Breaking the Vicious Circle: Self-Adaptive Microservice Circuit Breaking and Retry

Mohammad Reza Saleh Sedghpour *, David Garlan †, Bradley Schmerl †, Cristian Klein *, Johan Tordsson *
* Department of Computing Science, Umeå University, Umeå, Sweden. {msaleh, cklein, tordsson}@cs.umu.se
† School of Computer Science, Carnegie Mellon University, Pittsburgh, USA. {garlan, schmerl}@cs.cmu.edu

***Abstract*—Microservice-based architectures consist of numerous, loosely coupled services with multiple instances. Service meshes aim to simplify traffic management and prevent microservice overload through circuit breaking and request retry mechanisms. Previous studies have demonstrated that the static configuration of these mechanisms is unfit for the dynamic environment of microservices. We conduct a sensitivity analysis to understand the impact of retrying across a wide range of scenarios. Based on the findings, we propose a retry controller that can also work with dynamically configured circuit breakers. We have empirically assessed our proposed controller in various scenarios, including transient overload and noisy neighbors while enforcing adaptive circuit breaking. The results show that our proposed controller does not deviate from a well-tuned configuration while maintaining carried response time and adapting to the changes. In comparison to the default static retry configuration that is mostly used in practice, our approach improves the carried throughput up to 12x and 32x respectively in the cases of transient overload and noisy neighbors.**

***Index Terms*—reliability, retry mechanism, circuit breaker pattern, service mesh, microservices**

I. Introduction

In recent years, microservices have emerged as a popular architectural style for designing and building software systems. This architectural style involves building applications that comprise hundreds (or even thousands) of small services with multiple instances that function as independent and cohesive processes communicating via messages [1]. Due to their perceived benefits, such as flexibility, simplicity, and scalability, over traditional approaches, there has been a surge of interest in adopting microservices in various areas, including IoT applications [2], [3], [4], big data processing [5], [6], [7], and high-performance computing (HPC) [8], [9], [10].

In a microservice architecture, network latency, resource overutilization, and service failures can cause short-term issues. These types of failures can impact the overall performance of the application. While transient failures are not exclusive to microservices, the distributed nature of their architecture makes them more prone to such failures, particularly when compared to monolithic applications [11]. Thus, managing inter-service communication and traffic in microservices architecture can be challenging due to the high degree of complexity and interdependence of the services.

Service meshes provide an infrastructure layer that integrates directly into microservices, abstracting the network and providing functionalities such as service discovery, load balancing, traffic management, security, observability, and policy enforcement [12]. Circuit breaking and retry mechanisms are among the various traffic management policies offered by service meshes. The retry mechanism improves application performance and service availability, preventing permanent failures due to transient failures of services or network congestion, while circuit breaking rejects incoming requests to maintain latency at the expense of availability, increasing the application's robustness and resilience against transient failures of dependent services [13], [14].

Despite the recent interest in service meshes both in academia and industry [15], few studies have investigated performance tuning of such resiliency patterns during (transient) failures. In a previous study, we demonstrated the difficulty of configuring the circuit breaker and retry mechanism to achieve maximum throughput while keeping the carried response time at a minimum level [13]. This challenge is particularly pronounced in dynamic application environments where changes such as increased workload, variations in the number of replicas, or the release of new replica versions with higher resource utilization are frequent. In another study, we proposed an adaptive circuit breaker mechanism that improves the throughput of a single service application, while maintaining the carried response time [14].

When using both retry and circuit breaker mechanisms together, there are complex, non-linear interactions between their configuration parameters that are often complex and counterintuitive. Therefore, it is essential to develop automated techniques for configuration optimization that accounts for these interactions. To address such challenge for all services in a microservice-based application for a transient failure scenario, this work seeks to answer two research questions:

RQ1. **How should the retry mechanism be configured to improve the resiliency of a microservice, in particular when circuit breaking is used?**

RQ2. **How is the answer to RQ1 affected for complex architectures with multiple microservices?**

To address these questions, we performed a series of experiments to extract a dataset containing multiple features (See Table II) as the input of sensitivity analysis. We conducted a sensitivity analysis using Structural Equation Modeling (SEM) to determine the impact of configuration parameters of both the circuit breaker and retry patterns, and environmental changes on the performance of the application (Section III). Based on our findings from the sensitivity analysis, we propose a con-

troller that improves throughput by dynamically adjusting the number of retry attempts and retry timeout without negatively affecting carried response time (Section IV). We evaluate the proposed controller against multiple static configurations in various experiments and traffic scenarios (Section V).

The presented experiments spanned 85 hours and involved generating more than 4.9 million requests. The proposed controller effectively mitigates overload and failures in downstream services and helps prevent the occurrence of limp lock problems [16]. In general, the proposed controller improved the carried throughput throughout the experiments (10 minutes) up to 12x and 32x in comparison to the static default retry configuration that is mostly used in practice in case of transient overload and noisy neighbors respectively while maintaining the carried response time. Our code and tools[1] are publicly available.

II. Background and Related works

In this section, we discuss previous studies relevant to the work presented herein and our previous work on adaptive circuit-breaking mechanism.

A. Traffic engineering

1) Network level traffic engineering:

a) Dropping requests: Dropping requests as a means of managing system performance is a technique that has been in use for many years. For example, the random early detection (RED) algorithm has been used to manage congestion and prevent packet loss [17]. RED is designed to randomly drop packets before the router's buffer becomes full, thereby reducing the likelihood of congestion and improving overall network performance. Similar load-shedding techniques have been applied to data stream processing systems to improve system scalability and maintain response times under overloads [18], [19]. The idea behind load shedding is to selectively drop requests or reduce the quality of service for low-priority requests to prioritize more important ones. Several load-shedding techniques have been proposed in the literature, including probabilistic load shedding, feedback control, and adaptive filtering. These techniques are designed to dynamically adjust the load-shedding policy based on system workload and performance metrics, thereby maximizing system performance while maintaining a high level of service quality.

b) Retransmission (retry): In the early days of packet network intercommunications, the idea of retransmission (retrying) a failed or not delivered request was essential for ensuring reliable data transmission. This concept was developed to overcome the limitations of early networking technology, which suffered from poor reliability due to network congestion, hardware failures, and other factors [20].

2) Application level traffic engineering:

a) Circuit breaker: The circuit breaker design pattern is extensively employed in software development to detect failures and encapsulate the process of avoiding recurring failures in situations such as maintenance, temporary outages of external systems, or unforeseen system issues. This facilitates quicker failure of the system. This pattern was officially introduced in [21]. From an industrial perspective, the earliest library that implemented the circuit breaker pattern was Hystrix. It entails enveloping Java code in a mechanism that can be regulated by the circuit breaker [22]. Surendro and Sunindyo present a comprehensive summary of current research on circuit breakers, maps the subject of research, and identify potential directions for future research [23].

Several studies focus on different implementations of such patterns. For instance, Montesi et al. distinguish three discrete circuit breaker patterns [24]. The first one is known as the *client-side circuit breaker* pattern, where each client features a distinct circuit breaker that intercepts calls to all external services the client may invoke. The second one is the *service-side circuit breaker* pattern, where all client requests received by a service are initially processed by an internal circuit breaker, which determines whether the request should be processed or not. The third pattern is called the *proxy circuit breaker* pattern, in which circuit breakers are employed in a proxy service that acts as an intermediary between clients and services and manages all incoming and outgoing messages. The latter category includes service mesh-based technologies as they introduce a sidecar proxy.

Some studies model the behavior of circuit breaker pattern alongside other resiliency patterns for a single service application [25], [26], while another work illustrated the advantages of employing Envoy as a sidecar proxy to enhance the resiliency of microservices via mechanisms such as retry, circuit breaking patterns, and rate limiting [27].

Finally, many use cases have explored the benefits of service meshes, including research conducted in the context of the 5G core [28], [29], as well as efforts to enhance scheduling algorithms [30], [31].

b) Retry mechanism: Retry mechanisms are a well-known strategy in distributed systems that generate a new request in response to a failed initial request, either to the same instance of the called service or to a different one [32]. This process can occur multiple times, with the maximum number of attempts specified by a retry attempt parameter and the time interval between consecutive retries set by a retry timeout.

Heorhiadi et al. introduced Gremlin, a framework that enables systematic testing of the failure-handling capabilities of microservices [33]. Gremlin facilitates the design of tests that incorporate resiliency patterns, including retry mechanisms, thereby simplifying the task for the operator. In previous work, we conducted an empirical investigation to uncover the effects of various resiliency patterns, including circuit breaking and retry mechanisms, in different scenarios [13]. Raj Karn et al. emphasized the utilization of resiliency patterns within the Istio service mesh to enable automated testing and enhancement of the resiliency of microservices [34].

[1]https://doi.org/10.5281/zenodo.8189211

Listing 1 Overview of the study process.

```
1-  Conduct a series of experiments with
    the experimental space in Table I to
    extract a dataset.
2-  Perform the SEM on the extracted
    dataset in the previous step to explore
    the relation between variables.
3-  Design and implement the retry
    controller based on the results
    from step 3.
4-  Evaluate the controller.
```

B. Adaptive circuit breaking

In a previous study, we proposed an adaptive circuit breaker mechanism that improves application throughput without compromising carried response time [14]. This controller considers multiple performance metrics to monitor the system's behavior and adjust the circuit breaker configuration accordingly. Among these metrics, the carried response time of the service is crucial as it indicates the current system load and helps identify any potential bottlenecks that may lead to slow response times. Additionally, tracking the current circuit breaker configuration enables the identification of the number of pending requests waiting to be processed by the service and provides a historical record of the configuration. By leveraging these performance metrics, the circuit breaker controller can adjust the circuit breaker thresholds to ensure reliability and availability while preventing overloading or underutilization.

Algorithm 1 uses the smoothened historic ratio of carried response time and circuit breaker configuration to compute a new circuit breaker configuration based on the target carried response time. By using a smoothed average of the response time to set the circuit breaker threshold, the algorithm ensures that the circuit breaker is neither too restrictive, which can lead to false positives, nor too lenient, which can lead to overload situations. This approach allows the algorithm to adapt the circuit breaker threshold to the service's current performance and ensure responsiveness and availability under varying load conditions. However, the performance of the circuit breaker can be influenced by parameter selection, such as the target carried response time and the smoothing factor p, which should be carefully considered when designing the adaptive controller. The p parameter can affect the controller's responsiveness by determining the weight of current measurements compared to previous measurements. In the current study, we employ this controller to maintain the latency of the services.

While there has been an increase in academic attention toward microservices, there are few research studies that have focused on microservice resiliency patterns. This study distinguishes itself from prior work in microservice resiliency by presenting an adaptive controller for retry mechanisms in the presence of adaptive circuit breaking. The development of this controller was informed by a sensitivity analysis that drew upon empirical data regarding various circuit breaker patterns and retry mechanisms.

III. Methodology

This section discusses the methodology used in this paper. It begins with a discussion of the overview of the study process. Then the experimental methods are discussed. We then present the sensitivity analysis and its results.

Large microservice-based applications are distinguished by their dynamic attributes, implying that services and infrastructure can undergo frequent changes. Consequently, the interdependencies between microservices may change rapidly, leading to complex and unpredictable transient failure patterns. Microservice architectures must be capable of scaling and adjusting to changing circumstances, which presents a fundamental challenge in such dynamic environments. In this context, we conduct a sensitivity analysis on the retry mechanism and circuit breaker pattern to determine the impact of each configuration parameter. Subsequently, we utilize the outcomes of the sensitivity analysis to devise the controller, which is later assessed through an additional set of experiments. (See Listing 1 for the overview of the study process).

A. Experimental method

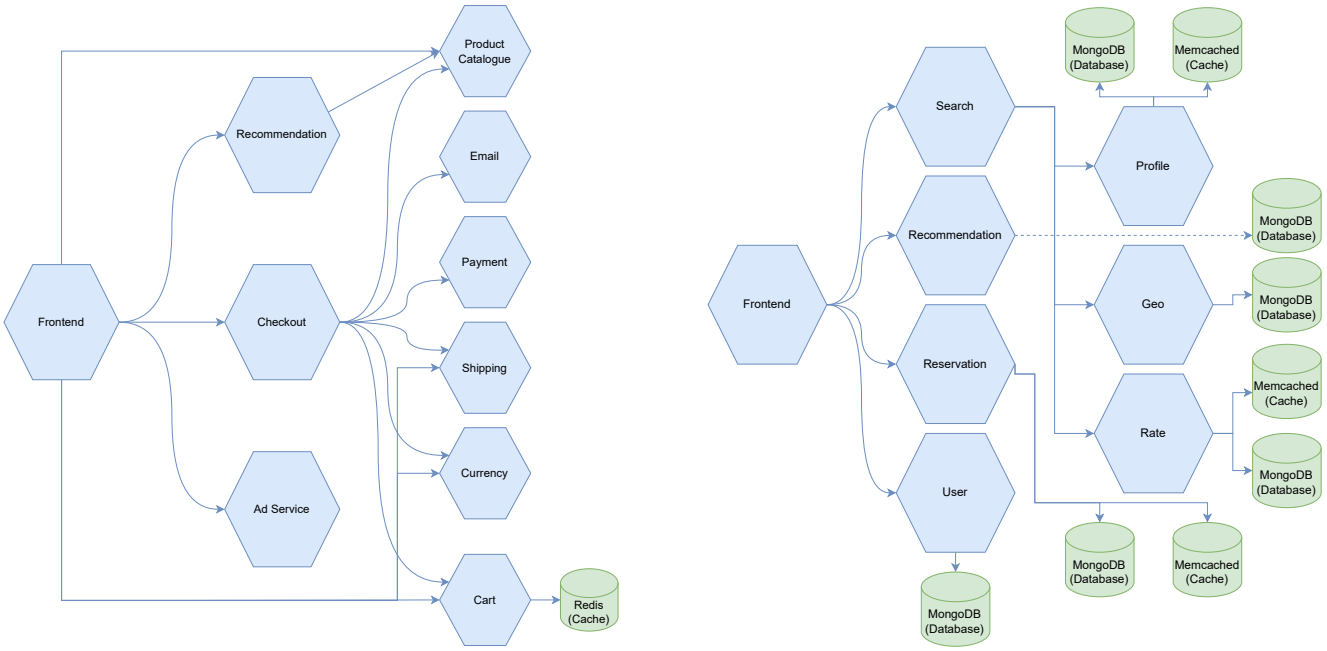
1) Subjects and experiments: In this study, we utilized the two most well-known microservice benchmarks, Online Boutique and DeathStarBench, as the sample application for all experiments. Online Boutique is a web-based e-commerce app that consists of 11 microservices, allowing users to browse items, add them to a cart, and purchase them [35]. The architecture of this application is depicted in Fig. 1(a). Our focus was on one of the application's endpoints, *frontend/cart*, which triggers a chain of three microservices. The frontend service receives the initial request and then sends one request to the recommendation service, while the product catalog service receives four requests. Additionally, the recommendation service sends a request to the product catalog service. Thus, six internal requests are necessary to provide a successful response to the external client.

The DeathStarBench is a benchmark suite designed to evaluate the performance of datacenter-scale systems [36]. One of the applications in the suite is the Hotel Reservation, which models a hotel reservation system. For this application, our focus was on *frontend/* endpoint, which triggers a chain of 5 services excluding the databases. The frontend service receives the initial requests and then sends one request to the search service. The search service then calls Geo service to search for nearby hotels. The Geo service responds with three hotels initially and then the search service sends three requests to profile and rate services for all three hotels. Thus, eight internal requests are necessary to provide a successful response to the external client.

To conduct the experiments, we created a tool that deploys the HTTPmon load generator and applications, along with all controllers for each service on a repeated basis.

Algorithm 1: Adaptive circuit breaking design [14]

Result: Circuit breaker configuration
Parameters: $targetCarriedResponseTime$, p;
while *Controller is running* **do**
  wait for observation interval (5 seconds);
  retrieve $currentCarriedResponseTime$, $currentCircuitBreakerConfiguration$;
  $notSmoothAlpha = (currentCarriedResponseTime/max(currentCircuitBreakerConfiguration, 1))$;
  $smoothAlpha = (p * smoothAlpha) + (1 - p) * notSmoothAlpha$;
  $newCircuitBreakerConfiguration = targetCarriedResponseTime/smoothAlpha$;
  $setnewCircuitBreakerConfiguration$;
end



(a) The Online Boutique application is composed of 11 microservices that are developed in various languages.

(b) The Hotel Reservation (DeathStarBench) is composed of 17 microservices that are developed in GoLang.

Fig. 1. The architecture of sample applications used in this study.

TABLE I
EXPERIMENTAL SPACE: THE SELECTED VALUES FOR DIFFERENT PARAMETERS.

Parameter	Selected Values
Traffic ratio	0.6, 0.8, 1.0, 1.2, and 1.4
Max requests	1, 5, 10, 20, 50, 500, and 1000
Retry attempts	1, 2, 5, 10, and 20
Timeout per attempt	1ms, 5ms, 10ms, 50ms, and 100ms

2) Testbed setup: Eight bare metal machines running Ubuntu 20.04 LTS constitute our testbed. Using Kubernetes 1.23.7, Docker 20.10.16 [37], and Istio 1.14.0 [38], we created a cluster that comprises of one control plane node and seven worker nodes. Each worker node possesses two Intel Xeon E5430 2.66 GHz CPUs (with four cores and hyper-threading enabled), 16GB of RAM, and a 256 GB NVMe drive.

3) Tools and instrumentation:

a) HTTPmon: An open-source tool named HTTPmon [39] is utilized to produce HTTP traffic and simulate load. This tool allows customization of parameters such as the number of concurrent requests, duration, and think time. To model short-lived sessions with concurrent clients, this tool employs an open model.

b) Monitoring stack: Resource utilization metrics are collected using the Prometheus 2.31.1 [40]. In addition, traffic performance metrics are gathered using Istio sidecar proxies' monitoring capabilities along with Prometheus.

c) Important definitions: Here we present the definitions we used in our experiments.

Capacity: This refers to the maximum number of successful requests per second that the system can handle while keeping the response time below a threshold of *100 ms*.

Latency: This refers to the carried response time of requests at the 95^{th} percentile, which is considered a good indicator of

TABLE II
THE DEFINITION OF EACH CONFIGURATION PARAMETER AND METRIC.

Feature/Column	Type	Description
traffic	Configuration Parameter	The incoming *traffic* to the application.
retryAttempt	Configuration Parameter	The maximum number of times the sidecar proxy attempts to connect to a service if the initial call fails.
retryPerTryTimeout	Configuration Parameter	The interval between retries when attempting to connect to a service.
circuitBreakerMaxRequests	Configuration Parameter	It specifies the size of the queue in the circuit breaker's configuration.
successRate	Monitored metric	It shows the number of successful requests.
failureRate	Monitored metric	It shows the number of failed requests.
circuitBrokenRate	Monitored metric	It shows the number of circuit broken requests (rejected due to circuit breaker configuration).
carriedResponseTime	Monitored metric	It shows the 95th percentile carried response time.
retryRate	Monitored metric	It shows the number of retried requests.

user experience [41] while also being less affected by outliers than higher percentiles.

d) Important tuning parameters: We utilized Istio to enforce and evaluate the configuration of circuit breaker and retry mechanisms. Here we present the parameters we used for the circuit breaker and retry mechanism.

circuitBreakerMaxRequests: This parameter represents the maximum number of requests per second that can be transmitted through the circuit breaker to the service [42].

retryAttempt: This parameter specifies the number of times a sidecar proxy should retry establishing a connection to a service in case the initial attempt fails [43].

retryPerTryTimeout: This parameter sets the duration of the timeout for each retry attempt, including the initial attempt.

B. Sensitivity analysis

To answer RQ1, we perform a sensitivity analysis. It enables us to determine the impact of various factors on the performance of services in a microservice application. Specifically, we focus on the effects of circuit breakers, retry patterns and changes in environmental parameters such as traffic on the performance of services.

To gather data for the sensitivity analysis, we set up various values for different parameters in the load generator, retry mechanism, and circuit breaker pattern. These selected values are presented in Table I. Each experiment spanned five minutes, with the first minute of each experiment serving as a warm-up phase. Throughout the experiments, we monitored the throughput and latency of each service within the Online Boutique and Hotel Reservation.

1) Model specification: After completing the experiments, we assembled a dataset containing multiple features, as outlined in Table II. Subsequently, we employ covariance-based structural equation modeling (CB-SEM) as a statistical technique to analyze the relationships between the input parameters and measured values. By utilizing the CB-SEM, we can analyze multiple dependencies concurrently and model the causal relationships between variables while accounting for measurement errors and the covariance structure of the data [44], [45]. It also enables us to estimate latent variables, which are unobservable variables that are inferred from observable indicators [46].

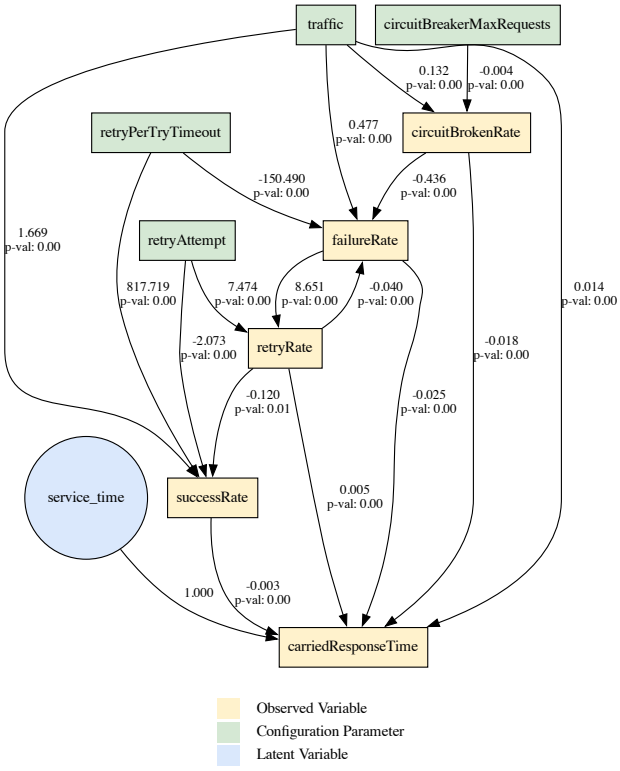


Fig. 2. The structural model represents the relationship between configured parameters (light green) and measured throughput and latency (light yellow).

In this study, a structural model is employed for SEM analysis to test the proposed relationships between variables and latent variables (The latent variable is extracted using principal component analysis). The model verifies the conceptual representation of these relationships and determines whether the proposed model is suitable for representing them.

2) Model analysis: The initial idea is to see how input parameters (*traffic*,*circuitBreakerMaxRequests*, *retryAttempt*, and *retryPerTryTimeout*) impact the throughput and latency (*successRate*, *failureRate*, *circuitBrokenRate*, *retryRate*, and *carriedResponseTime*). The definition of each parameter is shown in Table II. To this end, we tried various models, and the best model/relationship with the minimum error is reported in

TABLE III
DESCRIPTIVE INFORMATION OF MODEL FIT INDEX [47].

Statistic	Recommended Value	Obtained Value
Degrees of Freedom(DoF)	-	22
χ^2	-	33.398
χ^2/DoF	<3.00	1.518
$\chi^2 - p$	>0.05	0.056
Comparative Fit Index (CFI)	>0.90	0.984
Goodness of Fit index (GFI)	>0.90	0.957
Adjusted Goodness of Fit index (AGFI)	>0.90	0.922
Normed Fit Index (NFI)	>0.90	0.957
Tucker-Lewis Index (TLI)	>0.90	0.972
Root Mean Square Error of Approximation (RMSEA)	<0.08	0.051

Fig. 2 and Table. III. Fig. 2 shows that the model includes five endogenous variables (output variables) and four exogenous variables (input variables) and two latent variables for Online Boutique. The relationships in the model include direct effects. The direct effects are represented by the arrows. For example, the arrow from variable *circuitBrokenRate* is regressed onto *traffic* with an estimated coefficient of *0.132*. This means that a one-unit increase in *traffic* is associated with a *0.132* unit increase in *circuitBrokenRate*, holding all other variables in the model constant. The most important point about this model is that it shows the direction of the impact of tuning parameters on performance metrics. This means that we can use SEM to determine how changes in a given parameter affect the overall performance of the model.

However, it is important to note that while SEM can provide insight into the direction of impact, the magnitude of the effect depends on the application and other factors. For instance, we also extracted the best-fit model for all services in the Hotel Reservation application. The results showed that the models had consistent and similar directional relationships between the variables, although the magnitudes of these relationships differed between the two applications. These findings suggest that the SEM models used in this study are generalizable, and can be applied to other microservice applications to gain insights into the impact of these variables on service performance. With this in mind, the key points to consider for the retry mechanism based on Fig. 2 are:

- Increasing the number of retry attempts, referred to as the *retryAttempt*, is found to have a detrimental impact on the *successRate* and a beneficial impact on the *retryRate*.
- Increasing the value of *retryPerTryTimeout* has a notable negative impact on the *failureRate*, but a positive effect on the *successRate*.
- To counteract the impact of *retryRate* on *successRate*, which is caused by the positive effect of *failureRate* on *retryRate*, it is recommended to configure the *retryAttempt* in the opposite direction of the *failureRate*.
- If the *failureRate* is on the rise, it would be logical to increase the *retryPerTryTimeout* to counteract the negative influence of *failureRate* on both *successRate* (indirectly) and *carriedResponseTime* (directly).

With an understanding of the effects of input parameters on system outputs, the necessary information has been obtained to enable the design of a controller that can maintain input parameters at the desired target values.

IV. CONTROLLER DESIGN

While circuit breakers have shown to be effective in improving the reliability and availability of microservices (See Section II-B), simply employing them may not be enough to improve the overall throughput of the system. In fact, in a complex microservice-based architecture, employing a circuit breaker may even lead to decreased throughput due to its restrictive nature. To address this issue, a retry controller can be employed to improve the throughput by allowing failed requests to be retried in a controlled and efficient manner. In this context, we present a retry controller that works in conjunction with a circuit breaker controller to enhance the performance of microservice-based architectures. Fig. 3 shows the overview of the flows between components in our study.

To design an effective controller for the retry mechanism in a microservice-based architecture, several critical performance metrics must be considered, as detailed in Section III-B. While a circuit breaker controller can help maintain system reliability and availability by adjusting the circuit breaker threshold based on observed performance metrics, simply employing a circuit breaker controller may not be enough to improve the throughput in a complex microservice-based architecture. Instead, a retry controller must be employed to optimize the performance of the service. One of the key metrics that a retry controller should consider is the service carried response time, which has a direct impact on the user experience. Monitoring the carried response time allows the controller to adjust the retry configuration and prevent users from experiencing long wait times or timeouts due to failed requests. Additionally, the number of failed requests and the circuit breaker state must also be taken into account to avoid overloading the service with excessive retry requests. By dynamically adjusting the retry configuration based on these performance metrics, the controller can optimize the performance, reliability, and availability of the service.

Our control design draws inspiration from the additive increase, multiplicative decrease algorithm, which was effectively employed for Transmission Control Protocol (TCP) congestion control. We believe that incorporating proven techniques that have been used for over three decades and are easy to implement, can lead to a more robust solution [48].

The presented Algorithm 2 outlines an adaptive retry mechanism controller that aims to configure the retry timeout and retry attempt based on the observed service performance. The controller takes in the target carried response time as input and proceeds to a loop that waits for an observation interval of 5 seconds. It is because the scrape interval in our monitoring stack (Prometheus) is configured to be 5 seconds which is the minimum possible configuration. This timing affects the responsiveness of our controller to the changes. Within this loop, the current number of failed requests and the current number of circuit-broken requests are retrieved and treated as not responded requests. The controller also obtains the current carried response time. If the current carried response time exceeds the target carried response time or if there are unan-

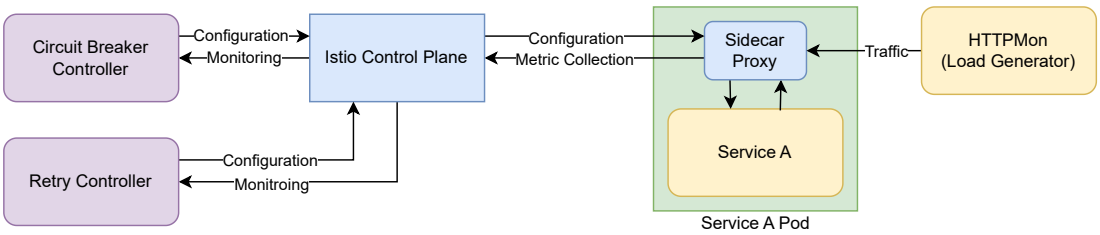


Fig. 3. The overview of different flows between proposed controllers and applications/services.

Algorithm 2: Adaptive retry mechanism design considering the circuit breaker configuration

Result: Retry mechanism configuration
Parameters: $targetCarriedResponseTime$;
// As the minimum possible retry timeout is 1ms
$maximumRetryAttempt = targetCarriedResponseTime$;
while *Service is running* **do**
  wait for observation interval (5 seconds);
  retrieve $currentFailedRequests$, $currentCircuitBrokenRequests$, $currentCarriedResponseTime$;
  $notRespondedRequests = currentFailedRequests + currentCircuitBrokenRequests$;
  if $(currentCarriedResponseTime > targetCarriedResponseTime) or (notRespondedRequests > 0)$ **then**
    $retryAttempt = int(retryAttemp/2)$;
  else
    $retryAttempt = retryAttemp + 1$;
  end
  $retryPerTryTimeout = max(1, int(targetCarriedResponseTime/retryAttempt))$;
  set $retryAttempt$,$retryPerTryTimeout$;
end

swered requests (*notRespondedRequests*), the controller cuts the retry attempt by half, implementing an exponential backoff for retry attempts. Conversely, if the current carried response time is less than or equal to the target carried response time, the controller increases the retry attempt by one. The retry timeout is then calculated as the target carried response time divided by the retry attempt, with a minimum retry timeout of *1 ms*. The maximum retry attempt initially matches the target response time. Finally, the controller sets the retry attempt and retry timeout. By dynamically adjusting the retry configuration based on important performance metrics such as the carried response time and the number of unanswered requests, the controller can optimize the performance of the application.

V. Performance Evaluation

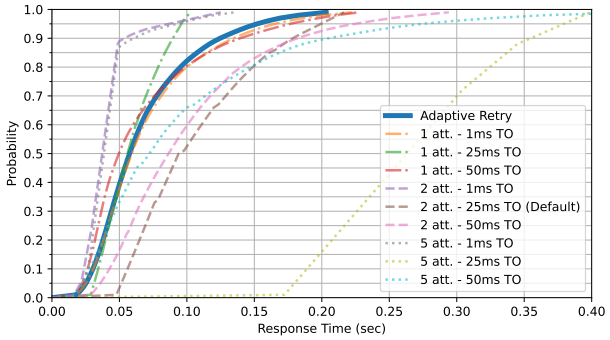
We conducted a series of experiments to assess the effectiveness of the suggested controller. During these experiments, we enforced the circuit breaker controller for all services and compared the retry controller to static retry configurations. To ensure that the proposed retry controller architecture can handle unexpected or abnormal situations that may occur in a production environment, we evaluated it in two different scenarios, transient failures, and noisy neighbors [49]. Transient overload refers to a temporary surge in traffic that exceeds the capacity of a service, which can lead to degradation in performance or even service failure. Noisy neighbors, on the other hand, refer to situations where a service is impacted by the resource usage of other services, which can also result in degraded performance or failure.

A. Transient overload

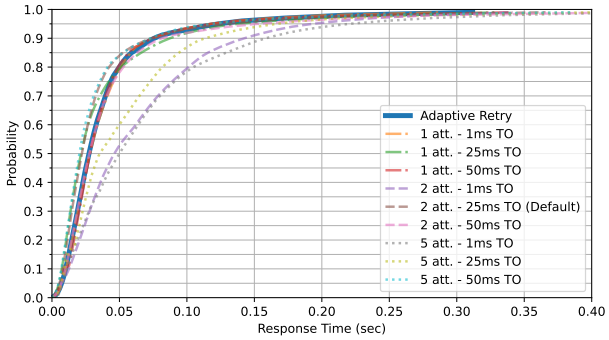
As microservices are dynamic systems, the controllers for circuit breaker pattern (See Section II-B) and retry mechanism must be able to adapt to changes to maintain resiliency. To test the effectiveness of our proposed controller for the retry mechanism, we present a series of experiments with the incoming loads that include spikes in traffic. These experiments were conducted to evaluate the performance of our controller under conditions that more closely resemble real-world scenarios.

In these experiments, the incoming traffic was set at 60% of capacity most of the time, jumping directly to 140% of capacity for 10 seconds at 50 seconds intervals. Both controllers for the retry mechanism and circuit breaker pattern were enforced for each service. We repeated each experiment 10 times but as there were no significant differences between the results of replicated experiments we only discuss the results of individual experiments. Levene's test is used to compare similarities across repeated experiments by assessing the equality of variances [50]. (The achieved p-values are in the range of 0.72 to 0.78, which is higher than 0.05.)

The results, as shown in Fig. 4 and Table IV, indicate that the adaptive retry mechanism is more effective than static



(a) Cumulative density function of carried response times for Online Boutique.



(b) Cumulative density function of carried response times for Hotel Reservation.

Fig. 4. Performance of both Online Boutique (first column) and Hotel Reservation (second column) in terms of carried response time (a, b) when there are load spikes (transient overload) and adaptive circuit breaking is enforced.

TABLE IV
THE ACHIEVED THROUGHPUT REGARDING THE RATIO OF SUCCESSFUL REQUESTS TO TOTAL REQUESTS FOR BOTH APPLICATIONS WHEN THERE ARE LOAD SPIKES (TRANSIENT OVERLOAD) AND ADAPTIVE CIRCUIT BREAKING IS ENFORCED.

Retry Configuration	**Carried Throughput**	
	Online Boutique	**Hotel Reservation**
Adaptive Retry	**53.27%**	**98.52%**
1 attempt with *1 ms* timeout	52.15%	97.61%
1 attempt with *25 ms* timeout	2.00%	82.16%
1 attempt with *50 ms* timeout	45.56%	93.78%
2 attempts with *1 ms* timeout	0.65%	96.82%
2 attempts with *25 ms* timeout (Default Configuration)	4.41%	59.81%
2 attempts with *50 ms* timeout	47.55%	98.51%
5 attempts with *1 ms* timeout	0.57%	97.79%
5 attempts with *25 ms* timeout	0.63%	98.18%
5 attempts with *50 ms* timeout	51.00%	98.41%

retry configurations in improving throughput with 53.27% and 98.52% for Online Boutique and Hotel Reservation in DeathStarBench respectively while maintaining carried response times during load spikes. Moreover, the static retry configurations tested included the default retry mechanism (most are used with 2 retry attempts and 25 milliseconds of timeout) and additionally, we explored several fine-tuned retry configurations which were customized for specific applications. These fine-tuned configurations involved adjusting the number of retry attempts, timeout values, and other relevant parameters to optimize the retry behavior for different use cases. Despite the optimization efforts, the adaptive retry mechanism still outperformed the static configurations.

Furthermore, there is a significant difference between the success rate of the Online Boutique and Hotel Reservation in DeathStarBench. It is because of their different implementation strategies. Despite the difference, the proposed adaptive retry mechanism, improved the throughput while maintaining the response time in comparison to all static retry configurations.

B. Noisy neighbours

In a microservice architecture, the term *noisy neighbor* is used to refer to a neighboring service that is causing poor performance and negatively impacting other services within the application. When encountering a noisy neighbor, a common strategy is to retry failed requests. With this approach, if a request fails due to the noisy neighbor, the system can attempt to redirect the request to another instance of the same service. This can potentially improve the overall performance of the system by reducing the number of failed requests and minimizing the impact of the noisy neighbor on other services.

In order to replicate the effects of noisy neighbors in the Online Boutique application, we introduced three versions of the *productCatalogue* service. In order to simulate a scenario where one version of the application (v1) is resource-constrained, we intentionally limited the resources available to v1 to only one-third of the resources available to the other versions. This was achieved by applying resource limits in Kubernetes deployments. By creating a bottleneck in this way, we aimed to evaluate the performance and resilience of v1 under conditions of resource scarcity. In these experiments, the incoming traffic was set at 100% of capacity most of the time. The circuit breaker controller was enforced for each service in the application. We tested different scenarios where either a static retry configuration or the proposed retry controller was enforced for each service. We repeated each experiment 10 times but as there were no significant differences between the results of replicate experiments we only discussed the results of individual experiments. Levene's test is used to compare similarities across repeated experiments by assessing the equality of variances (The achieved p-values are in the range of 0.68 to 0.83, which is higher than 0.05.).

The carried response times and carried throughput ratio of these experiments are shown in Fig. 5 and Table V. The results show that the proposed retry mechanism improves throughput (86.73%) while maintaining the response time at the minimum possible when there is a noisy version of the *Product Catalogue* service and the circuit breaker controller

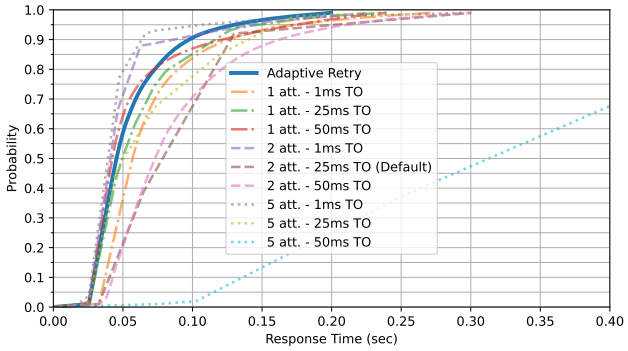


Fig. 5. Performance of the application in terms of cumulative density function of carried response time when there is a noisy version of *product catalogue* service with limited resources and adaptive circuit breaking is enforced.

TABLE V
THE ACHIEVED THROUGHPUT REGARDING THE RATIO OF SUCCESSFUL REQUESTS TO TOTAL REQUESTS FOR ONLINE BOUTIQUE WHEN THERE IS A NOISY VERSION OF *product catalogue* SERVICE WITH LIMITED RESOURCES AND ADAPTIVE CIRCUIT BREAKING IS ENFORCED.

Retry Configuration	Carried Throughput
Adaptive Retry	**86.73%**
1 attempt with *1 ms* timeout	86.20%
1 attempt with *25 ms* timeout	10.71%
1 attempt with *50 ms* timeout	52.38%
2 attempts with *1 ms* timeout	1.65%
2 attempts with *25 ms* timeout (Default Configuration)	2.62%
2 attempts with *50 ms* timeout	48.81%
5 attempts with *1 ms* timeout	0.78%
5 attempts with *25 ms* timeout	0.74%
5 attempts with *50 ms* timeout	0.59%

is enforced.

To sum up, our proposed controller yielded up to 12x and 32x improvement in carried throughput during the 10-minute experiments, as compared to the static default retry configuration that is mostly used in practice, when faced with transient overload and noisy neighbors, respectively. This improvement was achieved while maintaining the carried response time. Additionally, the performance of our controller does not deviate from the well-tuned retry configuration.

VI. Threats to Validity

Despite the careful research design, empirical studies such as the one presented here inevitably have limitations and factors that can potentially undermine their validity.

A. External validity

The research was conducted on specific versions and configurations of the study subject – in particular Istio. It is import to acknowledge that the Istio data-plane has recently seen a lot of change and can be configured in three main flavours: side-car, daemonset and ambient (eBPF-based). Variations in performance between these data-planes could affect the generalizability of the findings. While we cannot directly apply our results to other data planes or future versions thereof, given that we focus on their functionality and not on their exact implementation, we anticipate that comparable outcomes could be seen even if alternate data-planes were utilized. Additionally, the scale of the microservices under investigation may be perceived as limited in complexity, potentially raising concerns about the extent to which the findings can be generalized to more complex systems. To address this threat, future research should encompass a broader range of microservices, ideally involving hundreds of instances by using tools such as HydraGen [51] in real-world cloud environments, to ensure the results' applicability to real-world scenarios where parameter space mapping becomes inherently challenging.

B. Internal validity

Internal validity is inevitably affected by the fact that some design decisions must be made when defining the configuration values to test. Although empirical data analysis practices suggest that there are no significant differences in the behavior of the entire stack when using different values, it is impossible to definitively prove this.

A potential internal validity threat is that a significant portion of the experiments was performed on bare-metal servers, which might have impacted the obtained results. Nevertheless, we deem it unlikely that the specific hardware chosen for the study would compromise the validity of the findings. Virtualized platforms are usually utilized to deploy microservices, which may have performance variations from bare-metal platforms due to the presence of an additional scheduling layer and shared hardware resources. Nevertheless, we do not foresee that the overall validity of the outcomes would be impacted by employing a bare-metal platform. https://www.overleaf.com/project/63e627d7349cdf6c63d8c2f1 Another threat to validity is a large number of traffic scenarios. Due to space limitations, we can only report on a handful of traffic scenarios, which helped increase the breadth of conditions tested and improved the reliability and robustness of our findings. Additionally, we conducted experiments using different values to further enhance the validity of our results. Despite these efforts, we did not observe any meaningful differences from the findings presented in this paper.

VII. Discussion

RQ1 asks *"How should the retry mechanism be configured to improve the resiliency of a microservice, in particular when circuit breaking is used? "*

To understand the magnitude of the effect of different tuning parameters, we conducted a sensitivity analysis using SEM. Using SEM, we were able to derive the relationship between the configured parameters of the circuit breaker pattern and retry mechanism and the measured throughput and latency in a sample microservice application.

The study revealed several important points about the retry mechanism. Increasing the number of retry attempts, or *retryAttempt*, had a negative impact on the *successRate*, while positively affecting the *retryRate*. Similarly, increasing the value of *retryPerTryTimeout* had a negative impact on

the *failureRate*, but a positive effect on the *successRate*. To counteract the impact of *retryRate* on *successRate*, which is caused by the positive effect of *failureRate* on *retryRate*, it is recommended to configure the *retryAttempt* inversely proportional to the *failureRate*. Finally, if the *failureRate* is increasing, it is advisable to increase the *retryPerTryTimeout* to counteract the negative impact of *failureRate* on both the *successRate* (indirectly) and the *carriedResponseTime* (directly).

According to the SEM analysis in a dynamic environment, adjusting the *retryAttempt* in the opposite direction of the *failureRate* is suggested to mitigate the impact of *retryRate* on *successRate*, which is caused by the positive effect of *failureRate* on *retryRate*. Additionally, in case of an increasing *failureRate*, increasing the *retryPerTryTimeout* is recommended to counteract its negative impact on both the *successRate* (indirectly) and the *carriedResponseTime* (directly).

RQ2 asks *"How is the answer to RQ1 affected for complex architectures with multiple microservices?"*

Based on the findings from answering RQ1, we designed an adaptive controller that can cope with both transient overload and the existence of noisy neighbors even when there is an adaptive circuit breaker mechanism.

The results obtained confirm those reported in [13]. Specifically, our observations indicate that the circuit breaker controller configures a circuit breaker with more restrictive settings to operate at services that are closer to the user or client-facing interface, thereby enhancing the user experience. We also noted that the retry controller allows for a limited number of retry attempts and sets a retry timeout that strikes a balance between being too low or too high to maximize the throughput without negatively affecting response times. Furthermore, when faced with challenges such as noisy neighbors or transient overloads, the retry controller adjusts both parameters to effectively address the situation.

VIII. Conclusion

In summary, this study performs a sensitivity analysis using SEM to determine the impact of configuration parameters of both circuit breaker and retry pattern, and environmental changes on the performance of the application. By using the lessons learned from sensitivity analysis, this study utilizes a control theory approach to design a retry mechanism to enhance application performance, prevent transient failures and overload, and maintain carried response times. The aim is to overcome the issues associated with existing approaches that rely on static configurations by developing a dynamic retry mechanism. An adaptive controller is suggested, and its effectiveness is evaluated through experiments that span over 85 hours and involve more than 4.9 million requests. The evaluations are conducted under various conditions utilizing diverse static configurations. The findings reveal that the proposed method can successfully adjust the retry configuration to remediate the performance and latency of the application in case of transient overload and noisy neighbors when the adaptive circuit breaker is enforced.

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*. Cham: Springer International Publishing, 2017, pp. 195–216.

[2] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic Computing: A New Paradigm for Edge/Cloud Integration," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 76–83, 11 2016.

[3] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a Smart City Internet of Things Platform with Microservice Architecture," in *2015 3rd International Conference on Future Internet of Things and Cloud*. IEEE, 8 2015, pp. 25–30.

[4] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 9 2016, pp. 1–6.

[5] J. Gao, W. Li, Z. Zhao, and Y. Han, "Provisioning big data applications as services on containerised cloud: a microservices-based approach," *International Journal of Services Technology and Management*, vol. 26, no. 2/3, p. 167, 2020.

[6] J. L. Schnase, D. Q. Duffy, G. S. Tamkin, D. Nadeau, J. H. Thompson, C. M. Grieg, M. A. McInerney, and W. P. Webster, "MERRA Analytic Services: Meeting the Big Data challenges of climate science through cloud-enabled Climate Analytics-as-a-Service," *Computers, Environment and Urban Systems*, vol. 61, pp. 198–211, 1 2017.

[7] R. Laigner, M. Kalinowski, P. Diniz, L. Barros, C. Cassino, M. Lemos, D. Arruda, S. Lifschitz, and Y. Zhou, "From a Monolithic Big Data System to a Microservices Event-Driven Architecture," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 8 2020, pp. 213–220.

[8] F. Z. Benchara, M. Youssfi, O. Bouattane, and H. Ouajji, "A new efficient distributed computing middleware based on cloud microservices for HPC," in *2016 5th International Conference on Multimedia Computing and Systems (ICMCS)*. IEEE, 9 2016, pp. 354–359.

[9] C. de Alfonso, A. Calatrava, and G. Moltó, "Container-based virtual elastic clusters," *Journal of Systems and Software*, vol. 127, pp. 1–11, 5 2017.

[10] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, G. Morgan, and R. Ranjan, "A study on the evaluation of HPC microservices in containerized environment," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 7, pp. 1–1, 4 2021.

[11] H. Knoche and W. Hasselbring, "Using microservices for legacy software modernization," *IEEE Software*, vol. 35, no. 3, pp. 44–49, 2018.

[12] K. Ponomarev, "Attribute-based access control in service mesh," in *Dynamics '19*. Russia: IEEE, 2019, pp. 1–4.

[13] M. R. Saleh Sedghpour, C. Klein, and J. Tordsson, "An empirical study of service mesh traffic management policies for microservices," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 17–27. [Online]. Available: https://doi.org/10.1145/3489525.3511686

[14] M. Saleh Sedghpour, C. Klein, and J. Tordsson, "Service mesh circuit breaker: From panic button to performance management tool," in *HAOC '21*. USA: ACM, 2021, p. 4–10.

[15] M. R. Saleh Sedghpour and P. Townend, "Service mesh and ebpf-powered microservices: A survey and future directions," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022, pp. 176–184.

[16] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi, "Limplock: Understanding the impact of limpware on scale-out cloud systems," in *Proceedings of the 4th annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2523616.2523627

[17] B. Birscoe and J. Manner, " Byte and Packet Congestion Notification," Internet Requests for Comments, RFC Editor, RFC 7141, February 2014. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7141.txt

[18] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao, "Load shedding in stream databases: A control-based approach," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, ser. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, p. 787–798.

[19] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *Proceedings of the 20th International Conference on Data Engineering*, ser. ICDE '04. USA: IEEE Computer Society, 2004, p. 350.

[20] V. Cerf and R. Kahn, "A protocol for packet network intercommunication," *IEEE Transactions on communications*, vol. 22, no. 5, pp. 637–648, 1974.

[21] N. Michael, "Release it!–design and deploy production-ready software," 2007.

[22] Netflix, "Hystrix: Latency and fault tolerance for distributed systems," 2023. [Online]. Available: https://github.com/Netflix/Hystrix/

[23] K. Surendro and W. Sunindyo, "Circuit breaker in microservices: State of the art and future prospects," in *MSE*, vol. 1077. UK: IOP, 2021, pp. 1–10.

[24] M. Montesi and J. Weber, "Circuit breakers, discovery, and api gateways in microservices," 2016.

[25] N. C. Mendonça, C. M. Aderaldo, J. Cámara, and D. Garlan, "Model-based analysis of microservice resiliency patterns," in *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2020, pp. 114–124.

[26] L. J. Jagadeesan and V. B. Mendiratta, "When failure is (not) an option: Reliability models for microservices architectures," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 19–24.

[27] N. Dattatreya Nadig, "Testing resilience of envoy service proxy with microservices," Master's thesis, KTH, School of Electrical Engineering and Computer Science, 2019.

[28] B. Dab, I. Fajjari, M. Rohon, C. Auboin, and A. Diquélou, "Cloud-native service function chaining for 5g based on network service mesh," in *ICC 2020*. USA: IEEE, 2020, pp. 1–7.

[29] M. Akbarisamani, "Service based architecture with service mesh platform in the context of 5g core," Master's thesis, Tampere University, 2019.

[30] X. Xiaojing and S. Govardhan, "A service mesh-based load balancing and task scheduling system for deep learning applications," in *CCGRID '20*. USA: IEEE, 2020, pp. 843–849.

[31] Ł. Wojciechowski *et al.*, "Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh," in *INFOCOM '21*. USA: IEEE, 2021, pp. 1–9.

[32] Y.-M. Wang, Y. Huang, and W. Fuchs, "Progressive retry for software error recovery in distributed systems," in *FTCS '93*. USA: IEEE, 1993, pp. 138–144.

[33] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *ICDCSW '16*. USA: IEEE, 2016, pp. 57–66.

[34] R. R. Karn, R. Das, D. R. Pant, J. Heikkonen, and R. Kanth, "Automated Testing and Resilience of Microservice's Network-link using Istio Service Mesh," in *2022 31st Conference of Open Innovations Association (FRUCT)*, 2022, pp. 79–88.

[35] Google Inc, "Sample cloud-native application," 2023. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[36] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.

[37] Docker Inc., "Docker: Accelerated, containerized application development," 2023. [Online]. Available: https://www.docker.com/

[38] Istio Community, "Istio: Simplify observability, traffic management, security, and policy with the leading service mesh," 2023. [Online]. Available: https://istio.io/

[39] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more robust cloud applications," in *ICSE '14*. USA: ACM, 2014, p. 700–711.

[40] Prometheus Authors, "Prometheus - monitoring system and time series database," 2023. [Online]. Available: https://prometheus.io/

[41] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, p. 74–80, Feb. 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408794

[42] Envoy Community, "Circuit breaking / envoy documentation," 2023. [Online]. Available: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/circuit_breaking

[43] Istio Community, "Istio / virtual service - httpretry." [Online]. Available: https://istio.io/latest/docs/reference/config/networking/virtual-service/

[44] L.-Y. Hu and P. M. Bentler, "Application of structural equation modeling in nursing research: a brief review," *Taiwanese Journal of Obstetrics and Gynecology*, vol. 53, no. 4, pp. 520–524, 2014.

[45] G. Dash and J. Paul, "Cb-sem vs pls-sem methods for research in social sciences and technology forecasting," *Technological Forecasting and Social Change*, vol. 173, p. 121092, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0040162521005254

[46] T. Raykov and G. A. Marcoulides, "Introduction to applied psychometrics," *Routledge*, 2005.

[47] D. A. K. Kenny, "Measuring model fit," 2020. [Online]. Available: http://davidakenny.net/cm/fit.htm

[48] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM computer communication review*, vol. 18, no. 4, pp. 314–329, 1988.

[49] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[50] J. L. Gastwirth, Y. R. Gel, and W. Miao, "The impact of levene's test of equality of variances on statistical theory and practice," *Statistical Science*, vol. 24, no. 3, pp. 343–360, 2009.

[51] M. R. Saleh Sedghpour, A. Obeso Duque, X. Cai, B. Skubic, E. Elmroth, C. Klein, and J. Tordsson, "Hydragen: A microservice benchmark generator," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 2023, pp. 189–200.