Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

A Declarative Approach and Benchmark Tool for Controlled Evaluation of Microservice Resiliency Patterns

Carlos M. Aderaldo[1] | **Thiago M. Costa**[1] | **Davi M. Vasconcelos**[1] | **Nabor C. Mendonça**[1] | **Javier Cámara**[2] | **David Garlan**[3]

[1]Center for Technological Sciences, University of Fortaleza, Ceará, Brazil

[2]ITIS Software, University of Málaga, Málaga, Spain

[3]Institute for Software Research, Carnegie Mellon University, Pittsburgh, USA

Correspondence
Corresponding author Nabor C. Mendonça Post Graduate Program in Applied Informatics, University of Fortaleza, Av. Washington Soares, 1321, Edson Queiroz, 60811-905 Fortaleza, Ceará, Brazil.
Email: nabor@unifor.br

Abstract

Microservice developers increasingly use resiliency patterns such as Retry and Circuit Breaker to cope with remote services that are likely to fail. However, there is still little research on how the invocation delays typically introduced by those resiliency patterns may impact application performance under varying workloads and failure scenarios. This paper presents a novel approach and benchmark tool for experimentally evaluating the performance impact of existing resiliency patterns in a controlled setting. The main novelty of this approach resides in the ability to declaratively specify and automatically generate multiple testing scenarios involving different resiliency patterns, which one can implement using any programming language and resilience library. The paper illustrates the benefits of the proposed approach and tool by reporting on an experimental study of the performance impact of the Retry and Circuit Breaker resiliency patterns in two mainstream programming languages (C# and Java) using two popular resilience libraries (Polly and Resilience4j), under multiple service workloads and failure rates. Our results show that, under low to moderate failure rates, both resiliency patterns effectively reduce the load over the application's target service with barely any impact on the application's performance. However, as the failure rate increases, both patterns significantly degrade the application's performance, with their effect varying depending on the service's workload and the patterns' programming language and resilience library.

KEYWORDS
Microservices, Resiliency patterns, Benchmarking

1 | INTRODUCTION

Microservice-based applications are typically fragile like many distributed systems. Multiple types of failures, such as network delays, hardware defects, or server overloads, might render any microservice temporarily inaccessible to its clients.[1] Anticipating and dealing with different types of failures is part of a fundamental design paradigm commonly referred to as *design for failure*,[2] which is one of the tenets of the microservice architectural style.[3] To mitigate the impact of partial service outages, microservice-based application developers must build resilient services that can gracefully respond to failures.[2] One common way to do this is by implementing service-to-service interactions using well-known *resiliency patterns*, such as Retry, Circuit Breaker, and Fail Fast.[4,5]

In the context of microservices, resilience refers to the system's ability to maintain an acceptable level of service in the face of faults and failures. This involves not just handling faults but also ensuring that failures do not propagate, thereby preventing cascading failures throughout the system.[6] Effective resilience strategies, therefore, must be able to anticipate and mitigate the impact of service outages or degradation on the overall application. Resiliency patterns, such as Retry and Circuit Breaker, play a key role in achieving this resilience.[5] The Retry pattern allows a microservice to attempt failed operations multiple times, accommodating temporary network glitches or service unavailability.[7] The Circuit Breaker pattern, on the other hand, prevents a microservice from repeatedly calling a failing service, thus avoiding unnecessary load and allowing the failing service time to

recover.[8] These patterns, when properly implemented and configured, help maintain the overall health and performance of a microservices-based system, even when individual services experience issues.[9]

Despite the popularity of resiliency patterns amongst practitioners, thus far, there has been relatively little interest from the research community in studying how their use, particularly the invocation delays they typically introduce under failure, may affect application performance. A few notable exceptions are the works of Mendonça *et al.*,[9] Jagadeesan and Mendiratta,[10] and Sedghpour *et al.*[11] Mendonça *et al.* and Jagadeesan and Mendiratta have formally modeled and analyzed the behavior of the Retry and Circuit Breaker patterns and their expected impact on application performance using the PRISM probabilistic model-checker.[12] Sedghpour *et al.*,[11] in turn, have evaluated the performance impact of these two patterns in the context of the Istio service mesh middleware.[13] However, none of these works have experimentally evaluated the performance impact of resiliency patterns implemented using popular resilience libraries, e.g., Java's Resilience4j[14] and C#'s Polly.[15] Evaluating the performance impact of existing resiliency pattern implementations is critical because the back-off mechanism used by those resiliency patterns to avoid overloading an unresponsive service may have the undesirable side effect of significantly degrading the application's performance.[9]

A systematic evaluation of the resiliency patterns solutions provided by existing resilience libraries requires planning and executing many performance tests to account for all possible ways application developers can configure those patterns. In addition, the evaluation must consider other external factors that may affect the patterns' impact on the application's performance, such as the target service's workload and failure rate.[9] In that regard, we are unaware of any current work that has focused on *facilitating the specification and execution of multiple performance tests involving resiliency patterns implemented using different programming languages and resilience libraries under varying workloads and failure conditions*.

This paper presents a novel approach to support the experimental evaluation of resiliency patterns. The main novelty of this approach resides in the ability to declaratively specify and automatically generate multiple testing scenarios involving multiple resiliency pattern libraries, configurations, workloads, and failure rates. These parameters are crucial for a comprehensive testing approach: resiliency pattern libraries provide multiple implementations of several resiliency patterns in different programming languages; configurations allow for adjusting pattern behavior; workloads simulate various levels of system stress; and failure rates represent different scenarios of service unreliability. By varying these test parameters, we can assess the performance impact of typical microservice resiliency patterns across a wide range of realistic operational scenarios.

The declarative nature of our approach is instrumental in facilitating the task of specifying a potentially large variety of test scenarios in a compact, high-level manner. We have embodied our approach into an open-source benchmark tool, called ResilienceBench,[16] which automatically executes a given test scenario specification in a controlled containerized environment. ResilienceBench also collects and consolidates several performance metrics during scenario execution and reports them after completing the evaluation. Beyond our declarative approach and support tool, this paper further contributes with an experimental study of the performance impact of the Retry and Circuit Breaker resiliency patterns in two mainstream programming languages (C# and Java) using two popular resilience libraries (Polly and Resilience4j). The study used ResilienceBench to evaluate the performance impact of different configurations of Retry and Circuit Breaker across multiple workloads and failure rates. To conduct the study, we implemented a simple client application to continuously invoke a target HTTP service using each resiliency pattern implementation until it reached a certain number of successful invocations. Overall, our results show that, under low to moderate failure rates, both resiliency patterns can effectively reduce the load over the HTTP service with barely any impact on the application's execution time. However, as the failure rate increases, both patterns can significantly degrade the application's performance, with their effect varying depending on the application's programming language/resilience library.

While our approach supports the evaluation of multiple resiliency patterns, this paper focuses primarily on the Retry and Circuit Breaker patterns. We chose these patterns for several reasons:

Ubiquity Retry and Circuit Breaker are among the most commonly used resiliency patterns in microservice architectures.[17,18,19,20]

Complementary nature These patterns often work in tandem, with Retry attempting to recover from transient failures and Circuit Breaker preventing cascading failures during prolonged outages.

Performance impact Both patterns can significantly affect system performance, making their evaluation crucial for optimizing microservice applications.

Configurability Retry and Circuit Breaker offer various configuration options, allowing us to demonstrate the flexibility of our approach in handling diverse pattern behaviors.

By conducting an in-depth evaluation of these two patterns, we aim to provide insights that can be generalized to other resiliency patterns. The lessons learned from this study—such as the impact of different configurations on performance, the interplay between patterns and failure rates, and the variations across programming languages and libraries—can guide developers in implementing and configuring other resiliency patterns effectively. Moreover, our approach and tool can be readily applied to evaluate additional patterns, enabling researchers and practitioners to extend this work to other aspects of microservice resilience.

In summary, this paper makes three significant contributions. First, it introduces a novel declarative approach for experimentally evaluating resiliency patterns, implemented using any programming language and resilience library, in a controlled setting. Second, it presents an open-source benchmark tool, ResilienceBench, which has been developed as a proof-of-concept of the proposed approach. Finally, it offers a practical demonstration of the proposed approach and benchmark tool for the systematic evaluation of the Retry and Circuit Breaker patterns in both C# and Java, using the Polly and Resilience4j libraries, under multiple environmental conditions. ResilienceBench is already being applied to investigate how to best configure the Retry pattern under a wider range of operating scenarios and pattern configurations.[21] By making our benchmark tool freely available, we hope the microservice research and development communities can leverage our work to experimentally investigate a more diverse set of resiliency pattern implementations and test scenarios, thus further extending our collective knowledge of how to develop more reliable and efficient microservice applications.

The rest of the paper has the following organization: The next section gives an overview of the Retry and Circuit Breaker patterns and illustrates how they can be implemented and configured in C# and Java using Polly and Resilience4J. Section 3 introduces our proposed approach. Section 4 describes the ResilienceBench tool. Section 5 presents the research questions, method, and results of our experimental study of the Retry and Circuit Breaker patterns using ResilienceBench. Section 6 compares and contrasts our contributions with related work. Finally, Section 7 offers our conclusions and directions for future research.

2 | RESILIENCY PATTERNS

Some of the most well-known resiliency patterns used today, e.g., Retry, Fail Fast, Bulkhead, and Circuit Breaker, were introduced over a decade ago in the seminal book *Release It!* by Michael Nygard.[4] Since then, those patterns have grown in popularity, especially in cloud-native microservice applications.

This section briefly describes the *purpose*, *context*, *solution*, and *implementation* of the Retry and Circuit Breaker resiliency patterns. Apart from being very popular amongst practitioners,[19,17,18] we have selected these two resiliency patterns because they both use a time-based back-off mechanism that makes it easier to compare their performance impact across multiple test scenarios. Our pattern description draws mainly from the documentation provided on Microsoft Azure's resiliency patterns website.[5] While this pattern documentation is sufficient for the scope of this paper, future work could explore more comprehensive resiliency pattern languages[22] to cover a wider range of resiliency strategies.

This section also illustrates the behavior of each pattern in an example scenario, followed by an illustrative description of the patterns' implementation and configuration parameters using Polly and Resilience4J.

2.1 | Retry

Purpose Enable an application to handle transient failures when it tries to invoke a remote service by transparently retrying a failed operation.

Context A distributed application must be resilient to the transient faults that can occur in a distributed environment. These faults, (e.g., momentary loss of network connectivity, temporary unavailability of a service, and timeouts that occur when a service is busy) are typically self-correcting. If an application repeats a failed request after a suitable delay, it will likely succeed.

Solution The Retry pattern is based on the premise that if a client application detects a transient failure when sending a request to a remote service, it should wait a suitable amount of time ("back off") before retrying the request.[7] The application repeats this process until the request succeeds or reaches a certain failure threshold. In that case, the application considers that the operation has failed.

Implementation One should define the period between retries to spread the requests from multiple application instances as evenly as possible. This strategy reduces the chance of the target service becoming more overloaded. If necessary, the Retry mechanism can increase the delays between retry attempts until it reaches some maximum number of retries. To this end, the

Retry mechanism can use either linear or exponential delay increments, depending on the type of failure and the probability that the remote service will restore its normal state during this time. If a request still fails after a significant number of retries, the Retry mechanism should prevent further requests from going to the same resource and simply report a failure.

2.2 | Circuit Breaker

Purpose Enable an application to handle faults that might take a variable amount of time to recover when invoking a remote service or resource.

Context In a distributed environment, calls to remote resources and services can fail due to unanticipated events (e.g., loss of connectivity, service failure) and might take much longer to fix than typically self-correcting transient faults. In these situations, it might be pointless for an application to continue to retry an operation that is unlikely to succeed. Instead, the application should quickly accept that the operation has failed and handle this failure accordingly, thus preventing the target service from overloading with failing requests.

Solution The Circuit Breaker pattern, inspired by electrical circuit breakers, typically operates in three states: Closed, Open, and Half-Open.[23] In the Closed state, requests are forwarded to the target service normally. When a predetermined failure threshold is reached, the circuit transitions to the Open state, blocking further requests and returning an error to the client application. After a specified timeout period, the circuit enters the Half-Open state where only a limited number of requests are allowed through. If these succeed, the circuit goes back to the Closed state; if failures persist, it returns to the Open state. This mechanism allows the system to self-heal and prevent further damage during periods of high failure rates.

Implementation The Circuit Breaker pattern is customizable and can be adapted according to the type and expected duration of the possible failure. For example, one could initially place the circuit breaker in the Open state for a few seconds, then if the failure persists, increase the timeout to a few minutes, and so on. In some cases, rather than the Open state returning failure and raising an exception, it could be helpful to return a meaningful default value to the application. Essentially, this strategy would turn the circuit breaker into a temporary surrogate for the failed service.[24]

2.3 | Example scenario

We illustrate the behavior of each pattern using an example service invocation scenario where a single client service sequentially invokes an operation provided by a target remote service (see Fig. 1a). In this scenario, the target service serves requests made by the client service with a given *fail rate*, representing the probability of the target service failing to send a correct response to the client service. This probabilistic behavior means that each request may either succeed ("OK" response) or fail ("error" response) due to the target service being either unavailable or too slow.

In this scenario, an important challenge related to implementing the client service is dealing with a failed target service. Two undesirable cases to avoid under such circumstances are:

1. If the client service continually retries each failed request, it will increase the load over the target service, thus contributing to further degrading its response time;
2. If, in contrast, the client service backs off for a certain amount of time before retrying every failed request, as a way to alleviate the load over the target service, it will increase its own execution time.

The Retry and Circuit Breaker patterns offer different solutions to cope with the two undesirable cases described above. Fig. 1b and Fig. 1c illustrate the possible behavior of a client service using either of the Retry and the Circuit Breaker patterns, respectively, in the context of the example scenario shown in Fig. 1a. In Fig. 1b, we can see that the Retry mechanism attempts to invoke the target service twice before receiving a successful response. Note that the Retry mechanism increases the retry delay after the first attempt, allowing more time for the target service to recover. In contrast, Fig. 1c shows that the Circuit Breaker transitions from Closed to Open after two failed requests; and transitions from Half-Open back to Closed after one successful request.

In principle, application developers can combine the Retry and Circuit Breaker patterns, e.g., by using a Retry mechanism to invoke a remote operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient.[23] Although some

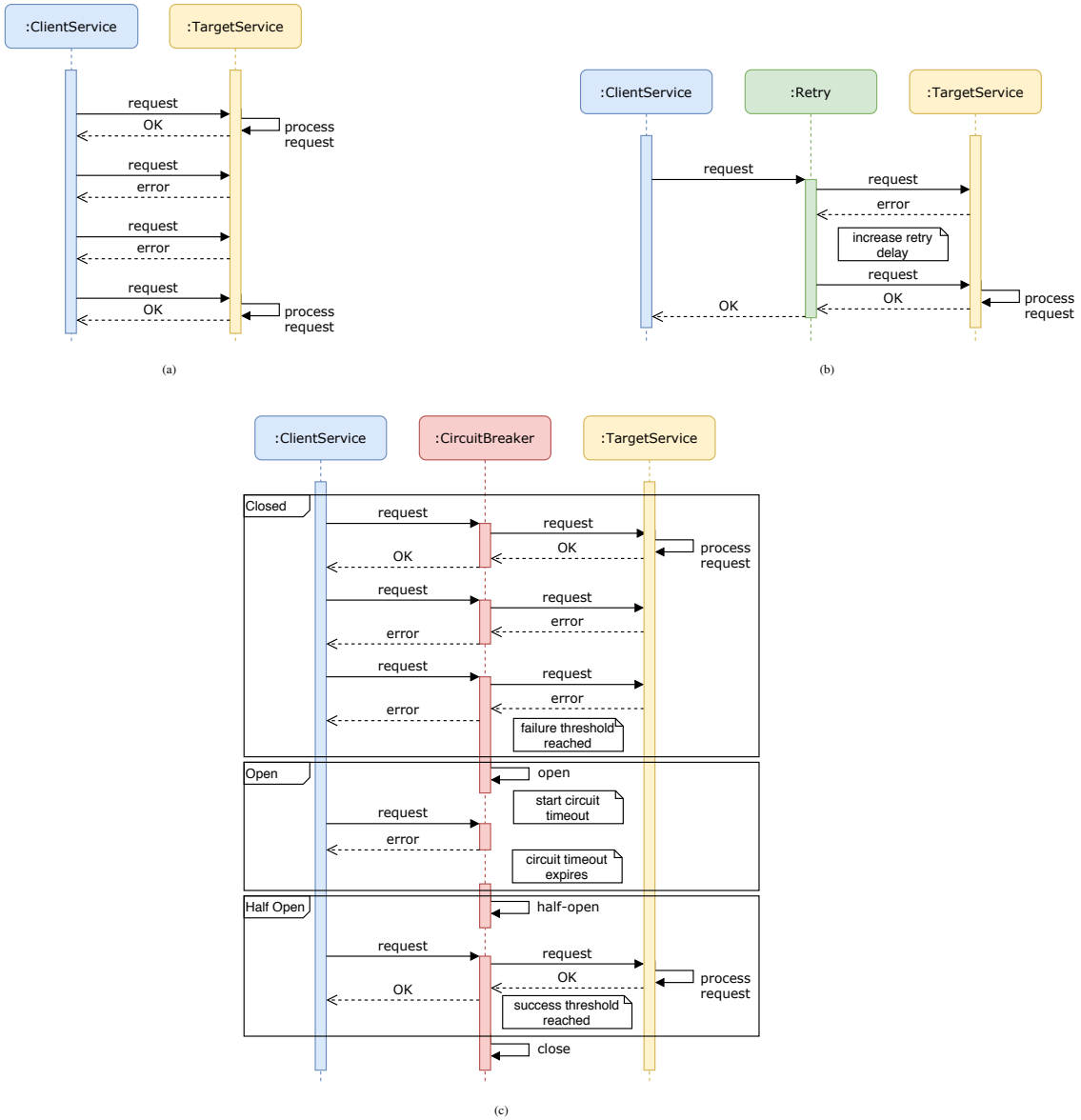

FIGURE 1 An example invocation scenario (a) and possible behavior of the Retry (b) and Circuit Breaker (c) patterns in that scenario.

resilience libraries support the combination of multiple resiliency patterns, hybrid pattern implementations are out of the scope of our current work.

2.4 | Resilience libraries

Several open-source libraries implement Retry, Circuit Breaker, and other resiliency patterns in various programming languages. Well-known examples include Java's Hystrix [25] and Resilience4j; [14] Scala's Finagle; [26] JavaScript's Cockatiel; [27] and C#'s Polly. [15] Some functionalities of the Retry and Circuit Breaker patterns, in particular, are also provided as part of the resilience features of existing service mesh middleware, e.g., Istio. [13]

In our work, we have used Polly and Resilience4j, two of the most popular resilience libraries amongst Java and C# microservice developers.[†]

[†] At the time of writing, Polly and Resilience4j have over 12,000 and 8,000 stars on GitHub, respectively.

Listing 1 Retry and Circuit Breaker in C# with Polly.

```
1  // Retry configuration
2  RetryPolicy retry = Policy
3    .Handle<HttpRequestException>()
4    .WaitAndRetry(
5      int retryCount,
6      Func<int, TimeSpan> sleepDurationProvider,
7      Action<Exception, int> onRetry
8    );
9
10 // Circuit Breaker configuration
11 var circuitBreaker = Policy
12   .Handle<HttpRequestException>()
13   .CircuitBreaker(
14     double exceptionsAllowedBeforeBreaking,
15     TimeSpan durationOfBreak
16   );
```

2.4.1 | Polly

Polly[15] is an open-source resilience and transient-fault handling library for the C#/.NET platform. Polly was first released in January 2013 with support for the Retry and Circuit Breaker patterns. Subsequent versions supported other resiliency patterns, including Bulkhead and Fall Back. We used Polly version 7.2.2+9 in our work, released on April 11, 2021.

Listing 1 shows examples of how the Retry and Circuit Breaker patterns can be configured in C# using Polly. In the Retry example (lines 2–8), the Retry mechanism will wait for the time interval returned by the function *sleepDurationProvider* (line 6) before retrying a failed HTTP request up to *retryCount* times (line 5), with the action *onRetry* (line 7) being executed on every retry. In the Circuit Breaker example (lines 11–16), the circuit will wait in the Open state for a fixed interval given in parameter *durationOfBreak* (line 15) after a certain number of failures given in parameter *exceptionsAllowedBeforeBreaking* (line 14). By adjusting the values of the configuration parameters of each pattern, developers can exert more fine-grained control over each pattern's behavior upon failure. For instance, developers can provide a customized version of the Retry's *sleepDurationProvider* function to implement a linear or exponential back-off delay strategy. Similarly, they can increase or decrease the value of the Circuit Breaker's *exceptionsAllowedBeforeBreaking* parameter to make the circuit less or more prone to break upon failures.

2.4.2 | Resilience4j

Resilience4j[14] is an open-source lightweight fault tolerance library for Java. Resilience4j was first released in January 2016, inspired by Netflix's Hystrix,[25] but designed for Java 8's functional programming features. Resilience4j provides higher-order functions (decorators) to enhance any available interface, lambda expression, or method reference with support for implementing multiple resiliency patterns, including Retry, Circuit Breaker, Rate Limiter, and Bulkhead. With Resilience4j, developers can stack more than one resiliency patterns decorator, thus creating hybrid versions of the supported resiliency patterns. We used Resilience4j version 1.7.1 in our work, released on June 25, 2021.

Listing 2 shows examples of how the Retry and Circuit Breaker patterns can be configured in Java using Resilience4j. In the Retry example (lines 2–8), the Retry configuration includes the following parameters: the maximum number of retries per call (line 3); the wait time between retries (line 4); the call result, and the exception types that trigger a retry (lines 5–6); and a parameter to indicate whether the Retry must fail after the given maximum number of retries (line 7). In addition to those parameters, Resilience4j also provides an *intervalFunction* parameter through which developers can customize the Retry behavior when calculating the wait time between retries, e.g., by providing a function to increase the wait time linearly or exponentially before each subsequent retry. In the Circuit Breaker example (lines 11–20), in turn, the Circuit Breaker mechanism configuration includes the following parameters: the percentages of failures (line 12) and slow calls (line 13) necessary to open the circuit; the wait time in the open state (line 14); the minimum threshold for call response times to be considered slow (line 15); the minimum number of calls to close the circuit when in the Half-Open state (line 16); the minimum number of calls to start calculating the Circuit Breaker metrics (line 17); the sliding window size (number of calls) within which to calculate the Circuit Breaker metrics (line 18); and the call exception types to be considered as failures (line 19). As with Polly, Resilience4j users can also adjust the values of the Retry and Circuit Breaker configuration parameters to control how each pattern will behave upon failure.

Listing 2 Retry and Circuit Breaker in Java with Resilience4j.

```
// Retry configuration
RetryConfig retryConfig = RetryConfig.custom()
  .maxAttempts(3)
  .waitDuration(Duration.ofMillis(1000))
  .retryOnResult(response -> response.getStatus() == 500)
  .retryOnException(e -> e instanceof WebServiceException)
  .failAfterMaxAttempts(true)
  .build();

// Circuit Breaker configuration
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
  .failureRateThreshold(100)
  .slowCallRateThreshold(100)
  .waitDurationInOpenState(Duration.ofMillis(1000))
  .slowCallDurationThreshold(Duration.ofSeconds(6))
  .permittedNumberOfCallsInHalfOpenState(2)
  .minimumNumberOfCalls(10)
  .slidingWindowSize(2)
  .recordException(e -> e instanceof WebServiceException)
  .build();
```

TABLE 1 Some of the Retry and Circuit Breaker configuration parameters provided by Polly and Resilience4j.

Library	Pattern	Parameter	Description
Polly	Retry	*retryCount*	Maximum number of invocation attempts (excluding the first one)
		sleepDuration	Initial back-off delay
		sleepDurationType	Back-off delay increment policy (e.g., linear, exponential, etc.)
		exponentialBackoffPow	Multiplication factor for the exponential policy
	Circuit Breaker	*exceptionsAllowedBeforeBreaking*	Required number of invocation failures to open the circuit
		durationOfBreaking	Amount of time the the circuit remains open
Resilience4j	Retry	*maxAttempts*	Maximum number of invocation attempts (including the first one)
		initialIntervalMillis	Initial back-off delay
		intervalFunction	Back-off delay increment policy (e.g., linear, exponential, etc.)
		multiplier	Multiplication factor for the exponential policy
	Circuit Breaker	*slowCallDurationThreshold*	The duration threshold above which invocations are considered slow
		slowCallRateThreshold	Percentage of slow invocations (per sliding window) upon or above which the circuit is open
		failureRateThreshold	Percentage of failed invocations (per sliding window) upon or above which the circuit is open
		minimumNumberOfCalls	Minimum number of invocations required (per sliding window) before the Circuit Breaker can calculate the failed or slow invocation rate
		slidingWindowSize	Size of the sliding window (in number of invocations) used to record the outcome of invocations when the circuit is closed
		permittedNumberOfCallsInHalfOpenState	Number of permitted invocations when the circuit is half open
		waitDurationInOpenState	Amount of time the the circuit remains open

Table 1 summarizes the Retry and Circuit Breaker configuration parameters provided by Polly and Resilience4j used in our work. We will refer to those parameters in Section 5 when describing our empirical evaluation method.

3 | APPROACH

Our approach facilitates the resiliency pattern testing cycle by providing a high-level declarative notation for engineers to specify and execute a variety of resiliency pattern-based *test scenarios*. The approach consists of four steps, as Fig. 2 shows: first, an engineer specifies a *test space* using our test specification notation (described in Section 4.2); second, a *scenario generation tool* expands that test space specification into a set of *test scenarios*; third, a *scenario execution tool* executes each test scenario and collects a set of *performance metrics* during the scenarios' execution; finally, the scenario execution tool consolidates the collected metrics and reports them to the engineer as the *test results*.

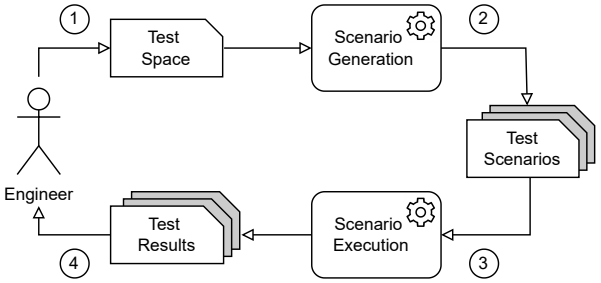

FIGURE 2 The proposed resiliency pattern evaluation approach.

We now describe and formally define the concepts of a test scenario and a test space, along with an algorithmic description of the test scenario generation process. The formalization is meant to make our definitions clear, rigorous, and unambiguous. This is critical when introducing new concepts as it avoids misinterpretation and facilitates understanding across diverse audiences.

3.1 | Scenario specification

In our approach, a test scenario describes a particular resiliency pattern testing session in which a *client service* creates a certain number of *virtual users* to concurrently invoke a *target service* until they reach a required number of *successful invocations*. Moreover, during the test scenario execution, the target service may fail according to a certain *failure rate*. Upon failure of the target service, the client service's virtual users may behave differently, depending on the *resiliency strategy* they use to invoke the target service and the values of their resiliency strategy's *configuration parameters*.

Formally, we define a test scenario as follows.

Definition 1 (**Test Scenario).** Let p be a development platform, let $RES(p)$ be the set of resiliency strategies supported by p, and let $CONF(rs)$ be the set of all possible configurations of a resiliency strategy $rs \in RES(p)$. A test scenario is a tuple $T = (ts, f, cs, rs, conf, u, n)$ where

- ts is a target service;
- f $(0 \leq f < 1)$ is the failure rate of ts;
- cs is a client service instance developed in p;
- $rs \in RES(p)$ is a resiliency strategy used by cs;
- $conf \in CONF(rs)$ is a unique configuration of rs;
- $u \in \mathbb{Z}^+$ is the number of virtual users cs creates to invoke ts using rs configured with $conf$; and
- $n \in \mathbb{Z}^+$ is the minimum number of successful invocations of ts each cs virtual user is required to perform during a test.

A *test space* is a generalized test scenario specification where one or more of the following scenario elements can have multiple definitions: the target service's failure rate, the client service's number of virtual users, the client service instance, and the client service instances' resiliency strategy configuration.

Formally, we define a test space as follows.

Definition 2 (**Test Space).** Let P be a set of development platforms, let $p \in P$ be a development platform, let $RES(p)$ be the set of resiliency strategies supported by p, let $CONF(rs)$ be the set of all possible configurations of a resiliency strategy $rs \in RES(p)$, and let $Plat(cs) \in P$ be the development platform of a client service cs. A test space is a tuple $S = (ts, F, CS, RS, CONFS, U, n)$ where

- ts is a target service;
- $F = \{f_i | 0 \leq f_i < 1, i \in \mathbb{Z}^+\}$ is an ordered set of failure rates of ts;
- $CS = \{cs_i | Plat(cs_i) \in P, i \in \mathbb{Z}^+\}$ is an ordered set of client service instances;
- $RS = \{rs_i | rs_i \in RES(Plat(cs_i)), cs_i \in CS, 1 \leq i \leq |CS|\}$ is an ordered set of resiliency strategies used by each client service instance in CS;

Algorithm 1 Scenario generation process

Require: a test space $S = (ts, F, CS, RS, CONFS, U, n)$
Ensure: $|TS| = Size(S)$, where TS is a set of generated test scenarios

```
TS ← ∅
for all f ∈ F do
   for all u ∈ U do
      for all cs, rs, CONF ∈ CS, RS, CONFS do
         for all conf ∈ CONF do
            T ← (ts, f, cs, rs, conf, u, n)
            TS ← TS ∪ {T}
         end for
      end for
   end for
end for
return TS
```

- $CONFS = \{CONF_i | CONF_i \subseteq CONF(rs_i), rs_i \in RS, 1 \leq i \leq |RS|\}$ is an ordered set containing subsets of all possible configurations of each resiliency strategy in RS;
- $U = \{u_i | u, i \in \mathbb{Z}^+\}$ is an ordered set of numbers of virtual users each client service in CS creates to invoke ts; and
- $n \in \mathbb{Z}^+$ is the minimum number of successful invocations of ts each client service virtual user is required to perform during a test.

We denote the *size of a test space* as the total number of test scenarios one can derive by expanding the test space's multi-value elements.

Definition 3 (**Test Space Size).** Let $S = (ts, F, CS, RS, CONFS, U, n)$ be a test space. Then, the size of S, denoted $Size(S)$, is given by Equation 1.

$$Size(S) = |F| \times |U| \times \sum_{i=1}^{|CS|} |CONF_i|, CONF_i \in CONFS \tag{1}$$

Note that Equation 1 calculates the cartesian product of all combinations of a test space's multi-value elements.

3.2 | Scenario generation

The scenario generation process expands a given test space specification by instantiating multiple test scenarios containing all possible combinations of the test space's multi-value elements. To facilitate understanding and reproducibility of our approach, we provide an algorithmic description of the scenario generation process in Algorithm 1. From that algorithm, we can see that this process produces new test scenarios grouped by failure rate, the number of virtual users, client service instance/resiliency strategy, and resiliency strategy configuration. At execution time, the scenario execution tool executes and collects the performance metrics from each generated test scenario in that order, optionally multiple times per scenario, as we will explain next.

4 | RESILIENCEBENCH

We have implemented ResilienceBench,[‡] a resiliency pattern benchmarking tool, as a proof-of-concept for our approach. Below we describe ResilienceBench's run-time architecture and test space notation.

[‡] https://github.com/ppgia-unifor/resilience-bench

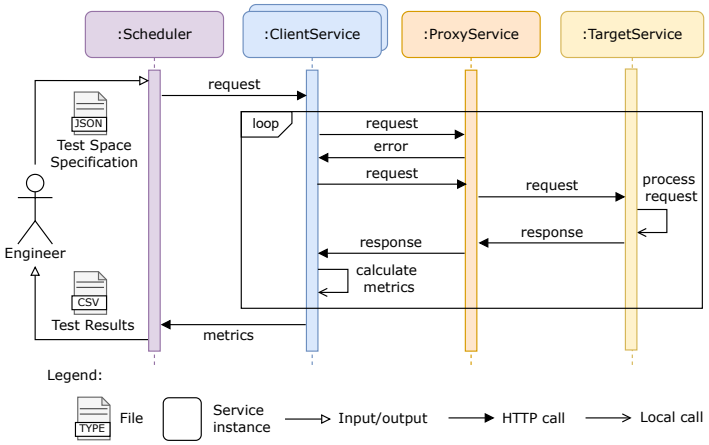

FIGURE 3 ResilienceBench's run-time architecture.

TABLE 2 ResilienceBench's collected performance metrics.

Metrics	Description
SuccessfulCalls *UnsuccessfulCalls* *TotalCalls*	Number of successful/unsuccessful/total invocations of the target service by the client service (excluding pattern-level re-invocations)
SuccessfulRequests *UnsuccessfulRequests* *TotalRequests*	Number of successful/unsuccessful/total invocations of the target service by the client service (including pattern-level re-invocations)
TotalSuccessTime *AverageSuccessTime*	Total/average time the client service spends waiting for successful responses from the target service (in seconds)
TotalErrorTime *AverageErrorTime*	Total/average time the client service spends waiting for failed responses from the target service (in seconds)
Throughput	Average number of (either successful or unsuccessful) invocations of the target service by the client service per second
TotalExecutionTime	Total time the client service takes to complete the required number of successful invocations of the target service (in seconds)
TotalContentionTime	Total accumulated time the client service spends invoking or waiting for a response from the target service (in seconds)
ContentionRate	Rate between the client service's *TotalContentionTime* and *TotalExecutionTime*

4.1 | Architecture

ResilienceBench's run-time architecture comprises four main services, as depicted in Fig. 3: a *scheduler*, a *client service*, a *proxy service*, and a *target service*. The scheduler plays the role of both the scenario generation and the scenario execution tools in our approach. It (i) parses and expands the test space specification provided by the engineer as a JSON file; (ii) invokes each of the client service instances at a time, passing them the required test parameters (e.g., the target service's URL and failure rate, the client service's number of virtual users and resiliency strategy configuration, etc.); (iii) collects and consolidates the performance metrics received from each client service instance; and (iv) returns the consolidated performance metrics to the engineer 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. Table 2 describes the set of performance metrics ResilienceBench currently collects during the execution of a test scenario.

The proxy service is responsible for injecting faults of a certain type (e.g., an abort failure) into the target service's request stream according to the specified failure rate. For instance, a 25% failure rate means that the proxy service will inject one failure for every three requests the target service receives from the client application. Finally, the target service serves the client service's requests and is entirely oblivious of ResilienceBench' other services. The decision to use a separate proxy service for fault

Listing 3 A ResilienceBench test space specified in JSON.

```
1  {
2    "rounds": 10,
3    "users": [50, 100, 150],
4    "succRequests": 10,
5    "maxRequests": 500,
6    "targetUrl": "http://server:9211/bytes/1000",
7    "fault": {
8      "type": "abort",
9      "percentage": [0, 20, 40, 60, 80],
10     "status": 503
11   },
12   "clientSpecs": [
13     {
14       "platform": "Java",
15       "strategy": "Retry",
16       "lib": "Resilience4j",
17       "url": "http://resilience4j:8080/retry",
18       "patternConfig": {
19         "maxAttempts": 6,
20         "multiplier": 1.5,
21         "intervalFunction": "EXPONENTIAL_BACKOFF",
22         "initialIntervalMillis": [20, 40, 60]
23       }
24     },
25     {
26       "platform": ".NET",
27       "strategy": "Circuit Breaker",
28       "lib": "Polly",
29       "url": "http://polly/circuitbreaker",
30       "patternConfig": {
31         "exceptionsAllowedBeforeBreaking": 2,
32         "durationOfBreaking": [20, 40, 60]
33       }
34     }
35   ]
36 }
```

injection is justified since this allows ResilienceBench users to change the target service's failure rate independently of the actual target service used.

The scheduler is a native service implemented in Python. In contrast, developers can implement new client service instances on any platform using any available resilience library. Currently, ResilienceBench includes two versions of the client service, implemented in C# and Java, using the Polly and Resilience4j resilience libraries, respectively. Both versions implement three resiliency strategies for invoking the target service during scenario execution: (i) using the Retry pattern; (ii) using the Circuit Breaker pattern; and (iii) using no resiliency pattern at all, which we refer to as the Baseline strategy. For the proxy service, ResilienceBench uses Envoy,[28] a well-known proxy service and sidecar for microservice applications. However, any other proxy service capable of transparently injecting faults into an HTTP request stream would also fit the bill. Finally, for the target service, ResilienceBench uses httpbin,[29] a simple HTTP service commonly used to test client-side web applications. Again, due to the loosely coupled nature of ResilienceBench' architecture, any other HTTP service could fill that role.

We deploy the scheduler and the client service instances in two separate Docker containers and the proxy and the target services in a single Docker container. The reasoning behind the decision to deploy the two backend services in a single container is three-folded: (i) to minimize the impact of potential communication delays between those two services; (ii) to make the existence of the proxy service fully transparent to the client service; and (iii) to avoid having the proxy service artificially reducing the load over the target service during fault injection. The latter reasoning is justified as the proxy service injects failures by filtering out requests from the target service request stream. This means that during failure injection the proxy service effectively reduces the load on the target service. By deploying the proxy service and the target service in the same container, we counter this effect by having both services share the container's resources.

4.2 | Test space notation

ResilienceBench uses the JSON data format as the notation for the specification of test spaces. Listing 3 shows an example of a JSON specification for a test space. That test space contains two parts: the first part (lines 2–11) defines the test control parameters; the second part (lines 12–35), in turn, defines two client service instances with their respective resiliency pattern

configurations. The test control parameters are the number of executions (rounds) of each scenario (line 2); the number of client service virtual users that will currently invoke the target service (line 3); the required number of successful target service invocations (line 4); the maximum number of target service invocations allowed (line 5); the target service's URL (line 6); and the target service's composite fault parameter, which includes the fault's type (line 8), percentage (line 9), and status code (line 10). The two client service instances contain the following attributes: development platform (lines 14 and 26); resilience strategy (lines 15 and 27); resiliency library (lines 16 and 28); access URL (lines 17 and 29); and resiliency pattern configuration (lines 18–23 and 30–33), which is an optional composite element that may contain a set of pattern-specific resiliency parameters.

Note that the test space specification in Listing 3 defines several parameters as a list of primitive values. These list-like parameters correspond to the multi-value elements of a test space (see Definition 2), including the number of client service virtual users (line 3), with three values, and the target service's fault percentage (line 9), with five values. In addition, that test space specification also assigns multiple values to some of the client services' pattern-specific configuration parameters, namely, Resilience4j's *initialIntervalMillis* Retry parameter (line 22) and Polly's *durationOfBreaking* Circuit Breaker parameter (line 32), both with three values. At scenario execution time, the scheduler service of ResilienceBench will follow the process described in Algorithm 1 to generate multiple test scenarios containing all possible combinations of the test space's list-like parameters. Applying Algorithm 1 to the test space specification shown in Listing 3 will produce a total of 90 ($3 \times 5 \times (3 + 3)$), as per Equation 1, unique test scenarios after expansion of all its list-like parameters.

5 | EXPERIMENTAL EVALUATION

We have conducted an experimental evaluation of the Retry and Circuit Breaker patterns to highlight the benefits of our approach. Below we describe our research questions, method, and experimental results. We also discuss the implications and limitations of our findings.

We should emphasize that our experimental setting and test scenarios are not exhaustive and are meant to illustrate the current features and capabilities of our benchmark tool.

5.1 | Research questions

Our evaluation aims to shed light on the following two research questions:

RQ1 What is the performance impact experienced by a client service that continuously invokes a failure-prone target service using different configurations of the Retry and Circuit Breaker resiliency patterns?

RQ2 How is that impact influenced by factors such as the client service's programming language/resilience library and the target service's workload and failure rate?

In principle, each client service virtual user should immediately re-invoke a failed or unresponsive target service to complete the required number of successful invocations as quickly as possible. However, if all virtual users follow this naive strategy, they would also increase the risk of overloading the target service with far too many unnecessary requests, which could inadvertently degrade the client service's execution time. The Retry and Circuit Breaker resiliency patterns aim to strike a balance between these two performance constraints by deliberately deciding when to re-invoke or back off from re-invoking an unresponsive target service. Our experimental evaluation shows how well and under what conditions the Retry and Circuit Breaker patterns strike that balance in a controlled test environment.

5.2 | Method

We have used ResilienceBench to specify and execute multiple test scenarios to answer the above questions. To answer RQ1, we have analyzed two of the performance metrics ResilienceBench collects during the tests: the target service's *average success time* and the client application's *total execution time*. The target service's response time measures the average response time of the target service, as measured by the virtual users of the client application. A lower response time indicates a lower load exerted on the target service by the client application. The client application's execution time measures the average time taken by

Listing 4 Test space specification used in the experimental evaluation.

```
1  {
2    "rounds": 10,
3    "users": [50, 100, 150],
4    "succRequests": 10,
5    "maxRequests": 500,
6    "targetUrl": "http://server:9211/bytes/1000",
7    "fault": {
8      "type": "abort",
9      "percentage": [0, 20, 40, 60, 80],
10     "status": 503
11   },
12   "clientSpecs": [
13     {
14       "platform": ".NET",
15       "strategy": "Baseline",
16       "lib": "",
17       "url": "http://polly/baseline",
18       "patternConfig": {}
19     },
20     {
21       "platform": ".NET",
22       "strategy": "Retry",
23       "lib": "Polly",
24       "url": "http://polly/retry",
25       "patternConfig": {
26         "retryCount": 5,
27         "sleepDurationType":
           ↪  "EXPONENTIAL_BACKOFF",
28         "exponentialBackoffPow": 1.5,
29         "sleepDuration": [20, 40, 60]
30       }
31     },
32     {
33       "platform": ".NET",
34       "strategy": "Circuit Breaker",
35       "lib": "Polly",
36       "url": "http://polly/circuitbreaker",
37       "patternConfig": {
38         "exceptionsAllowedBeforeBreaking": 2,
39         "durationOfBreaking": [20, 40, 60]
40       }
41     },
42     {
43       "platform": "Java",,
44       "strategy": "Baseline",
45       "lib": "",
46       "url":
         ↪  "http://resilience4j:8080/baseline",
47       "configTemplate": {}
48     },
49     {
50       "platform": "Java",
51       "strategy": "Retry",
52       "lib": "Resilience4j",
53       "url": "http://resilience4j:8080/retry",
54       "patternConfig": {
55         "maxAttempts": 6,
56         "multiplier": 1.5,
57         "intervalFunction":
           ↪  "EXPONENTIAL_BACKOFF",
58         "initialIntervalMillis": [20, 40, 60]
59       }
60     },
61     {
62       "platform": "Java",
63       "strategy": "Circuit Breaker",
64       "lib": "Resilience4j",
65       "url": "http://resilience4j:8080/cb",
66       "patternConfig": {
67         "slowCallRateThreshold": 100,
68         "slowCallDurationThreshold": 1000,
69         "slidingWindowSize": 2,
70         "failureRateThreshold": 100,
71         "minimumNumberOfCalls": 2,
72         "permittedNumberOfCallsInHalfOpenState
           ↪  ": 1,
73         "waitDurationInOpenState": [20, 40, 60]
74       }
75     }
76   ]
77 }
```

each virtual user of the application to complete the required number of successful invocations of the target service. A lower execution time indicates higher efficiency of the client application. In addition, we have evaluated multiple configurations of each pattern by varying the value of their back-off delay parameter. Finally, we have compared the performance of each resiliency pattern against the performance of a Baseline strategy which does not use any resiliency pattern. This Baseline strategy has no consideration for the load exerted on the target service and immediately re-invokes every failed invocation with no back-off delay until it reaches the required number of successful invocations.

To answer RQ2, we have evaluated the three resiliency strategies (using the two patterns plus the Baseline strategy) implemented in two programming languages (C# and Java) using different resilience libraries (Polly and Resilience4j, respectively). Moreover, during the tests, we varied the target service's failure rate and workload by providing multiple values to the fault percentage and number of virtual users parameters.

5.2.1 | Test space

Listing 4 shows the test space specification we have created in our experimental evaluation. In that test space, we have defined the main control parameters with the following values: 10 rounds per scenario (line 2); 50, 100, and 150 client service virtual users (line 3); 10 required successful invocations of the target service per virtual user (line 4); and 0, 20, 40, 60, and 80 percentages of abort faults injected into the target service invocation stream with 503 status code (lines 7–11). Moreover, to investigate how the three resiliency strategies perform across the two development platforms, we have defined six client service instances (lines 13–19, 20–1, 32–41, 42–48, 49–60, and 61–75), each involving a different programming language and a different resilience strategy. Finally, to experiment with multiple resiliency pattern configurations, we have defined the different Retry and Circuit Breaker back-off delay parameters provided by Polly and Resilience4j with the same set of values, namely, 20, 40, and 60 ms (lines 29, 39, 58, and 73).

TABLE 3 Test parameters used in the experiment.

Library	Pattern	Parameter	Value(s)
All	All	No. executions per test	10
		No. virtual users	50, 100, 150
		No. successful invocations per virtual user	10
		Failure rate (%)	0, 20, 40, 60, 80
Polly	Retry	*retryCount*	5
		sleepDuration	20ms, 40ms, 60ms
		sleepDurationType	exponential
		exponentialBackoffPow	1.5
	Circuit Breaker	*exceptionsAllowedBeforeBreaking*	2
		durationOfBreaking	20ms, 40ms, 60ms
Resilience4j	Retry	*maxAttempts*	6
		initialIntervalMillis	20ms, 40ms, 60ms
		intervalFunction	exponential
		multiplier	1.5
	Circuit Breaker	*slowCallDurationThreshold*	1000
		slowCallRateThreshold	100%
		failureRateThreshold	100%
		minimumNumberOfCalls	2
		slidingWindowSize	2
		permittedNumberOfCallsInHalfOpenState	1
		waitDurationInOpenState	20ms, 40ms, 60ms

Table 3 lists all test parameters utilized in our experiment, along with their respective context and values. The parameter values were carefully selected to encompass a representative range of Retry configurations and test profiles. For instance, we distributed the values of the backoff delay multiplier and maximum number of retries parameters around 1.5 and 5, respectively, which are either equal or close to the default values of those parameters in Resilience4j. Additionally, we conducted preliminary tests to determine values for the initial backoff delay parameter that would have a significant impact on the client application's execution time as the target service's failure rate increases. Finally, we intentionally limited the number of values assigned to the number of virtual users and failure rate parameters to ensure a clear presentation and to keep the number of plots shown in the paper manageable.

We have defined all other configuration parameters with fixed values. For example, for the Retry pattern, we have limited the maximum number of re-invocations to 5 (lines 26 and 55). For the Circuit Breaker pattern, we have defined the number of failed requests necessary to open the circuit to 2 (lines 38 and 69–70) and the number of successful requests needed to close the circuit to 1 (lines 71–72).[§]

Applying Algorithm 1 to the test space specification shown in Listing 4 will produce a total of 210 ($3 \times 5 \times (1+3+3+1+3+3)$), as per Equation 1, unique test scenarios. This number shows the power of using a declarative approach to easily specify a significant number of resiliency pattern-based test scenarios, under various operational conditions, in a compact machine-readable format.

5.3 | Results

Fig. 4 and Fig. 5 show the impact of the target service's fault percentage on the two performance metrics evaluated, respectively, for the three resiliency strategies the client service uses to invoke the target service. Both figures display a 2×3 grid with six plots, where the grid columns show the plots for the three workloads, and the grid rows show the plots for the two development platforms/resilience libraries. Each plot depicts the median of one of the two performance metrics for each resiliency strategy computed over the results of all client services' virtual users and all scenario executions, with a 95% confidence interval. The plots also depict the results for each resiliency strategy with a different color, marker, and line style. The increasing widths of each resiliency strategy's line represent that strategy's increasing back-off delays. For visual consistency, we assume that the Baseline strategy has a single back-off delay value of 0 ms.

[§] Note in Table 1 that Polly does not provide a parameter to configure the number of successful requests needed to close the circuit, which is always one.

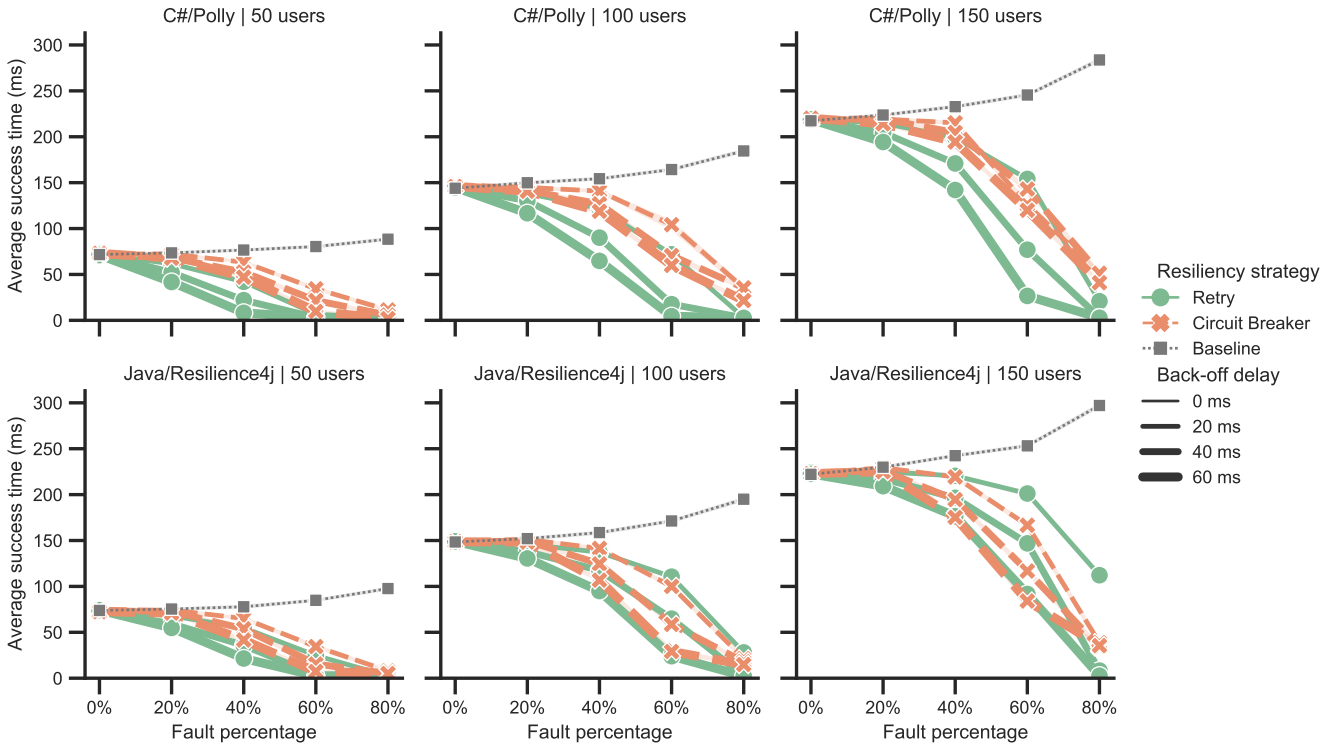

FIGURE 4 Impact of fault percentage, number of virtual users, and development platform/resilience library on the target service's average success time.

5.3.1 | Average success time

As we can see in Fig. 4, when the client service uses the Baseline strategy, the target service's average success time gradually increases as the target service's fault percentage and workload increase. These results hold across the two development platforms. The increase in the target service's average success time is due to the number of client service's concurrent requests also increasing. We observe this behavior under higher failure rates, due to the client service virtual users having to re-invoke the target service more often, and under higher workloads, as there are more virtual users to invoke the target service.

We can also see in Fig. 4 that both patterns significantly reduce the target service's average success time compared to the Baseline strategy in both development platforms. Moreover, the patterns' impact on the target service's average success time increases with the value of their back-off delay parameter and the target service's fault percentage. The explanation for these results is that, in contrast to the Baseline strategy, which continuously re-invokes the target service upon failure, the Retry and Circuit Breaker strategies force the client service virtual users to back off for a certain amount of time before re-invoking a failed target service. This self-imposed blocking of the client service requests alleviates the load over the target service under higher failure rates. In addition, since the Retry pattern exponentially increases the back-off delay after successive failed invocations of the target service, it tends to further impact the target service's response time compared to the Circuit Breaker pattern. The results for the C#/Polly platform in Fig. 4 clearly show this trend.

5.3.2 | Total execution time

While both Retry and Circuit Breaker strategies can reduce the target service's average success time compared to the Baseline strategy, such an impact may not necessarily decrease the client service's total execution time. The reason is that both resiliency patterns will inevitably delay the client service's execution as they force its virtual users to back off before re-invoking the target service upon failure. In practice, the overall impact of the Retry and Circuit Breaker strategies on the client service's total execution time will result from a trade-off between each pattern's capability to reduce the target service's response time and the inevitable delays they impose on the client service's execution.

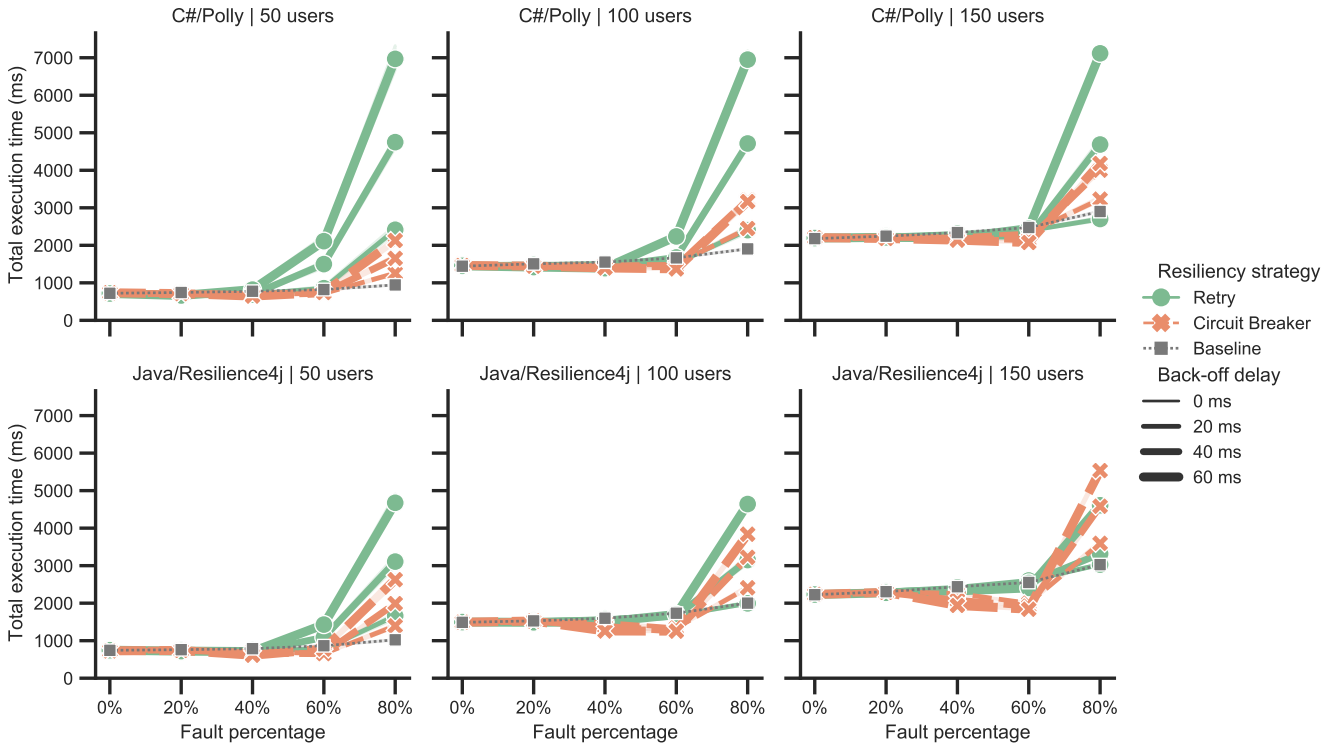

FIGURE 5 Impact of fault percentage, number of virtual users, and development platform/resilience library on the client service's total execution time.

As we can see in Fig. 5, compared to the Baseline strategy, the Retry and Circuit Breaker strategies barely have any impact on the client service's total execution time up to a 40% failure rate of the target service, in both platforms, across all workloads. These results mean that, up to that failure rate, the patterns' reduction of the target service's response time compensates for the delays they introduce to the client service's execution. The Circuit Breaker strategy even shows a slight performance gain over the Baseline for the 40-60% failure rate range, with the magnitude of that gain being more visible in the Java/Resilience4j platform. Above that range, the results of the Retry and Circuit Breaker strategies start to outgrow those of the Baseline rapidly. This behavior means that, under high failure rates, the delays both patterns introduce to the client service's execution far outweigh the patterns' reduction of the target service's response time.

In both platforms, we observe that the results obtained with the Retry pattern configured with higher back-off delays tend to be greater than those of the Circuit Breaker pattern configured with the same back-off delays. Again, these differences are more prominent in the C#/Polly platform. However, as the workload increases, the results of the two patterns under higher failure rates start to converge. This behavior is evident in the Java/Resilience4j platform with 150 concurrent virtual users, where the Circuit Breaker results completely outgrow the Retry results as the target service's failure rate reaches 80%. One possible explanation for those results is that, under high workloads, the target service's response time has a more significant impact on the client service's total execution time than the back-off strategies each resiliency pattern implements, which remain unchanged across the three workloads.

5.4 | Discussion

Our experimental evaluation of the Retry and Circuit Breaker patterns shows that both resiliency strategies can effectively reduce the target service's response time compared to the Baseline strategy across all workloads and development platforms investigated. However, our evaluation also shows that the patterns' gain in the target service's response time only compensates for their loss in the client service's total execution time under low to moderate failure rates (up to 40%-60%). Nevertheless, we believe that the failure rate range where the two patterns are most effective is arguably the most realistic for production-grade microservice

applications. Thus, based on our experimental results, we recommend using both patterns as a mitigation strategy to invoke a remote service that is (slightly to moderately) likely to fail.

Another point of discussion is how to define each pattern's back-off delay parameter. On the one hand, if the patterns' back-off delay value is too large, using either strategy will be counterproductive, as both will delay the application's execution for far too long. On the other hand, if the patterns' back-off delay value is too small, their use also will be counterproductive, as both strategies will essentially behave like the Baseline strategy. Therefore, one has to find a proper balance when configuring the back-off delay parameters of the Retry and Circuit Breaker patterns.

In our experimental evaluation, we set the back-off delay values of the two patterns as increasing fractions of the target service's average response time under low demand. As Fig. 4 shows, the target service's average response time with 50 concurrent virtual users (the lowest workload evaluated) is about 60-70 ms; for this reason, we varied the back-off delay of both patterns in increments of about 1/3 of the target service's average response time under the lowest workload. As Fig. 4 also shows, the smallest back-off delay value (20 ms) is enough to make both patterns impose a significant reduction in the target service's average response time, with that reduction increasing even further as the target service's failure rate increases. These results suggest that using the target service's average response time under low demand as a reference may be a valuable rule of thumb to determine the best values for the Retry and Circuit Breaker's back-off delay parameters.

Regarding the two development platforms/resilience libraries evaluated, we have observed some slight discrepancies in their experimental results. In particular, in the C#/Polly platform, the Retry pattern outperforms the Circuit Breaker pattern by a considerable margin in reducing the target service's average success time (a positive impact) and increasing the client service's total execution time (a negative impact). In the Java/Resilience4j platform, the two patterns show a more similar behavior overall, with the Retry pattern initially outperforming the Circuit Breaker pattern in both metrics under the lowest workload and the Circuit Breaker pattern subsequently catching up and finally surpassing the Retry pattern as the workload increases. The results for Java/Resilience4j differ from our expectation, which was somewhat of a surprise since we expected the Retry strategy, which increments the back-off delay exponentially, to outperform the Circuit Breaker strategy in both platforms, especially under higher failure rates. One possible explanation for this discrepancy between the results of the two platforms is that their two resilience libraries might provide slightly different implementations of the two patterns, potentially causing the patterns' behavior to differ from one library to another at run time. A subsequent investigation of the impact of the Retry pattern using ResilienceBench, considering a wider range of pattern configurations, has confirmed this hypothesis.[21]

5.5 | Threats to validity

Some of our methodological decisions may have affected the validity of our results. First, we might have failed to configure each resiliency pattern to behave similarly using either resilience library. This threat might explain the discrepancies we have observed when comparing the results obtained with each platform/resilience library. Second, we only varied the value of a single configuration parameter of each resiliency pattern and experimented with a single fault type. This threat means that other possible pattern configurations (e.g., with different values for the maximum number of retry attempts or the minimum number of failed requests required to open the circuit) might have produced different results. Third, we only have investigated two resiliency patterns implemented in two programming languages using two resilience libraries. This threat means our results might not generalize to other resiliency patterns, resilience libraries, and development platforms. Fourth, we have conducted our experiment in a tightly controlled setting, using a benchmark-specific client application with a single target service. This threat means that our results might not generalize to more realistic microservice applications with multiple downstream services. Nevertheless, we believe that conducting experimental evaluations in a controlled setting provides distinct advantages that are particularly welcome before moving on to more complex real-world scenarios. A controlled setting offers precise control over variables, enabling accurate measurement of their impact, and promoting reproducibility. It also simplifies the system under investigation, reducing complexity and potential confounding factors. Moreover, data collected in such a setting tends to be of high quality, free from uncontrolled noise that might distort results. We plan to experiment with more realistic applications and workloads in future work. Finally, a significant challenge in assessing resilience in microservices is the testing oracle problem, where it may be difficult to determine the correctness of the system's behavior under all relevant failure scenarios due to the absence of an explicit oracle. This can impact the assurance and guarantees provided by the resilience tests. Future work could explore metamorphic testing approaches[30] or the identification of resilience anti-patterns[31] to mitigate this issue.

6 | RELATED WORK

Our work falls within the broader context of resilient systems.[32] There is already an extensive body of research literature in this area, including topics such as system reliability engineering,[33,6] software reliability modeling and prediction,[34,35,36,37] and, more recently, microservice resiliency evaluation and testing.[38,39,40,41,42] The need for further research on enhancing the fault tolerance and resilience of microservice-based systems has also been recognized by recent literature reviews on this topic.[43,44] However, relatively few studies have evaluated the use and impact of resiliency patterns in microservice applications.[45,46,47,9,10,11] Below we discuss the nature and contributions of some of those studies, in addition to another related study on microservice anomaly detection,[48] and how they compare with our work.

Montesi and Weber[45] have implemented several resiliency patterns in the context of the Jolie microservice language,[49] including three variants of the Circuit Breaker pattern. Preuveneers and Joosen[46] have proposed a circuit breaker framework enhanced with the notion of Quality of Context to improve the resiliency of context-aware distributed applications. Aquino *et al.*[47] have studied the use of the Circuit Breaker pattern in the context of Internet of Things (IoT) applications and evaluated its potential benefits in a prototype traffic light system. More recent work[9,10] has focused on the formal modeling and analysis of microservice resiliency patterns. Mendonça *et al.*[9] have proposed a model checking-based approach to analyze the behavior of the Retry and Circuit Breaker patterns as continuous-time Markov chains (CTMC).[50] They have used the PRISM probabilistic model checker[12] to quantify the patterns' performance impact in a simple client-service interaction scenario with a single client. Jagadeesan and Mendiratta[10] have followed a similar model-based approach to analyze the behavior of the Circuit Breaker pattern in a more elaborate yet still contrived multi-client microservice interaction scenario, also using PRISM. Finally, Sedghpour *et al.*[11] have empirically studied the impact of the Retry and Circuit Breaker patterns in the context of an existing service mesh middleware, Istio,[13] where the service mesh administrator is responsible for enabling and configuring both resiliency patterns at the infrastructure-level.

In contrast to the works described above, we have focused on providing a more general benchmarking approach for empirically evaluating multiple resiliency patterns, implemented in any development platform and resilience library currently available. In addition, we have developed and practically demonstrated the use of ResilienceBench, an open-source resiliency pattern benchmark tool based on our proposed approach. Nevertheless, our work still has some limitations compared to the above. In particular, ResilienceBench currently only supports running evaluation experiments in an extensible yet restricted containerized application with a single target service. Moreover, our test space notation lacks the flexibility and richer semantics of a fully-fledged probabilistic model checker, such as PRISM. Finally, our focus on application-level resiliency patterns makes evaluating multiple pattern configurations across application services implemented in different development platforms more complex than other infrastructure-level approaches.

Due to the popularity of resiliency patterns amongst practitioners, Retry, Circuit Breaker and other patterns have recently been the focus of several industry forums and technology blogs.[24,51,17,18,19,20] While those forums and blogs can be a rich source of technical information on how to use resiliency patterns in production, they mainly offer anecdotal evidence to support their claims. In that sense, our work can complement those industry-led discussions by supporting a more systematic analysis of the benefits and risks of using well-known resiliency patterns under varying workloads and failure scenarios.

Another related line of work is the design of self-adaptive resiliency mechanisms.[52,53] In that direction, Sedghpour *et al.*[52] recently proposed a self-adaptive circuit-breaker on top of Istio, which is capable of dynamically adjusting some of its configuration parameters based on a given set of system metrics. The concept of a self-adaptive service mesh proposed by Mendonça and Aderaldo[53] follows a similar approach. The current version of ResilienceBench does not support dynamic adaptation of the resilience patterns' configuration parameters. We plan to add this feature in a future version.

Finally, the increasing complexity of microservice systems has also led to research on anomaly detection and fault diagnosis, leveraging distributed tracing tools like OpenTracing.[54] Khanahmadi *et al.* proposed a model that utilizes OpenTracing in combination with machine learning algorithms to detect and categorize software anomalies with high accuracy.[48] Their work emphasizes the dynamic extraction of service dependency graphs from trace data, which contrasts with previous approaches that often rely on static dependency graphs. Although their focus was on anomaly detection, their work complements our approach by highlighting the importance of dynamic tracing and dependency graph analysis in understanding the behavior of microservices under various failure scenarios. Integrating similar dynamic tracing methodologies could enhance the resilience benchmarking process in our work, especially when dealing with complex failure modes in microservices.

7 | CONCLUSION

Microservice developers are increasingly using resiliency patterns, such as Retry and Circuit Breaker, to improve the reliability of their applications. However, thus far, few studies have focused on supporting application developers in better understanding the impact of those patterns on application performance. This paper introduced a novel declarative approach and its supporting benchmark tool for experimentally evaluating the performance impact of existing resiliency pattern implementations in a controlled setting. An experimental study of the performance impact caused by the Retry and Circuit Breaker patterns implemented in both C# and Java showed how the proposed approach and tool could significantly facilitate the specification and execution of a large variety of resiliency pattern-based tests under multiple environmental conditions. We have made our tools and experimental data publicly available[16] to stimulate collaboration with other research groups and to facilitate the replication of our experimental results. In that regard, we invite the microservice development and research communities to try and experiment with ResilienceBench and to contribute to its development and evolution.

Looking ahead, our main topics for future work include: evaluating the impact of other resiliency pattern configurations and resilience libraries; extending ResilienceBench to support more realistic microservice applications with multiple upstream and downstream services; and exploring and understanding the interplay between existing resiliency patterns and other more advanced resiliency mechanisms, e.g., *rate-limiting*[55] and *deadlines*.[56]

FINANCIAL DISCLOSURE

Nabor C. Mendonça is partly supported by Brazil's National Council for Scientific and Technological Development (CNPq) under grant no. 313558/2023-0.

CONFLICT OF INTEREST

The authors declare no potential conflict of interests.

References

1. Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software.* 2018;35(3):24–35.
2. Lindsay B. Designing for failure may be the key to success—interview by Steve Bourne. *ACM Queue.* 2004;2(8).
3. Lewis J, Fowler M. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html; 2014. [Last access on August 12, 2024].
4. Nygard M. *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.
5. Microsoft Azure . Resiliency patterns. https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency; 2017. [Last accessed on August 12, 2024].
6. Beyer B, Jones C, Petoff J, Murphy NR. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly, 2016.
7. Microsoft Azure . Retry Pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/retry; 2017. [Last accessed on August 12, 2024].
8. Fowler M. CircuitBreaker. https://martinfowler.com/bliki/CircuitBreaker.html; 2014. [Last access on August 12, 2024].
9. Mendonça NC, Aderaldo CM, Cámara J, Garlan D. Model-based analysis of microservice resiliency patterns. In: IEEE International Conference on Software Architecture (ICSA). IEEE. 2020:114–124.
10. Jagadeesan LJ, Mendiratta VB. When Failure is (Not) an Option: Reliability Models for Microservices Architectures. In: 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE. 2020:19–24.
11. Saleh Sedghpour MR, Klein C, Tordsson J. An Empirical Study of Service Mesh Traffic Management Policies for Microservices. In: ACM/SPEC Int. Conf. Performance Engineering (ICPE). ACM. 2022:17–27.
12. Kwiatkowska M, Norman G, Parker D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In: Proc. 23rd International Conference on Computer Aided Verification (CAV'11). Springer 2011:585–591.
13. Istio.io . Istio: Connect, secure, control, and observe services. https://istio.io/; 2023. [Last accessed on August 12, 2024].
14. Resilience4j . Resilience4j: A Fault tolerance library designed for functional programming. https://github.com/resilience4j/resilience4j; 2022. [Last accessed on August 12, 2024].
15. Microsoft . The Polly Project. http://www.thepollyproject.org; 2023. [Last accessed on August 12, 2024].

16. ResiliecenBench. https://github.com/ppgia-unifor/resilience-bench; 2022. [Last accessed on August 12, 2024].

17. Scott C. Designing Resilient Systems: Circuit Breakers or Retries? (Part 1). Grab Tech Blog, https://engineering.grab.com/designing-resilient-systems-part-1; 2018. [Last accessed on August 12, 2024].

18. Scott C. Designing Resilient Systems: Circuit Breakers or Retries? (Part 2). Grab Tech Blog, https://engineering.grab.com/designing-resilient-systems-part-2; 2019. [Last accessed on August 12, 2024].

19. Tran D. Circuit Breaker and Retry. https://dantt.medium.com/circuit-breaker-and-retry-64830e71d0f6; 2018. [Last accessed on August 12, 2024].

20. Minkowski P. Circuit breaker and retries on Kubernetes with Istio and Spring Boot. Piotr's TechBlog, https://piotrminkowski.com/2020/06/03/circuit-breaker-and-retries-on-kubernetes-with-istio-and-spring-boot/; 2020. [Last accessed on August 12, 2024].

21. Aderaldo CM, Mendonça NC. How The Retry Pattern Impacts Application Performance: A Controlled Experiment. In: Proceedings of the XXXVII Brazilian Symposium on Software Engineering (SBES). ACM. 2023:47–56.

22. Khwaja S, Alshayeb M. Survey On Software Design-Pattern Specification Languages. *ACM Computing Surveys (CSUR).* 2016;49(1):1–35.

23. Microsoft Azure . Circuit Breaker pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker; 2017. [Last accessed on August 12, 2024].

24. Ibryam B. It takes more than a Circuit Breaker to create a resilient application. https://developers.redhat.com/blog/2017/05/16/it-takes-more-than-a-circuit-breaker-to-create-a-resilient-application/; 2017. [Last accessed on August 12, 2024].

25. Netflix . Hystrix: Latency and Fault Tolerance for Distributed Systems. https://github.com/Netflix/Hystrix; 2018. [Last accessed on August 12, 2024].

26. Twitter . Finagle: A fault tolerant, protocol-agnostic RPC system. https://github.com/twitter/finagle; 2022. [Last accessed on August 12, 2024].

27. Cockatiel . Cockatiel. https://npm.io/package/cockatiel; 2023. [Last accessed on August 12, 2024].

28. Envoy . Envoy Proxy. https://www.envoyproxy.io; 2023. [Last accessed on August 12, 2024].

29. *HttpBin*. https://github.com/postmanlabs/httpbin; 2011. [Last accessed on August 12, 2024].

30. Luo G, Zheng X, Liu H, et al. Verification of Microservices Using Metamorphic Testing. In: Proceedings of the 19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP). Springer. 2020:138–152.

31. Taibi D, Lenarduzzi V, Pahl C. Microservices Anti-patterns: A Taxonomy. *Microservices: Science and Engineering.* 2020:111–128.

32. Systems Engineering Body of Knowledge . System Resilience. https://www.sebokwiki.org/wiki/System_Resilience; 2020. [Last access on August 12, 2024].

33. Birolini A. *Reliability Engineering: Theory and Practice*. Springer Science & Business Media, 2013.

34. Sharma VS, Trivedi KS. Reliability and Performance of Component Based Software Systems with Restarts, Retries, Reboots and Repairs. In: 2006 17th International Symposium on Software Reliability Engineering (ISSRE). IEEE. 2006:299–310.

35. Brosch F, Buhnova B, Koziolek H, Reussner R. Reliability Prediction for Fault-Tolerant Software Architectures. In: Joint ACM SIGSOFT Conference and ACM SIGSOFT Symposium on Quality of Software Architectures (QoSA) and Architecting Critical Systems (ISARCS). ACM. 2011:75–84.

36. Brosch F, Koziolek H, Buhnova B, Reussner R. Architecture-Based Reliability Prediction with the Palladio Component Model. *IEEE Transactions on Software Engineering.* 2011;38(6):1319–1339.

37. Mirandola R, Potena P, Riccobene E, Scandurra P. A Reliability Model for Service Component Architectures. *Journal of Systems and Software.* 2014;89:109–127.

38. Düllmann TF, Hoorn vA. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In: 8th ACM/SPEC on International Conference on Performance Engineering Companion. ACM. 2017:171–172.

39. Long Z, Wu G, Chen X, Cui C, Chen W, Wei J. Fitness-guided Resilience Testing of Microservice-based Applications. In: 2020 IEEE International Conference on Web Services (ICWS). IEEE. 2020:151–158.

40. Pietrantuono R, Russo S, Guerriero A. Testing microservice architectures for operational reliability. *Software Testing, Verification and Reliability.* 2020;30(2):e1725.

41. Yin K, Du Q, Wang W, Qiu J, Xu J. On representing and eliciting resilience requirements of microservice architecture systems. *arXiv preprint arXiv:1909.13096.* 2019.

42. Heorhiadi V, Rajagopalan S, Jamjoom H, Reiter MK, Sekar V. Gremlin: Systematic Resilience Testing of Microservices. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). IEEE. 2016:57–66.

43. Joseph CT, Chandrasekaran K. Straddling the crevasse: A review of microservice software architecture foundations and recent advancements. *Software: Practice and Experience.* 2019;49(10):1448–1484.

44. Li S, Zhang H, Jia Z, et al. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology.* 2021;131:106449.

45. Montesi F, Weber J. Circuit breakers, discovery, and API gateways in microservices. *arXiv preprint arXiv:1609.05830.* 2016.

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

47. Aquino G, Queiroz R, Merrett G, Al-Hashimi B. The circuit breaker pattern targeted to future iot applications. In: International Conference on Service-Oriented Computing. Springer. 2019:390–396.

48. Khanahmadi M, Shameli-Sendi A, Jabbarifar M, Fournier Q, Dagenais M. Detection of microservice-based software anomalies based on OpenTracing in cloud. *Software: Practice and Experience.* 2023;53(8):1681–1699.

49. Jolie Language . Jolie: The first language for Microservices. https://www.jolie-lang.org/; 2020. [Last accessed on August 12, 2024].

50. Kwiatkowska M, Norman G, Parker D. Stochastic Model Checking. In: Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07). Springer. 2007:220–270.

51. Brooker M. Exponential Backoff And Jitter. AWS Architecture Blog, https://aws.amazon.com/pt/blogs/architecture/exponential-backoff-and-jitter/; 2015. [Last accessed on August 12, 2024].

52. Sedghpour MRS, Klein C, Tordsson J. Service mesh circuit breaker: From panic button to performance management tool. In: 1st Workshop on High Availability and Observability of Cloud Systems (HAOC). ACM. 2021:4–10.

53. Mendonça NC, Aderaldo CM. Towards First-Class Architectural Connectors: The Case for Self-Adaptive Service Meshes. In: 35th Brazilian Symposium on Software Engineering (SBES). ACM. 2021:404–409.

54. OpenTracing: Vendor-neutral APIs and instrumentation for distributed tracing. https://opentracing.io/; 2022. [Last accessed on August 12, 2024].

55. Google Cloud . Rate-limiting strategies and techniques. https://cloud.google.com/architecture/rate-limiting-strategies-techniques; 2019. [Last accessed on August 12, 2024].

56. Sheerin G. gRPC and Deadlines. https://grpc.io/blog/deadlines/; 2018. [Last accessed on August 12, 2024].