National College of Ireland

Enhancing Resilience of Terraform-Provisioned Kubernetes Cluster using Chaos Engineering

MSc Research Project
Cloud Computing

Suyog Rajesh Redkar
Student ID: X23419504

School of Computing
National College of Ireland

Supervisor: Shreyas Setlur Arun

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Suyog Rajesh Redkar
Student ID:	X23419504
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Shreyas Setlur Arun
Submission Due Date:	20/12/2018
Project Title:	Enhancing Resilience of Terraform-Provisioned Kubernetes Cluster using Chaos Engineering
Word Count:	XXX
Page Count:	19

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	28th January 2026

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhancing Resilience of Terraform-Provisioned Kubernetes Cluster using Chaos Engineering

Suyog Rajesh Redkar
X23419504

Abstract

In this study, we have used Terraform for reliable and repeatable infrastructure deployment to automate the provisioning and resilience testing of Kubernetes on Amazon Elastic Kubernetes Service (EKS). LitmusChaos, which is a Chaos Engineering framework was applied to introduce controlled failures into the cluster in order to test system reliability and fault tolerance during disruptive conditions. The finding of the experiment was also used to find weak aspects of recovery methods and application. The test involved the use of Prometheus as a monitoring system to capture performance metrics in real-time to allow an in-depth analysis of the stability of the system. Infrastructure as Code automation and observability-based insights are utilized in this work to prove the significance of proactive resilience testing. The main aim is to enhance the robustness of operations, reveal infrastructure and workload-level vulnerabilities, and develop a systematic method for assessing reliability in cloud-native workloads. This research is beneficial for increasing the use of chaos engineering among current Kubernetes ecosystems.

Keywords: Terraform, Kubernetes, Eks, Chaos Engineering, LitmusChaos, Prometheus

1 Introduction

As the cloud environment rapidly evolves, Infrastructure as Code (IaC) and container orchestration usage have impacted the implementation and maintenance of scalable systems (Yevick and Jones; 2020). Terraform and Kubernetes are becoming key technologies that will help automate and organize a cloud native environment in an efficient manner. These technologies, though powerful, do not have any in-built procedures to take proactive resilience measures in case of failure, and thus, the system is subject to risks like deletion of pods, network delay, and node failures, which may lead to outage of applications and business interruptions. (Murphy; 2022). Kubernetes is a paradigm shift as it automates load balancing, resource allocation, and fault tolerance. Such activities usually require a lot of human intervention. Kubernetes utilizes declarative syntax that enables the repetition of deployments and self-repair, which ensures reliability and availability. Nonetheless, a minconfigured setup or insufficient testing may lead to security breaches or failure to launch a deployment or even a inefficient architectures that require dependable validation techniques (Dame; 2022).

1.1 Background

It is important to test the resilience of Kubernetes cluster to create reliable and highly resilient cloud-native applications. The decentralized and dynamic characteristics of such environments requires more effective testing methods to facilitate the accurate assessment of infrastructure settings and the performance of system under unfavorable circumstances. This type of testing is needed to achieve scalable, secure, and fault tollerant cloud systems and minimize potential production breakdowns that may lead to significant down time and data loss. The addition of resilience testing to the infrastructure lifecycle is necessary to detect faults proactively and enable automatic recovery to redeem maximum resource allocation, enhance security postures, and self healing abilities. Basiri et al. (2016).

LitmusChaos is a Kubernetes native chaos engineering tool that is fundamental in simulating real-life disruptions by injecting controlled failures, exposing weakness in the system. It is a tool that fosters resilient structures that follow the DevOps principels in the sense that enable the organizations to test and enhance fault tollerance in advance before the issues escalate in production. In terms of observability Prometheus is a dependable monitoring platform to collect time-series measures. Therefore End-to-End resilience validation is automated. This make the deployment uniform, fast to fix, paving the way to safe and successful cloud-native operations.(Rosenthal and Jones; 2020).

1.2 Motivation

Considering the significance of the transition to the cloud-native infrastructure and implementing container orchestration software such as Kubernetes in a scalable DevOps pipeline, it was necessary to propose a more effective and improved framework that includes Chaos Engineering to test the resilience of nodes and pods of the Kubernetes cluster proactively. In this study, Terraform and LitmusChaos were combined with Kubernetes to have a dependable solution. Real-time monitoring and visualizations are implemented using Prometheus, and fault tolerance is enhanced using LitmusChaos. One more important element to facilitate sustainable and repeatable deployments is the successful maintenance of Terraform configurations and the implementation of a self-healing mechanism, especially when implementing multi-cloud environments. Combining these issues and applying the best practices of the industry in the cloud context, this work offers an extensive approach to the optimization of the Kubernetes cluster and IaC-managed processes. This Study fills the gap to meet the expectation of the current infrastructure management and enhances the cloud-native methods, ensuring scalability, fault tolerance, and reliability in Kubernetes systems.

1.3 Research Question

Resilience testing is a importance facet in cloud-native infrastructure management. Hence, a baseline framework integrating chaos engineering with a Terraform-provisioned Kubernetes cluster is essential. Therefore, this research investigates:

"To what extent does integration of chaos engineering tools with Kubernetes clusters managed by Terraform enhance the resilience of cloud-native systems?"

Hence, in this study, we intend to address the following problems:

- Implementation of a standardized methodology for automating resilience testing and failure injection in Kubernetes deployments.
- To determine and utilize observability tools like Prometheus and Grafana for performance monitoring, together with Chaos Engineering tools like Litmus Chaos to replicate real-world disruptions like pod deletions, network delay, and node failures.
- Further, in order to measure recovery efficiency, we also want to establish mechanisms for evaluating key resilience metrics, including Mean Time to Recovery (MTTR), system uptime/availability, and resource utilization (CPU and Memory).
- lastly, to identify and implement for fault-tolerance and Iac-driven self healing, making sure that cloud providers interate seamlessly to improve overall system reliability.

1.4 Objectives

The key point of this study is to assess and develop a uniform, computerized approach toward integrating chaos engineering and observability concepts into Terraform-Managed Kubernetes assemblies, which would result in heightened resilience engineering.

- Resilience checking via Chaos Engineering: LitmusChaos can be used to methodically introduce systematic disruptions to a Kubernetes cluster, abd quantitatively evolve recovery patterns and check the robustness.
- Observability-Driven Insights: Combine Prometheus and Grafana to enable real-time, fine-grained monitoring of resilience metrics such as Mean Time to Recovery(MTTR), Uptime, availability, and resource utilization, providing empirical evidence for performance under failure conditions.
- Adaptive Fault Tolerance Development: Reduce downtime and guarantee operational continuity by incorporating self-healing system into Terraform procecsses by with Kubernetes-native functionalities (such as replica sets and auto scaling groups) and extend them with resiliency policies.
- Cross Scalability and Reusability: Construct the framework based on the principles of modular Terraform designs to offer interoperability with most cloud environments and enable enterprises to use resilience validation as a cloud-agnostic strategy that can be reused.

1.5 Contribution

The presented work has its contribution to the research on cloud-native resilience engineering, providing the integration of Terraform-based Infrastructure as Code with Chaos Engineering and Observability as the infrastructure provisioning layer. The paper describes the approach to resilience validation in Kubernetes clusters, contrasting with previous literature that primarily considered Chaos at the application or runtime level. The study demonstrates how to measure the key metrics, such as Mean Time to Recovery (MTTR), uptime, and resource usage in case of stress by making pod and node

failures with the help of LitmuChaos and monitoring recovery with the help of Prometheus. This bridges a significant gap in the literature by extending IaC past provisioning and automation into resilience testing and validation.

Practically, the proposed methodology will offer organizations an automated, repeatable way of ensuring that resiliency checks are included in CI/CD pipelines. Organizations that deploy Kubernetes may apply this method to discover the vulnerabilities proactively, minimize downtimes, and ensure the continuity of services prior to malfunctions of production systems.

1.6 Paper Structure

Title	**Brief Description**
Chapter 1: Introduction	Introduces the research topic, outlines the research gap, and motivation for enhancing Kubernetes resilience using Chaos Engineering.
Chapter 2: Related Work	Focuses on preliminary work that was completed as part of this research. The paper reviews relevant studies to highlight major findings and identify research gaps.
Chapter 3: Methodology	It explains the significance of resilience testing in a Kubernetes Cluster. It also introduces the tools used and the metrics of evaluation.
Chapter 4: Design Specification	This section explains the details of the implementation of the study and the specified architecture diagram.
Chapter 5: Implementation	Explains the practical infrastructure implementation, including Terraform provisioning, Chaos Experiments execution, and monitoring tool setting.
Chapter 6: Evaluation	Discusses the analysis of Kubernetes cluster system behaviors by injecting faults through Chaos Experiments and the improvements made post-fault injection.
Chapter 7: Conclusion and Future Work	Summarises key findings and concludes with a final note of the study outlining the scope for future research.

Table 1: Structure Overview

2 Related Work

Understanding Infrastructure as Code and Kubernetes orchestration is essential for looking into options to improve resilience in cloud-native systems. This section analyzes the relevant existing research in the domain, highlighting advances in chaos engineering systems for fault-tolerant systems. It summarizes previous initiatives to address issues such as system failures and recovery inefficiencies in distributed infrastructures while emphasizing the limitations of these approaches.

Author	Framework	Approach	Advantage	Limitation
(Blohowiak et al.; 2016)	Netflix CAP	Microservices based automated chaos experiments.	Consistant resilience testing	Lack of IaC integration
(Chinnasamy; 2025)	DevOps Chaos in Azure	Kubernetes cluster-wide pod/region failure using CI/CD	Multiple cloud support	Difficult setup, high resource requirements, and potential for unforeseen impacts.
(Basiri et al.; 2016)	Netflix's Chaos Monkey	Injection of faults by arbitrarily terminating production virtual machines.	Enhanced resilience during runtime.	Restricted to the application layer only.
(Torkura et al.; 2020)	CloudStrike	Chaos engineering is used for proactive security and fault injection in cloud systems.	Enhanced resilience and security of infrastructure.	Restricted tool integration
(Hausenblas; 2023)	OpenTelemetry + Prometheus	Microservices monitoring and observability using metrics and traces	Standardized data gathering and complicated hybrid system setup	Complex setup in hybrid systems
(Malik et al.; 2023)	CHESS	Self-healing systems' adaptability was assessed using chaos scenarios.	Supports self-healing.	Emphasis on runtime rather than infrastructure.
(Simonsson et al.; 2021)	Simulator for AWS Fault Injection	FIS was used to introduce controlled faults in a cloud-native environment.	Cloud-native implementation	Lacks control over infrastructure provisioning.
(Saxena et al.; 2024)	IaC pipelines for DevOps	Code-based continious pipeline and infrastructure provisioning.	Robust IaC automation.	Lacks focus on chaos and fault injection.
This Research	**Terraform + Litmus Chaos**	**Infrastructure-level chaos testing with Monitoring**	**Adaptive resilience validation based on IaC**	**Operational costs and difficulty of integration.**

Table 2: Summarising Previous Research Work

2.1 Evolution in Infrastucture as Code(IaC)

The article (Murphy; 2022) has examined the introduction of Infrastructure as Code in enterprise environments, namely the reduction of configuration drift and human error in the cloud installations and infrastructure development by declarative provisioning. The research revealed that even though IaC enhances auditability and repeatability, it is also accompanied by other challanges such as dependency issues and misconfigurations of security. Equally (Hasan and Ansary; 2023) examined automated cloud architecture using IaC tools and found that automated provisioning saves a lot of time in development and produces a lot more consistency in the environment. Despite both these studies pay much attention to automation and compliance without taking into account infrastructure resilience.

An experimental study of scalability and agility in IT operations based on the use of IaC by (MUSTYALA; 2020) showed that the use of declarative architecture reduces the operational overhead and would have to be continuously validated to ensure accuracy.

2.2 Declarative Provisioning in Terraform

Terraform, which was created by HashiCrop, has become an important tool on the in the area of provisioning, managing, and maintaining cloud infrastructure. It applies declarative syntax where the users specify the desired final state of the infrastructure instead of stepwise sequence of actions available to accomplish the state (Yevick and Jones; 2020). This process of declaration provides consistency, repeatability, and transparency in the environment by storing the statefiles of the current infrastructure configuration. These state files allow Terraform to make distinction between the desired configuration and the infrastructure actually deployed. The main advantage of Terraform is that it is cloud-agnostic. Its provider ecosystem is extensive, supporting AWS, Azure, and Google Cloud, because of its strong provider ecosystem. (Turnbull; 2014). This multi-cloud provider's flexiblity allows enterprises to implement hybrid or multi-cloud strategies while using a single provisioning architecture. The modular structure of the Terraform architecture enables the developers to compose the elements of the infrastructure in reusable parts, enhancing sustainability and easing the process of large-scale deployments. Flexiblity of this multi-cloud provider permits the enterprises to create hybrid or multi-cloud strategies using a singular provisioning architecture. (Saxena et al.; 2024) indicated that the architecture is modular, which means that developers can build foundation blocks into reusable modules.

State management is one of the most significant components of provisioning of Terraform. The state file is the main point of truth of the installed infrastructure. Nonetheless, mishandling of statefiles could also lead to configuration drift as mentioned by (Yevick and Jones; 2020), where the deployed resources actually used do not match the stated configuration.

2.3 Orchestration and Scalability in Kubernetes Ecosystems

Kubernetes could be described as an open-source container orchestration system to scale, administer, and deploy application packages in a containerized manner. There is an official Kubernetes documentation [1], which says that the architecture comprises of a

[1] https://kubernetes.io/docs/concepts/architecture/

control plane (that contains: API server, etcd store, controllers, and schedulers) and single or multiple worker nodes which execute the actual application in pods. The control plane establishes the desired state of the cluster, and the worker nodes execute workload and update status on a regular basis to the control plane. Also most recent has begun to examine Kubernetes failure in real fault conditions. The fault/error injection campaign was conducted by (Barletta et al.; 2024) on the data store (etcd), which stores clusterstate and analyzes the types of failures.As they have shown, a small configuration error or corruption of data may lead to cascade failures. scalability under load is another factor.

Kubernetes, too,relies on the infrastructure in support, in particular, state storage. In one of the studies (Larsson et al.; 2020), the researcher studies the impact of etcd deployment decisions on orchestration latency and dependability. As they found, etcd operations executed slowly obstruct the Kubernetes ability to respond promptly to the change, which affects load balancing, scaling, and scheduling of pods.

2.4 Methodologies and Frameworks in Chaos Engineering

Chaos Engineering is a practice that intentionally adds failure to systems to know their vulnerable points, determine how the system can withstand failures, and instruct improvements to reliability. To study the behavior of large-scale distributed systems during stress, (Basiri et al.; 2016) formalized this approach at Netflix by modeling failures, e.g. termination of instances and network outages. Besides informing the system owners on the ways to improve redundancy and failover behaviour, these tests were helpful in the process of finding unnoticed single points of failure. A study empirically conducted by (Al-Said Ahmad et al.; 2024) injection effects on performance and reliability in both. serverless services and containerized applications. To simulate a different user count at once (e.g., 300, 600, 1200 users) the authors injected delay with different magnitudes (hundreds of milliseconds up to tens of seconds) into lambda functions and microservices. Their throughput and latency increased with the increase in user load and delay. CHESS frame work was proposed by (Malik et al.; 2023), a framework that employs microservice systems to evaluate self-adaptive systems in both functional and infrastructure issues. The authors used five fault injection cases to case studies among them Yelb application and Smart Office application. They perceived the way self-observing and service. restarts, and other methods of system adoption assist in maintaining performance and lessen the spread of failures across the microservices.

2.5 Litmus Chaos

LitmusChaos is a Kubernetes based chaos engineering framework created by the Cloud Native Computing Foundation (CNCF) and is used to declare fault injection through Custom Resource Definitions (CRDs). The LitmusChaos is a modular application enabling the user to run experiments on Kubernetes clusters. According to the article published by LitmusChaos, its integration with CI/CD pipelines allows embedding chaotic workflows into the continuous testing environment (Simonsson et al.; 2021). LitmusChaos offers a full-scale resilience testing coverage because it allows application-level and infrastructure-level experiments.

2.6 Prometheus and Grafana in Resilience Contexts

The Reliability and robustness of the cloud-native systems can be tested through monitoring and observability. Prometheus is a time-series monitoring system written in open-source. It has a pull-based paradigm collection of metrics and exporters collection. live information about infrastructure elements, containers, and pods. It enables the ability to fine-grain analysis of performance measurements of a system by storing multidimensional data which is defined by names of metrics and key value labels. According to (Burns; 2024), observability systems such as Prometheus are necessary in understanding the dynamics of distributed systems to dynamic workloads, particularly in systems that are managed such as Kubernetes. Their joint stack of capabilities is a complete observability stack that enables the engineer to associate chaos events with performance degradation by visualizing metrics in real-time (Hausenblas; 2024), Grafana has a compromised dashboarding capability, which simplifies the visualization of time-series information of numerous sources. The authors suggest that by using the connection of Grafana and Kubernetes data, researchers can observe inter-node interactions, resource bottlenecks, and delays in applications (Faticanti et al.; 2021), researchers may monitor inter-node interactions, resource bottlenecks, and application delays by combining Grafana with Kubernetes data.

The recent study has described how Prometheus and Grafana can be used in predictive resilience modeling besides monitoring. To enhance proactive defense of the system, (Ganesan; 2022) described how Prometheus data can be used as input into anomaly detection algorithms to anticipate the future occurrence of failure patterns.

3 Methodology

3.1 Selected Methodology

A number of methodologies were evaluated and and then an experimental design of this study was decided. One of the options was to test the resilience with the help of synthetic Kubernetes cluster where test scenarios were pre-established. These simulations however are not applicable in practice because they cannot recreate the dynamic and reliable nature of real world cloud environments.

The methodology suggested in this paper is a blend of IaC, controlled fault injection, and observability-driven analysis to make sure it is accurate and reproducible. The experimental setup provisions an Amazon EKS cluster that has three worker nodes distributed across two subnets to provide fault isolation. It is a live environment, which approximates production-grade infrastructure and can be tested in real time regarding system resilience under more realistic workloads. Chaos testing is done using LitmusChaos, a CNCF-certified framework based on fault injection in Kubernetes. Among the solutions that have been considered, such as Chaos Mesh, Germilin, and Chaos Monkey, LitmusChaos was selected due to its declarative Kubernetes integration and open-source. The fact that it can model faults at both pod and node-level is a perfect match to the objective of the research.

Also, Prometheus was added to the cluster to log performance data in real-time and show metrics such as Mean Time to Recovery, CPU utilization, and Latency. These observability methods offer a model of determining the impact of chaos experiments. The design of the experiment is as follows:- (1) Terraform infrastructure provisioning, (2)

Containerized sample workload deployment, (3) conducting Chaos experiments, and (4) collection of the metrics of assessment by Prometheus.

3.2 Technologies Used, Tools, and System Configurations

The research environment has been configured to be more similar to a real-world, production-grade cloud-native system. The important configurations and technologies utilized are the following:

- Terraform: This is used to do automated provisioning of the infrastructure. The configuration outlines an Amazon Elastic Kubernetes (EKS) cluster configuration of Nodegroups, and a Virtual Private Cloud (VPC) with two subnets spread across the availability zones. The use of infrastructure-as-code guarantees consistency in environment setup and reproducibility during tests. In accordance with best practices with version control and reusability, the Terraform files are arranged into modular directories for network, EKS, and Nodegroup.

- Kubernetes (AWS EKS with 3 Worker nodes): Amazon EKS offers and manages the Kubernetes control plane, and worker nodes in a managed nodegroup of AWS EC2 operate the Kubernetes pods. This environment consists of three worker nodes that comprise the compute layer of the cluster giving full control over instance types, capacity, scaling and, networking. Each pod operates in an isolated, configurable compute environment, by executing in a dedicated nodegroup, which lowers the operational overhead and enhances security by regulating node-level policies. In order to run LitmusChaos, monitoring, and test workloads, the EKS cluster uses Role-Based Access Control (RBAC) as a measure of namespace isolation and secure access.

- LitmusChaos: LitmusChaos: LitmusChaos is a CNCF-incubated chaos engineering framework that is deployed on the EKS cluster via Helm charts. It also coordinates controlled fault injections within the nodegroup environment such as network latency, pod deletions and pod-cpu-hog experiments. In the case of actual failure, the tests determine the resilience of the system and the self-healing capacities of Kubernetes.

- Prometheus and Grafana: The experiments of chaos will require Prometheus to gather the time-series metrics, such as the CPU usage, memory usage, and the availability of the pods.Grafana is set up to use the dynamic dashboards to display this information. The observability stack makes it easier to track how the system behaves both during and after interruptions, allowing for in-depth evaluation of fault tolerance and recovery performance.

3.3 Metrics of Evaluation

The evaluation emphasizes on quantitative resilience and performance measures that objectively assess system behaviour under fault scenarios. The study comprises these metrics:

Parameter	Configuration
Cloud Platform	AWS
Containerization Orchestrator Software	Kubernetes (EKS)
Region	us-east-1
EKS Cluster Type	Nodegroup (3 nodes t3.medium)
Subnets	2 private subnets across different availability zones
Kubernetes Version	1.31
Observability Stack	Prometheus + Grafana
Failure Injection Tools	LitmusChaos CRDs and ChaosCenter

Table 3: Cloud Specification

- Mean Time to Recovery (MTTR): The average time it takes for the system to recover to a steady state following a failure (e.g., pod or node failure).
- System Uptime Percentage: Represents the percentage of time the system remains operational during chaos events.
- CPU Utilisation: Measures the amount of resources used before, during, and after chaos experiments in order to calculate overhead and efficiency.

Prometheus exporters and Kubernetes logs are used to gather these metrics, which are then displayed for interpretation using Grafana dashboards. In order to verify whether the architecture improves the overall resilience of cloud-native systems, the results are used to identify patterns of degradation and recovery.

4 Design Specification

4.1 Proposed Architecture

Figure 1 provides an overview of proposed architecture that integrates Chaos Engineering and IaC to test and enhance the resilience of cloud-native systems. The automated infrastructure provisioning is handled with Terraform, the orchestration is handled with Kubernetes, and fault injection is controlled with LitmusChaos, and the observability and resilience are assessed through Prometheus. All of this is deployed into AWS environment, and it guarantees a secure and reproducible architecture that is controlled. The foundation or the architecture is Terraform, which deployed and configured the required componenets of cloud infrastructure. This includes configuring the Virtual Private Cloud (VPC), subnets across various availability zones, internet and NAT gateways, security groups, and an EKS cluster. Since Terraform executions are state-managed and version-controlled, infrastructure change can be traced over experimentation.

After the EKS cluster has been provisioned, Kubernetes acts as orchestration layer which handles containerized workloads. The Sock Shop microservices application is placed in the cluster to recreate a distributed cloud-native system of production quality. At the experimental layer, LitmusChaos is integrated into Kubernetes via Helm. It conducts controlled failure experiments as pod-delete, node-drain, and CPU-Hog by serving as a native Chaos engineering framework in Kubernetes. The framework collects measurements of Mean Time to Recovery, uptime and resource utilization and checks the system behaviour on failures. Prometheus is employed in the collection of metrics and scraping

information on Kubernetes pods and system nodes and application endpoints to facilitate real-time analysis and post-experiment analysis. The observability stack facilitates qualitative resilience assessment by providing insights based on data about the way the infrastructure responds to created issues.

Inn general, this planned architecture forms a framework of end-to-end resilience checks that mechanizes the simulation and checking of failures. It demonstrates how an active identification of vulnerabilities, shorter recovery times, as well as enhanced reliability of cloud-native systems can be established by integrating Chaos Engineering into deployments of Kubernetes.

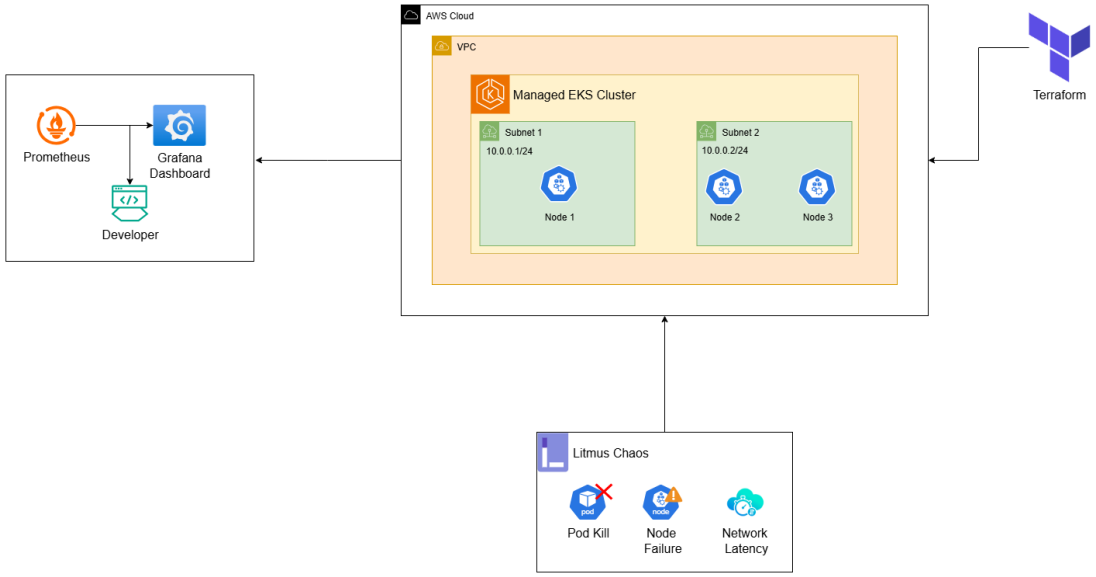

Figure 1: Architecture Diagram

5 Implementation

5.1 Infrastructure Provisioning with Terraform

The first part of the experiment setup includes the development of a fully cloud-native environment, using Terraform. The setup developed the minimum infrastructure to initiate the Chaos Engineering system, the sock-shop app, and Kubernetes cluster. Terraform enhanced stability of the infrastructure by codifying every infrastructure part, which ensures a constant and versioned deployment in different environments. The necessary networking, compute, and orchestration resources on AWS were defined and deployed through the Terraform configuration. In order to enhance the fault tolerance and high availability, multiple private subnets had to be developed in the various availability zones within this VPC. A routing table and network access control lists (ACLs) were configured to lock internal traffic so that there was an isolation between communication between the application and Prometheus, and chaos testing services. from external networks.

The security groups were formed to enhance the connectivity and scalability of clusters with strict rules allowing inbound and outbound traffic. Only the ports that were required were open: Prometheus scraping of metrics (port 9090) and secure Kubernetes API (port

433). This configuration was able to accommodate the operational need og the experiments and observe a solid security posture through the least privilege principle. The AWS EKS was established using the scripts of the Terraform modules, which define certain resources such as the cluster control plane and Nodegroup. The infrastructure was established using the Terraform command in AWS environment by defining the configuration. Terraform managed to create all resources in sequence and autonomously by using the AWS API through the AWS CLI. Terraform provided a baseline environment on which the experiments were run and for which systematic assessment of resilience metrics were possible.

5.2 Kubernetes Cluster Setup

The Kubernetes cluster is the most popular orchestration environment of cloud-native infrastructure deployment and management. To construct a managed Kubernetes cluster in this research, Amazon Elastic Kubernetes Service (EKS) was utilized to ensure high availability, scaling, and secure connection with other AWS resources. Terraform was used to provision the cluster. The deployment was carried out in `us-east-1` region, which was selected because it supportes multiple availablity zones. To balance between computational efficiency and cost optimization, the EKS cluster was set up with three worker nodes each having two vCPUs and 4GB of RAM. To achieve fault tolerance and reduce the effects of potential zone-level outages, these nodes were distributed in two subnets, which are in different availability zones. To set up the infrastructure that meets the requirements of the existing stability and ensures compatibility with the LitmusChaos and Prometheus integrations, the Kubernetes version 1.31 was used.

The AWS VPC CNI plugin was used to manage the networking in the cluster and allow pods to communicate with the help of IP addresses provided by the VPC. CoreDNS was used to resolve internal names and kube-proxy was used to provide load balancing and routing between the services. `kubectl` was configured to log in to the cluster after its deployment which provided access to the cluster in a secure manner. The Sock Shop 5 microservices application has been deployed to emulate the production-grade workload consisting of a number of interrelated services, such as front-end, carts, catalogue, and orders. To test and monitor, the microservice was exposed internally via ClusterIP and externally via LoadBalancer services once it was deployed with YAML manifests. Commands like `kubectl get nodes` and `kubectl get pods -n sock-shop` were used to confirm that the deployment was successful and all resources were operational.

```
$ kubectl get pods -n default
NAME                              READY   STATUS    RESTARTS   AGE
carts-db-594b68dc85-bs7tc         1/1     Running   0          3m52s
carts-f4f6c98b9-8v5gv             1/1     Running   0          3m54s
catalogue-6d8b8988d-rwggr         1/1     Running   0          3m51s
catalogue-db-5c796cfb68-2tcp8     1/1     Running   0          3m50s
front-end-55c969566c-qs28z        1/1     Running   0          3m49s
orders-685d756fb5-mdwz9           1/1     Running   0          3m48s
orders-db-67c8d5f459-dm62q        1/1     Running   0          3m47s
payment-56bcfc857f-dnn24          1/1     Running   0          3m46s
queue-master-5f4477db4c-4brkw     1/1     Running   0          3m45s
rabbitmq-97cbcbdbc-69hhd          2/2     Running   0          3m44s
session-db-67c8bfcb9c-hpftd       1/1     Running   0          3m43s
shipping-5c7bc7f48f-jn99j         1/1     Running   0          3m41s
user-6776ddd957-rswst             1/1     Running   0          3m40s
user-db-7dbb5fcd6f-vdchq          1/1     Running   0          3m39s
```

Figure 2: Sock-Shop Application Deployment

5.3 LitmusChaos Installation

LitmusChaos was installed using Helm, which offers a streamlined approach to manage Kubernetes apps. In order to maintain clarity and isolation within the cluster, a separate namespace `litmus` was created for LitmusChaos resources. This separation improves governance by preventing conflicts with other workloads and allowing greater control over the Chaos platform's components. During the installation, the necessary litmus services such as ChaosCenter, MongoDB backend, and other control plane utilities were deployed using Helm charts, which contain all the configuration values, container images, and dependency requirements. Helm makes lifecycle management easier by packing these parts into reusable templates, which ensure consistency between deployments and allow for easy customisation for various environments. The installation process guarantees that the ChaosCenter, which acts as the central administration interface for organizing and managing chaotic experiments, is properly setup with is supporting components. MongoDB which serves as the persistent layer for experimental metadata, workflows, and analytics is immediately setup as part of deployment. This enables the tool to retain past experiment data while also providing useful reports on its dashboard. Litmus provides a web-based user interface that enables operations to interact with ChaosCenter directly from the local environment. This interface allows user to plan, schedule, and execute chaos processes.

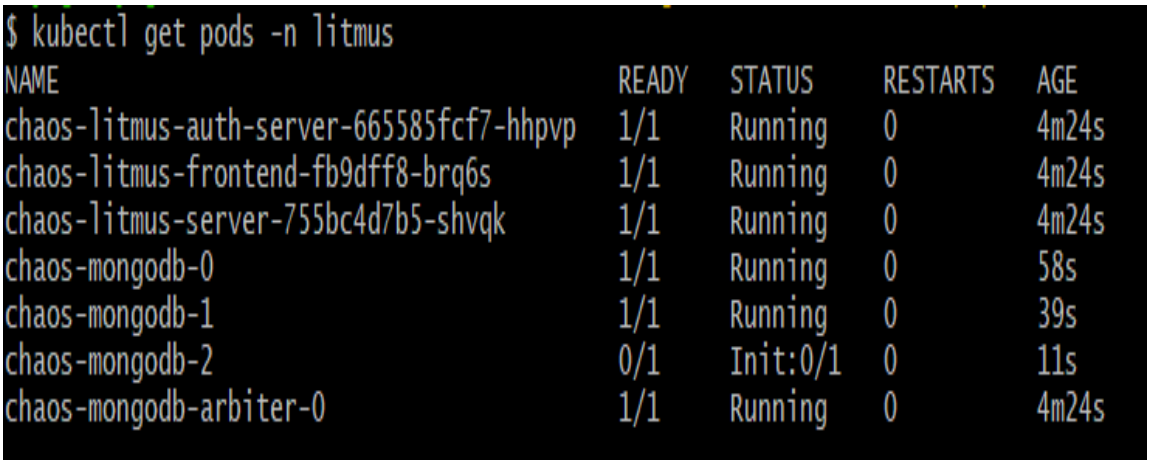

Figure 3: LitmusChaos Setup

5.4 Monitoring and Alerting Configuration

Prometheus was installed through the kube-prometheus-stack Helm chart that comes with Prometheus, Alertmanager, Grafana, and other tools to monitor. The entire monitoring stack was put in a separate namespace in order to make it easier to maintain and offer clean separations of concerns. This kind of isolation of Prometheus allows its resources like Stateful-Sets, Services, ConfigMaps and persistentStorage to be logically grouped and decoupled in order to simplify the process of scalability, debugging and lifecycle management of application workloads. The persistent storage of Prometheus was enabled, which ensured that the data on monitoring, time-series metrics are preserved during pod restarts or cluster events. After being launched, Prometheus stack promptly identifies Kubernetes components through service discovery features and starts to gather node, pod, workloads, and cluster system metrics. These metrics are maintained in the form of

time series, and their analysis can be done to view the performance of the application, cluster behaviour, and resource utilisation in extensive ways with time.

```
$ kubectl get pods -n monitoring
NAME                                                     READY   STATUS    RESTARTS   AGE
alertmanager-monitoring-kube-prometheus-alertmanager-0   2/2     Running   0          57s
monitoring-grafana-6587d4bd48-n2hvb                      3/3     Running   0          61s
monitoring-kube-prometheus-operator-9888744d8-zps8m      1/1     Running   0          61s
monitoring-kube-state-metrics-689d998768-7ghx8           1/1     Running   0          61s
monitoring-prometheus-node-exporter-lcdcl                1/1     Running   0          61s
monitoring-prometheus-node-exporter-m6nzt                1/1     Running   0          61s
monitoring-prometheus-node-exporter-vgkmp                1/1     Running   0          61s
prometheus-monitoring-kube-prometheus-prometheus-0       2/2     Running   0          57s
```

Figure 4: Prometheus & Grafana Setup

5.5 GitHub

GitHub is an open-source repository used to store source code, where we may make modifications, push them to Git and merge them with earlier files. Additionally we can pull the source code and work on it whenever needed on our local machine. We have stored the Terraform scripts and other files on it [2].

6 Evaluation

In this section, we will perform multiple experiments to determine system robustness and failure in the cluster. The architecture involves using an EKS to run a three-node Kubernetes cluster and deploying microservices over it. To ensure accurate findings, the experiment will use the same settings and configurations throughout.

6.1 Network Loss

The experiment simulated network packet loss in the carts microservice pod to determine how poor network connectivity affects request handling and data flow. As depicted in the graph, incoming network traffic dropped drastically during the fault injection phase, followed by unpredictable spikes as the service tried to maintain consistent connectivity. Once the injection duration ended, network traffic gradually returned to baseline pattern, demonstrating that the service had successfully recovered from the caused disruption.

[2]https://github.com/SuyogRedkar/x23419504-RIC.git

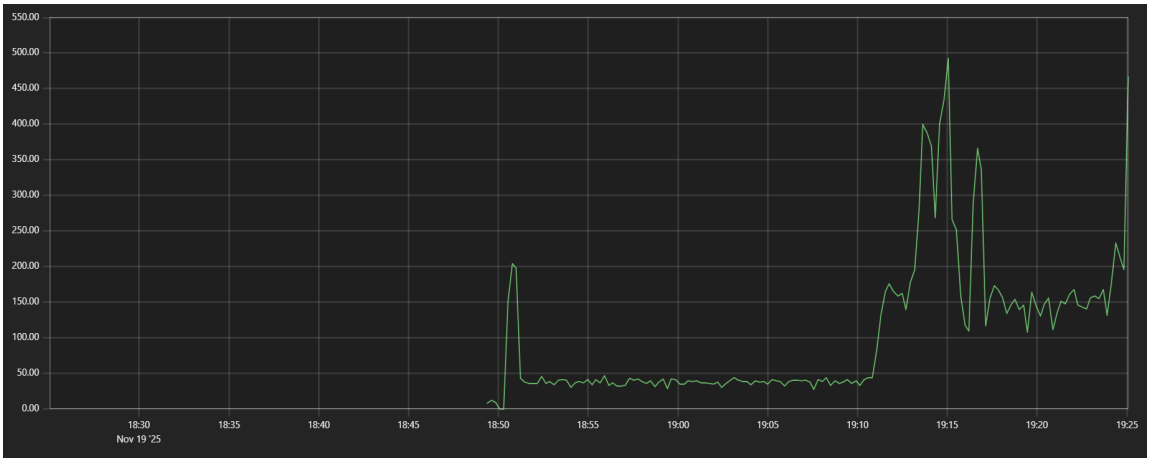

Figure 5: Network Loss

6.2 Pod Termination Under Non-Replicated Workload

The pod deletion chaos tests on a non-replicated charts service reduced the pod count to zero, resulting in total service outage for the disruption period. Because no extra replicas were specified, Kubernetes needed time to recover the terminated pod before making the application available again. This illustrates that single-pod workloads are particularly prone to failure and cause significant recovery delays.

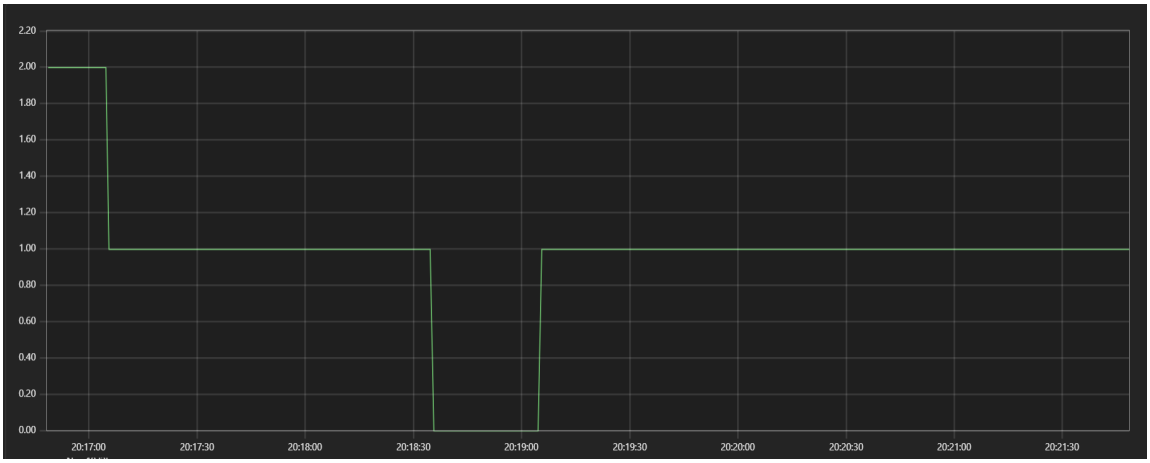

Figure 6: Service Availability without Replica Set

6.3 Pod Termination Under Replicated Workload

In the pod deletion chaos test with ReplicaSet configuration, the carts pods was intentionally terminated, but another replica remained running throughout the experiment. This ensured that the cart service continued to respond without experiencing ant visible downtime. Kubernetes instantly launched a replacement pod to keep the appropriate repica count, allowing the system to recover smoothly.

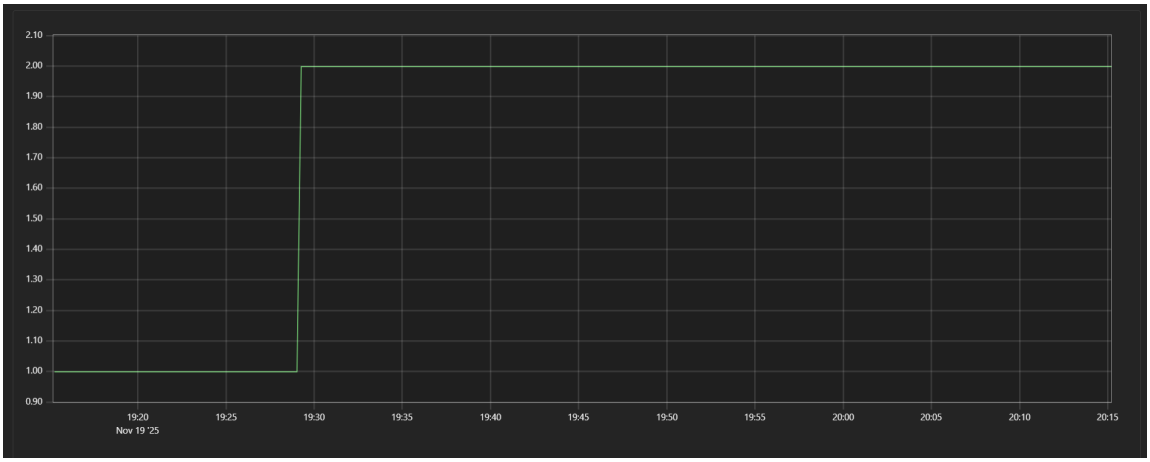

Figure 7: Service Availability with Replica Set

6.4 CPU stress Without HPA

During the CPU stress test, just one instance of the carts service was running without Horizontal Pod Autoscaler support. AS pressure increased, CPU utilization surged dramatically and stayed continuously high, indicating that the workload was struggling to manage incoming requests. Because no additional copies were automatically provisioned, the service performed poorly and had limited processing capacity when subjected to persistent CPU stress.

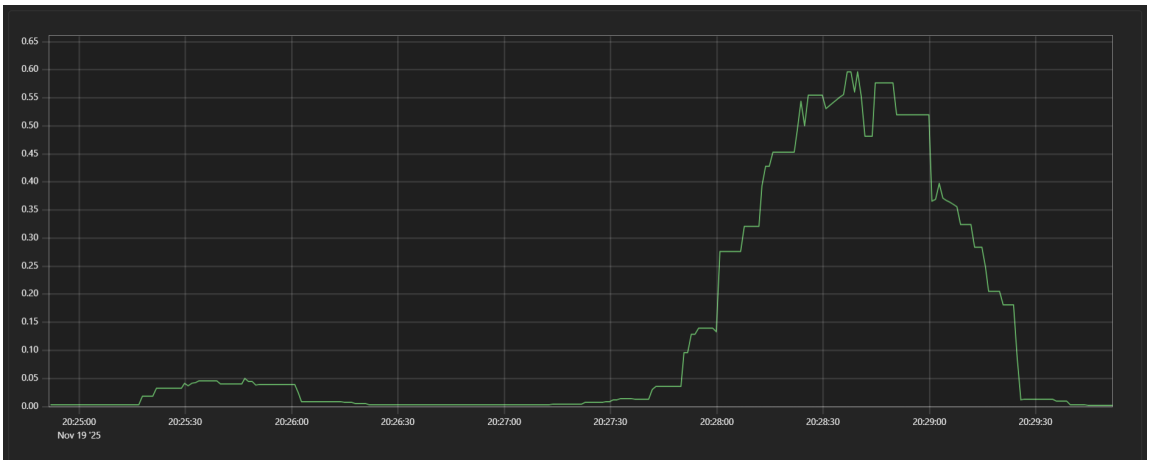

Figure 8: CPU Utilization without HPA Enabled

6.5 CPU stress With HPA

When the CPU stress experiment was run with Horizontal Pod Autoscaler enabled, the system automatically added more replicas as CPU pressure increased. The workload was dispersed over numerous pods, preventing a rapid increase in usage for any single instance. As a result, the service maintained consistent performance and avoided resource saturation during the stress period.

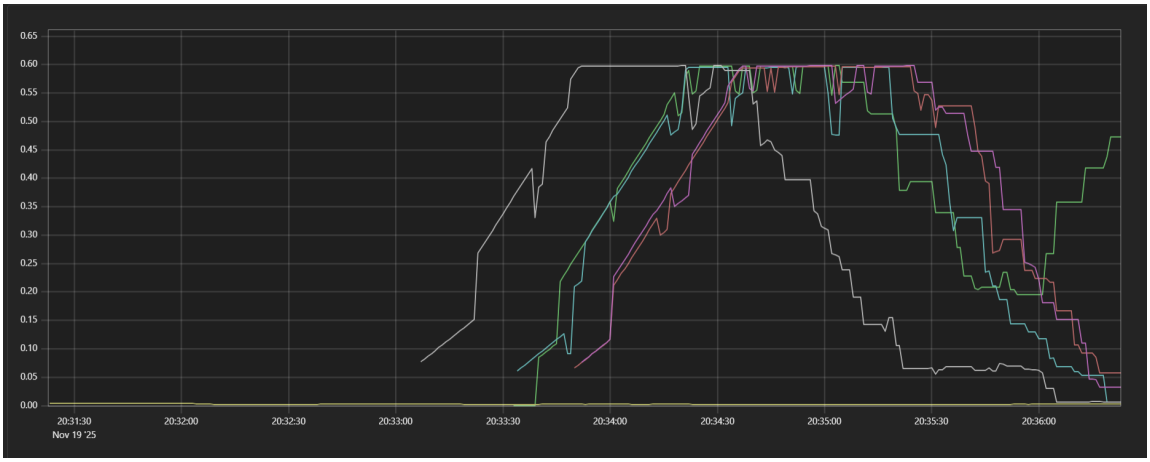

Figure 9: CPU Utilization with HPA Enabled

6.6 Result and Discussion

Chaos test provide insight into the issue of the great role of redundancy, autoscaling and network reliability on service availability and performance. The pod delete test with no ReplicaSet led to the full service failure, but replicated deployment was able to provide continuous service and showed importance of redundancy. Similarly, without HPA, The stress on the CPU created more usage and reduced stability, whilst HPA can support effective efficient resource management and performance. The network loss test indicated that there was a temporary worsening of the network, but it recovered successfully without involving a user. All in all, the presented results demonstrate that self-healing and scaling tools are essential in the creation of robust and highly available cloud-native applications.

Experiment	**Metric**	**Result**
Network Loss	Latency	27 ms
Pod Delete Without ReplicaSet	MTTR	45 seconds
	Availability	85%
Pod Delete With ReplicaSet	MTTR	0 seconds
	Availability	100%
CPU Stress Without HPA	Max CPU Utilization	65%
CPU Stress With HPA Enabled	Max CPU Utilization	30%

Table 4: Experiment Results

7 Conclusion and Future Work

Kubernetes configurations that are resilience-oriented are critical in maintaining the reliability of the service as illustrated by the discussion of the application behaviour with the failure of pods, network interruptions, and saturation of CPUs. Workloads deployed without autoscaling or redundancy showed complete outages and performance degradation, indicating their susceptibility to real-world failure. ReplicaSets and Horizontal

Pod Autoscaling, in their turn, significantly enhanced the stability of the services and enabled the recovery and service continuation even during the stress. The present study can be furthered by future studies by incorporating more elaborate failure conditions, including node-level failures or storage failures, to learn more about the limits of resilience. Moreover, implementing automated chaos testing in CI/CD pipeline would allow reliability testing in production-like conditions whenever necessary.

Further development of this research by integrating Service Mesh or other such methods as KEDA (Kubernetes Event Driven Autoscaling) could provide further information on the optimization of performance. On balance, it is possible to state that the results of the given study testify to the effectiveness of chaos engineering as a means of proactive enhancement of cloud-native applications in combination with Infrastructure-as-Code and continuous monitoring.

References

Al-Said Ahmad, A., Al-Qora'n, L. F. and Zayed, A. (2024). Exploring the impact of chaos engineering with various user loads on cloud native applications: an exploratory empirical study, *Computing* **106**(7): 2389–2425.
URL: *https://doi.org/10.1007/s00607-024-01292-z*

Barletta, M., Cinque, M., Di Martino, C., Kalbarczyk, Z. T. and Iyer, R. K. (2024). Mutiny! how does kubernetes fail, and what can we do about it?, *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, p. 1–14.
URL: *http://dx.doi.org/10.1109/DSN58291.2024.00016*

Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J. and Rosenthal, C. (2016). Chaos engineering, *IEEE Software* **33**(3).
URL: *http://dx.doi.org/10.1109/MS.2016.60*

Blohowiak, A., Basiri, A., Hochstein, L. and Rosenthal, C. (2016). A platform for automating chaos experiments, *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE.
URL: *http://dx.doi.org/10.1109/ISSREW.2016.52*

Burns, B. (2024). *Designing distributed systems*, " O'Reilly Media, Inc.".

Chinnasamy, P. (2025). Multi-cloud chaos engineering with azure devops: Automating reliability across kubernetes with ai possibilities, *International Research Journal of Modernization in Engineering Technology and Science* **07**: February–2025.

Dame, M. (2022). *The Kubernetes Operator Framework Book: Overcome complex Kubernetes cluster management challenges with automation toolkits*, Packt Publishing Ltd.

Faticanti, F., Santoro, D., Cretti, S. and Siracusa, D. (2021). An application of kubernetes cluster federation in fog computing, *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 89–91.

Ganesan, P. (2022). Observability in cloud-native environments challenges and solutions, **7**: 52–63.

Hasan, M. R. and Ansary, M. S. (2023). Cloud infrastructure automation through iac (infrastructure as code), *Int. J. Comput.(IJC)* **46**(1): 34–40.

Hausenblas, M. (2023). *Cloud Observability in Action*, Manning Publications, Shelter Island, NY.

Hausenblas, M. (2024). *Cloud Observability in Action*, Simon and Schuster.

Larsson, L., Tärneberg, W., Klein, C., Elmroth, E. and Kihl, M. (2020). Impact of etcd deployment on kubernetes, istio, and application performance.
URL: *https://arxiv.org/abs/2004.00372*

Malik, S., Naqvi, M. A. and Moonen, L. (2023). Chess: A framework for evaluation of self-adaptive systems based on chaos engineering.
URL: *https://arxiv.org/abs/2303.07283*

Murphy, O. (2022). Adoption of infrastructure as code (iac) in real world; lessons and practices from industry.

MUSTYALA, A. (2020). Infrastructure as code (iac): Achieving scalability and agility in it operations, *EPH-International Journal of Science And Engineering* **6**(1): 61–64.

Rosenthal, C. and Jones, N. (2020). *Chaos Engineering: System Resiliency in Practice*, O'Reilly Media.
URL: *https://books.google.ie/books?id=jVjbDwAAQBAJ*

Saxena, A., Singh, S., Prakash, S., Yang, T. and Rathore, R. S. (2024). Devops automation pipeline deployment with iac (infrastructure as code), *2024 IEEE Silchar Subsection Conference (SILCON 2024)*, IEEE, p. 1–6.
URL: *http://dx.doi.org/10.1109/SILCON63976.2024.10910699*

Simonsson, J., Zhang, L., Morin, B., Baudry, B. and Monperrus, M. (2021). Observability and chaos engineering on system calls for containerized applications in docker, *Future Generation Computer Systems* **122**: 117–129.
URL: *http://dx.doi.org/10.1016/j.future.2021.04.001*

Torkura, K. A., Sukmana, M. I. H., Cheng, F. and Meinel, C. (2020). Cloudstrike: Chaos engineering for security and resiliency in cloud infrastructure, *IEEE Access* **8**: 123044–123060.

Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization*, James Turnbull.

Yevick, T. and Jones, C. (2020). *Infrastructure as Code with Terraform*, O'Reilly Media.