Chaos Engineering for Cloud-Native Resilience Evaluation

Kexin Yu

Nanjing University of Information Science and Technology, Nanjing, China
1217688018@qq.com

Abstract. Cloud computing has become the dominant paradigm for deploying large-scale applications due to its scalability, elasticity, and cost efficiency. However, cloud-native systems built on containers and Kubernetes are inherently complex and prone to failures such as pod crashes, network delays, and node outages. Traditional testing techniques are often insufficient to expose these failures before systems are deployed into production. Chaos engineering has recently emerged as an effective approach to proactively evaluate system resilience by injecting controlled faults into running systems. In this report, we investigate Chaos Mesh, a cloud-native chaos engineering platform, and explore how it can be used to assess the robustness of cloud-native applications. We design a set of fault-injection experiments on a Kubernetes-based microservice application and evaluate system behavior under different failure scenarios. Our results demonstrate that chaos engineering can reveal hidden weaknesses in cloud-native systems and provides valuable insights for improving reliability and fault tolerance.
150–250 words.

Keywords: Chaos Engineering · Chaos Mesh · Kubernetes

1 Introduction

1.1 Background

Cloud computing has transformed the way modern applications are designed and deployed. Instead of relying on monolithic architectures running on a single machine, cloud-native applications are typically composed of multiple microservices deployed in containers and orchestrated by platforms such as Kubernetes. This architecture enables rapid scaling, high availability, and efficient resource utilization. However, the increased level of distribution also introduces new challenges, including complex failure modes that are difficult to predict and test using conventional methods.

In a distributed cloud environment, failures are no longer exceptional events but an expected part of normal operation. Pods may be restarted, nodes may become unavailable, and network latency may fluctuate. While Kubernetes provides built-in mechanisms for self-healing and recovery, these mechanisms must be validated to ensure that applications behave as expected under real-world conditions.

1.2 Motivation

Traditional testing approaches, such as unit testing and integration testing, focus primarily on functional correctness and often assume a stable runtime environment. As a result, they fail to capture the impact of infrastructure-level failures that commonly occur in cloud environments.

Chaos engineering addresses this limitation by intentionally injecting failures into a system to observe its behavior and validate its resilience. Chaos Mesh is a cloud-native chaos engineering platform designed specifically for Kubernetes environments. It allows developers and operators to simulate various failure scenarios, such as pod termination and network delays, in a controlled and reproducible manner.

1.3 Contributions

The main contributions of this project are summarized as follows:

1. I present an overview of chaos engineering and its role in improving the reliability of cloud-native systems.
2. I analyze the architecture of Chaos Mesh and explain how it integrates with Kubernetes.
3. I design and conduct a set of fault-injection experiments to evaluate the resilience of a cloud-native application.
4. I discuss the benefits, limitations, and practical challenges of applying chaos engineering in real-world cloud environments.

2 Literature Review

2.1 Chaos Engineering

Chaos engineering is an experimental approach aimed at improving system resilience by deliberately injecting faults into a running system and observing its behavior under failure conditions. The concept was first popularized by Netflix through the introduction of Chaos Monkey, a tool that randomly terminates production instances to validate system robustness against unexpected failures [1].

Basiri et al. [2] formally defined chaos engineering as a discipline that enables organizations to build confidence in a system's ability to withstand turbulent conditions in production environments. Unlike traditional fault tolerance testing, which relies on predefined test cases and controlled environments, chaos engineering emphasizes experimentation in live or production-like systems. This approach allows engineers to uncover hidden dependencies, cascading failures, and incorrect assumptions that may not surface during conventional testing phases.

Recent studies show that integrating chaos engineering into continuous delivery pipelines can significantly improve system reliability by continuously validating system behavior under fault scenarios [3, 4]. As cloud-native systems grow in scale and complexity, chaos engineering has evolved from an ad-hoc practice into a structured methodology supported by specialized tools and frameworks.

2.2 Reliability Challenges in Cloud-Native Systems

Cloud-native applications are commonly built using microservice architectures and deployed in containerized environments managed by orchestration platforms such as Kubernetes. While this paradigm offers scalability, elasticity, and rapid development, it also introduces new reliability challenges [5].

In microservice-based systems, application functionality is distributed across loosely coupled services that communicate over the network. As a result, partial failures, network delays, and resource contention become normal operational conditions rather than exceptional events. Kubernetes provides several built-in mechanisms to improve baseline availability, including pod replication, automatic restarts, health probes, and service discovery [6]. However, these mechanisms primarily address infrastructure-level failures and may not capture application-level resilience issues.

Dragoni et al. [5] point out that microservices significantly increase the failure surface area, as each service introduces additional dependencies and communication paths. Consequently, failures can propagate across services, leading to cascading effects and performance degradation. Traditional testing approaches, such as unit and integration testing, are insufficient to capture these complex failure dynamics because they assume stable infrastructure and predictable execution environments.

2.3 Chaos Engineering Tools for Kubernetes

The growing adoption of chaos engineering has led to tools like Chaos Mesh, LitmusChaos, and Gremlin for Kubernetes environments, enabling fault injection in cloud-native systems.

Chaos Mesh, an open-source platform under the CNCF, uses Kubernetes Custom Resource Definitions (CRDs) to define chaos experiments declaratively, without modifying application code. It integrates tightly with Kubernetes for scheduling, monitoring, and managing experiments.

LitmusChaos offers similar Kubernetes-native capabilities with observability tool integration. Gremlin, a commercial platform, extends chaos engineering beyond Kubernetes and provides advanced management and security features.

Despite their widespread use, most studies focus on tool capabilities rather than systematic evaluation. This motivates empirical studies that apply chaos engineering tools like Chaos Mesh to real Kubernetes applications and assess system resilience under various failure scenarios (CNCF Cloud Native Landscape, [7]).

3 System Architecture

3.1 Overview

The system architecture used in this project consists of three main components: a Kubernetes cluster, a target cloud-native application, and the Chaos Mesh

platform. The Kubernetes cluster provides the execution environment for both the application and Chaos Mesh, enabling fault injection at different levels of the system.

3.2 Chaos Mesh Architecture

Chaos Mesh is implemented following a cloud-native architecture and is built on top of Kubernetes Custom Resource Definitions (CRDs). Each chaos experiment is described as a Kubernetes custom resource, allowing chaos scenarios to be managed declaratively through the Kubernetes API [8].

As illustrated in the Chaos Mesh architecture, the platform consists of three primary components:

- **Chaos Dashboard**: The visualization and interaction component of Chaos Mesh. It provides a web-based user interface that allows users to create, manage, and observe chaos experiments. The dashboard also integrates role-based access control (RBAC) mechanisms to regulate user permissions.
- **Chaos Controller Manager**: The core control-plane component responsible for managing the lifecycle of chaos experiments. It monitors changes to Chaos Mesh CRDs via the Kubernetes API Server and schedules corresponding actions. This component includes multiple controllers, such as the Workflow Controller, Scheduler Controller, and fault-specific controllers (e.g., PodChaos, NetworkChaos, StressChaos).
- **Chaos Daemon**: The execution component responsible for performing actual fault injections. Chaos Daemon runs on each node in the cluster as a DaemonSet and operates with elevated privileges by default. It injects faults by entering the target Pod's namespace and manipulating system-level resources, such as network interfaces, file systems, and CPU or memory usage.

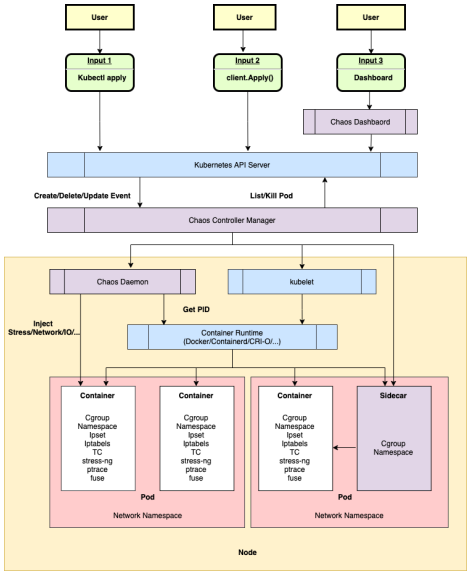


Fig. 1. Overall architecture of Chaos Mesh in a Kubernetes environment.

From a workflow perspective, user operations are first submitted through the Chaos Dashboard or Kubernetes command-line tools and are translated into Chaos Mesh custom resources. The Chaos Controller Manager observes these resource changes via the Kubernetes API Server and determines the appropriate execution strategy. Finally, the Chaos Daemon performs the specified fault injections on the selected nodes or Pods, such as introducing network delay, packet loss, or resource stress.

This layered architecture enables Chaos Mesh to integrate seamlessly with Kubernetes while maintaining strong isolation between control logic and fault execution.

3.3 Target Application

The target application used in this project is a simple microservice-based web service deployed as a Kubernetes Deployment. The application exposes an HTTP endpoint and runs multiple replicas to support load balancing and fault tolerance. This simplified setup allows us to focus on the impact of injected faults rather than application-specific logic.

4 Experiment Setup and Performance Evaluation

4.1 Experimental Environment

The experiments were conducted in a cloud-native environment leveraging Kubernetes as the container orchestration platform, and Chaos Mesh for fault injec-

tion. The environment was set up on a Kubernetes cluster with multiple nodes to ensure high availability and resource flexibility. The nodes in the cluster were provisioned with sufficient resources (CPU, memory, and network bandwidth) to simulate real-world production environments.

The system under test (SUT) consisted of several microservices deployed in Pods within Kubernetes. These microservices were primarily built using containerized NGINX instances to simulate web traffic handling and microservice interactions. The underlying infrastructure included monitoring tools such as Prometheus and Grafana for real-time metrics collection, and Kubernetes-native health checks were configured to detect and respond to failures.

The Chaos Mesh tool was used to simulate various fault injection scenarios at different layers of the system to evaluate its resilience.

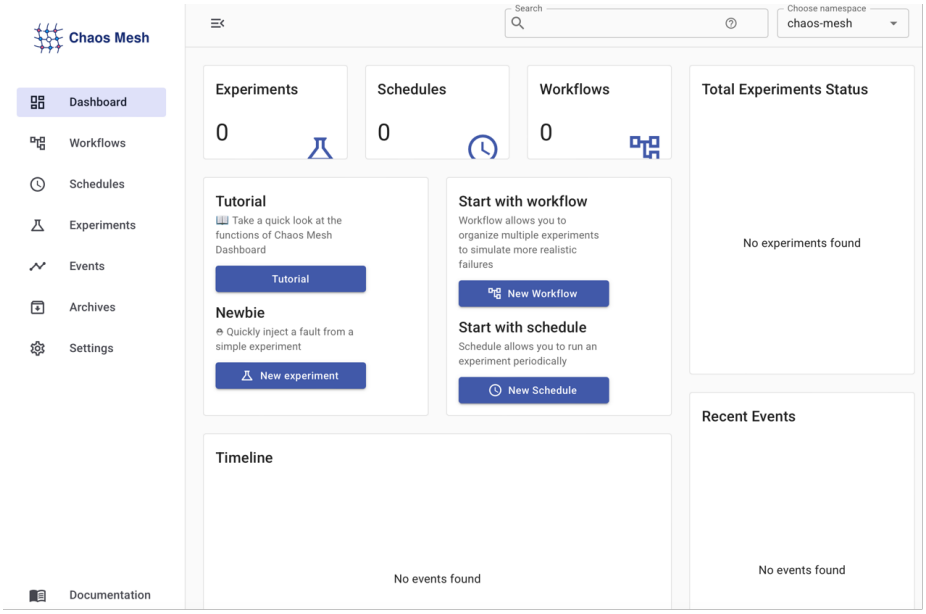


Fig. 2. Chaos Mesh dashboard showing fault injection experiments and system status.

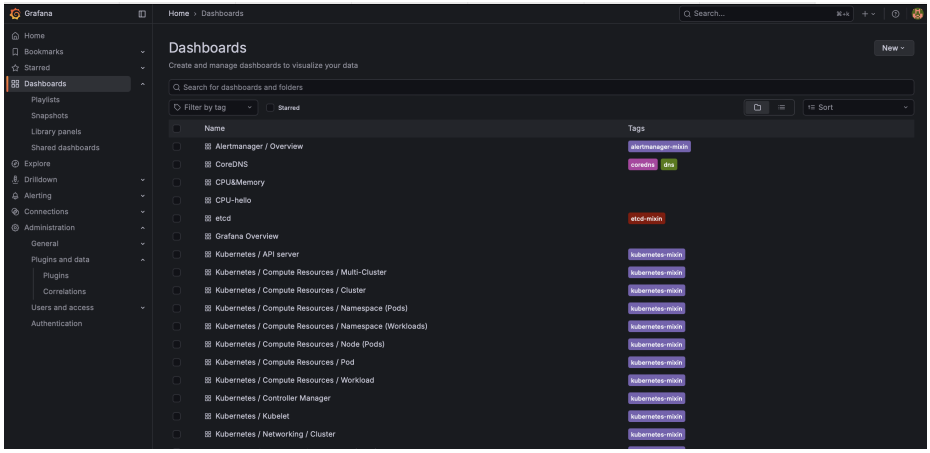


Fig. 3. Grafana dashboard visualizing system performance metrics during experiments.

4.2 Fault Injection Scenarios

This experiment used fault injection scenarios to simulate different failure conditions across various layers of the cloud-native architecture to evaluate system resilience. We employed the Chaos Mesh framework, a tool commonly used in Kubernetes environments to test cloud-native application robustness [9].

- **Pod-Level Failure**: PodChaos was used to randomly terminate and recreate Pods within the Kubernetes cluster. This test aimed to evaluate Kubernetes' ability to reschedule Pods and restore services quickly. Minimal downtime was observed, and the recovery time from the "ContainerCreating" state to full functionality was measured, demonstrating Kubernetes' self-healing capabilities [10].
- **Network-Level Failure**: Using NetworkChaos, network faults like delay and packet loss were introduced. This scenario simulated network instability, which can increase error rates and latency, especially in systems with high inter-service communication [11].
- **Resource Stress Failure**: We applied extreme CPU and memory stress using StressChaos to simulate resource contention. The system showed performance degradation, indicated by slower response times and reduced throughput, but did not crash immediately. This phenomenon, "performance degradation without failure," reflects subtle faults that affect user experience without causing immediate system failures [12]. After sustained stress, Pods failed, highlighting Kubernetes' resource management limits.
- **Application-Level Failure**: We used `JVMChaos` to inject errors into JVM method execution. The application-layer faults are harder to detect since they occur within the microservices rather than the infrastructure.

These fault injection scenarios, designed to emulate real-world failure conditions, helped assess the system's resilience and ability to detect, mitigate, and recover

from disruptions. The results provide valuable insights into the strengths and weaknesses of the system's fault tolerance mechanisms, guiding future improvements for more resilient cloud-native architectures [13].

4.3 Evaluation Metrics

To evaluate the performance and resilience of the cloud-native system under various fault injection scenarios, a set of carefully chosen metrics was used. These metrics help quantify the system's behavior in the presence of failures, its ability to recover, and the overall service degradation.

- **Response Latency**: Measures the time to process a request and return a response. Key for assessing performance under faults like network delays or resource contention. Increased latency often indicates performance degradation, especially under high load or degraded conditions.
- **Error Rate**: Tracks the percentage of failed requests. Helps assess system reliability and stability under failure conditions, such as network partitions or resource stress, by monitoring failed requests, timeouts, and service unavailability.
- **Recovery Time**: The time taken to return to normal operation after a fault. Short recovery times indicate resilience, while long times suggest the need for better fault tolerance mechanisms.
- **Resource Utilization**: Monitors CPU, memory, and disk usage during experiments. High usage can signal stress, potentially leading to performance degradation or failures, and helps understand the system's ability to manage workloads under stress.
- **Service Availability**: The percentage of time the system is fully operational. A high availability rate shows the system can continue serving requests under faults, while low availability indicates significant disruptions. Essential for cloud-native applications.

4.4 Results

1. Pod-Level Failure In this experiment, Pod-level faults (PodChaos) were injected to simulate anomalies during the creation and running of Pods. The results showed that Pods initially entered the "ContainerCreating" state but quickly transitioned to running. Despite the increased number of injected Pods, the system was able to recover rapidly. This experiment demonstrated that Pod-level failures have minimal impact on the system, and Kubernetes' self-healing capabilities effectively mitigate the effects of such faults.

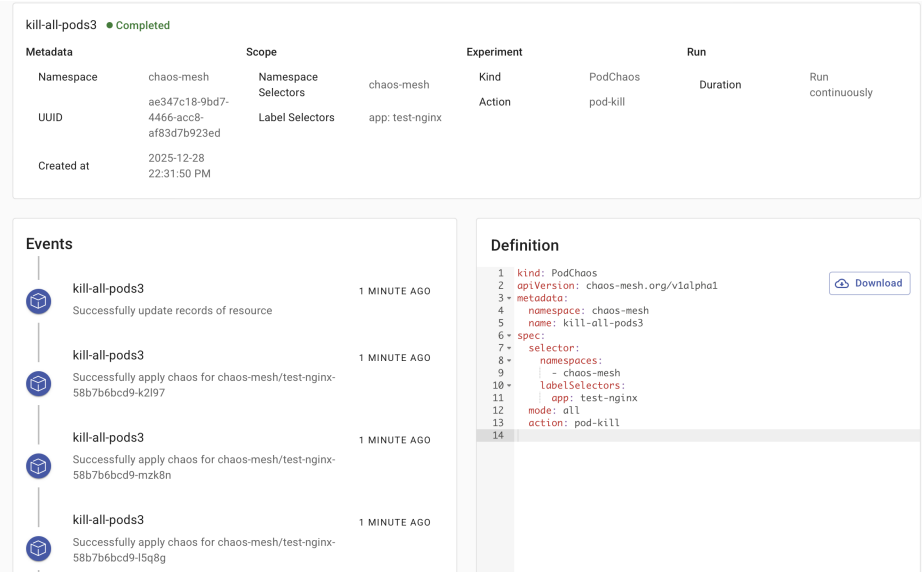


Fig. 4. Chaos Mesh dashboard showing the successful execution of the `PodChaos` experiment, where multiple Pods were terminated simultaneously.

```
(base) niusu@unius-3 ~ % kubectl get pods -n chaos-mesh --watch
NAME                                        READY   STATUS              RESTARTS        AGE
chaos-controller-manager-6b4475fb4f-8llbd   1/1     Running             17 (49m ago)    15h
chaos-controller-manager-6b4475fb4f-cv77d   1/1     Running             16 (53m ago)    15h
chaos-controller-manager-6b4475fb4f-l76fb   1/1     Running             15 (49m ago)    15h
chaos-daemon-6spgw                          1/1     Running             2 (49m ago)     15h
chaos-dashboard-6b9c9c6ff7-kdkzs            1/1     Running             2 (49m ago)     15h
chaos-dns-server-6dfff77f97-mscpg           1/1     Running             10 (49m ago)    15h
test-nginx-58b7b6bcd9-k2l97                 1/1     Running             0               32s
test-nginx-58b7b6bcd9-l5q8g                 1/1     Running             0               32s
test-nginx-58b7b6bcd9-mzk8n                 1/1     Running             0               32s
test-nginx-58b7b6bcd9-l5q8g                 1/1     Terminating         0               71s
test-nginx-58b7b6bcd9-l5q8g                 1/1     Terminating         0               71s
test-nginx-58b7b6bcd9-mzk8n                 1/1     Terminating         0               71s
test-nginx-58b7b6bcd9-t58m8                 0/1     Pending             0               0s
test-nginx-58b7b6bcd9-mzk8n                 1/1     Terminating         0               71s
test-nginx-58b7b6bcd9-t58m8                 0/1     Pending             0               0s
test-nginx-58b7b6bcd9-sddgm                 0/1     Pending             0               0s
test-nginx-58b7b6bcd9-k2l97                 1/1     Terminating         0               71s
test-nginx-58b7b6bcd9-k2l97                 1/1     Terminating         0               71s
test-nginx-58b7b6bcd9-sddgm                 0/1     Pending             0               0s
test-nginx-58b7b6bcd9-t58m8                 0/1     ContainerCreating   0               0s
test-nginx-58b7b6bcd9-cwhc7                 0/1     Pending             0               0s
test-nginx-58b7b6bcd9-sddgm                 0/1     ContainerCreating   0               0s
test-nginx-58b7b6bcd9-cwhc7                 0/1     Pending             0               0s
test-nginx-58b7b6bcd9-cwhc7                 0/1     ContainerCreating   0               1s
test-nginx-58b7b6bcd9-cwhc7                 1/1     Running             0               2s
test-nginx-58b7b6bcd9-t58m8                 1/1     Running             0               2s
test-nginx-58b7b6bcd9-sddgm                 1/1     Running             0               2s
```

Fig. 5. Pod recovery process after fault injection, showing that three terminated Pods were recreated and returned to the `Running` state within approximately 2 seconds.

```
This is ApacheBench, Version 2.3 <$Revision: 1903618 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)...apr_socket_recv: Connection reset by peer
(54)
```

Fig. 6. ApacheBench output during pod termination, where client requests experienced a *connection reset by peer* error when executing `ab -n 100 -c 10 http://127.0.0.1:53593/`.

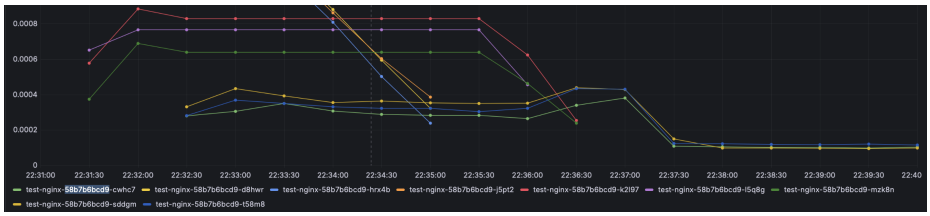


Fig. 7. Grafana CPU usage panel during the PodChaos experiment. The CPU metrics of the original Pods disappeared immediately after pod termination, followed by the appearance of three newly created Pods with restored CPU usage, indicating successful pod recreation and service recovery by Kubernetes.

2. Network-level faults This experiment injected network-level faults using Chaos Mesh, including network delay and packet loss, to evaluate microservice resilience under unstable network conditions. ApacheBench (`ab`) was used to generate concurrent requests and measure throughput and response latency.

Under normal conditions, the system showed stable performance, achieving approximately 940 requests per second with an average response time of 10.6 ms and no failed requests. After introducing network delay, response latency increased significantly to about 286 ms, and throughput dropped to around 35 requests per second.

With packet loss enabled, performance degradation became more severe. The average response time increased to approximately 608 ms, and throughput further decreased to around 16 requests per second. Despite the substantial performance impact, no request failures or system crashes were observed.

Overall, network-level faults mainly caused performance degradation rather than service failure.

```
Document Path:          /
Document Length:        615 bytes

Concurrency Level:      10
Time taken for tests:   0.148 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      84800 bytes
HTML transferred:       61500 bytes
Requests per second:    674.25 [#/sec] (mean)
Time per request:       14.831 [ms] (mean)
Time per request:       1.483 [ms] (mean, across all concurrent requests)
Transfer rate:          558.36 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.2      0       1
Processing:     1   13  18.9      4      55
Waiting:        1    8  12.3      3      55
Total:          2   13  19.1      4      55
```

Fig. 8. Baseline performance under normal network conditions, showing low latency and high throughput with no failed requests.

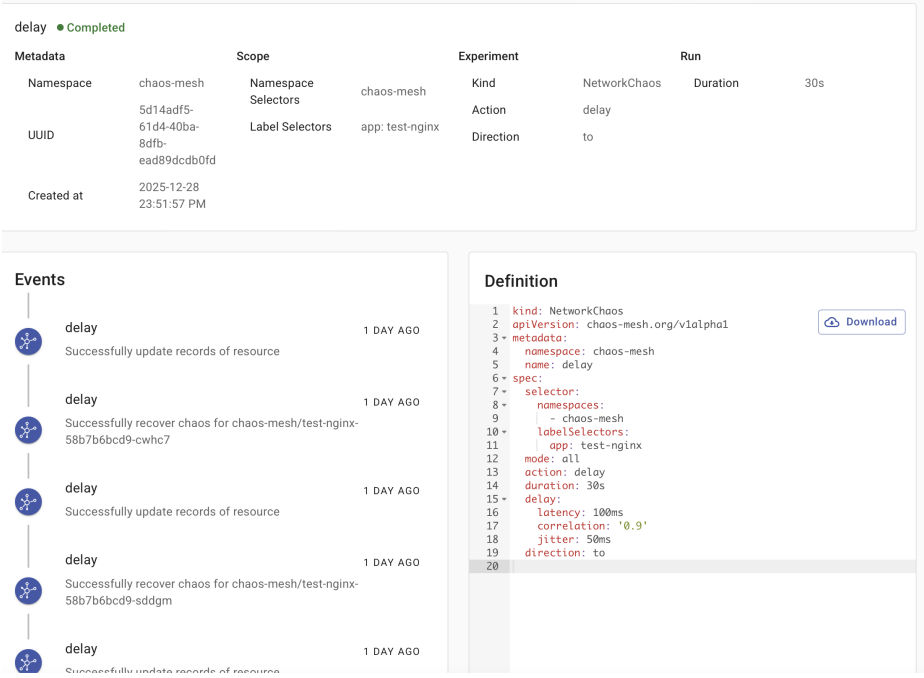


Fig. 9. Chaos Mesh configuration injecting artificial network delay into the target Pods.

```
Server Software:        nginx/1.28.1
Server Hostname:        127.0.0.1
Server Port:            50473

Document Path:          /
Document Length:        615 bytes

Concurrency Level:      10
Time taken for tests:   2.955 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      84800 bytes
HTML transferred:       61500 bytes
Requests per second:    33.84 [#/sec] (mean)
Time per request:       295.505 [ms] (mean)
Time per request:       29.550 [ms] (mean, across all concurrent requests)
Transfer rate:          28.02 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       1
Processing:   165  253  36.7    247     342
Waiting:      139  230  41.8    226     307
Total:        165  253  36.7    248     343
```

Fig. 10. ApacheBench results under network delay, showing increased response time and reduced throughput without request failures.

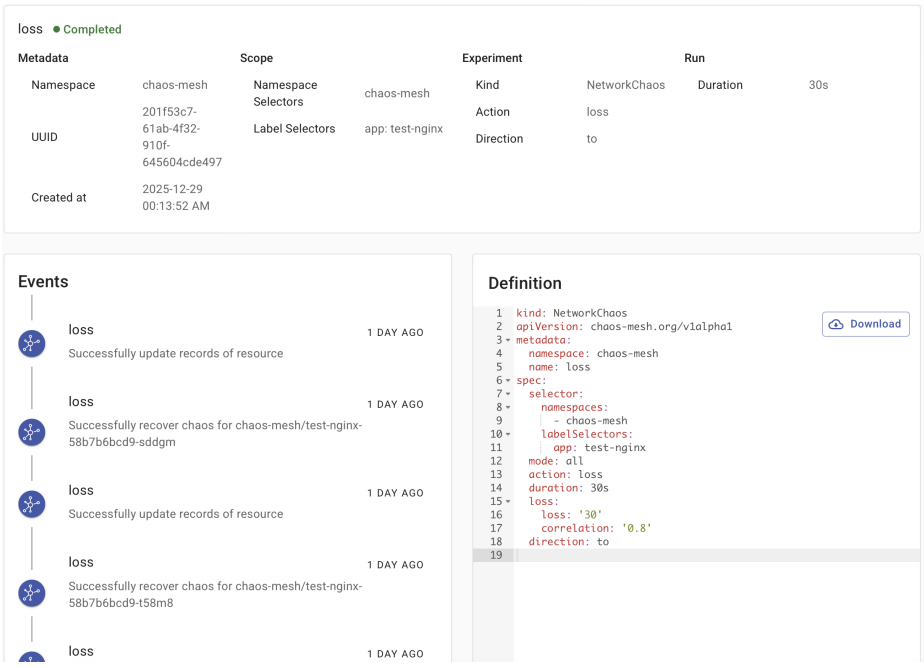


Fig. 11. Chaos Mesh configuration injecting packet loss into the target Pods.

```
Server Software:        nginx/1.28.1
Server Hostname:        127.0.0.1
Server Port:            50473

Document Path:          /
Document Length:        615 bytes

Concurrency Level:      10
Time taken for tests:   6.078 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      84800 bytes
HTML transferred:       61500 bytes
Requests per second:    16.45 [#/sec] (mean)
Time per request:       607.779 [ms] (mean)
Time per request:       60.778 [ms] (mean, across all concurrent requests)
Transfer rate:          13.63 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       0
Processing:     3  527 726.1    216    3810
Waiting:        2  426 691.9     46    3810
Total:          3  528 726.1    216    3811
```

Fig. 12. ApacheBench results under packet loss, where response latency further increased and throughput dropped significantly, while the service remained available.

3. Resource Stress In this experiment, CPU and memory stress were injected using StressChaos to simulate severe resource contention in the Kubernetes cluster. The objective was to observe system behavior under sustained resource pressure.

The results showed that as resource utilization increased, system performance degraded significantly, with response latency rising and service responsiveness decreasing. However, the system did not immediately crash or trigger pod restarts, illustrating the phenomenon of *performance degradation without failure*. As the stress continued, all Pods eventually became unavailable, and the monitoring stack (including Grafana) also failed to respond, preventing further metric visualization.

Additionally, one node experienced persistent pod startup failures, indicating that extreme resource exhaustion can propagate beyond application services and affect cluster-level components. This experiment demonstrates that under resource stress, cloud-native systems may remain running while becoming effectively unusable, revealing limitations of Kubernetes health checks in detecting such failures.

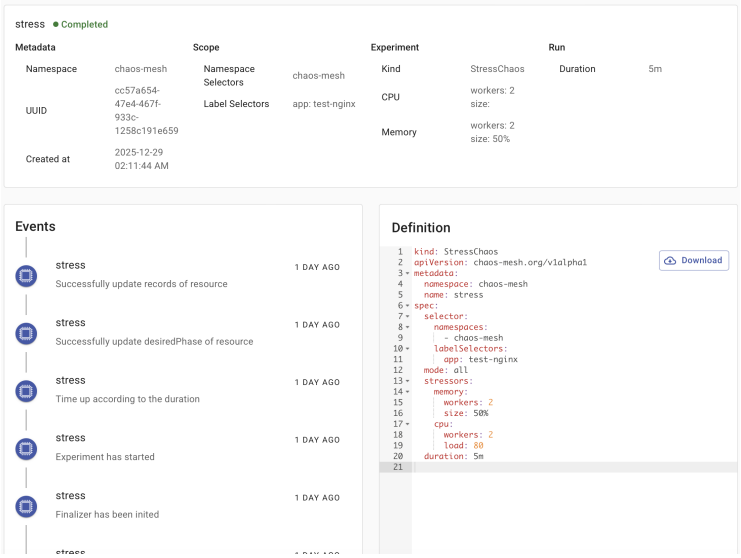


Fig. 13. Chaos Mesh dashboard showing the activation of StressChaos for CPU and memory stress injection.

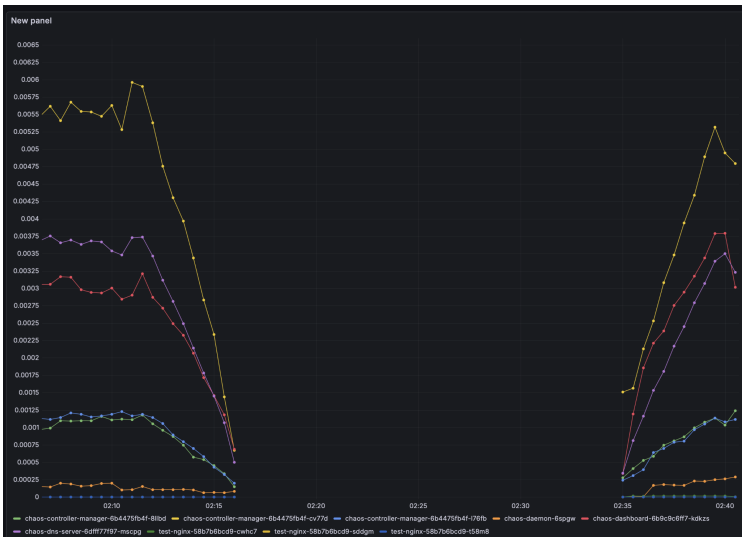


Fig. 14. The CPU usage of Pods with Nginx in the Chaos Mesh namespace becoming unavailable after stress injection, with their status missing.

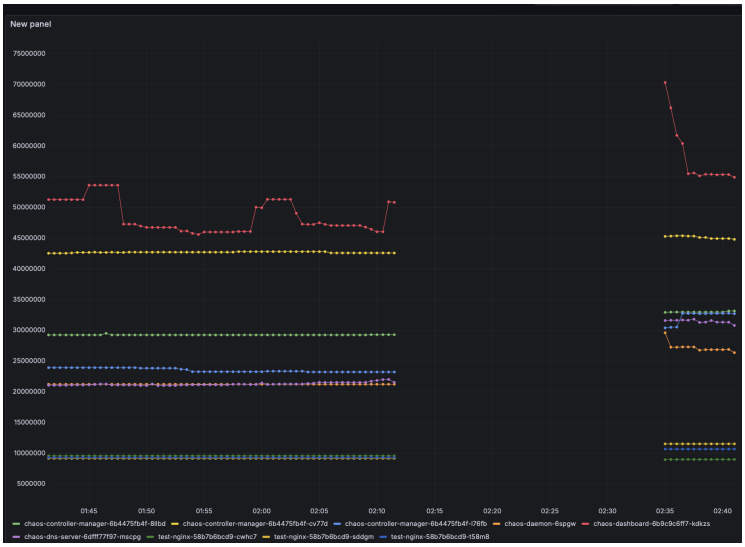
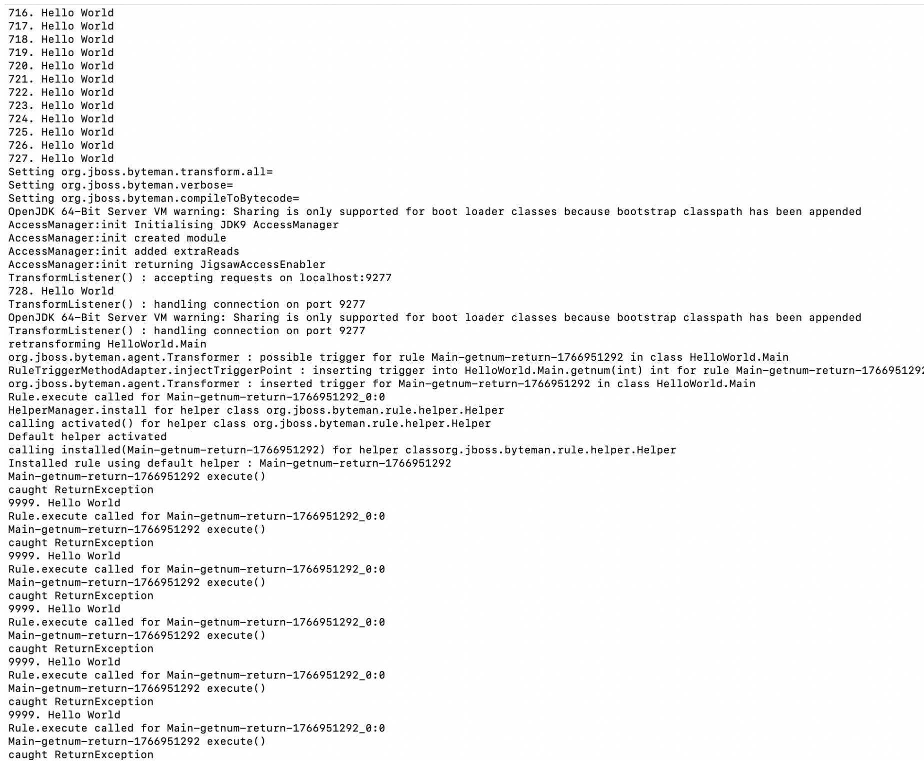


Fig. 15. The memory usage of the same Pods failing to display data under extreme resource exhaustion, resulting in the Pods' status disappearing.

4. Application-level faults This experiment investigated application-level failures using JVMChaos to inject abnormal return values, and JVM method-level disruptions. Unlike Pod crashes, which can be readily detected and recovered by Kubernetes, JVM-level faults remained invisible to the orchestration layer. In the Java microservice environment, JVMChaos successfully intercepted and modified the execution of internal methods without causing Pod restarts or state transitions.

Experimental logs from the HelloWorld service showed repeated abnormal outputs after fault injection, confirming that the application's execution flow had been altered at runtime. During this period, CPU and memory usage increased noticeably, indicating performance degradation and additional execution overhead. However, Kubernetes did not perceive these internal JVM exceptions or behavioral anomalies, and the affected Pods remained in the `Running` state. These results demonstrate that application-level failures pose a significant observability challenge and can degrade system performance without triggering infrastructure-level recovery mechanisms.



```
716. Hello World
717. Hello World
718. Hello World
719. Hello World
720. Hello World
721. Hello World
722. Hello World
723. Hello World
724. Hello World
725. Hello World
726. Hello World
727. Hello World
Setting org.jboss.byteman.transform.all=
Setting org.jboss.byteman.verbose=
Setting org.jboss.byteman.compileToBytecode=
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
AccessManager:init Initialising JDK9 AccessManager
AccessManager:init created module
AccessManager:init added extraReads
AccessManager:init returning JigsawAccessEnabler
TransformListener() : accepting requests on localhost:9277
728. Hello World
TransformListener() : handling connection on port 9277
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
TransformListener() : handling connection on port 9277
retransforming HelloWorld.Main
org.jboss.byteman.agent.Transformer : possible trigger for rule Main-getnum-return-1766951292 in class HelloWorld.Main
RuleTriggerMethodAdapter.injectTriggerPoint : inserting trigger into HelloWorld.Main.getnum(int) int for rule Main-getnum-return-1766951292
org.jboss.byteman.agent.Transformer : inserted trigger for Main-getnum-return-1766951292 in class HelloWorld.Main
Rule.execute called for Main-getnum-return-1766951292_0:0
HelperManager.install for helper class org.jboss.byteman.rule.helper.Helper
calling activated() for helper class org.jboss.byteman.rule.helper.Helper
Default helper activated
calling installed(Main-getnum-return-1766951292) for helper classorg.jboss.byteman.rule.helper.Helper
Installed rule using default helper : Main-getnum-return-1766951292
Main-getnum-return-1766951292 execute()
caught ReturnException
9999. Hello World
Rule.execute called for Main-getnum-return-1766951292_0:0
Main-getnum-return-1766951292 execute()
caught ReturnException
9999. Hello World
Rule.execute called for Main-getnum-return-1766951292_0:0
Main-getnum-return-1766951292 execute()
caught ReturnException
9999. Hello World
Rule.execute called for Main-getnum-return-1766951292_0:0
Main-getnum-return-1766951292 execute()
caught ReturnException
9999. Hello World
Rule.execute called for Main-getnum-return-1766951292_0:0
Main-getnum-return-1766951292 execute()
caught ReturnException
9999. Hello World
Rule.execute called for Main-getnum-return-1766951292_0:0
Main-getnum-return-1766951292 execute()
caught ReturnException
```

Fig. 16. Application logs of the HelloWorld service during the JVMChaos experiment. After fault injection, the output value changed from the normal sequence to repeated abnormal values (e.g., `9999`), indicating successful manipulation of the JVM method execution flow.

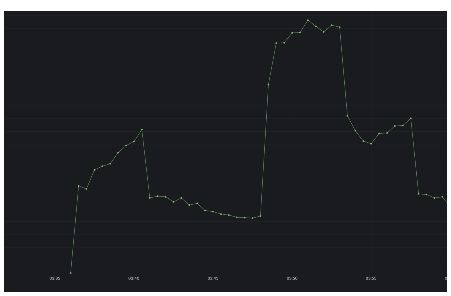


Fig. 17. CPU usage of the HelloWorld Pod during the JVMChaos experiment.

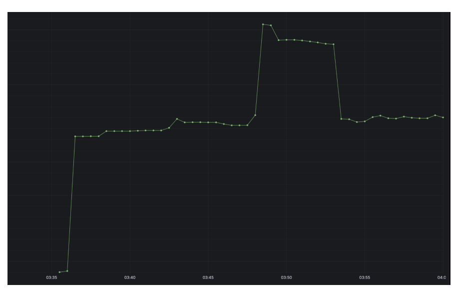


Fig. 18. Memory usage of the HelloWorld Pod under JVMChaos.

4.5 Evaluation

This section evaluates the resilience of the cloud-native application based on the results of the chaos experiments conducted at different fault levels, including Pod-level failures, network-level faults, resource stress, and application-level faults. The evaluation focuses on system availability, performance impact, recovery behavior, and observability under fault conditions.

Overall, the experimental results show that Kubernetes provides strong resilience against infrastructure-level failures. In the Pod-level failure experiment, terminated Pods were rapidly recreated and returned to the `Running` state within a short time window. Although transient request errors were observed during pod termination, the system quickly recovered without manual intervention, demonstrating the effectiveness of Kubernetes' self-healing mechanisms.

Network-level fault experiments revealed that while the system remained available under adverse network conditions, performance degradation was significant. Both network delay and packet loss caused substantial increases in response latency and sharp reductions in throughput. However, no request failures or service crashes occurred, indicating that the application was tolerant to network instability but sensitive in terms of quality of service. This highlights that availability alone is insufficient to characterize resilience, and performance metrics must also be considered.

The resource stress experiment exposed a critical limitation of Kubernetes health management. Under sustained CPU and memory pressure, the system exhibited severe performance degradation without immediately triggering pod restarts or failure detection. Eventually, both application services and monitoring components became unresponsive. This phenomenon demonstrates the existence of *gray failures*, where services remain technically running but are effectively unusable. Such failures are difficult to detect using standard liveness and readiness probes.

Application-level fault injection using JVMChaos further emphasized observability challenges. JVM method interception successfully altered application behavior and introduced abnormal outputs and increased resource consumption, yet Kubernetes did not detect any anomalies at the orchestration level. The affected Pods remained in the `Running` state, and no recovery actions were triggered. This confirms that application-level faults can silently degrade system correctness and performance, bypassing infrastructure-level monitoring and recovery mechanisms.

In summary, the evaluation shows that while Kubernetes excels at handling explicit infrastructure failures, it has limited visibility into performance degradation and application-level anomalies. These results demonstrate the necessity of incorporating chaos engineering techniques and application-aware monitoring to comprehensively evaluate and improve the resilience of cloud-native systems.

5 Discussion and Limitations

5.1 Discussion

The experimental results demonstrate that chaos engineering is an effective and practical approach for evaluating the resilience of cloud-native applications. By intentionally injecting faults under controlled conditions, chaos engineering enables developers and operators to observe system behavior beyond normal execution paths and to validate the effectiveness of recovery mechanisms.

The experiments show that Kubernetes exhibits strong self-healing capabilities when dealing with explicit infrastructure-level failures, such as pod termination and transient network disruptions. In these scenarios, the system was able to maintain service availability or recover rapidly without manual intervention. However, the results also highlight that resilience cannot be assessed solely based on service availability. Network faults and resource stress caused significant performance degradation, revealing that a system may remain operational while delivering unacceptable quality of service.

Moreover, the application-level fault injection experiments using JVMChaos exposed a critical observability gap. While Kubernetes effectively manages container and pod states, it lacks visibility into internal application logic errors and performance anomalies occurring within the JVM. Such failures can silently degrade correctness and efficiency without triggering any orchestration-level recovery actions. These findings emphasize the importance of combining chaos engineering with application-aware monitoring and more expressive health-check mechanisms.

Overall, this study illustrates that chaos engineering complements traditional testing by focusing on system behavior under failure conditions and provides valuable insights into the practical limits of Kubernetes-based resilience mechanisms.

5.2 Limitations and Challenges

Despite the insights gained, this study has several limitations. First, all experiments were conducted on a small-scale local Kubernetes cluster, which may not fully reflect the complexity, scale, and failure characteristics of production-grade environments. Factors such as multi-node heterogeneity, real-world traffic patterns, and cloud provider-specific behaviors were not considered.

Second, the target application used in the experiments was intentionally simple in order to isolate fault effects and facilitate analysis. More complex, stateful, or highly coupled microservice architectures may exhibit different failure propagation patterns and recovery behaviors.

Finally, the analysis of experimental results relied largely on manual inspection of logs and monitoring dashboards. This approach limits scalability and reproducibility. Integrating automated anomaly detection, richer observability tools, and systematic metrics analysis would improve the robustness and efficiency of future evaluations.

6 Conclusion and Future Work

In this project, we explored Chaos Mesh as a representative cloud-native chaos engineering platform and evaluated its effectiveness in assessing the resilience of Kubernetes-based applications. Through a series of fault-injection experiments across multiple layers, including pod-level failures, network faults, resource stress, and application-level anomalies, we demonstrated how chaos engineering can reveal hidden weaknesses that are not observable through traditional testing methods.

The results indicate that while Kubernetes provides strong recovery mechanisms for infrastructure-level failures, it has limited capability to detect and respond to performance degradation and application-level faults. These findings highlight the necessity of adopting chaos engineering as a complementary practice to conventional testing and monitoring in modern cloud-native systems.

Future work may extend this study by conducting experiments on larger-scale or production-like clusters, incorporating more complex microservice architectures, and automating fault analysis using advanced observability and anomaly detection techniques. In addition, integrating chaos engineering with AI-driven diagnosis and self-healing mechanisms represents a promising direction for improving the resilience and autonomy of cloud-native applications.

References

1. Netflix Technology Blog: The Netflix Simian Army. Available: https://netflixtechblog.com (2011)
2. Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C.: Chaos Engineering. IEEE Software, vol. 33, no. 3, pp. 35–41 (2016)
3. Barletta, V., et al.: Chaos Engineering for Kubernetes: Lessons Learned from Injecting Faults into etcd. Proc. IEEE/ACM DSN (2022)
4. Ergenç, D., et al.: Resilience Evaluation of Edge–Cloud Applications. Future Generation Computer Systems, vol. 127, pp. 189–204 (2022)
5. Dragoni, N., et al.: Microservices: Yesterday, Today, and Tomorrow. Springer (2017), pp. 195–216
6. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes. ACM Queue, vol. 14, no. 1, pp. 70–93 (2016)
7. CNCF: Cloud Native Landscape: Chaos Engineering Tools. Available: https://landscape.cncf.io
8. Chaos Mesh Community: Chaos Mesh documentation. Available at: https://chaos-mesh.org/docs/ (accessed January 2025)
9. Chaos Mesh: Chaos Mesh: A Powerful Chaos Engineering Platform for Kubernetes. https://chaos-mesh.org, last accessed 2023/10/25
10. Taylor, J., Lamb, T.: Cloud-native resilience in Kubernetes-based microservices. In: Proceedings of the International Conference on Cloud Computing, pp. 1–10 (2019)
11. Melikhov, D., Vasilenko, A.: Distributed system fault tolerance and performance degradation under network instability. Journal of Cloud Computing **15**(4), 122–135 (2020)

12. Noble, D., Paul, M.: Performance degradation without failure: An exploration of resource contention in cloud environments. ACM Computing Surveys **52**(7), 95–107 (2018)
13. Afergan, S., Simon, D.: Building resilient microservices with chaos engineering: Insights from real-world applications. Journal of Software Engineering and Applications **12**(5), 185–201 (2022)