ResilienceBench-Operator: A Kubernetes Extension for Orchestrating Resilience Experiments on Microservice Applications

Carlos M. Aderaldo
Programa de Pós-Graduação em Informática Aplicada
Universidade de Fortaleza
Fortaleza, Ceará, Brazil
cmendesce@gmail.com

Nabor C. Mendonça
Programa de Pós-Graduação em Informática Aplicada
Universidade de Fortaleza
Fortaleza, Ceará, Brazil
nabor@unifor.br

ABSTRACT

Microservice-based applications often rely on resiliency patterns such as Retry and Circuit Breaker to mitigate the impact of failures in service-to-service communication. However, there is still limited tooling support for systematically and reproducibly evaluating the performance impact of these patterns in realistic deployment environments. This paper presents ResilienceBench-Operator, a Kubernetes native tool that orchestrates resilience experiments directly on microservice applications deployed in Kubernetes clusters. ResilienceBench-Operator is an evolution of the original ResilienceBench tool, which was focused on controlled environments with predefined client-server interactions. The new version introduces a fully declarative approach to define test spaces, automatically expands them into concrete test scenarios, injects failures, and configures resiliency strategies across real service dependencies. This paper describes the motivation, architecture, and main functionalities of ResilienceBench-Operator, and illustrates how it enables systematic evaluation of resiliency strategies in a representative microservice system. **Demo video**: https://youtu.be/ZSQcx6Ab37w.

KEYWORDS

microservices, resiliency patterns, benchmarking

1 Introduction

Microservice-based applications are inherently fragile due to their distributed nature [16]. Network delays, transient errors, hardware failures, and server overloads are common sources of problems that can compromise individual components and cause cascading effects throughout the system [17]. To mitigate these issues, developers frequently adopt resiliency patterns such as *Retry* and *Circuit Breaker*, which are designed to enhance fault tolerance and prevent system-wide degradation [18].

However, these patterns introduce trade-offs. When misconfigured, they may increase latency, reduce throughput, or worsen the impact of failures [7]. Their real effect on application performance depends on several factors, including workload, the failure rate of downstream services, and the configuration of the pattern [2]. Despite their widespread adoption, there is limited tooling support for systematically evaluating such trade-offs under realistic and reproducible test conditions.

To start addressing this gap, our previous work introduced ResilienceBench [1, 3], a benchmark tool and a declarative approach to specify and execute controlled experiments to assess the impact of resilience patterns. The tool enabled automated test generation based on combinations of fault types, workloads, programming languages, and pattern configurations. Further experimental studies demonstrated the benefits of this approach and offered practical insights into the behavior of resilience patterns in multiple scenarios [2, 7].

Despite its contributions, the original ResilienceBench tool has key limitations. Its architecture relies on a predefined and artificially contrived client-server model, containing a single client application and single target service, which constrains the realism of the evaluated scenarios. In particular, it does not support experiments involving topologies with multiple downstream services, diverse communication paths, or production-grade microservice applications. This limits its ability to generalize experiment findings and to evaluate application resilience under realistic operational scenarios.

To overcome these challenges, this paper presents ResilienceBench-Operator, a Kubernetes-native tool that extends the original ResilienceBench approach by enabling in situ experimentation on realistic microservice applications deployed in Kubernetes clusters. The new operator orchestrates the execution of resilience benchmark experiments using familiar Kubernetes constructs such as Custom Resource Definitions (CRDs), controllers, jobs, and declarative resource configurations [5]. By embedding the experimentation process into Kubernetes, ResilienceBench-Operator facilitates the evaluation of resilience strategies on existing microservice applications with minimal disruption to existing infrastructure. In addition, the operator-centered design of the tool aligns with the declarative principles of cloud-native development, allowing engineers and researchers to define and automate realistic resilience benchmark experiments using YAML-based Kubernetes specifications [13].

The remainder of this paper gives a quick overview of the original ResilienceBench tool (Section 2); describes the architecture and key functionalities of ResilienceBench-Operator (Section 3); illustrates its usage to assess the resilience of a representative open-source microservice demo application (Section 4); and discusses related work (Section 5). The paper ends with some conclusions and directions for future research (Section 6).

2 The Original ResilienceBench

ResilienceBench [3] was developed as a benchmark tool and declarative approach to evaluate microservice resiliency patterns in a controlled environment. One of its main contributions was the introduction of the *test space* abstraction—a high-level declarative

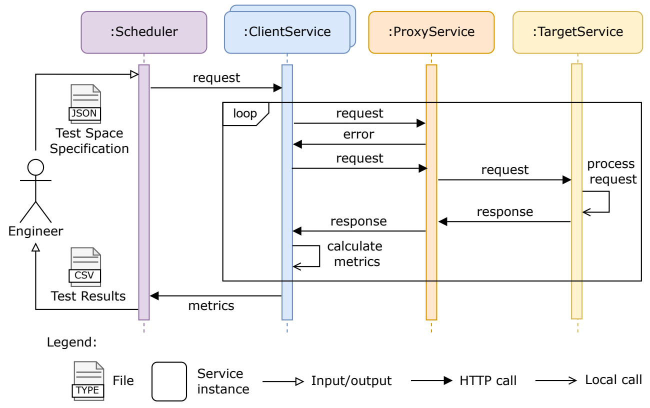

Figure 1: ResilienceBench's run-time architecture.

specification that compactly encodes a wide range of testing conditions, including fault injection rates, workloads, pattern configurations, and resilience libraries [1]. The tool expands each test space specification into concrete *test scenarios*, which are then executed sequentially to enable systematic and repeatable experiments. Another important feature of ResilienceBench is its extensibility. The architecture allows developers to implement custom clients in different programming languages and with different resilience libraries, enabling comparative evaluations in different software ecosystems, such as Java with Resilience4j [20] or C# with Polly [19].

At run-time, ResilienceBench comprises four main services, as depicted in Fig. 1: a *scheduler*, a *client service*, a *proxy service*, and a *target service*. The scheduler plays the role of both the scenario generator and the scenario executor. It (i) parses and expands the test space specification provided by the user as a JSON file; (ii) invokes each of the client service instances at a time, passing them the required test parameters (e.g., the target service's URL and failure rate, the client service's number of virtual users and resiliency strategy configuration, etc.); (iii) collects and consolidates the performance metrics received from each client service instance; and (iv) returns the consolidated performance metrics to the user as a set of CSV files. Each client service instance, in turn, (i) continuously invokes the target service, using the provided resilience strategy configuration, until they reach the required number of successful invocations; (ii) collects and calculates a set of pre-defined performance metrics after each target service invocation; and (iii) returns the collected performance metrics to the scheduler. The proxy service is responsible for injecting faults of a certain type (e.g., an abort failure) into the target service's request stream according to the specified failure rate. For instance, a 25% failure rate means that the proxy service will inject one failure for every three requests the target service receives from the client application. Finally, the target service serves the client service's requests and is entirely oblivious of the other services.

While these design choices make ResilienceBench a powerful platform for benchmarking of resiliency patterns, they also impose inherent constraints when applied to more realistic microservice-based applications. Because the system under test is built around a fixed and isolated client-server topology, the patterns are evaluated independently of real-world service dependencies, failure propagation paths, and dynamic runtime behaviors. The tight coupling between load generation, metric collection, and client implementation introduces redundancy and overhead when scaling to more complex environments. Moreover, although the tool leverages containerization for environment consistency and isolation, it lacks integration with popular cloud-native platforms like Kubernetes. This restricts its applicability to real microservice deployments, where resilience strategies must be validated within the context of heterogeneous microservices, orchestrated environments, and production-grade infrastructure.

In summary, ResilienceBench established a solid foundation for the controlled evaluation of resiliency patterns, offering valuable abstractions and experimental capabilities. However, to meet the demands of evaluating patterns in realistic microservice scenarios, a new approach is required: one that preserves the declarative nature of ResilienceBench while enabling integration with more realistic cloud-native applications. This need motivated the development of ResilienceBench-Operator, described next.

3 ResilienceBench-Operator

ResilienceBench-Operator is a Kubernetes extension for orchestrating resilience benchmarking experiments in real microservice environments. It builds upon two foundational concepts: (i) the declarative test specification and scenario-based execution model introduced in the original ResilienceBench [1], and (ii) the controller-based extensibility mechanism provided by Kubernetes operators [8]. By combining these concepts, ResilienceBench-Operator enables seamless evaluation of resilience strategies within Kubernetes-managed applications, supporting realistic deployment topologies and fault conditions.

For readers unfamiliar with Kubernetes, key concepts used throughout the paper include: a *controller* (a background process that manages application state), a *job* (a one-off task), a *pod* (the smallest deployable unit in Kubernetes), and a *cluster* (a set of nodes that run containerized workloads) [5].

3.1 Architecture Overview

As illustrated in Fig. 2, ResilienceBench-Operator is deployed as a Kubernetes controller and interacts with both user-defined and internally managed resources to coordinate resilience experiments.

The architecture defines a set of Custom Resource Definitions (CRDs) that extend the Kubernetes API with five new resource types:

- `Benchmark`: defines the overall test specification, including the test space parameters.
- `Workload`: defines the load generation configuration (e.g., number of users, duration).
- `ResilientService`: describes the application service and is referenced in benchmark connectors as a source or destination.
- `Scenario`: represents a fully instantiated test scenario, generated from a benchmark.
- `ExecutionQueue`: manages the sequencing of scenario executions.

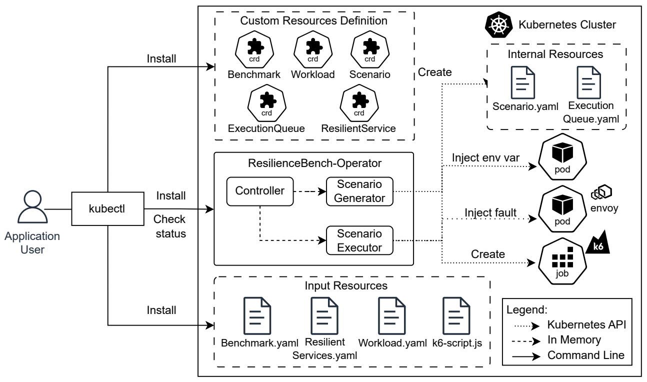

Figure 2: ResilienceBench-Operator's architecture

The operator consists of a Controller, a Scenario Generator, and a Scenario Executor. When a new `Benchmark` resource is submitted to the Kubernetes API, the controller detects it and triggers the generation of test scenarios. Each scenario is derived by expanding the test space into combinations of workloads, failure injection rates, and resilience configurations. These scenarios are enqueued in an ExecutionQueue, which controls their sequential execution.

To support in-cluster fault injection and workload execution, ResilienceBench-Operator integrates two widely adopted cloud-native tools: Envoy and K6. Envoy is a high-performance proxy that can be programmatically configured to simulate failures in communication paths utilizing not only HTTP but also gRPC and generic TCP, making it ideal for injecting faults between microservices [9]. K6 is a modern, scriptable load testing tool, designed for testing the performance and resilience of web services over HTTP, gRPC, and WebSockets using custom scenarios defined in JavaScript [11].

After scenario expansion, each test scenario results in the creation of an Envoy-based fault injector, which applies the specified failure rate to inter-service communication; a K6 job, which runs the user-defined load script against the application under test; and internal Kubernetes resources, including the expanded `Scenario.yaml` and `ExecutionQueue.yaml`.

This design enables the tool to evaluate any Kubernetes-based microservice application that supports HTTP/gRPC communication and exposes configurable resilience mechanisms via environment variables. By decoupling workload generation from the core benchmarking logic and leveraging Kubernetes-native components, ResilienceBench-Operator supports scalable, reproducible, and automated experiments on a wide range of protocols supported by Envoy and K6.

3.2 Scenario Specification and Expansion

A benchmark experiment is specified using YAML files that describe the services to be evaluated (`ResilientServices.yaml`), the desired workloads (`Workload.yaml`), the test space parameters, including failure rates and resilience settings (`Benchmark.yaml`), and the K6 load test script (`k6-script.js`). The Scenario Generator expands this test space into a series of concrete `Scenario` resources. Each scenario targets a specific fault configuration and resilience strategy, applied to one or more connectors: logical edges in the application topology representing communication between services. These connectors define the fault injection levels (e.g., 25%, 50%) and some environment variables to configure the resilience logic of the calling service. Although our current evaluation focuses on the Retry pattern, the operator's declarative model is designed to support additional resilience strategies, such as circuit breakers or timeouts, through environment variable injection. These capabilities are compatible with most resilience libraries that offer configurable parameters at the application level.

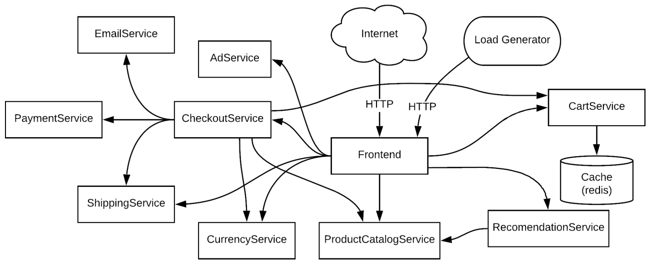

Figure 3: Online Boutique's microservice architecture

ResilienceBench-Operator supports both scenario-level and service-level configuration granularity. In the former, identical fault injection levels are applied across multiple services simultaneously, modeling coordinated failures. In the latter, each service can be independently configured, allowing more granular exploration of fault conditions.

3.3 Scenario Execution and Metrics Collection

The Scenario Executor dequeues and runs each scenario, triggering the deployment of an Envoy sidecar (or external proxy) to simulate faults and the execution of the associated K6 test script as a Kubernetes job. The use of K6 allows users to define flexible and expressive load patterns while collecting detailed performance metrics. These metrics can be persisted locally or pushed to an S3-compatible object storage, depending on the configuration.

4 Usage Example

To demonstrate the use of ResilienceBench-Operator, we conducted a resilience experiment using a modified version of the Online Boutique microservice demo application [10]. We first describe the experimental setup and the tool's configuration, explaining how the tool orchestrates the multiple test scenarios from its main configuration file. Then we present and discuss the results of the experiment.

4.1 Target Application: Online Boutique

Online Boutique is an open source microservice application developed by Google to simulate an e-commerce platform [10]. As shown in Fig. 3, the application comprises multiple microservices, such as *frontend*, *productcatalog*, *checkout*, *payment*, and *recommendation*, implemented in various programming languages (Go, Java, Node.js, Python, and C#). These services communicate primarily through gRPC. This diversity of services and implementation technologies, combined with an official Kubernetes deployment manifest, makes the Online Boutique a useful and representative testbed for cloud native resilience evaluation.

To enable run-time (re)configuration of retry policies, we extended the gRPC clients in selected Online Boutique services to read four environment variables: `GRPC_MAX_ATTEMPTS`, `GRPC_INITIAL_BACKOFF`, `GRPC_MAX_BACKOFF`, and `GRPC_BACKOFF_MULTIPLIER`.

When set, these variables activate gRPC client retries with the specified parameters; otherwise, the service behaves as originally implemented. The resulting modified application is publicly available at: https://github.com/cmendesce/microservices-demo.

4.2 Experiment Configuration

Listing 1 shows the complete YAML specification used to configure the experiment. This configuration uses three custom resources: `ResilientService`, `Workload`, and `Benchmark`. These resources are submitted by the user via the Kubernetes API and automatically processed by the ResilienceBench-Operator.

We define two application services in lines 1–21 using the `ResilientService` kind. Each service is identified by a label selector (lines 8 and 19) and annotated with the name of the container to be controlled (lines 6 and 17). In this case, `checkoutservice` plays the role of the client (`"server"`), while `paymentservice` is the fault injection target (`"envoy"`).

The load generation strategy is described in lines 23–35 using a `Workload` resource. It specifies 3000 iterations per virtual user (line 30), and a configurable user load of 300, 500, or 700 users (line 35). The JavaScript test script is provided via a ConfigMap (lines 32–34), allowing flexible and reusable test logic.

The main experiment is defined in lines 37–64 as a `Benchmark` named `onlineboutique`. It references the workload resource defined earlier (line 42) and defines a single scenario, `retry-checkout` (line 44), which tests the resilience of the checkout-to-payment communication path under varying retry configurations and injected faults. Fault injection is applied to the `paymentservice` using Envoy as the fault provider (line 46), with failure rates of 25%, 50%, and 75% (line 47). A single `connector` is defined (line 51), representing a gRPC interaction from `checkoutservice` to `paymentservice`. Finally, the connector's source (lines 52–62) specifies four retry-related environment variables, `GRPC_MAX_ATTEMPTS`, `GRPC_INITIAL_BACKOFF`, `GRPC_MAX_BACKOFF`, and `GRPC_BACKOFF_MULTIPLIER`, with 4, 3, 1, and 3 values, respectively, to be combined combinatorially at scenario execution time.

In total, this benchmark specification expands into $4 \times 3 \times 1 \times 3 = 36$ unique retry configurations per fault level and workload at execution time, resulting in $3 \times 3 \times 36 = 324$ concrete scenarios automatically generated and executed by the operator across all test scenarios.

```yaml
1  apiVersion: resiliencebench.io/v1beta1
2  kind: ResilientService
3  metadata:
4    name: checkoutservice
5    annotations:
6      resiliencebench.io/container: "server"
7  spec:
8    selector:
9      matchLabels:
10       app: checkoutservice
11 ---
12 apiVersion: resiliencebench.io/v1beta1
13 kind: ResilientService
14 metadata:
15   name: paymentservice
16   annotations:
17     resiliencebench.io/container: "envoy"
18 spec:
19   selector:
20     matchLabels:
21       app: paymentservice
22 ---
23 apiVersion: resiliencebench.io/v1beta1
24 kind: Workload
25 metadata:
26   name: fixed-iterations-loadtest
27 spec:
28   options:
29     - name: K6_ITERATIONS
30       value: "3000"
31   script:
32     configMap:
33       name: k6-config
34       file: k6.js
35   users: [300, 500, 700]
36 ---
37 apiVersion: resiliencebench.io/v1beta1
38 kind: Benchmark
39 metadata:
40   name: onlineboutique
41 spec:
42   workload: fixed-iterations-loadtest
43   scenarios:
44     - name: retry-checkout
45       fault:
46         provider: envoy
47         percentages: [25, 50, 75]
48         services:
49          - paymentservice
50       connectors:
51         - name: checkout-payment
52           source:
53             name: checkoutservice
54             envs:
55             - name: GRPC_MAX_ATTEMPTS
56               value: ["2", "3", "4", "5"]
57             - name: GRPC_INITIAL_BACKOFF
58               value: ["0.5s", "1s", "1.5s"]
59             - name: GRPC_MAX_BACKOFF
60               value: ["15s"]
61             - name: GRPC_BACKOFF_MULTIPLIER
62               value: ["1", "1.5", "2.0"]
63           destination:
64             name: paymentservice
```

Listing 1: ResilienceBench-Operator's configuration.

4.3 Test Environment and Metrics Collection

All experiments were executed in a Kubernetes cluster provisioned on a dedicated server with an AMD Ryzen 9 3950X (16 cores) and 32GB RAM. The cluster was composed of one control plane node (2 vCPUs, 4GB RAM) and three worker nodes (4 vCPUs, 4GB RAM each), all running Ubuntu 22.04 and Kubernetes 1.29. The entire environment was provisioned using Vagrant [12], ensuring reproducibility. The use of a dedicated physical server was a deliberate choice to reduce variability in performance measurements, minimizing performance variability typically introduced by a shared cloud infrastructure.

Workload generation and resilience metric collection were performed using a custom K6 script written in JavaScript. The script simulates realistic e-commerce user sessions by navigating through the Online Boutique application's core user cases: visiting the homepage, browsing products, adding products to the cart, viewing the cart, setting a currency, and performing a checkout operation. Each simulated user executes a complete sequence of operations, enabling fine-grained performance and fault-tolerance evaluation at both the service and session level. The K6 script collects two main metrics after each scenario execution: the *95th percentile of the session execution time* and the *checkout success rate*. To enable cross-scenario comparison, the min-max normalization was applied to normalize the session execution times, where 0 corresponds to the fastest observed execution time in a particular scenario and 1 to the slowest execution time in that scenario. The checkout success rate was calculated as the fraction of completed checkout operations relative to the total number of attempts issued by each client (simulated user).

4.4 Results

Fig. 4 presents the performance results for all 36 retry configurations across nine test scenarios, defined by three fault injection levels (25%, 50%, 75%) and three workload levels (300, 400, and 500 virtual users). Each point in the scatter plots represents a unique retry configuration, with the x-axis showing its relative (min-max normalized) session execution time and the y-axis indicating the checkout success rate. The five configurations with the best trade-offs—those combining the highest success rates and lowest execution times—are highlighted with a dark edge. In each plot, a dotted horizontal gray line marks the baseline success rate achieved by the same application configuration but with the retry mechanism disabled. This baseline varies by scenario, since it reflects the outcome of issuing requests under 25%, 50%, and 75% fault injection levels without any resilience mechanism.

Across all test scenarios, retry mechanisms consistently outperform the baseline, validating their effectiveness in improving success rates even under high failure conditions. For instance, at 75% fault injection, baseline performance drops significantly to 25% whereas many retry configurations achieve 75% success or higher. However, this increase in resilience often comes at the cost of increased latency.

A closer look reveals that achieving a high success rate does not uniquely determine execution time. Multiple configurations yield similar levels of reliability but exhibit widely varying execution times. This effect is visible in nearly every plot—several points aligned along the top of the plot (success rate near 1.0) are dispersed across the x-axis, indicating that some configurations are considerably more efficient than others. For example, in the 25% failure and 300-users scenario, most configurations achieve near-perfect success rates, but their execution times vary widely. Conversely, under heavier load and fault conditions (e.g., 75% failure and 5000 users), fewer configurations meet higher reliability goals, and the performance gap among them widens.

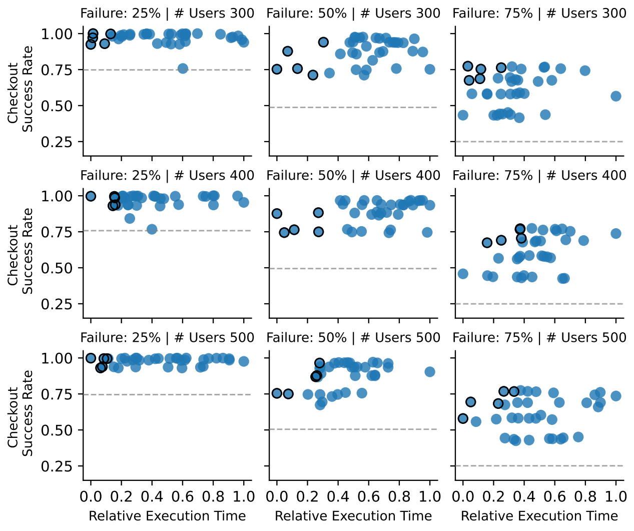

Figure 4: Relative execution time and checkout success rate for all 36 Retry configurations across all nine test scenarios. The five Retry configurations with the lowest relative execution times and the highest checkout success rates in each test scenario are highlighted with a dark edge.

These results highlight the complexity of selecting optimal resilience strategies manually and reinforce the importance of having a fully automated and flexible approach to support this exploratory process.

5 Related Work

The evaluation of resilience strategies in microservice architectures has attracted significant attention from both industry and academia. Various tools and frameworks have been developed to assess the robustness of microservices under fault conditions, each with its unique focus and methodology. In this section, we provide a brief overview of some of those solutions and discuss how they compare with our work.

Chaos Engineering Tools. Tools like Gremlin [14] and Chaos Toolkit [6] have been instrumental in introducing controlled failures to test system resilience, raising awareness for a new software reliability practice commonly referred to as *Chaos Engineering* [21]. Gremlin offers a suite of fault injection capabilities, allowing users to simulate CPU spikes, memory leaks, and network latencies to observe system behavior under stress. Chaos Toolkit provides a declarative approach to chaos experiments, enabling integration with continuous delivery pipelines. However, these tools primarily focus on inducing failures rather than systematically benchmarking resilience patterns like retries or circuit breakers.

Service Meshes and Fault Injection. Istio, a prominent service mesh for Kubernetes, incorporates fault injection features that allow developers to simulate delays and aborts in service communication [15]. This facilitates testing the resilience of microservices without modifying application code. While Istio's capabilities are powerful, they are primarily designed for traffic management and observability, lacking a comprehensive framework for benchmarking resilience strategies across diverse scenarios.

Cloud Provider Solutions. Some cloud providers have recently introduced native tools for resilience testing. AWS Fault Injection Service (FIS) [4] enables users to simulate faults within AWS environments, such as EC2 instance terminations or network disruptions, to test application resilience. While AWS FIS offers deep integration with AWS services, its applicability is limited to AWS-centric infrastructures and does not provide granular control over resilience pattern configurations.

Resilience Pattern Libraries. Libraries like Polly [19] for .NET and Resilience4j [20] for Java offer developers the means to implement resilience patterns such as retries, circuit breakers, and bulkheads within their applications. These libraries are invaluable for embedding resilience into application logic but do not provide mechanisms for benchmarking or evaluating the effectiveness of these patterns under varying fault conditions.

Resilience Frameworks. Resilience benchmarking frameworks such as ORCAS[23] and MicroRes[24] have addressed microservice resilience evaluation from complementary angles, but differ substantially from ResilienceBench-Operator in scope, methodology, and practical integration. ORCAS proposes an architectural approach to resilience benchmarking based on the interplay between resilience patterns, antipatterns, and fault injection strategies. It combines simulation-based and measurement-based techniques to reduce the number of necessary benchmark executions. Although ORCAS emphasizes efficiency and architectural awareness, its architecture is largely conceptual and simulation-centric. MicroRes is a versatile profiling framework that quantifies resilience by analyzing how performance degradation in system-level metrics propagates to user-facing metrics under injected faults. It introduces an innovative metric lattice and degradation indexing technique to assess resilience levels without the need for predefined rules. However, MicroRes was not designed to explore or evaluate the impact of different resilience pattern configurations, nor does it offer a declarative scenario specification approach or automation mechanism for benchmarking such configurations.

Runtime Pattern Adaptation. Recent work by Sedghpour et al. [22] addresses the limitations of static resilience pattern configurations by proposing an adaptive retry controller that dynamically adjusts retry configurations in conjunction with circuit breaker thresholds. The method is validated on two benchmark applications, including Online Boutique, and demonstrates significant throughput gains under transient overload and noisy neighbor conditions. Although that work shares our motivation to improve resilience through adaptive configuration, it focuses on run-time control strategies and parameter adaptation, rather than scenario generation and experimentation.

In contrast to existing solutions, ResilienceBench-Operator is the only tool that combines (i) Kubernetes-native integration through CRDs, (ii) declarative test space specification, (iii) automated scenario expansion, (iv) fault injection via Envoy, and (v) performance benchmarking using K6. Tools like Chaos Toolkit and Gremlin focus primarily on chaos injection and exploratory validation, with limited automation and no support for resilience pattern configuration. Istio offers fine-grained fault injection but lacks benchmarking and declarative experimentation capabilities. ORCAS and MicroRes provide valuable frameworks for resilience evaluation, but are not tightly integrated into Kubernetes workflows and do not offer automatic orchestration of real experiments. Compared to these tools, ResilienceBench-Operator fills an important gap by enabling reproducible and systematic resilience benchmarking directly within Kubernetes clusters, while maintaining the flexibility to evaluate different configuration trade-offs.

6 Conclusion

This paper presented ResilienceBench-Operator, a Kubernetes-native tool that enables automated benchmarking of resilience patterns in realistic microservice environments. Building upon the declarative test space model of the original ResilienceBench tool, the operator integrates seamlessly into Kubernetes workflows, allowing users to define and execute controlled resilience experiments using custom resource definitions and familiar YAML specifications. Our illustrative evaluation using the Online Boutique application demonstrated the operator's ability to uncover subtle trade-offs between execution time and reliability across diverse retry configurations, workloads, and fault conditions.

In future work, we plan to extend the evaluation to other microservice applications and resilience patterns (e.g., circuit breakers, timeouts, bulkheads). Although the tool already supports the use of those patterns, a comprehensive analysis requires addressing their distinct configuration spaces and performance trade-offs. We also plan to integrate multiobjective optimization and real-time analysis dashboards.

ARTIFACT AVAILABILITY

The source code for ResilienceBench-Operator is available at https://github.com/ppgia-unifor/resilience-bench-operator/. The dataset and configuration files used in the experimental study reported in this article are available at https://zenodo.org/records/16218841.

ACKNOWLEDGMENTS

Nabor C. Mendonça is partially supported by Brazil's National Council for Scientific and Technological Development (CNPq) under grant 313558/2023-0. The ChatGPT and Gemini generative AI tools were used to assist in structuring and revising this manuscript.

REFERENCES

[1] Carlos M Aderaldo, Thiago M Costa, Davi M Vasconcelos, Nabor C Mendonça, Javier Cámara, and David Garlan. 2025. A declarative approach and benchmark tool for controlled evaluation of microservice resiliency patterns. *Software: Practice and Experience* 55, 1 (2025), 170–192.

[2] Carlos Mendes Aderaldo and Nabor Das Chagas Mendonca. 2023. How The Retry Pattern Impacts Application Performance: A Controlled Experiment. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. 47–56.

[3] Carlos M. Aderaldo and Nabor C. Mendonça. 2022. ResilienceBench: Um Ambiente para Avaliação Experimental de Padrões de Resiliência para Microsserviços. In *Anais Estendidos do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos* (Fortaleza, CE). SBC, Porto Alegre, RS, Brasil, 65–72.

[4] Amazon Web Services. 2025. AWS Fault Injection Service: Improve resilience and performance with controlled experiments. https://aws.amazon.com/fis/.

[5] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. 2022. *Kubernetes: Up and Running: Dive into the Future of Infrastructure.* O'Reilly Media.

[6] ChaosToolkit. 2024. The chaos engineering toolkit for developers. https://chaostoolkit.org/.

[7] Thiago Costa, Davi Vasconcelos, Carlos Aderaldo, and Nabor Mendonça. 2022. Avaliação de Desempenho de Dois Padrões de Resiliência para Microsserviços: Retry e Circuit Breaker. In *Anais do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos* (Fortaleza, CE). SBC, Porto Alegre, RS, Brasil, 517–530.

[8] Jason Dobies and Joshua Wood. 2020. *Kubernetes Operators: Automating the Container Orchestration Platform.* O'Reilly Media.

[9] Envoy. 2025. Envoy Proxy. https://www.envoyproxy.io.

[10] Google Cloud Platform. 2024. Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo.

[11] Grafana Labs. 2025. Grafana k6: Load testing for engineering teams . https://k6.io/.

[12] HashiCorp. 2025. Vagrant: Development Environments Made Easy. https://www.vagrantup.com/.

[13] Sören Henning, Benedikt Wetzel, and Wilhelm Hasselbring. 2021. Reproducible Benchmarking of Cloud-Native Applications With the Kubernetes Operator Pattern. In *Proceedings of Symposium on Software Performance*. CEUR, Leipzig, Germany.

[14] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS).* 57–66.

[15] Istio.io. 2025. The Istio service mesh. https://istio.io/.

[16] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35.

[17] Shanshan Li et al. 2021. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology* 131 (2021), 106449.

[18] Nabor C. Mendonca, Carlos Mendes Aderaldo, Javier Cámara, and David Garlan. 2020. Model-based analysis of microservice resiliency patterns. In *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 114–124.

[19] Microsoft. 2022. Polly. https://github.com/App-vNext/Polly.

[20] Resilience4j. 2022. Resilience4j: A Fault tolerance library designed for functional programming. https://github.com/resilience4j/resilience4j.

[21] Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, and Ali Basiri. 2017. *Chaos Engineering: Building Confidence in System Behavior through Experiments.* O'Reilly.

[22] Mohammad Reza Saleh Sedghpour, David Garlan, Bradley Schmerl, Cristian Klein, and Johan Tordsson. 2023. Breaking the Vicious Circle: Self-Adaptive Microservice Circuit Breaking and Retry. In *2023 IEEE international conference on cloud engineering (IC2E)*. IEEE, 32–42.

[23] Andre Van Hoorn, Aldeida Aleti, Thomas F Düllmann, and Teerat Pitakrat. 2018. ORCAS: Efficient Resilience Benchmarking of Microservice Architectures. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 146–147.

[24] Tianyi Yang, Cheryl Lee, Jiacheng Shen, Yuxin Su, Cong Feng, Yongqiang Yang, and Michael R Lyu. 2024. MicroRes: Versatile Resilience Profiling in Microservices via Degradation Dissemination Indexing. In *33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 325–337.