

# Enhancing Microservices Resilience: Chaos Engineering with Istio Service Mesh on Kubernetes

MSc Research Project  
MSc Cloud Computing

Shubham Singh  
Student ID: 22170341

School of Computing  
National College of Ireland

Supervisor: Shaguna Gupta

National College of Ireland  
Project Submission Sheet  
School of Computing



<b>Student Name:</b>	Shubham Singh
<b>Student ID:</b>	22170341
<b>Programme:</b>	Cloud Computing
<b>Year:</b>	2023/2024
<b>Module:</b>	MSc Research Project
<b>Supervisor:</b>	Shaguna Gupta
<b>Submission Due Date:</b>	14/12/2023
<b>Project Title:</b>	Enhancing Microservices Resilience: Chaos Engineering with Istio Service Mesh on Kubernetes
<b>Word Count:</b>	7463
<b>Page Count:</b>	22

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.

<b>Signature:</b>	Shubham Singh
<b>Date:</b>	31st January 2024

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:**

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

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 Microservices Resilience: Chaos Engineering with Istio Service Mesh on Kubernetes

Shubham Singh  
22170341

## Abstract

Microservices cloud-native architectures are gaining popularity for deploying applications, employing small modules, and each of these modules handles specific tasks. They have various advantages, such as scalability, improved maintainability, and elasticity. However, the architecture's complexity rises with multiple separate modules, posing challenges. Microservices running on Kubernetes platforms often face failures that degrade application performance and availability. This research report explores bolstering microservice resilience within cloud-native setups through chaos engineering using Istio Service Mesh on Kubernetes. Evaluating Istio's impact on microservices performance, the research reveals substantial resilience improvements and performance enhancements. Istio performs consistently under severe loads and for extended periods of time, highlighting its important role in bolstering microservices against failures. The methodology emphasises the configuration of Istio features such as circuit breaking, retries, and traffic control. These ensure the resilience of microservices and infrastructure. The results illustrate the efficacy of Istio in identifying disruptions, limiting their scope, and restoring application performance with minimal impact on users. We have collected metrics such as response time, error rate, and total requests per second to assess the impact of Istio on the recovery and resilience of microservices. The experiment results proved that the proposed design performs significantly better as compared to the traditional architecture by offering stability, faster response time, and persistence in failures.

Keywords: GKE, Kubernetes, Chaos Engineering, Resilience, Istio service mesh

## 1 Introduction

Cloud Computing has become the widely accepted platform for deploying web applications. Its on-demand infrastructure, platform, and software availability support microservices. The ability to scale resources dynamically to match the demand allows microservices to run independently.

Many companies are migrating to a microservices architecture as they are good for scalability, fault isolation, and continuous integration and deployment Balalaie et al. (2018). Additional to this they also provide extensibility and maintainability. From Figure 1 we can see that microservices is an application build as a collection of small services, each running in its own process and deployed independently whereas the monolithic application, including its user interface, business logic, and data access layers, is built and deployed as a single unit. Microservices are mostly deployed using containers to avoid problems with deployment and management of it. Kubernetes is one of the

many available container orchestration platforms in the industry. It helps us to achieve automated deployment, and horizontal scaling and provides built-in security features Bernstein (2014). The complexity of microservices ecosystems also poses challenges. The expansion of services and containers across dynamic infrastructure increases the risk of pod failures Monteiro et al. (2017).

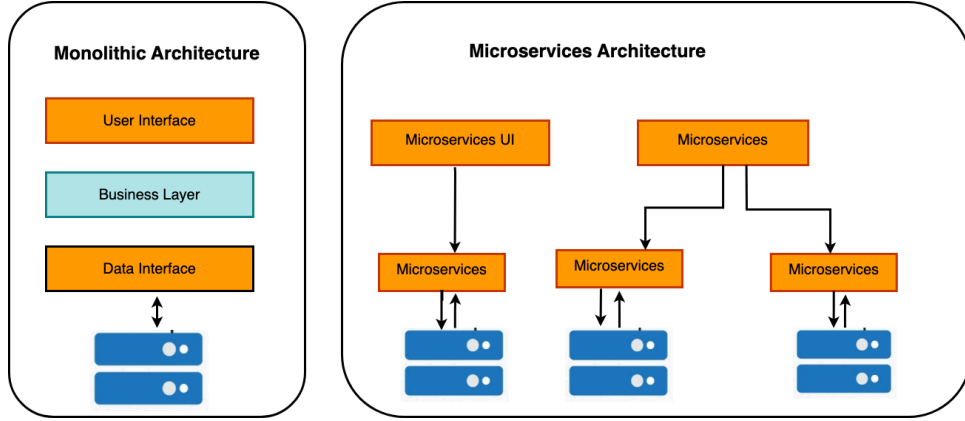


Figure 1: Monolithic Architecture & Microservices

In today’s world, a single failure has the power to disrupt the whole platform and the applications running on it. It’s important to observe and keep the application functional. Making copies of pods and checking on their health daily is an important cornerstone. This makes sure that if one pod fails, there is always another one ready to take its place. It’s like having a backup pod ready to deploy in case one pod fails because the application should be up and running in all cases. Service meshes have become a crucial element in the microservices framework. Istio, a widely used open-source service mesh, plays a key role in overseeing traffic patterns, implementing policies, and consolidating telemetry data among microservices. It serve as a facilitator to make sure that services can talk to each other easily. It has features like circuit breaking and a retry mechanism to make sure there’s a backup in case some services go down.

But it’s not always about defence, sometimes we need to fake chaos to get ready for what might happen. This is where chaos engineering comes in. It’s like stress-testing our system by making it fail on purpose to see how it responds. This practice helps us to make our application ready for failure and makes our plan stronger in case something goes wrong. Giving the system the ability to automatically scale and use smart techniques like retries at the application level and the exponential backoff method also makes it much more resilient.

To ensure success, it is crucial to design and set up the application environment while considering the possibility of failure. The platform should be created in a manner that can endure and bounce back from faults, all while ensuring that the end user remains unaware of any disruptions. By incorporating strategies to anticipate and manage disruptions, the overall system gains increased reliability and resilience. Instead of simply responding to problems, this approach guarantees that the system is strong enough to sustain uninterrupted service and swiftly recover from challenges. Microservices like API Gateway, Backend, Database, Messaging, Monitoring, and Authentication Microservices can utilize the proposed architecture. It depends on the business and technical requirements of the application to determine which microservices to use. Basically mostly all

types of microservices can be used in our proposed application. A combination of different types of microservices creates a scalable and resilient architecture.

## 1.1 Motivation

Applicant deployment in cloud-native environments is now recognised as an essential component of modern computing in the current era of digital transformation. By virtue of their exceptional adaptability, scalability, and resource management, cloud technologies facilitate the implementation of microservices architectures. Nevertheless, the adoption of microservices as a paradigm change presents challenges in effectively handling failures, which can have a significant detrimental effect on the dependability and efficiency of applications. One of the primary issues in modern computing regards to the vulnerability of microservices architectures to single-point malfunctions. A single error occurring in either a service or a module has a possibility for developing into a disruption that spans the entire application ecosystem. Research into robust solutions is driven by the urgent requirement to guarantee continuous functionality and resilience when confronted with such failures.

The motivation behind this research is to strengthen the resilience of microservices in frameworks that are associated to the cloud. The objective of this study is to investigate the management of the Istio Service Mesh on Kubernetes. To achieve this, chaos engineering will be utilised to simulate failures and assess the effectiveness of Istio in improving system stability. Through an examination of the effects of Istio on the performance, responsiveness, and dependability of microservices in high-pressure situations, this research attempts to identify approaches that eliminate interruptions and enhance the robustness of the system. The primary objective is to identify practical strategies that enable microservices to withstand failures, thereby reducing periods of inactivity and maximising the efficiency of applications. The findings obtained from this research have the potential to enhance the dependability and resilience of microservices architectures, making a substantial contribution to the constantly evolving domain of cloud-native computing.

## 1.2 Research Question

How do different configurations of Istio control plane components affect the latency, performance, and availability of microservices when evaluated under pod failures using chaos mesh and increasing service requests?

**Problem Statement:** In real-world Kubernetes deployment, multiple challenges emerge, such as network connectivity, robustness, and load handling. Network complexities include pod-to-pod communication, managing service discovery, and addressing network overhead. The traffic routing is also very difficult to achieve in Kubernetes. These issues are significant because they directly impact an organisation's ability to deliver reliable, secure, and high-performance applications in a microservices architecture. Failing to address these issues can result in less reliable applications with frequent connection issues, increased operational complexity, and negative impact on customer satisfaction. The solution to these problems is service mesh, which resolves these critical challenges in real-world Kubernetes deployments. It offers features such as retries, circuit breaking, and timeouts. Furthermore, it even handles and manages the load on the cluster perfectly.

### 1.3 Structure of the Paper

There are six sections in this research report. In the first section, we introduce service mesh and microservices and outline the motivation driving this study. In the second section, an extensive review of the literature on the existing research done in this field is discussed. The study’s instruments and methodology are described in Section 3. The architecture of the system and the design parameters of our experimental setup are covered in detail in the next section, section four. The proposed approach’s stepwise execution is methodically presented in Section 5. Ultimately, a comprehensive examination of the experiment data and implications is presented in parts six and seven.

## 2 Related Work

Understanding Kubernetes and microservices is required for researching and finding a solution to achieve optimal performance. This section will discuss existing research projects in our domain. It will explain what has already been done to address microservices issues and highlight the limitations of their work.

### 2.1 Monolith and Microservices

Monolith architecture is a development approach that was used by big companies like Amazon and Netflix. In this traditional approach, all the functionality of an application is packed into one application. Monolithic applications are easy to build and test. However, as technology changes, the complexity and size of applications also increase. Making it difficult to manage and scale the application. On the other hand, we have a microservices architecture that is well-suited for large and complex applications that need scalability, agility, and the ability to scale as required. Here, the application is divided into small services that work together to provide the overall functionality of the applicationLauretis (2019).

### 2.2 Analysing Microservices

MIPaRT is a platform to automate the testing of microservices for performance and reliabilityCamilli et al. (2022). The proposed platform offers easy integration into a DevOps cycle, where it plays a critical role in supporting continuous testing and monitoring. Firstly, the platform generates and executes performance-reliability ex-vivo testing sessions. It then collects vital monitoring data to keep a watchful eye on the system’s performance.

In his research, Pietrantuono et al. (2018) developed a new algorithm called Microservice Adaptive Reliability Testing-MART. The technique enables evaluation of the reliability of a microservices-based application even when resources for testing are limited. This assessment is conducted on-demand during the application’s operation, utilizing real-world data pertaining to microservice utilization and the results of both successful and unsuccessful requests. For the experimentation, a pet clinic microservice is used. They conclude that the algorithm’s overall accuracy is high, but the time taken to evaluate the results is high. Toffetti et al. (2015) proposed an architecture that provides self-management of microservices. It monitors the application and the properties related to the infrastructure of the application to provide on-time solutions to the failures. To test

the architecture of microservices, fault injection is being used by Meinke and Nycander (2015). For the setup, a distributed microservice architecture known as triCalculate is analyzed. The application was created by TriOptima AB for the OTC market. To assess the robustness of software fault injection, such as restarting services, starting several service instances, communication faults, and killing service instances are performed. The experimentation resulted in limited accuracy as they were constructed from a small sample size.

Alhamazani et al. (2014) research introduced cross-layer multi-cloud application monitoring as a service-CLAMS, an application monitoring framework developed for multi-cloud platforms. This framework collects QoS performance data for various cloud layers. However, it has a constraint: it can solely monitor the performance of virtual machines, and it's constructed exclusively for monitoring web applications. Another framework called PyMon is based on "monit" and is an application built using Django to collect and analyse the monitoring data collected. clusters. The setup is easy and collects a large amount of data. However, it might not possess the capability to oversee VM performanceGroßmann and Klug (2017). Camilli and Russo (2021) presents a new automated modelling approach for analysing microservice resilience. It models the performance violations over time using growth theory and stochastic processes. The model provides insights into recovery capabilities and degradation trends. However, a more varied study is needed to predict accuracy and to look for wider applicability across domains.

## 2.3 Kubernetes

Kubernetes can be termed as a container orchestration platform that helps deploy the containers to the Kubernetes cluster. Kubernetes is well known for some of its excellent features, including scaling, high availability, fault tolerance, and flexibility. Karn et al. (2022) Kubernetes helps in monitoring the health of services and the infrastructure. Also, it helps in efficiently allocating and managing computing resources, ensuring that the application runs smoothly. Another important feature of Kubernetes is self-healing. When the container fails, Kubernetes automatically relaces failed containers and reschedules them on healthy nodes, making sure that the application is up and running in case of network failureFaticanti et al. (2021).

Dame and Michael (2022) mentioned that a cluster is a group of computers or machines that work together to run containerised applications and manage the entire Kubernetes environment. A Kubernetes cluster typically consists of two main components: the master nodes and the worker nodes. The master node is responsible for controlling and managing the cluster state, orchestrating tasks, and serving as the central point for all operations within the cluster. The worker nodes are responsible for executing the workloads and providing the computational resources needed by the applicationsIslam et al. (2022). The API server is the path for admin tasks and user communication, while etcd, a distributed key-value store, preserves the configuration data. Kubernetes can be used in the deployment of microservices and other web applications.

## 2.4 Service Mesh Istio

A service mesh is a dedicated infrastructure layer for handling service-to-service communication within a microservice architecture. It also monitors and manages network traffic between microservices. Redirecting, granting, or limiting access to the traffic as needed

to protect the system. As per the discussion of Li et al. (2019), it has been noticed that service mesh helps to control the requests of the services, and in addition, it balances the load evenly across multiple instances of a service. This ensures proper resource utilisation and reliability. Istio service mesh is also very helpful to mitigate latency issues at the edge. Ganguli et al. (2021) installed Istio on the Kubernetes cluster at the edge to measure the ratio between latency and throughput. This is crucial in edge deployments because the entire premise of edge deployment revolves around minimising latency. As in canary deployment, we can route one portion of traffic to a new version of a service for testing before a full rollout Ponomarev and Y. (2019). It also offers rate limiting, which controls the number of requests that a service can accept within a specific time frame. It helps to prevent attacks such as denial-of-service attacks on the microservices. According to Calcote and Butcher (2019) Istio service mesh helps to address the challenges of a distributed microservices architecture faced by the developers. Istio employs an extended iteration of the Envoy proxy, a high-performance proxy created in C++, to manage the flow of both incoming and outgoing traffic for all services within the service mesh. Istio harnesses Envoy’s capabilities, including dynamic service discovery, load balancing, TLS decryption, support for HTTP/2 and gRPC Remote Procedure Call (gRPC) proxying, circuit-breaking mechanisms, health checks, phased deployment rollouts with traffic distribution based on percentages, controlled introduction of faults, and comprehensive metric collection. The Anjali et al. (2020) mentioned that the architecture of Istio helps to secure the connection and monitor the authentication of the services. Malki et al. (2019) have written in their paper that one of the prominent features of Istio is that it facilitates secure service-to-service communication within clusters while providing robust traffic control and automated load-balancing capabilities.

## 2.5 Chaos Engineering

Chaos engineering is a discipline where we proactively introduce chaos and failures into a system to find the vulnerabilities, weaknesses, and potential points of breakdown. The primary goal of chaos engineering is to improve system reliability and resilience by uncovering problems before they happen in the real world. As per Torkura et al. (2020) chaos engineering is injecting faults intentionally into the system to test its resilience. Monitoring and observability tools are used to collect data from the experiments, showing how the system performs in chaos. Research by Malik et al. (2023) suggested a chess framework, where they have used two microservices called the Yelb. Here they are experimenting with five fault injection scenarios focusing on the infrastructure as well as functional faults. Many of the big companies use chaos engineering, such as Netflix, LinkedIn, Facebook, and Google Rosenthal and Jones (2020). Utilising this application offers additional advantages, such as ensuring stakeholder and end-user contentment, providing room for the collaboration team’s progress, and elevating their confidence Simonsson et al. (2021).

## 2.6 Summary of Literature Review

Microservices architecture research thus far has explored a wide range of topics, from monitoring frameworks to performance testing. Learning-based testing, multi-cloud monitoring frameworks, and integrated performance testing approaches are just some of the methods that have been the subject of research. However, our planned study will go down a new path by deliberately injecting faults into Kubernetes clusters and then using



the Istio service mesh to fix them. Our work differentiates from the existing literature since we concentrate on chaos engineering tests and the effect Istio has on network latency, robustness, and resilience. By simulating real-world network disturbances using chaotic mesh, we want to comprehend how Istio affects the behaviour and performance of microservices applications in Kubernetes clusters.

Our study’s significance comes in its examination of how service meshes, and in particular Istio, reduce network latency and improve resilience in microservices systems. No prior research has delved deeply into how service meshes cope with deliberate defects and network disturbances. With the knowledge gained from our work, Istio and other service mesh tools may be optimised to better support microservice-based systems, making them more resilient and responsive in the face of network difficulties. In Table 1, different related works done previously are compared on the basis of their framework, scenario, and advantage. The different research summarised below focuses on monitoring microservices and proposing different frameworks. Whereas we will be proposing a different framework, which will be tuned according to the faults that we will be injecting into the system.

Author	Framework	Approach	Advantage	Limitation
Barakat (2017)	Monitoring and Analysis of Microservices Performance	Used Kieker framework to monitor and analyze microservices applications	Kieker provides monitoring and analysis capabilities for microservices	Did not discuss integration with other tools or dashboards
Seifermann (2017)	Application Performance Monitoring in Microservices-Based Systems	Conducted industrial case study with Instana, proposed monitoring concept with Grafana	Evaluated commercial tool Instana, provided monitoring concept with automation and dashboards	Limited by industrial environment, static thresholds led to false detections
Meinke and Nycander (2015)	Learning-based Testing (LBT) of Distributed Microservice Architectures	Used LBT to evaluate correctness and robustness of a microservices system	LBT enables automated blackbox testing and fault injection	Handling of distributed systems and high latency needs more research
Alhamazani et al. (2014)	CLAMS - Cross-Layer Multi-Cloud Application Monitoring Framework	Proposed a monitoring framework using manager-agent approach across cloud layers	Enables cross-layer and multicloud monitoring	Further evaluation on complex systems needed
Großmann and Klug (2017)	Monitoring Container Services at the Network Edge	Developed PyMon framework to monitor containerized microservices on edge devices	Low overhead monitoring tailored for resource-constrained edge devices	Future work on automated placement based on monitoring data

Noor et al. (2019)	Multi-microservices Multi-virtualization Multi-cloud (M3) monitoring framework	Decentralized agents-based approach to monitor per- formance of microservices across platforms and multi-cloud environments	Cloud agnostic can monitor VMs and con- tainers; flexible centralized or decentralized architecture	Requires in- stalling moni- toring agents on each VM over- head caused by communication between agents and manager
<b>This Research</b>	<b>Istio-Chaos Engineering</b>	<b>Fault Injec- tion</b>	<b>Improved resiliency, robustness, traffic man- agement and security</b>	<b>Comprehensive Setup &amp; cost</b>

Table 1: Summarising Previous Research Work

### 3 Methodology

This research contributes to the ultimate performance of micro-services with service mesh Istio in Kubernetes clusters. The proposed idea will solve the concerns about resiliency, robustness, and reliability. The suggested solution not only outperforms the traditional microservices model installed on Kubernetes but also results in improved performance and higher application uptime. To sum it up, this suggested approach addresses typical challenges encountered within microservices and offers corresponding solutions to mitigate these issues effectively.

#### 3.1 Selected Methodology & It’s Justification

Several methodologies were considered and evaluated before arriving at the experimental approach used for our experiments. One alternative approach was an analytical model-based approach to predict the performance of microservices under different Istio settings. However, the complexity of microservice communication limits the accuracy of analytical models. Another approach we considered was a controlled simulation where fixed workloads and fault injections are executed in a fixed Kubernetes environment. But the instability and dynamism that come with real-world deployments are not present in simulations. The goal is to strengthen resilience against real-life unpredictable disruptions.

Furthermore, we evaluated the considerations of choosing different tools for the experiments. Among the approaches considered, one was to choose different service meshes for the experiments, such as Linkerd or Consul. However, Istio was selected because of its extensive integration within the Kubernetes ecosystem and its robust features. Also, its world-wide popularity and acceptance as a service mesh were considered. Various chaos engineering techniques and tools exist, like chaos monkey or chaos mesh, but to decrease the complexity and cost, we have chosen to write a bash script to execute the chaos engineering. We have also considered different load testing tools, such as JMeter,

but the locust’s flexibility, easy-to-use functions, and suitability for simulating various user behaviours are helpful for our experiments.

The chosen approach was executed meticulously and rigorously to ensure it aligned with our research objective. The selected approach also provides the right balance of realism and, at the same time, allows us to control the different settings by deploying live microservices in a real Kubernetes cluster while still allowing us to inject the failures. By doing so, we are able to observe the actual system behaviour under chaotic conditions, going beyond just theoretical models. The setup of the test environment is also done rigorously for reproducibility across different cloud platforms. Multiple iterations of control with variable parameters like user loads, failure rates, and duration are needed to cover an adequate range. Test runs are also executed for longer durations to find the cause of failure when the application is exposed to extended load and time.

### 3.2 Research Flow

In this section we will explaining the research flow as shown in figure 2. To conduct the setup, we have set up a multi-node Kubernetes cluster on the Google Kubernetes Engine. The application for the experiment is called test-app-hello and test-app-employee, which were created for this experiments using FastAPI, a Python-based high-performance web framework, and the application is tailored as per the needs of our experiments. The test-app-hello application has multiple endpoints. The hello endpoint is to check if our application is working fine. The fallible endpoint utilises the path parameters “pass\_weight” and “fail\_weight” to potentially influence the response behaviour based on specified weights. Employee\_id endpoint relies on a path parameter, “employee\_id”, facilitating the retrieval of specific employee details. Httpbin/headers endpoint accesses a URL and displays the response.

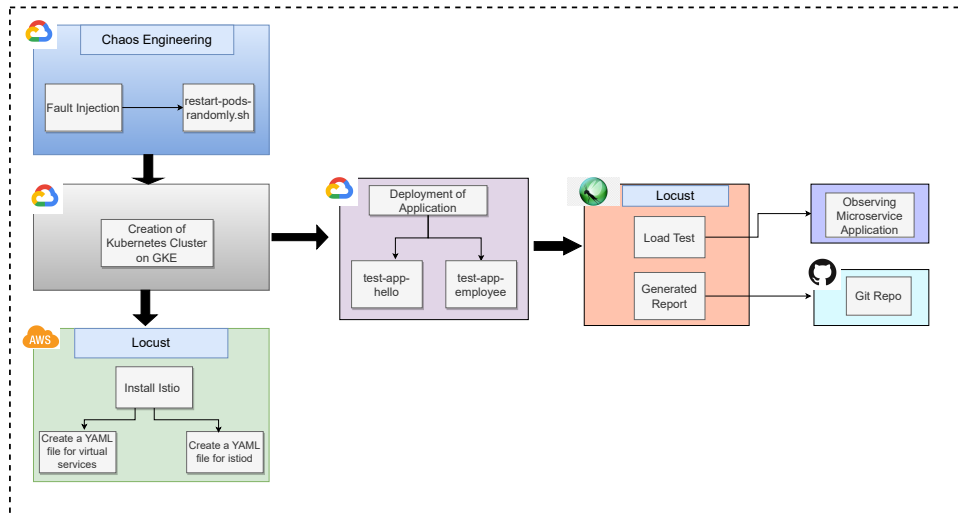


Figure 2: Flow Diagram

This application is deployed on the Kubernetes cluster using the command line (`kubectl apply -f test-app-hello.yaml`). After deployment of the application, we expose it to external traffic by making it of load balancer type with an external IP. Now we will install Istio in the newly created istio-system namespace and create two services named istio-ingress gateway and istiod. Lastly, we will install Locust on the AWS instance. On this

setup, we will be running the experiments, we will inject faults, hit the application with multiple user requests, and observe application behavior. After this, we will install Istio, run the same set of experiments, and observe the application’s performance. Afterward, we will fine-tune Istio to ensure the application delivers optimal performance and rerun the experiments.

### 3.3 Metrics for Evaluation

After running the experiment for all the combinations, we will be analysing the multiple values:

- Total request per second: This indicates the number of requests our system is handling within a second during the load test.
- Response time (ms): This indicates how long our system takes to respond to each request in milliseconds.
- Failure rate: This represents the number of requests that occurred during the load test. Failures could be due to various reasons, like timeouts, errors, or the system being overloaded.

Based on these, we will be evaluating and judging the performance of the proposed architecture.

## 4 Design Specification

Our implementation leverages cloud computing for its scalability, agility, and extensibility. The cloud infrastructure provides us with the flexibility to scale resources dynamically, respond to changing demands, and easily extend our system’s capabilities as needed.

The table 2 below shows the hardware specifications for the GKE & Kubernetes cluster.

### 4.1 Cloud Specification

Design Specification	
Cloud Platform	GCP
Operating System	Ubuntu Server
Containerization Orchestrator Software	Kubernetes
Machine Type	e2-standard-4
Number of Nodes	3
Version	1.27.3-gke.100
Number of vCPU	4
Total Memory	16 GB

Table 2: Required configuration for the GKE

The specification for AWS EC2 instance is as follows operating system is ubuntu, instance type is t2 large, number of CPU is 2, total memory is 50GB, and locust is

installed on this EC2 instance. We are using the Google Cloud(GCP) and AWS Cloud, on GCP we have Kubernetes available by default called Google Kubernetes Engine(GKE). We have created a standard e2 cluster on GCP. AWS cloud is used to install and run the locust software. Our test-app-hello application is deployed on Google Cloud using the application image stored on docker. The application has multiple endpoints like fallible where we can give pass weight and fail weight. Employee id where we can provide the employee-id and fetch the employee records. The application will be bombarded with multiple requests and the response will be captured and visualized.

## 4.2 System Architecture

In our system architecture as shown in figure 3, we have two microservices, test-app-hello and test-app-employee. Both microservices are deployed on different pods on GKE. A service is exposed for each microservice, which allows other microservices and clients to access them. Test-app-employee is accessing the test-app-hello in the system. The Istio control plane is deployed in a dedicated namespace called istio-system. The control plane consists of a number of components:

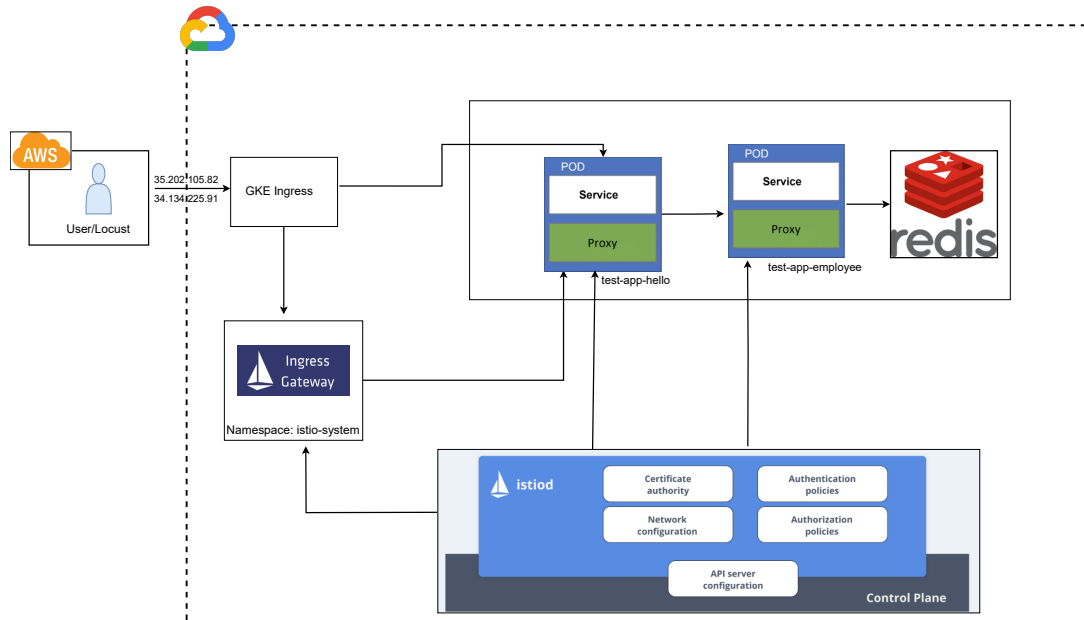


Figure 3: System Architecture

**Istio-ingress gateway:** This service acts as the entry point for external traffic into the Istio service mesh. It manages incoming requests and routes them to the right services in the mesh. The istio-ingress gateway is responsible for receiving incoming traffic, using Istio features like security, tracking, and routing rules, and then sending the traffic to services in the Istio mesh. It exposes services outside the cluster and handles traffic routing based on rules and configurations set.

**Istiod:** This component is the control plane part of Istio and is responsible for setting up and managing the whole service mesh. It organizes and manages how sidecar proxies that are used with services work. Istiod is in charge of finding services, managing traffic, setting security rules, and gathering data through monitoring. It ensures that all the microservices in the mesh follow the policies, traffic rules, and settings that the admin

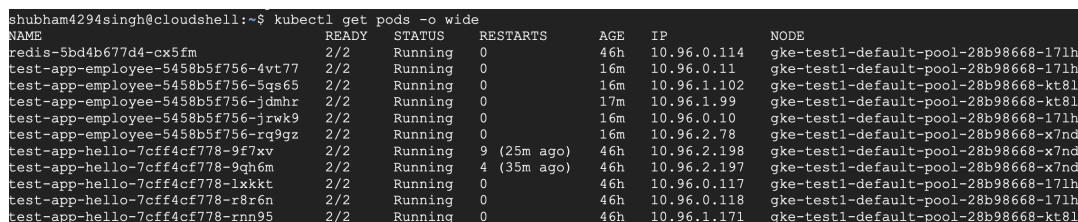
has set. It controls and makes sure that all the settings in the Istio service mesh are in sync with each other. Istio-ingressgateway and istiod are both very important parts of the Istio service chain. The istio-ingress gateway handles incoming data and routes it within the service mesh. The istiod manages and controls the service mesh as a whole, including how it acts and what it is set up to. A proxy is deployed in each pod. The proxy intercepts all traffic to and from the microservice and sends it to the Istio control plane. The control plane then decides how to route the traffic, based on the Istio configuration. An ingress gateway is used to expose the microservices to the outside world. The ingress gateway terminates incoming traffic and routes it to the appropriate microservice. The Redis DB which operates on a key-value pair, is used to store the employee records. It is connected to the test-app employee. The Locust is installed on AWS which will be used to run the load test on test-app-employee.

## 5 Implementation

The microservices called test-app-hello and test-app-employee has been developed using the FastAPI, a modern Python web framework, used for creating high-performance RESTful APIs. Here we have used the requests library for forwarding HTTP requests to external microservices. The application is built to have logging capabilities, providing detailed information about each request, including request and response IDs. The test-app-hello and test-app-employee both are running on the same namespace called “test-app”. For both the microservices, we are having 5 pods each.

### 5.1 Kubernetes Cluster Setup

Kubernetes, often known as K8s, is an open-source system for automating containerised application deployment, scaling, and management. For the experimentation, we have used Google Cloud Kubernetes Engine to create and configure a cluster named “test1” with a three-node setup. Within this cluster, we deployed our application, which is stored in Docker as an Image. To manage and deploy the application a YAML file named test-app-hello.yaml is written having the configuration details. Application is exposed by creating an service of type LoadBalancer and exposed on external IP 34.134.225.91. As in figure 4 our both microservices each have five pods running.



NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
redis-5bd4b677d4-cx5fm	2/2	Running	0	46h	10.96.0.114	gke-test1-default-pool-28b98668-171h
test-app-employee-5458b5f756-4vt77	2/2	Running	0	16m	10.96.0.11	gke-test1-default-pool-28b98668-171h
test-app-employee-5458b5f756-5gs65	2/2	Running	0	16m	10.96.1.102	gke-test1-default-pool-28b98668-kt8l
test-app-employee-5458b5f756-jdmhr	2/2	Running	0	17m	10.96.1.99	gke-test1-default-pool-28b98668-kt8l
test-app-employee-5458b5f756-jrwk9	2/2	Running	0	16m	10.96.0.10	gke-test1-default-pool-28b98668-171h
test-app-employee-5458b5f756-rg9gz	2/2	Running	0	16m	10.96.2.78	gke-test1-default-pool-28b98668-x7nd
test-app-hello-7cff4cf778-9f7xv	2/2	Running	9 (25m ago)	46h	10.96.2.198	gke-test1-default-pool-28b98668-x7nd
test-app-hello-7cff4cf778-9qh6m	2/2	Running	4 (35m ago)	46h	10.96.2.197	gke-test1-default-pool-28b98668-x7nd
test-app-hello-7cff4cf778-1xkkt	2/2	Running	0	46h	10.96.0.117	gke-test1-default-pool-28b98668-171h
test-app-hello-7cff4cf778-r8r6n	2/2	Running	0	46h	10.96.0.118	gke-test1-default-pool-28b98668-171h
test-app-hello-7cff4cf778-rnn95	2/2	Running	0	46h	10.96.1.171	gke-test1-default-pool-28b98668-kt8l

Figure 4: Kubernetes Pods

To deploy the application and make it accessible on the external IP a service has been created named “test-app-hello-ext” of type LoadBalancer. **Kubect1:** The Kubect1 command-line utility helps us execute commands on the Kubernetes cluster. With this, we can deploy our application, examine and do the admin tasks on the cluster, and access log data.

## 5.2 Istio Installation

It is installed on the Google Cloud Kubernetes Engine. Istio extends Kubernetes by creating a configurable, application-aware network that uses the robust envoy service proxy. In Istio, sidecar proxies are deployed along with each microservice. They help in many terms, such as traffic routing, where they intercept the incoming and outgoing requests, authorising Istio's control plane to dictate routing and other traffic policies for its management. Whether it is about dealing with Kubernetes or traditional workloads, Istio introduces consistent traffic management, telemetry, and security features to intricate deployments. It also gives us control over the service mesh. For the installation of Istio, we have created a new namespace named istio-system. In this namespace we have installed Istio service mesh, installing Istio in a separate namespace allows for better management, isolation, and organization of Istio resources. It helps keep Istio's parts from clashing with other resources in the cluster, which makes it easier to manage Istio-specific settings and better at allocating resources. Here we have used yet another markup file, also known as the YAML file, to explain how load balancing should work, how service-to-service authentication will work, monitoring, and much more. A YAML file basically describes how a pod should run and interact with other objects. YAML is popular because it is easy to understand for humans and is far more flexible. The YAML file for ingress gateway has main parameters as follows port-specifies the port information for the server and hosts-specifies the hosts that this gateway will accept traffic for. In our case, the \* indicates it will accept traffic for any host.

## 5.3 Locust Installation

In our study, we use the powerful load testing tool Locust to examine the operation of our Kubernetes-deployed microservice-based application. We mimic a spike in incoming requests to the 'test-app-employee' application using Locust. We can thoroughly assess the Microservice behavior under elevated user traffic by employing our load testing technique. Our goals are to test the scalability of the application and its separate services, comprehend how they react to increased request volumes, and locate any possible bottlenecks or performance problems. Locust is installed on the AWS cloud using "pip" (pip install locust). This command will download and install Locust and its dependencies from the Python Package Index (PyPI). We have preferred to install it on the cloud rather than the Google Kubernetes Engine or our local machine because we aim to streamline our processes to reduce the load on the system and minimize CPU usage. Because if we run a test on the system and suddenly experiencing a spike in CPU usage it can change the experiment results. This could happen because our system is running multiple processes in the background, and this can generate a sudden spike in CPU usage. We have written a bash script file to execute locust testing. To run locust first, we have to create a virtual environment and activate it. Once it's activated, we can run our .sh file(./test-employee.sh). In our bash script we will have three main components that we have configured:

**Users-** The number of virtual users who will access the system under load test. This parameter aids in simulating the expected traffic on our system in real-world scenarios.

**Spawn Rate-** The rate at which new virtual users will be spawned per second. High spawn rate means more the load test's ramp-up pace.

**Run Time-** Time for which the load test will run. This will control how long the load on our system will be simulated.

**Export\_locustfile="employee.py"**: Sets the `_locustfile` variable to the name of the Locust file (`employee.py` in our case).

**Html "reports/\${\_locustfile}-\${hostname}-\${\_now}.html"**: Generates an HTML report of the load test results with a file name based on the Locust file name, the host-name, and the timestamp.

## 5.4 Chaos Engineering

The script is made to execute commands for Kubernetes, particularly for deleting pods from a cluster. It is developed in the Bash scripting language. This script deletes a predetermined amount of available pods at predetermined intervals inside a Kubernetes cluster and searches continually for pods that match specific criteria. The script has following parameters:

**Max\_pods\_to\_kill="\$1:-3"**: Sets the maximum number of pods to kill. It takes the value passed as the first argument when running the script, defaulting to 3 if no value is provided.

**Kill\_interval="\$2:-5"**: Specifies the time interval between pod kills. It takes the value passed as the second argument when running the script, defaulting to 5 if no value is provided.

## 5.5 GitHub

GitHub is open source repositories allowing us to make changes push it to Git and merge those to earlier files. Also, when needed we can pull the code files on our local system to work upon. We have stored project files and the locust reports on it.

## 5.6 Experiment Execution

The experiments are run first without Istio and fault injection checking the application performance when load is applied using the locust. Again experiments are run now with Istio enabled now noticing the change in performance of microservices. After that will inject failures one test will be without Istio to note the impact on microservice and than with Istio enabled to check the performance of our proposed system. To compare the performance and collect the metrics we are getting the reports from Locust.

# 6 Evaluation

In this section, we will be performing multiple experiments to find the response time, failures and total request per second for the proposed system as well as for the default system. The proposed architecture will use GKE to run a three-node Kubernetes cluster and the created microservices will be installed over it. The experiment will be run with the same settings and configurations to get accurate results. Results will be calculated and analyzed using the locust reports.



## 6.1 Experiment 1

### 6.1.1 Without Istio without Fault Injection

In this scenario we will run the experiment by load testing on test-app-employee without enabling Istio and without the fault injection. Figure 5 showcases the total request per

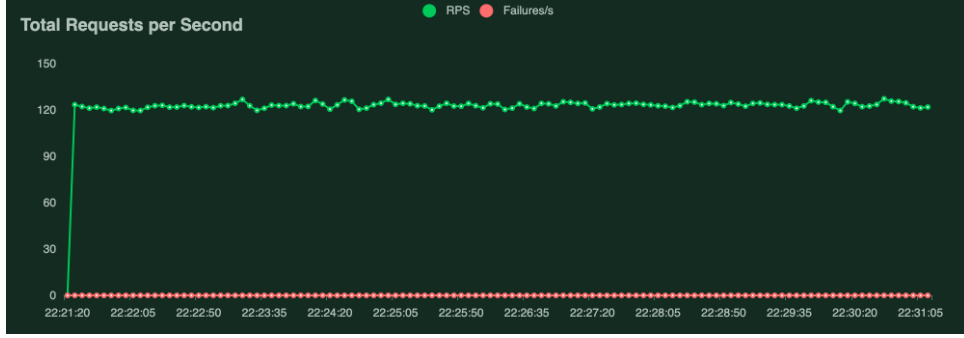


Figure 5: Total requests per second without Istio & Pod failure

second and the failure rate is indicated by the red marking. Figure 6 is showing response time(ms) for the user request. For load testing we ran Locust with settings: 290 users, spawn rate 100, and runtime 300 seconds. Under stress, the microservice was stable. Figure 6 the response time ranged between 200-325 ms, figure 5 total requests per second remained constant around 120 RPS and the failure rate was zero as the fault was not injected in this test.

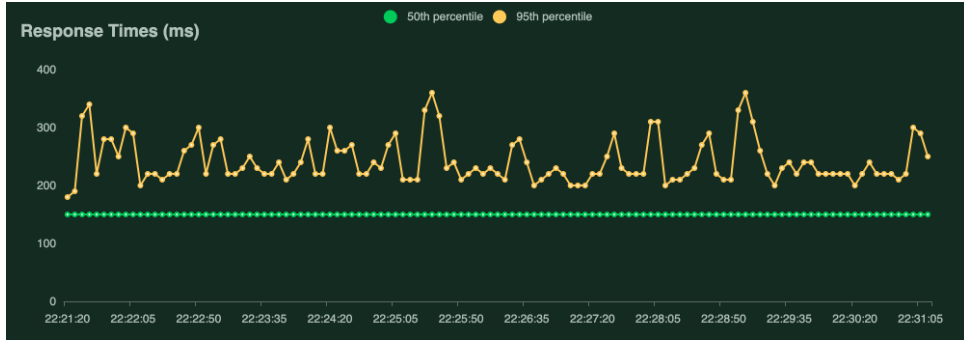


Figure 6: Response time(ms) without Istio & Pod failure

### 6.1.2 With Istio without Fault Injection

In this scenario we will run the experiment by load testing on test-app-employee by enabling the Istio and without the fault injection.

The experiment was kept similar to the previous experiment, but with Istio installed. The microservice exhibited better stability compared to the previous experiment. Istio's tuned failure recovery mechanisms were active but not triggered due to the absence of deliberate pod failures. The response time maintained consistency and hovered around the 200 ms range as in figure 8, which is much better than the previous one. Also, the

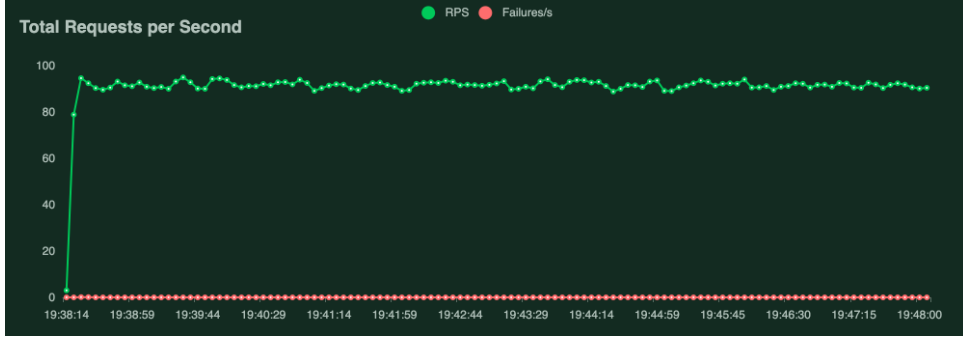


Figure 7: Total requests per second without Pod failure(Istio Enabled)

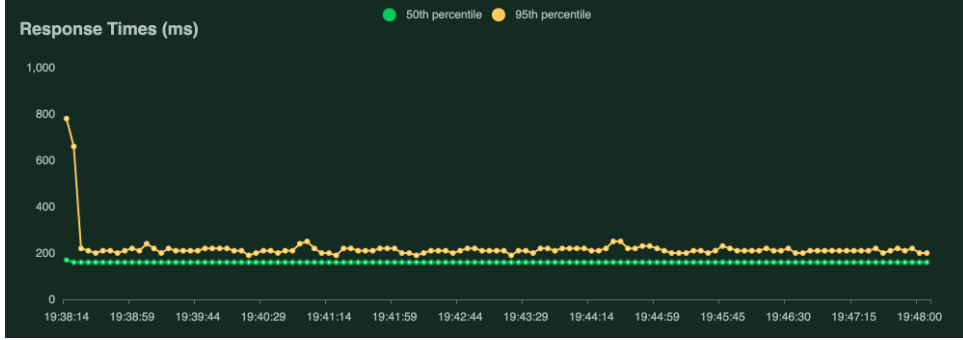


Figure 8: Response time(ms) without Pod failure(Istio Enabled)

total requests per second remained stable at around 100 RPS as in figure 7, showcasing the resilience of the proposed architecture even without encountering pod failures.

**Inference:** After enabling the Istio from figures 7 and 8, we conclude that the microservices show a significant improvement in response time (ms), whereas the total request per second has dropped from 120 to 95 RPS.

## 6.2 Experiment 2

### 6.2.1 Pod Failure and without Istio

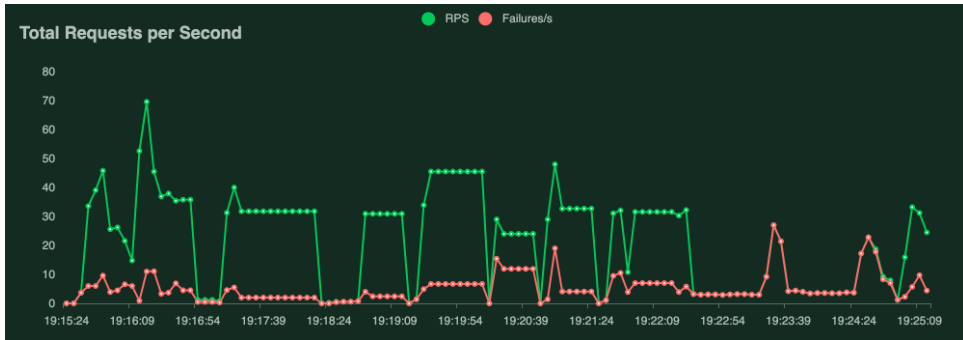


Figure 9: Total requests per second with Pod failure(Istio Disabled)

In this scenario, we will run the experiment by load testing on test-app-employee without enabling the Istio and with the fault injection. For load testing, we ran locust with the same configuration settings as mentioned in experiment 1. While under load, the microservice are injected with pod failures. Response time and requests per second were affected. Response time showed a wider fluctuation, peaking at 80,000 ms as in figure 10. Total requests per second peaked at 70 RPS and failure rates showed a more erratic pattern due to the concurrent pod failures, hovering between 5-31 as in figure 9.

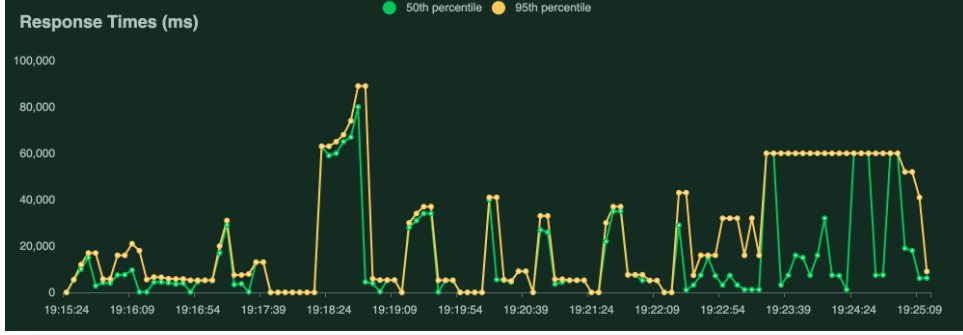


Figure 10: Response time(ms) with Pod failure(Istio Disabled)

### 6.2.2 With Istio and Pod Failure

In this scenario we will run the experiment by testing on 'test-app-employee' by enabling the Istio and with the fault injection.

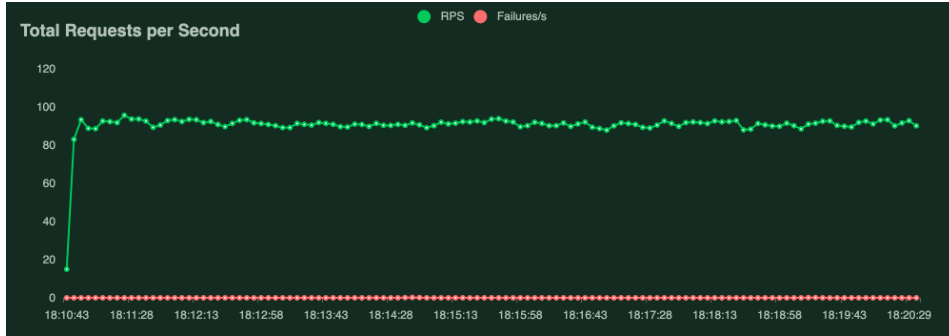


Figure 11: Total requests per second with Pod failure & Istio

The experiment was run with the same configurations but here we have done chaos engineering by introducing pod failure. Istio's tuned settings showcased resilience against concurrent pod failures. The microservice maintained a response time range of 210-380 ms as in figure 12. Total requests per second were around 97 RPS as in figure 11. The results show a clear improvement in response time from 80,000 ms to 380 ms. Despite concurrent failures, requests per second remained relatively stable due to Istio's recovery actions.

**Inference:** After enabling the Istio with the fault injection (pod failures) from figures 11 and 12, we conclude that the microservices show a great improvement in response time (ms), the total request has stabilised at 100 RPS, and the error rate is zero as the proposed system is recovering from the error.

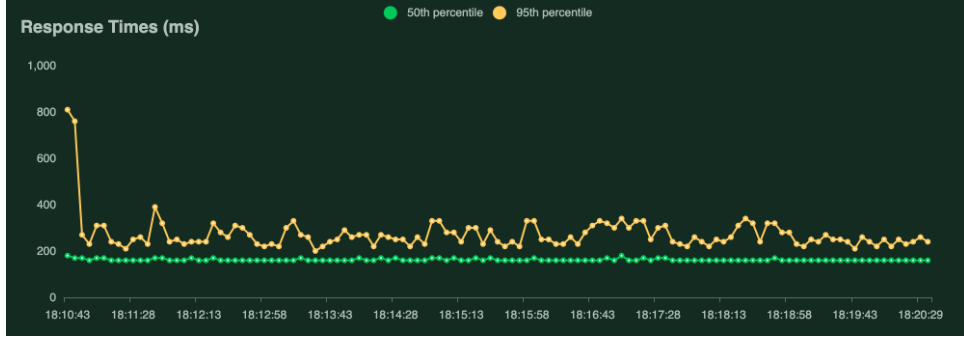


Figure 12: Response time(ms) with Pod failure & Istio

## 6.3 Experiment 3

### 6.3.1 Without Istio and with Pod failure for extended duration

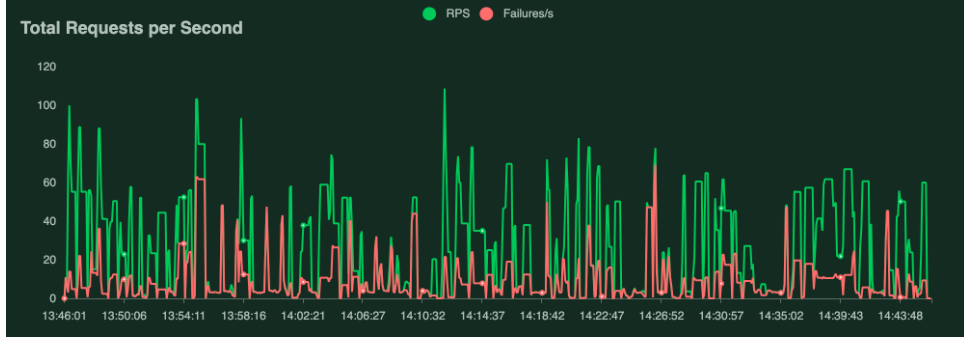


Figure 13: Total requests per second with Pod failure & without Istio

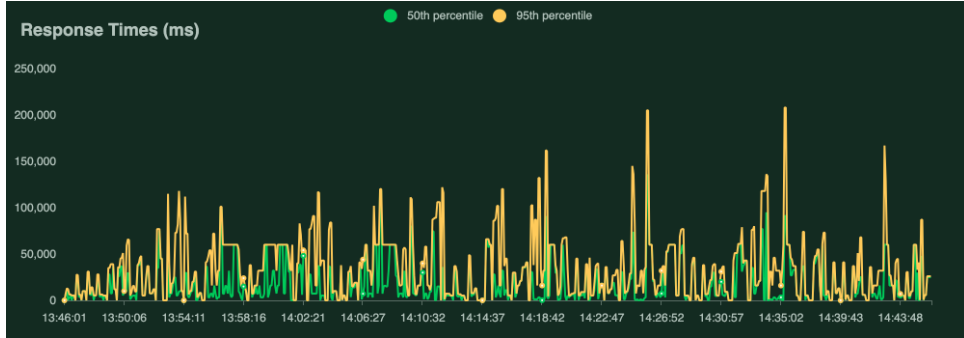


Figure 14: Response time(ms) with Pod failure & without Istio

We replicated the experiment here, but for a large user base, and the duration of load testing was also increased to 1 hour to test whether the proposed architecture works for a large number of users accessing the microservices continuously for a long time.

The proposed system works exceptionally well, the response time for the pod failure scenario (without Istio) reaches 200,000 ms as in figure 14 and the total request per second peaked at 100 RPS as in figure 13.

### 6.3.2 With Istio and Pod Failure for extended duration

With pod failure and Istio enabled the response time hovers around 3800 ms as in figure 16 and total request per second was 120 RPS as in figure 15.

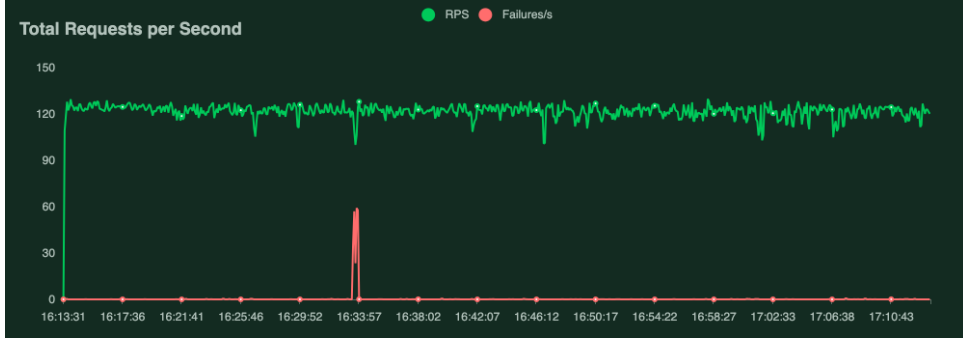


Figure 15: Total requests per second with Pod failure & Istio

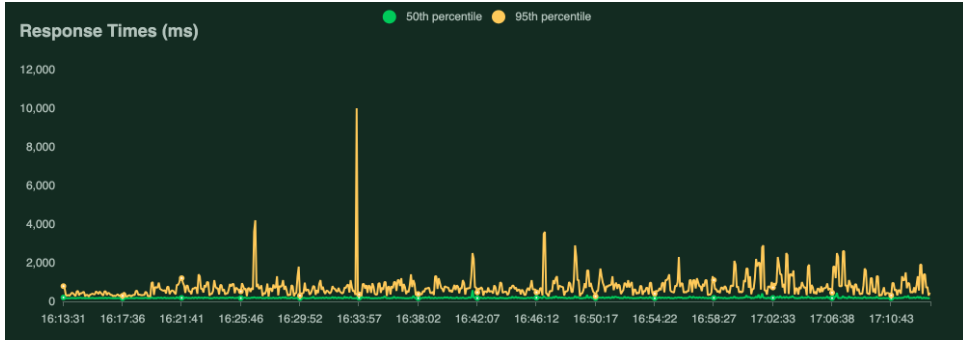


Figure 16: Response time(ms) with Pod failure & Istio

**Inference:** After enabling the Istio with the fault injection (pod failures) from figures 16 and 15, we conclude that the microservices show a great improvement in response time(ms), the total request has stabilised at 120 RPS, and the error rate is averaging out to be zero as the proposed system is recovering from the error.

## 6.4 Discussion

The experiments provided valuable insights into the behavior of the system when subjected to different loads. After carefully examining all of the experiments and assessing them based on the research conducted by Flora et al. (2023), who have conducted studies on the aging and fault tolerance of microservices in Kubernetes. They utilized a tuned Kubernetes probe with fault injection on two microservice testbeds, and it is evident that our study produced exceptional results, showcasing the advantages of incorporating Istio. When Istio was not installed during the load testing without pod failures, the system demonstrated major variations in response time, with values hovering around 300 ms, as shown in figure 6. However, with the Istio enabled and referring to figure 8, the response time remains consistently stable at approximately 200 ms, indicating a significant improvement. The total request per second was approximately equal in both cases, at

around 100 RPS as shown in figures 7 and 5. When the application experienced a heavy load, its performance took a significant hit. The response time increased dramatically reaching a staggering 80,000 ms, as shown in figure 10. Additionally, the total number of requests per second fluctuated unpredictably, with an average of around 70 RPS, as depicted in figure 9.

These numbers indicate a lack of stability when subjected to a heavy load. These experiments demonstrated the vulnerability of the system to concurrent pod failures when Istio's coordinated resilience features are not present. Nevertheless, by making certain adjustments during the installation of Istio, the outcomes show a remarkable enhancement. Despite experiencing pod failures, as shown in figure 9 and 12, the microservices demonstrated a significant increase in requests per second, improving from 80,000 ms to 350 ms. These numbers indicate a remarkable improvement. The total requests per second remained relatively stable, hovering around 97 RPS, as shown in figure 11. In addition, the system experiences a significant number of failures when Istio is not present. However, once Istio is enabled, the error rate drops to zero, as shown in figure 11. This demonstrates how Istio handles the error. In general, it demonstrates how effective Istio is at reducing the impact of pod failures. Even when subjected to high workloads and a prolonged testing period, the proposed system consistently demonstrates superior performance, as illustrated in figures 14, 13, 16, and 15. With Istio enabled, the response time has significantly improved to 1500 ms and the total number of requests per second has increased to 120 RPS. This is a significant improvement compared to the figures shown in figure 14 and 16, where Istio was not enabled and the response time was 80,000 ms with a total of 85 RPS. Highlight its ability to handle high user traffic and extended periods of use. These findings demonstrate the effectiveness of Istio in ensuring consistent performance and stability, especially in demanding scenarios. They emphasize the important role Istio plays in enhancing system resilience and mitigating pod failures. One aspect that could be assessed in the future is the overall number of user requests achieved when Istio is enabled, which is a potential limitation of its implementation. With the introduction of Istio, additional components like sidecar proxies are brought into the mix. As a result, this extra layer of functionality adds to the processing load. This can be examined and further explored to determine how it can be eradicated. Another aspect that requires additional investigation is the financial implications of deploying a service mesh. This comprehensive setup has the potential to substantially raise the overall project expenses.

## 7 Conclusion and Future Work

The analysis of microservices performance, resilience, and stability under varying loads and simulated failures highlighted the importance of Istio in improving system robustness. The system demonstrated significant performance loss, emphasizing the exposure to multiple pod failures when the load testing was performed without enabling the Istio service mesh. However, with Istio installed and fine-tuned, remarkable performance improvements were observed. Not only during the normal user request period, but even under a high number of user requests, the microservices showcased improved request processing and reduced response times, showcasing Istio's efficacy in mitigating the impact of pod failures. Furthermore, when exposed to even higher loads and extended test durations, the proposed system performed better than earlier, exhibiting stability, robustness, and resiliency. The experiment results validate Istio's effectiveness in maintaining improved

system performance and stability, especially under high user calls, highlighting its critical role in strengthening containerised microservices resilience against failures.

One challenge of using Istio is the steep learning curve due to its complexity, performance overhead from sidecar proxies, and increased resource usage. The study can be continued further by analysing the effect of having extra component Istio on total user requests and a study on cost performance to understand the trade-offs between the improvement in performance achieved by Istio and the associated resource utilisation costs.

**Video Link**-[https://studentncirl-my.sharepoint.com/:v:/g/personal/x22170341\\_student\\_ncirl\\_ie/ERTTSIR1qRxLhgh3mD9tt0kB0naB9Rf9dijB1ZpzxOAJiQ?email=Shaguna.Gupta](https://studentncirl-my.sharepoint.com/:v:/g/personal/x22170341_student_ncirl_ie/ERTTSIR1qRxLhgh3mD9tt0kB0naB9Rf9dijB1ZpzxOAJiQ?email=Shaguna.Gupta)

## References

- Alhamazani, K., Ranjan, R., Mitra, K., Jayaraman, P. P., Huang, Z., Wang, L. and Rabbi, F. (2014). Clams: Cross-layer multi-cloud application monitoring-as-a-service framework, *x(x)*: 22–34.
- Anjali, Khatri and Vishal (2020). *Mastering Service Mesh: Enhance, secure, and observe cloud-native applications with Istio, Linkerd, and Consul*, Packt Publishing Ltd.
- Balalaie, A., Heydarnoori, A., Jamshidi, P. and Lynn, T. G. (2018). Microservices migration patterns, *Software: Practice and Experience* **48**(3): 298–306.
- Barakat, S. A. (2017). Monitoring and analysis of microservices performance, *x(x)*: 86–99.
- Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes, *IEEE Cloud Computing* **1**(3): 81–84.
- Calcote, L. and Butcher, Z. (2019). *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*, O'Reilly Media.
- Camilli, M., Guerriero, A., Janes, A., Russo, B. and Russo, S. (2022). Microservices integrated performance and reliability testing, *Journal Name x(x)*: 41–69.
- Camilli, M. and Russo, B. (2021). Modeling performance of microservices systems with growth theory, *Journal of Empirical Software Engineering* pp. 505–514.
- Dame and Michael (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)*, IEEE, pp. 89–91.
- Ganguli, M., Ranganath, S., Ravisundar, S., Layek, A., Ilangovan, D. et al. (2021). Challenges and opportunities in performance benchmarking of service mesh for the edge, *Journal Name x(x)*: 111–129.
- Großmann, M. and Klug, C. (2017). Monitoring container services at the network edge, *Journal Name x(x)*: 44–66.

- Islam, S. S., Gibson, J. A., Morrison, P. and Rahman, A. (2022). Benefits, challenges, and research topics: A multi-vocal literature review of kubernetes, *arXiv e-prints* pp. arXiv–2211.
- Karn, R. R., Das, R., Pant, D. R., Heikkonen, J. and Kanth, R. (2022). Automated testing and resilience of microservice’s network-link using istio service mesh, *IEEE* .
- Lauretis, L. D. (2019). From monolithic architecture to microservices architecture, *Journal Name* **x(x)**: 26–44.
- Li, W., Lemieux, Y., Gao, J., Zhao and Han, Y. (2019). Service mesh: Challenges, state of the art, and future research opportunities, *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, IEEE, pp. 122–1225.
- Malik, S., Naqvi, M. A. and Moonen, L. (2023). Chess: A framework for evaluation of self-adaptive systems based on chaos engineering, *Simula Research Laboratory* **170**: 110798.
- Malki, E., Amine, Zdun and Uwe (2019). Guiding architectural decision-making on service mesh-based microservice architectures, *Software Architecture: 13th European Conference*, Vol. 13, Springer International Publishing, pp. 3–19.
- Meinke, K. and Nycander, P. (2015). Learning-based testing of distributed microservice architectures: Correctness and fault injection, *Journal Name* **x(x)**: 37–42.
- Monteiro, L. A., Hazin, R. R., Lima, A. C. D. and Almeida, W. H. C. (2017). Survey on microservice architecture - security, privacy and standardization on cloud computing environment, *Icsea 2017* **1(3)**: Conference.
- Noor, A., Jha, D. N., Mitra, K., Jayaraman, P. P., Souza, A., Ranjan, R. and Dustdar, S. (2019). A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments, **x(x)**: 87–99.
- Pietrantuono, R., Russo, S. and Guerriero, A. (2018). Run-time reliability estimation of microservice architectures, *Journal Name* **x(x)**: 57–77.
- Ponomarev and Y., K. (2019). Attribute-based access control in the service mesh, *2019 Dynamics of Systems, Mechanisms and Machines (Dynamics)*, IEEE, pp. 1–4.
- Rosenthal, C. and Jones, N. (2020). *Chaos Engineering: System Resiliency in Practice*, O’Reilly Media.
- Seifermann, V. (2017). Monitoring and analysis of microservices performance, **x(x)**: 24–47.
- Simonsson, J., Zhang, L., Morin, B., Baudry, Benoit and Monperrus, M. (2021). Observability and chaos engineering on the system calls for containerized applications in docker, *Future Generation Computer Systems* **122**: 117–129.
- Toffetti, G., Brunner, S., Blochlinger, M., Dudouet, F. and Edmonds, A. (2015). An architecture for self-managing microservices, *Journal Name* **x(x)**: 68–81.
- Torkura, A., K., Sukmana, M. I., Cheng, Feng and Christoph, M. (2020). Cloudstrike: Chaos engineering for security and resiliency in cloud infrastructure, *IEEE Access* **8**: 123044–123060.