UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gabriel Antônio de Oliveira Leite

Resiliência em Microsserviços: Análise dos Padrões Circuit Breaker e Retry

Uberlândia, Brasil

2026

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gabriel Antônio de Oliveira Leite

Resiliência em Microsserviços: Análise dos Padrões Circuit Breaker e Retry

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Orientador: Marcelo de Almeida Maia

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2026

Gabriel Antônio de Oliveira Leite

Resiliência em Microsserviços: Análise dos Padrões Circuit Breaker e Retry

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 10 de agosto de 2026:

Marcelo de Almeida Maia
Orientador

Fabiano Azevedo Dorça

Leandro Nogueira Couto

Uberlândia, Brasil
2026

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Faculdade de Computação
Av. João Naves de Ávila, 2121, Bloco 1B - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4393 - www.facom.ufu.br - facom@ufu.br

ATA DE DEFESA - GRADUAÇÃO

Curso de Graduação em:	Sistemas de Informação				
Defesa de:	FACOM31802 - Trabalho de Conclusão de Curso 2				
Data:	10/08/2026	Hora de início:	17h	Hora de encerramento:	17:40h
Matrícula do Discente:	12111BSI222				
Nome do Discente:	Gabriel Antônio de Oliveira Leite				
Título do Trabalho:	Resiliência em Microsserviços: Análise dos Padrões Circuit Breaker e Retry				

A carga horária curricular foi cumprida integralmente?	(x) Sim ( ) Não

Reuniu-se na plataforma Teams , da Universidade Federal de Uberlândia, a Banca Examinadora composta pelo prof. Dr. Fabiano Azevedo Dorça - FACOM/UFU, pelo prof. Dr. Leandro Nogueira Couto - FACOM/UFU e pelo prof. Dr. Marcelo de Almeida Maia - FACOM/UFU, orientador(a) do(a) candidato(a).

Iniciando os trabalhos, o(a) presidente da mesa, Dr. Marcelo de Almeida Maia, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao discente a palavra para a exposição do seu trabalho. A duração da apresentação do discente e o tempo de arguição e resposta foram conforme as normas do curso.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o(a) candidato(a):

(x) Aprovado Nota: 95

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Marcelo de Almeida Maia**, **Professor(a) do Magistério Superior**, em 10/08/2026, às 17:52, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Fabiano Azevedo Dorça**, **Professor(a) do Magistério Superior**, em 10/08/2026, às 17:52, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Leandro Nogueira Couto**, **Professor(a) do Magistério Superior**, em 10/08/2026, às 17:52, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **7582753** e o código CRC **9CC1992A**.

Referência: Processo nº 23117.055192/2026-27 SEI nº 7582753

À minha mãe, pela resiliência e esforço que me tornaram o homem que sou.

Ao meu pai, pelo apoio nos momentos em que mais necessitei.

Aos meus irmãos, pelos tantos conselhos e conversas que me ajudaram a crescer.

A toda a minha família, pelo suporte incondicional que tornou o caminho possível.

Agradecimentos

À minha mãe, cujo esforço incansável e resiliência foram a base da formação de nossa família; sua doação e sacrifícios constantes para que eu e meus irmãos pudéssemos alcançar o ensino superior e a realização de nossos objetivos são a maior prova de seu amor e dedicação.

Ao meu pai, pelo suporte silencioso e por estender a mão exatamente nos momentos de maior incerteza e necessidade ao longo desta caminhada.

Aos meus irmãos, que se tornaram um dos meus principais pilares de apoio durante a graduação; são fonte de inspiração, e sou grato por tê-los ao meu lado. As nossas conversas e conselhos foram essenciais para manter a minha estabilidade, foco e motivação ao longo de toda a jornada.

À tia Rosa, pelo acolhimento com o afeto de uma avó que não pude vivenciar na infância; ao meu avô Adão e aos meus tios, que sempre garantiram que nada nos faltasse. Agradeço a toda a minha família pelo suporte incondicional proporcionado desde a nossa infância e por nos acolherem e amarem como a extensão de seus próprios lares; essa união transformou os desafios do percurso em um caminho viável e realizável.

Aos amigos que cultivei durante estes anos na universidade, que me apresentaram o real significado da amizade. Sou profundamente grato pelas risadas, pelas conversas sinceras sobre a vida e o futuro, pelos momentos simples compartilhados e por uma parceria que transformou a rotina acadêmica em uma jornada leve, deixando memórias que todos nós guardaremos com o mesmo carinho.

Ao Professor Marcelo Maia, pela orientação precisa, paciência e condução cirúrgica ao longo do desenvolvimento deste projeto, cuja visão foi fundamental para a consolidação do trabalho.

À Faculdade de Computação (FACOM) e à Universidade Federal de Uberlândia (UFU), pela excelência na formação acadêmica e pelas portas que este ambiente me abriu. Agradeço imensamente pelas oportunidades de crescimento e por todo o ecossistema universitário que me permitiu evoluir não apenas como profissional, mas como cidadão.

Resumo

O presente trabalho analisa a eficácia de uma **arquitetura de resiliência híbrida, composta pela integração sinérgica dos padrões Circuit Breaker e Retry**, na mitigação de falhas em cascata e esgotamento de conexões (*thread exhaustion*) em microsserviços. A dependência síncrona entre serviços distribuídos cria vulnerabilidades onde a latência de um único componente pode comprometer a disponibilidade de todo o sistema. Para avaliar essa abordagem, utilizou-se uma **metodologia experimental quantitativa** aplicada a um ecossistema conteinerizado via **Docker Compose**, desenvolvido em **Java com Spring Boot 3 e Spring Cloud (Netflix Eureka)**. A arquitetura híbrida foi implementada de forma desacoplada por meio de **Programação Orientada a Aspectos (AOP) com a biblioteca Resilience4j**, estruturando hierarquicamente o *Retry* como aspecto interno sob o escudo do *Circuit Breaker* como aspecto externo. O ambiente foi submetido a testes sistemáticos de estresse, latência controlada e injeção de falhas com a ferramenta **k6**, sob perfis de carga escalonados de até 300 usuários virtuais simultâneos divididos em **estágios de *warm-up*, *ramping-up* e *cool-down***.

Os resultados demonstraram que, sob saturação extrema, o sistema desprotegido colapsou rapidamente devido à retenção de threads, superando latências de 4,4 segundos. Em contrapartida, a **atuação da arquitetura híbrida** com o disjuntor em modo *fail-fast* interceptou as chamadas em memória, impedindo o enfileiramento de requisições no Tomcat. Essa abordagem preservou a estabilidade da infraestrutura, limitando a latência de cauda (p_{95}) a 241,84 ms e sustentando uma vazão (*throughput*) de 1.820,10 requisições por segundo, com um *overhead* computacional de aspecto de aproximadamente 20 ms sob falhas contínuas. Adicionalmente, atestou-se a capacidade de auto-recuperação (*self-healing*) da arquitetura, que processou mais de 220 mil chamadas durante a transição dinâmica de estados, comprovando a hipótese de que o isolamento híbrido e estrutural de falhas garante a continuidade e a confiabilidade de sistemas distribuídos em nuvem.

Palavras-chave: Microsserviços; Padrões de resiliência; Circuit Breaker; Retry; Esgotamento de threads, Falhas em cascata; Programação Orientada a Aspectos; Resilience4j.

Abstract

This work analyzes the effectiveness of a **hybrid resilience architecture, composed of the synergistic integration of the Circuit Breaker and Retry patterns**, in mitigating cascading failures and thread exhaustion in microservices. The synchronous dependency between distributed services creates vulnerabilities where the latency of a single component can compromise the entire system. To evaluate this approach, a **quantitative experimental methodology** was executed on a containerized ecosystem using **Docker Compose**, developed with **Java, Spring Boot 3, and Spring Cloud (Netflix Eureka)**. The hybrid architecture was decoupled from the business logic via **Aspect-Oriented Programming (AOP) with the Resilience4j library**, hierarchically structuring the Retry (internal aspect) under the Circuit Breaker shield (external aspect). The environment was subjected to systematic stress, controlled latency, and fault injection tests using the **k6** tool, under load profiles of up to 300 simultaneous virtual users divided into ***warm-up*, *ramping-up*, and *cool-down* stages**.

The experimental results demonstrate that, under extreme saturation, the unprotected system collapsed due to thread exhaustion, reaching latencies exceeding 4.4 seconds. In contrast, the **hybrid architecture** operating in *fail-fast* mode intercepted calls in memory, avoiding thread queuing in Tomcat. This approach preserved infrastructure stability, keeping the tail latency (p_{95}) controlled at 241.84 ms and sustaining a throughput of 1,820.10 requests per second, with an AOP execution overhead of approximately 20 ms. Additionally, the architecture's self-healing capacity was verified, processing over 220,000 requests during dynamic state transitions, proving the hypothesis that structured, hybrid fault isolation ensures the continuity and reliability of cloud-based distributed systems.

Keywords: Microservices; Resilience patterns; Circuit Breaker; Retry; Thread exhaustion; Cascading failures; Aspect-Oriented Programming; Resilience4j.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AOP	*Aspect-Oriented Programming* (Programação Orientada a Aspectos)
API	*Application Programming Interface* (Interface de Programação de Aplicações)
ARCB	*Auto Retry Circuit Breaker*
CB	*Circuit Breaker*
CPU	*Central Processing Unit* (Unidade Central de Processamento)
CTMC	Cadeias de Markov de Tempo Contínuo
DNS	*Domain Name System* (Sistema de Nomes de Domínio)
FIT	*Fault Injection Testing*
FSM	*Finite State Machine* (Máquina de Estados Finita)
gRPC	*gRPC Remote Procedure Calls*
HTTP	*Hypertext Transfer Protocol*
I/O	*Input/Output* (Entrada e Saída)
IP	*Internet Protocol*
JDK	*Java Development Kit*
JVM	*Java Virtual Machine* (Máquina Virtual Java)
OSI	*Open Systems Interconnection*
POO	Programação Orientada a Objetos
RAM	*Random Access Memory* (Memória de Acesso Aleatório)
REST	*Representational State Transfer*
SDK	*Software Development Kit* (Kit de Desenvolvimento de Software)
VU	*Virtual User* (Usuário Virtual)

Sumário

1 Introdução

A adoção do padrão arquitetural de microsserviços tem se consolidado na engenharia de software devido à sua capacidade de promover escalabilidade e independência no desenvolvimento de serviços autônomos (PAHL; JAMSHIDI, 2016). Impulsionada pela evolução da computação em nuvem e pela conteinerização, essa arquitetura descentralizada substitui aplicações monolíticas rígidas por redes de serviços cooperativos, otimizando fluxos de integração e entrega contínua (SILL, 2016). No entanto, esta decomposição introduz complexidades inerentes a sistemas distribuídos, e a literatura atual aponta que garantir a resiliência de rede tornou-se um dos principais desafios nessas implementações (FALAHAH; SURENDRO; SUNINDYO, 2021).

Ao migrar de sistemas monolíticos para arquiteturas distribuídas, a escolha do modelo de comunicação é determinante para a confiabilidade global do ecossistema (NEWMAN, 2021). A literatura categoriza essas interações nos paradigmas síncrono e assíncrono, recomendando cada modelo para cenários operacionais distintos (TANENBAUM; STEEN, 2017).

O paradigma assíncrono, estruturado sobre intermediários de mensageria (*message brokers*) como Apache Kafka ou RabbitMQ, é altamente recomendado para tarefas de longa duração, gravações intensivas ou fluxos que toleram consistência eventual (NEWMAN, 2021). Casos de uso típicos incluem a emissão e transmissão de notas fiscais eletrônicas, a geração de relatórios analíticos complexos ou o envio de e-mails de confirmação, em que o usuário não necessita aguardar a conclusão física da tarefa para continuar navegando. Embora esse modelo ofereça excelente desacoplamento temporal, ele eleva a complexidade de desenvolvimento ao exigir o tratamento de transações distribuídas, idempotência e monitoramento de atrasos nas filas (*consumer lag*) (ZHOU et al., 2021).

Por outro lado, o paradigma síncrono (como chamadas REST sobre HTTP ou gRPC) é a escolha indicada para operações de leitura em tempo real e transações que exigem consistência forte imediata (NEWMAN, 2021). Casos de uso clássicos envolvem a autorização imediata de transações no checkout, validação de credenciais de login, consultas instantâneas de saldo ou a verificação de estoque em tempo real. A despeito de sua simplicidade e adequação para fluxos interativos, a comunicação síncrona impõe um forte acoplamento temporal que resulta em um dos problemas mais críticos e documentados nesses ecossistemas: a vulnerabilidade a falhas em cascata (*cascading failures*) (DUEñAS-OSORIO; VEMURU, 2009).

Na prática, quando um serviço provedor (*downstream*) sofre com alta latência, os

componentes chamadores (*upstream*) não falham imediatamente, mas mantêm suas conexões retidas aguardando uma resposta. Esse acúmulo contínuo de requisições esgota prematuramente o *pool* de *threads* do servidor (como o Apache Tomcat). Consequentemente, a lentidão de um único componente propaga-se, tornando todo o sistema indisponível e exigindo a adoção de padrões que restrinjam o escopo do erro (MOLCHANOV; ZHMAIEV, 2018).

Delimita-se, portanto, que este trabalho atua estritamente no paradigma síncrono. O objetivo não é defender a substituição de fluxos síncronos por assíncronos, mas sim avaliar mecanismos de resiliência híbrida capazes de blindar serviços síncronos tradicionais contra o colapso por esgotamento de conexões, garantindo a viabilidade prática desse modelo amplamente adotado no mercado.

1.1 Problema e Hipótese

A problemática central deste trabalho reside na necessidade de implementar mecanismos que permitam a um sistema manter-se resiliente sob condições adversas. Em testes preliminares, observou-se que a lentidão de um microsserviço propaga-se integralmente para o usuário final, travando a aplicação *upstream* e comprometendo a disponibilidade de todo o fluxo.

Para mitigar esse cenário, a literatura propõe a adoção de padrões de resiliência. O padrão *Retry* atua na recuperação de falhas transientes, reexecutando automaticamente chamadas de rede instáveis antes de retornar um erro ao cliente (PUNITHAVATHY; PRIYA, 2024). Já o padrão *Circuit Breaker (CB)* atua como um mecanismo de proteção de infraestrutura: ao detectar que um serviço dependente ultrapassou um limiar de falhas ou lentidão, ele "abre o circuito"e bloqueia novas requisições, promovendo o retorno imediato de erro (*fail-fast*) (FALAHAH; SURENDRO; SUNINDYO, 2021).

Dessa forma, a hipótese desta pesquisa é que a combinação estrutural de falha rápida e retentativas controladas restringe o raio de impacto de um serviço degradado, reduzindo drasticamente a latência de cauda ($p95$) e preservando a capacidade de processamento (*throughput*) da arquitetura global.

1.2 Objetivos

O objetivo geral deste estudo é avaliar experimentalmente o comportamento dinâmico e a capacidade de auto-recuperação de microsserviços sob estresse, analisando o impacto dos padrões *Circuit Breaker* e *Retry* na mitigação do esgotamento de recursos do servidor de aplicação.

Os objetivos específicos incluem:

- Implementar uma bancada de testes distribuída para realização de um *benchmark* utilizando a biblioteca *Resilience4j* em uma arquitetura *Spring Boot 3* isolada;
- Analisar o comportamento do servidor padrão de aplicação do *Spring Boot 3*, *Apache Tomcat*, sob carga progressiva extrema para forçar o esgotamento de conexões (*thread exhaustion*);
- Avaliar o impacto do padrão *Retry* isolado e em composição, mensurando sua eficácia na mitigação de falhas transientes e o risco arquitetural de amplificação de carga (*retry storm*) em cenários de degradação contínua;
- Investigar experimentalmente o comportamento da máquina de estados (*Open*, *Half-Open* e *Closed*) do *CB* frente à degradação e posterior recuperação de um serviço *downstream*;
- Realizar uma análise comparativa entre o sistema em estado desprotegido e sob o efeito da estratégia *fail-fast*, avaliando métricas de latência de cauda ($p95$) e vazão máxima (*throughput*) em cenários de estresse;

1.3 Justificativa

A justificativa deste estudo fundamenta-se na natureza crítica da disponibilidade em arquiteturas de microsserviços, onde a interdependência entre componentes distribuídos amplia a superfície de falhas e o risco de degradação sistêmica em cascata. A relevância científica reside na necessidade de implementar estratégias que garantam a resiliência estrutural através de mecanismos de falha rápida (*fail-fast*) e auto-recuperação, assegurando que anomalias em serviços individuais não comprometam a estabilidade global do ecossistema (RASHEEDH; SARADHA, 2022).

Sob a perspectiva metodológica, a investigação é motivada pela carência de parâmetros determinísticos para a configuração de políticas de resiliência em ambientes produtivos. A injeção sistemática de falhas emerge como uma técnica essencial para validar a robustez de sistemas distribuídos sob condições severas de estresse, permitindo observar o comportamento real de recursos críticos, como o esgotamento do *pool* de *threads* (YU et al., 2025; HEORHIADI et al., 2016). Ao quantificar o impacto de latências e falhas em um ambiente controlado, este trabalho busca preencher a lacuna entre as modelagens teóricas de confiabilidade e os desafios da engenharia de software contemporânea, fornecendo uma base experimental para a construção de sistemas distribuídos tolerantes a falhas (MENDONCA et al., 2020).

1.4 Resultados Esperados

Espera-se confirmar experimentalmente a ineficácia de um sistema não protegido, o qual deverá apresentar latências diretamente proporcionais ao atraso injetado, culminando no esgotamento prematuro de recursos do servidor. Projeta-se também que a aplicação isolada do padrão *Retry* se mostrará incapaz de mitigar gargalos puramente temporais, e tenderá a agravar a saturação da infraestrutura quando exposta a falhas explícitas, atuando como um amplificador de carga. Em contrapartida, ao introduzir a resiliência híbrida, espera-se que o *Retry* melhore a taxa de recuperação inicial mitigando perdas transientes (PUNITHAVATHY; PRIYA, 2024). Posteriormente, projeta-se que o *CB*, ao abrir o circuito e isolar a falha de forma preventiva, garantirá uma redução drástica na latência observada pelo cliente (*fail-fast*), priorizando a resposta de erro controlado em detrimento do estrangulamento da aplicação.

Como contribuição metodológica complementar e garantia de reprodutibilidade científica (YU et al., 2025), o pacote de replicação completo deste experimento — contendo o código-fonte dos microsserviços desenvolvidos, os scripts de injeção de carga e estresse no k6, os logs coletados e os arquivos de orquestração do Docker Compose — está disponível publicamente no GitHub através do link: <https://github.com/zgabrieloliveira/microservice-resilience-benchmark>.

2 Fundamentação Teórica

2.1 Arquitetura de Microsserviços

O desenvolvimento de software moderno tem migrado massivamente do modelo monolítico para a arquitetura de microsserviços. Esta abordagem estrutura uma aplicação como uma coleção de serviços fracamente acoplados e organizados em torno de capacidades de negócio (PAHL; JAMSHIDI, 2016). Ao contrário dos monólitos, onde a invocação de métodos ocorre em espaço de memória compartilhado, os microsserviços comunicam-se através de protocolos de rede leves (como *Hypertext Transfer Protocol / Representational State Transfer* - HTTP/REST ou *gRPC Remote Procedure Calls* - gRPC), o que permite escalonamento horizontal direcionado e poliglotismo tecnológico (SILL, 2016).

Contudo, essa transição transfere a complexidade da aplicação para a infraestrutura de rede. A latência, que em um monólito é da ordem de nanossegundos, passa a ser medida em milissegundos, exigindo que o engenheiro de software trate a comunicação entre os nós como inerentemente instável e sujeita a falhas.

2.2 A Complexidade da Distribuição e as Falácias da Rede

Apesar dos benefícios arquiteturais, a transição para sistemas distribuídos introduz desafios intrínsecos de confiabilidade. O principal erro arquitetural na concepção de microsserviços ocorre quando engenheiros assumem que a comunicação de rede entre dois componentes possui o mesmo determinismo de uma chamada de método local em memória.

Essa desconexão entre expectativa e realidade foi formalizada nas clássicas "Oito Falácias da Computação Distribuída", originalmente postuladas por L. Peter Deutsch na Sun Microsystems, e amplamente consolidadas na literatura fundamental de sistemas distribuídos por Tanenbaum e Steen (2017). Dentre as falácias arquiteturais que induzem sistemas ao erro, três são diretamente mitigadas por padrões de resiliência (DEUTSCH et al., 1994):

1. A premissa de que a rede é confiável;
2. A ilusão de que a latência é zero;
3. A crença de que a largura de banda é infinita.

Ao projetar a comunicação entre microsserviços ignorando essas premissas — assumindo falsamente que a rede não falha e que o tempo de resposta é imediato —, o sistema torna-se suscetível ao risco mais severo da interdependência: a ocorrência de falhas em cascata (*cascading failures*). Esse fenômeno ocorre quando uma perturbação local em um componente (como um pico de latência que contradiz a segunda falácia) sobrecarrega os componentes adjacentes, causando um colapso sistêmico em cadeia (DUEñAS-OSORIO; VEMURU, 2009).

Em levantamentos industriais sobre depuração e análise de falhas em microsserviços, o bloqueio síncrono resultante de latência externa é frequentemente citado como um dos gatilhos mais comuns para a degradação total do sistema (ZHOU et al., 2021). No contexto de comunicação via rede (como requisições HTTP/REST), quando um serviço chamador (*upstream*) aciona um provedor degradado (*downstream*), as *threads* de execução do cliente ficam retidas no estado de *Input/Output (I/O) Wait* aguardando o retorno. O acúmulo contínuo dessas transações pendentes exaure o *pool* de *threads* do servidor de aplicação (*thread exhaustion*), propagando a falha em cadeia e evidenciando a necessidade crítica de mecanismos de isolamento estrutural (MOLCHANOV; ZHMAIEV, 2018).

É imperativo destacar que o servidor de aplicação padrão do *Spring Boot 3*, o *Apache Tomcat*, opera sob o modelo *Thread-per-Request*. Nesse paradigma, cada requisição HTTP síncrona recebida por um serviço (*upstream*) é vinculada a uma *thread* dedicada de um *pool* limitado durante todo o seu ciclo de vida (VMWARE, INC., 2024). Em cenários de comunicação bloqueante (*Blocking I/O*), caso esse serviço necessite consultar um componente externo, a *thread* permanece retida e inoperante enquanto aguarda o retorno do serviço *downstream*. Embora paradigmas como programação reativa ou o advento das *Virtual Threads* (Java 21) ofereçam alternativas para escalabilidade não bloqueante, o modelo tradicional síncrono permanece como o padrão predominante em sistemas corporativos e integrações legadas, tornando-os criticamente vulneráveis ao esgotamento de recursos frente a latências não nulas (ZHOU et al., 2021; FALAHAH; SURENDRO; SUNINDYO, 2021).

2.3 Padrões de Resiliência e Tolerância a Falhas

Para contornar o efeito cascata e garantir a alta disponibilidade, a literatura de engenharia de software propõe a adoção de Padrões de Resiliência. O objetivo primário desses padrões não é evitar que a falha ocorra — uma vez que falhas de rede são inevitáveis —, mas sim isolar o componente degradado e preservar a saúde geral do ecossistema (MOHAMMAD, 2025).

2.3.1 O Padrão Circuit Breaker

O *CB* (Disjuntor) é um padrão de projeto estrutural que atua como um *proxy* monitorando as chamadas a um serviço remoto. Inspirado na engenharia elétrica, seu propósito é interromper operações que têm alta probabilidade de falhar, poupando recursos computacionais e promovendo o princípio do *fail-fast* (falha rápida) (FALAHAH; SURENDRO; SUNINDYO, 2021).

O padrão é implementado como uma máquina de estados finitos que transita entre três estados principais:

- **Fechado (*Closed*):** Estado normal de operação. O tráfego flui livremente para o serviço de destino. O *CB* monitora as chamadas registrando os resultados (sucessos, falhas ou lentidão) em uma janela deslizante baseada em um volume fixo de requisições.
- **Aberto (*Open*):** Quando a proporção de falhas atinge um limiar crítico dentro dessa janela de observação (ex: 50% de erro nas últimas 100 requisições avaliadas), o circuito "abre". Todas as requisições subsequentes são interceptadas e rejeitadas imediatamente com uma exceção local (*fail-fast*), sem acionar o serviço remoto. Isso preserva os recursos da aplicação *upstream* e permite que o serviço degradado se recupere sem ser sobrecarregado.
- **Meio-Aberto (*Half-Open*):** Após um intervalo de retenção configurável, o circuito transita para este estado de sondagem, permitindo a passagem de um número estritamente limitado de requisições de teste. Se a taxa de sucesso dessa amostra for aceitável, o circuito retorna ao estado Fechado, redefinindo as métricas. Em caso de nova falha, o circuito volta ao estado Aberto, reiniciando o temporizador de bloqueio.

Para além da transição de estados, o funcionamento lógico do interceptor durante o processamento de uma requisição HTTP é detalhado no fluxograma da Figura 1. Esta representação destaca o mecanismo de interrupção precoce, que impede que as *threads* do servidor fiquem retidas aguardando respostas de serviços degradados.

2.3.1.1 Mecanismos de Janela Deslizante

A precisão de um *CB* na detecção de falhas depende do mecanismo de coleta de métricas, comumente implementado através de janelas deslizantes (*sliding windows*). Na biblioteca *Resilience4j*, existem dois modelos principais de janelas para a contabilização de resultados (MOLCHANOV; ZHMAIEV, 2018):

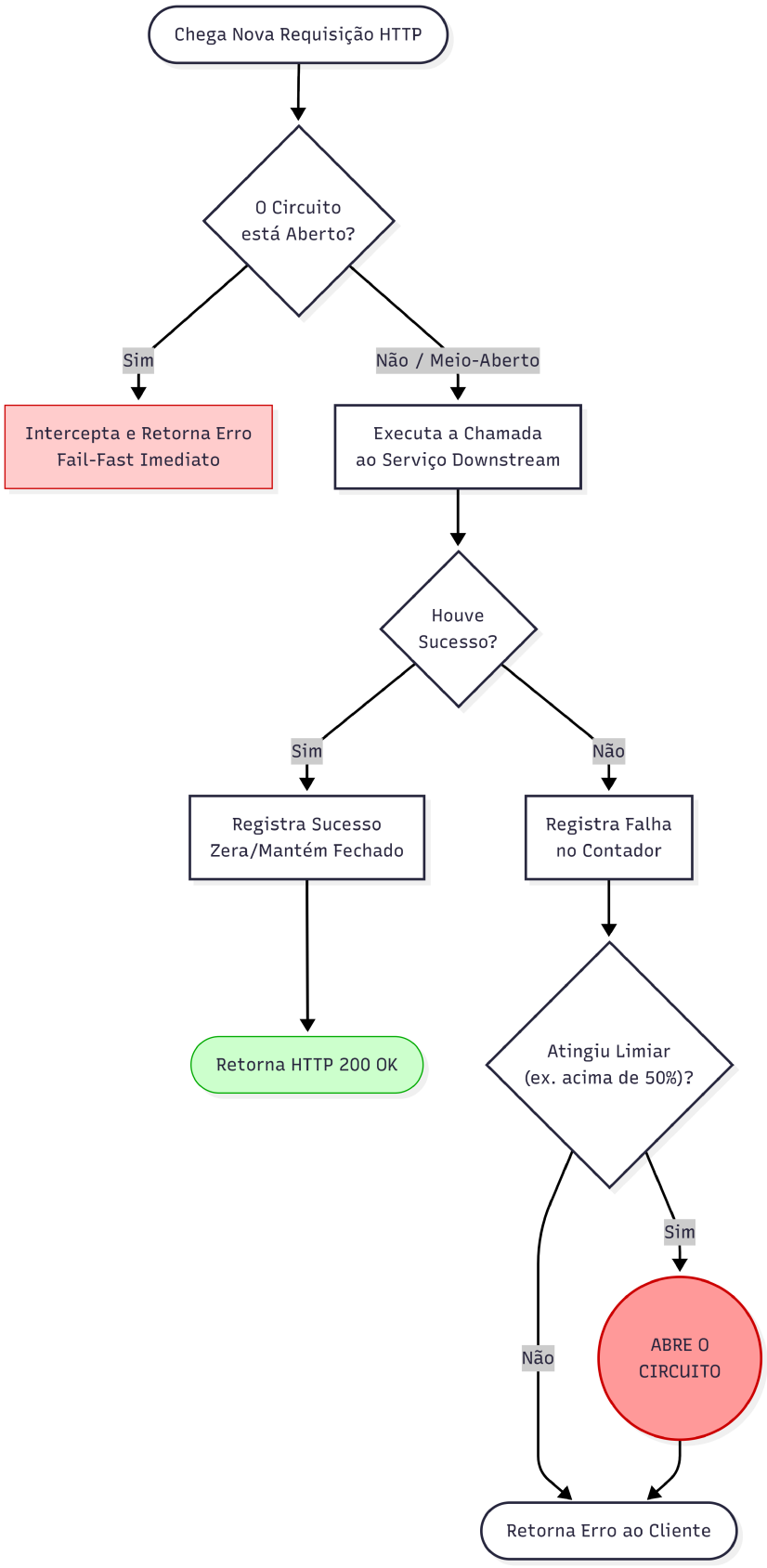

Figura 1 – Fluxograma de Decisão e Lógica de *Fail-Fast* do *CB*.

Fonte: Elaborado pelo autor (2026).

- **Janela Baseada em Contagem (*Count-based*):** Monitora o resultado das últimas N requisições, independentemente do tempo decorrido. A decisão de transição de estado é tomada assim que a N-ésima chamada é registrada, sendo ideal para cenários onde a previsibilidade estatística e o volume fixo de amostras são prioritários.
- **Janela Baseada em Tempo (*Time-based*):** Avalia as requisições ocorridas nos últimos T segundos. Nesse modelo, o estado do circuito pode mudar mesmo sem novas chamadas, caso amostras antigas expirem e saiam da janela temporal. É indicado para ambientes de altíssima vazão onde falhas transientes podem ser diluídas no tempo.

A escolha entre os modelos de janela impacta diretamente o determinismo do sistema frente a picos isolados de latência ou erros estruturais persistentes (FALAHAH; SURENDRO; SUNINDYO, 2021).

2.3.2 O Padrão Retry

Enquanto o *CB* lida com o isolamento de falhas prolongadas, o padrão *Retry* é focado na recuperação de falhas transientes. Falhas transientes são anomalias de curta duração, como uma breve perda de conectividade, um descarte de pacote ou um pico momentâneo de carga no banco de dados (PUNITHAVATHY; PRIYA, 2024).

O *Retry* atua interceptando o erro e reexecutando a operação silenciosamente por um número determinado de vezes. Na literatura e em implementações de mercado, a estratégia padrão utiliza o *Backoff Exponencial*. Nessa abordagem, o intervalo de tempo entre as tentativas aumenta gradativamente para evitar o agravamento da carga em um serviço que já se encontra sobrecarregado (MOHAMMAD, 2025), dinâmica ilustrada na Figura 2.

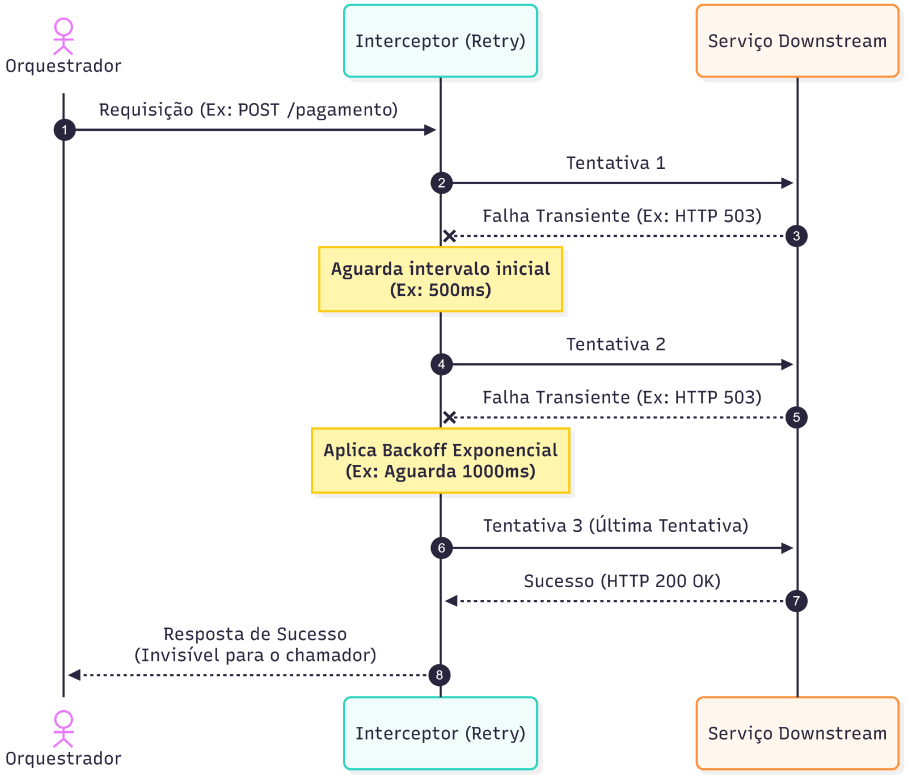

Figura 2 – Diagrama de Sequência: Padrão Retry com *Backoff Exponencial*.

Fonte: Elaborado pelo autor (2026).

Contudo, a adoção de intervalos de espera variáveis e exponenciais introduz flutuações temporais que podem mascarar a latência real em um ambiente de *benchmark*. Por esse motivo, para o escopo e rigor experimental desta pesquisa, optou-se pela utilização do *Retry* com intervalo fixo (*Fixed Delay*), conforme ilustrado na Figura 3.

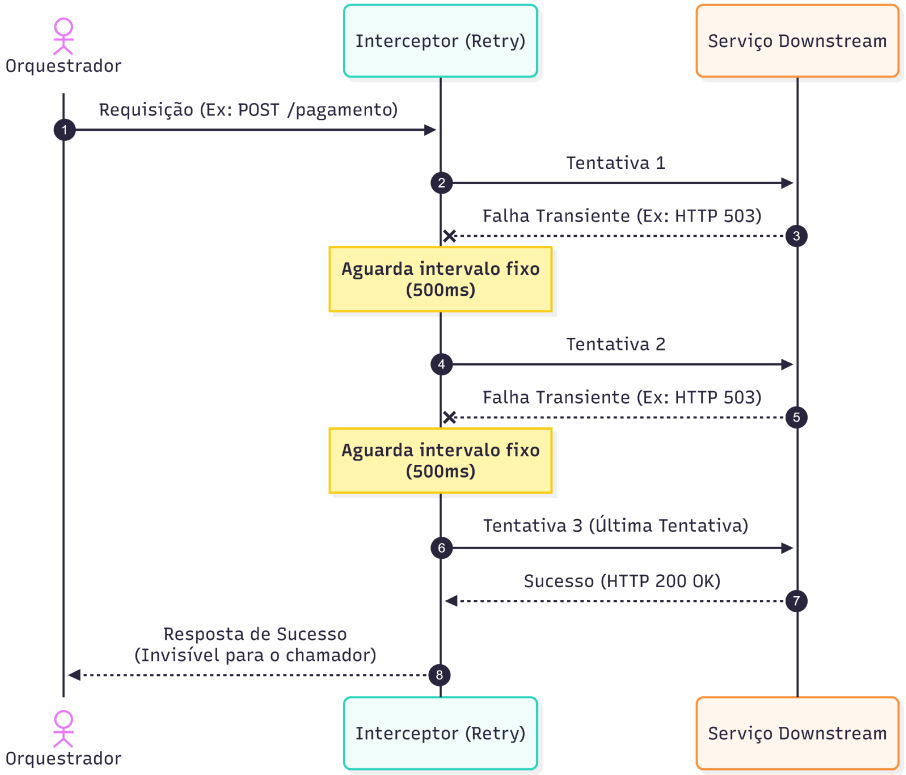

Figura 3 – Diagrama de Sequência: Padrão Retry com Intervalo Fixo (*Fixed Delay*).

Fonte: Elaborado pelo autor (2026).

Essa escolha garante um tempo de espera acumulado estritamente determinístico. Ao remover a variabilidade do tempo de retentativa, assegura-se que a falha e a consequente abertura do *CB* ocorram de forma puramente previsível, isolando as métricas de performance durante os testes.

2.3.3 A Abordagem Híbrida: Sinergia entre Padrões

Estudos recentes demonstram que a aplicação isolada de um único padrão raramente é suficiente para cobrir todos os cenários de degradação. O uso excessivo do *Retry* pode agravar uma sobrecarga (causando tempestades de requisições), enquanto um *CB* agressivo pode rejeitar transações viáveis diante de microinstabilidades (RASHEEDH; SARADHA, 2022).

Portanto, a combinação híbrida destes padrões é a abordagem recomendada para otimizar a tolerância a falhas. Nesse modelo de composição, o *Retry* atua como a primeira linha de defesa, mascarando falhas transientes do usuário final. Caso as tentativas esgotem-se e o erro persista, o *CB* contabiliza essa falha definitiva e abre o circuito, protegendo o serviço *upstream* do esgotamento de recursos e garantindo a estabilidade da arquitetura (PUNITHAVATHY; PRIYA, 2024).

Esta dinâmica de contenção é ilustrada no diagrama de sequência da Figura 4, onde as retentativas esgotadas culminam no acionamento do bloqueio preventivo (*fail-fast*) para as chamadas subsequentes.

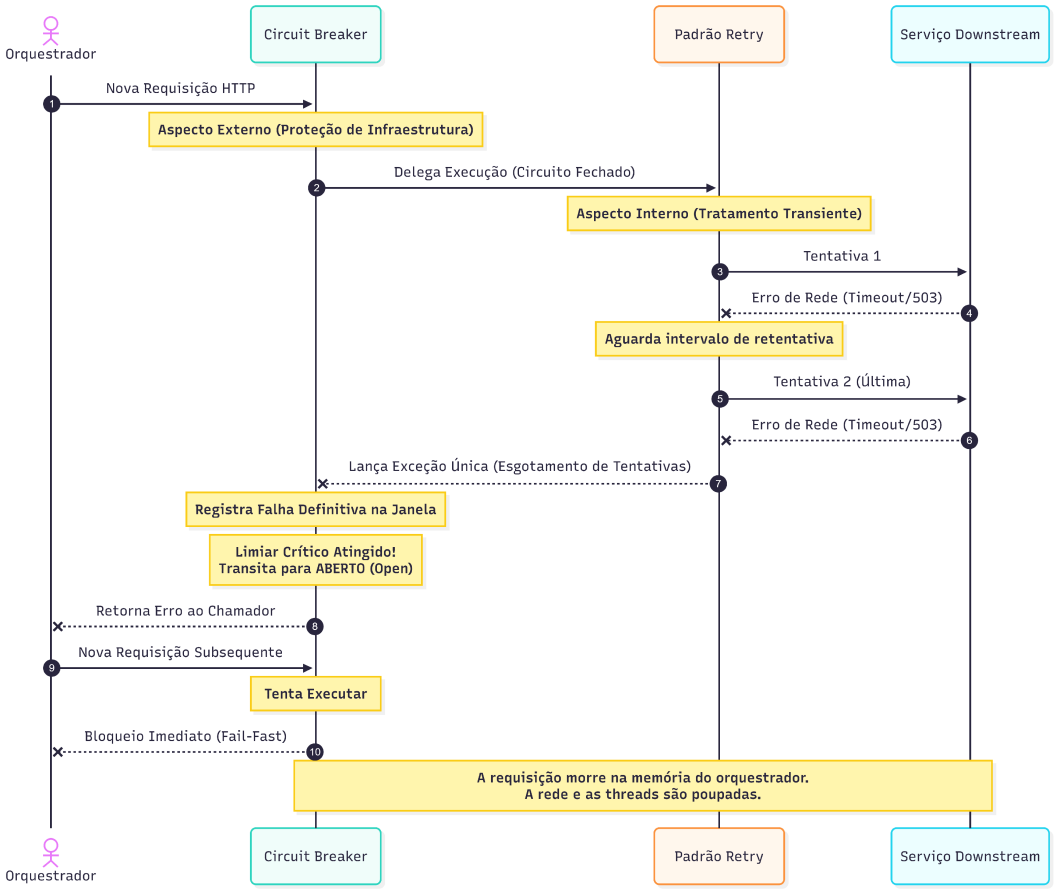

Figura 4 – Diagrama de Sequência ilustrando a Sinergia Híbrida entre *Retry* e *Circuit Breaker*.

Fonte: Elaborado pelo autor (2026).

2.4 Métricas de Avaliação em Sistemas Distribuídos

A validação de resiliência e performance em arquiteturas de microsserviços exige um afastamento das métricas tradicionais de aplicações monolíticas. Isso porque em um ambiente distribuído sujeito a flutuações de rede, a utilização da latência média (média aritmética simples) é considerada uma métrica enganosa, pois valores extremos (*outliers*) decorrentes de falhas transientes distorcem o resultado global, mascarando a real experiência do usuário (PAPAPANAGIOTOU; CHELLA, 2018).

Para garantir o rigor no *benchmarking* de microsserviços, a literatura estabelece a necessidade de analisar a capacidade de processamento aliada à distribuição da latência

através de percentis (ADERALDO et al., 2017). Neste contexto, destacam-se três métricas fundamentais:

- **Latência Mediana ($p50$): A experiência nominal do sistema.** Esta métrica traduz o comportamento esperado da aplicação para a maioria de seus consumidores. Matematicamente, representa o percentil central da distribuição, indicando que 50% das requisições foram processadas em um tempo igual ou inferior a este valor. Ao contrário da média aritmética simples, a latência mediana é estatisticamente robusta para descrever a saúde da arquitetura em condições normais de operação.
- **Latência de Cauda ($p95$ ou *Tail Latency*): O limiar de colapso estrutural.** Esta métrica expõe a gravidade do cenário de falha e o limite de resistência da infraestrutura. Estatisticamente, isola os 5% piores tempos de resposta da amostra. Em cadeias de chamadas síncronas, a latência de cauda dita a taxa de sobrevivência do servidor, pois revela o tempo exato em que as *threads* do serviço *upstream* permanecem sequestradas no estado de *I/O Wait* aguardando serviços *downstream* degradados. É o acompanhamento do $p95$ que determina o estrangulamento da aplicação e justifica a ativação de mecanismos de *fail-fast* (PAPAPANAGIOTOU; CHELLA, 2018).
- **Vazão (*Throughput*): A capacidade volumétrica e de sobrevivência.** Representa a taxa de requisições processadas pelo sistema em um determinado intervalo de tempo (medida em *req/s*). Sob estresse de I/O em ambientes síncronos, a queda brusca do *throughput* é o sintoma primário e definitivo do *thread exhaustion* (ADERALDO et al., 2017). Em cenários protegidos, a manutenção de um *throughput* elevado — mesmo que implicando o retorno de erros 503 instantâneos — é o principal indicador de que o servidor isolou a falha com sucesso e evitou o colapso em cascata (FALAHAH; SURENDRO; SUNINDYO, 2021).

O emprego conjunto destas três métricas fornece o embasamento matemático necessário para auditar o sistema em todo o seu espectro de operação. Enquanto a latência mediana ($p50$) atesta a estabilidade da arquitetura em seu estado nominal, a combinação entre o $p95$ e o *throughput* audita o cenário de colapso estrutural. Sob estresse extremo, o $p95$ quantifica a profundidade da degradação (o tempo de retenção das *threads*), e o *throughput* mede o impacto global (a perda de capacidade de processamento). É a correlação inversa entre estas duas últimas grandezas que permite comprovar experimentalmente a eficácia da abordagem híbrida (*CB* e *Retry*) na preservação da infraestrutura (ADERALDO et al., 2017).

2.5 Trabalhos Relacionados

Apesar do crescente interesse da comunidade científica e da ampla adoção de arquiteturas de microsserviços na indústria, grande parte da literatura sobre padrões de resiliência concentra-se em modelagens probabilísticas teóricas ou em métricas abstratas de alta disponibilidade. Embora pesquisas recentes busquem aproximar a teoria da prática, avaliações experimentais focadas estritamente no esgotamento de recursos das aplicações *upstream* (como o *thread exhaustion*) ainda representam um campo em aberto. Esta seção contextualiza o presente trabalho em relação ao estado da arte, evidenciando sua contribuição científica ao isolar e testar empiricamente esses cenários de falha.

No campo da modelagem probabilística, Mendonca et al. (2020) propuseram uma abordagem baseada em verificação de modelos utilizando Cadeias de Markov de Tempo Contínuo (CTMC) para analisar o comportamento dos padrões *Retry* e *CB*. Embora o estudo consiga prever o comportamento sob diferentes condições de disponibilidade e configurar parâmetros de *timeout* e limites de falha, ele restringe-se a uma representação puramente matemática. O presente trabalho difere dessa abordagem ao avançar da modelagem teórica para a avaliação experimental com injeção de falhas real através da ferramenta k6.

Na vertente de arquiteturas de alta disponibilidade, Rasheedh e Saradha (2022) propuseram o desenvolvimento de microsserviços resilientes utilizando padrões de design híbridos. Os autores argumentam que a combinação de técnicas é essencial para manter a estabilidade em aplicações baseadas em nuvem. No entanto, embora abordem a composição de padrões, a literatura ainda carece de validações experimentais focadas no esgotamento de recursos em aplicações *upstream*. O presente trabalho preenche essa lacuna ao avaliar de forma prática a **abordagem híbrida** (*Retry* atuando em conjunto com o *Circuit Breaker*), analisando sua eficácia específica frente à saturação do *pool* de *threads* no servidor *Apache Tomcat*.

Outra frente de pesquisa concentra-se em criar variações customizadas dos padrões clássicos. Punithavathy e Priya (2024), por exemplo, propuseram o *Auto Retry Circuit Breaker* (ARCB), um algoritmo dinâmico e adaptativo que altera os limiares de tolerância do disjuntor em tempo real com base no comportamento da rede. Embora abordagens adaptativas ofereçam flexibilidade teórica, elas introduzem imprevisibilidade comportamental e dificultam a observabilidade em ambientes de produção. Em contrapartida a essa complexidade algorítmica, o presente trabalho defende uma solução estritamente arquitetural e previsível. A contribuição desta pesquisa atesta que não é necessário reescrever a ferramenta com limiares mutáveis: o uso de bibliotecas padronizadas de mercado (como a *Resilience4j*), parametrizadas de forma estática e determinística, é plenamente capaz de mitigar o colapso estrutural, desde que orquestradas em rigorosa sinergia híbrida.

Por fim, a escolha das ferramentas de monitoramento deste estudo é chancelada por pesquisas em ambientes nativos de nuvem. Gudi (2025) validou que a utilização do *k6* em conjunto com painéis *Grafana* e *Prometheus* representa o estado da arte para simulação de performance e observabilidade em microsserviços sob estresse. Essa validação consolida a legitimidade dos cenários de teste aplicados e da coleta de latência de cauda ($p95$) documentada nos resultados deste trabalho.

3 Desenvolvimento e Metodologia

Para transcender as modelagens teóricas discutidas no capítulo anterior e avaliar empiricamente a resiliência em sistemas distribuídos, este estudo adota uma abordagem experimental quantitativa baseada em testes de carga e nos princípios de *Fault Injection Testing* (FIT). Conforme estabelecido pela literatura (YU et al., 2025; HEORHIADI et al., 2016), a injeção sistemática e deliberada de anomalias na rede atua como o padrão-ouro para observar a ativação dos mecanismos de defesa arquiteturais e validar a robustez de microsserviços sob estresse. Alinhada a essa premissa, a metodologia aqui descrita estrutura um laboratório de testes isolado que induz a degradação estrutural do ambiente, permitindo analisar na prática o comportamento de retenção de *threads* e a eficácia dos padrões de resiliência.

No entanto, a execução determinística desses testes exige não apenas um ambiente rigorosamente isolado, mas também o controle exato sobre como a aplicação reage às falhas. Para materializar essa premissa, a metodologia deste trabalho foi operacionalizada em duas frentes complementares: primeiramente, a configuração minuciosa dos mecanismos internos de proteção (Camada de Resiliência) e, em seguida, a construção da infraestrutura de simulação (Bancada de Testes).

3.1 Camada de Resiliência: Mecanismos Internos da *Resilience4j*

Antes de detalhar a topologia da bancada de testes, é imperativo compreender os mecanismos estruturais da ferramenta escolhida para governar a tolerância a falhas da arquitetura distribuída. A biblioteca *Resilience4j* foi selecionada por ser uma solução modular e leve, desenhada especificamente para tirar proveito da programação funcional e de padrões decoradores no ecossistema Java moderno.

Diferente de soluções mais antigas, como o *Netflix Hystrix*, a *Resilience4j* não cria *pools* de *threads* separados para isolar falhas (*Thread Isolation*). Essa técnica legada, embora eficiente, consumia muita memória e exigia alto processamento apenas para realizar a troca de contexto entre as *threads*. Em vez disso, a *Resilience4j* atua de forma leve: utilizando Programação Orientada a Aspectos (AOP), ela apenas intercepta as chamadas de rede existentes, aplicando as regras de resiliência sem criar novas *threads* de controle (WINKLER; ALI; MUSHKEVYCH, 2024).

Para a metodologia deste trabalho, essa leveza arquitetural é fundamental. Ela garante que a latência medida no *benchmark* seja resultado exclusivo do estresse sobre o servidor *Apache Tomcat*, evitando que a própria biblioteca de proteção gere sobrecarga e

mascare os resultados da infraestrutura.

Para demonstrar como a biblioteca atinge essa eficiência operacional no controle de falhas, as subseções a seguir detalham os três pilares de sua engenharia interna: o gerenciamento otimizado do consumo de memória, as regras matemáticas que regem seu estado operacional e o modelo de intercepção não intrusiva de tráfego.

3.1.1 *Ring Bit Buffer* e Eficiência de Memória

Para manter o registro dos resultados dentro da janela deslizante sem comprometer a performance, a *Resilience4j* utiliza uma estrutura baseada em um *BitSet* circular, referida como *Ring Bit Buffer*. Diferente de implementações que armazenam objetos complexos para cada requisição, esta abordagem reduz a ocupação de memória ao mínimo indispensável, armazenando apenas o estado binário de cada chamada (0 para sucesso, 1 para falha ou lentidão) (WINKLER; ALI; MUSHKEVYCH, 2024).

Como ilustrado na Figura 5, a estrutura funciona como um *array* de tamanho fixo onde um ponteiro de índice avança a cada nova requisição. Ao atingir o limite definido para a janela deslizante (N), o ponteiro retorna à posição inicial, sobrescrevendo os dados mais antigos. Essa mecânica garante que o custo de memória seja estritamente constante, $O(1)$, e permite que o cálculo da taxa de erro seja realizado de forma otimizada através da contagem de bits ativos (WINKLER; ALI; MUSHKEVYCH, 2024). Além disso, a implementação utiliza atualizações atômicas para assegurar a consistência dos dados em ambientes de alta concorrência (*thread-safety*).

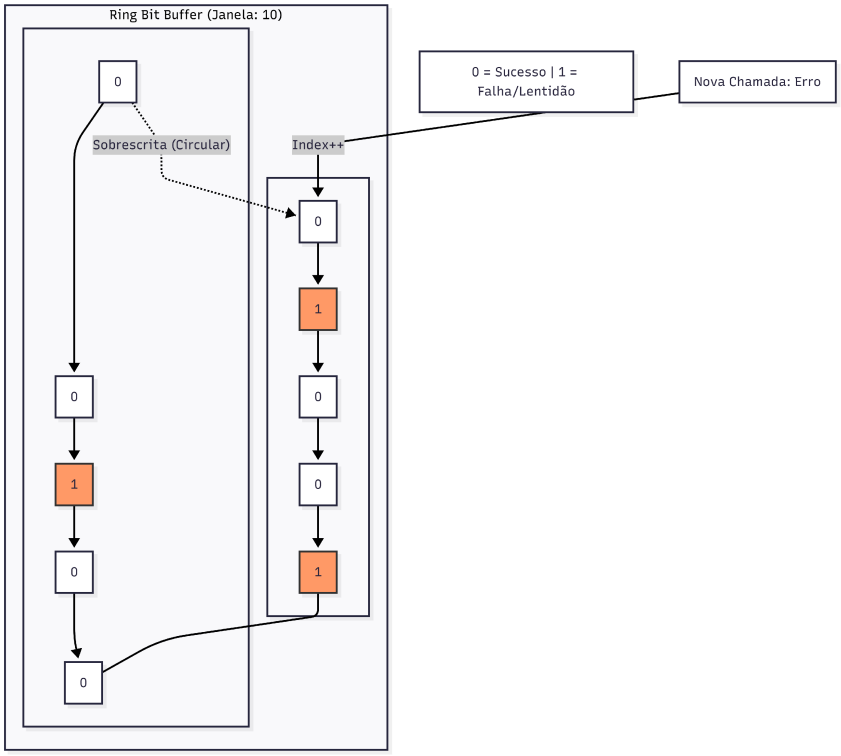

Figura 5 – Representação Esquemática do Ring Bit Buffer

re

Fonte: Elaborado pelo autor (2026).

3.1.2 Lógica de Transição da Máquina de Estados

O funcionamento central do *CB* é gerido por uma *Finite State Machine* (FSM) que consome os dados processados pelo *Ring Bit Buffer*. Enquanto o padrão teórico define o comportamento esperado dos estados (NEWMAN, 2021), a *Resilience4j* implementa a lógica de decisão de forma não bloqueante. As transições ocorrem por meio de atualizações atômicas em memória, garantindo que o monitoramento do circuito não insira contenção ou concorrência nas *threads* de requisição (WINKLER; ALI; MUSHKEVYCH, 2024).

A transição do estado *CLOSED* para *OPEN* ocorre de forma instantânea no momento em que a taxa de erro calculada ultrapassa o limiar definido. Uma característica crítica desta implementação é o comportamento no estado *HALF_OPEN*, que funciona como um ambiente de prova: a FSM permite apenas um número fixo de chamadas para testar a integridade da aplicação *downstream*. Se o limite de falhas for atingido novamente nesta fase, o circuito é reaberto imediatamente sem passar por um novo estado de retenção, garantindo proteção instantânea contra serviços que ainda apresentam instabilidade estrutural (WINKLER; ALI; MUSHKEVYCH, 2024).

3.1.3 Integração via Programação Orientada a Aspectos (AOP)

A Programação Orientada a Aspectos (do inglês, *Aspect-Oriented Programming* — AOP) é um paradigma de engenharia de software desenhado para complementar a Programação Orientada a Objetos (POO). Enquanto a POO é exímia em modelar o domínio da aplicação (regras de negócio), ela apresenta limitações estruturais ao lidar com requisitos sistêmicos que afetam múltiplos módulos simultaneamente, como auditoria (*logs*), gestão de transações e tolerância a falhas. Na literatura, esses requisitos globais são denominados interesses transversais (*cross-cutting concerns*) (KICZALES et al., 1997).

Quando implementados de forma tradicional, os interesses transversais obrigam o engenheiro a espalhar lógicas de infraestrutura — como blocos de repetição e captura de exceções de rede — por todo o código-fonte. Esse fenômeno gera dois anti-padrões arquiteturais clássicos: o espalhamento de código (*code scattering*) e o emaranhamento (*code tangling*), cenários onde a regra de negócio principal torna-se ilegível e fortemente acoplada a falhas externas. O paradigma AOP resolve esse problema ao extrair esses comportamentos repetitivos e encapsulá-los em entidades independentes, chamadas de "Aspectos"(KICZALES et al., 1997).

No contexto desta pesquisa, a integração da biblioteca *Resilience4j* ao ecossistema *Spring Boot* utiliza a AOP exatamente para resolver esse emaranhamento. O *framework* permite a separação estrutural de responsabilidades, encapsulando a complexa lógica de tolerância a falhas em *proxies* interceptadores. Dessa forma, garante-se que os componentes de negócio permaneçam limpos, isolados e estritamente agnósticos às complexidades e instabilidades da infraestrutura de rede (WINKLER; ALI; MUSHKEVYCH, 2024).

Através do uso de anotações declarativas como `@CircuitBreaker` e `@Retry`, o *Spring Boot* identifica quais métodos devem ser monitorados e, em tempo de execução, envolve esses métodos nesses *proxies* de interceptação (VMWARE, INC., 2024). Dessa forma, quando uma requisição é disparada, ela passa por uma pilha de aspectos que decidem se a chamada deve seguir para o serviço remoto ou se o fluxo deve ser interrompido precocemente.

Para a validade metodológica deste experimento, a ordem de precedência empilhada entre esses aspectos é determinante, conforme detalhado na Tabela 1.

Tabela 1 – Cadeia de Precedência dos Aspectos Utilizados

Ordem	**Padrão**	**Função da Intercepção**
1 (Externa)	Circuit Breaker	Intercepta o resultado final para gerir o estado do circuito.
2 (Interna)	Retry	Intercepta a falha e dispara novas tentativas.

Fonte: Elaborado pelo autor (2026).

Esta hierarquia é crucial para a integridade das métricas do disjuntor. Ao configurar o *CB* como o aspecto externo (maior precedência), ele delega a execução para o *Retry* (aspecto interno). Dessa forma, o *Retry* gerencia todas as retentativas transientes internamente. Apenas se todas as tentativas se esgotarem e a falha persistir, uma única exceção é propagada de volta ao *CB*.

Caso a ordem fosse invertida, o *Retry* atuaria externamente, reexecutando a chamada ao *CB* múltiplas vezes para uma mesma requisição. Isso faria com que o circuito contabilizasse cada tentativa individualmente em sua janela deslizante, provocando a abertura precoce do descritor frente a instabilidades que poderiam ter sido resolvidas por uma simples reexecução imediata (WINKLER; ALI; MUSHKEVYCH, 2024).

3.2 Infraestrutura de Simulação: Arquitetura da Bancada de Testes

A bancada de testes consistiu em um ecossistema de microsserviços desenvolvido em Java (Java Development Kit - JDK 17) utilizando o *framework Spring Boot 3*. Para simular um ambiente realista de alta concorrência, a arquitetura foi decomposta entre serviços de infraestrutura e serviços de domínio de negócio, conforme ilustrado na Figura 6.

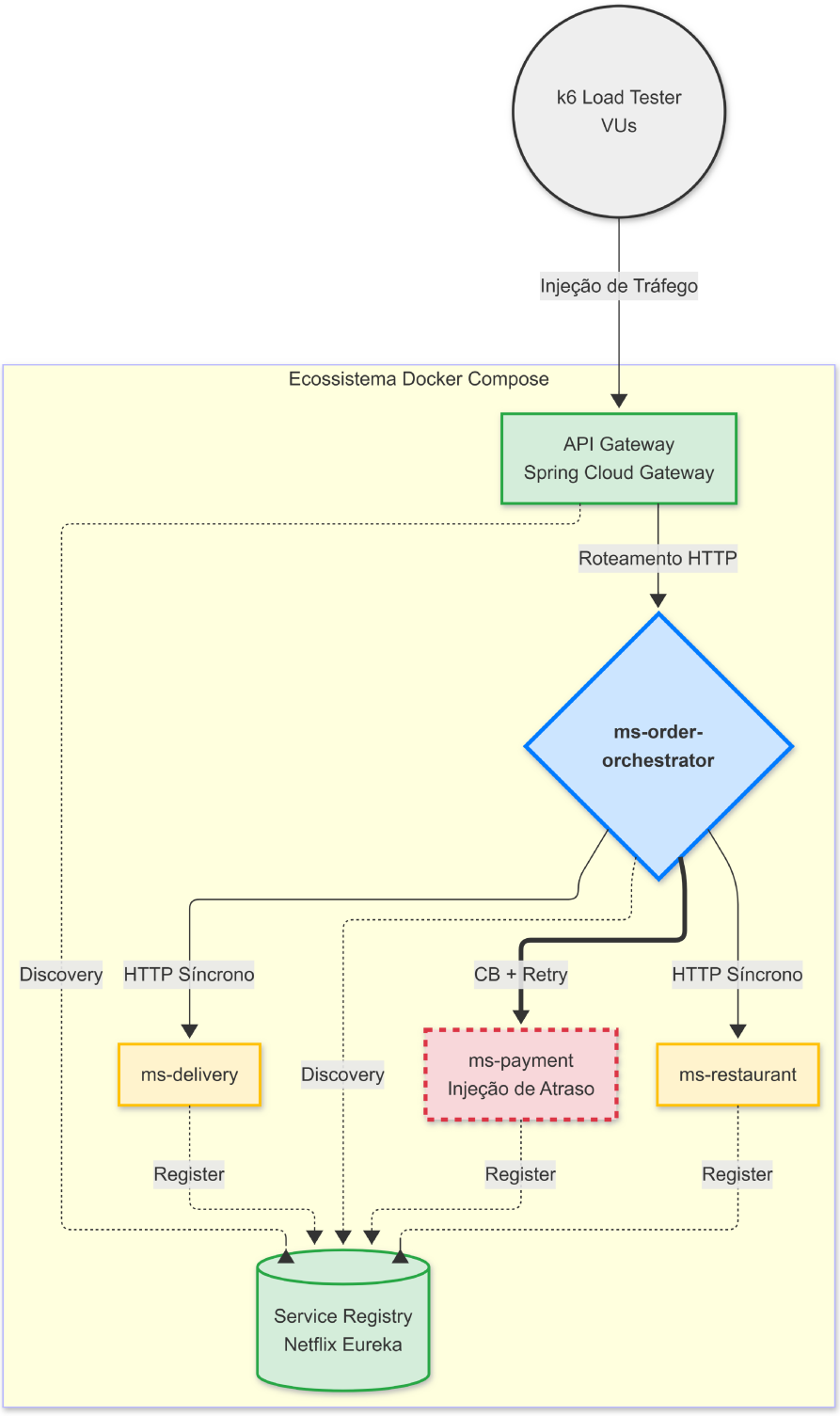

Figura 6 – Diagrama Arquitetural do Ecossistema de Testes.

Fonte: Elaborado pelo autor (2026).

3.2.1 Componentes de Infraestrutura e Domínio

A topologia de rede interna dependeu de dois contêineres essenciais para gerenciar o tráfego: o *API Gateway* (Spring Cloud Gateway), atuando como ponto único de entrada e balanceamento inicial, e o *Service Registry* (Netflix Eureka), permitindo a descoberta dinâmica dos serviços sem o acoplamento a endereços *Internet Protocol* (IP) estáticos.

O fluxo de negócio simulou um sistema de pedidos distribuído, fragmentado em quatro microsserviços independentes. É imperativo destacar que, para garantir o rigor e o isolamento das variáveis do experimento, os serviços *downstream* não executam processamento real de regras de negócio ou consultas a banco de dados. Eles operam estritamente como *stubs* representativos, retornando *payloads* estáticos com o único propósito de gerar tráfego de rede e validar o comportamento da arquitetura sob estresse:

- **ms-order-orchestrator:** O componente central do experimento. Responsável por receber a requisição e orquestrar as chamadas HTTP síncronas para os demais nós. É nele que o padrão híbrido de resiliência é aplicado e é ele o serviço vulnerável ao esgotamento de *threads*.
- **ms-payment:** Representa o domínio financeiro da transação. Em condições normais, apenas retorna um sucesso imediato. Neste desenho experimental, no entanto, foi o alvo deliberado da injeção de falhas, sendo configurado para sofrer atrasos artificiais (*Thread.sleep*) para simular a degradação externa de latência.
- **ms-restaurant** e **ms-delivery:** Microsserviços periféricos que simulam, respectivamente, os domínios de validação de itens e logística. Eles atuam como nós passivos de aprovação instantânea, sendo essenciais puramente para compor a topologia de rede e o tempo nominal de resposta de uma transação distribuída completa.

3.2.2 Containerização e Injeção de Resiliência

Para garantir o rigor científico e a validade do *benchmark*, todos os componentes foram conteinerizados e provisionados via *Docker* (MERKEL, 2014) e *Docker Compose* (Docker Inc., 2024). Esta abordagem garante o isolamento completo da rede interna e a total reprodutibilidade do experimento (ADERALDO et al., 2017), mitigando a interferência do sistema operacional hospedeiro no consumo de recursos e garantindo a paridade entre os ambientes de desenvolvimento e teste.

A biblioteca *Resilience4j* foi integrada de forma pontual no *ms-order-orchestrator*, atuando como um *proxy* estrutural que intercepta exclusivamente as chamadas HTTP direcionadas ao contêiner *ms-payment* (WINKLER; ALI; MUSHKEVYCH, 2024).

3.2.3 Parametrização

A eficácia da resiliência arquitetural depende diretamente do ajuste fino (*tuning*) dos parâmetros que definem o comportamento dos padrões implementados. A definição do tamanho da janela deslizante, do limiar de erro e do intervalo de espera do *Retry* foi fundamentada na necessidade de detectar a degradação em tempo hábil para evitar o esgotamento do *pool* de *threads* (MENDONCA et al., 2020). Para este experimento, a biblioteca *Resilience4j* foi configurada conforme detalhado na Listagem 3.1.

Listing 3.1 – Configuração do Resilience4j no application.properties

```
# ---Configuracao do Retry ---
resilience4j.retry.configs.default.maxAttempts=3
resilience4j.retry.configs.default.waitDuration=500ms

# ---Configuracao do Circuit Breaker ---
resilience4j.circuitbreaker.configs.default.failureRateThreshold=50
resilience4j.circuitbreaker.configs.default.slidingWindowSize=10
resilience4j.circuitbreaker.configs.default.waitDurationInOpenState=10s
resilience4j.circuitbreaker.configs.default.permittedNumberOfCallsInHalfOpenState=2
resilience4j.circuitbreaker.configs.default.slowCallRateThreshold=100
resilience4j.circuitbreaker.configs.default.slowCallDurationThreshold=2000
```

A definição destes valores seguiu premissas de rigor experimental e isolamento de variáveis. Para o padrão *Retry*, definiu-se um limite de 3 tentativas com um intervalo de espera fixo (*fixed delay*) de 500ms. Embora a literatura recomende o uso de *Exponential Backoff* para mitigar sobrecargas em ambientes produtivos (MOHAMMAD, 2025), tal estratégia foi descartada neste cenário laboratorial devido à hierarquia de *proxies* (Tabela 1).

Como o *CB* atua como o interceptador externo, o seu temporizador contabiliza o tempo total do ciclo, incluindo as pausas executadas pelo *Retry*. A adoção de um intervalo fixo garante um tempo de espera acumulado estritamente determinístico. Se flutuações exponenciais fossem introduzidas, os longos intervalos de retentativa poderiam, sozinhos, ultrapassar o limiar de lentidão do disjuntor (2000ms), gerando falsos positivos. Esse controle experimental impede que a própria política de retentativa mascare a latência real de 3 segundos injetada no serviço, assegurando que o circuito abra estritamente em função da degradação da rede *downstream*.

Para o *CB*, optou-se pela utilização de uma janela deslizante baseada em contagem (*count-based*), conforme detalhado na Seção 2.3.1.1. A janela foi configurada para avaliar as últimas 10 requisições (*slidingWindowSize=10*), com um limiar de falha de 50% para a transição ao estado Aberto (*Open*) (MOLCHANOV; ZHMAIEV, 2018). A escolha pelo modelo baseado em contagem em detrimento do temporal justifica-se pela busca por determinismo estatístico no *benchmark*, seguindo os requisitos de rigor metodológico para sistemas distribuídos (ADERALDO et al., 2017). Enquanto uma janela temporal

depende da vazão variável da rede para compor sua amostra, a janela por contagem assegura que a transição de estado ocorra invariavelmente após a décima requisição registrada. Essa abordagem permite isolar a eficácia do algoritmo de flutuações externas de latência — fenômeno intrínseco à comunicação sem fio e que corrobora as falácias de rede distribuída (TANENBAUM; STEEN, 2017) — garantindo que a abertura do circuito seja uma resposta reproduzível ao volume de falhas injetadas, e não a um intervalo de tempo arbitrário.

Por fim, para reger o comportamento dinâmico de auto-recuperação (*self-healing*), o temporizador de retenção no estado Aberto foi fixado em 10 segundos (`waitDuration InOpenState=10s`), seguido por um limite estrito de duas chamadas de sondagem no estado Meio-Aberto (`permittedNumberOfCallsInHalfOpenState=2`). A redução do tempo de espera para 10 segundos, em detrimento do padrão de 60 segundos da biblioteca, foi uma decisão deliberada para forçar ciclos de sondagem frequentes durante as janelas de estresse do k6. Essa agressividade na verificação permite mapear o exato momento de recuperação da rede. Simultaneamente, o limite de apenas duas chamadas de prova garante que, caso a instabilidade persista, o circuito retorne ao isolamento preventivo quase imediatamente, minimizando a exposição da infraestrutura a requisições destinadas à falha.

3.2.4 Especificações do Ambiente de Execução

Para garantir a integridade dos resultados e a estabilidade da bancada de testes, todos os experimentos foram executados em um ambiente controlado. As especificações de *hardware* e a *stack* de *software* utilizada estão detalhadas na Tabela 2.

Tabela 2 – Especificações Técnicas do Ambiente de Testes

Componente	**Especificação**
Hardware	
Processador	Intel Core i5-9300H (4 núcleos / 8 threads) @ 2.40 GHz
Memória RAM	16.0 GB DDR4
Armazenamento	SSD Kingston 477 GB
Placa de Vídeo	NVIDIA GeForce GTX 1650 (4 GB GDDR5)
Software	
Sistema Operacional	Ubuntu 22.04 LTS (64 bits)
Versão do Kernel	Linux 6.8.0-94-generic
Versão do Java	JDK 17 (LTS)
Spring Boot	3.5.7
Spring Cloud	2025.0.0
Resilience4j	v2.3.0 (via Spring Cloud BOM)
Docker Engine	v28.4.0
Docker Compose	v2.39.1
Grafana k6	v1.6.1
Rede	
Conectividade	Interface Wireless (IEEE 802.11)

Fonte: Elaborado pelo autor (2026).

3.3 Configuração do Experimento e Matriz de Testes

O protocolo experimental foi desenhado de forma bidimensional, avaliando diferentes configurações de resiliência arquitetural sob variados perfis de carga e degradação. Esta matriz de testes garante o isolamento das variáveis e permite uma comparação experimental rigorosa das falhas em cascata (ADERALDO et al., 2017).

Para garantir o rigor estatístico, cada teste foi precedido por um estágio de *setup* para estabilização do ambiente via *endpoints* de diagnóstico.

3.3.1 Configurações Arquiteturais Avaliadas

Para comprovar a eficácia isolada e combinada dos padrões, o ecossistema foi testado em quatro estados distintos de configuração, ativados e desativados no orquestrador:

1. **Sistema Desprotegido (Baseline de Falha):** Nenhuma política do *Resilience4j* ativa. Serve como grupo de controle para demonstrar o esgotamento natural da infraestrutura.

2. **Apenas Padrão *Retry*:** Ativação exclusiva das retentativas. Avalia-se a hipótese de que o uso isolado deste padrão é ineficaz contra a latência da rede, mas agrava a

saturação do servidor frente a erros explícitos, gerando uma tempestade de requisições (RASHEEDH; SARADHA, 2022).

3. **Apenas *Circuit Breaker*:** Ativação exclusiva do disjuntor. Analisa-se a proteção da infraestrutura (*fail-fast*) em detrimento da taxa de sucesso de recuperação inicial.

4. **Abordagem Híbrida (CB + Retry):** Ativação conjunta. O objetivo é validar a sinergia arquitetural recomendada pela literatura (PUNITHAVATHY; PRIYA, 2024).

3.3.2 Perfis de Carga e Estresse (Cenários k6)

A definição do roteiro de injeção de falhas fundamentou-se em metodologias consolidadas de testes de resiliência em sistemas distribuídos (YU et al., 2025). Para a injeção de tráfego concorrente e a aferição rigorosa das métricas de vazão e latência de cauda, utilizou-se o *Grafana k6*, uma ferramenta *open-source* de *benchmarking* orientada a engenharia de performance, configurada para emular *Virtual Users* (VUs) isolando o custo de *I/O* da máquina hospedeira (GRAFANA LABS, 2024). Os cenários de teste foram estruturados da seguinte forma:

1. **Cenário de Baseline (Carga Nominal):** Execução com carga leve (5 VUs) sob condições ideais de rede. **Justificativa:** Estabelecer o consumo basal de recursos da infraestrutura para fins de controle (ADERALDO et al., 2017).

2. **Cenário de Latência Controlada (Resiliência Básica):** Injeção de 3 segundos de atraso no serviço *downstream* sob carga constante de 50 VUs. **Justificativa:** Validação inicial da contenção do acoplamento temporal através de falhas transientes simuladas (HEORHIADI et al., 2016).

3. **Cenário de Ruptura e Estresse (Saturação de Threads):** Implementação de carga progressiva (*ramping-up*) atingindo um pico de 300 VUs simultâneos com latência de 3s. **Justificativa:** Este volume satura matematicamente a capacidade de recepção do *Apache Tomcat*, preenchendo simultaneamente o limite de 200 *threads* de processamento (*maxThreads*) e a fila de retenção padrão de 100 conexões no *backlog* TCP (*acceptCount*) (VMWARE, INC., 2024). O objetivo é forçar o esgotamento absoluto de recursos para documentar as falhas em cascata (DUEñAS-OSORIO; VEMURU, 2009).

4. **Cenário de Falha Rápida (Erro Imediato):** Execução com carga constante de 50 VUs durante um minuto e injeção exclusiva de erros HTTP 503, removendo o atraso da rede. **Justificativa:** Comprovar que o disjuntor reage puramente à taxa

de erro, operando em regime de *fail-fast* de forma agnóstica à anomalia (MOLCHANOV; ZHMAIEV, 2018), além de quantificar o custo operacional (*overhead*) introduzido pelos interceptadores na memória da aplicação.

5. **Cenário de Auto-Recuperação (*Self-Healing*):** Teste de longa duração (120 segundos) sob carga constante de 50 VUs, com remoção forçada da falha na metade da execução. **Justificativa:** Capturar o comportamento dinâmico do disjuntor através das transições entre os estados Aberto, Meio-Aberto e Fechado, atestando o restabelecimento autônomo da arquitetura (MOHAMMAD, 2025).

3.3.3 Matriz de Execução (Cruzamento de Variáveis)

Na engenharia de software experimental, a execução de testes de força bruta combinatória frequentemente gera ruído estatístico sem agregar valor analítico à hipótese investigada (YU et al., 2025). Para otimizar a extração de métricas, aplicou-se o princípio de isolamento sistemático de variáveis (ADERALDO et al., 2017), definindo-se um roteiro de execuções cirúrgicas focado estritamente nas interseções de maior criticidade arquitetural, conforme detalhado na Tabela 3.

Tabela 3 – Matriz de Execução de Testes: Cenários vs. Configuração Arquitetural

Perfil de Carga (k6)	**Desprotegido**	**Apenas Retry**	**Apenas CB**	**Híbrido (CB+Retry)**
1. Baseline Nominal	Executado	-	-	-
2. Latência Controlada	Executado	-	-	Executado
3. Ruptura e Saturação	Executado	Executado	Executado	Executado
4. Erro HTTP 503	Executado	-	-	Executado
5. Auto-Recuperação	-	-	-	Executado

Fonte: Elaborado pelo autor (2026).

A ausência de certas combinações na matriz de testes obedece à própria mecânica dos componentes avaliados. Por exemplo, submeter o sistema desprotegido ou configurado apenas com *Retry* ao cenário de "Auto-Recuperação"é inviável, pois essas topologias não possuem o mecanismo de disjuntor para transitar entre os estados dinâmicos (*Open*, *Half-Open* e *Closed*) frente à resolução da falha (MOLCHANOV; ZHMAIEV, 2018). O *Baseline*, por sua vez, serve estritamente como grupo de controle, registrando o consumo padrão de recursos do servidor sem a aplicação de falhas ou políticas de resiliência.

Dessa forma, o foco analítico principal reside no Cenário 3 (Ruptura e Saturação), o único submetido às quatro configurações arquiteturais. Esta decisão baseia-se na diretriz de que o estresse extremo é a condição primária para validar experimentalmente o limite elástico da arquitetura (DUEñAS-OSORIO; VEMURU, 2009), contrastando o colapso do *Apache Tomcat* causado pelo uso isolado de padrões com a estabilidade fornecida pela sinergia híbrida (PUNITHAVATHY; PRIYA, 2024).

4 Resultados e Discussão

Os dados extraídos da bateria de testes validaram experimentalmente as premissas discutidas nos capítulos anteriores quanto ao colapso da infraestrutura sob acoplamento temporal. Para garantir o rigor estatístico e o isolamento das variáveis, a análise a seguir foi segmentada conforme os perfis de carga e as configurações de resiliência estipulados na Matriz de Execução (Tabela 3).

Através do acompanhamento quantitativo da vazão (*throughput*) e da latência de cauda ($p95$), é possível observar com clareza a degradação estrutural do servidor *Apache Tomcat* frente à injeção deliberada de falhas (DUEñAS-OSORIO; VEMURU, 2009). Em contrapartida, os mesmos dados demonstram como o esgotamento desses recursos é efetivamente mitigado pela aplicação estratégica da abordagem híbrida (*Circuit Breaker* e *Retry*), quantificando o ganho real de disponibilidade do ecossistema (ADERALDO et al., 2017).

4.1 Cenário de Baseline (Carga Nominal)

A execução do cenário de *Baseline* teve como objetivo estabelecer o grupo de controle do experimento, definindo o consumo basal de recursos da infraestrutura em condições ideais. Conforme estabelecido por Aderaldo et al. (2017), a avaliação de degradação em sistemas distribuídos exige o conhecimento prévio do custo computacional do ecossistema em seu estado saudável. Para este fim, a arquitetura foi submetida a uma carga nominal de 5 Usuários Virtuais (VUs), operando sem injeção de falhas.

Os resultados consolidados na Tabela 4 demonstram a estabilidade operacional da bancada de testes. O sistema manteve uma taxa de sucesso de 100% com uma vazão média (*throughput*) de 4,73 requisições por segundo. A latência de cauda ($p95$) estabilizou-se em 174,96 ms, um valor que reflete o tempo total de ida e volta (*Round Trip Time*) sob o regime de descoberta dinâmica de serviços e roteamento via *Gateway*.

Este patamar de latência nominal serve como a métrica de referência absoluta para este estudo. O registro do comportamento estável permite quantificar, nos cenários subsequentes, o impacto real do aprisionamento de *threads* no servidor de aplicação e a eficiência da estratégia *fail-fast* em preservar a responsividade do orquestrador frente à degradação de serviços *downstream*.

Tabela 4 – Resultados do Cenário 1: Baseline Nominal (5 VUs)

Métrica k6	**Valor Obtido**
Requisições Totais Realizadas	50
Vazão Média (*Throughput*)	4,73 req/s
Latência Mediana ($p50$)	27,29 ms
Latência de Cauda ($p95$)	174,96 ms
Taxa de Sucesso (HTTP 200)	100,00%

Fonte: Elaborado pelo autor (2026).

4.2 Cenário de Latência Controlada (Resiliência Básica)

Este cenário avaliou o comportamento do ecossistema sob uma carga constante de 50 VUs, com a injeção deliberada de um atraso de 3.000 ms no microsserviço de pagamentos (*ms-payment*). O objetivo foi contrastar a arquitetura desprotegida com a configuração híbrida (*CB + Retry*), observando como a retenção de recursos impacta a vazão do orquestrador (YU et al., 2025).

Os resultados consolidados na Tabela 5 revelam um contraste crítico de performance. No estado desprotegido, a latência de cauda ($p95$) atingiu 3,43 s, resultando em uma vazão limitada de apenas 12,17 req/s. Em contrapartida, a ativação da resiliência híbrida permitiu que o sistema detectasse a lentidão e aplicasse o bloqueio preventivo, reduzindo o $p95$ para 66,76 ms e elevando o *throughput* para 43,23 req/s.

Tabela 5 – Comparativo: Sistema Desprotegido vs. Híbrido (Latência de 3s)

Métrica k6	**Desprotegido**	**Híbrido (CB+Retry)**
Total de Requisições	400	1.338
Vazão (*Throughput*)	12,17 req/s	43,23 req/s
Latência Mediana ($p50$)	3,01 s	6,73 ms
Latência de Cauda ($p95$)	3,43 s	66,76 ms
Taxa de Sucesso (HTTP 200)	100,00%	4,03%

Fonte: Elaborado pelo autor (2026).

A análise técnica destes dados evidencia o fenômeno do acoplamento temporal. No cenário desprotegido, embora a taxa de sucesso seja de 100%, o sistema opera no limite da exaustão, com as requisições retidas em estado de *I/O Wait* por períodos superiores a 3 segundos. Tal comportamento corrobora as falácias da computação distribuída quanto à latência não nula (TANENBAUM; STEEN, 2017).

Já na arquitetura híbrida, a redução drástica na latência mediana para 6,73 ms comprova a eficácia da interrupção mecânica do fluxo (*fail-fast*) (MENDONCA et al., 2020). Embora a taxa de sucesso caia para 4,03%, a preservação da vazão média (quase quatro vezes superior ao modelo desprotegido) atesta que o orquestrador permanece responsivo, processando 1.338 chamadas no mesmo intervalo em que o sistema desprotegido processou apenas 400. Esta economia de recursos de *threads* é essencial para evitar o colapso sistêmico em cadeias de chamadas síncronas (DUEñAS-OSORIO; VEMURU, 2009).

4.3 Cenário de Ruptura e Estresse (Saturação de Threads)

Este cenário representa o teste de estresse máximo da arquitetura, submetendo o orquestrador a uma carga progressiva de até 300 VUs simultâneos com uma latência injetada de 3.000 ms. O objetivo primordial foi forçar o colapso estrutural do *Apache Tomcat*. Como o servidor é configurado por padrão com 200 *threads* síncronas para processamento (*maxThreads*) e uma fila de espera de 100 conexões (*acceptCount*), a injeção exata de 300 usuários garante o esgotamento total do *pool* de *threads* e da capacidade de enfileiramento do sistema operacional, permitindo observar nitidamente a propagação da falha em cascata (VMWARE, INC., 2024; DUEñAS-OSORIO; VEMURU, 2009).

Os resultados consolidados na Tabela 6 permitem uma análise comparativa profunda entre o sistema desprotegido, o uso isolado de padrões e a sinergia híbrida.

Tabela 6 – Comparativo de Stress e Saturação (300 VUs / 3s Latência)

Métrica k6	**Desprotegido**	**Apenas Retry**	**Apenas CB**	**Híbrido**
Total Requisições	4.765	4.769	205.176	219.075
Vazão (*Throughput*)	38,80 req/s	38,82 req/s	1.707,07 req/s	1.820,10 req/s
Latência $p50$ (Mediana)	3,02 s	3,02 s	49,69 ms	42,52 ms
Latência $p95$ (Cauda)	4,42 s	4,43 s	224,15 ms	241,84 ms
Taxa de Sucesso	100,00%	100,00%	0,01%	0,01%

Fonte: Elaborado pelo autor (2026).

A análise técnica revela que tanto o sistema **Desprotegido** quanto a configuração **Apenas Retry** sofreram um colapso idêntico. Com latências de cauda atingindo 4,42 s e 4,43 s, respectivamente, o *throughput* foi estrangulado para cerca de 38 req/s. Como as chamadas degradadas não retornaram erros explícitos, mas sim respostas tardias (taxa de sucesso de 100%), o mecanismo de retentativa não foi sequer acionado. Este comportamento comprova a hipótese de que o padrão *Retry*, quando aplicado isoladamente frente a gargalos de latência, é arquiteturalmente inócuo. Ele é incapaz de aplicar limites temporais de *fail-fast*, mantendo as *threads* do *Apache Tomcat* sequestradas em estado de *I/O Wait*

e confirmando a sua ineficácia na proteção preventiva da infraestrutura (MENDONCA et al., 2020).

Em contrapartida, as configurações que utilizam o **CB** (isolado ou híbrido) demonstraram a eficácia da estratégia *fail-fast*. Ao detectar que as chamadas excediam o limiar de lentidão, o componente abriu o circuito, interrompendo o fluxo para o serviço degradado e libertando instantaneamente as *threads* de execução (MOLCHANOV; ZHMAIEV, 2018). Isso permitiu que a vazão saltasse para patamares superiores a 1.700 req/s, com a latência $p95$ contida abaixo dos 250 ms.

Embora a taxa de sucesso tenha caído para 0,01%, o sistema preservou a sua capacidade operacional global, rejeitando pedidos de forma controlada em vez de sucumbir ao travamento completo do processo. Esta evidência experimental valida a necessidade de isolamento estrutural para garantir a resiliência de orquestradores sob stress extremo (YU et al., 2025; ADERALDO et al., 2017).

4.4 Cenário de Falha Rápida (Erro Imediato)

Diferente dos cenários anteriores onde a degradação era causada por latência (retenção temporal), este cenário simulou uma queda imediata e persistente do serviço de pagamentos, com a injeção exclusiva de erros `HTTP 503 (Service Unavailable)`. O objetivo foi avaliar o comportamento do padrão *Retry* na tentativa de mitigação da falha e a intervenção do *CB* ao identificar que o erro não era transiente (PUNITHAVATHY; PRIYA, 2024). O teste foi executado sob uma carga de 50 VUs durante um minuto.

Os dados apresentados na Tabela 7 consolidam a comparação de desempenho entre a topologia desprotegida e a arquitetura híbrida sob falha estrutural imediata.

Tabela 7 – Comparativo: Sistema Desprotegido vs. Híbrido (Erro HTTP 503)

Métrica k6	**Desprotegido**	**Híbrido (CB+Retry)**
Total de Requisições	98.669	92.864
Vazão (*Throughput*)	1.638,85 req/s	1.540,01 req/s
Latência Mediana ($p50$)	23,99 ms	21,85 ms
Latência de Cauda ($p95$)	63,76 ms	83,62 ms
Taxa de Sucesso	0,00%	0,00%

Fonte: Elaborado pelo autor (2026).

A análise técnica destes resultados revela uma dinâmica fundamental sobre o custo-benefício de interceptadores de rede. No modelo desprotegido, a vazão atinge o pico de 1.638 req/s e o $p95$ permanece extremamente baixo (63,76 ms) simplesmente porque o

serviço *downstream* está rejeitando as conexões instantaneamente de forma nativa. Não há retenção de *threads*, mas a propagação do erro em cascata para o cliente é garantida e incontrolável.

Na configuração Híbrida, observa-se que a taxa de sucesso manteve-se em 0,00%. Este resultado é esperado: o *Retry* é desenhado para contornar falhas transientes (PUNITHAVATHY; PRIYA, 2024); se a falha *downstream* é definitiva e persiste ao longo de todo o teste, retentativas matemáticas não alterarão o cenário. No entanto, a análise do comportamento dinâmico é evidenciada pela métrica de latência de cauda ($p95$) de 83,62 ms e pela ligeira queda na vazão geral (1.540 req/s).

O acréscimo de aproximadamente 20 ms no $p95$ em relação ao modelo desprotegido quantifica experimentalmente o custo arquitetural da máquina de estados do *Resilience4j*. Este delta de milissegundos reflete o processamento não bloqueante da biblioteca ao classificar os erros (FALAHAH; SURENDRO; SUNINDYO, 2021). Mais importante, a queda na vazão comprova a eficácia da hierarquia de aspectos definida na metodologia: ao atingir o limiar de falhas estipulado, o *Circuit Breaker* (aspecto externo) transita para o estado Aberto e passa a rejeitar a carga prematuramente na própria memória do orquestrador. Esta interrupção mecânica evita o acionamento repetitivo do *Retry* (aspecto interno), poupando a rede interna de ser bombardeada por uma tempestade de retentativas (*retry storm*). Este pequeno *overhead* operacional de 20 ms demonstra ser um custo computacional irrisório em troca do isolamento instantâneo da degradação.

4.5 Cenário de Auto-Recuperação (*Self-Healing*)

O último cenário da bateria de testes avaliou a capacidade de auto-recuperação da arquitetura distribuída. Para a validação deste cenário, o teste foi parametrizado com uma carga constante de 50 VUs. Este teste de longa duração (120 segundos) consistiu na injeção de latência severa durante a primeira metade da execução, seguida pela interrupção intencional da anomalia no ponto médio do experimento (segundo 60). O objetivo foi observar a transição dinâmica entre os estados *Open*, *Half-Open* e *Closed* da máquina de estados do *Resilience4j* (MOLCHANOV; ZHMAIEV, 2018).

Os resultados consolidados na Tabela 8 demonstram a resiliência operacional do ecossistema frente à recuperação do serviço *downstream*.

A análise técnica da taxa de sucesso de 52,93% fornece a prova experimental definitiva da eficácia do componente. Em um teste de dois minutos, no qual a falha externa foi removida no ponto médio da execução, um sistema sem capacidade de auto-recuperação (*self-healing*) manteria as conexões retidas ou exigiria uma reinicialização manual. O valor obtido comprova que o *CB* operou em modo *fail-fast* de forma responsiva durante o período de degradação. Graças à configuração da propriedade `waitDurationInOpenState=10s`,

Tabela 8 – Resultados do Cenário 5: Auto-Recuperação (120s)

Métrica k6	**Valor Obtido**
Requisições Totais Realizadas	220.018
Vazão Média (*Throughput*)	1.830,09 req/s
Latência Mediana ($p50$)	20,37 ms
Latência de Cauda ($p95$)	61,03 ms
Taxa de Sucesso (HTTP 200)	52,93%

Fonte: Elaborado pelo autor (2026).

o interceptador não aguardou passivamente a resolução da anomalia, mas realizou sondagens periódicas a cada 10 segundos transitando para o estado *Half-Open*. Durante o primeiro minuto, essas sondagens falharam, forçando o retorno imediato ao estado Aberto. Contudo, na primeira sondagem após o segundo 60, a janela baseada em contagem (*count-based*) registrou sucesso no número fixo de chamadas permitidas, provando a detecção autônoma da recuperação da rede e o consequente fechamento do circuito, restabelecendo o fluxo normal da aplicação.

Neste estado de sondagem, a passagem de requisições de teste permitiu ao orquestrador validar a integridade do microsserviço de pagamentos e restabelecer o tráfego integral, retornando ao estado Fechado (*Closed*) (MOHAMMAD, 2025). A convergência da latência $p95$ para 61,03 ms ao final da execução confirma que a estratégia híbrida neutraliza a propagação de anomalias e garante a continuidade do processamento sem degradação residual, validando a hipótese de resiliência autonômica proposta por esta pesquisa (ADERALDO et al., 2017).

4.6 Síntese Comparativa e Discussão Final

A consolidação dos dados experimentais permite uma análise transversal sobre o equilíbrio entre estabilidade e disponibilidade sob estresse extremo. Enquanto as seções anteriores isolaram comportamentos específicos frente a diferentes anomalias de rede, esta síntese correlaciona os achados para atestar a robustez da abordagem híbrida. O foco recai sobre a validação experimental de como a orquestração conjunta dos interceptadores previne o esgotamento de recursos do servidor e atua como barreira definitiva contra a propagação de falhas em cascata (YU et al., 2025).

O principal indicador de sucesso da estratégia híbrida não reside apenas na taxa de sucesso pontual das requisições, mas na preservação da vazão do orquestrador. Como observado no Cenário 3 (Tabela 6), o sistema híbrido sustentou uma vazão 4.700% superior ao modelo desprotegido sob estresse máximo. Essa disparidade quantifica o custo da

omissão de mecanismos de defesa: a perda quase total da capacidade de processamento em favor de um serviço dependente em colapso, o que caracteriza a propagação de falhas em cascata descrita na literatura (DUEñAS-OSORIO; VEMURU, 2009). O bloqueio síncrono resultante da latência externa atua como o gatilho principal para o esgotamento do *pool* de *threads*, tornando o orquestrador incapaz de processar fluxos de negócio independentes que compartilham a mesma infraestrutura (ZHOU et al., 2021; MOLCHANOV; ZHMAIEV, 2018).

Adicionalmente, o experimento permitiu mapear o *trade-off* de latência inerente às arquiteturas de microsserviços. O custo de intercepção de aproximadamente 20 ms identificado no Cenário 4 (Tabela 7) representa o impacto arquitetural da máquina de estados e do *proxy* de aspecto (AOP) do *Resilience4j*. Em sistemas de missão crítica, esse atraso basal é um investimento necessário para garantir a previsibilidade operacional. A garantia de que o sistema operará em modo *fail-fast* impede que picos de tráfego se transformem em colapsos sistêmicos, respeitando as premissas de que a latência de rede é uma variável não nula e persistente (DEUTSCH et al., 1994; TANENBAUM; STEEN, 2017).

Conclui-se, a partir das evidências experimentais, que a aplicação isolada do padrão *Retry* é insuficiente e apresenta risco severo à integridade da infraestrutura caso não seja governada por um *Circuit Breaker* (MENDONCA et al., 2020; MOHAMMAD, 2025). A sinergia híbrida validada nesta pesquisa comprova ser o modelo mais robusto para arquiteturas *Spring Boot* modernas, pois alia a mitigação de falhas transientes à proteção rigorosa dos recursos subjacentes (PUNITHAVATHY; PRIYA, 2024). Essa abordagem garante que o ecossistema tolere instabilidades parciais e mantenha sua capacidade de auto-recuperação (*self-healing*) sem comprometer a estabilidade do orquestrador, confirmando a hipótese central deste trabalho. Os principais achados e o veredito técnico sobre cada configuração avaliada estão sintetizados na Tabela 9.

Tabela 9 – Matriz de Veredito Técnico por Configuração Arquitetural

Configuração	**Principal Vantagem**	**Principal Risco**
Desprotegido	Latência nominal mínima	Colapso por *thread exhaustion*
Apenas Retry	Recuperação transiente	Amplificação de carga (*retry storm*)
Apenas CB	Proteção (*fail-fast*)	Rejeição de transações viáveis
Híbrido	**Estabilidade e Disponibilidade**	***Overhead* residual de latência**

Fonte: Elaborado pelo autor (2026).

Com a validação dos cenários e a consolidação dos resultados comparativos, encerra-

se a fase de análise experimental deste estudo. Os dados coletados não apenas comprovam a eficácia dos padrões de resiliência, mas também fornecem diretrizes práticas para a parametrização de sistemas distribuídos sob condições de estresse. No capítulo seguinte, serão apresentadas as conclusões finais do trabalho, as limitações encontradas durante a pesquisa e as sugestões para investigações futuras que possam expandir o escopo da tolerância a falhas em microsserviços.

5 Conclusão

O presente trabalho propôs-se a investigar a mitigação de falhas em cascata em arquiteturas de microsserviços por meio da aplicação de padrões de resiliência. A premissa central deste estudo baseou-se na hipótese de que o uso combinado de *Circuit Breaker* e *Retry* seria capaz de restringir o raio de impacto de serviços degradados, prevenindo o esgotamento prematuro de recursos de serviços *upstream* e garantindo a continuidade operacional.

Através de uma metodologia baseada em injeção de falhas (*Fault Injection Testing*) e testes de estresse em uma bancada isolada (*Spring Boot 3* e *Resilience4j*), foi possível observar experimentalmente os limites da comunicação síncrona. Os testes de ruptura confirmaram o colapso do servidor *Apache Tomcat* quando desprotegido, evidenciando o aprisionamento veloz do *pool* de *threads* na presença de latência induzida.

A análise dos cenários experimentais validou integralmente a hipótese de pesquisa. Comprovou-se empiricamente que o padrão *Retry*, quando operando de forma isolada, demonstra-se incapaz de mitigar gargalos temporais (latência) e atua como um perigoso amplificador de carga frente a falhas explícitas, acelerando a saturação da infraestrutura. Em contrapartida, a implementação da arquitetura híbrida demonstrou alta eficácia. A estratégia de *fail-fast* provida pelo *CB* não apenas protegeu o *pool* de *threads* do orquestrador contra o *I/O Wait*, como garantiu a sustentação da vazão do sistema, rejeitando requisições com latências estritamente controladas mesmo durante falhas estruturais severas.

O ápice do experimento ocorreu com a validação da auto-recuperação (*self-healing*). A transição fluida da máquina de estados para *Half-Open* atestou que a arquitetura é capaz de se curar autonomamente através de chamadas de sondagem, restabelecendo a conectividade sem exigir intervenção humana ou *restarts* operacionais. Desta forma, conclui-se que o encapsulamento preventivo de chamadas de rede não é apenas uma boa prática arquitetural, mas um requisito estrito para a sobrevivência de ecossistemas distribuídos de alta disponibilidade.

5.1 Limitações do Estudo

Apesar do rigor estatístico do *benchmarking* realizado, algumas limitações arquiteturais e metodológicas devem ser reconhecidas. Primeiramente, as métricas de esgotamento focaram-se exclusivamente no *pool* de conexões e na retenção de *threads* (*Central Processing Unit* (CPU) e *I/O Wait*). Limites inerentes ao consumo de memória *Random*

Access Memory (RAM), *Garbage Collection* e saturação de conexões em banco de dados sob alta concorrência não foram incluídos no escopo da observação.

Em segundo lugar, a simulação de falhas restringiu-se à injeção de latência via temporizadores na camada de aplicação e retorno forçado de erros HTTP. Anomalias provenientes das camadas inferiores do modelo *Open Systems Interconnection* (OSI), como perdas reais de pacotes de rede (*packet loss*), falhas de resolução de *Domain Name System* (DNS) ou "instabilidades físicas no hospedeiro dos contêineres (*host OS*) não foram contempladas.

Adicionalmente, a arquitetura avaliada baseou-se estritamente no modelo de processamento síncrono e bloqueante (*Blocking I/O*) fornecido pelo *Apache Tomcat* tradicional. Paradigmas modernos de escalabilidade que mitigam o *thread exhaustion* por design, como a programação reativa (Spring WebFlux) ou a utilização de *Virtual Threads* introduzidas no Java 21 (*Project Loom*), não foram contemplados no escopo desta pesquisa. Consequentemente, as métricas de colapso de infraestrutura documentadas aplicam-se ao modelo *Thread-per-Request*, exigindo novas aferições para ecossistemas assíncronos.

Por fim, o mecanismo de *Retry* foi configurado com um intervalo de espera fixo (*fixed delay*) para manter o determinismo estatístico do experimento, abdicando do padrão de recuo exponencial (*Exponential Backoff*), que é o mais recomendado em topologias de produção elásticas.

5.2 Trabalhos Futuros

Considerando as limitações mapeadas e os resultados obtidos, propõem-se as seguintes frentes para a continuidade desta pesquisa:

- **Exploração do Padrão *Bulkhead* (Compartimentalização):** Como um desdobramento natural dos mecanismos oferecidos nativamente pela biblioteca *Resilience4j*, propõe-se aprofundar a mitigação do *thread exhaustion* através do isolamento físico de recursos. Enquanto o *CB* protege o ecossistema de forma reativa, o *Bulkhead* atua proativamente fatiando o *pool* de *threads* do servidor de aplicação. O trabalho futuro consistiria em parametrizar cotas máximas de *threads* por dependência (ex: limitando o serviço de pagamentos a um teto de 30% da capacidade total do Tomcat). O objetivo é avaliar experimentalmente se essa contenção impede o colapso sistêmico em cenários onde a degradação é pulverizada entre múltiplas integrações simultâneas, validando a sinergia arquitetural entre compartimentalização e *fail-fast*.
- **Avaliação em Ecossistemas Não-Bloqueantes e *Virtual Threads*:** Replicar a matriz de testes substituindo o servidor web tradicional pelo Spring WebFlux (arquitetura reativa baseada em eventos) ou habilitando as *Virtual Threads* do Java

21. O objetivo seria investigar se a eliminação do esgotamento físico de *threads* do sistema operacional reduz a criticidade do *CB* como protetor de infraestrutura, rebaixando-o a um mecanismo puramente lógico de *fail-fast*.

- **Investigação de Novos Paradigmas de Comunicação (gRPC e Mensageria):** Investigar a adaptação dos padrões de resiliência em protocolos binários como o gRPC, avaliando se a alta performance e a multiplexação alteram os limiares de intercepção. Adicionalmente, propõe-se o estudo da transição para arquiteturas orientadas a eventos (*Kafka* ou *RabbitMQ*), onde o desafio da resiliência deixa de ser o bloqueio de *threads* no orquestrador e passa a ser a gestão do acúmulo de mensagens e a integridade das filas durante períodos de indisponibilidade do consumidor.
- **Migração para Arquiteturas de *Service Mesh*:** Investigar o desacoplamento da lógica de resiliência do código-fonte da aplicação (baseada em *Software Development Kit* - SDK) para a infraestrutura de rede (*Proxy-based*), utilizando tecnologias como *Istio* ou *Linkerd*. O objetivo consiste em quantificar o *overhead* de CPU e memória, além da latência adicional introduzida pelo padrão *sidecar*, comparando a eficiência dessa abordagem agnóstica a linguagem com a implementação nativa via *Resilience4j* utilizada neste trabalho.

5.3 Declaração de Uso de Tecnologias de Inteligência Artificial

Durante a preparação deste trabalho, o autor utilizou tecnologias de Inteligência Artificial generativa (como *Google Gemini* e equivalentes) com o objetivo exclusivo de aprimorar a clareza da linguagem, realizar revisões gramaticais e refinar a coesão estrutural do texto acadêmico. Após a utilização destas ferramentas de assistência à escrita, o autor revisou, validou e editou minuciosamente o material, assumindo total e irrestrita responsabilidade pelo rigor técnico, pela integridade metodológica e pelo conteúdo final do documento publicado.

Referências

ADERALDO, C. M.; MENDONçA, N. C.; PAHL, C.; JAMSHIDI, P. Benchmark requirements for microservices architecture research. In: **2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)**. [S.l.: s.n.], 2017. p. 8–13. Citado 9 vezes nas páginas 26, 35, 36, 38, 39, 40, 41, 44 e 46.

DEUTSCH, L. P. et al. **The Eight Fallacies of Distributed Computing**. [S.l.]: Sun Microsystems, 1994. Disponível na literatura técnica da Sun Microsystems. Citado 2 vezes nas páginas 18 e 47.

Docker Inc. **Docker Compose documentation**. [S.l.], 2024. Acessado em: 21 abr. 2026. Disponível em: <https://docs.docker.com/compose/>. Citado na página 35.

DUEñAS-OSORIO, L.; VEMURU, S. M. Cascading failures in complex infrastructure systems. **Structural Safety**, v. 31, n. 2, p. 157–167, 2009. ISSN 0167-4730. Risk Acceptance and Risk Communication. Disponível em: <https://www.sciencedirect.com/science/article/pii/S016747300800057X>. Citado 7 vezes nas páginas 14, 19, 39, 40, 41, 43 e 47.

FALAHAH; SURENDRO, K.; SUNINDYO, W. D. Circuit breaker in microservices: State of the art and future prospects. **IOP Conference Series: Materials Science and Engineering**, IOP Publishing, v. 1077, n. 1, p. 012065, feb 2021. Disponível em: <https://doi.org/10.1088/1757-899X/1077/1/012065>. Citado 7 vezes nas páginas 14, 15, 19, 20, 22, 26 e 45.

GRAFANA LABS. **k6 Documentation: Open-source load testing tool and SaaS for engineering teams**. [S.l.], 2024. Acesso em: 17 abr. 2026. Disponível em: <https://k6.io/docs/>. Citado na página 39.

GUDI, S. R. Monitoring and deployment optimization in cloud-native systems: A comparative study using openshift and helm. In: **2025 4th International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)**. [S.l.: s.n.], 2025. p. 792–797. Citado na página 28.

HEORHIADI, V.; RAJAGOPALAN, S.; JAMJOOM, H.; REITER, M. K.; SEKAR, V. Gremlin: Systematic resilience testing of microservices. In: **2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)**. [S.l.: s.n.], 2016. p. 57–66. Citado 3 vezes nas páginas 16, 29 e 39.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M.; IRWIN, J. Aspect-oriented programming. In: SPRINGER. **European conference on object-oriented programming**. [S.l.], 1997. p. 220–242. Citado na página 32.

MENDONCA, N. C.; ADERALDO, C. M.; CAMARA, J.; GARLAN, D. Model-based analysis of microservice resiliency patterns. In: **2020 IEEE International Conference on Software Architecture (ICSA)**. [S.l.: s.n.], 2020. p. 114–124. Citado 6 vezes nas páginas 16, 27, 36, 43, 44 e 47.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. **Linux journal**, v. 2014, n. 239, p. 2, 2014. Citado na página 35.

MOHAMMAD, M. Resilient microservices: A systematic review of recovery patterns, strategies, and evaluation frameworks. **arXiv preprint arXiv:2512.16959**, 2025. Citado 6 vezes nas páginas 19, 22, 36, 40, 46 e 47.

MOLCHANOV, H.; ZHMAIEV, A. Circuit breaker in systems based on microservices architecture. **Advanced Information Systems**, v. 2, n. 4, p. 74–77, 2018. Citado 8 vezes nas páginas 15, 19, 20, 36, 40, 44, 45 e 47.

NEWMAN, S. **Building Microservices**. [S.l.]: O'Reilly Media, Inc., 2021. Referência fundamental para a teoria de Circuit Breaker. Citado 2 vezes nas páginas 14 e 31.

PAHL, C.; JAMSHIDI, P. Microservices: A systematic mapping study. **CLOSER (1)**, p. 137–146, 2016. Citado 2 vezes nas páginas 14 e 18.

PAPAPANAGIOTOU, I.; CHELLA, V. Ndbench: Benchmarking microservices at scale. **arXiv preprint arXiv:1807.10792**, 2018. Citado 2 vezes nas páginas 25 e 26.

PUNITHAVATHY, E.; PRIYA, N. Auto retry circuit breaker for enhanced performance in microservice applications. **International Journal of Electrical & Computer Engineering (2088-8708)**, v. 14, n. 2, 2024. Citado 10 vezes nas páginas 15, 17, 22, 24, 27, 39, 40, 44, 45 e 47.

RASHEEDH, J. A.; SARADHA, S. Design and development of resilient microservices architecture for cloud based applications using hybrid design patterns. **Indian J. Comput. Sci. Eng**, v. 13, n. 2, p. 365–378, 2022. Citado 4 vezes nas páginas 16, 24, 27 e 39.

SILL, A. The design and architecture of microservices. **IEEE Cloud Computing**, v. 3, n. 5, p. 76–80, 2016. Citado 2 vezes nas páginas 14 e 18.

TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems**. 3. ed. [S.l.]: Pearson Education, 2017. Citado 5 vezes nas páginas 14, 18, 37, 42 e 47.

VMWARE, INC. **Spring Boot Reference Documentation: Common Application Properties**. [S.l.], 2024. Acesso em: 07 mar. 2026. Disponível em: <https://docs.spring.io/spring-boot/documentation.html>. Citado 4 vezes nas páginas 19, 32, 39 e 43.

WINKLER, R.; ALI, M.; MUSHKEVYCH, B. **Resilience4j: Fault Tolerance Library for Java**. [S.l.], 2024. Acesso em: 17 abr. 2026. Disponível em: <https://resilience4j.readme.io/docs>. Citado 6 vezes nas páginas 29, 30, 31, 32, 33 e 35.

YU, S.; WU, H.; NIU, X.; NIE, C. A systematic literature review on fault injection testing of microservice systems. **IEEE Transactions on Services Computing**, v. 18, n. 6, p. 4364–4385, 2025. Citado 8 vezes nas páginas 16, 17, 29, 39, 40, 42, 44 e 46.

ZHOU, X.; PENG, X.; XIE, T.; SUN, J.; JI, C.; LI, W.; DING, D. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. **IEEE Transactions on Software Engineering**, v. 47, n. 2, p. 243–260, 2021. Citado 3 vezes nas páginas 14, 19 e 47.

Apêndices

APÊNDICE A – Configurações de Infraestrutura e Resiliência

Este apêndice apresenta o arquivo de orquestração de contêineres e as propriedades da aplicação, definindo a topologia do ecossistema e os parâmetros da Máquina de Estados do *Resilience4j*.

Listing A.1 – docker-compose.yml

```
version: '3.8'

services:
  ms-eureka-server:
    build: ./ms-eureka-server
    ports:
      -"8761:8761"

  ms-payment:
    build: ./ms-payment
    environment:
      -EUREKA.CLIENT.SERVICE-URL.DEFAULTZONE=http://ms-eureka-server:8761/eureka/
    depends_on:
      -ms-eureka-server

  ms-restaurant:
    build: ./ms-restaurant
    environment:
      -EUREKA.CLIENT.SERVICE-URL.DEFAULTZONE=http://ms-eureka-server:8761/eureka/
    depends_on:
      -ms-eureka-server

  ms-delivery:
    build: ./ms-delivery
    environment:
      -EUREKA.CLIENT.SERVICE-URL.DEFAULTZONE=http://ms-eureka-server:8761/eureka/
    depends_on:
      -ms-eureka-server

  ms-order-orchestrator:
    build: ./ms-order-orchestrator
    environment:
      -EUREKA.CLIENT.SERVICE-URL.DEFAULTZONE=http://ms-eureka-server:8761/eureka/
    depends_on:
      -ms-eureka-server
      -ms-payment
      -ms-restaurant
      -ms-delivery

  ms-gateway:
    build: ./ms-gateway
    ports:
```

```
    -"8888:8888"
  environment:
    -EUREKA.CLIENT.SERVICE-URL.DEFAULTZONE=http://ms-eureka-server:8761/eureka/
  depends_on:
    -ms-eureka-server
    -ms-order-orchestrator
```

Listing A.2 – application.properties do orquestrador

```
spring.application.name=ms-order-orchestrator
server.port=8080

eureka.client.service-url.defaultZone=http://ms-eureka-server:8761/eureka/

resilience4j.retry.enabled=true
resilience4j.cb.enabled=true

# ---Config do Retry ---
resilience4j.retry.configs.default.maxAttempts=3
resilience4j.retry.configs.default.waitDuration=500ms
resilience4j.retry.instances.paymentService.baseConfig=default

# ---Config do Circuit Breaker ---
resilience4j.circuitbreaker.configs.default.failureRateThreshold=50
resilience4j.circuitbreaker.configs.default.slidingWindowSize=10
resilience4j.circuitbreaker.configs.default.waitDurationInOpenState=10s
resilience4j.circuitbreaker.configs.default.permittedNumberOfCallsInHalfOpenState=2
resilience4j.circuitbreaker.configs.default.slowCallRateThreshold=100
resilience4j.circuitbreaker.configs.default.slowCallDurationThreshold=2000
resilience4j.circuitbreaker.instances.paymentService.baseConfig=default
```

APÊNDICE B – Scripts de Injeção de Carga no k6

Abaixo estão detalhados os scripts submetidos ao *Grafana k6*. Cada arquivo representa um dos cinco cenários experimentais (C1 a C5), sendo responsáveis por orquestrar os estágios de carga, o tempo de sustentação, a quantidade de Usuários Virtuais (VUs) e o acionamento dos gatilhos de injeção de falhas na arquitetura.

Listing B.1 – baseline.js (C1 - Baseline)

```
import http from 'k6/http';
import { check, sleep, group, fail } from 'k6';

const ORCHESTRATOR_SETUP_URL = 'http://localhost:8888/api/v1/setup/resilience';
const PAYMENT_DEBUG_URL = 'http://localhost:8888/api/v1/payment/debug';
const BASE_URL = 'http://localhost:8888/api/v1/orders';

export const options = {
    vus: 5,
    duration: '10s',
    thresholds: {
        'checks{test_type:happy_path}': ['rate>0.99'],
        'http_req_duration{test_type:happy_path}': ['p(95)<200'],
    },
};

export function setup() {
    // 1. Force Unprotected Architecture (Baseline constraint)
    const archRes = http.post(`${ORCHESTRATOR_SETUP_URL}?retry=false&cb=false`);
    if (archRes.status !== 200) fail(`Architecture setup failed. Status: ${archRes.status}`);

    // 2. Ensure all faults are OFF
    const res = http.get(`${PAYMENT_DEBUG_URL}/turn-off`);
    if (res.status !== 200) fail('Failed to disable faults. Aborting test.');

    console.log('--- BASELINE SETUP: Architecture UNPROTECTED | Faults OFF ---');
}

export default function () {
    group('Happy Path Request', function () {
        const res = http.post(BASE_URL);
        check(res, {
            'status is 200 (OK)': (r) => r.status ===200,
        }, { test_type: 'happy_path' });
    });
    sleep(1);
}
```

Listing B.2 – latency-scenario.js (C2 - Latência Controlada)

```
import http from 'k6/http';
import { check, sleep, group, fail } from 'k6';

const ORCHESTRATOR_SETUP_URL = 'http://localhost:8888/api/v1/setup/resilience';
const PAYMENT_DEBUG_URL = 'http://localhost:8888/api/v1/payment/debug';
const BASE_URL = 'http://localhost:8888/api/v1/orders';

export const options = {
    vus: 50,
    duration: '30s',
};

export function setup() {
    // 1. Read environment variables (default to false if not provided)
    const retry = __ENV.RETRY ==='true';
    const cb = __ENV.CB ==='true';

    // 2. Configure Orchestrator Architecture
    const archRes = http.post(`${ORCHESTRATOR_SETUP_URL}?retry=${retry}&cb=${cb}`);
    if (archRes.status !== 200) fail(`Architecture setup failed. Status: ${archRes.status}`);
    console.log(`--- ARCHITECTURE SETUP: Retry=${retry} | CircuitBreaker=${cb} ---`);

    // 3. Enable Latency Fault
    const res = http.get(`${PAYMENT_DEBUG_URL}/latency/on`);
    if (res.status !== 200) fail('Failed to enable latency. Aborting test.');

    console.log('--- LATENCY SCENARIO: 3s Latency ENABLED ---');
}

export default function () {
    group('Latency Scenario Request', function () {
        const res = http.post(BASE_URL);
        check(res, {
            'status is 200 (OK)': (r) => r.status ===200,
        });
    });
    sleep(1);
}

export function teardown() {
    http.get(`${PAYMENT_DEBUG_URL}/turn-off`);
    console.log('--- CLEANUP: Faults DISABLED ---');
}
```

Listing B.3 – stress-rupture.js (C3 - Ruptura e Estresse)

```
import http from 'k6/http';
import { check, group, fail } from 'k6';

const ORCHESTRATOR_SETUP_URL = 'http://localhost:8888/api/v1/setup/resilience';
const DEBUG_URL = 'http://localhost:8888/api/v1/payment/debug';
const ORDERS_URL = 'http://localhost:8888/api/v1/orders';

export const options = {
    stages: [
        { duration: '30s', target: 50 }, // Warm-up
        { duration: '1m', target: 300 }, // Stress: Exceeding the 200-thread Tomcat limit
```

```
        { duration: '30s', target: 0 }, // Cool-down
    ],
};

export function setup() {
    const retry = __ENV.RETRY ==='true';
    const cb = __ENV.CB ==='true';

    const archRes = http.post(`${ORCHESTRATOR_SETUP_URL}?retry=${retry}&cb=${cb}`);
    if (archRes.status !== 200) fail(`Architecture setup failed. Status: ${archRes.status}`);
    console.log(`--- ARCHITECTURE SETUP: Retry=${retry} | CircuitBreaker=${cb} ---`);

    const faultRes = http.get(`${DEBUG_URL}/latency/on`);
    if (faultRes.status !== 200) fail('Failed to enable latency simulation');
    console.log('--- RUPTURE TEST: 3s Latency ENABLED ---');
}

export default function () {
    group('Rupture Scenario', function () {
        const res = http.post(ORDERS_URL);
        check(res, {
            'is status 200': (r) => r.status ===200,
        });
    });
}

export function teardown() {
    http.get(`${DEBUG_URL}/turn-off`);
    console.log('--- CLEANUP: Simulations DISABLED ---');
}
```

Listing B.4 – test-failure.js (C4 - Falha Rápida)

```
import http from 'k6/http';
import { check, group, fail } from 'k6';

const ORCHESTRATOR_SETUP_URL = 'http://localhost:8888/api/v1/setup/resilience';
const DEBUG_URL = 'http://localhost:8888/api/v1/payment/debug';
const ORDERS_URL = 'http://localhost:8888/api/v1/orders';

export const options = {
    vus: 50,
    duration: '1m',
};

export function setup() {
    // 1. Force Hybrid Architecture (CB + Retry)
    const archRes = http.post(`${ORCHESTRATOR_SETUP_URL}?retry=true&cb=true`);
    if (archRes.status !== 200) fail(`Architecture setup failed. Status: ${archRes.status}`);

    // 2. Enable immediate 503 errors
    const faultRes = http.get(`${DEBUG_URL}/error/on`);
    if (faultRes.status !== 200) fail('Failed to enable error simulation');

    console.log('--- FAST FAILURE TEST: 503 Errors ENABLED | Hybrid Architecture ---');
}
```

```
export default function () {
    group('Error Scenario', function () {
        const res = http.post(ORDERS_URL);
        check(res, {
            'is status 200': (r) => r.status ===200,
        });
    });
}

export function teardown() {
    http.get(`${DEBUG_URL}/turn-off`);
    console.log('--- CLEANUP: Simulations DISABLED ---');
}
```

Listing B.5 – self-healing.js (C5 - Auto-Recuperação)

```
import http from 'k6/http';
import { check, group, fail } from 'k6';

const ORCHESTRATOR_SETUP_URL = 'http://localhost:8888/api/v1/setup/resilience';
const DEBUG_URL = 'http://localhost:8888/api/v1/payment/debug';
const ORDERS_URL = 'http://localhost:8888/api/v1/orders';

export const options = {
    vus: 50,
    duration: '2m',
};

export function setup() {
    // 1. Force Hybrid Architecture (CB + Retry)
    const archRes = http.post(`${ORCHESTRATOR_SETUP_URL}?retry=true&cb=true`);
    if (archRes.status !== 200) fail(`Architecture setup failed. Status: ${archRes.status}`);

    // 2. Start the test with the system in a failure state
    const res = http.get(`${DEBUG_URL}/latency/on`);
    if (res.status !== 200) fail('Failed to enable latency');

    console.log('--- SELF-HEALING TEST: System starting in FAILURE state | Hybrid Arch ---');
    console.log('!!! ACTION REQUIRED: Run "turn-off" cURL at the 01:00 mark !!!');
}

export default function () {
    group('Recovery Scenario', function () {
        const res = http.post(ORDERS_URL);
        check(res, {
            'is status 200': (r) => r.status ===200,
        });
    });
}

export function teardown() {
    http.get(`${DEBUG_URL}/turn-off`);
    console.log('--- CLEANUP: Faults DISABLED ---');
}
```

APÊNDICE C – Implementação dos Padrões e Injeção de Falhas

Este apêndice apresenta o código-fonte da classe `PaymentService`, responsável por gerenciar a comunicação síncrona com a dependência de pagamentos. A implementação destaca o uso da abordagem funcional do *Resilience4j* para decorar dinamicamente a requisição HTTP com os padrões *Retry* e *Circuit Breaker*, formando a arquitetura híbrida avaliada nos testes. Adicionalmente, o código demonstra a estratégia de *fallback*, responsável por interceptar falhas e exceções de roteamento para retornar respostas de falha rápida (*fail-fast* com *status* HTTP 503), impedindo o travamento da *thread* original.

Listing C.1 – PaymentService.java

```
package br.ufu.gabrieloliveira.ms_order_orchestrator.service;

import io.github.resilience4j.circuitbreaker.CircuitBreaker;
import io.github.resilience4j.circuitbreaker.CircuitBreakerRegistry;
import io.github.resilience4j.retry.Retry;
import io.github.resilience4j.retry.RetryRegistry;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.util.function.Supplier;

@Service
public class PaymentService {

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private CircuitBreakerRegistry circuitBreakerRegistry;

    @Autowired
    private RetryRegistry retryRegistry;

    private static final String PAYMENT_SERVICE_CB = "paymentService";

    @Value("${resilience4j.retry.enabled:true}")
    private boolean retryEnabled;

    @Value("${resilience4j.cb.enabled:true}")
    private boolean cbEnabled;

    public ResponseEntity<String> callPaymentService() {
        Supplier<ResponseEntity<String>> paymentCall = () -> {
```

```
            String response = restTemplate.postForObject(
                    "http://ms-payment/pay",
                    null,
                    String.class
            );
            return ResponseEntity.ok(response);
        };

        if (retryEnabled) {
            Retry retryInstance = retryRegistry.retry(PAYMENT_SERVICE_CB);
            paymentCall = Retry.decorateSupplier(retryInstance, paymentCall);
        }

        if (cbEnabled) {
            CircuitBreaker cbInstance = circuitBreakerRegistry.circuitBreaker(PAYMENT_SERVICE_CB)
                ↪ ;
            paymentCall = CircuitBreaker.decorateSupplier(cbInstance, paymentCall);
        }

        try {
            return paymentCall.get();
        } catch (Throwable t) {
            return fallbackPayment(t);
        }
    }

    public ResponseEntity<String> fallbackPayment(Throwable t) {
        String errorBody = "{\"status\":␣\"PAGAMENTO_FALHOU\",␣\"error_details\":␣\"" + t.
            ↪ getMessage() + "\"}";
        return ResponseEntity.status(503).body(errorBody);
    }

    public void setArchitecture(boolean retry, boolean cb) {
        this.retryEnabled = retry;
        this.cbEnabled = cb;
    }
}
```