ResilienceBench: Um Ambiente para Avaliação Experimental de Padrões de Resiliência para Microsserviços*

Carlos M. Aderaldo[1], **Nabor C. Mendonça**[1]

[1] Programa de Pós-Graduação em Informática Aplicada
Universidade de Fortaleza (UNIFOR)
Fortaleza – CE

cmendesce@gmail.com,nabor@unifor.br

Abstract. *Microservice developers increasingly use resiliency patterns, such as Retry and Circuit Breaker, to cope with potential failures during the invocation of remote services. However, there is still little work on the impact of those resiliency patterns on application performance. This paper introduces ResilienceBench, a language-agnostic environment to support microservice developers in experimentally evaluating the performance impact of resilience patterns. The paper describes ResilienceBench's motivation, architecture, and core functionalities, and illustrates its use by presenting an experimental evaluation of the performance impact of the Retry pattern implemented by two popular open source resilience libraries: Polly, for C#, and Resilience4j, for Java.*

Resumo. *Desenvolvedores de microsserviços cada vez mais utilizam padrões de resiliência, como Retry e Circuit Breaker, para lidar com possíveis falhas durante a invocação de serviços remotos. Porém, ainda há poucos trabalhos sobre o impacto do uso desses padrões no desempenho das aplicações. Este artigo apresenta ResilienceBench, um ambiente independente de linguagem para apoiar os desenvolvedores de microsserviços na avaliação experimental do impacto de desempenho de padrões de resiliência. O artigo descreve a motivação, a arquitetura, e as principais funcionalidades do ambiente ResilienceBench, e ilustra o seu uso por meio da avaliação do impacto de desempenho do padrão Retry utilizando duas populares bibliotecas de resiliência abertas: Polly, para a linguagem C#, e Resilience4j, para a linguagem Java.*

1. Introdução

Aplicações de microsserviços, como qualquer outro tipo de sistema distribuído, são propensas a diferentes tipos de falhas operacionais. Causas diversas, como falhas hardware, de software, ou excesso de carga na rede ou em serviços externos, podem tornar os microsserviços da aplicação indisponíveis ou inacessíveis a seus clientes, a qualquer momento [Jamshidi et al. 2018]. Para mitigar o impacto da indisponibilidade temporária dos microsserviços, os desenvolvedores precisam equipar suas aplicações com mecanismos para lidar com falhas de forma robusta e resiliente [Beyer et al. 2016]. Um abordagem

*O repositório do ambiente ResilienceBench, incluindo seu código fonte, arquivos de configuração, instruções de instalação e uso, dados experimentais, e vídeo de demonstração, está disponível publicamente em `https://github.com/ppgia-unifor/resiliency-pattern-benchmark/`

cada vez mais adotada na indústria para lidar com diferentes tipos de falhas operacionais em aplicações de microsserviços é invocar serviços remotos utilizando *padrões de resiliência* [Nygard 2007].

Padrões de resiliência, como Retry e Circuit Breaker, introduzem mecanismos capazes de aguardar passivamente a recuperação de serviços remotos cuja invocação falhou, evitando, assim, que a aplicação desperdice valiosos recursos de processamento e de rede re-invocando continuamente serviços que estejam sobrecarregados ou momentaneamente inoperantes [Microsoft Azure 2017]. No entanto, a documentação disponível sobre o uso desses padrões é, tipicamente, omissa quanto ao seu potencial impacto no desempenho das aplicações. Entender o impacto de padrões de resiliência no desempenho de aplicações distribuídas é importante porque, ao retardarem a re-invocação de serviços remotos em situações de falha, tais padrões podem aumentar significativamente o tempo total de execução dos serviços que os utilizam, podendo, por sua vez, levar a falhas em cascata de outros serviços da aplicação [Mendonca et al. 2020].

Apesar do crescente interesse da comunidade científica pelo estudo de modelos, técnicas e *benchmarks* para avaliar a resiliência de aplicações de microserviços [Aderaldo et al. 2017], ainda há relativamente poucos trabalhos publicados na literatura com foco na avaliação de padrões de resiliência [Preuveneers and Joosen 2017, Aquino et al. 2019, Mendonca et al. 2020, Jagadeesan and Mendiratta 2020, Saleh Sedghpour et al. 2022, Costa et al. 2022]. Em particular, com exceção do trabalho de Costa *et al.*, nenhum dos outros trabalhos citados avaliou experimentalmente o impacto de desempenho de padrões de resiliência implementados por bibliotecas de resiliência amplamente utilizas na indústria, nem disponibilizou publicamente ferramentas para apoiar os desenvolvedores de microsserviços nessa tarefa.

Este artigo apresenta *ResilienceBench*, um ambiente independente de linguagem para apoiar a avaliação experimental de padrões de resiliência em aplicações de microsserviços. ResilienceBench foi recentemente utilizado por Costa *et al.* para a avaliar o impacto de desempenho dos padrões Retry e Circuit Breaker [Costa et al. 2022]. Este artigo complementa o trabalho de Costa *et al.* ao descrever em mais detalhes a arquitetura e as principais funcionalidades do ambiente ResilienceBench (Seção 2). Além disso, o artigo apresenta um exemplo de uso de ResilienceBench para avaliar o impacto de desempenho do padrão Retry utilizando duas populares bibliotecas de resiliência, Polly [App vNext 2022] e Resilience4j [Resilience4j 2021] (Seção 3). Por fim, o artigo apresenta as conclusões do trabalho e oferece sugestões para trabalhos futuros (Seção 4).

2. ResilienceBench

2.1. Arquitetura

A arquitetura do ambiente ResilienceBench contém quatro componentes principais, como mostra a Figura 1: um *escalonador*, um *serviço cliente*, um *serviço intermediário*, e um *serviço alvo*. O escalonador é responsável por (i) interpretar e executar os cenários de teste especificados pelo usuário do ambiente na forma de um arquivo JSON; (ii) invocar o serviço cliente com os parâmetros de resiliência especificados nos cenários de teste; e (iii) coletar os resultados dos testes recebidos do serviço cliente e devolvê-los ao usuário na forma de um arquivo CSV. Alternativamente, o usuário do ambiente pode configurar

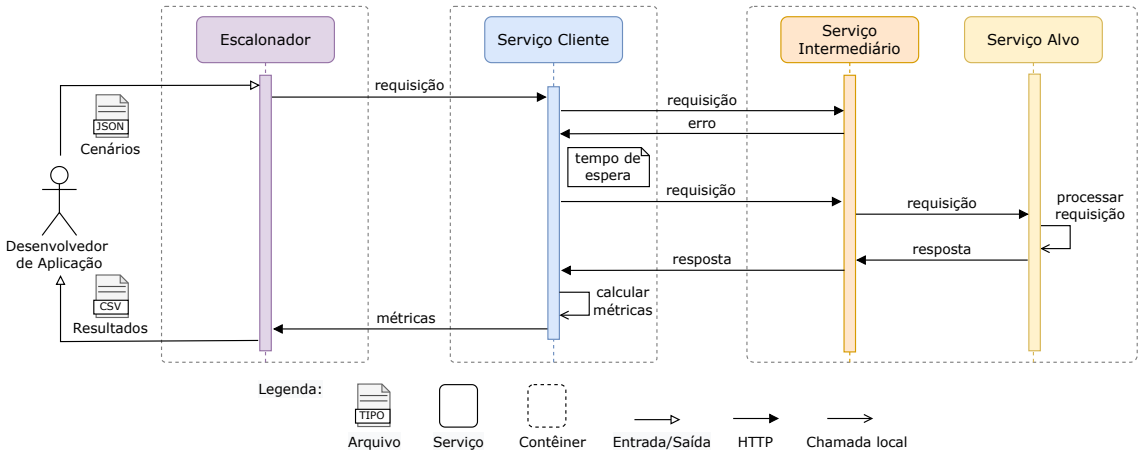

Figura 1. Arquitetura do ambiente ResilienceBench.

o escalonador para armazenar o arquivo CSV localmente, ou em um serviço de armazenamento na nuvem. O serviço cliente, por sua vez, é responsável por (i) invocar o serviço alvo utilizando uma determinada estratégia de resiliência, tal como definida pelos parâmetros recebidos do escalonador; (i) calcular um conjunto de métricas de desempenho a partir do resultado das invocações ao serviço alvo; e (iii) devolver as métricas coletadas ao escalonador. Já o serviço intermediário é encarregado de injetar falhas de um determinado tipo no fluxo de invocação do serviço alvo pelo serviço cliente, de acordo com um determinada percentual de falha. Por fim, o serviço alvo tem como única função processar e responder às requisições do serviço cliente.

O escalonador é implementado em Python, sendo o único componente fixo do ambiente. O serviço cliente pode ser implementado em qualquer linguagem de programação que ofereça suporte à utilização de padrões de resiliência nas invocações a serviços remotos. Até o momento, o ambiente inclui duas versões do serviço cliente com suporte aos padrões Retry e Circuit Breaker. Uma versão é implementada em C#, utilizando a biblioteca de resiliência Polly [App vNext 2022], e outra, em Java, utilizando a biblioteca Resilience4j [Resilience4j 2021]. O serviço intermediário pode ser qualquer serviço *proxy* com suporte a injeção de falhas em fluxo de requisições HTTP. A atual versão do ambiente utiliza o Envoy [Envoy 2022] como serviço intermediário, um serviço *proxy* muito utilizado para gerenciamento de tráfego e injeção de falhas em serviços de malha para microsserviços [Calcote 2018]. Por fim, o serviço alvo pode ser qualquer serviço que responda requisições seguindo o protocolo HTTP, sendo completamente independente dos outros serviços que compõem o ambiente de teste. Nos experimentos descritos neste trabalho, foi escolhido como serviço alvo o httpbin [Postman 2017], um serviço HTTP de código aberto muito utilizado em testes de aplicações Web.

O escalonador e o serviço cliente são implantados, separadamente, em dois contêineres Docker [Merkel 2014], enquanto o serviço intermediário e o serviço alvo são implantados, conjuntamente, em um terceiro contêiner (ver Figura 1). A razão para a implantação desses dois últimos serviços em um mesmo contêiner é tripla: (i) minimizar o impacto de eventuais atrasos de comunicação entre os dois serviços no resultado dos testes; (ii) tornar a presença do serviço intermediário transparente para o serviço cliente; e (iii) evitar que, ao bloquear parte das requisições enviadas pelo serviço cli-

```
1 {
2   "concurrentUsers": [25, 50, 100],
3   "rounds": 10,
4   "maxRequestsAllowed": 100,
5   "targetSuccessfulRequests": 25,
6   "fault": {
7     "type": "abort",
8     "percentage": [0, 25, 50],
9     "status": 503
10  },
11  "patterns": [
12    {
13      "pattern": "retry",
14      "platform": "dotnet",
15      "lib": "polly",
16      "url": "http://polly/retry",
17      "patternConfig": {
18        "count": 5,
19        "sleepDurationType": "EXP_BACKOFF",
20        "exponentialBackoffPow": 1.5,
21        "sleepDuration": [50, 75, 100]
22      }
23    },
24    {
25      "pattern": "retry",
26      "platform": "java",
27      "lib": "resilience4j",
28      "url": "http://resilience4j/retry",
29      "patternConfig": {
30        "maxAttempts": 6,
31        "multiplier": 1.5,
32        "intervalFunction": "EXP_BACKOFF",
33        "initialIntervalMillis": [50, 75, 100]
34      }
35    }
36  ]
37 }
```

Listagem 1: Especificação de múltiplos cenários de teste com o padrão Retry.

ente ao serviço alvo, o serviço intermediário acabe por reduzir artificialmente a carga de trabalho imposta ao (contêiner do) serviço alvo durante os testes. As ferramentas Vagrant [HashiCorp 2022] e Docker Compose [Docker 2021] são utilizadas para, respectivamente, criar a máquina virtual onde são executados os componentes do ambiente, e configurar, implantar, e executar os contêineres Docker contendo esses componentes.

2.2. Cenários de Teste

Um cenário de teste consiste em uma sessão de testes na qual o serviço cliente invoca continuamente o serviço alvo até atingir uma determinada quantidade de invocações bem sucedidas. Durante os testes, o serviço alvo pode falhar seguindo uma determinada taxa de falha. A falha do serviço alvo pode levar o serviço cliente a reinvocar o serviço alvo imediatamente após cada falha, ou a aguardar por um determinado período de tempo antes de reinvocar o serviço novamente, a depender do padrão de resiliência utilizado (por exemplo, Retry ou Circuit Breaker) e dos valores dos parâmetros escolhidos para configurar esse padrão (por exemplo, o número máximo de reinvocações permitidas). Desta forma, a utilização de diferentes padrões de resiliência, e de diferentes configurações desses padrões, afetará diretamente o comportamento do serviço cliente diante de situações de falha. Esse comportamento, por sua vez, terá impacto direto não apenas no tempo total de execução do serviço cliente, como também no seu nível de utilização de recursos externos, como a rede e o próprio serviço alvo.

A especificação de um cenário de testes contém todas as informações necessárias para a execução do cenário pelos componentes do ResilienceBench. A Listagem 1 mostra um exemplo de uma especificação contento múltiplos cenários de teste com o padrão Retry envolvendo as duas versões do serviço cliente disponíveis no ambiente ResilienceBench, implementadas em C# e Java, respectivamente. As linhas 2-10 definem os parâmetros de controle dos testes, incluindo a quantidade de usuários concorrentes (instâncias do serviço cliente) que irão invocar o serviço alvo em cada cenário (linha 2); a quantidade de execuções de cada cenário (linha 3); a quantidade máxima de requisições ao serviço alvo permitidas por cenário (linha 4); quantidade mínima esperada de requisições bem sucedidas ao serviço alvo (linha 5); e o tipo, o percentual, e o código das falhas a serem injetadas pelo serviço intermediário (linhas 6-10). As demais linhas da especificação definem os parâmetros de configuração do padrão Retry para as versões do serviço cliente implementadas em C# (linhas 12-23) e em Java (linhas 24-35).

Tabela 1. Métricas de desempenho coletadas pelo serviço cliente.

Métrica	Descrição
SuccessfulCalls *UnsuccessfulCalls* *TotalCalls*	Quantidade de invocações bem sucedidas / mal sucedidas / no geral ao serviço alvo realizadas pelo serviço cliente, sem considerar eventuais bloqueios e re-invocações realizados pelo padrão de resiliência utilizado
SuccessfulRequests *UnsuccessfulRequests* *TotalRequests*	Quantidade de invocações bem sucedidas / mal sucedidas / no geral ao serviço alvo realizadas diretamente pelo padrão de resiliência utilizado pelo serviço cliente
SuccessTime *SuccessTimePerRequest*	Tempo de resposta acumulado / médio das invocações bem sucedidas ao serviço alvo pelo serviço cliente (em milissegundos)
ErrorTime *ErrorTimePerRequest*	Tempo de resposta acumulado / médio das invocações mal sucedidas ao serviço alvo pelo serviço cliente (em milissegundos)
Throughput	Taxa de invocações ao serviço alvo pelo serviço cliente (em invocações por milissegundo)
TotalExecutionTime	Tempo total de execução do serviço cliente até completar o número esperado de invocações bem sucedidas ao serviço alvo (em milissegundos)
TotalContentionTime	Tempo total acumulado pelo serviço cliente invocando ou aguardando resposta do serviço alvo, considerando tanto invocações bem sucedidas quanto mal sucedidas (em milissegundos)
ContentionRate	Razão entre *TotalContentionTime* e *TotalExecutionTime*

2.3. Expansão dos Cenários

Após ler e interpretar o arquivo JSON com a especificação dos cenários de teste, o escalonador expande esta especificação em múltiplos cenários distintos, para então executá-los sequencialmente, de maneira a evitar que a execução de um cenário interfira na execução dos outros. Para isso, o escalonador expande todos os parâmetros da especificação dos cenários que foram definidos com uma lista de valores, gerando diferentes permutações de cenários, com os parâmetros de cada cenário contendo apenas valores individuais. Por exemplo, na especificação mostrada na Listagem 1, serão expandidos os parâmetros referentes à quantidade de usuários que irá invocar o serviço alvo (linha 2), ao percentual de falhas que será injetada pelo serviço intermediário (linha 8), e aos intervalos de espera iniciais entre as re-invocações do serviço alvo de cada implementação do padrão Retry (linhas 21 e 33, respectivamente).

2.4. Métricas de Desempenho

Durante a execução de cada cenário de teste, o serviço cliente coleta um conjunto de métricas de desempenho, o qual é devolvido ao escalonador ao final da execução do cenário. A Tabela 1 descreve as métricas coletadas pelas duas versões do serviço cliente atualmente implementadas no ambiente ResilienceBench. O escalonador consolida todas as métricas recebidas do serviço cliente e, ao final dos testes, devolve os dados consolidados ao usuário do ambiente em um arquivo no formato CSV.

3. Exemplo de Uso

Para ilustrar o uso do ambiente ResilienceBench, esta seção apresenta e analisa brevemente os resultados obtidos a partir da execução dos cenários de teste especificados na Listagem 1. Uma análise mais detalhada do impacto de desempenho do uso de diferentes configurações dos padrões Retry e Circuit Breaker, utilizando as bibliotecas de resiliência Polly e Resilience4j, pode ser encontrada em [Costa et al. 2022].

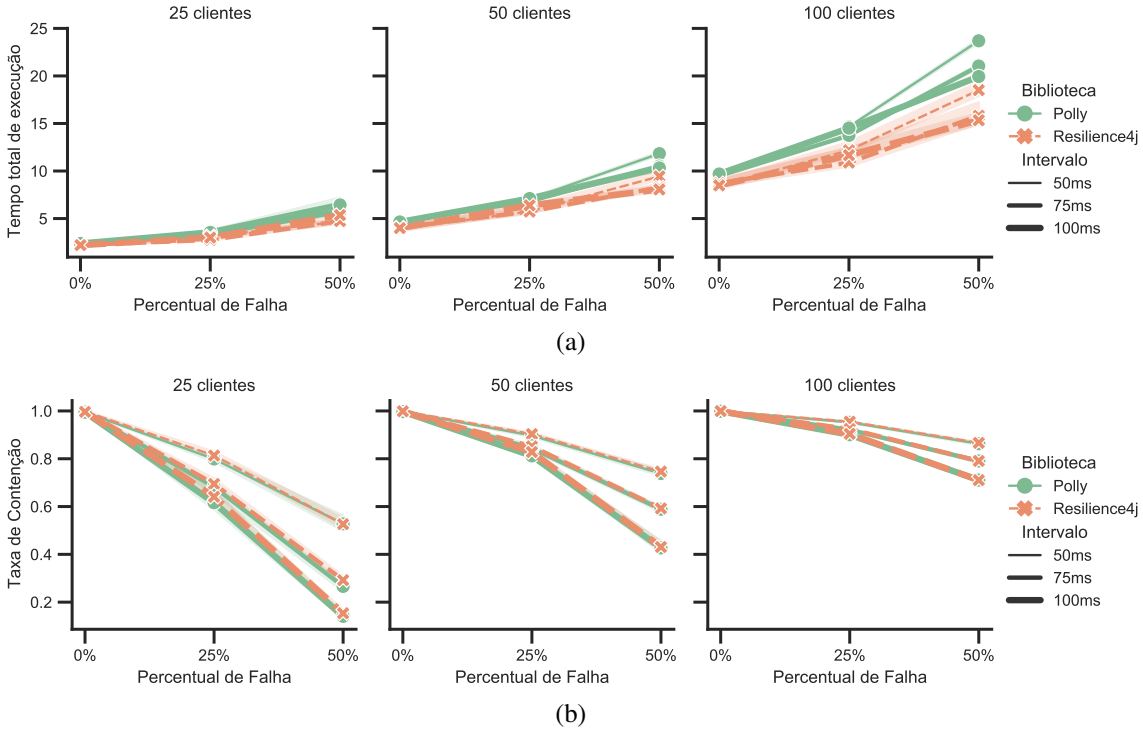

Figura 2. Impacto do percentual de falha do serviço alvo no tempo total de execução (a) e na taxa de contenção (b) do serviço cliente utilizando o padrão Retry com as duas bibliotecas de resiliências avaliadas.

A análise descrita aqui foca nas métricas tempo total de execução (*TotalExecutionTime*) e taxa de contenção (*ContentionRate*). Estas duas métricas foram escolhidas por duas razões: (i) ambas representam importantes atributos de qualidade de uma aplicação de microsserviços; e (ii) buscar otimizá-las simultaneamente durante situações de falha pode constituir objetivos mutuamente conflitantes [Costa et al. 2022].

3.1. Tempo Total de Execução

A Figura 2(a) mostra os resultados para o tempo total de execução. Por essa figura, observa-se que o tempo total de execução do serviço cliente utilizando o padrão Retry cresce à medida que crescem o percentual de falha do serviço alvo e a quantidade de usuários invocando o serviço alvo concorrentemente. Este resultado está dentro do esperado, uma vez que um maior percentual de falha do serviço alvo implica que o serviço cliente precisa gastar mais tempo reinvocando e aguardando pela recuperação do serviço alvo. Além disso, com o aumento da carga de trabalho, o serviço alvo fica mais sobrecarregado e demora mais a responder as requisições do serviço cliente, o que também contribui para aumentar o tempo total de execução do serviço cliente.

Quando comparam-se os resultados obtidos pelas duas bibliotecas de resiliência, observa-se que os tempos de execução do serviço cliente são ligeiramente maiores quando este utiliza o padrão Retry implementado pela biblioteca Polly. No entanto, o fato dessas duas bibliotecas serem implementadas em diferentes linguagens de programação, com cada linguagem tendo seu próprio ambiente de execução, impede que essas diferenças nos resultados possam ser atribuídas exclusivamente a diferenças na forma como cada biblioteca implementa o padrão Retry.

3.2. Taxa de Contenção

A Figura 2(b), por sua vez, mostra que o uso do padrão Retry consegue reduzir substancialmente a taxa de contenção do serviço cliente, com ambas bibliotecas de resiliência, e que esta redução diminui à medida que crescem o percentual de falha e a carga de trabalho do serviço do alvo. Este resultado justifica-se pelo fato do padrão Retry forçar o serviço cliente a aguardar pela recuperação do serviço alvo após a falha deste, evitando, assim, que o serviço cliente reinvoque o serviço alvo imediatamente. Já a redução do impacto do padrão Retry na taxa de contenção do serviço cliente para maiores níveis de carga de trabalho acontece porque, como explicado anteriormente, sob maiores níveis de carga de trabalho, o tempo total de execução do serviço cliente passa a ser dominado pelo tempo de resposta do serviço alvo, sofrendo menos influência dos tempos de espera impostos pelo uso do padrão Retry.

Curiosamente, a Figura 2(b) mostra que ambas as bibliotecas de resiliência obtiveram resultados praticamente idênticos em termos do impacto do padrão Retry na taxa de contenção do serviço cliente. Esses resultados parecem conflitar com a hipótese levantada anteriormente, de que as diferenças de desempenho entre as duas bibliotecas poderiam ser devidas a diferenças inerentes às suas respectivas linguagens de programação ou a seus ambientes de execução. Em todo caso, estes resultados evidenciam não apenas a importância de se avaliar experimentalmente o desempenho de bibliotecas de resiliência, mas também a necessidade de se investigar mais a fundo as suas similaridades e diferenças.

4. Conclusão

Este trabalho apresentou a arquitetura, as principais funcionalidades, e um exemplo de uso do ambiente ResilienceBench, desenvolvido com o objetivo de apoiar a avaliação experimental de padrões de resiliência em aplicações de microsserviços. Espera-se que as facilidades oferecidas pelo ambiente ResilienceBench possam ser úteis tanto aos desenvolvedores quanto aos pesquisadores interessados em microsserviços. Os desenvolvedores poderão se beneficiar do ambiente para estimar o impacto que o uso de padrões de resiliência poderá gerar no desempenho de suas aplicações. Já os pesquisadores poderão utilizar e customizar o ambiente para investigar mais sistematicamente a relação entre padrões de resiliência e diferentes atributos de qualidade.

ResilienceBench está disponível publicamente e continua sendo ativamente melhorado e estendido com novas funcionalidades.[1] Algumas dessas extensões incluem: implementar novas versões do serviço cliente, utilizando outros padrões e bibliotecas de resiliência, e outras linguagens de programação; oferecer suporte a aplicações de microsserviços mais realistas, contendo múltiplos serviços de *upstream* e *downstream*; e permitir a avaliação de padrões de resiliências implementados no contexto de malhas de serviços [Saleh Sedghpour et al. 2022].

Referências

Aderaldo, C. M. et al. (2017). Benchmark Requirements for Microservices Architecture Research. In *1st Int. Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 8–13.

[1] `https://github.com/ppgia-unifor/resiliency-pattern-benchmark/`

App vNext (2022). Polly. https://github.com/App-vNext/Polly. [Último acesso em 17 de Maio de 2022].

Aquino, G. et al. (2019). The circuit breaker pattern targeted to future IoT applications. In *Int. Conf. Service Oriented Computing (ICSOC)*, pages 390–396. Springer.

Beyer, B. et al. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly.

Calcote, L. (2018). *The Enterprise Path to Service Mesh Architectures*. O'Reilly.

Costa, T. M. et al. (2022). Avaliação de Desempenho de Dois Padrões de Resiliência para Microsserviços: Retry e Circuit Breaker. In *XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*. Aceito para publicação.

Docker (2021). Overview of Docker Compose. https://docs.docker.com/compose/. [Último acesso em 17 de Maio de 2022].

Envoy (2022). Envoy Proxy. https://www.envoyproxy.io. [Último acesso em 17 de Maio de 2022].

HashiCorp (2022). Vagrant: Development Environments Made Easy. https://www.vagrantup.com/. [Último acesso em 17 de Maio de 2022].

Jagadeesan, L. J. and Mendiratta, V. B. (2020). When failure is (not) an option: Reliability models for microservices architectures. In *IEEE Int. Symp. Software Reliability Engineering Workshops (ISSREW)*, pages 19–24.

Jamshidi, P. et al. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.

Mendonca, N. C. et al. (2020). Model-based analysis of microservice resiliency patterns. In *IEEE Int. Conf. Software Architecture (ICSA)*, pages 114–124.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2.

Microsoft Azure (2017). Resiliency patterns. https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency. [Último acesso em 17 de Maio de 2022].

Nygard, M. (2007). *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf.

Postman (2017). httpbin(1): HTTP Request & Response Service. https://github.com/postmanlabs/httpbin. [Último acesso em 17 de Maio de 2022].

Preuveneers, D. and Joosen, W. (2017). QoC^2 Breaker: intelligent software circuit breakers for fault-tolerant distributed context-aware applications. *Journal of Reliable Intelligent Environments*, 3(1):5–20.

Resilience4j (2021). Fault tolerance library designed for functional programming. https://github.com/resilience4j/resilience4j. [Último acesso em 17 de Maio de 2022].

Saleh Sedghpour, M. R., Klein, C., and Tordsson, J. (2022). An Empirical Study of Service Mesh Traffic Management Policies for Microservices. In *ACM/SPEC Int. Conf. Performance Engineering (ICPE)*, pages 17–27.