Journal of Software Engineering Research and Development, 2026, 14:9, doi: 10.5753/jserd.2026.6965

ResilienceBench-Operator: A Kubernetes Extension for Systematic Evaluation of Microservice Resilience Patterns

Carlos M. Aderaldo [**Programa de Pós-Graduação em Informática Aplicada, Universidade de Fortaleza** | ***cmendesce@gmail.com***]
Nabor C. Mendonça [**Programa de Pós-Graduação em Informática Aplicada, Universidade de Fortaleza** | ***nabor@unifor.br***]

Abstract

Microservice-based applications commonly employ resilience mechanisms such as Retry and Circuit Breaker to mitigate failures in service-to-service communication, yet configuring these mechanisms involves complex trade-offs between reliability and latency. Systematically evaluating these trade-offs in cloud-native Kubernetes environments remains a challenge due to the lack of tools supporting controlled, reproducible resilience experimentation. This paper presents ResilienceBench-Operator, a Kubernetes-native extension of the original ResilienceBench framework that enables *in situ* benchmarking of resilience mechanisms directly on microservice applications deployed in Kubernetes. The operator preserves the declarative evaluation-space model of ResilienceBench while integrating scenario generation, environment reconfiguration, fault injection, workload execution, and metric collection into Kubernetes through Custom Resource Definitions, controllers, and jobs. The paper formalizes the underlying scenario-based evaluation model, details the design and implementation of the operator, and demonstrates its capabilities through a controlled benchmark experiment on the Online Boutique application, covering hundreds of automatically orchestrated evaluation scenarios across multiple workloads, failure levels, and retry configurations. The experimental results reveal nuanced performance–reliability trade-offs and show, through Pareto-frontier and correlation-based analyses, how retry parameters affect checkout success and tail latency under different downstream failure and workload conditions. The paper also discusses practical challenges encountered when conducting controlled experiments in Kubernetes and extracts lessons that can guide future benchmarking efforts in this area.

Keywords: Resilience Patterns, Microservices, Kubernetes Extension, Benchmarking

1 Introduction

Microservice-based applications are particularly susceptible to failures due to their distributed nature and reliance on remote communication (Jamshidi et al., 2018). Network delays, transient faults, hardware glitches, and overload conditions can compromise individual components and trigger cascading effects across the system (Li et al., 2021). To mitigate these risks, practitioners commonly adopt resilience mechanisms such as *Retry* and *Circuit Breaker*, which are widely discussed in microservice design and cloud reliability guidance (Jamshidi et al., 2018; Heorhiadi et al., 2016). Yet these mechanisms introduce non-trivial trade-offs: when misconfigured, they may increase tail latency, reduce throughput, or even amplify the impact of failures under stress (Aderaldo and Mendonça, 2023). Their true effect depends on the interplay among workload intensity, downstream failure rates, and pattern configuration.

Our prior work introduced *ResilienceBench*, a benchmark tool and declarative approach for controlled experimentation on resilience patterns (Aderaldo and Mendonça, 2022; Aderaldo et al., 2025). ResilienceBench compactly specifies an evaluation space—combining pattern configurations, workloads, fault types, and implementation libraries—and expands it into concrete scenarios executed sequentially under reproducible conditions. Subsequent experimental studies showed that such controlled evaluations can reveal practical insights about performance–reliability trade-offs across diverse operational settings (Costa et al., 2022; Aderaldo and Mendonça, 2023).

Despite these benefits, the original tool is constrained by its predefined single-client/single-service execution model, limiting its ability to capture realistic communication patterns, multi-hop dependencies, or infrastructure effects typically found in production-grade microservice deployments. As a result, findings obtained from ResilienceBench cannot be directly extrapolated to distributed topologies where multiple services interact concurrently and resilience mechanisms may exhibit non-linear, emergent behaviors.

To address this gap, we present *ResilienceBench-Operator*, a Kubernetes-native extension that integrates the scenario-based evaluation workflow directly into microservice deployments (Aderaldo and Mendonça, 2025). The operator builds upon familiar cloud-native constructs, including Custom Resource Definitions (CRDs), controllers, and jobs, to orchestrate *in situ* resilience experiments using YAML-based specifications. By embedding the experimental pipeline in Kubernetes, ResilienceBench-Operator enables systematic benchmarking of resilience strategies on realistic microservice applications with minimal disruption to existing deployments.

This paper is a revised and substantially extended version of our earlier publication on ResilienceBench-Operator (Aderaldo and Mendonça, 2025). Beyond the architectural and implementation aspects previously reported, this new version introduces several important additions:

- **A formalization of the scenario-based evaluation model** underlying ResilienceBench-Operator, defining core abstractions and providing an algorithmic treatment of scenario generation and execution.
- **A more in-depth description of the operator's cloud-native design and implementation**, including how its CRDs compose to instantiate large evaluation spaces directly in Kubernetes and the full technology stack used

during its development.

- **A controlled benchmark experiment**, applying the operator to a representative open-source microservice application and analyzing the results across workloads, fault levels, and retry configurations using Pareto-frontier analysis, parameter-level inspection of Pareto-optimal configurations, and correlation-based sensitivity analysis.
- **A new section on challenges and lessons learned**, discussing practical challenges encountered when conducting resilience experiments in production-like Kubernetes environments.
- **An expanded related work section**, situating ResilienceBench-Operator within the broader landscape of resilience benchmarking and fault injection literature and tools.

The remainder of this paper is organized as follows. Section 2 reviews the original ResilienceBench tool. Section 3 formalizes the scenario-based model that grounds our evaluation methodology. Section 4 describes the architecture and design of ResilienceBench-Operator. Section 5 presents a controlled benchmark experiment on a cloud-native microservice application and discusses its results and threats to validity. Section 6 discusses challenges and lessons learned. Section 7 surveys related tools and frameworks. Finally, Section 8 offers some concluding remarks and directions for future research.

2 The Original ResilienceBench Tool

ResilienceBench was developed as a benchmark tool and declarative approach to evaluate microservice resilience patterns in a controlled environment (Aderaldo and Mendonça, 2022). One of its main contributions was the introduction of the *evaluation space* abstraction—a high-level declarative specification that compactly encodes a wide range of evaluation conditions, including fault injection rates, workloads, pattern configurations, and resilience libraries (Aderaldo et al., 2025). The tool expands each evaluation space specification into concrete *evaluation 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 (Resilience4j, 2022) or C# with Polly (Microsoft, 2022).

At run-time, ResilienceBench comprises four main services, as depicted in Figure 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 evaluation 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 evaluation parameters (e.g., the target service's URL and failure rate, the client service's number of virtual users and resilience strategy configuration, etc.); (iii) collects

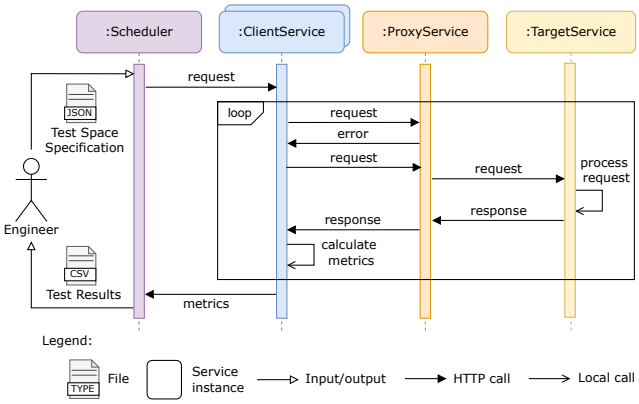


Figure 1. ResilienceBench's run-time architecture.

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 aborts one out of every four requests sent to the target service. Finally, the target service serves the client service's requests and is entirely oblivious of the other services.

ResilienceBench was intentionally designed for controlled experiments in which a benchmark-specific client service generates requests, applies the resilience strategy under study, and reports the resulting metrics. This design is well suited to isolating and comparing resilience patterns under controlled conditions. However, it also means that experiments are centered on a client service interacting with a single target service, rather than on an application exercised through its existing deployment topology.

When the goal is to evaluate resilience strategies in realistic microservice applications, this design creates a different set of requirements. The benchmark must exercise existing service interactions, preserve multi-service dependencies and failure propagation paths, and account for the behavior of the surrounding execution environment. In the original ResilienceBench design, supporting this setting would require reimplementing or wrapping the relevant interaction logic inside the benchmark client, thereby coupling the experiment to benchmark-specific application code. These limitations motivated the decoupled design of ResilienceBench-Operator, in which workload generation, fault injection, scenario orchestration, and metric collection are integrated with Kubernetes while remaining separate from the application logic. In this way, ResilienceBench-Operator preserves the controlled experimentation model of ResilienceBench while extending it to cloud-native deployments composed of heterogeneous, orchestrated services.

In summary, ResilienceBench established a solid foundation for the controlled evaluation of resilience patterns, offering valuable abstractions and experimental capabili-

ties. 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 and its formal underpinnings, described next.

3 A Declarative Scenario-Based Model for Resilience Evaluation

ResilienceBench-Operator builds on a declarative, scenario-based model that makes resilience experiments explicit and reproducible. Instead of hard-coding test logic into the application or the benchmarking tool, users describe an *evaluation space* that compactly captures the services under test, their communication paths, the workloads to be applied, and the fault injection conditions. This specification is then expanded into a set of concrete *scenarios*, each of which is executed in isolation to collect performance and reliability metrics. In this section, we formalize the core concepts of this model and the scenario generation process that underpins ResilienceBench-Operator.

3.1 Scenario Specification

In our model, a *scenario* represents a concrete resilience evaluation session executed against a *target application*. The application is composed of multiple services that communicate through explicit *connectors*. During scenario execution, a workload generator exercises the system while controlled faults are injected into selected services. The observed behavior depends on the configuration of resilience mechanisms applied along the communication paths.

Definition 1 (Target Application) A target application is a tuple $A = (S, C)$ where:

- $S = \{s_1, s_2, \ldots, s_n\}$ is the set of services composing the application; and
- $C = \{c_1, c_2, \ldots, c_m\}$ is the set of connectors defining communication among these services.

Each service encapsulates an independently deployable component of the application, implemented using some platform or technology stack.

Definition 2 (Service) A service is a tuple $s = (name, p)$ where:

- $name$ is a unique identifier for the service; and
- $p \in P$ is the development platform of the service, where P is the set of all development platforms.

Connectors represent concrete communication paths between services, annotated with the resilience configuration applied at the source.

Definition 3 (Connector) Let $A = (S, C)$ be a target application. A connector is a tuple $c = (name, src, dst, rs, conf)$ where:

- $name$ is a unique identifier for the connector;
- $src \in S$ is the source service;
- $dst \in S$ is the destination service;
- $rs \in RES(src.p)$ is a resilience strategy supported by the source service's platform, where $RES(p)$ denotes the set of resilience strategies available in platform p; and
- $conf \in CONF(rs)$ is a configuration of rs, where $CONF(rs)$ denotes the set of all possible configurations of resilience strategy rs.

Faults are injected according to a specification that defines the type, intensity, and targets of the failure.

Definition 4 (Fault Injection Specification) A fault injection specification is a tuple $fis = (type, f, targets)$ where:

- $type$ is the fault type (e.g., abort, delay, packet loss);
- f $(0 \leq f < 1)$ is the failure rate; and
- $targets \subseteq S$ is the set of services where faults will be injected.

Workloads define how traffic is supplied to the tool, applied to the target application, and bounded during scenario execution.

Definition 5 (Workload Specification) A workload specification is a tuple $w = (gen, criterion, params)$ where:

- gen identifies the workload driver supplied to the tool, such as a load-test script, and defines how requests are issued to the target application;
- $criterion$ defines the completion condition of the workload execution (e.g., number of requests, total duration, or convergence of a metric); and
- $params$ is a set of additional parameters controlling workload execution.

Given these building blocks, a scenario combines a specific communication path, a fault injection configuration, and a workload.

Definition 6 (Scenario) A scenario is a tuple $T = (C_T, fis, w)$ where:

- $C_T = \{c_1, c_2, \ldots, c_k\} \subseteq C$ is the set of connectors representing the communication paths under evaluation;
- fis is a fault injection specification; and
- w is a workload specification.

Manually enumerating all relevant scenarios quickly becomes infeasible as the number of services, connectors, fault levels, and workloads grows. To address this, the model introduces the notion of an *evaluation space*, which provides a compact, parameterized description of many scenarios at once.

Definition 7 (Evaluation Space) Let $A = (S, C)$ be a target application. An evaluation space is a tuple $\mathcal{E} = (C_{spec}, FIS, W)$ where:

- C_{spec} is a connector specification template that defines:

  - a set $P_C \subseteq S \times S$ of valid communication pairs reflecting the application's actual communication graph;
  - for each source service src such that $(src, dst) \in P_C$ for some $dst \in S$, a set of resilience strategies $RS(src) \subseteq RES(src.p)$;
  - for each resilience strategy rs, a set of configurations $CONFS(rs) \subseteq CONF(rs)$;
- $FIS = \{fis_1, fis_2, \ldots, fis_m\}$ is a set of fault injection specifications; and
- $W = \{w_1, w_2, \ldots, w_l\}$ is a set of workload specifications.

In practice, multiple evaluation spaces can be defined for the same application, in the context of a *Benchmark Suite*, allowing different subsets of connectors, fault models, and workloads to be explored.

Definition 8 (Benchmark Suite) A benchmark suite is a tuple $B = (A, W_{shared}, \{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n\})$ where:

- A is the target application under evaluation;
- W_{shared} is a set of workload specifications shared across all evaluation spaces; and
- $\{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n\}$ is a set of evaluation spaces, each representing a distinct experimental configuration for A.

3.2 Scenario Generation

Given an evaluation space $\mathcal{E} = (C_{spec}, FIS, W)$, the scenario generation process systematically enumerates all admissible combinations of connectors, fault specifications, and workloads. Conceptually, $\mathcal{E}$ is a compact description of a large Cartesian product of experimental parameters, and scenario generation corresponds to expanding this product into concrete scenarios.

Algorithm 1 provides a procedural definition of this expansion. The algorithm receives an evaluation space $\mathcal{E}$ as input and produces a set of evaluation scenarios $ES = \{Sc_1, Sc_2, \ldots, Sc_n\}$, where each scenario Sc_i is defined as in Definition 6. The outer loops iterate over all combinations of fault injection specifications and workloads, while the inner loops expand the connector specification template C_{spec} by varying valid source–destination pairs, resilience strategies, and their configurations. For each combination, the algorithm creates a new connector c and a corresponding scenario Sc.

The total number of scenarios generated from an evaluation space can be derived analytically.

Definition 9 (Evaluation Space Size) Let $\mathcal{E} = (C_{spec}, FIS, W)$ be an evaluation space. Let N_C denote the number of unique connector configurations derivable from C_{spec}, calculated as:

$$N_C = \sum_{(src,dst) \in P_C} \sum_{rs \in RS(src)} |CONFS(rs)|$$

Then, the size of $\mathcal{E}$, denoted Size($\mathcal{E}$), is given by:

$$\text{Size}(\mathcal{E}) = N_C \times |FIS| \times |W| \quad (1)$$

Algorithm 1: Scenario generation process

Input: evaluation space $\mathcal{E} = (C_{spec}, FIS, W)$
Output: set of evaluation scenarios ES such that $|ES| = \text{Size}(\mathcal{E})$

```
ES ← ∅;
foreach fis ∈ FIS do
    foreach w ∈ W do
        foreach (src, dst) ∈ P_C do
            foreach rs ∈ RS(src) do
                foreach conf ∈ CONFS(rs) do
                    c ← (name, src, dst, rs, conf);
                    Sc ← ({c}, fis, w);
                    ES ← ES ∪ {Sc};
                end
            end
        end
    end
end
return ES
```

Algorithm 1 enumerates only valid communication pairs in P_C, rather than independently combining arbitrary source and destination services. This ensures that generated scenarios correspond to connectors that exist in the target application's communication graph. Although Algorithm 1 describes a pure combinatorial expansion, practical implementations can incorporate filtering rules to discard invalid or redundant combinations (e.g., unsupported service pairs or incompatible fault targets). ResilienceBench-Operator implements this model directly in Kubernetes by mapping services, workloads, and benchmarks to Kubernetes resources and by using the controller logic to expand evaluation spaces into concrete scenarios, as described in the next section.

4 ResilienceBench-Operator

ResilienceBench-Operator instantiates the declarative scenario-based evaluation model presented in the previous section to support in-cluster experimentation on real microservice applications. It retains the core concepts of evaluation spaces, and scenario generation and execution introduced in the original ResilienceBench tool (Aderaldo et al., 2025), while adapting them to the constraints and opportunities of Kubernetes environments. The Operator decouples the benchmarking process from application code through Kubernetes Custom Resource Definitions (CRDs) (App Delivery Working Group, 2022; Dobies and Wood, 2020), leverages cloud-native tools for fault injection and load generation, and supports evaluation of complex service topologies with multiple connectors and realistic use cases.

The operator is intentionally agnostic to the specific resilience mechanism under evaluation. It does not implement *Retry*, *Circuit Breaker*, or other resilience mechanisms itself. Instead, it orchestrates experiments over mechanisms already implemented by the application, library, middleware, or infrastructure layer, provided that their relevant parameters can be externally configured between scenario executions. In the current implementation, this reconfiguration is performed by updating environment variables de-

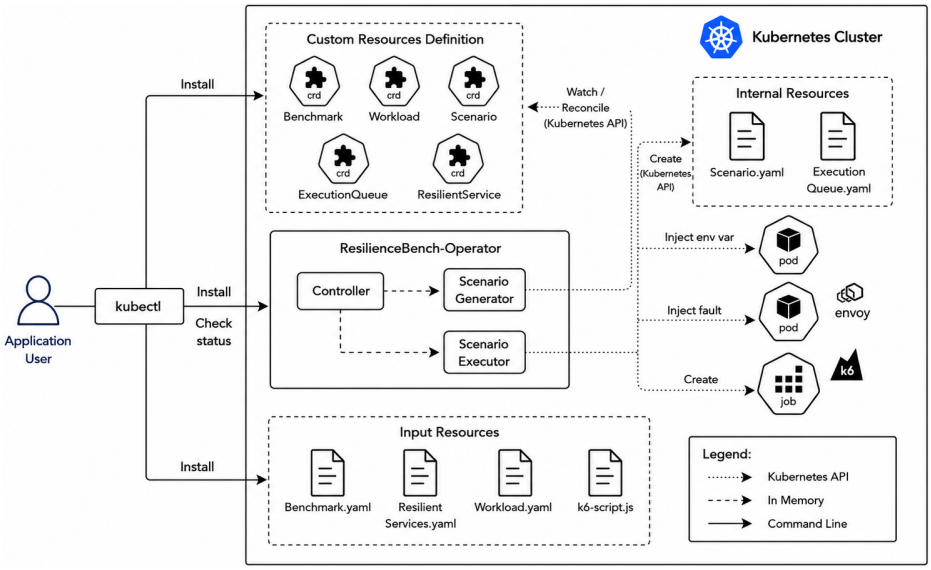


Figure 2. ResilienceBench-Operator's cloud-native architecture.

clared in the benchmark specification and applying them to the source service before each scenario execution. Consequently, application-level mechanisms such as *Retry* and *Circuit Breaker*, as well as service-mesh-level resilience policies, are within the conceptual scope of the tool whenever their behavior can be exposed through runtime configuration parameters. The experiment reported in Section 5 focuses specifically on application-level gRPC Retry.

4.1 Architecture Overview

Figure 2 presents an overview of the ResilienceBench-Operator architecture. At its core, the tool is implemented as a Kubernetes controller that continuously monitors changes to its set of defined custom resources. For example, when a new `Benchmark` resource is created by the tool user, the controller initiates the scenario generation pipeline, expanding the evaluation space into concrete scenarios. These scenarios are then sequentially executed, allowing controlled experimentation across combinations of fault injection levels and resilience configurations.

Internally, the operator is composed of three main components: (i) the Controller, responsible for managing the lifecycle of custom resources; (ii) the Scenario Generator, which computes all valid combinations of workload and configuration parameters; and (iii) the Scenario Executor, which coordinates the actual execution of each scenario and manages the necessary runtime components.

As illustrated in Figure 2, the controller watches and reconciles the custom resources through the Kubernetes API, following the standard Kubernetes operator pattern (Henning et al., 2021).

To simulate failure conditions and generate traffic, ResilienceBench-Operator integrates with two external tools: Envoy (Envoy, 2025) and k6 (Grafana Labs, 2025). Envoy is used as a sidecar proxy for services under test, enabling the injection of faults such as artificial latency and request aborts. It is the user's responsibility to configure the sidecar in the relevant deployments. k6 is responsible for executing the load test and collecting performance metrics throughout the scenario execution. The load script used by k6 is also provided by the user as an input file. These metrics can be stored locally or exported to an object storage bucket, depending on the configuration defined in the workload resource.

By leveraging native Kubernetes capabilities, ResilienceBench-Operator decouples scenario orchestration from application code and enables declarative, in-cluster evaluation of microservice resilience strategies under realistic and reproducible conditions. This decoupling, however, still assumes a minimal integration contract with the target application: the services involved in an experiment must be identifiable through Kubernetes selectors; the selected fault-injection mechanism must be deployable for the relevant services; the workload behavior must be encoded in a k6 script provided by the experimenter; and the resilience parameters under study must be externally configurable, for example through environment variables, configuration files, or service-mesh policies. Thus, the operator reduces, but does not eliminate, application-specific adaptation.

4.2 Scenario Specification

ResilienceBench-Operator uses multiple YAML-based resources that compose together to define an evaluation space. This modular design aligns with Kubernetes principles and enables progressive definition of experiments: services can be defined independently, workloads can be reused across multiple benchmarks, and complex scenarios can be built by combining simpler components.

The specification is organized around five primary Custom Resources (see Figure 2): `ResilientService`, `Workload`, `Benchmark`, and the automatically-generated `Scenario` and `ExecutionQueue` resources. To illustrate these resources, we use the Online Boutique (Google Cloud Platform, 2024) demo microservice application.

Online Boutique is an open-source system developed by

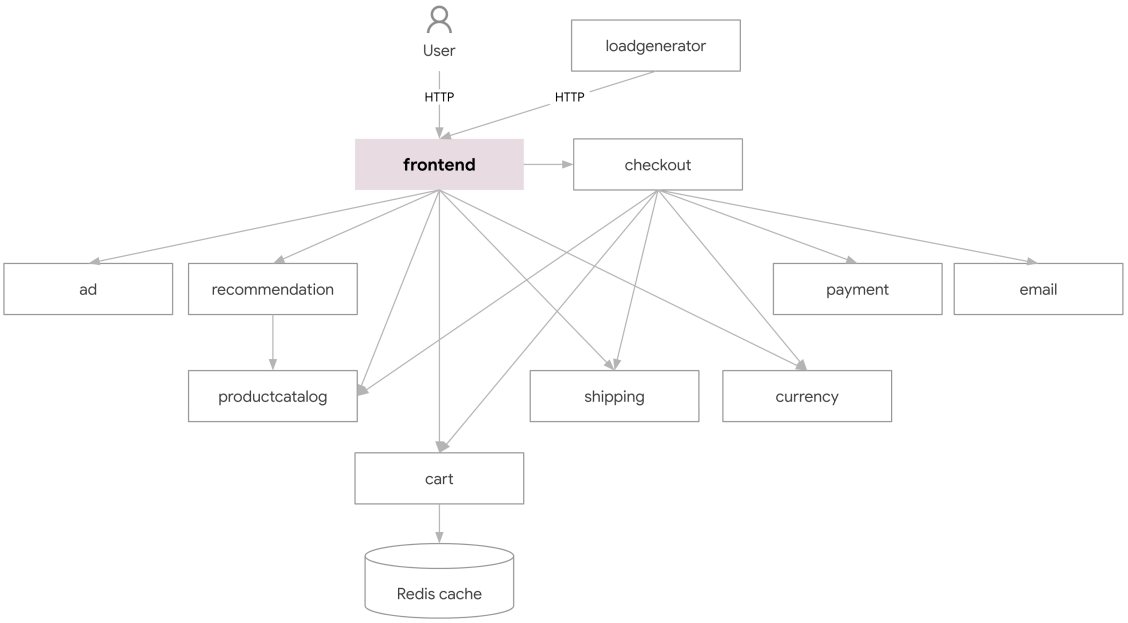


Figure 3. Online Boutique's service architecture

Google to simulate an e-commerce platform (Google Cloud Platform, 2024). As shown in Figure 3, the application comprises multiple services, such as *frontend*, *productcatalog*, *checkout*, *payment*, and *recommendation*, implemented in various programming languages (Go, Java, Node.js, Python, and C#). In this paper, we refer to these elements as services following the naming convention adopted by Online Boutique, while acknowledging that components such as *frontend* play a user-facing application role rather than representing a backend business microservice in the strict architectural sense. These services communicate primarily through gRPC. This diversity of services and implementation technologies, combined with an official Kubernetes deployment manifest, makes Online Boutique a useful testbed for demonstrating and validating Kubernetes-native resilience benchmarking workflows. At the same time, Online Boutique should not be interpreted as representative of the full diversity of industrial microservice architectures, since it does not include architectural elements such as API gateways or asynchronous messaging. Applications of ResilienceBench-Operator to larger-scale microservice systems, with richer service topologies and interaction patterns, will be reported in future work.

In the following subsections, we describe each of the primary ResilienceBench-Operator resources in detail using Online Boutique as a running example.

4.2.1 ResilientService

The `ResilientService` resource defines an application service that participates in the resilience experiment. Each `ResilientService` uses the `selector` field with Kubernetes Label Selectors to identify application pods associated with the service. The `resiliencebench.io/container` annotation specifies which container within the pod should be targeted by the operator—either the application container (e.g., "server") or a sidecar container (e.g., "envoy") where fault injection will be applied.

```yaml
apiVersion: resiliencebench.io/v1beta1
kind: ResilientService
metadata:
  name: checkoutservice
  annotations:
    resiliencebench.io/container: "server"
spec:
  selector:
    matchLabels:
      app: checkoutservice
---
apiVersion: resiliencebench.io/v1beta1
kind: ResilientService
metadata:
  name: paymentservice
  annotations:
    resiliencebench.io/container: "envoy"
spec:
  selector:
    matchLabels:
      app: paymentservice
```

Listing 1: ResilientService resource examples.

Listing 1 defines two services, `checkoutservice` and `paymentservice`, that represent the source and destination of the communication paths to be evaluated in the benchmark scenario.

4.2.2 Workload

The `Workload` resource defines how load is applied to the target application during scenario execution. It corresponds to multiple workload instances, each with a different workload level (e.g., number of virtual users). The `Workload` resource is referenced by the `Benchmark` resource and can be reused across multiple benchmarks, following the principle of composition established in the formal model. The operator does not synthesize the application workload automatically. Instead, users provide a workload driver, currently a k6 script, that encodes the user interactions or request sequences to be exercised during each scenario.

```yaml
apiVersion: resiliencebench.io/v1beta1
kind: Workload
metadata:
  name: fixed-iterations-loadtest
spec:
  options:
    - name: K6_ITERATIONS
      value: "3000"
  script:
    configMap:
      name: k6-config
      file: k6.js
  users: [300, 500]
```

Listing 2: Workload resource example.

Listing 2 shows an example of a `Workload` resource. This `Workload` resource includes:

- `options` (lines 6–8): Key-value pairs that configure the k6 load testing engine. These map directly to k6 runtime parameters, allowing fine-grained control over scenario behavior. The `K6_ITERATIONS` option specifies how many iterations each virtual user executes, directly influencing scenario duration and intensity.
- `script` (lines 9–12): References a JavaScript file (via Kubernetes ConfigMap) that contains the load generation logic. This script defines the actual requests to be issued and the endpoints to be tested, enabling realistic user case simulation beyond simple request generation. For a workload script to be usable by the operator, it must accept the runtime options supplied through the `Workload` resource, exercise the intended application behavior, and emit the metrics expected by the analysis pipeline. In the experiment reported in Section 5, the script executes complete e-commerce sessions and records both checkout success and session-duration metrics.
- `users` (line 13): An array of integers representing different workload (i.e., virtual user) values. In our example, there are two workload values (`[300, 500]`), each corresponding to a distinct level of access concurrency.

4.2.3 Benchmark

The `Benchmark` resource is the top-level specification that integrates workload configuration, services, fault injection strategy, and resilience pattern parameters into a cohesive evaluation space. It orchestrates the experiment by composing previously-defined `ResilientService` and `Workload` resources.

Listing 3 presents an example of a `Benchmark` resource that defines an evaluation space for the Online Boutique application. This `Benchmark` resource includes:

- `workload` (line 6): References a previously-defined `Workload` resource by name, establishing the shared workload configuration.
- `scenarios` (lines 7–28): Array of scenario specifications, each defining a distinct aspect of the evaluation space to explore. The `name` field (line 8) identifies the scenario group.

```yaml
apiVersion: resiliencebench.io/v1beta1
kind: Benchmark
metadata:
  name: onlineboutique
spec:
  workload: fixed-iterations-loadtest
  scenarios:
    - name: retry-checkout
      fault:
        provider: envoy
        percentages: [25, 50, 75]
        services:
         - paymentservice
      connectors:
        - name: checkout-payment
          source:
            name: checkoutservice
            envs:
            - name: GRPC_MAX_ATTEMPTS
              value: ["2", "3", "4", "5"]
            - name: GRPC_INITIAL_BACKOFF
              value: ["0.5s", "1s", "1.5s"]
            - name: GRPC_MAX_BACKOFF
              value: ["15s"]
            - name: GRPC_BACKOFF_MULTIPLIER
              value: ["1.0", "1.5", "2.0"]
          destination:
            name: paymentservice
```

Listing 3: Benchmark resource example.

- `fault` (lines 9–13): The fault injection specification. The `provider` field specifies the fault injection mechanism (Envoy for inter-service communication), `services` lists the target services where faults will be injected, and `percentages` is an array generating multiple fault injection rates. In our example, three fault injection rates are defined: `[25, 50, 75]`.
- `connectors` (lines 14–28): Defines the communication paths under evaluation. Each connector represents a directed communication from source to destination service:
  - `source` and `destination` (lines 16, 26): Identify the source and destination services (referenced from `ResilientService` resources)
  - `envs` (lines 18–26): Arrays of environment variables that configure the resilience strategy on the source service. In our example, there are four environment variables with 4, 3, 1, and 3 values, respectively, generating $4 \times 3 \times 1 \times 3 = 36$ unique connector configurations.

After scenario expansion, the specification in Listing 3 produces $36 \times 3 \times 2 = 216$ individual scenarios in total, as per Equation 1. The operator will execute these scenarios sequentially, grouping them first by fault injection rate, then by workload level, and finally by connector configuration.

4.2.4 Scenario

The `Scenario` resource captures a fully resolved experiment derived from the combinatorial expansion of a `Benchmark` specification. While `Benchmark` represents the entire evaluation space, each `Scenario` represents a single, concrete

configuration with specific values for virtual users, fault percentage, and environment variable parameters.

These resources are not created manually; the operator automatically generates them during scenario expansion (Algorithm 1). They serve two purposes: (i) validation of the exact configuration of each experiment case before execution and (ii) traceability of the precise parameters used, facilitating reproducibility and result interpretation.

4.2.5 ExecutionQueue

The `ExecutionQueue` resource orchestrates the sequential execution of generated scenarios. It acts as a scheduling mechanism that dequeues scenarios from the generated `Scenarios` and dispatches them for execution, ensuring that each scenario runs in isolation and preventing concurrent executions that could interfere with measurements.

This queue-based execution model aligns with Algorithm 1, which produces scenarios in a specific order. The `ExecutionQueue` preserves this order and maintains controlled experimental context, ensuring result reproducibility and measurement integrity.

4.3 Implementation Stack

The implementation of the ResilienceBench-Operator adheres to cloud-native principles, emphasizing modularity, automation, and seamless integration with Kubernetes environments. Its source code is publicly available on GitHub at `https://github.com/ppgia-unifor/resilience-bench-operator`, where users can find detailed documentation, including installation instructions and guidelines for local or cloud-based deployments. This open-source availability reinforces the extensibility of the tool, enabling other researchers and practitioners to evaluate, reproduce, and build upon our work.

The project is structured using Maven 3 as the build system, combined with Java 17 as the runtime environment, using the Java Operator SDK (version 4.8.2) (Red Hat, 2024b), which simplifies the creation of Kubernetes controllers by providing abstractions for reconciliation logic and resource handling.

The application logic is organized as a Spring Boot application, built upon version 3.0.6, which provides a robust and extensible programming model for developing cloud-ready services (Walls, 2022). The Spring ecosystem facilitates the integration of various components and offers out-of-the-box support for dependency injection, configuration management, and health monitoring.

To enable communication with the Kubernetes API server, the operator uses the Fabric8 Kubernetes Client (Red Hat, 2024a), which provides a fluent Java API for managing Kubernetes resources. This library supports advanced features such as server-side apply, custom resource management, and asynchronous event handling, which are critical for the development of responsive and declarative controllers.

To containerize the application, we use the Jib Java containerizer (Google, 2024), allowing container images to be built directly from the Maven project without requiring a Docker daemon. The use of GitHub Actions (GitHub, 2024) provides automation for continuous integration and deployment (CI/CD), orchestrating tasks such as running tests, building images, and pushing artifacts to registries based on changes in the version control system.

Testing is performed using JUnit and Mockito (Meszaros, 2007). JUnit offers a structured environment for defining and executing test cases, while Mockito supports the creation of mock objects, enabling the isolation of external dependencies and verification of application behavior through controlled test scenarios.

5 Controlled Benchmark Experiment

To illustrate the capabilities of ResilienceBench-Operator in a realistic cloud-native setting, we conducted a controlled benchmark experiment using an extended version of the Online Boutique microservice demo application, previously introduced in Section 4. We characterize this evaluation as a controlled benchmark experiment, rather than a case study or a usage example, because the objective is not to study Online Boutique in its natural organizational context, but to systematically exercise a controlled evaluation space over a fixed application, workload, fault-injection mechanism, and retry-configuration grid. The experiment demonstrates how the declarative evaluation space is instantiated as Kubernetes resources, how the operator orchestrates the resulting scenarios, and what types of resilience insights can be derived from the collected metrics.

5.1 Study Design

We extended the gRPC clients in the Online Boutique source code to support retry behavior. The original application contained no retry logic on inter-service calls; we added it through four environment variables that follow the gRPC retry-policy semantics: `GRPC_MAX_ATTEMPTS` (the total number of request attempts, including the original call), `GRPC_INITIAL_BACKOFF` (the delay, in seconds, before the first retry attempt), `GRPC_MAX_BACKOFF` (the maximum delay, in seconds, allowed between retries), and `GRPC_BACKOFF_MULTIPLIER` (the factor by which the wait time grows between successive retries). For simplicity, henceforth we refer to these variables as *Max Attempts*, *Initial Backoff*, *Max Backoff*, and *Backoff Multiplier*, respectively.

We implemented retries in the application-level gRPC client code, rather than through Envoy, because the experiment focuses on application-level retry policies exposed through source-service configuration parameters. In this setup, Envoy is used only as the fault-injection mechanism. Service-mesh-level retries are also relevant and could be evaluated with ResilienceBench-Operator, but they represent a different treatment because retry decisions would be configured and executed in the proxy layer rather than in the application client.

The extended version of the application is publicly available at `https://github.com/cmendesce/microservices-demo`.

Although broader benchmark suites such as DeathStarBench provide multiple applications and richer interaction patterns, we selected Online Boutique because it offers an official Kubernetes deployment, a complete e-commerce workflow, multiple services implemented in different languages, and gRPC-based inter-service communication. These characteristics make it suitable for a focused first controlled benchmark experiment of ResilienceBench-Operator. At the same time, the choice of a single demo application limits the generality of the empirical findings, especially for systems with API gateways, asynchronous messaging, or more complex service-mesh policies. We revisit this limitation in Section 5.3.

For this experiment, we focus on a single connector in the Online Boutique checkout workflow, namely `checkoutservice` → `paymentservice`, which encapsulates the interaction with the payment backend and is directly affected by injected faults. We selected this connector because it represents a payment-critical communication path in the checkout workflow, where downstream failures have a direct and visible impact on the user-facing outcome measured in the experiment. We did not randomly select the connector because the objective is not to estimate an average retry effect across arbitrary application connectors, but to demonstrate controlled exploration of a resilience-configuration space on a connector whose failures are strongly exercised by the workload and directly reflected in the checkout success metric. Random connector selection could choose a path weakly exercised by the workload or weakly reflected in the measured outcome, reducing interpretability. Extension to multiple connectors and connector-selection strategies is left as future work.

5.1.1 Research Questions

The experiments focus on evaluating the impact of gRPC retry policies applied to the `checkoutservice` → `paymentservice` connector in the application workflow. In particular, we investigate how different retry configurations affect the trade-off between resilience and performance when faults are injected into the downstream payment service.

The study is guided by the following research questions:

RQ1 How does enabling gRPC retries at a particular application connector affect system-level resilience and performance under different failure rates and workload levels, compared to a no-retry baseline?

RQ2 How do variations in retry parameters shape the performance–reliability trade-offs when retries are applied at the selected connector?

5.1.2 Benchmark Configuration

The experiment's evaluation space is specified using the `ResilientService`, `Workload`, and `Benchmark` custom resources introduced in Section 4.2. We reuse the `ResilientService` and `Workload` resources from Section 4, and define a single `Benchmark` resource that targets the `checkoutservice` → `paymentservice` connector.

The `Benchmark` resource configuration mirrors the example in Listing 3, with the resulting connector configuration grid comprising $4 \times 3 \times 1 \times 3 = 36$ retry configurations, as explained in Section 4.2. We consider two evaluation spaces:

- **Baseline (no retry).** A `Benchmark` specification without any connector-level retry configuration.
- **Checkout-only retry.** A `Benchmark` specification with the single `checkoutservice` → `paymentservice` connector configured as above.

For each evaluation space, ResilienceBench-Operator expands the evaluation space into concrete scenarios following the formal model from Section 3. Combining failure rates (3 levels), workloads (2 levels), and retry configurations (36 for the checkout-only case, 1 for the baseline), we obtain:

$$1 \times 3 \times 2 = 6 \quad \text{scenarios (baseline)}$$
$$36 \times 3 \times 2 = 216 \quad \text{scenarios (checkout-only)}$$

for a total of 222 scenarios automatically orchestrated by the operator.

Table 1 summarizes the main elements of the experimental design. The independent variables are the injected failure rate, workload level, and retry-configuration parameters. The dependent variables are checkout success rate and P95 session execution time. The baseline condition is the same workload and fault-injection setup without retries enabled.

5.1.3 Environmental Setup

All experiments were executed in a Kubernetes cluster running on a dedicated physical host (AMD Ryzen 9 3950X, 32GB RAM), to reduce performance noise common in shared cloud environments. The cluster comprised one control-plane node (2 vCPUs, 4GB RAM) and three worker nodes (4 vCPUs, 4GB RAM), all running Ubuntu 22.04 and Kubernetes 1.29. Cluster provisioning used Vagrant (HashiCorp, 2025). All our experiment configuration scripts and Kubernetes manifests are publicly available at `https://github.com/cmendesce/resilience-bench-operator-k8s-env`.

5.1.4 Workload Generation and Performance Metrics

Workload execution and metric collection were performed using a custom k6 script that simulates complete e-commerce sessions with Online Boutique, including the following operations: *open home page*, *browse product catalog*, *add product to cart*, *view cart*, *set currency*, and *checkout*. The workload is therefore not synthesized automatically by the operator; it is encoded as a script supplied by the experimenter and executed by k6 under the workload levels declared in the `Workload` resource. For each scenario, we customize the k6 script to collect the following two metrics:

- **Checkout success rate**: fraction of sessions that complete the checkout workflow successfully.
- **95th percentile session execution time**: end-to-end duration (from session start to checkout completion or failure), focusing on tail latency.

Table 1. Summary of the controlled benchmark experiment design.

Element	Description
Object of study	ResilienceBench-Operator orchestration capabilities
Target application	Extended Online Boutique application
Connector under evaluation	`checkoutservice` → `paymentservice`
Baseline	No retry configuration
Failure rates	25%, 50%, 75% request aborts
Workloads	300 and 500 virtual users
Independent variables	Retry parameters: *Max Attempts*, *Initial Backoff*, *Max Backoff*, *Backoff Multiplier*
Dependent variables	Checkout success rate and P95 session execution time
Experimental design	Full enumeration of the declared retry-configuration space

Table 2. Raw P95 session-time ranges used for min–max normalization.

Failure Rate	Users	Min P95 (s)	Max P95 (s)
25%	300	24.741	36.032
25%	500	40.583	57.050
50%	300	24.495	37.514
50%	500	38.664	56.949
75%	300	25.566	36.663
75%	500	39.216	53.816

To facilitate comparison across failure rates and workloads, session times were min–max normalized to the interval $[0, 1]$ within each scenario group. The normalized values are used as a relative performance indicator for comparing retry configurations under the same failure-rate/workload condition, rather than as an absolute characterization of end-user latency. Table 2 reports the raw P95 session-time ranges, in seconds, used to min–max normalize the execution-time values within each failure-rate/workload group.

5.1.5 Fault Injection Configuration

Fault injection is applied at the request level using Envoy sidecars associated with `paymentservice`. For each scenario, the configured failure rate (25%, 50%, or 75%) expresses the probability that a given request is aborted before reaching the service. These rates model distinct classes of degradation commonly observed in microservice deployments, such as near-100% failure windows caused by rate limiting exhaustion or quota depletion; and high but transient failure rates (50–75%) caused by the temporary loss of replicas in small deployments (2–3 replicas) until Kubernetes reschedules new pods.

Thus, although the failure rates might initially appear high, they reflect realistic conditions during incident windows rather than steady-state behavior.

5.1.6 Analysis Plan

To address **RQ1** and **RQ2**, we analyze the experimental results using three complementary techniques: Pareto frontier analysis, radar-chart analysis, and correlation-based sensitivity analysis.

Pareto Frontier Analysis For each combination of failure rate and workload, we compute the Pareto frontier across all 36 retry configurations at the `checkoutservice` → `paymentservice` connector. A configuration is Pareto-optimal if no other configuration achieves both a higher checkout success rate and a lower 95th percentile session time. Because the configuration space is moderate, we employ a brute-force enumeration of all parameter combinations and filter non-dominated points, ensuring exhaustive coverage without heuristic optimization.

The resulting Pareto frontiers provide a compact view of the performance–reliability trade-offs achievable by retry configurations at this connector, and allow us to quantify the improvement over the no-retry baseline. This analysis directly supports **RQ1**.

Radar-Chart Analysis To better understand how specific retry parameters contribute to Pareto-optimal behavior, we visualize the configurations that lie on each Pareto frontier using radar charts. Each polygon represents a distinct Pareto-optimal configuration, with axes corresponding to *Initial Backoff*, *Backoff Multiplier*, and *Max Attempts*. Max Backoff is not shown as a radar-chart axis because it is fixed at 15 seconds in all retry configurations. The geometry and clustering of these polygons reveal whether a small number of parameter profiles dominates (indicating stable and robust regions); and how sensitive the trade-off between success rate and latency is to parameter changes under different failure rates and workloads. This analysis addresses **RQ2** by characterizing how parameter choices shape the observed trade-offs.

Correlation-Based Sensitivity Analysis Finally, we compute Spearman rank correlations between each retry parameter and the two outcome metrics: checkout success rate and normalized P95 session time. The correlations are computed separately for each combination of failure rate and workload, rather than globally across the entire dataset. This stratification is necessary because retry parameters are expected to interact with operational conditions: the same parameter value may have different associations depending on the workload and failure intensity, and reliability gains may come at the cost of increased latency.

The baseline scenarios are not included in this correlation analysis because they do not represent retry-parameter configurations. In the baseline, retry-parameter values are undefined rather than zero-valued treatments. Including them would artificially conflate the absence of retries with particular retry-parameter settings. Therefore, the baseline is used only as a reference for assessing the improvement achieved by retry configurations, while the correlation analysis is restricted to the checkout-only retry scenarios. This analysis complements the radar charts by quantifying the direc-

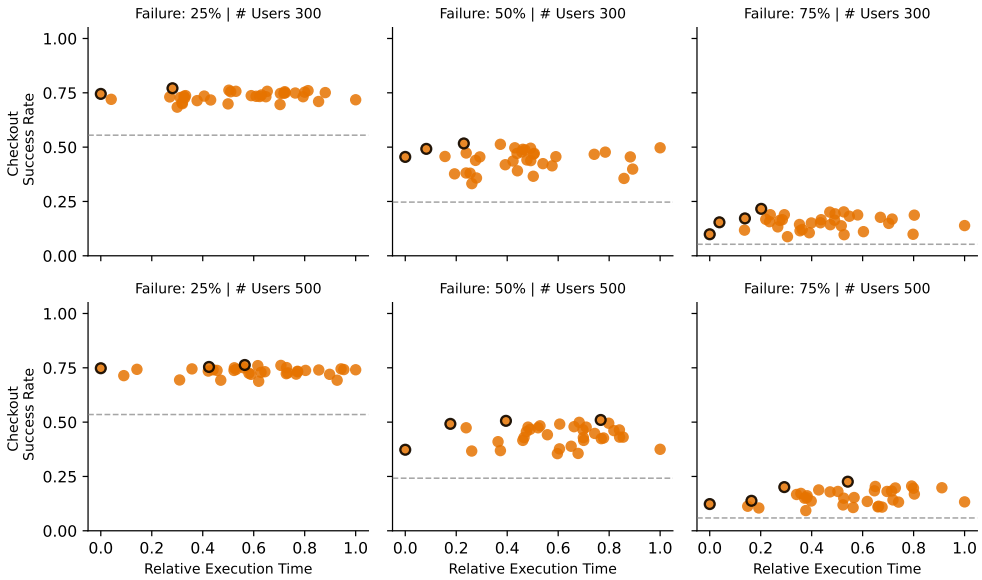


Figure 4. Pareto frontiers for retry policies at the `checkoutservice` → `paymentservice` connector.

tion and strength of association between each retry parameter and the observed outcomes within each operational scenario, thereby further supporting **RQ2**.

5.2 Results

5.2.1 R1: Impact on Resilience and Performance

Figure 4 depicts the results of the Pareto frontier analysis for the `checkoutservice` → `paymentservice` connector across the three failure rates and two workload levels. The colored circles correspond to retry configurations, and dark-edge circles denote the Pareto-optimal subset. The baseline (no retry) appears as a horizontal gray dashed line in each plot. Because it performs no retries, and therefore incurs no backoff delays, its execution time is always the minimum observed in each scenario and is normalized to zero. The baseline's success-rate value is fixed for each failure-rate/workload combination and serves as the reference point for all retry configurations. Together, the dashed baseline and the scatter of retry circles reveal the trade-off space for the connector, showing how different retry configurations progressively exchange higher latency for increased reliability.

From Figure 4, we observe that across all operational scenarios, enabling retries at the `checkoutservice` → `paymentservice` connector yields substantial improvements in resilience relative to the baseline. Under 25% failure rate, Pareto-optimal configurations reach success rates between approximately 0.70 and 0.75, compared to baseline values close to 0.55. At 50% failure rate, success rates in the Pareto set range between 0.40 and 0.50, whereas the baseline often remains below 0.25. Under the most severe condition (75% failure rate), Pareto-optimal configurations still achieve success rates in the interval 0.15–0.20, while the baseline approaches zero.

These results indicate that retries applied near the failure source (here, the payment service) can more than double or even triple the checkout success rate in high-failure scenarios, at the expense of moderate increases in session latency. The shape of the Pareto frontiers suggests that this trade-off is non-linear: small increases in latency can unlock large gains in reliability up to a certain point, after which additional retries yield diminishing returns.

The breadth of the frontiers also shows that, even for a fixed failure rate and workload, multiple configurations can be considered optimal depending on the relative importance of reliability versus latency. For example, at 50% failure rate, some Pareto-optimal configurations favor lower latency with modest success-rate gains over the baseline, while others accept larger latency increases in exchange for higher success rates. This diversity of options is precisely what ResilienceBench-Operator is designed to expose, enabling practitioners to choose configurations that align with their service level objectives.

Overall, these findings answer **RQ1**: *applying retries at the* `checkoutservice` → `paymentservice` *connector significantly improves resilience across all evaluated conditions, while preserving a controllable trade-off with performance.*

5.2.2 R2: Effect of Configuration Parameter Variations

Figure 5 presents the radar charts for the Pareto-optimal configurations at the `checkoutservice` → `paymentservice` connector. Each polygon corresponds to a distinct configuration on the Pareto frontier, with axes for *Initial Backoff*, *Backoff Multiplier*, and *Max Attempts*. The charts are grouped by failure rate and workload level. Table 3 complements these visualizations by listing the exact parameter values, success rates, and normalized execution times for each Pareto-optimal configuration.

A first observation from the radar charts is the distinct geometric patterns that emerge based on failure intensity. In scenarios with lower failure rates (25%), the Pareto-optimal configurations are highly consistent, characterized by high persistence. As shown in Table 3, these scenarios are dominated by the maximum number of attempts (5), com-

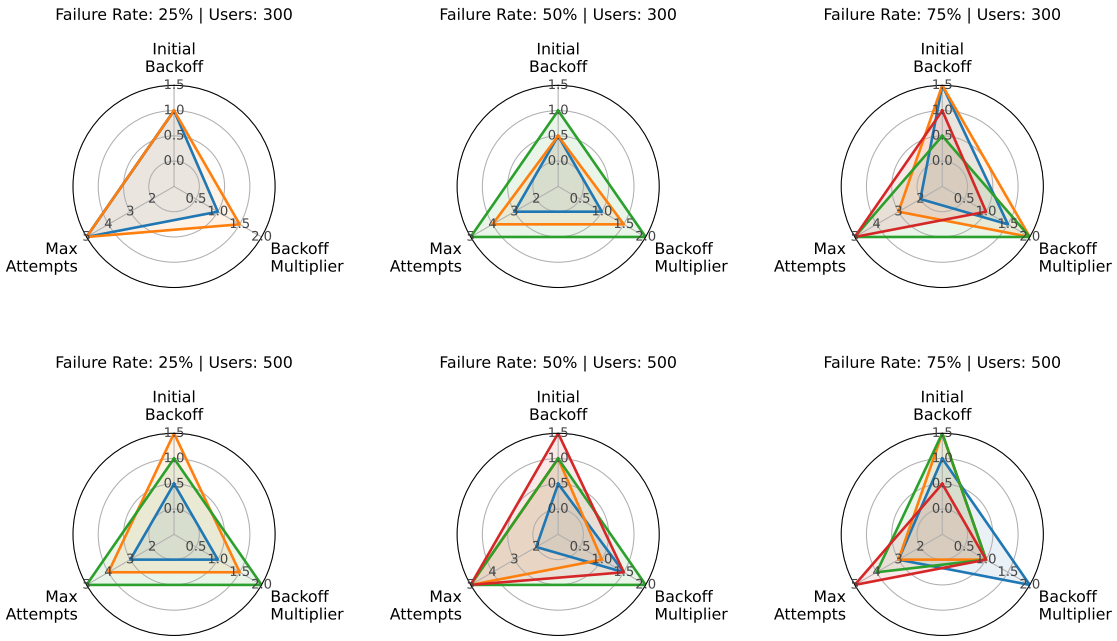


Figure 5. Radar charts representing the Pareto-optimal retry configurations at the `checkoutservice` → `paymentservice` connector.

Table 3. Parameter values for the Pareto-optimal retry configurations at the `checkoutservice` → `paymentservice` connector.

		Retry Configuration				
Fail. Rate	**Users**	**Init. Back.**	**Back. Mult.**	**Max Att.**	**Succ. Rate**	**Norm. Exec. Time**
25%	300	1.0	1.0	5	0.745	0.000
		1.0	1.5	5	0.771	0.281
50%	300	0.5	1.0	3	0.455	0.000
		0.5	1.5	4	0.492	0.083
		1.0	2.0	5	0.517	0.230
75%	300	1.5	1.5	2	0.099	0.000
		1.5	2.0	3	0.154	0.038
		0.5	2.0	5	0.216	0.202
		1.0	1.0	5	0.172	0.138
25%	500	0.5	1.0	3	0.748	0.000
		1.5	1.5	4	0.763	0.565
		1.0	2.0	5	0.754	0.425
50%	500	0.5	1.5	2	0.373	0.000
		1.0	1.0	5	0.492	0.177
		1.0	2.0	5	0.510	0.767
		1.5	1.5	5	0.506	0.395
75%	500	1.0	2.0	3	0.138	0.164
		1.5	1.0	3	0.123	0.000
		1.5	1.0	4	0.201	0.293
		0.5	1.0	5	0.226	0.542

bined with moderate initial backoffs (0.5–1.0 s) and low-to-moderate multipliers (1.0–1.5). This suggests that when failures are sporadic, the system can afford aggressive retries to maximize success rates without incurring prohibitive latency penalties.

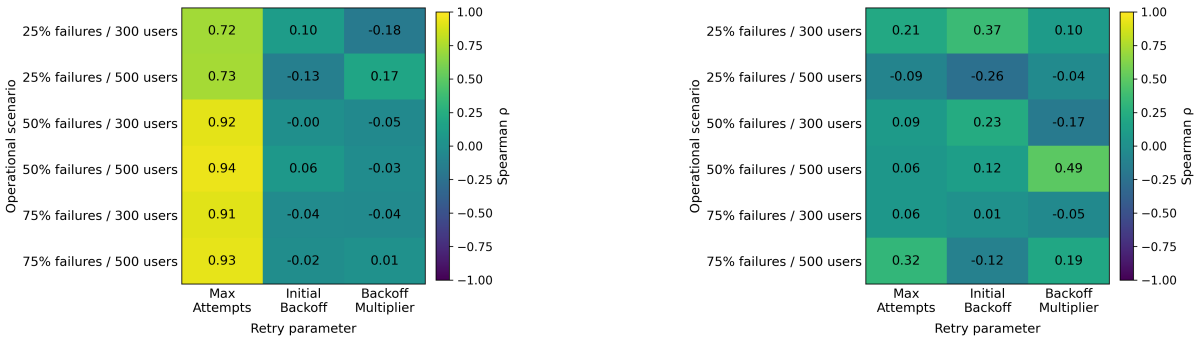
However, as the failure rate increases to 50% and 75%, the polygons in Figure 5 become more diverse, indicating a bifurcation in optimal strategies. The Pareto frontier expands to include two distinct types of configurations:

- **High Persistence/Low Delay:** Configurations that maintain 5 attempts but minimize wait times (e.g., 0.5 s initial backoff) to sustain throughput despite frequent errors.
- **Low Persistence/High Backoff:** Configurations that reduce the attempt count (2 or 3) but increase the initial backoff (up to 1.5 s) and multipliers (up to 2.0).

This shift is particularly visible in the 75% failure rate scenarios (right-most column of radar charts), where the shapes stretch towards the *Initial Backoff* and *Backoff Multiplier* axes while retreating on the *Max Attempts* axis. This behavior indicates that under severe fault injection, simply retrying more often (aggressive behavior) becomes counter-productive for latency. Instead, a "patience-based" strategy—waiting longer between fewer attempts—often yields a better trade-off between success rate and execution time.

Correlation Analysis To further characterize the relationship between retry parameters and the two outcome metrics, we complemented the Pareto-frontier and radar-chart analyses with a stratified Spearman correlation analysis. We computed correlations separately for each operational scenario, i.e., for each combination of failure rate and workload. This choice avoids collapsing distinct operational regimes into a single global coefficient, since retry parameters may interact with workload and failure intensity: a value that improves reliability in one scenario may have a weaker, different, or even opposite association in another.

Figure 6(a) shows the Spearman correlations between the three retry parameters and checkout success rate. The clearest result concerns *Max Attempts*, which exhibits a strong positive association with checkout success rate in all six operational scenarios. The correlation ranges from 0.72 under 25% failures / 300 users to 0.94 under 50% failures / 500 users. This result is expected, since increasing the maximum number of attempts raises the probability that at least one invocation to the payment service succeeds despite injected failures. However, the stability of this association also provides quantitative evidence that retry persistence is the dominant parameter for improving reliability in the evaluated connector.



(a) Checkout success rate.

(b) Normalized P95 session time.

Figure 6. Spearman rank correlations between retry parameters and outcome metrics, computed separately for each failure-rate/workload scenario.

In contrast, *Initial Backoff* and *Backoff Multiplier* show weak and scenario-dependent correlations with checkout success rate. Their correlations remain close to zero in most operational scenarios and occasionally change sign. This suggests that, in this experiment, these two parameters do not directly determine whether checkout completes successfully. Instead, they mostly affect how retries are spaced over time, which becomes more relevant when analyzing latency and the trade-off between reliability and performance.

Figure 6(b) presents the corresponding correlations with normalized P95 session time. Unlike the success-rate heatmap, the associations are weaker and less stable across scenarios. For example, *Max Attempts* has only a small positive correlation with normalized P95 session time in most cases, but the association becomes more visible under 75% failures / 500 users. Similarly, *Backoff Multiplier* has a moderate positive correlation under 50% failures / 500 users, but weak or negative correlations in other scenarios. *Initial Backoff* also varies in both magnitude and sign, indicating that its latency impact depends on the surrounding workload and failure conditions.

The correlation heatmaps reinforce the interpretation provided by the Pareto-optimal configurations. Increasing *Max Attempts* is consistently associated with higher checkout success rates, which explains why many Pareto-optimal configurations use high retry persistence. However, the latency effects of the retry parameters are weaker and more context-dependent. Therefore, retry tuning cannot be reduced to maximizing a single parameter. It requires balancing reliability gains against latency costs under each operational scenario.

These findings answer **RQ2**: *retry effectiveness at the* `checkoutservice` → `paymentservice` *connector is not governed by a single "stable" parameter region. Instead, it relies on a dynamic trade-off. At low failure rates, high persistence is generally beneficial for improving checkout success. Under high failure stress, however, Pareto-optimal configurations increasingly combine retry persistence with more conservative timing parameters, such as longer backoffs or lower attempt counts, to prevent latency degradation. The stratified correlation analysis reinforces this interpretation: while* Max Attempts *is consistently associated with higher success rates, the latency impact of all retry parameters remains scenario-dependent.*

5.3 Threats to Validity

This subsection discusses the main threats to the validity of the controlled benchmark experiment and the mitigation measures adopted where applicable. We organize the discussion around internal, construct, external, and conclusion validity.

Internal Validity. Internal validity concerns whether the observed effects can be attributed to the retry configurations rather than to uncontrolled factors in the experimental setup. A first threat is that Kubernetes runtime behavior may introduce transient interference between consecutive scenarios, for example due to rolling updates, delayed pod readiness, or residual resources from previous executions. This threat is particularly relevant because ResilienceBench-Operator evaluates scenarios by repeatedly reconfiguring services, applying fault-injection settings, and executing workload jobs in the same cluster. To mitigate this risk, scenarios were executed sequentially, and the operator was designed to coordinate configuration updates, workload execution, and resource cleanup before advancing to the next scenario. The lessons discussed in Section 6 further reflect the importance of treating rollout stabilization, readiness verification, and cleanup as part of the experimental protocol rather than as merely operational details.

A second internal validity threat concerns the modification of the Online Boutique source code to support gRPC retry policies. These modifications may have changed the behavior of the original application. We mitigated this threat by exposing retry behavior through environment variables and preserving a no-retry baseline, allowing retry-enabled scenarios to be compared against executions in which no retry configuration is applied. Nevertheless, the results should be interpreted as reflecting the behavior of the extended Online Boutique version used in this experiment, not necessarily the unmodified upstream application.

A third threat is related to the fault-injection mechanism. We inject abort faults at the request level using Envoy sidecars associated with `paymentservice`. This provides a controlled and reproducible failure model, but it abstracts away other types of failures observed in production systems, such as cascading failures, network partitions, bursty latency, resource exhaustion, or failures caused by dependencies outside the cluster. Therefore, the experiment supports conclusions about retry behavior under controlled request-abort

conditions, but not about all possible failure modes in cloud-native systems.

Construct Validity. Construct validity concerns whether the selected metrics and experimental constructs adequately capture the phenomena of interest. We use checkout success rate as the resilience metric because the selected connector lies on the checkout path and failures in `paymentservice` directly affect whether an e-commerce session completes successfully. This metric captures a user-facing reliability outcome, but it does not cover all dimensions of resilience, such as recovery time, throughput under degradation, resource consumption, or retry amplification effects on downstream services.

We use normalized P95 session execution time as the performance metric because the study focuses on tail-latency trade-offs among retry configurations under the same failure-rate/workload scenario. Normalization allows configurations to be compared within each operational scenario despite differences in absolute latency scales across workloads and failure rates. However, normalized values should not be interpreted as absolute end-user latency measurements or service-level objectives. They are relative indicators used to characterize the performance–reliability trade-off induced by retry-parameter choices.

Another construct validity threat concerns the terminology of resilience patterns and mechanisms. ResilienceBench-Operator does not implement resilience patterns itself; it orchestrates experiments over mechanisms already implemented by the application, library, middleware, or infrastructure layer. In this experiment, the treatment is application-level gRPC Retry configured through environment variables. Other mechanisms, such as *Circuit Breaker*, *Timeout*, *Bulkhead*, *Rate Limiting*, or service-mesh-level retries, may exhibit different trade-offs and require different configuration surfaces.

External Validity. External validity concerns the generalizability of the findings. The experiment uses Online Boutique, a widely used cloud-native demo application with an official Kubernetes deployment, multiple services, and gRPC-based inter-service communication. These characteristics make it suitable for a focused controlled benchmark experiment. However, Online Boutique is not an industrial production system and does not cover all architectural characteristics found in real microservice applications, such as API gateways, asynchronous messaging, complex service-mesh policies, or heterogeneous organizational constraints. Therefore, the empirical findings should not be generalized to all microservice architectures.

The experiment also focuses on a single connector, `checkoutservice` $\rightarrow$ `paymentservice`. This connector was selected because it is payment-critical, strongly exercised by the workload, and directly reflected in the checkout success metric. However, retry behavior may differ for other connectors, especially those with different call frequencies, different positions in the dependency graph, asynchronous interactions, or weaker influence on user-facing outcomes. Evaluating multiple connectors, randomly sampled connectors, or richer connector-selection strategies remains future work.

Finally, the experiment evaluates one retry mechanism, one application, one fault-injection model, and one Kubernetes-based execution environment. The results therefore support claims about the feasibility and usefulness of ResilienceBench-Operator for orchestrating controlled retry experiments in Kubernetes, but they should not be interpreted as universal guidelines for all retry implementations, all applications, or all cloud platforms.

Conclusion Validity. Conclusion validity concerns whether the analysis supports the conclusions drawn from the data. The experiment exhaustively enumerates the declared retry-configuration space for the selected connector and analyzes the resulting trade-offs using Pareto-frontier analysis, radar charts, and stratified Spearman correlations. These analyses are appropriate for characterizing the observed configuration space and identifying associations between retry parameters and outcome metrics.

However, the correlation analysis should be interpreted as descriptive rather than causal. We compute Spearman correlations separately for each failure-rate/workload scenario to avoid collapsing distinct operational regimes into a single global coefficient. This stratification reduces the risk of misleading aggregate correlations, but it does not establish that a parameter independently causes a given effect under all conditions. In particular, retry parameters interact with each other and with workload and failure intensity. Thus, the correlations complement the Pareto-frontier analysis by indicating the direction and strength of monotonic associations within each operational scenario, but the final interpretation is centered on multi-objective trade-offs rather than single-parameter optimization.

Another conclusion validity threat is that the analysis is limited to the finite configuration space declared in the benchmark specification. Other retry-parameter ranges, additional workload levels, or different fault intensities could reveal additional Pareto-optimal configurations. We therefore interpret the quantitative findings as evidence about the evaluated configuration space rather than as exact estimates of retry behavior under all possible configurations.

6 Challenges and Lessons Learned

Executing large-scale resilience experiments in Kubernetes clusters posed several practical challenges. Although Kubernetes provides powerful orchestration capabilities aimed at ensuring availability and automated recovery, controlled resilience experiments impose requirements, such as deterministic startup sequences, isolated pod restarts, and strict configuration synchronization, that fundamentally differ from Kubernetes' default behavior. This tension introduced a series of lessons learned that informed both the operator's architecture and its runtime mechanisms, and can guide practitioners conducting systematic benchmarking of distributed systems in Kubernetes environments.

Environment Reconfiguration and Pod Stabilization Each experiment requires modifying environment variables on target services to adjust retry parameters or fault-injection levels. Since Kubernetes treats environment variables as immutable, every modification triggers a Deployment rollout and restarts all pods. Initial experiments revealed that starting the next scenario before pods reached a stable `Ready` state introduced transient errors and inconsistent measurements. To ensure reproducibility, the operator implements an active readiness verification mechanism that blocks scenario execution until all pods complete their startup lifecycle.

Probe Configuration and Readiness Timing Default readiness and liveness probes often signaled availability too early, before services had completed warm-up behaviors such as dependency initialization. We observed that premature readiness frequently led to failures in early experimental stages. Adjusting probe timings proved essential to align Kubernetes' health checks with the synchronization guarantees required by controlled experiments.

Avoiding Mixed-Configuration Rollouts Kubernetes' default rolling updates aim to preserve service availability by overlapping old and new pods. While desirable in production, this overlap severely compromises experiment validity, as requests may hit pods running different configurations. To avoid configuration interference, we adopted a strict rollout policy that enforces sequential and fully isolated updates, at the cost of intentional downtime.

Job Lifecycle and Resource Cleanup Large experiments may create thousands of short-lived Jobs and Pods. Kubernetes retains completed Jobs indefinitely, and during early testing we observed accumulation of pods stuck in the *Terminating* state, degrading cluster responsiveness. Enabling automatic cleanup was essential to avoid unnecessary growth in API-server state and to maintain cluster stability over long executions.

Scenario Queue Management Managing thousands of scenarios requires robust sequencing. Instead of relying on external queues or databases, which would increase system complexity and deployment overhead, ResilienceBench-Operator leverages Kubernetes-native primitives. A custom `ExecutionQueue` CRD maintains the scenario list directly in etcd, and progress is driven through Kubernetes' event-based watch mechanism. This design ensures durability, transparency, and seamless integration with the operator pattern while avoiding the pitfalls of implementing separate orchestration infrastructure.

Implications for Kubernetes-Based Resilience Benchmarking These lessons suggest that Kubernetes-based resilience benchmarking requires more than deploying an application, injecting faults, and collecting metrics. The orchestration layer itself becomes part of the experimental method and may directly affect measurement validity. Mechanisms that are desirable in production, such as rolling updates, early readiness signaling, and automatic recovery, can introduce confounding effects in controlled experiments if they allow scenarios to overlap, start before stabilization, or leave residual resources in the cluster. Therefore, benchmark tools for cloud-native systems should treat configuration isolation, readiness verification, workload synchronization, and resource cleanup as first-class experimental concerns. This observation has implications beyond ResilienceBench-Operator: future resilience benchmarking frameworks should explicitly specify and enforce the operational protocol that separates one scenario from the next, making the execution environment auditable and reproducible rather than assuming that Kubernetes' default behavior is sufficient for empirical evaluation.

7 Related Work

The evaluation of resilience strategies in microservice architectures has attracted substantial attention from industry and academia. Prior research spans multiple complementary directions including fault injection, resilience mechanisms, resilience assessment and benchmarking, and dynamic resilience adaptation. This section reviews the most relevant tools, frameworks, and methodologies across these directions and positions our work relative to them.

7.1 Chaos Engineering and Fault Injection

Chaos engineering tools aim to uncover system vulnerabilities by injecting controlled failures into distributed systems. Early efforts such as Netflix's Chaos Monkey (Netflix, 2011) introduced random instance termination in production environments, laying the foundation for chaos engineering but focusing primarily on infrastructure-level faults and lacking modern observability and Kubernetes integration.

Modern chaos tools broaden the failure model. Chaos Toolkit (ChaosToolkit, 2024) adopts a platform-agnostic, declarative approach for defining experiments in JSON/YAML, offering extensibility through plugins. LitmusChaos (LitmusChaos Maintainers, 2020) provides native Kubernetes integration with predefined chaos experiments and CI/CD automation capabilities. Chaos Mesh (Chaos Mesh Community, 2025) goes further by introducing Kubernetes CRDs for fault injection covering CPU/memory stress, network partition, I/O delay, and kernel-level faults. Commercial platforms like Gremlin (Heorhiadi et al., 2016) offer operational safeguards and guided experiments, but proprietary abstractions limit extensibility.

These tools provide mature support for injecting infrastructure- and network-level failures, and some of them integrate naturally with Kubernetes. Their primary goal, however, is to expose weaknesses in running systems through controlled perturbations. ResilienceBench-Operator addresses a complementary problem: orchestrating controlled benchmark experiments that systematically vary resilience-configuration parameters, workloads, and fault levels, and then collect comparable performance and reliability metrics across the resulting scenarios.

7.2 Resilience Mechanisms

Resilience patterns such as *Retry*, *Circuit Breaker*, *Timeout*, and *Bulkhead* are widely supported by application-level libraries and frameworks. Polly (Microsoft, 2022) for .NET and Resilience4j (Resilience4j, 2022) for Java offer rich APIs for configuring the behavior of these resilience patterns. Java-based microservice frameworks such as Spring Cloud integrate resilience logic declaratively (VMware, 2025b,a), while the Eclipse MicroProfile Fault Tolerance specification (Eclipse Foundation, 2025) standardizes resilience annotations across Java ecosystems.

Beyond application frameworks, infrastructure-level enforcement is enabled by service meshes. Envoy (Envoy, 2025) supports *Retry*, *Circuit Breaker*, and *Timeout*, and fault injection at the proxy layer. Istio (Istio.io, 2025) builds upon Envoy with a powerful control plane providing declarative traffic management and observability. Linkerd (Linkerd.io, 2022) adopts a lightweight model offering automatic retries and latency-aware load balancing. Managed meshes such as AWS App Mesh (Amazon Web Services, 2025) and cloud-native tools like Azure Chaos Studio (Microsoft Azure, 2025) extend these capabilities to cloud-specific ecosystems.

These solutions are the mechanisms through which resilience behavior is implemented or enforced at runtime. ResilienceBench-Operator does not replace them. Instead, it provides an orchestration layer for evaluating how such mechanisms behave under controlled workload and failure conditions when their configuration parameters are varied. In the experiment reported in this paper, the evaluated mechanism is application-level gRPC Retry; however, the same orchestration model can be applied to other mechanisms whose parameters can be exposed through application, middleware, or infrastructure configuration.

7.3 Resilience Assessment and Benchmarking

Benchmarking resilience effectiveness remains an underexplored area relative to fault injection and resilience implementation. Among the few tools addressing systematic assessment, ORCAS (Van Hoorn et al., 2018) proposes a hybrid methodology combining simulation-based and measurement-based experimentation guided by architectural models extracted from tracing data. ORCAS aims to reduce the number of required benchmark executions via Bayesian inference but assumes predefined resilience patterns and does not support low-level pattern configuration or cloud-native orchestration of experiments.

MicroRes (Yang et al., 2024) introduces a model-based evaluation framework driven by a domain-specific language capturing architectural components, failure models, and resilience mechanisms. MicroRes automatically generates test cases to evaluate resilience levels but does not explore configuration spaces of Retry or Circuit Breaker patterns and lacks integration with Kubernetes-native execution pipelines.

Other tools such as Toxiproxy (Shopify, 2025) and Simmy (Polly-Contrib, 2020) enable application-centric fault injection but cannot orchestrate distributed experiments or benchmark system-level resilience strategies.

7.4 Adaptive Resilience Strategies

Recent research investigates dynamic adaptation of resilience parameters at runtime. Sedghpour et al. (Sedghpour et al., 2023) propose a feedback controller that adjusts retry and circuit breaker configurations based on observed latency and throughput. Their adaptive controller, also validated on Online Boutique, demonstrates notable gains under transient overload.

Other adaptive approaches leverage actor-based supervision trees (e.g., Akka (Lightbend, Inc., 2025)) or runtime sensitivity analysis techniques. While promising, these works optimize existing resilience configurations rather than providing methods to systematically evaluate the configuration space itself.

In contrast, ResilienceBench-Operator focuses on *offline, controlled, and exhaustive* benchmarking to reveal performance–reliability trade-offs, generating empirical evidence that can inform such adaptive strategies.

7.5 Summary and Positioning

Table 4 summarizes how ResilienceBench-Operator relates to representative tools and frameworks discussed in this section. The table is not intended to imply that all tools pursue the same objectives, nor that the absence of a feature constitutes a limitation in the context of each tool's original purpose. Rather, it highlights the constructs needed for the type of controlled resilience benchmarking targeted by our work. The columns indicate whether each tool or framework directly supports Kubernetes-native execution, explicit evaluation-space modeling, automatic expansion into concrete scenarios, systematic resilience-configuration variation, fault injection, workload execution, metric collection, and isolation between scenarios.

The comparison shows that ResilienceBench-Operator fills a specific gap by combining parameterized resilience-configuration spaces, automated scenario expansion, in-cluster workload execution, fault injection, metric collection, and scenario isolation in a single Kubernetes-native workflow. Its differentiator is not merely the use of a declarative language, but the integration of these constructs into a reproducible benchmarking process for cloud-native applications. This positioning complements existing chaos-engineering tools, resilience libraries, service meshes, assessment frameworks, and adaptive controllers, rather than replacing them.

8 Conclusion

This paper introduced ResilienceBench-Operator, a Kubernetes-native tool that automates the benchmarking of resilience mechanisms in microservice applications. By extending the declarative evaluation spaces of the original ResilienceBench framework, the operator embeds scenario generation, environment reconfiguration, workload execution, and metric collection directly within Kubernetes using CRDs, controllers, and jobs. This design enables systematic and reproducible in situ experimentation on cloud-native

Table 4. Comparison of ResilienceBench-Operator with related tools and frameworks.

Tool / Framework	K8s native	Eval. space	Scenario expansion	Config. variation	Fault injection	Workload execution	Metrics collection	Isolation control
Chaos Monkey (Netflix, 2011)	–	–	–	–	✓	–	–	–
Chaos Toolkit (ChaosToolkit, 2024)	○	○	–	–	✓	○	○	–
LitmusChaos (LitmusChaos Maintainers, 2020)	✓	○	–	–	✓	–	○	○
Chaos Mesh (Chaos Mesh Community, 2025)	✓	○	–	–	✓	–	○	○
Envoy / Istio (Envoy, 2025; Istio.io, 2025)	✓	–	–	○	✓	–	✓	–
ORCAS (Van Hoorn et al., 2018)	–	✓	○	○	✓	✓	✓	○
MicroRes (Yang et al., 2024)	–	✓	✓	○	✓	✓	✓	○
Original ResilienceBench (Aderaldo et al., 2025)	–	✓	✓	✓	✓	✓	✓	✓
ResilienceBench-Operator	✓	✓	✓	✓	✓	✓	✓	✓

Legend: ✓ explicitly supported; ○ partially supported or possible through integration/customization; – not a primary concern of the tool or framework

deployments while decoupling benchmark orchestration from the application logic under evaluation.

Our controlled benchmark experiment with the Online Boutique application demonstrates that ResilienceBench-Operator can reveal subtle performance–reliability trade-offs that are difficult to observe using traditional fault-injection or chaos engineering tools alone. The experiments highlight how retry configurations interact with workload levels and downstream failure rates, and illustrate the importance of carefully selecting both the placement and tuning of resilience mechanisms in microservice systems. The Pareto-frontier analysis exposed multiple non-dominated retry configurations for each operational scenario, while the stratified Spearman correlation analysis showed that *Max Attempts* is consistently associated with higher checkout success rates, whereas the latency impact of all retry parameters remains scenario-dependent.

The work also uncovered several practical challenges associated with large-scale experimentation in Kubernetes, including deterministic pod restarts, readiness synchronization, rollout control, and the management of thousands of short-lived jobs. The solutions incorporated into the operator (e.g., readiness gating, strict rollout strategies, Kubernetes-native scenario queues) represent lessons learned that can benefit the broader community conducting controlled experiments in cloud-native environments. More broadly, these findings indicate that Kubernetes' orchestration behavior is not merely part of the execution infrastructure, but also a potential source of experimental interference that must be explicitly controlled in resilience benchmarking studies.

Future extensions to ResilienceBench-Operator include expanding support for additional resilience mechanisms such as *Circuit Breakers*, *Timeouts*, and *Bulkheads*; integrating statistical analysis and optimization techniques to navigate large configuration spaces more efficiently; and supporting adaptive, runtime tuning of resilience policies through Kubernetes-native feedback loops. Applying the tool to larger industrial applications and diverse technology stacks, including richer benchmark suites such as DeathStarBench (Gan et al., 2019), also constitutes an important direction for validating its generality and practical impact.

Data Availability

The ResilienceBench-Operator source code and the research material needed to reproduce this study are publicly available at `https://zenodo.org/records/20784998`. The archive contains a replication package documenting the operator implementation, configuration scripts, Kubernetes manifests, modified application artifacts, experimental datasets, and analysis material used in the study.

Acknowledgements

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

References

Aderaldo, C. M., Costa, T. M., Vasconcelos, D. M., Mendonça, N. C., Cámara, J., and Garlan, D. (2025). A declarative approach and benchmark tool for controlled evaluation of microservice resiliency patterns. *Software: Practice and Experience*, 55(1):170–192.

Aderaldo, C. M. and Mendonça, N. C. (2025). ResilienceBench-Operator: A Kubernetes Extension for Orchestrating Resilience Experiments on Microservice Applications. In *Proceedings of the 39th Brazilian Symposium on Software Engineering (SBES)*, pages 983–989. SBC.

Aderaldo, C. M. and Mendonça, N. C. (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*, pages 65–72, Porto Alegre, RS, Brasil. SBC.

Aderaldo, C. M. and Mendonça, N. C. (2023). How The Retry Pattern Impacts Application Performance: A Controlled Experiment. In *Proceedings of the 37th Brazilian Symposium on Software Engineering (SBES)*, pages 47–56. SBC.

Amazon Web Services (2025). AWS App Mesh Documentation. `https://docs.aws.amazon.com/app-mesh`. Last accessed: July 17, 2026.

App Delivery Working Group (2022). Cncf operator white paper – final version. `https://github.com/cncf/tag-app-delivery/blob/163962c4b1cd70d085107fc579e3e04c2e14d59c/operator-wg/whitepaper/Operator-WhitePaper_v1-0.md`. Last accessed: July 17, 2026.

Chaos Mesh Community (2025). Chaos Mesh: A Chaos Engineering Platform for Kubernetes. `https://chaos-mesh.org`. Last accessed: July 17, 2026.

ChaosToolkit (2024). The chaos engineering toolkit for developers. `https://chaostoolkit.org/`.

Costa, T., Vasconcelos, D., Aderaldo, C., and Mendonça, N. (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*, pages 517–530, Porto Alegre, RS, Brasil. SBC.

Dobies, J. and Wood, J. (2020). *Kubernetes Operators: Automating the Container Orchestration Platform*. O'Reilly Media.

Eclipse Foundation (2025). Eclipse MicroProfile Fault Tolerance Specification. `https://microprofile.io/specifications/microprofile-fault-tolerance/`. Last accessed: July 17, 2026.

Envoy (2025). Envoy Proxy. `https://www.envoyproxy.io`.

Gan, Y. et al. (2019). An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18. ACM.

GitHub (2024). Github actions. `https://github.com/features/actions`. Last accessed: July 17, 2026.

Google (2024). Jib: Containerize your Java application. `https://github.com/GoogleContainerTools/jib`. Last accessed: July 17, 2026.

Google Cloud Platform (2024). Online Boutique. `https://github.com/GoogleCloudPlatform/microservices-demo`.

Grafana Labs (2025). Grafana k6: Load testing for engineering teams. `https://k6.io/`.

HashiCorp (2025). Vagrant: Development Environments Made Easy. `https://www.vagrantup.com/`.

Henning, S., Wetzel, B., and Hasselbring, W. (2021). Reproducible Benchmarking of Cloud-Native Applications With the Kubernetes Operator Pattern. In *Proceedings of Symposium on Software Performance*, Leipzig, Germany. CEUR.

Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K., and Sekar, V. (2016). Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66.

Istio.io (2025). The Istio service mesh. `https://istio.io/`.

Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.

Li, S. et al. (2021). Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and Software Technology*, 131:106449.

Lightbend, Inc. (2025). Welcome to Akka. `https://doc.akka.io`. Last accessed: July 17, 2026.

Linkerd.io (2022). Linkerd: Ultralight service mesh for Kubernetes and beyond. `https://linkerd.io/`.

LitmusChaos Maintainers (2020). LitmusChaos: Cloud-Native Chaos Engineering Framework. `https://litmuschaos.io`. Last accessed: July 17, 2026.

Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

Microsoft (2022). Polly. `https://github.com/App-vNext/Polly`.

Microsoft Azure (2025). Azure Chaos Studio. `https://learn.microsoft.com/en-us/azure/chaos-studio/`. Last accessed: July 17, 2026.

Netflix (2011). Chaos Monkey. `https://github.com/Netflix/chaosmonkey`. Last accessed: July 17, 2026.

Polly-Contrib (2020). Simmy: Chaos Engineering and Fault Injection for .NET. `https://github.com/Polly-Contrib/Simmy`. Last accessed: July 17, 2026.

Red Hat (2024a). Fabric8 Kubernetes Client. `https://github.com/fabric8io/kubernetes-client`. Last accessed: July 17, 2026.

Red Hat (2024b). Java operator sdk. `https://javaoperatorsdk.io`. Last accessed: July 17, 2026.

Resilience4j (2022). Resilience4j: A Fault tolerance library designed for functional programming. `https://github.com/resilience4j/resilience4j`.

Sedghpour, M. R. S., Garlan, D., Schmerl, B., Klein, C., and Tordsson, J. (2023). Breaking the Vicious Circle: Self-Adaptive Microservice Circuit Breaking and Retry. In *2023 IEEE international conference on cloud engineering (IC2E)*, pages 32–42. IEEE.

Shopify (2025). Toxiproxy. `https://github.com/Shopify/toxiproxy`. Last accessed: July 17, 2026.

Van Hoorn, A., Aleti, A., Düllmann, T. F., and Pitakrat, T. (2018). ORCAS: Efficient Resilience Benchmarking of Microservice Architectures. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 146–147. IEEE.

VMware (2025a). Spring Cloud Circuit Breaker. `https://spring.io/projects/spring-cloud-circuitbreaker`. Last accessed: July 17, 2026.

VMware (2025b). Spring Cloud Documentation. `https://spring.io/projects/spring-cloud`. Last accessed: July 17, 2026.

Walls, C. (2022). *Spring in Action*. Manning Publications, 6 edition.

Yang, T., Lee, C., Shen, J., Su, Y., Feng, C., Yang, Y., and Lyu, M. R. (2024). MicroRes: Versatile Resilience Profiling in Microservices via Degradation Dissemination Indexing. In *33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 325–337.