrón reacciona en los tiempos configurados.

4.2. Caso de estudio

El caso de estudio corresponde a un escenario cotidiano en el sector financiero, centrado en la consulta de la factura del servicio eléctrico a través de los canales digitales de una entidad financiera. Este proceso se apoya en un conjunto de microservicios internos y una dependencia externa: una API simulada de la empresa eléctrica encargada de proporcionar información de las facturas.

El escenario es ideal porque presenta dos desafíos clave que justifican el uso del patrón Circuit Breaker:

- **Altos picos de transaccionalidad:** Se ha observado un comportamiento recurrente de los clientes, quiénes tienden a consultar y pagar sus facturas justo en fechas cercanas al vencimiento. Este comportamiento genera presión en los sistemas involucrados.
- **Dependencias externas:** Se integra con una API de terceros, responsable de proporcionar el valor actualizado del servicio eléctrico. Dado que la API esta fuera del control de la entidad financiera, representa un punto crítico y un riesgo potencial en la disponibilidad del servicio.

Arquitectura y componentes

En la Figura 10 se presenta el diagrama de componentes que describe la arquitectura interna del sistema, mientras que en la Figura 11 se muestra un diagrama de secuencia que facilita su comprensión.

La solución expone dos canales digitales: un cliente web y un cliente móvil que permiten a los usuarios consultar el valor de sus facturas. Todas las solicitudes ingresan a través del API Gateway, el cual actúa como punto de entrada centralizado para la comunicación con los distintos microservicios del sistema.

El flujo de interacción se desarrolla de la siguiente manera:

a. El cliente móvil o web envía una solicitud al API Gateway para consultar el valor de una factura.
b. El API Gateway reenvía la solicitud al `bill-management-service`, responsable de gestionar la información relacionada con las facturas.
c. Este servicio consulta el monto y el estado de la factura a través del `electric-bill-service`, el cual se comunica con un servicio externo denominado `electric-bill-api`.
d. El `electric-bill-api` obtiene los datos de la factura desde el sistema externo y devuelve la información al `electric-bill-service`, que posteriormente la propaga hacia el cliente.

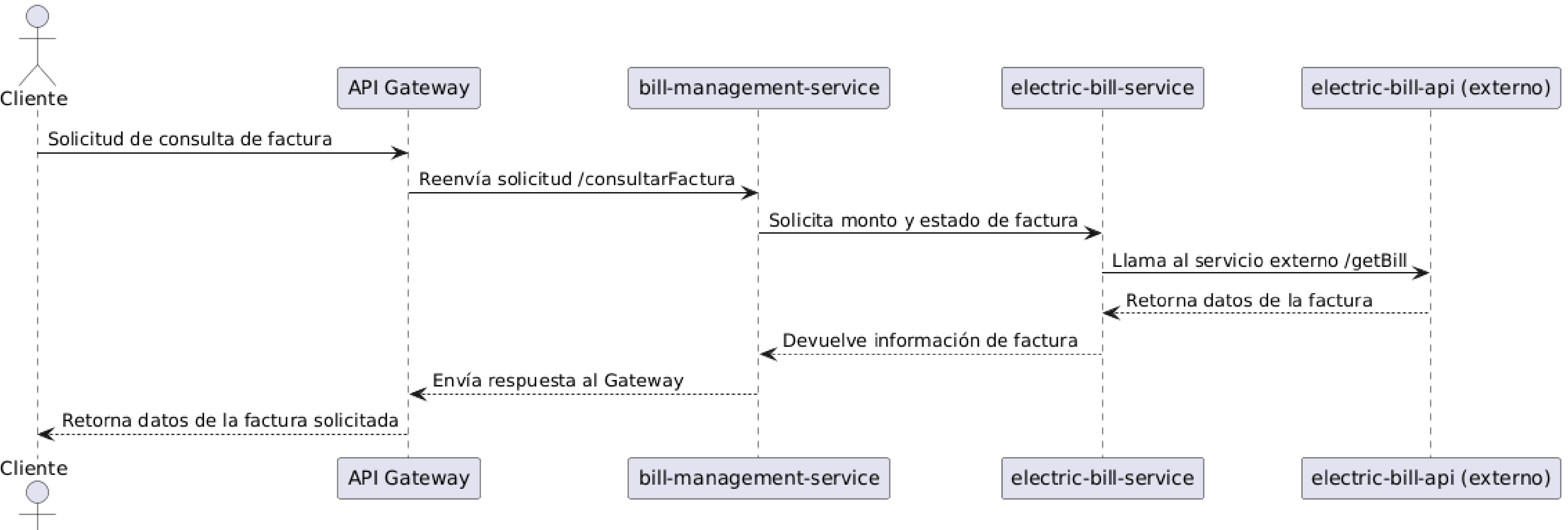
Adicionalmente, la arquitectura incorpora servicios internos complementarios:

- `account-service`: gestiona las cuentas bancarias de los clientes y permite realizar débitos directos.
- `credit-card-payment-service`: administra las operaciones de pago con tarjeta de crédito.
- `payment-management-service`: coordina las operaciones de pago, determinando si se procesan mediante cuenta bancaria o tarjeta, y registra las transacciones realizadas.

Figura 10 Arquitectura General del Sistema

Docker - Arquitectura Interna
Mobile client
a
consultar el valor de la factura
api-gateway
b
envía la solicitud de consulta de la factura
bill-management-service
c
consulta el monto y el estado de la factura
electric-bill-service
d
obtiene los datos de la factura y confirma el pago
«external-service»
electric-bill-api
ejecutar el pago mediante la cuenta bancaria o tarjeta de crédito
payment-management-service
listar las cuentas del cliente
account-service
realiza el débito de la cuenta
listar las tarjetas de crédito del cliente
credit-card-payment-service
realiza el registro del pago con tarjeta de crédito

Figura 11 Diagrama de secuencia del funcionamiento del sistema



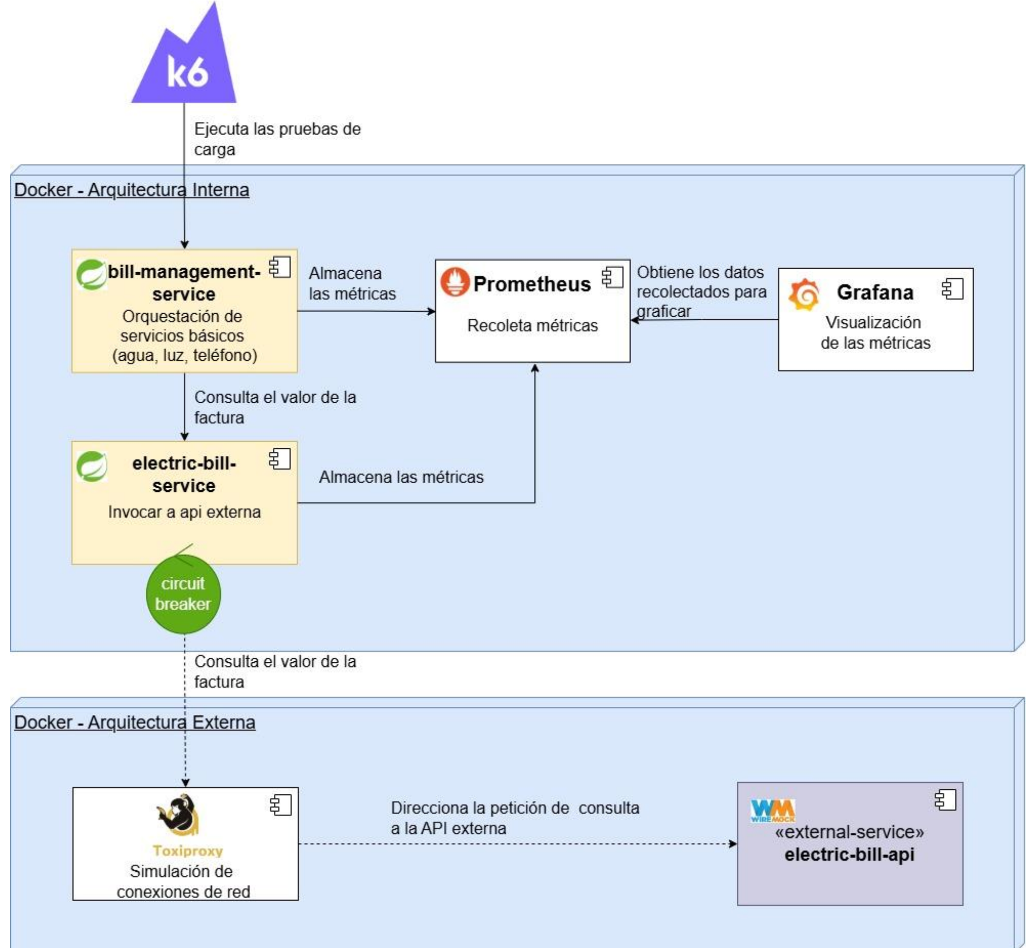
4.3. Entorno experimental

El microservicio `bill-management-service` es el punto de entrada donde se generan las pruebas de carga. El objetivo es observar cómo el patrón Circuit Breaker, implementado en el microservicio `electric-bill-service`, responde frente a distintos escenarios de degradación externa.

La Figura *12* muestra el diagrama de estudio del patrón, donde se evidencia:

- el flujo real de las solicitudes,
- los puntos de inyección de fallos,
- y el monitoreo continuo con Prometheus y Grafana.

Figura 12 *Diagrama de estudio del patrón Circuit Breaker utilizando monitoreo*



Para llevar a cabo los casos de prueba, se utilizó un conjunto de herramientas que permiten simular fallos externos, generar carga y monitorear métricas en tiempo real. La Tabla 8 resume el stack tecnológico empleado.

Tabla 8 Stack Tecnológico Usado en la Implementación del Circuit Breaker

Componente	Tecnología	Descripción
Framework Backend	Spring Boot 3.4.5/Java 21	Construcción de microservicios
Resiliencia	Resilience4j	Implementación del patrón Circuit Breaker
Gestión de dependencias	Gradle	Automatización de compilación, pruebas y empaquetado.
Observabilidad	Prometheus + Grafana	Monitoreo y visualización en tiempo real de métricas clave (latencia, errores, CPU, memoria, hilos).
Api externo	Wiremock	Servicio simulado que emula el comportamiento de una API externa, permitiendo recrear escenarios controlados de respuesta.
Simulación de fallos	Toxiproxy	Introducción controlada de latencias y fallos en la dependencia externa.
Pruebas de carga	K6	Generación de tráfico concurrente para evaluar el rendimiento bajo condiciones normales y degradadas.
Contenerización	Docker	Despliegue local reproducible de todos los servicios y herramientas auxiliares.

Además, las pruebas se ejecutaron en un entorno local, utilizando Docker como plataforma de despliegue, lo que garantiza que los servicios y herramientas se ejecuten de manera consistente y puedan ser replicados por otros investigadores o arquitectos. La Tabla 9 presenta las características de la máquina empleada durante las pruebas.

Tabla 9 Características de la máquina para ejecución de pruebas

Recurso	Especificación
Sistema Operativo	Windows 10 Home
Procesador (CPU)	Intel(R) Core (TM) i7-7500U CPU @ 2.70GHz 2.90 GHz
Memoria RAM	16 GB
Almacenamiento	SSD 1000 GB
Versión de Docker	28.0.4
Límite de recursos asignados a Docker	2GB

Finalmente, todo el código fuente, scripts de pruebas, configuraciones y archivos docker-compose utilizados en este entorno experimental se encuentran disponibles en el repositorio:

https://github.com/veroely/circuit-breaker-system

Este repositorio permite reproducir el entorno completo, levantar los servicios y ejecutar los escenarios definidos en este estudio.

4.4. Implementación del Patrón Circuit Breaker

El patrón Circuit Breaker se integró en el microservicio electric-bill-service mediante la librería Resilience4j. Este componente orquesta las llamadas a proveedores externos y controla los fallos persistentes para evitar que se propaguen al resto del sistema.

La arquitectura interna del microservicio sigue un enfoque hexagonal (puertos y adaptadores), lo que favorece el desacoplamiento entre la lógica de negocio y la infraestructura.

El componente `ElectricBillController` se encarga de recibir y validar las solicitudes HTTP, realizar el mapeo de los DTOs correspondientes y delegar la ejecución en el servicio de aplicación `ElectricBillService`.

Además, los umbrales y parámetros de configuración se definen de forma declarativa en el archivo `application.yml`, lo que permite ajustar su comportamiento sin modificar el código fuente.

En la Figura 13 se presenta el diagrama de clases del microservicio y los puntos específicos en los que se aplica el patrón Circuit Breaker.

Figura 13 Diagrama de clases del microservicio eletric-bill-service

Infrastructure_Input

ElectricBillController

+getBill(providerId, ref) : : ResponseEntity

Uses

ElectricBillDtoMapper

+toResponse(ElectricBill) : : BillResponseDTO

Uses

Application

«Interface»
GetBillUseCase

+getBillDetails(providerId, ref) : : ElectricBill

Implements

ElectricBillService

+getBillDetails(providerId, ref) : : ElectricBill
-alternativeOption(fallback) : : ElectricBill

Anotación: @CircuitBreaker\nConfig: application.yml

Uses

«Interface»
ElectricBillProviderPort

+findByReference(ref) : : ElectricBill

Returns

Infrastructure_Output

«Interface»
ElectricityProviderClient

+getBillRemote(ref) : : ProviderResponseDTO

Uses

ElectricProviderAdapter

+findByReference(ref) : : ElectricBill

Implements

ProviderMapper

+toDomain(ProviderResponseDTO) : : ElectricBill

Uses

Creates

Domain

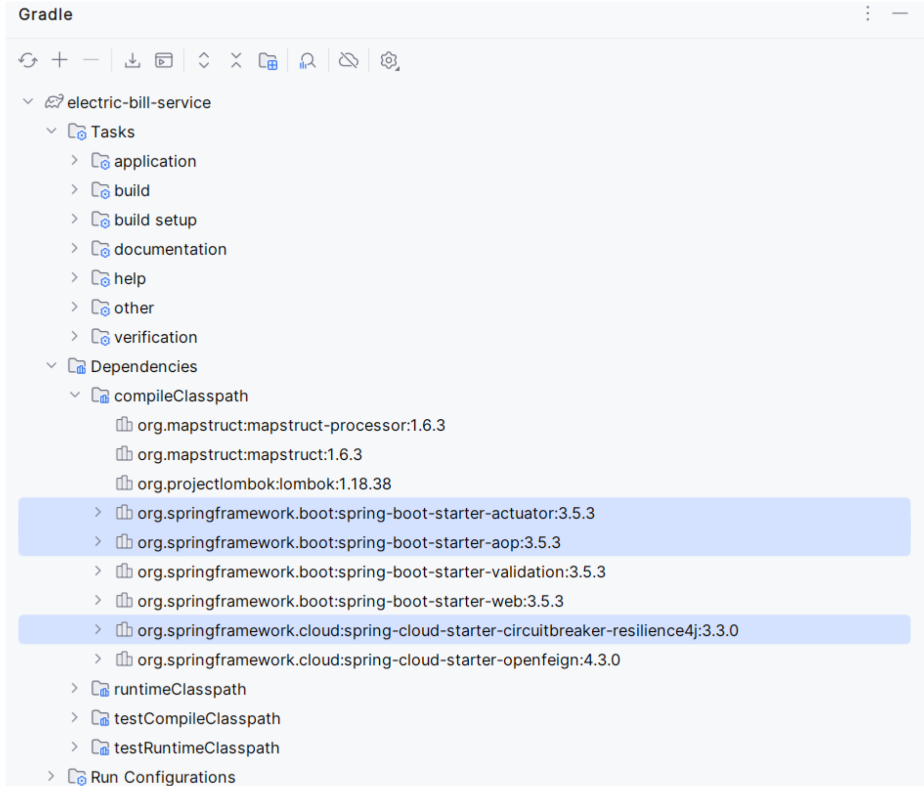
ElectricBill

+reference: String
+amount: BigDecimal
+dueDate: LocalDate

Los pasos principales de la implementación fueron los siguientes:

a. **Gestión de dependencias**: La incorporación de `spring-boot-starter-actuator` es esencial porque expone las métricas de Resilience4j hacia Prometheus, lo que permite monitorear los estados del circuito, los fallos, las latencias y las estadísticas de las ventanas deslizantes. Además, la dependencia `spring-boot-starter-aop` es necesaria para habilitar el funcionamiento del patrón Circuit Breaker.

Figura 14 Configuración de dependencias en el archivo build.gradle



b. **Implementación de código:** La lógica de negocio se realiza en la capa de servicio, específicamente en la clase `ElectricBillService`, encargada de orquestar la llamada a la API externa para obtener el valor de la factura.

Para proteger esta llamada, se utiliza la anotación `@CircuitBreaker`, que actúa como un proxy o interceptor sobre el método `getBillDetails()`, aplicando las reglas configuradas en el archivo `application.yml`.

La anotación recibe con dos parámetros esenciales:

- name = "getBillDetails": Identificador que vincula el método con la instancia definida en el archivo application.yml.
- fallbackMethod = "alternativeOption": Nombre del método de contingencia que se ejecuta cuando el circuito está en estado ABIERTO o cuando una llamada de prueba falla en el estado SEMI-ABIERTO.

El siguiente fragmento de código muestra la implementación del patrón Circuit Breaker dentro de la clase de servicio:

```
1    @CircuitBreaker(name = "getBillDetails", fallbackMethod = "alternativeOption")
2    @Override
3    public ElectricBill getBillDetails(String providerId, String referenceNumber) {
4        return electricBillRepository.findByReference(referenceNumber);
5    }
6
```

Nota. Fragmento de código tomado del repositorio del proyecto: https://github.com/veroely/circuit-breaker-system/blob/master/internal-architecture/electric-bill-service/src/main/java/com/ms/electric_bill_service/application/service/ElectricBillService.java

c. **Creación de la respuesta de contingencia:** Define la respuesta controlada del sistema ante un fallo. Su función es ofrecer una experiencia de usuario degradada pero funcional, evitando que el error se propague. El método debe cumplir con lo siguiente:

- Tener el mismo tipo de retorno que el método original.
- Aceptar, como mínimo, los mismos parámetros que el método original, y opcionalmente puede recibir la Exception que causó el fallo como último parámetro.

```
6
7    public ElectricBill alternativeOption(String providerId, String referenceNumber, Throwable throwable) {
8        return new ElectricBill(
9                referenceNumber,
10               BigDecimal.ZERO,
11               LocalDate.now(),
12               null,
13               null
14       );
15   }
```

Nota. Fragmento de código tomado del repositorio del proyecto: https://github.com/veroely/circuit-breaker-system/blob/master/internal-architecture/electric-bill-service/src/main/java/com/ms/electric_bill_service/application/service/ElectricBillService.java

d. **Configuración de propiedades:** En el archivo `application.yml` se define una instancia de Circuit Breaker llamada `getBillDetails`, la cual está asociada al método correspondiente en el código. Este bloque de configuración funciona como una plantilla reutilizable que se vincula a la lógica de negocio mediante anotaciones. Las propiedades definidas para esta instancia se detallan en la Tabla 10.

Tabla 10 Parámetros de Configuración del Circuit Breaker

Propiedad	Valor	Ejemplo
`sliding-window-size`	10	El sistema evalúa las últimas 10 llamadas a la API eléctrica. Si 5 fallan o son lentas → se abre el circuito.
`sliding-window-type`	COUNT_BASED	
`minimum-number-of-calls`	5	El circuito no se evaluará hasta que al menos 5 consultas de factura hayan ocurrido.
`failure-rate-threshold`	50	Si de 10 llamadas, 5 fallan con HTTP 500 o timeouts → el circuito se abre y se usa fallback.
`slow-call-duration-threshold`	3000	Cualquier consulta que tarde ≥ 3 segundos se registra como llamada lenta.
`slow-call-rate-threshold`	50%	Si 5 de las últimas 10 llamadas tardan más de 3s → el circuito se abre por lentitud.
`wait-duration-in-open-state`	6000	El circuito permanece 6 segundos sin enviar llamadas al proveedor eléctrico, devolviendo solo fallback.
`permitted-number-of-calls-in-half-open-state`	5	Se envían 5 llamadas reales a la API. Si todas son rápidas → se cierra; si fallan → vuelve a abrir.
`automatic-transition-from-open-to-half-open-enabled`	true	

Nota. La configuración completa se puede revisar en el siguiente repositorio https://github.com/veroely/circuit-breaker-system/blob/master/internal-architecture/electric-bill-service/src/main/resources/application.yml

4.5. Escenarios de prueba

Para evaluar el impacto del patrón Circuit Breaker en la resiliencia de los microservicios, se diseñaron los siguientes escenarios de prueba, utilizando herramientas clave para la simulación de carga y fallos, así como para el monitoreo en tiempo real.

Tabla 11 Resumen de los Escenarios de Prueba del Circuit Breaker

Escenario	Descripción	Circuit Breaker	Fallos simulados
E1. Línea base	Obtener comportamiento normal	Deshabilitado	No
E2. Sin Circuit Breaker con latencia	Evaluar degradación sin mecanismos de resiliencia	Deshabilitado	Sí (latencia)
E3. Circuit Breaker con latencia	Observar apertura del circuito por llamadas lentas del api externa.	Sí (latencia)	Sí (latencia)
E4. Circuit Breaker ante errores HTTP 5xx	Evaluar apertura por tasa de fallos	Habilitado	Sí (fallos 50–100%)
E5. Circuit Breaker con ajuste de parámetros	Identificar sensibilidad y puntos óptimos	Habilitado	Sí

Tras la definición de los escenarios de la Tabla 11, la ejecución de las pruebas se realizó en un entorno local controlado, desplegado mediante Docker Compose, lo que permitió garantizar reproducibilidad, aislamiento y consistencia en cada ejecución.

Para el monitoreo se habilitaron dos contenedores adicionales: Prometheus, encargado de recolectar métricas del microservicio principal y del servicio protegido por el Circuit Breaker, y Grafana, responsable de la visualización de métricas. Estas herramientas permitieron observar en tiempo real indicadores como latencia, tasa de errores, uso de CPU, consumo de memoria, número de hilos activos y estado del Circuit Breaker durante cada escenario.

4.5.1. E1: Línea Base

Dado que no se dispone de datos históricos de uso para esta funcionalidad específica, se tomará como referencia la métrica de login de usuarios en la aplicación móvil, estimando un volumen de aproximadamente 500 transacciones por segundo (TPS) como carga esperada en producción.

Se estima que 1 de cada 5 usuarios realiza la consulta de la factura eléctrica, lo que representa el 20% de los usuarios logueados. Aplicando esta proporción sobre los 500 TPS del login, se obtiene un volumen aproximado de 100 TPS para la funcionalidad de consulta de facturas en condiciones normales.

En esta etapa se realizará una prueba de línea base de carga, cuyo objetivo es:

Obtener el tiempo de respuesta del sistema bajo condiciones normales,

- Medir el rendimiento sin presencia de fallos en las dependencias externas,
- Establecer métricas de referencia para comparaciones futuras.

Hipótesis:

En condiciones normales, el sistema debería mantener las tasas de error bajas, latencia y uso de recursos estable.

Configuración:

- **Toxiproxy:** El proxy se configura en modo de paso (pass-through), es decir solo actúa como un proxy enviando las solicitudes al api externa operando de forma transparente sin inyectar fallos. La creación del proxy se realiza mediante la invocación del cliente con el siguiente comando:

```
cURL

1 curl --location 'http://localhost:8474/proxies' \
2 --header 'Content-Type: application/json' \
3 --data '{
4     "name": "electric_api_proxy",
5     "listen": "0.0.0.0:6000",
6     "upstream": "wiremock:8080",
7     "enabled": true
8 }'
```

- **k6:** Se ejecuta el script de carga provisto para generar un volumen significativo de tráfico concurrente.

```
1 export const options = {
2   vus: 100,               // volumen normal equivalente a 100 TPS aproximados
3   duration: '60s',        // prueba estable
4 };
```

Nota. El script completo se puede obtener de https://github.com/veroely/circuit-breaker-system/blob/master/performance-test/E1_1baseline-load-test.js

4.5.2. E2: Sin Circuit Breaker con latencia

¿Cómo se comporta el sistema cuando se presenta latencia en la dependencia externa, sin contar con mecanismos de resiliencia?

Hipótesis:

La presencia de latencia en la API externa provocará bloqueos en las peticiones, aumento en el consumo de recursos y un porcentaje de error elevado, que ocasionará los fallos en cascada.

Configuración:

- **Toxiproxy:** Replicar la configuración del Escenario 1 y, adicionalmente, agregar un "toxic" cuyo objetivo es simular latencia de 1000ms y luego 3000ms en el consumo de la API externa. Para ello, se puede utilizar el siguiente comando:

```
cURL
1 curl --location 'http://localhost:8474/proxies/electric_api_proxy/
    toxics' \
2 --header 'Content-Type: application/json' \
3 --data '{
4     "name": "test_latency",
5     "type": "latency",
6     "stream": "downstream",
7     "toxicity": 1.0,
8     "attributes": {
9         "latency": 1000,
10        "jitter": 0
11    }
12 }'
```

- **K6:** Se ejecuta un script similar al Escenario 1 para realizar la comparación más adelante.

4.5.3. E3: Circuit Breaker con latencia

¿El Circuit Breaker es capaz de evitar la degradación y mantener la estabilidad ante latencias elevadas en dependencias externas?

Hipótesis:

La activación del patrón Circuit Breaker y la configuración de los umbrales permitirá detectar retardos excesivos en las dependencias externas, lo que permitirá abrir el circuito y retornar respuestas controladas, evitando bloqueos y reduciendo la latencia.

Configuración:

- **Circuit Breaker:** En este escenario se configuraron las propiedades relacionadas con la detección de llamadas lentas, según se muestra en la Tabla 10.

  La combinación de estos valores permite que, a partir de la quinta llamada, el sistema evalúe si las respuestas superan el umbral de 3000 ms. En caso de sobrepasarlo, dichas llamadas se consideran lentas y se contabilizan para el cálculo de la tasa de llamadas lentas `slow-call-rate-threshold`, lo que puede llevar a la apertura del circuito si se excede el porcentaje configurado.

  Tabla 12 Umbrales para detección de llamada lentas

Propiedad	Valor predeterminado
`sliding-window-size`	10
`sliding-window-type`	COUNT_BASED
`minimum-number-of-calls`	5
`slow-call-duration-threshold`	3000
`slow-call-rate-threshold`	50

- **ToxiProxy:** En este escenario se va a configurar latencia de 3000 ms del ejercicio anterior para simular la persistencia de fallos en el API externa, pero ahora con el Circuit Breaker activo.

```
cURL

1 curl --location 'http://localhost:8474/proxies/electric_api_proxy/toxics' \
2 --header 'Content-Type: application/json' \
3 --data '{
4     "name": "test_latency",
5     "type": "latency",
6     "stream": "downstream",
7     "toxicity": 1.0,
8     "attributes": {
9         "latency": 3000,
10        "jitter": 0
11    }
12 }'
```

- **K6:** Se presenta un script incrementado el tiempo de ejecución y el número de usuarios virtuales para poder verificar cómo el sistema responde a los fallos con y sin el Ciruit Breaker

```
1 export const options = {
2   stages: [
3     { duration: "10s", target: 50 },    // warm-up
4     { duration: "120s", target: 150 }, // carga sostenida (ideal para ver CB abrir)
5     { duration: "10s", target: 0 },     // cool-down
6   ]
```

4.5.4. E4: Circuit Breaker ante errores HTTP 5xx

¿Cómo responde el sistema cuando la API externa falla 5xx y el Circuit Breaker está activo?

Hipótesis:

Cuando la tasa de fallos supere el umbral de `failure-rate-threshold`, el Circuit Breaker debe abrirse y devolver fallback, evitar saturación del hilo y mejorar el tiempo de respuesta respecto a E2.

Configuración:

- **Circuit Breaker:** Cuando se requiere controlar errores específicos se debe tomar en cuenta la propiedad `failure-rate-threshold y sliding-window-size` que representa el porcentaje de fallos que vamos a permitir.

Tabla 13 Umbrales para detección de llamada lentas

Propiedad	Valor predeterminado
sliding-window-size	10
sliding-window-type	COUNT_BASED
minimum-number-of-calls	5
failure-rate-threshold	50

- **ToxiProxy:** Aunque Toxiproxy no crea códigos HTTP, sí puede generar fallos en la red que tu microservicio puede mapear a errores internos.

```
cURL

curl --location 'http://localhost:8474/proxies/electric_api_proxy/toxics' \
--header 'Content-Type: application/json' \
--data '{
    "name": "reset",
    "type": "reset_peer",
    "stream": "downstream",
    "toxicity": 1.0,
    "attributes": {
        "timeout": 0
    }
}'
```

- **K6:** Se utiliza el script idéntico al Escenario 1, permitiendo una comparación directa de cómo el sistema responde a los fallos con y sin el Ciruit Breaker

4.5.5. E5: Circuit Breaker con ajuste de parámetros

¿Qué configuración del Circuit Breaker ofrece el mejor equilibrio entre protección, latencia y disponibilidad bajo distintas condiciones de fallo?

Hipótesis:

Los parámetros del Circuit Breaker, especialmente slow-call-rate-threshold, y wait-duration-in-open-state tienen un impacto directo en la capacidad del sistema para recuperarse de fallos y evitar saturación.

Configuración:

- **Circuit Breaker:** Se definen tres configuraciones experimentales para comparar:

Tabla 14 Variación de parámetros del Circuit Breaker

Configuración	Descripción	Propiedades
CB (estricto)	Apertura rápida ante fallos	`failure-rate-threshold = 20%, slow-call-rate-threshold = 20%, slow-call-duration-threshold = 2000, wait-duration-in-open-state = 6s`
CB (equilibrado)	Configuración recomendada	`failure-rate-threshold = 50%, slow-call-rate-threshold = 50%, slow-call-duration-threshold = 2000, wait-duration-in-open-state = 8s`
CB (tolerante)	Mayor resiliencia, menor sensibilidad	`failure-rate-threshold = 80%, slow-call-rate-threshold = 80%, slow-call-duration-threshold = 2000, wait-duration-in-open-state = 10s`

- **Toxiproxy:** Se introduce latencia de 2000 ms con una variación de +-500 ms

cURL

```
curl --location 'http://localhost:8474/proxies/electric_api_proxy/toxics' \
--header 'Content-Type: application/json' \
--data '{
    "name": "test_latency",
    "type": "latency",
    "stream": "downstream",
    "toxicity": 1.0,
    "attributes": {
        "latency": 3000,
        "jitter": 500
    }
}'
```

- **K6:** En este caso se va realizar las pruebas con el siguiente script, que se ejecuta en 17 minutos con la finalidad de verificar el comportamiento del circuit breaker.

```
export const options = {
  stages: [
    { duration: '2m', target: 50 },   // Fase 1: Normal
    { duration: '3m', target: 100 },  // Fase 2: Presión moderada
    { duration: '5m', target: 200 },  // Fase 3: Fallos por timeout
    { duration: '2m', target: 300 },  // Fase 4: Descenso
    { duration: '3m', target: 200 },  // Fase 5: Recuperación
    { duration: '2m', target: 0 }     // Fase 6: Normalización
  ]
};
```

5. Resultados y discusión

Esta sección presenta el análisis cuantitativo de los resultados obtenidos en cada escenario, utilizando los datos generados por k6 y las gráficas de Grafana, las cuáles evidencian el comportamiento del sistema.

5.1. Resultados E1: Línea Base

Este escenario establece la línea base del comportamiento del sistema bajo condiciones normales, sin degradación en servicios externos. Permite definir métricas de referencia y validar la estabilidad de los microservicios antes de aplicar mecanismos de resiliencia.

Tabla 15 Resultado de las métricas del Escenario 1

Métrica	**Valor Observado**	**Descripción**
Latencia promedio	141 ms	Promedio de duración de las respuestas HTTP; refleja un rendimiento general muy bueno.
Tiempo de respuesta p (90)	278 ms	El 90 % de las solicitudes fueron atendidas en menos de 104 ms.
Tiempo de respuesta p (95)	880 ms	El 95 % de las solicitudes fueron atendidas en menos de 150 ms (muy por debajo del umbral de 500 ms).
Latencia máxima (max)	2.14 s	
Tasa de error	0%	No se registraron errores en las solicitudes; todas fueron procesadas exitosamente.
Iteraciones procesadas	5924	
Duración total	60.4s	
Iteraciones promedio por segundo	86.76 req/s	Promedio de ejecuciones completas por segundo; refleja la carga efectiva generada (~81 TPS).

Los valores observados muestran que el sistema se comporta de forma estable bajo las condiciones normales.

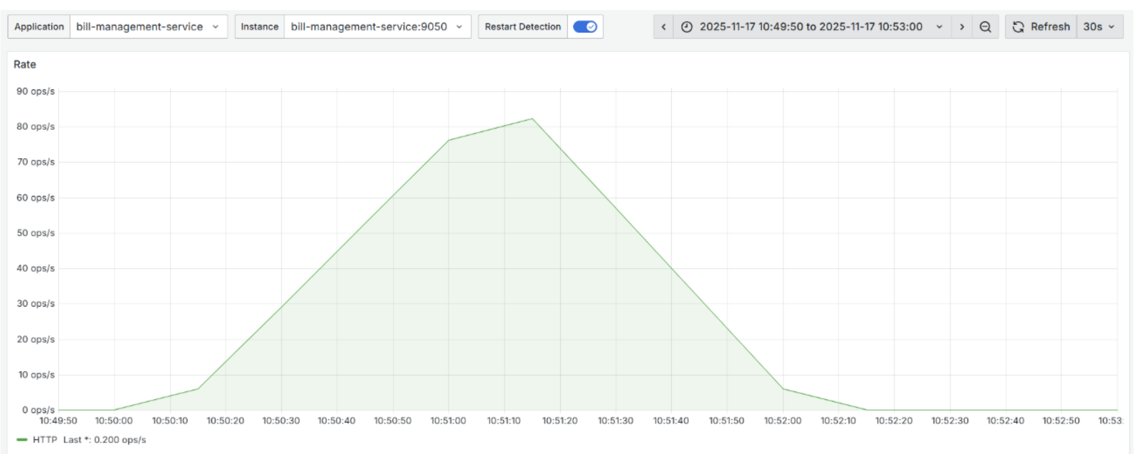
- La latencia promedio de 141 ms, y un porcentaje de 95% de las peticiones se procesan en 880 ms menor a un segundo, lo cual es adecuado para una aplicación transaccional.
- No se registraron errores durante la ejecución, lo que confirma que la comunicación con el servicio externo simulado se mantiene consistente en condiciones normales.
- La latencia máxima alcanzada 2.14 segundos corresponde a valores atípicos ocasionales esperables en un entorno Java por actividades internas de la JVM; no representa saturación ni degradación del servicio durante este escenario.

Gráficas complementarias en Grafana

a) Transacciones por segundo

En la Figura 15, se alcanza 86.76 solicitudes por segundo, que es coherente con la configuración del script utilizado y representa la capacidad estable del microservicio cuando no se introducen fallos ni retrasos artificiales.

Figura 15 Transacciones por segundo del microservicio bill-management-service en el Escenario 1



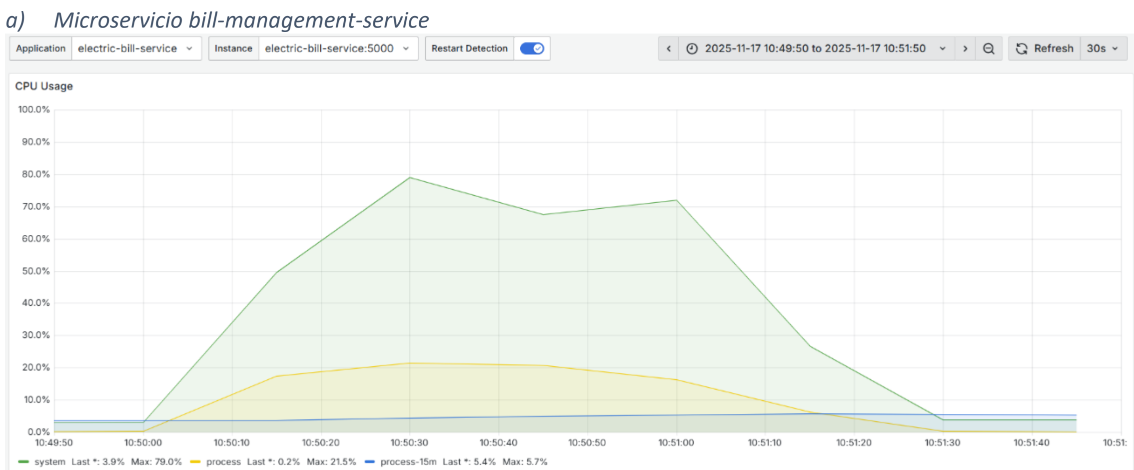
Nota. Datos obtenidos de Grafana durante la ejecución de la prueba de carga en K6.

b) Uso de CPU

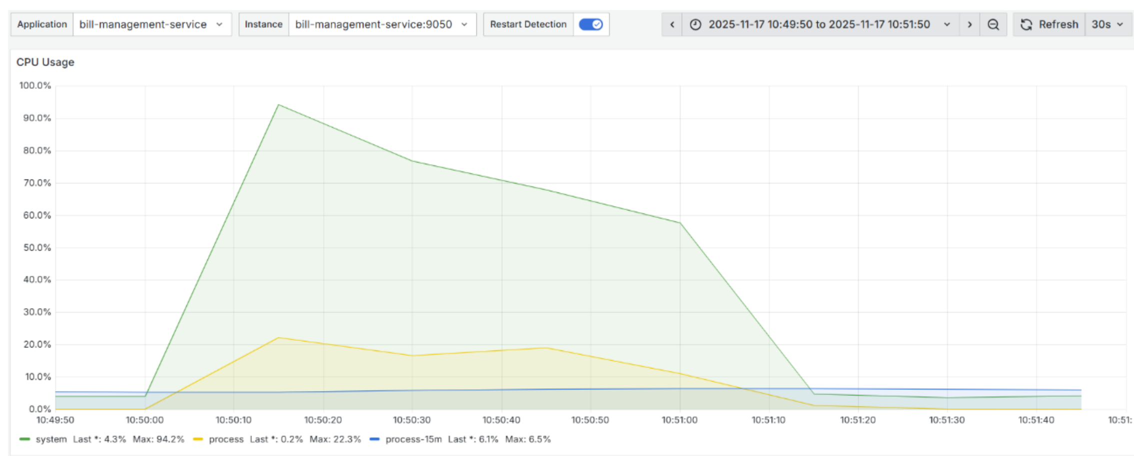
Al analizar la Figura 16, se observa que, aunque los microservicios llegan a usar bastante CPU de forma individual representado por la línea amarilla, alcanzando picos cercanos al 23%, la preocupación principal radica en el uso general del sistema. La línea verde se dispara y se mantiene consistentemente entre el 79% y el 94%, indicando que el servidor está funcionando a su máxima capacidad y está completamente sobrecargado.

Figura 16 Uso de CPU del Escenario 1

a) Microservicio bill-management-service



b) Microservicio electric-bill-service



Nota. Datos obtenidos de Grafana durante la ejecución de la prueba de carga en K6.

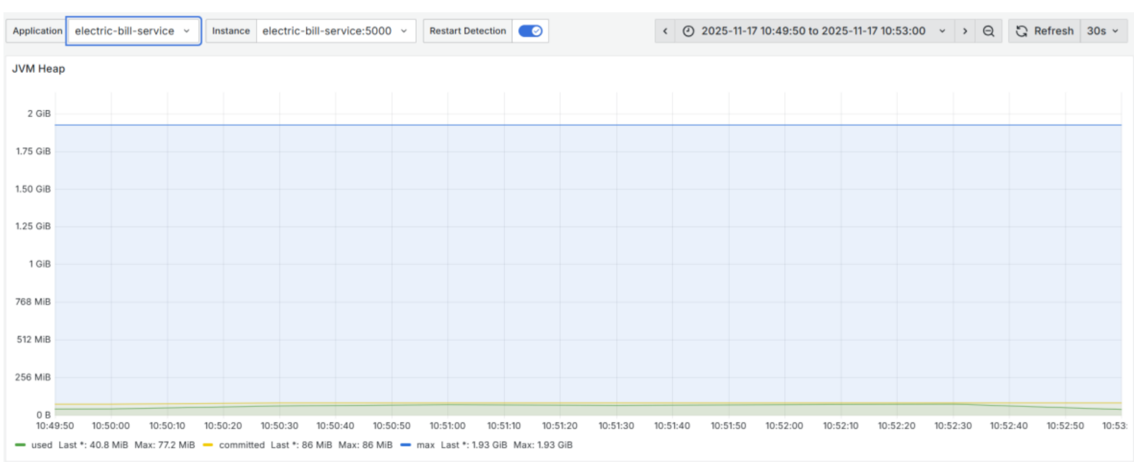
c) JVM Heap

En la

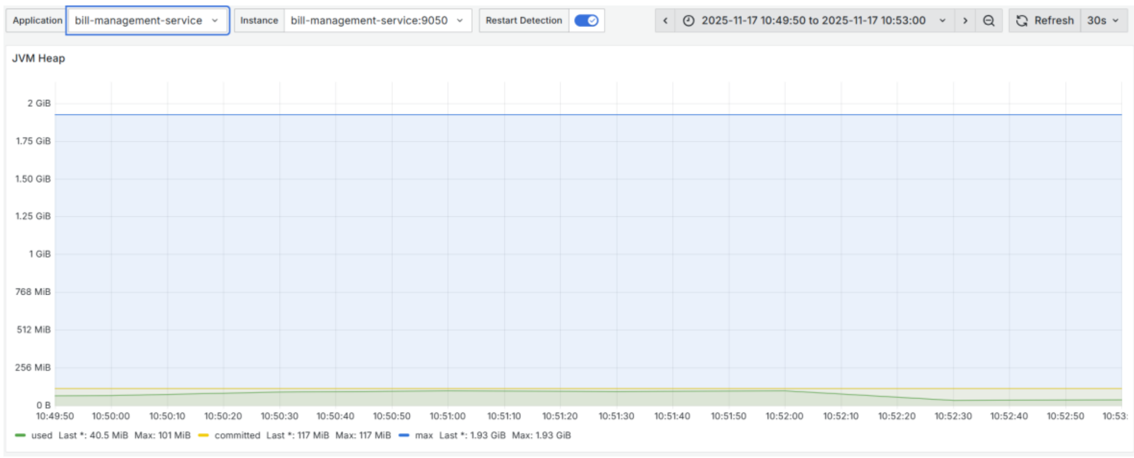
Figura 17, la línea azul casi 2 GB es la memoria máxima que los microservicios pueden usar. La línea amarilla muestra la memoria que la aplicación reservó, aumentando a 193 MB al haber más actividad. La línea verde indica la memoria usada activamente, que también subió, pero se mantiene muy por debajo de los límites. Esto significa que los microservicios tienen mucha memoria disponible y funcionan bien.

Figura 17 JVM Heap en el Escenario 1

a) Microservicio bill-management-service



b) Microservicio electric-bill-service



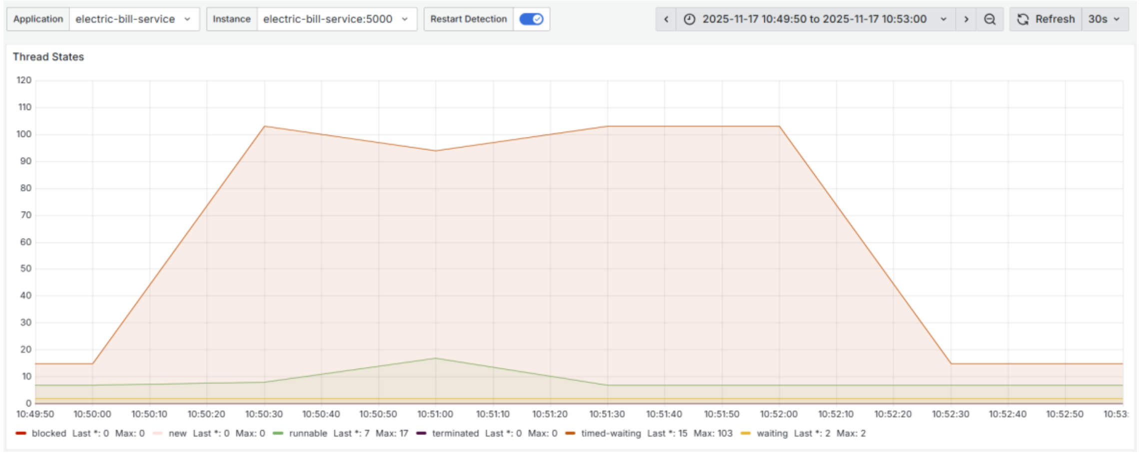
Nota. Datos obtenidos de Grafana durante la ejecución de la prueba de carga en K6.

c) Estados del Hilo

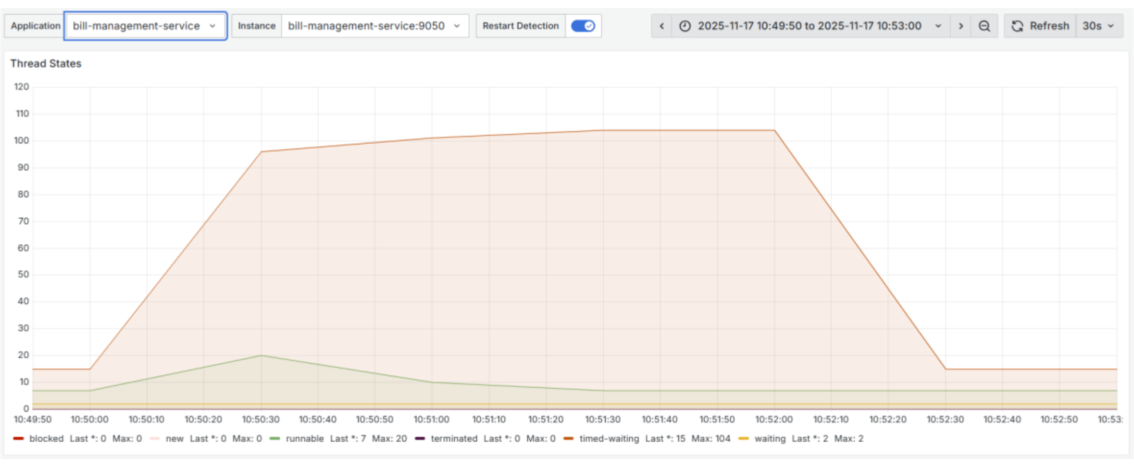
Figura 18, se observa que las gráficas de ambos microservicios presentan comportamientos muy similares, lo que evidencia una fuerte dependencia entre ellos. Esto implica que, ante una degradación en uno de los servicios, el impacto podría propagarse al otro, generando un efecto en cascada. La leyenda "time waiting" indica valores entre 103 y 104, indicando que los hilos están esperando la finalización de una operación con un tiempo de espera definido. Dado que esta es una prueba de línea base con Toxiproxy en modo pass-through, es probable que estos hilos estén esperando la respuesta de las dependencias externas.

Figura 18 Estados del Hilo en el Escenario 1

a) Microservicio bill-management-service



b) Microservicio electric-bill-service



Nota. Graficas obtenidas en Grafana

Una vez recolectadas las métricas y contrastadas con las gráficas obtenidas en Grafana, se confirma que bajo condiciones normales de operación el sistema mantiene un comportamiento estable: no se presentan errores, la latencia se mantiene dentro de rangos aceptables y los recursos del microservicio no muestran señales de saturación.

No obstante, el análisis detallado de los hilos del sistema evidencia que una parte de las peticiones entra en estado "time_waiting". Si bien este comportamiento es normal, resulta relevante mencionarlo ya que anticipa en escenarios con mayor latencia o degradación externa, este número puede incrementarse y convertirse en un factor de bloqueo que afecte la capacidad del sistema para procesar nuevas solicitudes.

5.2. Resultados E2: Sin Circuit Breaker con latencia

En la Tabla 16, se observa un incremento significativo en la latencia cuando la API externa responde lentamente. Al no contar con un mecanismo de aislamiento, la degradación se propaga internamente, afectando directamente la capacidad de respuesta de los microservicios y evidenciando un fallo en cascada.

Tabla 16 Resultado de las métricas del Escenario 2

Métrica	**Latencia 1000 ms**	**Latencia 3000ms**
Latencia promedio	1.08 s	3.09 s
Tiempo de respuesta p (90)	1.15 s	3.26 s
Tiempo de respuesta p (95)	1.32 s	3.31 s
Latencia máxima (max)	1.94 s	3.46 s
Tasa de error	0%	0%
Iteraciones procesadas	2196	1500
Duración total	60.4 s	62.0 s
Iteraciones promedio por segundo	47.01 req/s	24.20 req/s

Los resultados confirman la hipótesis respecto que al introducir latencia artificial en la API externa, el sistema mantuvo un 100 % de disponibilidad funcional todas las peticiones devolvieron estado HTTP 200 y no se registraron errores. Sin embargo, los tiempos de respuesta y el rendimiento global se degradaron de forma proporcional a la latencia impuesta.

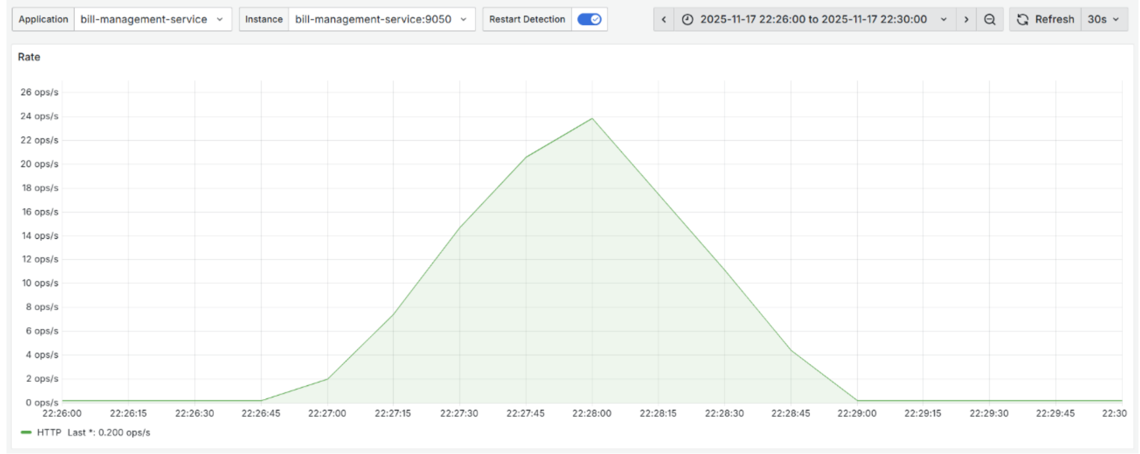
Gráficas complementarias en Grafana

a) Transacciones por segundo, latencia 3000ms

En la Figura 19, se verifica que cuando la latencia es 3000 ms es descenso de las iteraciones disminuye 24.20 indica que las solicitudes quedan en espera durante el tiempo de respuesta de la API externa, ocupando hilos del pool y limitando la capacidad concurrente.

Este comportamiento es consistente con el inicio de bloqueos parciales dentro de los microservicios dependientes.

Figura 19 Transacciones por segundo del microservicio bill-management-service en el Escenario 2



Nota. Datos obtenidos de Grafana durante la ejecución de la prueba de carga en K6.

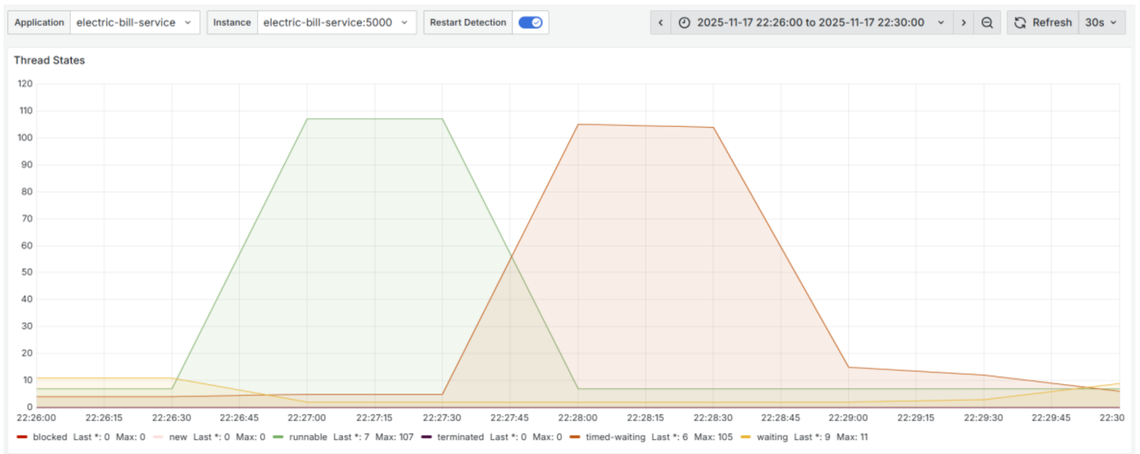
b) Estados del Hilo

En la Figura 20, dado que el microservicio `bill-management-service` depende de `electric-bill-service`, y este último depende del API externo, la latencia de 3000ms provoca que el bloqueo de hilos en un servicio se propague al siguiente.

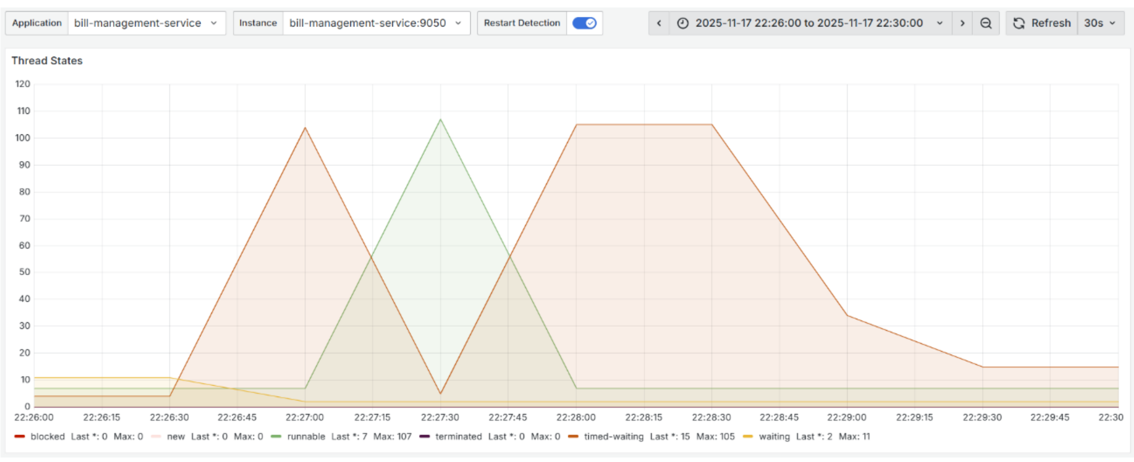
Esto se evidencia porque ambos servicios presentan picos similares en estado “runnable” tienen valores hasta 107 y en “timed-waiting” 105. Esto demuestra que las arquitecturas basadas en microservicios son especialmente vulnerables a la latencia de sus dependencias externas, ralentizado los tiempos de respuesta y afectando el rendimiento general del sistema.

Figura 20 Estados del Hilo en el Escenario 2

a) Microservicio bill-management-service



b) Microservicio electric-bill-service



En conjunto, estos resultados confirman la hipótesis planteada: en ausencia de mecanismos de resiliencia, la latencia elevada en una dependencia externa genera bloqueos, reduce la capacidad de procesamiento del sistema y produce degradación significativa, incluso sin presencia explícita de errores HTTP.

5.3. Resultados E3: Circuit Breaker con latencia

Antes de analizar las métricas gráficas, la Tabla 17 resume los valores obtenidos por k6 durante la ejecución. Esta tabla permite observar el impacto directo del Circuit Breaker sobre la latencia, iteraciones por segundo y éxito de las solicitudes en comparación con el escenario sin resiliencia (E2). En este caso el valor que considera del circuito es slow-call-rate-threshold mayor o igual al 50%.

Tabla 17 Resultado de las métricas del Escenario 3

Métrica	Latencia 3000ms
Latencia promedio	94.59 ms
Tiempo de respuesta p (90)	160.64 ms
Tiempo de respuesta p (95)	201.00 ms
Latencia máxima (max)	3491.24 ms
Tasa de error	0%
Iteraciones procesadas	121269
Duración total	2m20s
Iteraciones promedio por segundo	740.46 req/s

Estos resultados evidencian un rendimiento significativamente superior en comparación con el escenario 2, donde no se aplicó el patrón Circuit Breaker, a pesar de que la dependencia externa presenta la misma latencia. En este caso, la latencia promedio es de 111.13 ms, muy inferior a los 3.02 s registrados en el escenario 2. Esto se debe a la configuración de una respuesta fallback, que permite ofrecer una respuesta inmediata sin esperar el tiempo de latencia de la dependencia externa.

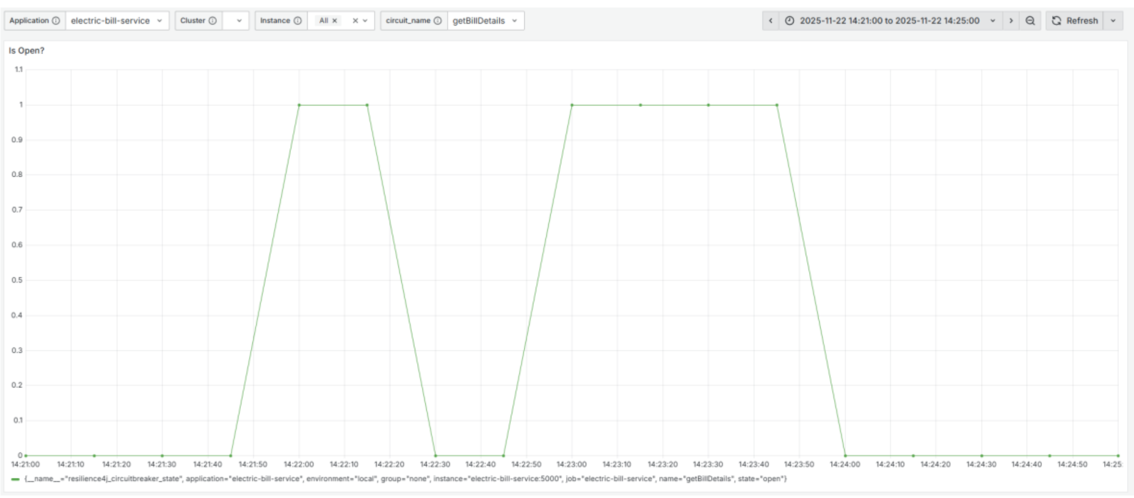
Gráficas complementarias en Grafana

a) Estados del patrón Circuit Breaker

En la Figura 21, se observa que el Circuit Breaker se abre en dos ocasiones durante la prueba. En la gráfica, el valor “1” representa el estado “ABIERTO”y el valor “0” indica el estado “CERRADO”. La apertura ocurre con muy poco tiempo entre cada evento, lo cual evidencia que la API externa no puede manejar la carga sostenida. Como consecuencia, los hilos del microservicio permanecen en estado TIMED-WAITING, comportamiento que coincide con las gráficas de estados de hilos analizadas previamente.

El Circuit Breaker está funcionando correctamente al proteger el microservicio; sin embargo, la causa raíz del comportamiento es la degradación del servicio externo bajo carga y un timeout configurado demasiado alto, lo que ocasiona picos de latencia y periodos prolongados de espera interna antes de que el circuito entre en estado OPEN.

Figura 21 Estados del Circuit Breaker en el Escenario 3



Nota. Gráfica obtenida de Grafana que representa la variación del estado del Circuit Breaker durante la ejecución del escenario de carga.

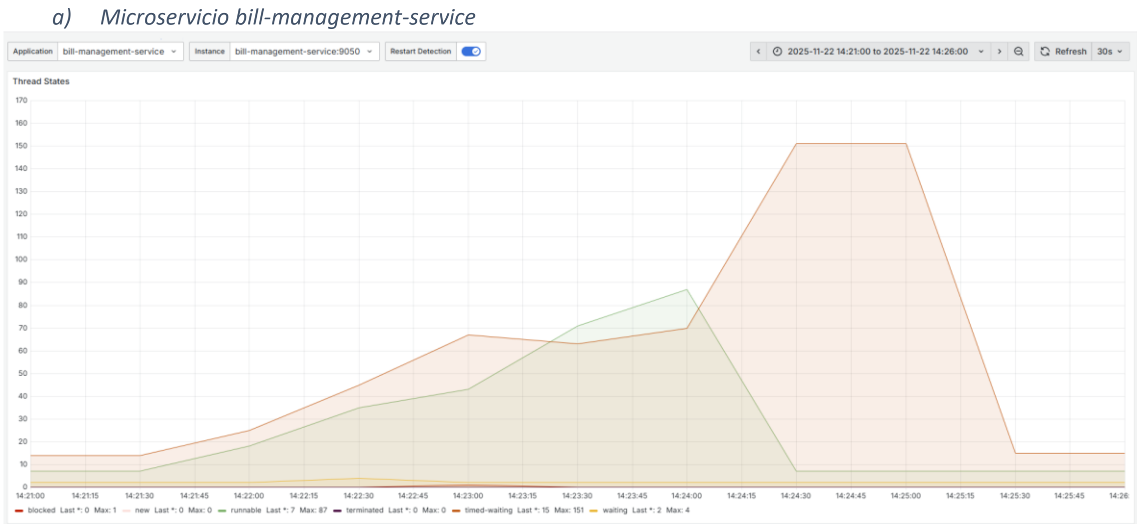
b) Estados de Hilo

En la Figura 22, se muestra que los estados de los hilos difieren en la figura a) se aprecia que presenta hilos “runnable”, mientras la opción b al depender del api externa la mayor cantidad e hilos está en “time waiting”.

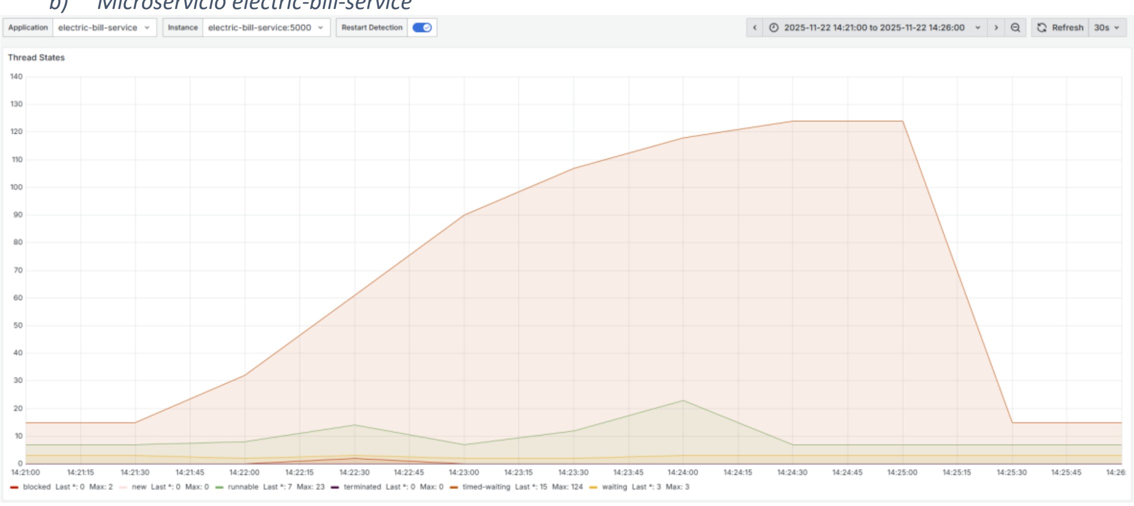
Esto indica que la presencia del Circuit Breaker no elimina el “time-waiting” porque mientras no supere el umbral del campo “slow-call-duration-threshold”, los hilos quedan esperando.

Figura 22 Estados del Hilo en el Escenario 3

a) *Microservicio bill-management-service*



b) *Microservicio electric-bill-service*



Se verifica que la hipótesis se cumple, ya que el patrón si está detectando respuestas lentas y abriendo el circuito, sin embargo, por la alta concurrencia, es posible que se necesite ajustar los valores de los umbrales para evitar aperturas innecesarias.

5.4. Resultados E4: Circuit Breaker ante errores HTTP 5xx

En la Tabla 18 evidencia cómo se comporta el sistema cuando se presentan fallos en la API externa, en esta ocasión el umbral que se considera para abrir el circuito es “failure-rate-threshold” sea igual o mayor al 50% de las llamadas realizadas.

Tabla 18 Resultado de las métricas del Escenario 4

Métrica	Latencia 3000ms
Latencia promedio	129.63 ms
Tiempo de respuesta p (90)	227.78 ms
Tiempo de respuesta p (95)	292.13 ms
Latencia máxima (max)	2.93 s
Tasa de error	0%
Iteraciones procesadas	82076
Duración total	2m20s
Iteraciones promedio por segundo	586.23 req/s

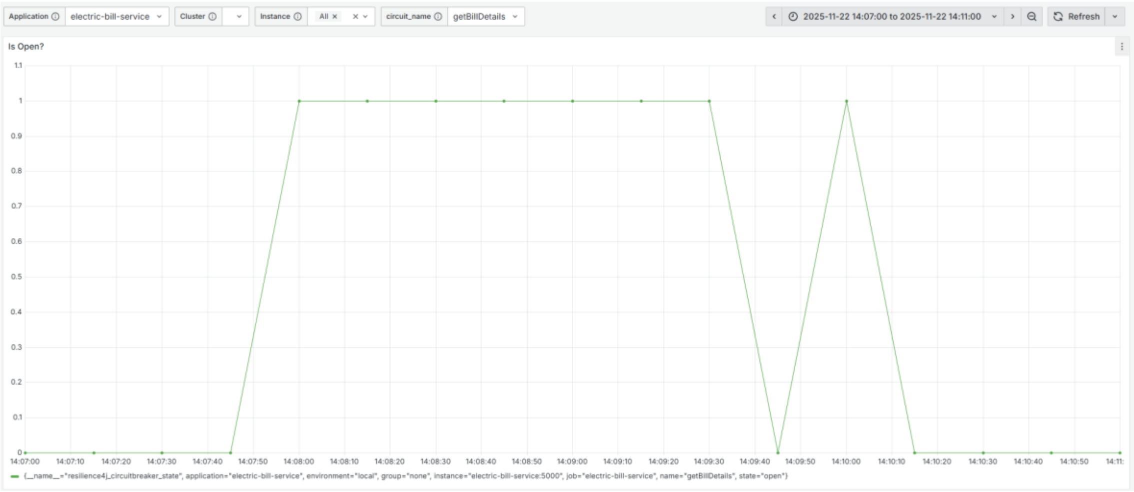
Se obtiene una latencia promedio de 129.63 ms lo que indica que en su mayor parte el circuito estuvo abierto e invoco al fallback, retornando una respuesta inmediata. Además, se observa incremento en las iteraciones procesadas, a pesar de que el servicio externo responde con demasiada latencia.

Graficas complementarias en Grafana

a) Circuit Breaker

En la Figura 23 se observa que durante la prueba que casi desde que inicio las pruebas el circuito se abrió y permaneció de esta manera hasta que la prueba finalizará, indicando que el api externo no estaba disponible.

Figura 23 Estados del Circuit Breaker en el Escenario 4

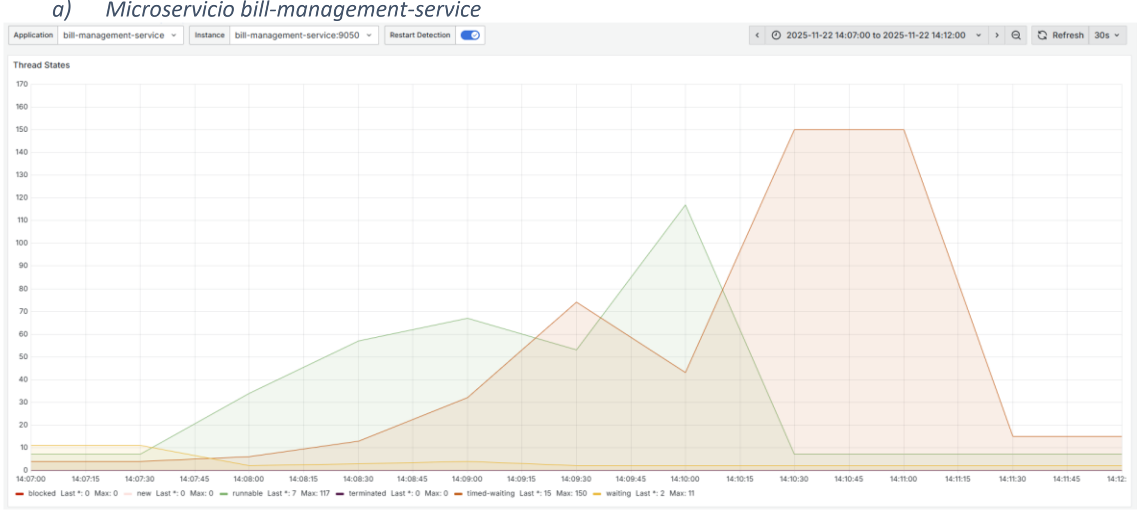


b) Estados de Hilo

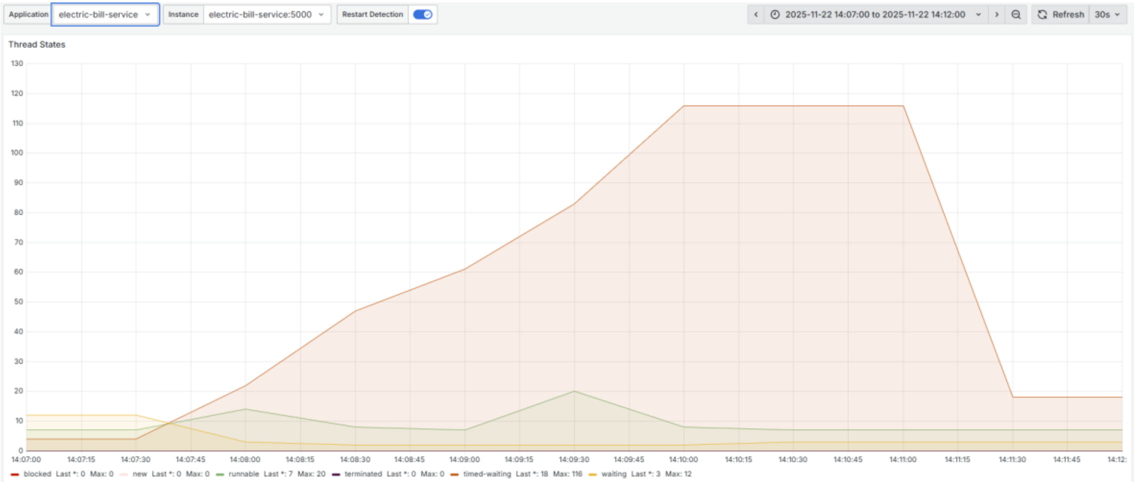
En este caso se observa que las gráficas tienen un comportamiento parecido al del escenario 3, a pesar no presentar lentitud, el servicio externo no está disponible, lo que hace que el circuito se abra y empiece a responder con la respuesta fallback.

Figura 24 Estados de hilos en ejecución

a) Microservicio bill-management-service



b) Microservicio electric-bill-service



Efectivamente como indica la hipótesis el circuito se abre y la respuesta es inmediata alcanzando un promedio 129.63 ms, no presenta latencia, sin embargo, retorna errores 500 indicando que el servicio no está disponible.

5.5. Resultados E5: Circuit Breaker con ajuste de parámetros

Este escenario tiene como objetivo evaluar cómo diferentes configuraciones del Circuit Breaker influyen en el rendimiento del sistema bajo condiciones de degradación constante. A diferencia de los escenarios anteriores, donde se comparaba la presencia o ausencia del patrón, en este caso se analizan tres configuraciones específicas para determinar cuál ofrece el mejor equilibrio entre detección de fallos, estabilidad operacional.

Tabla 19 Resultado de las métricas del Escenario 5

Métrica	CB Estricto	CB Equilibrado	CB Tolerante
Latencia promedio	24.81ms	85.86ms	125.45ms
Tiempo de respuesta p (90)	33.15ms	55.89ms	48.1ms
Tiempo de respuesta p (95)	57.79ms	134.44ms	292.2ms
Latencia máxima (max)	2.79s	4.01s	4.13s
Tasa de error	0.03%	0.65%	1.08%
Iteraciones procesadas	144409	136241	131661
Duración total	17m0s	17m0s	17m0s
Iteraciones promedio por segundo	141.56 req/s	133.45 req/s	128.78 req/s

El análisis comparativo de la Tabla 19 evidencia que la configuración estricta del Circuit Breaker proporciona el mejor desempeño global, reduciendo significativamente la latencia y la tasa de error, mientras incrementa el throughput(iteraciones promedio por segundo) efectivo. A medida que el Circuit Breaker se vuelve más tolerante, se observan mayores tiempos de respuesta, incremento de errores y disminución del volumen de solicitudes procesadas, lo que confirma que la sensibilidad del patrón impacta directamente en la estabilidad y capacidad operativa del sistema.

Gráficas complementarias en Grafana

CB Estricto

a) Estados del Circuit Breaker

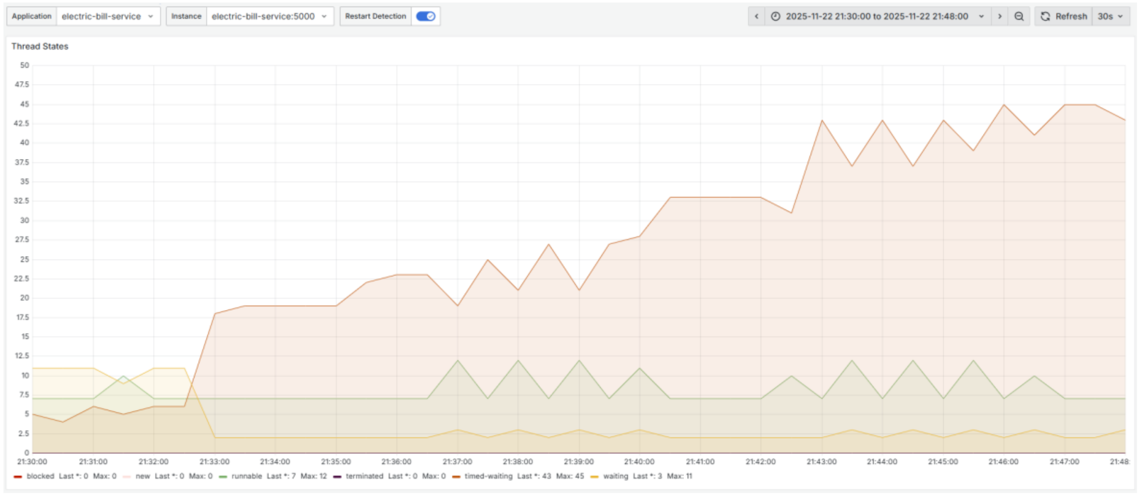
En la Figura 25, al tener un umbral de 20% y un tiempo de espera corto de 6s, el circuito se abre rápidamente ante degradaciones leves o errores, activando el mecanismo de fallback casi de inmediato. Esta configuración ayuda a cortar el tráfico peligroso antes de que se convierta en un problema grave.

Figura 25 Estados del Circuit Breaker del Escenario 5 – umbral 20 %

b) Estado del Hilo

En la Figura, se verifica que tiene pocos hilos en estado “time waiting” siendo entre 43 y 45, esto es debido a la sensibilidad del umbral y el tiempo de espera en abierto de 6s, enviando una respuesta promedio de 24.81 ms.

Figura 26 Estado del Hilo en el Escenario 5 - Umbral 20%

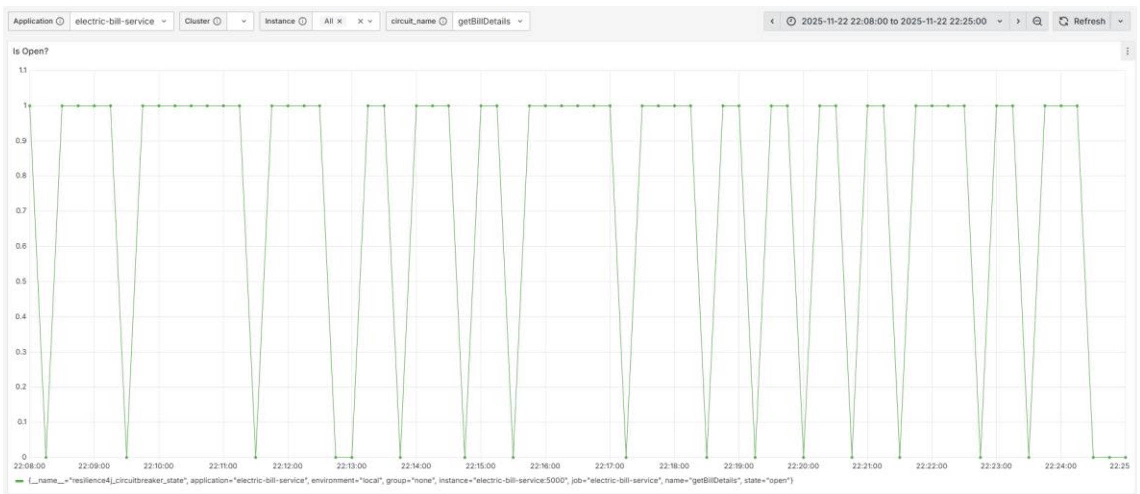


CB Equilibrado

a) Estados del Circuit Breaker

En la Figura 27, la duración en estado “ABIERTO” será de aproximadamente 8 s de acuerdo a la configuración, lo cual da un poco más de “tiempo de recuperación” antes de probar de nuevo. En “SEMI ABIERTO”, se permitirán más llamadas para prueba y puede más fácilmente cerrar si la dependencia ha sanado. Se ve un gráfico equilibrado que permite fallas moderadas y espera un poco más antes de reintentar, lo que disminuye las oscilaciones extremas.

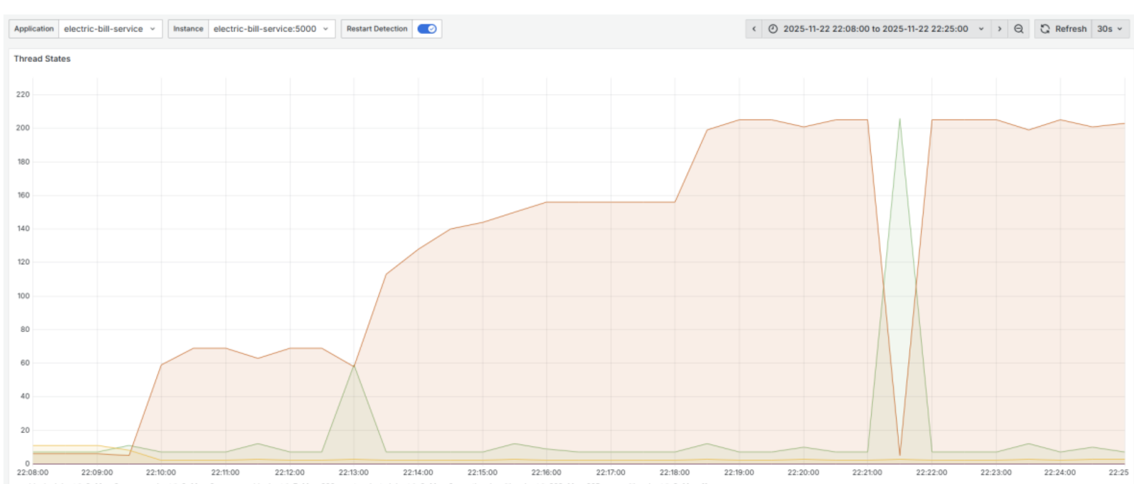
Figura 27 Estados del Circuit Breaker del Escenario 5 – 50 %



b) Estado del Hilo

En la Figura 28, este caso el umbral aplica el 50% para abrir el circuito, y 8 segundos para mantenerlo en abierto, parece ser funcional ya que espera un tiempo prudente en el caso de que este reiniciándose el servidor o base de datos. Se observa que los hilos se están acumulando lo que hace que aumente la latencia y reduzca el número de iteraciones.

Figura 28 Estado del Hilo en el Escenario 6 - Umbral 50%

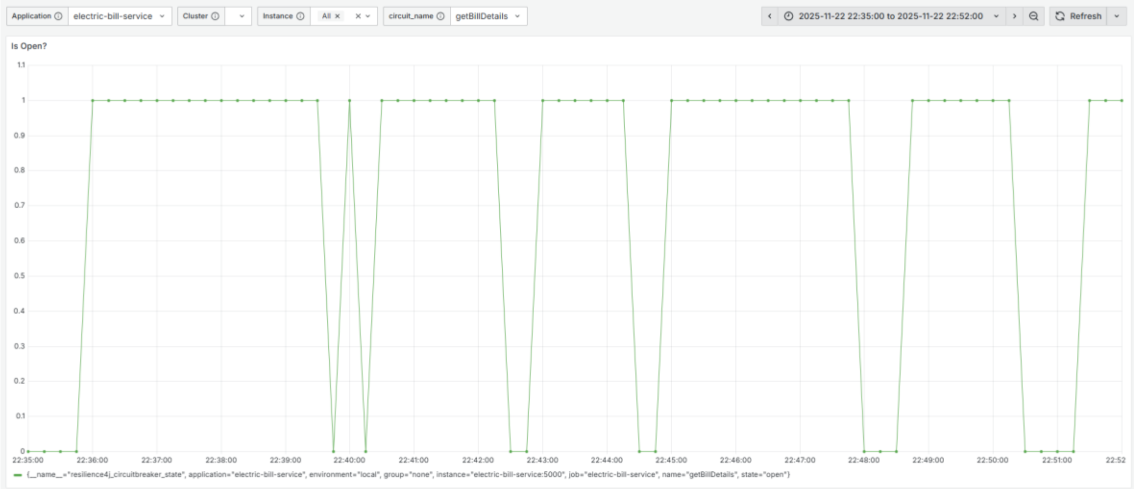


CB Tolerante

a) Estados del Circuit Breaker

En la Figura 29, se verifica que el circuito se abre en momentos de degradación muy severa ya que se necesita que 80% de llamadas sean fallidas/lentas para disparar. Cuando el Circuit Breaker se abre y permanece así por 10 s.

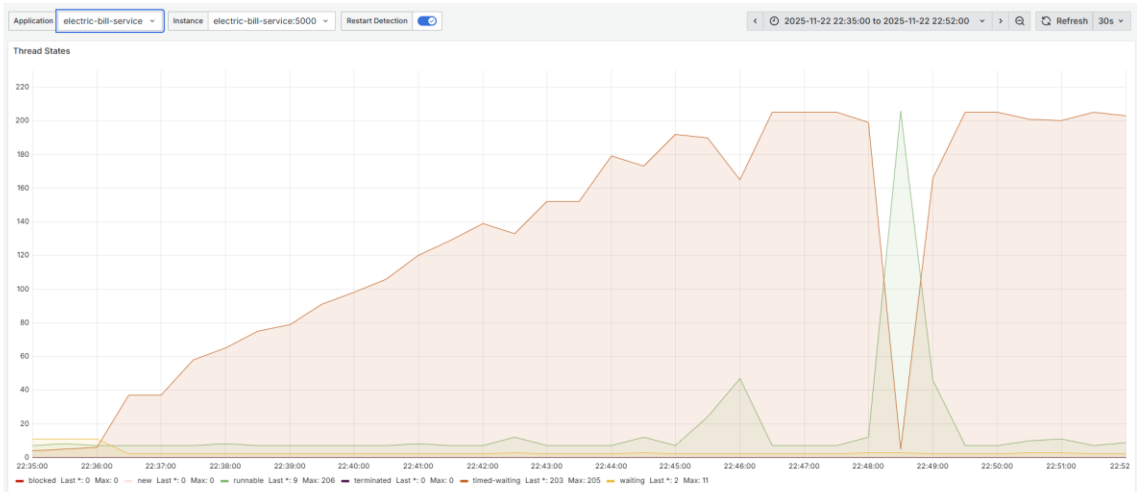
Figura 29 Estados del Circuit Breaker del Escenario 5 - 80%



b) Estado del Hilo

En la Figura 30, se observa que al ser más permisivo el circuit breaker esperando a que fallen el 80% de las peticiones, permitió que una mayor cantidad de tráfico intentara contactar al servicio degradado, provocando tiempos de espera innecesarios “time-waiting” y saturando parcialmente los hilos antes de abrir el circuito.

Figura 30 Estado del Hilo en el Escenario 5 - Umbral 80%



5.6. Discusión

Los resultados obtenidos en esta investigación permiten validar empíricamente la hipótesis de que la implementación del patrón Circuit Breaker es un mecanismo eficaz para garantizar la estabilidad operativa en arquitecturas de microservicios. A continuación, se analizan los hallazgos desde una perspectiva arquitectónica, contrastando la vulnerabilidad de los sistemas acoplados frente a la robustez de los sistemas resilientes.

- **Fragilidad del acoplamiento en Comunicaciones Síncronas**

El análisis del Escenario 2, sin Circuit Breaker, confirmó que la mayor fragilidad de los microservicios basados en HTTP/REST no es el fallo explícito, sino la latencia. Los datos mostraron que una degradación en la dependencia externa provocó un aumento de la latencia interna hasta 3.31 segundos (p95), correlacionado directamente con la saturación del pool de hilos en estado TIMED_WAITING.

Este fenómeno valida lo expuesto por Nygard (2018) sobre los "fallos en cascada": el microservicio dependiente no colapsó por un error de lógica, sino por agotamiento de recursos. Al quedarse esperando respuestas que no llegaban, el sistema perdió su capacidad de procesar nuevas solicitudes, evidenciando que, sin aislamiento, el acoplamiento temporal se convierte en un vector crítico de inestabilidad.

- **Eficiencia de la Estrategia "Fallo Rápido" y Protección de Recursos**

En contraste, el Escenario 3 demostró cómo el patrón Circuit Breaker altera el comportamiento del sistema bajo estrés. La reducción de la latencia promedio a ~95 ms, incluso cuando la dependencia externa tardaba 3000 ms, confirma la eficacia de la estrategia de "Fallo Rápido" (Fail Fast).

Desde el punto de vista arquitectónico, el valor del patrón no reside en "arreglar" la dependencia externa, sino en actuar como un escudo para el servicio interno. Al abortar las peticiones de manera proactiva, se impidió el bloqueo de hilos,

manteniendo el uso de CPU y memoria en niveles normales. Esto demuestra que la resiliencia no solo mejora la experiencia del usuario final evitando esperas interminables, sino que preserva la salud de la infraestructura.

- **Disponibilidad vs Funcionalidad**

Los resultados del Escenario 4 evidencian que el sistema priorizó la disponibilidad operativa por encima de la funcionalidad completa. A pesar de que la API externa se encontraba totalmente inoperativa (errores 5xx), el sistema mantuvo un throughput estable gracias al uso de respuestas de contingencia (fallback).

Si bien estas respuestas representan una degradación funcional, el usuario no obtiene información actualizada como el valor real de la factura debido a la indisponibilidad del servicio externo. En su lugar, recibe una respuesta genérica o un dato previamente almacenado en caché, lo que constituye una respuesta técnicamente exitosa, pero incompleta desde el punto de vista funcional.

Este comportamiento se alinea con los principios de degradación elegante (Graceful Degradation), en los cuales el sistema opta por ofrecer una versión reducida o limitada de la funcionalidad antes que interrumpir totalmente el servicio. En otras palabras, se privilegia la continuidad y disponibilidad del sistema frente al riesgo de colapso o indisponibilidad generalizada.

- **Importancia del Ajuste de Parámetros**

El Escenario 5 pone de manifiesto la complejidad de configurar un mecanismo de corte en sistemas distribuidos. Se observa que existe una correlación inversa entre la sensibilidad del circuito y la estabilidad del tráfico: a mayor sensibilidad (configuración estricta), menor tolerancia al ruido de la red, generando falsos positivos. Inversamente, una baja sensibilidad (configuración tolerante) incrementa el tiempo de exposición a fallos, permitiendo que se acumule deuda técnica en forma de hilos bloqueados. Este hallazgo discute la idea de que el Circuit Breaker es una solución "lista para usar", confirmando que requiere un análisis profundo del comportamiento del tráfico para evitar efectos colaterales indeseados.

6. Conclusiones

1. En el Escenario 2 se comprobó que, en ausencia de Circuit Breaker, la latencia externa de 3 segundos se propagó hacia los microservicios dependientes, elevando la latencia interna hasta 3.31 segundos en el percentil 95 (p95) y reduciendo de manera crítica el rendimiento. Este comportamiento se asoció al incremento de hilos en estado TIMED_WAITING, lo que evidencia que un sistema distribuido sin mecanismos de aislamiento es altamente susceptible a fallos en cascada y saturación de recursos, comprometiendo su estabilidad operativa.
2. Ante la caída total o degradación severa de la dependencia externa, el Circuit Breaker actuó eficazmente abriendo el circuito y ejecutando la estrategia de fallos rápido. Esto permitió devolver respuestas de contingencia controladas y reducir la latencia promedio a niveles de aproximadamente 292 ms en contraste con los tiempos de espera agotados del escenario sin protección. Se concluye que el patrón prioriza exitosamente la salud y estabilidad de recursos por encima del éxito funcional de las transacciones individuales.
3. El análisis comparativo del Escenario 5 evidenció que no existe una configuración universal. Mientras la configuración estricta minimizó la saturación de hilos, generó falsos positivos que limitaron la operatividad, por el contrario, la configuración tolerante (umbral 80%) expuso al sistema a acumulaciones peligrosas de hilos en espera. Se determinó que la configuración equilibrada (50%) ofreció la mayor estabilidad, confirmando que la eficacia del Circuit Breaker depende críticamente de un ajuste preciso de umbrales; de lo contrario, el patrón mismo puede convertirse en un punto de fallo o en un cuello de botella.
4. Se concluye que la implementación del patrón Circuit Breaker no puede realizarse de manera aislada o estática. La experimentación demostró que la integración de herramientas de pruebas de carga, junto con el monitoreo en tiempo real mediante Prometheus y Grafana, fue indispensable para identificar los puntos de saturación y calibrar con precisión parámetros críticos como

`slow-call-duration-threshold` y `wait-duration-in-open-state`. Por tanto, la observabilidad no es un añadido opcional, sino un prerrequisito arquitectónico para garantizar que los mecanismos de resiliencia respondan adecuadamente a la variabilidad de las cargas de trabajo reales.

5. Aunque el patrón Circuit Breaker suele asociarse al diseño de arquitecturas desde etapas tempranas, es importante destacar que puede implementarse de manera efectiva en sistemas que ya se encuentran en operación. Para lograrlo, la configuración de umbrales y parámetros debe basarse, idealmente, en el análisis de datos históricos de comportamiento, tales como tiempos de respuesta, tasas de error, volúmenes de carga y patrones de uso. Esta información permite definir valores realistas que reflejen el comportamiento operativo del sistema y minimicen aperturas innecesarias del circuito.

   En escenarios donde no se dispone de datos históricos, la configuración inicial puede alinearse con el área de negocio, considerando el público objetivo de la funcionalidad, los niveles de servicio esperados y las estimaciones de carga transaccional. Esta colaboración permite establecer umbrales coherentes con los objetivos operativos y los acuerdos de nivel de servicio, asegurando que el Circuit Breaker contribuya a la resiliencia del sistema sin afectar la experiencia del usuario o los procesos críticos.

Referencias

Badman, A., & Kosinski, M. (2025). *IBM Think*. Obtenido de IBM Think: https://www.ibm.com/think/topics/application-resiliency

Baeldung. (11 de Mayo de 2024). *Baeldung*. Obtenido de Guide to Resilience4j: https://www.baeldung.com/resilience4j

Baeldung. (8 de Enero de 2024). *Baeldung*. Obtenido de Life Cycle of a Thread in Java: https://www.baeldung.com/java-thread-lifecycle

Beyer, B., Jones, C., Petoff, J., & Niall, R. (2016). *Site Reliability Engineering: How Google Runs Production Systems.* Oslo: O'Reilly Media, Inc.

Bhushan, S. (2025). *Comprehensive Java Programming.* India: BPB Publications.

Blancarte, O. (04 de Diciembre de 2018). *Oscar Blancarte - Software Architecture*. Obtenido de Circuit Breaker pattern: https://www.oscarblancarteblog.com/2018/12/04/circuit-breaker-pattern/

Chakraborty, M., & Pratap Kundan, A. (2021). *Monitoring Cloud-Native Applications.* Gurugram: Apress.

Dasari, H. (2025). Resilience Engineering in Financial Systems: Strategies for Ensuring Uptime During Volatility. *The American Journal of Engineering and Technology*.

Finnovista and Banco Interamericano de Desarrollo and BID Invest. (2024). Fintech en América Latina y el Caribe: un ecosistema consolidado con potencial para aportar a la inclusión financiera regional. *Banco Interamericano de Desarrollo*, 27-34.

Fowler, S. J. (2016). *Production-Ready Microservices.* O'Reilly Media, Inc.

Grafana Labs. (2025). *Grafana Documentation*. Obtenido de https://grafana.com/docs/

Gremlin. (2023). *Gremlin Inc.* Obtenido de Gremlin Documentation: https://www.gremlin.com/docs

Hunt, C., & Binu, J. (2011). *Java™ Performance.* Addison-Wesley Professional.

Indrasiri, I., & Siriwardena, P. (2018). *Microservicios para la empresa: diseño, desarrollo e implementación.* Apress.

IT ahora. (22 de Octubre de 2024). *Indicadores de medición en el sector financiero*. Obtenido de IT ahora: https://itahora.com/2024/10/22/indicadores-de-medicion-en-el-sector-financiero/

Kleppmann, M. (2017). *Designing Data-Intensive Applications.* United States: O'Reilly Media, Inc.

Kumar, V. (9 de Septiembre de 2024). *Key patterns for resiliency in Microservices Architecture*. Obtenido de https://medium.com/techartifact-technology-learning/key-patterns-for-resiliency-in-microservices-architecture-992966edbd67

Larsson, M. (2019). *Hands-On Microservices with Spring Boot and Spring Cloud.* Packt Publishing.

Larsson, M. (2023). Microservices with Spring Boot 3 and Spring Cloud. En *Build resilient and scalable microservices using Spring Cloud, Istio, and Kubernetes* (pág. 706). Packt Publishing.

Mentores Tech. (05 de Febrero de 2025). *Mentores Tech*. Obtenido de Patrones de Observabilidad en Microservicios: https://www.mentorestech.com/resource-blog-content/patrones-de-observabilidad-en-microservicios

Microsoft. (16 de Julio de 2025). *Azure Architecture Center. Microsoft Learn.* Obtenido de Estilo de arquitectura de microservicios: https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices

Musib, S. (2022). *Spring Boot in Practice.* Shelter Island: Manning.

Narra, S. R. (2025). JVM Internals and Its Impact on Application Performance. *ResearchGate*, 17.

Netflix. (2018). *Netflix Open Source*. Obtenido de Chaos Monkey: https://netflix.github.io/chaosmonkey/

Newman, S. (2021). *Building Microservices, Designing Fine-Grained Systems.* O'Reilly Media, Inc.

Newman, S. (2025). *Building Resilient Distributed Systems.* O'Reilly Media, Inc.

Nygard, M. T. (2018). *Release It! Design and Deploy Production-Ready Software* (Second Edition ed.). Raleigh, North Carolina, United States of America: The Pragmatic Programmers, LLC.

Obregon, A. (30 de Marzo de 2023). *Medium*. Obtenido de Introduction to Java's Memory Model — Heap, Stack, and Metaspace: https://medium.com/@AlexanderObregon/introduction-to-javas-memory-model-heap-stack-and-metaspace-ceaeb565921c

Parasoft. (Septiembre de 11 de 2025). *Parasoft*. Obtenido de What Is Performance Testing?: https://www.parasoft.com/learning-center/performance-testing-guide/

Pivotto, J., & Brazil, B. (2023). *Prometheus: Up & Running.* O'Reilly Media, Inc.

Prometheus. (2025). *Prometheus Documentation*. Obtenido de https://prometheus.io/docs/introduction/overview/

resilience4j. (2025). *Resilience4j Documentation*. Obtenido de CircuitBreaker: https://resilience4j.readme.io/docs/circuitbreaker

Richards, M. (2022). *Software Architecture Patterns.* United States of America: O'Reilly Media, Inc.

Richardson, C. (2018). *Microservices Patterns.* Manning Publications.

Rosental, C., & Jones, N. (2020). *Chaos Engineering.* O'Reilly Media, Inc.

Shopify. (18 de Marzo de 2025). *GitHub - Shopify/ToxiProxy: a TCP proxy to simulate network and system conditions for chaos and resiliency testing*. Obtenido de https://github.com/Shopify/toxiproxy

Sithec. (30 de Agosto de 2024). *La Importancia de los Microservicios en el Desarrollo de Software*. Recuperado el 2025 de 4 de 6, de https://es.linkedin.com/pulse/la-importancia-de-los-microservicios-en-el-desarrollo-software-4edoe

Spring. (2025). *Spring Boot Documentation*. Obtenido de Spring.io / Spring Docs: https://docs.spring.io/spring-boot/

Suárez, S. L. (2023). Análisis de patrones de resiliencia en una arquitectura basada en microservicios. *SEDICI*.

Yorkston, K. (2021). *Performance Testing.* Ware: Apress.