An Empirical Study of Service Mesh Traffic Management Policies for Microservices

Mohammad Reza Saleh Sedghpour
Cristian Klein
Johan Tordsson
{msaleh,cklein,tordsson}@cs.umu.se
Department of Computing Science, Umeå University
Umeå, Sweden

ABSTRACT

A microservice architecture features hundreds or even thousands of small loosely coupled services with multiple instances. Because microservice performance depends on many factors including the workload, inter-service traffic management is complex in such dynamic environments. Service meshes aim to handle this complexity and to facilitate management, observability, and communication between microservices. Service meshes provide various traffic management policies such as circuit breaking and retry mechanisms, which are claimed to protect microservices against overload and increase the robustness of communication between microservices. However, there have been no systematic studies on the effects of these mechanisms on microservice performance and robustness. Furthermore, the exact impact of various tuning parameters for circuit breaking and retries are poorly understood. This work presents a large set of experiments conducted to investigate these issues using a representative microservice benchmark in a Kubernetes testbed with the widely used Istio service mesh. Our experiments reveal effective configurations of circuit breakers and retries. The findings presented will be useful to engineers seeking to configure service meshes more systematically and also open up new areas of research for academics in the area of service meshes for (autonomic) microservice resource management.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; *Reliability*; • **General and reference → Empirical studies**.

KEYWORDS

microservices; service mesh; traffic management; circuit breaking; retry; microservice resiliency

ACM Reference Format:
Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. 2022. An Empirical Study of Service Mesh Traffic Management Policies for Microservices. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3489525.3511686

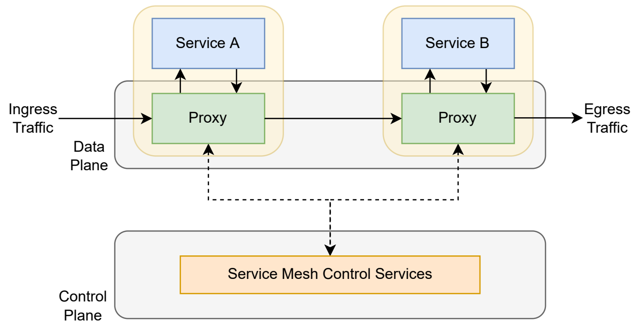


Figure 1: Service mesh architecture

1 INTRODUCTION

Microservices have recently attracted considerable attention in both academia and industry. A microservice architecture features hundreds or even thousands of small services with multiple instances that function as cohesive and independent processes interacting via messages [9]. Because microservice performance depends on the implemented functionality and the incoming workload, a continuously increasing load or a load spike may cause violation of a service level objective (SLO) [16]. Therefore, the inter-service communication and traffic management mechanisms used in such dynamic environments are complex. Service meshes were introduced to handle this complexity and to facilitate management, observability, and communication between microservices [25]. In essence, a service mesh is an infrastructure layer built directly into the microservices as a set of configurable proxies, shown in Fig. 1. For example, Envoy is a popular user-space proxy that functions as the data plane of the open source service mesh Istio [14]. This allows the network to be completely abstracted, providing a single point of network interaction for each service [24].

A service mesh provides a range of traffic management policies such as circuit breaking and retry mechanisms, which are claimed to make applications more robust and resilient towards failures of dependent services or the network. Circuit breaking rejects incoming requests to protect latency at the expense of availability, enabling faster reactions to overloads and load spikes than would be achievable through capacity auto-scaling [25]. A retry setting specifies the maximum number of times that a sidecar proxy will attempt to connect to a service if the initial call fails. The interval between retries prevents the called service (the server side proxy) from being overwhelmed with requests. In a service mesh, both circuit breakers and retry mechanisms are configured on the called services.

Despite the considerable interest in circuit breaking, retry mechanisms, and their potential benefits for microservice robustness and performance, there is a lack of systematic studies on how important these policies and their control parameters actually are for a resilient and robust service mesh. Some of them may have little impact on performance and could be ignored in some scenarios. However, in other scenarios with complex microservice topologies that may include multi-tier services with different fan-in and fan-out characteristics, improper configuration of the circuit breaker and retry mechanism in the backend services could cause a retry storm, in which the system becomes overwhelmed by queued retries and takes too long to recover [8]. To clarify the impact of traffic management policies based on circuit breakers and retry mechanisms as well as the effects of varying the parameter settings of these tools, this work seeks to answer three research questions:

- RQ1. **How should circuit breakers be configured to improve the resiliency of microservice-based applications with complex topologies?**
- RQ2. **How should retry mechanisms be configured to improve the resiliency of microservice-based applications with complex topologies, with and without circuit breakers?**
- RQ3. **How do workload characteristics impact the answers to RQ1 and RQ2?**

To answer these questions, we present a set of experiments performed using Online Boutique (Fig. 2) - a simple microservice based application consisting of 11 microservices with 33 traffic management parameters that can be tuned. In these experiments, traffic management is performed using Istio and a range of different traffic management policies and parameter settings are tested to evaluate their impact on performance and illustrate the challenges of configuring traffic management policies when using a service mesh.

The results obtained provide new insights into the practical use of traffic management policies in a service mesh and how such policies interact to enhance application performance and resiliency.

2 BACKGROUND

In this section, we discuss previous studies relevant to the work presented herein. We divide these earlier studies into works concerning service meshes, circuit breaker patterns, and retry mechanisms.

2.1 Service Meshes

Advances in microservice technology have significantly increased the speed and agility of software service delivery but have also increased the operational complexity of modern applications. The purpose of a service mesh is to mitigate this complexity by adding an infrastructure layer between microservices. Some recent publications have highlighted aspects of service mesh technologies in need of further research [20]. For example, one paper presented an in-depth analysis of different design decisions to help establish architectural decision-making guidelines for service meshes [10].

Additionally, to secure service mesh solutions such as Istio, a protected coordination scheme was developed [17], and Chandramouli et al. [5] offered deployment guidance for proxy-based service mesh components that collectively form a robust security infrastructure for supporting microservice-based applications.

Finally, the advantages of service meshes have been investigated in several different use cases including in the context of the 5G core [6] [1] and in efforts to improve scheduling algorithms [30] [29].

2.2 Circuit breaker patterns

Resilience is a key issue for any software architecture, and microservice architectures are no exception to this rule. Circuit breaker patterns were therefore introduced to handle run-time failure in microservice applications and improve the resilience of the software stack. The circuit breaker pattern is implemented to reject requests when a certain condition is satisfied - for example, returning a HTTP 503 error if the number of queued requests exceeds 20.

Several publications have investigated the advantages of circuit breaker patterns in various use cases including in the context of IoT [2].

Montesi et al. [23] identified three distinct circuit breaker patterns: (1) *The client-side circuit breaker* pattern, in which each client includes a separate circuit breaker for intercepting calls to each external service that the client may call, (2) *The service-side circuit breaker* pattern, in which all client invocations received by a service are first processed by an internal circuit breaker that decides whether the invocation should be processed or not. (3) *The proxy circuit breaker* pattern, in which circuit breakers are deployed in a proxy service that sits between clients and services and handle all incoming and outgoing messages. Service mesh-based technologies fall into the latter category because they add a sidecar proxy.

A review of the literature on circuit breakers for microservices has been published [27], and Saleh Sedghpour et al. [25] studied the impact of adaptive circuit breaker configuration on microservice performance.

2.3 Retry mechanisms

A retry mechanism generates a new request, either to the same instance of a called service or a different instance of a called service, when an initial request fails. This may happen several times; the permitted number of attempts is specified using a retry attempt parameter, and the interval between consecutive retries is set using a retry timeout. Retry mechanisms are not a novel concept in distributed systems; their use has been described in older studies [28] dealing with error recovery. Heorhiadi et al. [13] proposed Gremlin, a framework for systematically testing the failure-handling capabilities of microservices that allows the operator to easily design tests that include resiliency patterns. Another group [22] performed a continuous-time Markov chains analysis of circuit breaker behavior and retry mechanism patterns in a single-tier architecture, while Dattatreya Nadig demonstrated the benefits of Envoy as a sidecar proxy for increasing microservice resiliency using retry mechanisms, circuit breaking patterns, and rate limiting [7]. Rate limiting is conceptually similar to circuit breaking and may improve resiliency when many services are forwarding requests to a smaller number of services and the average request latency is low.

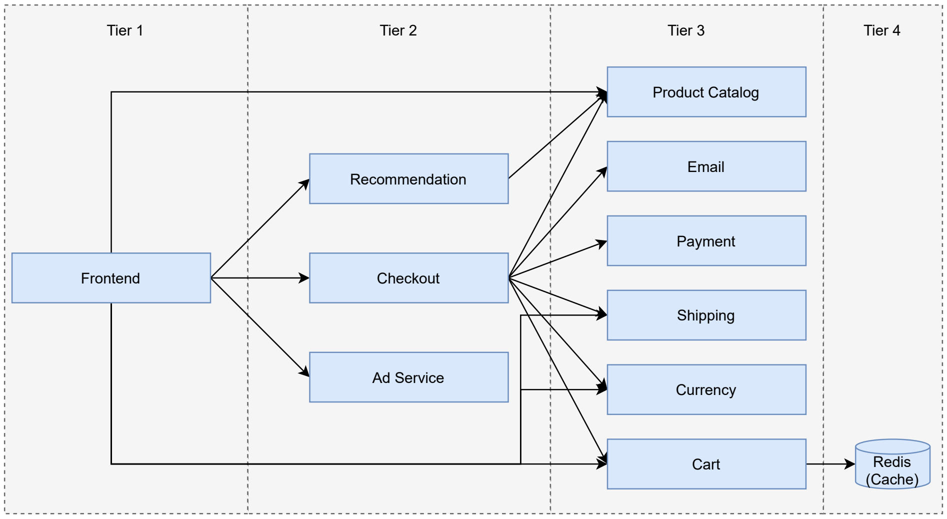


Figure 2: The Online Boutique model application consisting of 11 microservices written in different languages that talk to each other via gRPC.

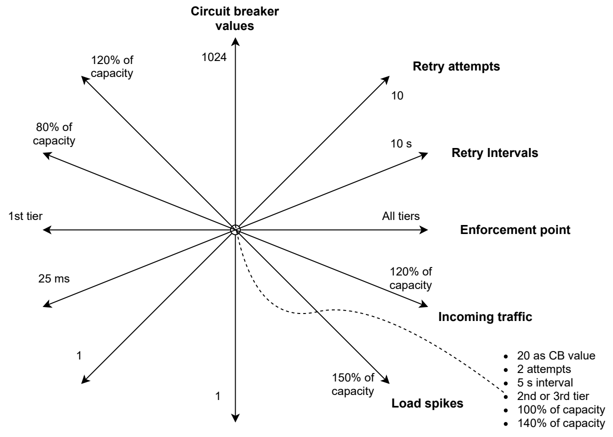


Figure 3: Experimental design space

Although Envoyproxy includes an exponential back-off algorithm to prevent retry storms, it is not configurable for different scenarios and is also not yet supported by Istio [11].

Despite the recent academic interest in microservices, few studies have investigated microservice resiliency patterns. This study differs from previous works on microservice resiliency in that it is based on a systematic analysis of circuit breaker patterns and retry mechanisms as traffic management policies, using several different circuit breaker and retry parameter settings in a 4-tier architecture deployed in service mesh.

3 APPROACH

In this section we introduce the subject of our study and our approach to data collection and analysis. We then discuss factors that could potentially reduce the validity of our experiments and the measures taken to ensure validity.

3.1 Study Subjects

Because referral-chain (snowball) sampling has been widely used in literature reviews focusing on software engineering problems [3], we used this method to select study subjects. In referral-chain sampling, items are selected based on their relationship to previously selected items, which facilitates the identification of relevant items outside the initial sampling frame.

We considered five different applications for use in our experiments: (1) Bookinfo [15] (2) DeathStarBench [12] which includes 3 different applications (3) Online Boutique. We chose to use only applications satisfying the following three conditions: a) have at least 3 tiers in their software stack b) support concurrent calls between microservices to enable overloading of the backend microservices c) do not use any service networking platform in their software stack so we can be certain that the traffic is passing through the sidecar proxies.

The only one of the five applications satisfying all three conditions is Online Boutique. This application consists of 11 microservices and is a web-based e-commerce app that allows users to browse items, add them to a cart, and purchase them [4]. Its architecture is shown in Fig 2. We chose to focus on one of the application's eleven endpoints, /cart, which invokes a chain of three microservices. Upon invoking this endpoint, a request is sent to a frontend service, which in turn invokes the recommendation service by sending one request, and four requests are sent to the product catalog service. The recommendation service also sends a request to the product catalog service. A total of 6 internal requests are thus required to deliver a successful complete response to the external client.

3.2 Data Collection

To collect data for our study, we developed a tool [26] that repeatedly configures selected traffic management policies in Istio. To do this, we manually identified three key traffic management parameters in Istio: a) *HTTPMaxRequest*, which is the maximum number of queued requests in the circuit breaker's configuration, b) *attempts*, which specifies the maximum number of times the sidecar proxy attempts to connect to a service if the initial call fails, c) *perTryTimeout*, which specifies the interval between retries when attempting to connect to a service. Then we ran the HTTPMon [18] traffic generator, which selects a think-time and a number of users, and maintains a number of client threads equal to the number of users. We configured HTTPMon to generate an open-loop workload in which the request rate is maintained independently of the response time of the system under load. During the execution of HTTPMon, the response times and responses of all services were monitored.

We deployed the tool and a service mesh cluster including Online Boutique on 5 bare-metal machines with 16GB of RAM, two Intel Xeon E5430 2.66 GHz CPUs with four cores and hyper-threading, and a 256 GB NVMe drive running Ubuntu 20.04 LTS. The service mesh cluster was set up with Kubernetes 1.19.14, Docker 19.03.15, and Istio 1.11.2. Each service of Online Boutique was limited to 2 CPU cores via the standard Kubernetes mechanism. We repeated each experiment 5 times and each experiment took 5 minutes, with the first minute of each experiment being considered as warm-up phase. In total, we collected data from more than one thousand experiments, but we only report 320 of them both due to space limitation and to focus on the important results. In the reported experiments, we generated 110M requests which took more than 130 hours to complete in total.

Capacity. In all performed experiments, we define capacity as the maximum achieved throughput, i.e., all successful requests for which the response time is less than 100 ms.

3.3 Analysis

Our initial goal in the analysis was to determine how much the parameter selection (see Fig. 3) impacted the performance of a black-box service. To this end, we performed visual analysis of all three possible response types (successful, failed, and circuit-broken) and response times.

For each configuration of traffic management in the service mesh, we plot the frequency of each response type (successful, failed and circuit-broken) over time and cumulative distribution functions (CDFs) of the response times of all requests and carried throughput. We then iteratively discuss and classify the resulting plots.

3.4 Threats to Validity

Despite careful research design, studies such as those presented here inevitably have limitations and factors that may reduce their validity. The most important of these limitations and factors for this work are summarized below.

3.4.1 External Validity. This paper used only a specific version of each tool. While we argue that the chosen tool versions are representative of tools that would be used in real world cases, the findings presented herein cannot be directly generalized to other versions, particularly since performance may differ between versions even if the same configuration policies are used. Additionally, we used snowball sampling to identify relevant study subjects. While this approach is common in empirical research, it inherently makes it impossible draw conclusions about the population in general and can lead to sampling a small subset of a larger population. To summarize, while we have no direct evidence that our results are applicable to the studied tools in all cases, we expect that similar results would be obtained if different variants of the chosen tools were used.

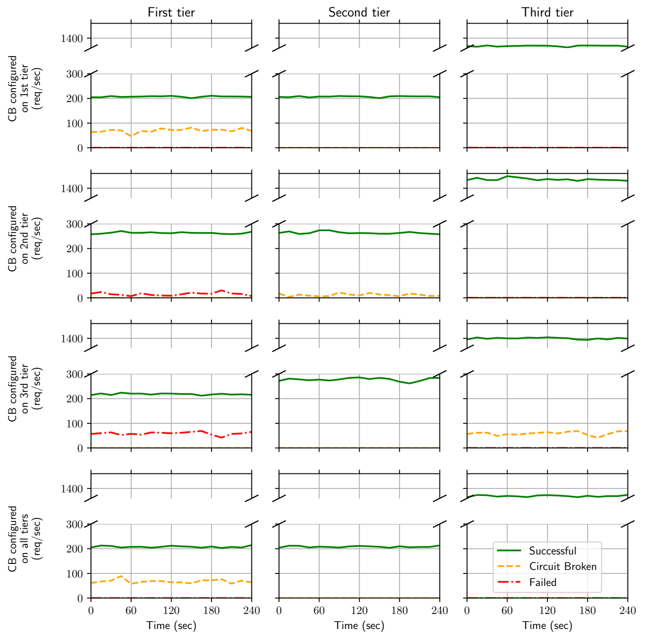
3.4.2 Internal Validity. Internal validity is inevitably affected by the fact that some design decisions must be made when defining the configuration values to test. Empirical data analysis has not suggested that the whole stack's behavior would have been radically different if values other than those chosen were used, but this is clearly impossible to prove. Another internal validity threat is that we performed all experiments on bare-metal platforms. Some of the executions in our study could thus have been affected by our choice of hardware for data collection. However, we consider it unlikely that the general validity of our results would be threatened by the specific hardware chosen for the study. On the other hand, service meshes are often employed on virtualized platforms whose performance characteristics may differ from those of bare-metal platforms due to the presence of an additional scheduling layer and sharing of hardware resources. It seems unlikely that the general validity of our results would be impacted by the use of a bare-metal platform. Another internal validity threat is that there are infinite potential traffic scenarios, which could impact some of our experiments. However, we tested a wide range of different traffic scenarios to maximize the breadth of the conditions tested in our experiments and improve the robustness and reliability of our conclusions (we also ran experiments with other values, which are not reported as the results were similar to reported ones.).

4 IMPACT OF DIFFERENT TRAFFIC MANAGEMENT POLICIES ON PERFORMANCE

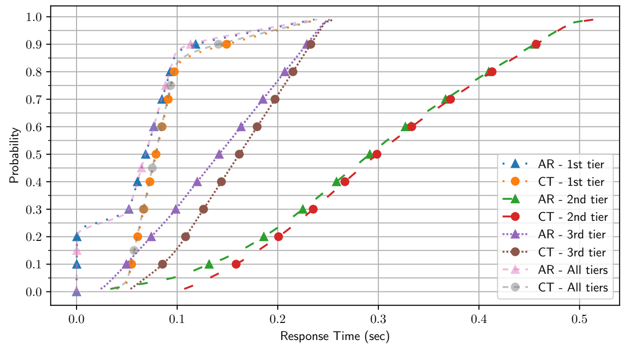
4.1 Impact of circuit breakers

To study the impact of circuit breakers, we performed a series of experiments in which we compared the response code frequencies observed with different circuit breaker parameter settings and for circuit breakers acting on different tiers of the studied application under different traffic scenarios. We repeated each experiment 5 times but since there were no significant differences between the results of replicate experiments we only discuss the results of individual experiments. We performed t-tests to evaluate the significance of differences between between related experiments, applying a significance threshold of $p < .01$ (with all results in the [0.42-0.63] range). The results of varying individual circuit breaker parameters are discussed below.

4.1.1 Enforcement point. To study the impact of varying the circuit breaker enforcement point, we conducted a set of experiments in which the emulated workload was 120% of capacity and the circuit breaker's maximum queue length was set to 20 while varying the application tier on which the circuit breaker acted. The impact of varying the maximum queued requests parameter is discussed in Section 4.1.3. The frequencies of the different status codes in these experiments are presented in Fig. 4a. Based on the results, if the



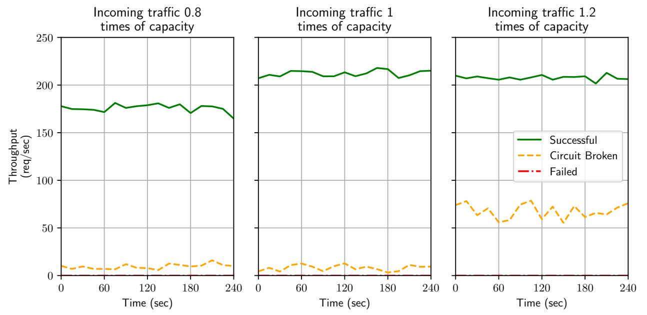
(a) Throughput achieved in terms of successful response codes (green solid lines), circuit broken requests (orange dashed lines), and failed requests (red dashed dotted lines).



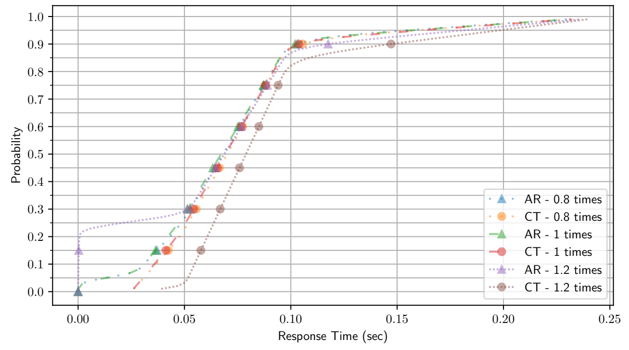
(b) Cumulative density function of response times of carried throughput (circle marker) and all requests (triangle marker) from an external client's perspective when the circuit breaker is enforced on the application's 1st tier (blue and orange loosely dotted lines), 2nd tier (green and red dashed lines), 3rd tier (purple and brown densely dotted lines), or all three tiers (pink and gray dashed dotted lines).

Figure 4: Impact of enforcement point for best circuit breaker configuration (20) on different tiers with same overload scenario (120% of capacity).

circuit breaker acts on the first tier, it prevents downstream tiers from consuming resources on overloading requests. Conversely, if the circuit breaker acts on the second or third tier, some requests are rejected (circuit broken) after traversing a few tiers. This means that although the result is the same from the external client's perspective – the request fails – more resources are consumed than when the circuit breaker acts on the first tier. Additionally, when the circuit breaker was applied on all tiers, the first tier dropped all the overloading requests. As mentioned previously, the first tier



(a) Throughput achieved in terms of successful response codes (green solid lines), circuit broken requests (orange dashed lines), and failed requests (red dashed dotted lines).



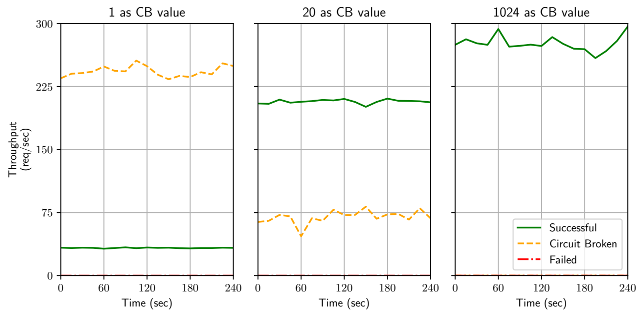
(b) Cumulative density function of response times of carried throughput (circle marker) and all requests (triangle marker) from the outside client's perspective when the volume of incoming traffic is equal to 0.8 × capacity (blue and orange loosely dotted lines), 1 × capacity (green and red dashed lines) and 1.2 × capacity (purple and brown densely dotted lines).

Figure 5: Impact of incoming traffic with the circuit breaker (20) acting on tier one under different overload scenarios.

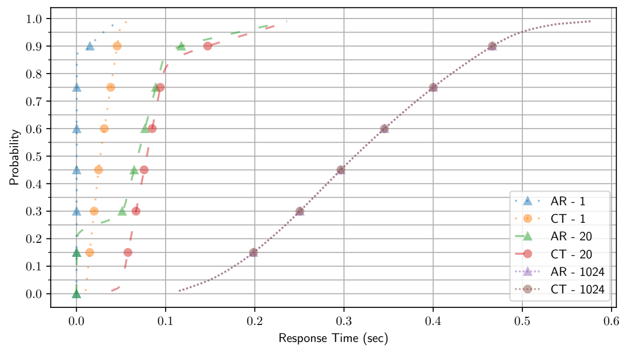
calls both second and third tier services (1 time and 4 times, respectively), while the second tier service calls the third tier service 1 time. Consequently, when varying the tier on which the circuit breaker acts, the numbers of circuit broken and successful requests on third tier also changes. In contrast, as shown in Fig. 4b, when we enforce the circuit breaker on the first tier or on all tiers, the response time from an outside client's perspective is lower than for enforcement on the second or third tier. This improves the user experience because requests are failed fast rather than failed slow. Additionally, when the circuit breaker acts on the second tier, the achieved response times are higher than when it acts on the third tier. The reason for this seems to be related to the implementation of the second and third tiers.

4.1.2 *Incoming traffic volume.*

To investigate the impact of varying the incoming traffic volume, we performed a series of experiments in which the emulated workload was varied between 80% of capacity and 120% of capacity. In this case we limited the circuit breaker's maximum queue length to 20 on the first tier (this was the best performing circuit breaker configuration, as shown in Section 4.1.3). The status code frequencies under these conditions are shown in Fig. 5a. It was clear that the volume of incoming traffic directly



(a) **Throughput achieved in terms of successful response codes (green solid lines), circuit broken requests (orange dashed lines), and failed requests (red dashed dotted lines).**



(b) **Cumulative density function of response times of carried throughput (circle marker) and all requests (triangle marker) from the outside client's perspective when the circuit breaker is enforced on the 1st tier with maximum queue lengths of 1 (blue and orange loosely dotted lines), 20 (green and red dashed lines), and 1024 (purple and brown densely dotted lines).**

Figure 6: Impact of circuit breaker value on first tier with same overload scenario (120% of capacity).

influenced the frequency of unsuccessful requests. Although the rate of successful requests was higher when the incoming traffic volume was 120% of capacity, higher incoming traffic volumes led to higher frequencies of unsuccessful requests in the microservice-based architecture. Additionally, we found that fast failure helped the microservice architecture to maintain an acceptable response time in the face of high incoming traffic volumes.

4.1.3 *Circuit breaker queue length.*

To study the impact of circuit breaker maximum queue length, we performed a series of experiments in which the maximum queue length was varied between 1, 20 and 1024 for the first tier. In this case we limited the emulated workload to 120% of capacity. Fig. 6a highlights the trade-off between the maximum request rate and the response time when choosing the circuit breaker's maximum queue length. A high maximum queue length value allows more requests "to pass through" and thus increases the request rate, but also increases response times, which is undesirable. In contrast, a low maximum queue length value reduces response times but also reduces the request rate. The maximum queue length used in the previous experiments, 20, was found to provide a good balance between these two factors.

4.2 Impact of including a retry mechanism

To investigate the impact of including a retry mechanism, we performed a series of experiments comparing the response codes and response times observed when using a retry mechanism in conjunction with a circuit breaker with enforcement on different application tiers under different traffic scenarios. We repeated each experiment 5 times; since there were again no significant differences between replicate experiments performed under the same conditions, we only present results for individual experiments. We performed t-tests to evaluate the significance of observed differences between related experiments, applying a significance threshold of $p < .01$ (with all results in the [0.56-0.68] range). To thoroughly evaluate the performance impact of including a retry mechanism, we divide the problem space as described below.

4.2.1 *Retry mechanism and circuit breaking.*

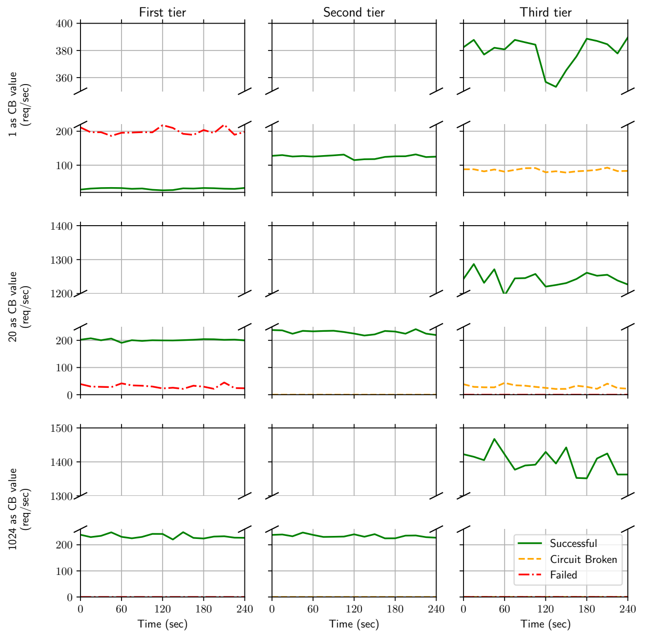
To analyze the interaction between retry mechanisms and circuit breaker patterns in a service mesh, we present a subset of the experiments in which the incoming traffic volume was fixed at 100% of capacity, the retry mechanism was configured to permit only a single retry, and the circuit breaker acted on the third tier with varying maximum queue lengths. The results obtained under these conditions are shown in Fig. 7. If circuit breaker's maximum queue length is too short or there are failures in the called service (with a retry mechanism active), we can expect a series of retried requests from the caller services (in this case, both first and second tier services call the third tier service). As shown in 7b, if the circuit breaker's maximum queue length is high, the response time increases, which is consistent with the previously discussed experiments without the retry mechanism. We also discussed similar results in our previous work [25], there is a trade-off between response times and carried throughput, to be handled per preference of the system administrator. The main difference between the cases with and without the retry mechanism is that including the retry mechanism increased the response time further.

4.2.2 *Impact of number of retry attempts.*

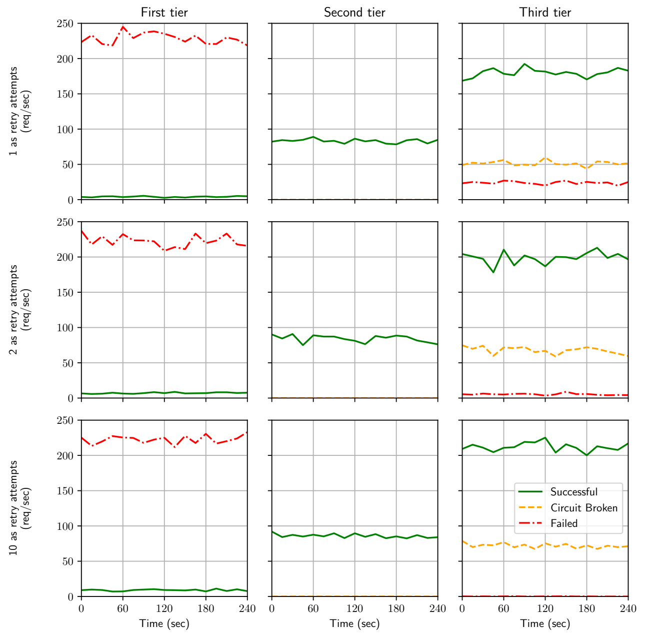
A key parameter controlling a retry mechanism in a service mesh is the number of permitted retries. To study the impact of varying this parameter, we performed a series of experiments in which the incoming traffic was fixed at 100% of capacity and the circuit breaker's maximum queue length was set to 1 (to maximize the impact of retry attempts) with different numbers of retry attempts on the third tier. As shown in Fig. 8a, increasing the number of permitted retries slightly increased the numbers of successful and circuit broken requests while reducing the number of failed requests. However, the relative magnitudes of the changes in request type frequencies were substantially smaller than the relative magnitude of the changes in the number of permitted retries. Additionally, when the number of permitted retries was set to zero under these conditions, there were fewer successful and circuit broken requests and more failed requests than when a single retry was allowed. As shown in Fig. 8b, the response time from the outside client's perspective increased slightly as the number of retry attempts increased.

4.2.3 *Impact of varying the retry interval.*

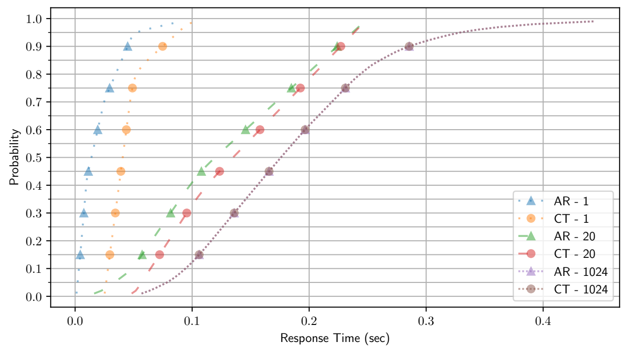
Another retry mechanism parameter that can be varied is the interval between retries. To investigate the impact of varying the retry interval, we present



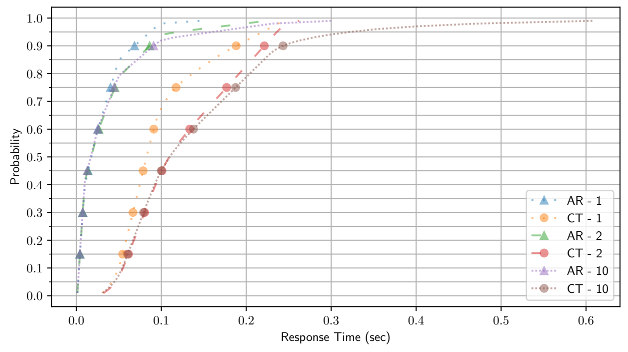
(a) Throughput achieved in terms of successful response codes (green solid lines), circuit broken requests (orange dashed lines), and failed requests (red dashed dotted lines).



(a) Throughput achieved in terms of successful response codes (green solid lines), circuit broken requests (orange dashed lines), and failed requests (red dashed dotted lines)



(b) Cumulative density function of response times of carried throughput (circle marker) and all requests (triangle marker) from the outside client's perspective when the circuit breaker queue length is set to 1 (blue and orange loosely dotted lines), 20 (green and red dashed lines), and 1024 (purple and brown densely dotted lines).



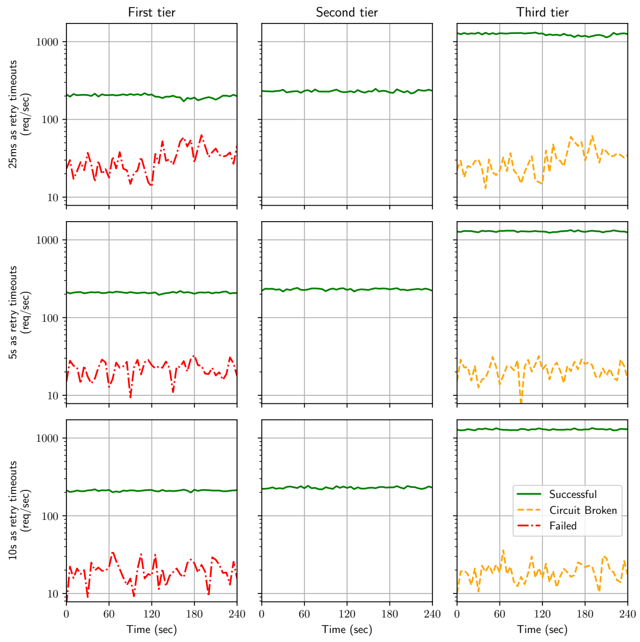
(b) Cumulative density function of response times of carried throughput (circle marker) and all requests (triangle marker) from the outside client perspective when number of retry attempts is set to 1 (blue and orange loosely dotted lines), 2 (green and red dashed lines), and 10 (purple and brown densely dotted lines).

Figure 7: Impact of the retry mechanism when combined with circuit breaker enforcement on the 3rd tier with different circuit breaker queue lengths. The retry mechanism setting (1 attempt) and overload condition (100% of capacity) were held constant.

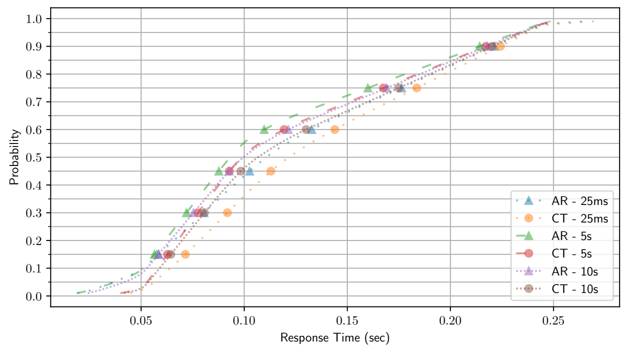
Figure 8: Impact of varying the number of retry attempts with the circuit breaker acting on the third tier and the circuit breaker's maximum queue length set to 1 under fixed overload conditions (100% of capacity).

a subset of experiments in which the retry interval was varied with the incoming traffic volume fixed at 100% of capacity and the circuit breaker queue length set to 20 with 10 retry attempts. In these experiments, both the retry mechanism and the circuit breaker were active on the third tier only. As shown in Fig. 9a, if the retry interval was too short, the frequency of circuit broken requests increased slightly but the relative magnitude of the change in the retry interval was substantially greater than the relative magnitude of the change in the frequencies of each request type (successful, circuit broken and failed). We thus conclude that the retry interval does not greatly affect performance. Additionally, as shown in Fig. 9b, varying the retry interval did not greatly affect the response time for the outside client.

4.2.4 Impact of workload spikes and different retry intervals. The results presented above show how retries interact with a circuit breaker under conditions of sustained overload. However, retries



(a) Throughput achieved in terms of successful response codes (green solid lines), circuit broken requests (orange dashed lines), and failed requests (red dashed dotted lines)



(b) Cumulative density function of response times of carried throughput (circle marker) and all requests (triangle marker) from the outside client's perspective when the retry interval is set to 25 milliseconds (blue and orange loosely dotted lines), 5 seconds (green and red dashed lines), and 10 seconds (purple and brown densely dotted lines).

Figure 9: Impact of varying the retry interval with a circuit breaker queue length of 20, permitted retry attempts of 10, fixed overload condition (100% of capacity), and both the circuit breaker and retry mechanism acting on the third tier service.

are often used as a mechanism to overcome short-lived workload spikes or transient failures. To study the impact of workload spikes and different retry intervals, we performed a series of experiments in which the incoming traffic was set at 100% of capacity most of the time, spiking to 120% of capacity for 5 seconds at 60 seconds intervals. In these experiments, the circuit breaker queue length was set to 20 and the number of permitted retry attempts was set to 10

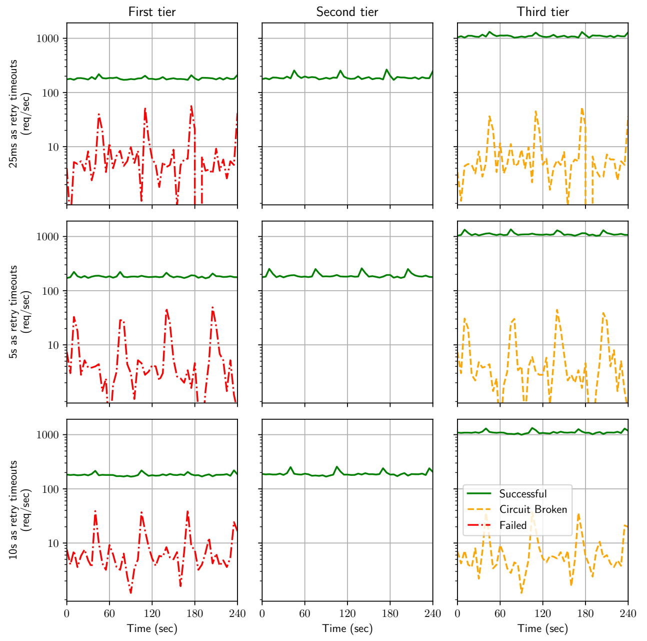


Figure 10: Impact of workload spikes and different retry intervals: Throughput achieved in terms of successful response codes (green solid lines), circuit broken requests (orange dashed lines), and failed requests (red dashed dotted lines) for different retry intervals with a third-tier circuit breaker queue length of 20 and permitted retry attempts of 10 in third tier with spikes (120% of capacity) in the incoming workload (100% of capacity).

while varying the retry interval; both the retry mechanism and the circuit breaker acted on the third tier. We also ran experiments with different traffic scenarios such as 90%, 110% and 120% of capacity and different spiking scenarios such as 110%, 130%, 140% of capacity. We used different spike duration such as 10, 15, 30 seconds and different intervals such as 30, 90 and 120 second. The result of these scenarios are not shown as they are similar to the reported experiments. As shown in Fig. 10, if the retry interval is too short, the frequency of circuit broken requests after spikes is slightly increased. The pattern of failed requests in first tier follows the pattern of circuit broken requests in third tier. The external response times for the workload with spikes are similar to those seen in Fig.9b (data not shown).

4.2.5 Impact of different workload spikes. We next compared the impact of varying the magnitude of the spikes in the incoming workload, as shown in Fig. 11. In these experiments, the incoming traffic volume was mostly set to 100% of the system's capacity while the spike volume was varied. The circuit breaker queue length was set to 20 and 10 retries were permitted while varying the retry interval on the third tier. The results obtained were similar to those observed when simply varying the retry interval (see section 4.2.4). As shown in Fig. 11, varying the spike volume caused the frequency of failed requests to increase at the same rate as the first tier spike volume. As mentioned previously, the pattern of failed requests in the first tier mirrored the pattern of circuit broken requests in

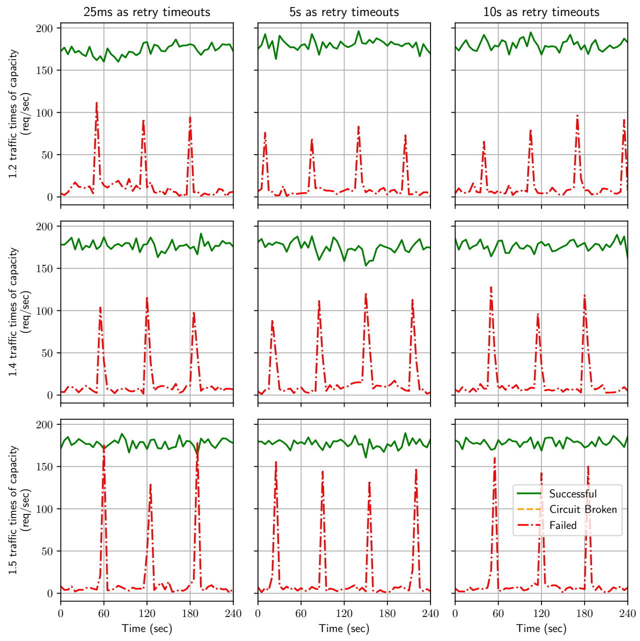


Figure 11: Impact of different workload spikes: Throughput achieved in terms of successful response codes (green solid lines), circuit broken requests (orange dashed lines), and failed requests (red dashed dotted lines) for different retry intervals and spikes in the incoming workload (100% of capacity) with a maximum circuit breaker queue length of 20 and permitted retry attempts of 10 on the third tier.

the third tier. Varying the magnitude of the workload spikes also had little effect on response times (data not shown); the pattern observed was similar to that seen in Fig.9b.

4.2.6 Impact of varying the enforcement points of the retry mechanism and circuit breaker configuration. Finally, to study the impact of varying the enforcement point of the retry mechanism and circuit breaker, we conducted experiments with a circuit breaker maximum queue length of 20, 2 permitted retry attempts, and a 5 second retry interval while varying the tier on which the circuit breaker and retry mechanism acted. A workload equivalent to 100% of system capacity was imposed in all cases. The results are shown in Fig. 12a. Because our workload generator did not support the use of the retry mechanism when the circuit breaker was set to act on the first tier, the results obtained in this case were identical for all retry mechanism configurations. Conversely, if the circuit breaker was set to act on the second tier, the use of the retry mechanism caused some increase in failed requests when it was set to act on all possible tiers (specially when it was acting on all tiers). As mentioned before, the pattern of failed requests in the first tier mirrors the pattern of circuit broken requests in tiers 2 and 3. When the circuit breaker was set to act on tier 3, the frequency of failed requests was highest when the retry mechanism was configured to act on tier 1 or on all tiers. Finally, when the circuit breaker was configured to act on all three tiers, varying the enforcement point of the retry mechanism had little effect on the frequencies of successful, failed, and circuit broken responses. The response times experienced by an outside client in these experiments are shown in Fig. 12b. It can be seen that configuring the circuit breaker to act on the first tier or on all tiers (including first tier) means that varying the enforcement point of the retry mechanism has no impact on response times, and response times in these cases are higher than when the circuit breaker is configured to act on tier 2 or tier 3. Conversely, if we configure the circuit breaker to act on tier 2, the response times are slightly higher when the retry mechanism is enforced on all tiers. Moreover, if we configure the circuit breaker to act on tier 3, the response times are slightly higher when the retry mechanism is configured to act on tier 1 or on all tiers.

4.3 Discussion

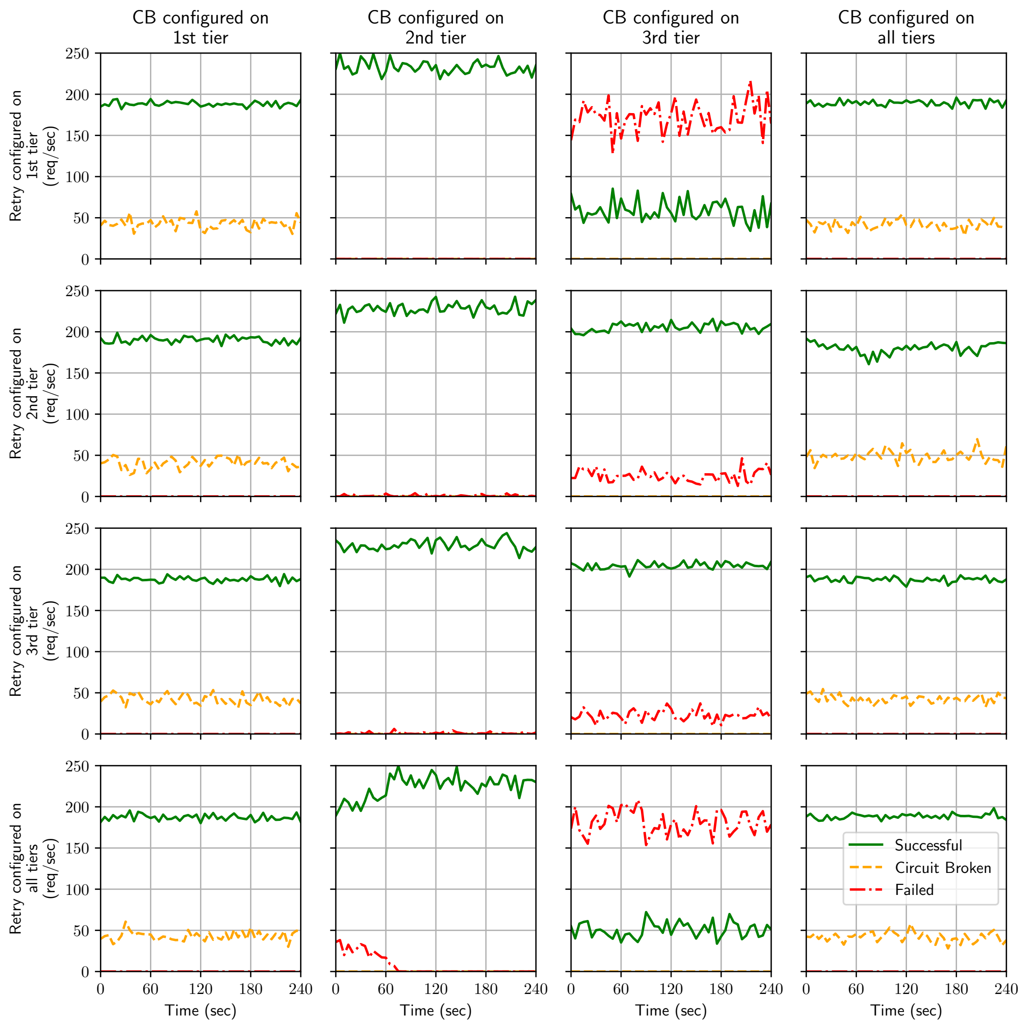
RQ1 asks *"How should circuit breakers be configured to improve the resiliency of microservice-based applications with complex topologies?"*

The key factors that determine the impact of a circuit breaker are presented in section 4.1, which begins with an analysis of the effect of varying the circuit breaker enforcement point. We also investigated the impact of the volume of incoming traffic on system performance when the circuit breaker is configured to act on the first application tier. Finally, we investigated how the circuit breaker queue length parameter affects system performance.

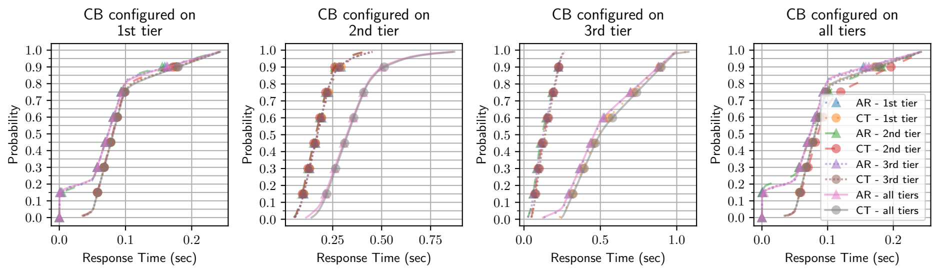
Our experiments show that a well-configured circuit breaker can maintain a favorable response time while maximizing application throughput. Circuit breakers enable fast failures and prevent clients from repeatedly trying to connect to an overloaded or failing service. Circuit breakers can also enhance the user experience and reduce recovery times after an outage. For instance, when Meta (Facebook) recently began a recovery process after a major outage, users started generating more requests than they would normally [21], which could cause infrastructure overload and increase recovery times. To summarize, we found that configuring circuit breakers to act on lower application layers improved the user experience, that setting the circuit breaker queue length to an excessively low value may increase the number of dropped requests, and that setting the circuit breaker queue length too high makes it impossible to guarantee an acceptable response time or user experience, which we expected based on the common sense rules for setting circuit breaker values. We also learned that identifying an optimal circuit breaker value is tricky and depends on many factors, which confirms results in our previous work [25]. Finally we conclude that circuit breakers are useful in the case of a transient failure or overload. Moreover, we found that the incoming traffic volume does not affect service resiliency if the circuit breaker is well tuned.

RQ2 asks *"How should retry mechanisms be configured to improve the resiliency of microservice-based applications with complex topologies, with and without circuit breakers?"*

Using the previous answer, we answered this question in section 4.2. This section starts by investigating the impact of enabling a retry mechanism while varying the circuit breaker queue length. We also studied the impact of varying the number of retry attempts and how they can affect system performance in term of response codes and response times. Then we investigated how varying the retry interval can improve resiliency. Finally, we studied the impact



(a) Throughput achieved in terms of successful response codes (green solid lines) and circuit broken requests (orange dashed lines), and failed requests (red dashed dotted lines).



(b) Cumulative density function of response times of carried throughput (circle marker) and all requests (triangle marker) from the outside client perspective when the retry mechanism is enforced on tier 1 (blue and orange loosely dotted lines), tier 2 (green and red dashed lines), tier 3 (purple and brown densely dotted lines), and all tiers (pink and gray solid lines).

Figure 12: Impact of varying the enforcement points of the retry mechanism and circuit breaker while keeping the circuit breaker's maximum queue length (20), retry mechanism settings (2 attempts with 5 seconds interval), and overload condition (100% of capacity) constant.

of varying the retry mechanism's enforcement point and the circuit breaker pattern simultaneously.

Based on our experiments, if a sensitive circuit breaker (i.e., one with a short queue length) is used, the simultaneous use of a retry mechanism may lead to a retry storm, which is expected. If the circuit breaker is not properly configured, a retry mechanism may, surprisingly, increase throughput during load spikes. We also learned that a high number of retry attempts is unhelpful and may diminish the user experience by increasing the response time. The same is true for a high retry interval, while an overly short retry interval may worsen overloads or failures. We conclude that retry mechanisms are useful only for managing transient failures or overload situations.

RQ3 asks *"How do workload characteristics impact the answers to RQ1 and RQ2?"*

We answer this question in sections 4.1 and 4.2. We first studied the impact of the incoming traffic volume on resiliency with a circuit breaker active but no retry mechanism. Then we investigated how workload spikes affect resiliency when a retry mechanism is enforced with different retry intervals. Furthermore we studied the impact of different workload spikes.

Based on the results of our experiments, overload is inevitable if the incoming traffic volume exceeds the capacity of the resources. During a transient overload, a circuit breaker can enable fast-failure, which improves the user experience. We also conclude that load spikes can be controlled by using proper circuit breaker configuration in conjunction with a small number of permitted retry attempts with a retry interval that is neither too low nor too high. We believe that if the overload situation is not transient, then the circuit breaker pattern and retry mechanism are not good choices for maintaining an acceptable user experience.

5 OUTLOOK

This paper studies service mesh traffic management policies for microservices to provide guidance on the practical use of these policies and to show how they can increase application performance and resiliency. The service mesh landscape is rapidly evolving and some features available in proxy sidecars cannot be controlled through the service mesh control plane; examples include the time out budget for individual requests, the back-off method for retries, and caching at the sidecar proxies [19] [20]. Because microservice performance depends on many factors, configuring traffic management policies well can be challenging [25]. The use of a service mesh enables outstanding observability without imposing any particular implementation costs during the development process, which suggests that it may be beneficial to develop methods for autonomous control of the service mesh. We therefore propose to build on the results presented herein by developing a controller to manage service mesh traffic management policies.

6 ACKNOWLEDGMENTS

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We thank the anonymous reviewers for their suggestions and comments.

REFERENCES

[1] M. Akbarisamani. 2019. *Service Based Architecture with Service Mesh Platform In The Context Of 5G Core.* Master's thesis. Tampere University.

[2] G. Aquino, R. Queiroz, G. Merrett, and B. Al-Hashimi. 2019. The circuit breaker pattern targeted to future iot applications. In *ICSOC 2019*. Springer, Switzerland, 390–396.

[3] S. Baltes and P. Ralph. 2020. Sampling in software engineering research: A critical review and guidelines. arXiv:arXiv:2002.07764

[4] Online Boutique. 2021. Sample cloud-native application. https://github.com/GoogleCloudPlatform/microservices-demo

[5] R. Chandramouli and Z. Butcher. 2020. Building secure microservices-based applications using service-mesh architecture. *NIST* 800 (2020), 204A.

[6] B. Dab, I. Fajjari, M. Rohon, C. Auboin, and A. Diquélou. 2020. Cloud-native service function chaining for 5G based on network service mesh. In *ICC 2020*. IEEE, USA, 1–7.

[7] N. Dattatreya Nadig. 2019. *Testing Resilience of Envoy Service Proxy with Microservices.* Master's thesis. KTH, School of Electrical Engineering and Computer Science.

[8] C. Davis. 2019. *Cloud Native Patterns: Designing change-tolerant software.* Simon and Schuster, USA.

[9] N. Dragoni et al. 2017. *Microservices: Yesterday, Today, and Tomorrow.* Springer, Switzerland, 195–216.

[10] A. El Malki and U. Zdun. 2019. Guiding architectural decision making on service mesh based microservice architectures. In *ECSA 2019*. Springer, Switzerland, 3–19.

[11] Envoy. 2021. Router configuration. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/router_filter

[12] Y. Gan et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ASPLOS '19* (Providence, RI, USA). ACM, USA, 3–18.

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

[14] Istio. 2021. Istio - Connect, secure, control, and observe services. https://istio.io/

[15] Istio. 2021. Istio / Bookinfo Application. https://istio.io/latest/docs/examples/bookinfo/

[16] A. Jindal, V. Podolskiy, and M. Gerndt. 2019. Performance modeling for cloud microservice applications. In *ICPE '19*. ACM, USA, 25–32.

[17] M. Kang, J. Shin, and J. Kim. 2019. Protected Coordination of Service Mesh for Container-Based 3-Tier Service Traffic. In *ICOIN '19*. IEEE, USA, 427–429.

[18] C. Klein, M. Maggio, K. Årzén, and F. Hernández-Rodriguez. 2014. Brownout: Building More Robust Cloud Applications. In *ICSE '14*. ACM, USA, 700–711.

[19] L. Larsson, W. Tärneberg, C. Klein, M. Kihl, and E. Elmroth. 2021. Towards Soft Circuit Breaking in Service Meshes via Application-agnostic Caching. arXiv:2104.02463

[20] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han. 2019. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In *SOSE '19*. IEEE, USA, 122–1225.

[21] D. Madory. 2021. Facebook's historic outage, explained. https://www.kentik.com/blog/facebooks-historic-outage-explained

[22] N. C. Mendonca, C. M. Aderaldo, J Camara, and D. Garlan. 2020. Model-Based Analysis of Microservice Resiliency Patterns. In *ICSA '2020*. IEEE, USA, 114–124.

[23] M. Montesi and J. Weber. 2016. Circuit Breakers, Discovery, and API Gateways in Microservices. arXiv:1609.05830 [cs.SE]

[24] K.Y. Ponomarev. 2019. Attribute-Based Access Control in Service Mesh. In *Dynamics '19*. IEEE, Russia, 1–4.

[25] M.R. Saleh Sedghpour, C. Klein, and J. Tordsson. 2021. Service Mesh Circuit Breaker: From Panic Button to Performance Management Tool. In *HAOC '21*. ACM, USA, 4–10.

[26] M.R. Saleh Sedghpour, C. Klein, and J. Tordsson. 2022. *The repository for traffic management policies.* https://doi.org/10.5281/zenodo.5834963

[27] K. Surendro and W.D. Sunindyo. 2021. Circuit Breaker in Microservices: State of the Art and Future Prospects. In *MSE*, Vol. 1077. IOP, UK, 1–10.

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

[29] Ł. Wojciechowski et al. 2021. NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh. In *INFOCOM '21*. IEEE, USA, 1–9.

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