UNIVERSITAS INDONESIA

WEB APPLICATION PERFORMANCE ANALYSIS OF MULTI-REGION GEO-DISTRIBUTED KUBERNETES CLUSTERS

SKRIPSI

JONATHAN NICHOLAS
1906293133

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI ILMU KOMPUTER
DEPOK
MEI 2023

UNIVERSITAS INDONESIA

WEB APPLICATION PERFORMANCE ANALYSIS OF MULTI-REGION GEO-DISTRIBUTED KUBERNETES CLUSTERS

SKRIPSI

Diajukan sebagai salah satu syarat untuk memperoleh gelar
Gelar Jurusan Anda

JONATHAN NICHOLAS
1906293133

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI ILMU KOMPUTER
DEPOK
MEI 2023

HALAMAN PERNYATAAN ORISINALITAS

Skripsi ini adalah hasil karya saya sendiri,
dan semua sumber baik yang dikutip maupun dirujuk
telah saya nyatakan dengan benar.

Nama	:	**Jonathan Nicholas**
NPM	:	**1906293133**
Tanda Tangan	:	[illegible]
Tanggal	:	**09 Juni 2023**

HALAMAN PENGESAHAN

Tugas Akhir ini diajukan oleh	:	
Nama	:	Jonathan Nicholas
NPM	:	1906293133
Program Studi	:	Sarjana Ilmu Komputer
Judul	:	Analisis Performa Aplikasi Web Service dalam Multi-Region Geo-Distributed Kubernetes Cluster

Telah berhasil dipertahankan di hadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana Ilmu Komputer pada Program Studi Sarjana Ilmu Komputer, Fakultas Ilmu Komputer, Universitas Indonesia

DEWAN PENGUJI

Pembimbing 1	:	Muhammad Hafizhuddin Hilman, S.Kom., M.Kom., Ph.D.	(Nilai telah diberikan melalui SISIDANG pada 26-06-2023, 10:12:10) (Revisi telah disetujui melalui SISIDANG pada 17-07-2023, 12:02:35)
Penguji	:	Amril Syalim, S.Kom., M.Eng., Ph.D.	(Nilai telah diberikan melalui SISIDANG pada 17-07-2023, 11:31:01) (Revisi telah disetujui melalui SISIDANG pada 17-07-2023, 11:31:10)
Penguji	:	Made Harta Dwijaksara, S.T., M.Sc., Ph.D.	(Nilai telah diberikan melalui SISIDANG pada 21-06-2023, 09:46:46) (Revisi telah disetujui melalui SISIDANG pada 14-07-2023, 21:37:06)

Ditetapkan di	: Depok, Jawa Barat
Tanggal	: 17 Juli 2023

ACKNOWLEDGEMENT

I would like to express my gratitude to God Almighty, whom without His blessings and grace, I would not be able to write this thesis.

First of all, I am deeply grateful to my supervisor, Muhammad Hafizhuddin Hilman, S.Kom., M.Kom., Ph.D. for his guidance and support throughout the research process. I'm truly fortunate to have the opportunity to learn from him and gain valuable research experience.

I would also like to acknowledge the companies I've worked at, as they provided me with important technical and non-technical skills which were integral to the completion of this research project. Furthermore, I would like to thank my colleagues who have supported me throughout my academic studies. Lastly, I'm immensely grateful to my parents and sister for believing in me and for their unwavering support.

Depok, 09 Juni 2023

Jonathan Nicholas

HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Universitas Indonesia, saya yang bertanda tangan di bawah ini:

Nama	:	Jonathan Nicholas
NPM	:	1906293133
Program Studi	:	Ilmu Komputer
Fakultas	:	Ilmu Komputer
Jenis Karya	:	Skripsi

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Indonesia **Hak Bebas Royalti Noneksklusif (*Non-exclusive Royalty Free Right*)** atas karya ilmiah saya yang berjudul:

Analisis Performa Aplikasi Web Service dalam Multi-Region Geo-Distributed Kubernetes Cluster

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Universitas Indonesia berhak menyimpan, mengalihmedia/formatkan, mengelola dalam bentuk pangkalan data (*database*), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyataan ini saya buat dengan sebenarnya.

Dibuat di	:	Depok
Pada tanggal	:	09 Juni 2023

Yang menyatakan

[illegible]

(Jonathan Nicholas)

ABSTRAK

Nama	:	Jonathan Nicholas
Program Studi	:	Ilmu Komputer
Judul	:	Analisis Performa Aplikasi Web Service dalam Multi-Region Geo-Distributed Kubernetes Cluster
Pembimbing	:	Muhammad Hafizhuddin Hilman, S.Kom., M.Kom.

Kebutuhan untuk menyediakan layanan kepada pengguna di seluruh dunia menyebabkan layanan aplikasi web untuk berdaptasi menggunakan teknologi baru dan memadai. Untuk mencapai hal tersebut, layanan cloud servis digunakan untuk memperluas jangkauan geografis dari layanan web di seluruh dunia. Peningkatan kualitas pengembangan *deployment* aplikasi web terlihat pada Kubernetes, alat yang diadopsi secara luas yang didukung di sebagian besar platform cloud, yang memungkinkan penerapan *geo-distributed clusters* untuk aplikasi yang memiliki pengguna multinasional. Dikarenakan kelangkaan studi mengenai *geo-distributed clusters* dan kinerjanya, penelitian ini bermaksud untuk menjembatani kesenjangan pengetahuan tersebut dengan mengimplementasikan solusi menggunakan Istio (Anthos Service Mesh), mesh layanan yang paling banyak digunakan untuk aplikasi Kubernetes, serta solusi cloud native di Google Cloud Platform menggunakan MultiClusterService. Studi ini menemukan bahwa kedua pendekatan tersebut dapat diandalkan, namun, Istio/ASM memiliki latensi yang sedikit lebih rendah untuk sebagian besar *request*. Kedua pendekatan tersebut merupakan pilihan baik untuk aplikasi global, karena keduanya menggunakan *geo-aware load balancing*, yang merutekan permintaan pengguna ke klaster terdekat yang tersedia. Basis kode studi dan hasil pengujian ini tersedia secara *open-sourced* untuk studi lebih lanjut tentang aplikasi berbasis *geo-distributed Kubernetes clusters*.

Kata kunci:
Kubernetes, Google Cloud Platform, MultiClusterService, Istio, Anthos Service Mesh

ABSTRACT

Name	:	Jonathan Nicholas
Study Program	:	Computer Science
Title	:	Web Application Performance Analysis of Multi-Region Geo-Distributed Kubernetes Clusters
Counsellor	:	Muhammad Hafizhuddin Hilman, S.Kom., M.Kom.

With the need of providing services to ever-growing worldwide users, web application services must adapt new technologies in order to fulfill these needs. As setting up physical servers across the globe is a daunting task, cloud service providers are an essential tool to reach geographical coverage for worldwide web services. Further advancements on the developer experience of deploying web applications can be seen in tools such as Kubernetes, a widely adopted tool that's supported in most cloud platforms that enables the implementation of geo-distributed clusters for applications with a multi-national user base. However, there is a scarcity of studies regarding geo-distributed clusters methods and its performance. Therefore, this study intends to bridge that knowledge gap by implementing a solution using Istio (Anthos Service Mesh), the most used service mesh for kubernetes applications as well as a cloud native solution on Google Cloud Platform using MultiClusterService. This study found that both approaches are reliable, however, Istio / ASM has a slightly lower latency for the vast majority of requests. In addition, both approaches are a viable choice for worldwide applications, as they both use geo-aware load balancing, which routes user requests to the nearest available cluster. This study's scripts and test results are open-sourced for further studies about geo-distributed Kubernetes-based applications.

Key words:
Kubernetes, Google Cloud Platform, MultiClusterService, Istio, Anthos Service Mesh

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

ATTACHMENTS

CHAPTER 1
INTRODUCTION

This chapter discusses the background, problem definition, research objectives, research scopes, and writing systematics. The goal of this chapter is to describe the background of the research topic and the problem that is intended to be solved while defining objectives, scopes, and writing systematics as a guide for this research.

1.1 Background

The number of internet users increases every year with approximately 5.3 billion users in 2023 (*Cisco Annual Internet Report (2018–2023) White Paper*, 2022). In anticipation of such growth, applications which serve users from all over the globe must be able to handle those kinds of traffic. Therefore, to handle traffic from different parts of the globe with high performance, multiple servers are deployed in multiple regions of the globe. An on-premise server approach introduces multiple complexities such as hardware and maintenance costs for every single physical machine. A simpler approach is to use a cloud-based service such as Google Cloud Platform.

Google Cloud Platform (GCP) is a cloud computing service that offers a wide range of technical solutions to help aid the development and deployment of web applications. One of the services that Google Cloud Platform provides is Google Kubernetes Engine, a cloud solution for containerized applications using Kubernetes. Kubernetes is an open-source platform that provides an abstraction of containers and simplifies deploying, monitoring, and scaling a web-based application. Containerized applications improve the developer experience as developers do not need to worry about the deployment process and instead can focus entirely on application development (Xie & Govardhan, 2020).

Kubernetes on the cloud offers a solution for applications with an international user base called geo-distributed clusters. Geo-distributed clusters are Kubernetes clusters that are spread across the world in order to minimize the latency of server response by reducing the physical distance between a user and the server, thus creating a better user experience. In a geo-distributed cluster configuration, traffic handling is done by a load balancer. The load balancer ideally distributes traffic efficiently to Kubernetes clusters. Kubernetes clusters are chosen based on several factors such as request-to-response distance, cluster workload (CPU) percentage, and many more. However, according to Andrew (2023), Google Cloud's load balancer does not handle geo-distributed applications very well. Therefore, there is a need to explore other load-balancing options that can work on a cloud-based application. An alternative is to use a service mesh.

A service mesh is an infrastructure layer used to manage communication between services. A major benefit of using service mesh is a more configurable load balancing which has the upside of having locality-aware load balancing which is a load balancer that routes a user to the closest server. Three of the service mesh with the most stars on GitHub are Istio, Consul, and Linkerd respectively. As the purpose of this study is not to compare different service meshes, Istio is chosen for being supported by the Google Cloud Platform under the name of Anthos Service Mesh. Istio is a service mesh compatible with existing Kubernetes clusters. Istio offers locality-aware load balancing that considers the incoming request's geographical location to determine which server clusters are used to process the response with the goal of increasing the performance of web applications.

1.2 Problem Definition

This research has the following problem definition:

- How does a geo-distributed cluster architecture improve the reliability of worldwide applications?
- How does the Istio approach improve application performance?

1.3 Research Objectives

This research has the following objectives:

- To implement and evaluate the effects of google cloud load balancing on the performance of a worldwide application.
- To implement and evaluate the effects of locality load balancing on the performance of a worldwide application.

1.4 Research Scopes

This research has the following scopes:

- Application performance testing in multi-region geo-distributed clusters is limited to simple web service applications.
- Multi-region geo-distributed cluster testing is limited to the cloud platform by Google Cloud Platform (GCP).

1.5 Writing Systematics

The research report consists of six chapters, namely the introduction, literature review, methodology, implementation, results and analysis, and conclusions. The following are the descriptions of each chapter,

- Chapter 1 INTRODUCTION
  This chapter is the introduction to this research which consists of the background, problem definition, research objectives, research scopes, and writing systematics.
- Chapter 2 LITERATURE REVIEW
  This chapter discusses the theoretical foundation of this study from literature reviews to bridge the gap between theory and practice.
- Chapter 3 METHODOLOGY
  This chapter discusses the research methodology which includes research stages, application infrastructure design, testing scenarios, and evaluation metrics.
- Chapter 4 DESIGN AND IMPLEMENTATION
  This chapter discusses the implementation of the application which is then deployed according to the test scenarios.
- Chapter 5 RESULTS AND ANALYSIS
  This chapter discusses the findings from the experiments and presents the analysis from each of the test scenarios.
- Chapter 6 CONCLUSION
  This chapter discusses the conclusion of this research as well as suggestions for future research.

CHAPTER 2
LITERATURE REVIEW

This chapter displays the results of the literature study that has been done to help further the research. The literature study is based on relevant topics that are used in this research.

2.1 Kubernetes

Kubernetes is used to manage, monitor, and scale containers of a cluster and can be extended to geo-distributed clusters when hosted on a cloud platform. At first, application deployment is done by hosting the application on a dedicated machine on-premise. This changed when Cloud providers were introduced, where Cloud-based services helped improve deployment management through monitoring and scalability through cloud servers, removing the need of maintaining a physical server. The deployment process itself did not change much until a breakthrough in the form of Docker containers was introduced in 2010.

Docker utilizes virtual machines that offer consistency and reliability, regardless of which machine it's being hosted on. This means that application behavior is consistent regardless of the machine itself as long as it is capable of running Docker. Docker's containerization solution, however, is limited to a single application. In reality, a complete application consists of multiple layers, for example, a three-tier application consisting of a frontend, backend, and database layer. In this case, Kubernetes extends Docker's containerization solution to a complete and production-ready application by managing containers and enabling them to communicate with each other.

Additionally, containers are deployed as Pods, the smallest unit of a Kubernetes resource. Furthermore, a higher level Service resource is used to group Pods, manage their lifecycle, and create replicas of them. (Jeffery, Howard, & Mortier, 2021). This is proven to be an extremely useful feature, as the Service resource can be grouped by unique identifiers such as service name and namespace, which can be used to deploy multiple Service resources inside multiple clusters. This is the main idea of geo-distributed clusters, where Kubernetes services are interconnected through clusters which has benefits such as geo-aware routing and additional reliability measures through cluster failover, which we will discuss in the next section.

2.2 Geo-distributed Clusters

The full capabilities of a Kubernetes application are put on display with the help of cloud providers. By using traditional cloud resources in conjunction with edge/fog computing resources located at the network edges, application execution can be done closer to data sources and consumers, thus increasing the scalability of applications and decreasing their response time (Rossi, Cardellini, Lo Presti, & Nardelli, 2020). Cloud providers such as Google Cloud, Amazon Web Services, and Azure, all have servers available in different areas of the globe, making them a great candidate for the adoption of this application architecture. By utilizing the wide coverage of cloud servers, an application can be deployed to every available region to cater to the users of each region without the need of having an on-premise self-hosted physical server. This can be seen in Figure 2.1, where this approach provides a better server coverage than just a centralized deployment.

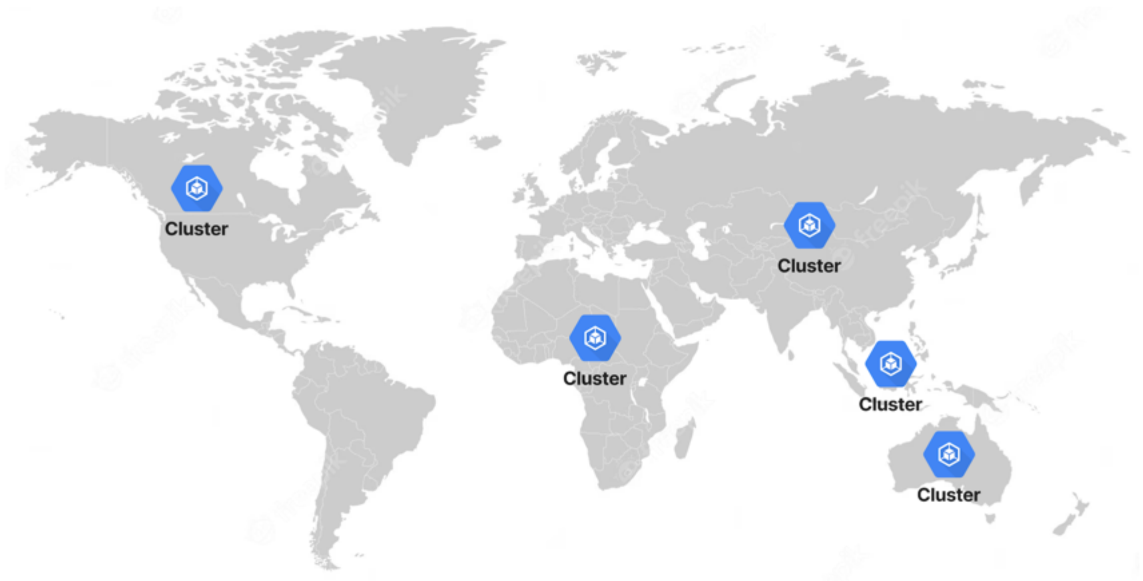


Figure 2.1: Geographically distributed clusters in different areas of the world.

Geo-distributed clusters are multiple clusters hosted in different areas of the region. When an application's user base grows and spreads globally, a single central deployment may not offer the best user experience. The physical distance of a user to the server matters, which makes geo-distributed application deployment a good solution to accommodate users from all over the world. With the help of Kubernetes and a cloud provider, multi-regional deployment has never been easier. It is, however, a relatively new area in software development, and not much has been studied about the different approaches of this architecture. For this research, we have chosen two approaches: MultiClusterService with MultiClusterIngress and Istio / Anthos Service Mesh, as they are both are supported on the Google Cloud Platform.

In addition, geo-distributed clusters can be extended to databases, which can be seen in implementations such as CocroachDB. As different countries have different regulations, it is becoming a necessity for multinational companies to store user data in their respective region in order to comply with regulations. Data should be stored close to the users who access it most frequently and follow them when they travel to prevent high latencies caused by long-distance data retrieval (Taft et al., 2020). We used a simpler approach to achieve database geo-awareness, however, as geo-distributed databases introduce added complexities such as partition and replication, which are outside of this research scope.

2.3 MultiClusterService (MCS) with MultiClusterIngress (MCI) (MCS with MCI

To connect identical services across geo-distributed clusters, Fleet, MultiClusterIngresss, and MultiClusterService must be configured. A Fleet is a group of clusters that are visible to the MultiClusterIngress and can be used as backends. Inside a Fleet, a cluster needs to be designated as a config cluster where MultiClusterIngress and MultiClusterService are configured and deployed. Clusters that are registered to a Fleet besides the config cluster are called member clusters.

MultiClusterService (MCS) is a custom GKE resource that allows services with the same selectors to be considered the same by creating derived services in the member clusters. The derived service creates a Network Endpoint Group (NEG) in every target cluster which tracks pod endpoints and allows service discovery, as shown in Figure 2.2. MCS integrates with Google's Fleet to define target clusters to create derived service resources. By default, every cluster registered to a Fleet is considered a target cluster.

MultiClusterIngress (MCI) is a custom resource that sends traffic to the default backend MCS or based on rules configured to route hosts to a certain backend MCS and creates a layer 7 Virtual IP (VIP) address that routes traffic to backends that are configured in multiple clusters as MCS and can be accessed by public users. MultiClusterService and MultiClusterIngress are configured identically to their non-multi counterpart and are only deployed in the config cluster.

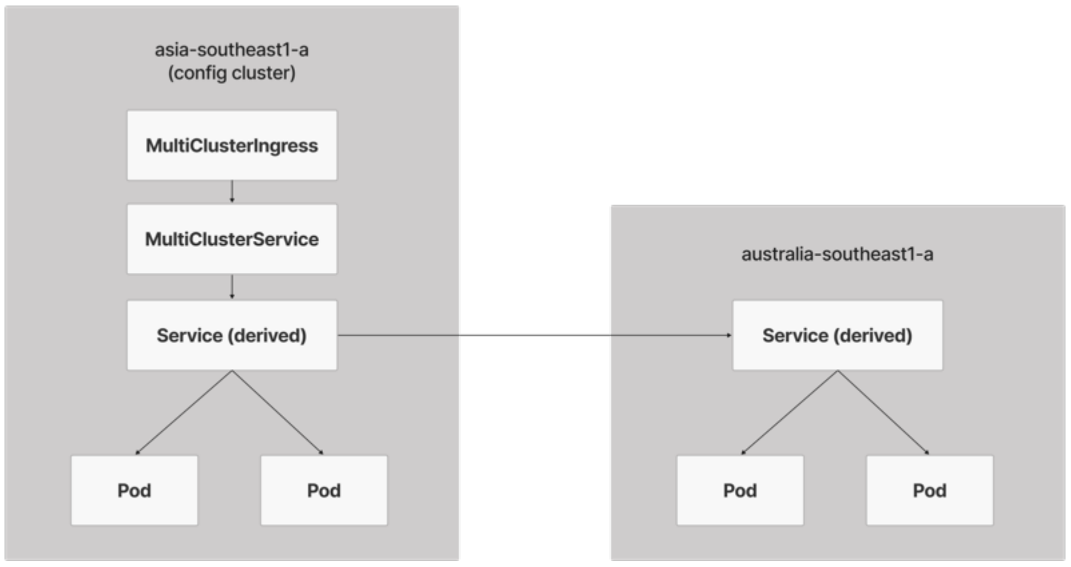


Figure 2.2: Service derivation using MultiClusterIngress and MultiClusterService.

In a multi-cluster GKE setup using MCS with MCI, MCI deploys load balancers across clusters registered to a Fleet. The type of load balancer deployed by an MCI is a global external load balancer and is categorized as north-south routing, meaning that it routes traffic from outside to inside of the data center, as seen in Figure 2.3.

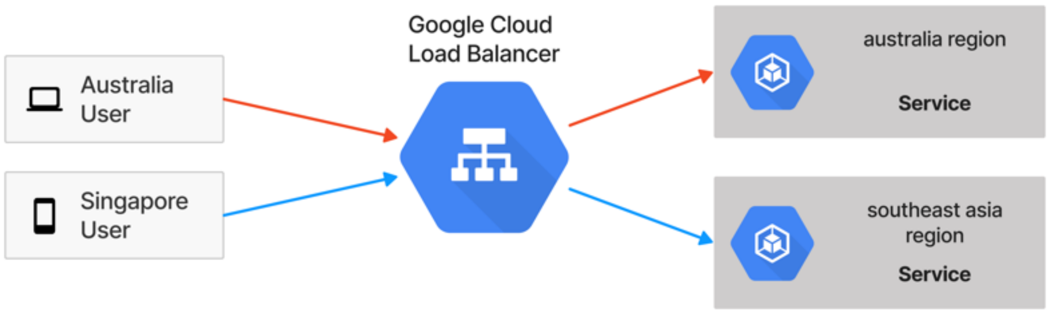


Figure 2.3: Google Cloud Load Balancer traffic routing.

The global external load balancer routes traffic to the closest Google Point of Presence (PoP) and uses a single anycast IP address deployed to multiple regions (Google, 2023b).

There is also a premium network tier that offers cold-potato routing to Google PoPs and hot-potato routing inside Google networks.

2.4 Istio / Anthos Service Mesh

The other option for multi-cluster Kubernetes deployment presented in this research is using Istio / Anthos Service Mesh. Istio is an open-sourced service mesh platform that is used to connect microservices and adds features such as observability, security, and traffic management. Istio on GKE is not supported since September 2022 but users can still use Istio under the name of Anthos Service Mesh on the Google Cloud Platform. Anthos Service Mesh (ASM) is compatible with Istio custom resources. ASM has the benefit of a managed control plane that handles automatic upgrades as well as a managed data plane that automatically upgrades sidecar proxies and injected gateways by evicting old pods that are running a prior version (Google, 2023a). Automatic sidecar injection is the core of Istio's functionality, allowing pods to be injected with the Envoy proxy at creation, as shown in Figure 2.4. Envoy proxy mediates all inbound and outbound traffic for all services in a service mesh. For a multi-cluster Anthos Service Mesh, ASM integrates with the Fleet API just like the MCS with MCI approach, but there is no config cluster. Instead, installing ASM on every cluster is needed to register them to the service mesh.

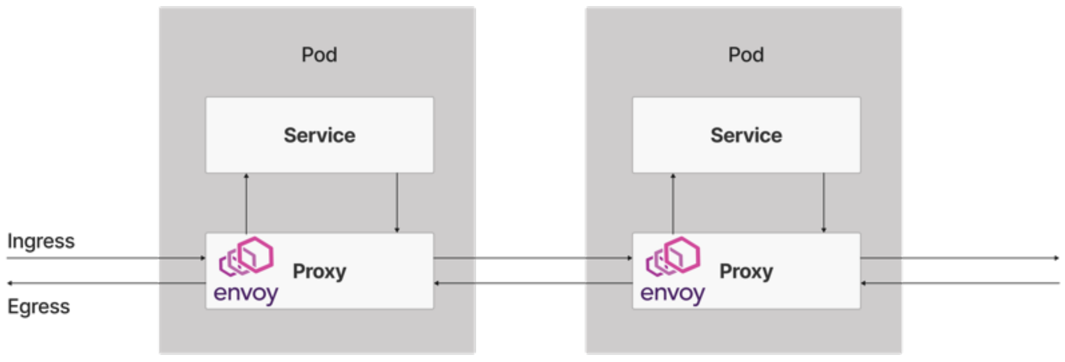


Figure 2.4: Envoy proxy injection and interaction with service inside of a pod.

The default load balancer algorithm for Istio is round-robin, where traffic is distributed to clusters in a circular way such that each cluster will be chosen once in a cycle. One of the benefits of this approach is that it prevents an instance where a single server is overloaded by requests by distributing traffic equally to all servers. An approach to reduce latency by closing the physical gap between user and server, locality load balancing is proposed as a better solution for an application with geo-distributed clusters. Locality load balancing is a feature of Istio from using another open-sourced application called Envoy. Locality load balancing, also known as geographical load balancing, is a location-aware load balancer that can efficiently route traffic to the nearest cluster available and can be seen in Figure 2.5. To enable locality load balancing, a DestinationRule resource must be configured. DestinationRule defines rules that apply to traffic for a service. DestinationRule can be configured to enable load balancing and outlier detection which work together to determine healthy hosts that are able to handle the traffic.

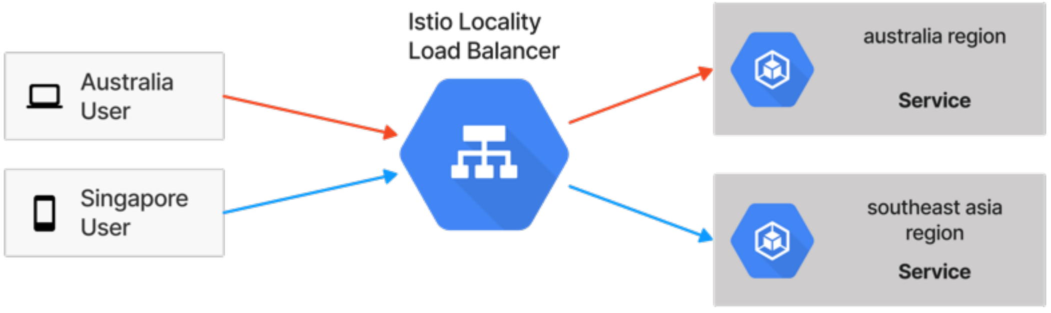


Figure 2.5: Istio Locality Load Balancer traffic routing.

The main difference between these two methods is their architecture. Istio / ASM introduces an additional layer of complexity, built on top of the Kubernetes architecture to add functionalities that utilize Istio's proxy injection on each pod. Meanwhile, the MCS with MCI approach is an extension of the Kubernetes API, where additional resources such as MultiClusterService and MultiClusterIngress are used to take advantage of Google Cloud's multi-region servers. In addition, MCS with MCI only has a fraction of features compared to Istio / ASM, as it is intended to be used in conjunction with other Google services such as the Google Cloud Load Balancer.

2.5 Performance

Performance is an important part of web applications and can be measured using different evaluation metrics. The metrics that can measure an application's performance are latency, requests per second, and success rate. Latency is the amount of time it takes for a server to respond to complete a user's request. Lower latency is better, as it results in more requests that are able to be handled and reduces the load of a server. Most importantly, lower latency creates a better user experience as it minimizes the time a user needs to wait while using an application. Latency itself doesn't paint the full picture of application performance, requests per second (RPS) can help measure an application's ability to handle multiple users simultaneously and is a measurement of how many requests an application can handle in a period of time. During peak hours when the application traffic is at its highest, having a high RPS reduces the occurrence of a server bottleneck.

Furthermore, another important metric for a web application to have is reliability. A web service's reliability is the likelihood of a server finishing a request under certain time and workload conditions (Bhalerao & Ingle, 2019). Measuring a server's reliability is useful to determine how well a server can handle certain real-world scenarios such as high-traffic hours, where a website can expect a heightened number of requests. Without application reliability, businesses that rely on web applications could potentially lose a significant amount of revenue as users aren't able to complete their transactions due to their servers not being able to respond to the client's requests. To measure the percentage of requests being processed as intended, success rate is a metric to quantify just that. Success rate is a measure of how often an application successfully responds to the user's request. A failure can occur when a user inputs something unintended or when a server is unable to process a request caused by the server being overloaded and is indicated by a response code outside of [200, 400], inclusive. The latter definition is used when referring to a failed response as requests are validated and standardized in testing scenarios.

To measure the effectiveness of each multi-cluster scenario, performance testing such as load testing and stress testing is done to simulate scenarios where an application receives various amounts of traffic with the possibility of server failure. Load testing is done to measure the performance of a web application at a certain load level to ensure that the website can be worked within the range of multi-user concurrency requirements (Yu, 2019). Load testing is also done to identify a server's maximum performance capability by evaluating performance metrics on every increase in load level. On the other hand, stress testing is done to analyze the web application's ability to do failure recovery, which happens after a server is forced to restart due to a server failure. Stress testing can identify a machine's readiness to adopt certain changes that consume resources, such as a service mesh. Performance testing and evaluation metrics are crucial to determine the performance of each approach for geo-distributed clusters.

CHAPTER 3
METHODOLOGY

This chapter explains the methodology used in this research. The discussion in this chapter includes research stages, application infrastructure design, testing scenarios, and evaluation metrics.

3.1 Research Stages

There are several stages in this research, which include problem formulation, literature study, application design and implementation, performance evaluation and analysis, and inference. The entirety of the research stages can be seen in Figure 3.1

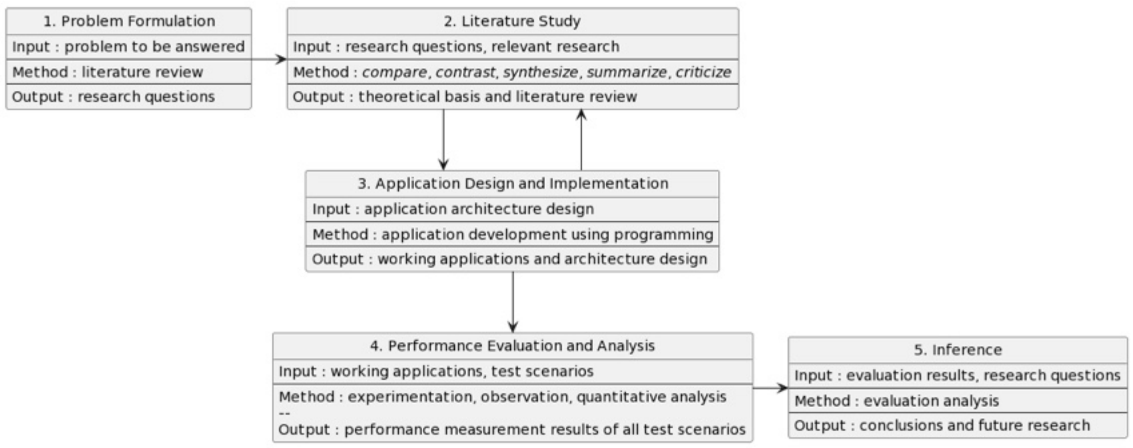

1. Problem Formulation
Input : problem to be answered
Method : literature review
Output : research questions
2. Literature Study
Input : research questions, relevant research
Method : *compare, contrast, synthesize, summarize, criticize*
Output : theoretical basis and literature review
3. Application Design and Implementation
Input : application architecture design
Method : application development using programming
Output : working applications and architecture design
4. Performance Evaluation and Analysis
Input : working applications, test scenarios
Method : experimentation, observation, quantitative analysis
--
Output : performance measurement results of all test scenarios
5. Inference
Input : evaluation results, research questions
Method : evaluation analysis
Output : conclusions and future research

Figure 3.1: Research stages.

In the first stage, problem formulation, existing problems are used with the help of literature review to produce research questions that are answered in the conclusion of this study. In the second stage, literature study, relevant research is used to define/create the basis of this research. The methods used in the literature study stage are the following: compare, contrast, synthesize, summarize, and criticize which helps to get the theoretical foundation and literature review of this research.

In the third stage, application design and implementation, the web application architecture is designed and implemented to be used as a common variable for testing to be done. The configuration of geo-distributed Kubernetes clusters and Istio service mesh is also done in this stage. In the fourth and penultimate stage, testing is done, and each testing scenario is evaluated to answer research questions and acquire a conclusion for this study, which is done in the final stage.

3.2 Testing Scenarios

In the testing scenario, load testing and stress testing are done to evaluate the performance of each geo-distributed cluster configuration. Being a quantitative research and quantitative experimentation, there are variables that need to be established. The independent variables are the different geo-distributed Kubernetes cluster configurations, load balancer algorithm, and the different requests per second configured during performance testing. For each geo-distributed cluster configuration, testing is done by increasing the RPS after each test case. Each test case occurs for 5 seconds to simulate a short period where a server experiences high traffic, for example during a flash sale. The RPS used in the testing configuration are 10, 50, and 100 where 100 is the upper limit chosen from the number of errors that occurred during preliminary tests. The independent variables are shown in Table 3.1.

Table 3.1: Configuration of geo-distributed cluster method, load balancer, and requests per second

Geo-Distributed Cluster Method	Load Balancer	Requests Per Second (RPS)
MCS with MCI	Multi-Cluster Geo-Aware	10
		50
	Single Cluster	100
Istio / ASM	Multi Cluster Geo-Aware	10
		50
	Single Cluster	100

The product of all of the different variables results in 12 different testing scenarios, which are repeated three times for each scenario to reduce variability. The dependent variables are evaluation metrics which include the number of requests, latency, and success ratio. In addition, each test scenario is executed independently from one another to avoid the effects of external variables. The test implementation details can be seen in section 4.4.

3.3 Evaluation Metrics

When analyzing web performance, several metrics can define an application's performance. These metrics include the number of requests made which is evaluated in terms of total requests, request rate, and throughput. Additionally, latency is another metric that tells us the amount of time taken from sending a request until the first byte of the response is received and is tracked in the form of the minimum latency, the maximum latency, the mean latency as well as 50th, 90th, 95th and 99th percentile.

Aside from performance, the purpose of having a high number of requests per second in the test scenario is to evaluate the server's capability to return to a healthy state after experiencing a server failure. The success ratio provides insight into the number of successful responses, defined by having a response code between 200 and 400 (non-inclusive), and is used to determine a configuration's resiliency.

CHAPTER 4
DESIGN AND IMPLEMENTATION

This chapter explains the design and implementation of this research. The discussion consists of the application, Kubernetes, MCS with MCI geo-distributed cluster, Istio / ASM geo-distributed cluster, and performance testing.

4.1 Application

There are two areas that were configured in the complete web application, the application server itself and the geo-distributed Kubernetes clusters. In the first step, we implemented the application itself. We chose a two-tiered application consisting of a server and a database tier for this research as it covers all the main components that were tested. The client tier is absent and is replaced by the Vegeta load testing tool.

After the application design/architecture was configured, next are the application implementation details. First of all, the server is a REST API that communicates with a database and simply returns the cluster name of where the responding server is located. We wrote the server in the Go programming language for its speed as well as language uniformity with Docker and Kubernetes. Furthermore, its implementation can be seen in `main.go`[1]. Finally, it was containerized using Docker and uploaded to Docker Hub to be used by Kubernetes. In addition, we chose Redis as the database for its simplicity as well as being lightweight, an important aspect of web performance. The integration of Kubernetes inside of a development workflow creates a separation of concern between application development and infrastructure development, which allows for concurrent progress between the two different areas and is a big reason why Docker and Kubernetes are widely adopted.

The next step after application configuration was the geo-distributed clusters configuration consisting of MCS with MCI and Istio / ASM. First, in the MCS with MCI configuration, we deployed the server service as a MultiClusterService custom resource, which was routed by the MultiClusterIngress custom resource and allows cross-cluster

[1]`https://github.com/jojonicho/skripsi/blob/master/server/main.go`

communication. Following this, a virtual IP address was created, which uses a Google Cloud Load Balancer to route traffic to the nearest cluster. The infrastructure for the MCS with MCI geo-distributed Kubernetes clusters can be seen in Figure 4.1.

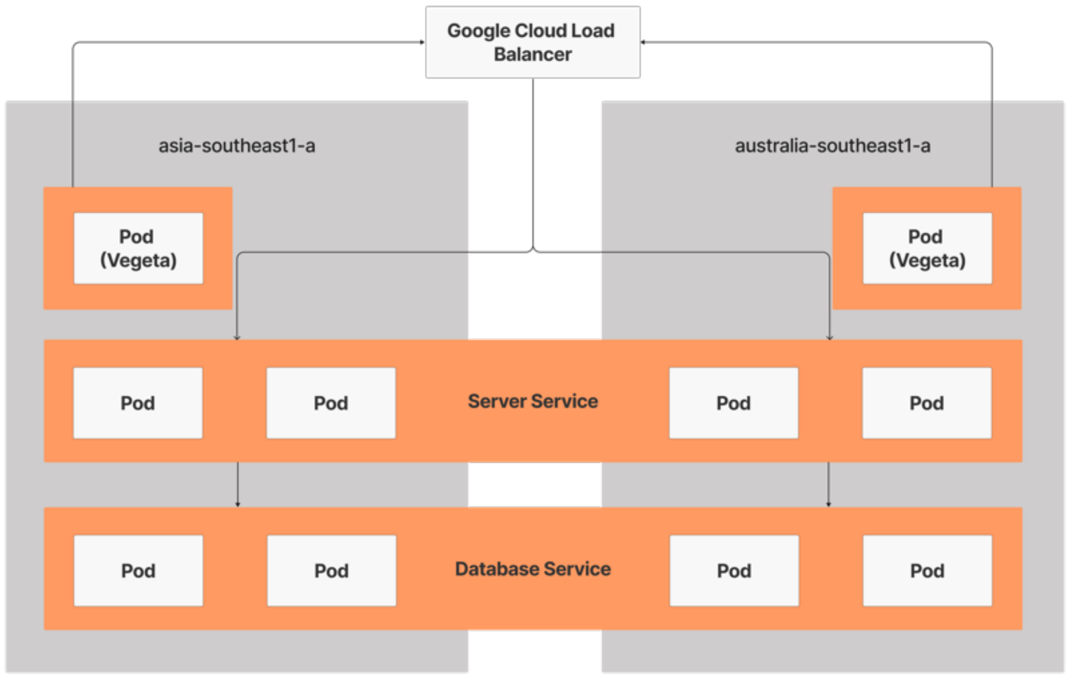


Figure 4.1: Geo-distributed Kubernetes clusters infrastructure on the MCS with MCI configuration.

Next, in the Istio / ASM configuration, we installed the Istio control and data plane in each cluster, which enabled automatic envoy proxy injection to each pod. We then deployed the DestinationRule resource to configure locality load balancing, which routes traffic to the nearest cluster. Following this, we deployed the Vegeta load testing pod in every region to simulate a request call from a specific region. We decided to use the server layer as a data layer, which is a server connected to each database instance because it is not possible to load-balance the database layer directly, as the database instance is a run-time dependency. Lastly, the Istio / ASM geo-distributed Kubernetes clusters' infrastructure can be seen in Figure 4.2.

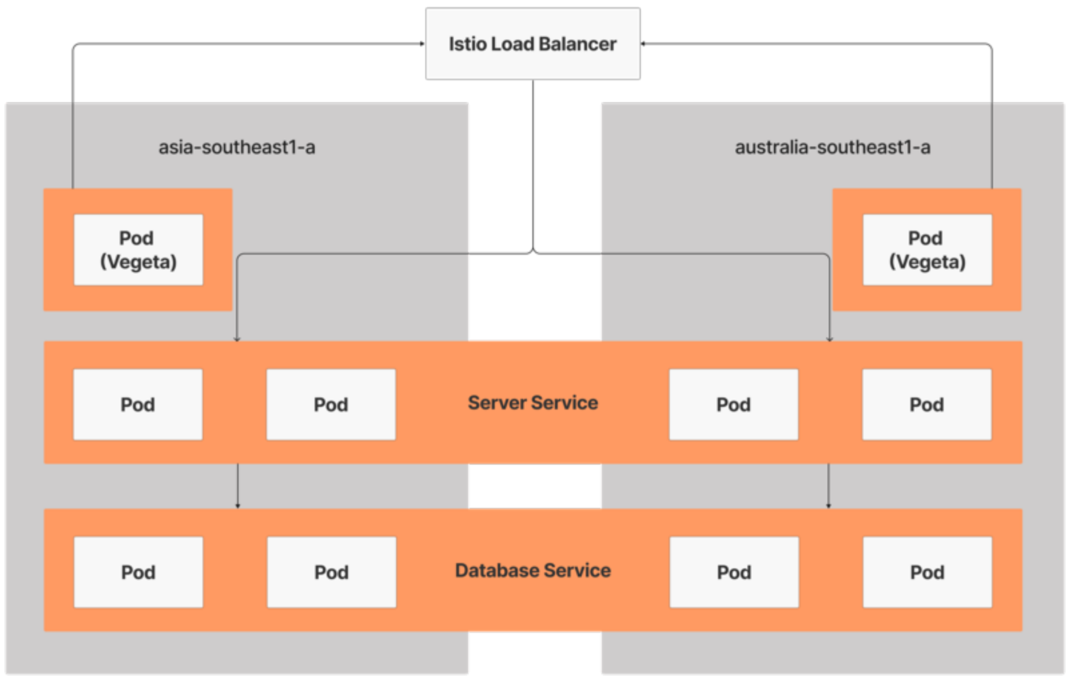


Figure 4.2: Geo-distributed Kubernetes cluster infrastructure on Istio / ASM configuration.

4.1.1 Kubernetes

Besides implementing the application, configuring and implementing a Kubernetes architecture was another important step to successfully deploy a geo-distributed cluster web application. Kubernetes implementation and configuration were done through YAML files, which we then applied using the `kubectl` command-line interface on Google Kubernetes Engine clusters, where its implementation can be seen in the GitHub repository[2]. One of the main selling points of Kubernetes is its networking features, which allow services to communicate with each other and conveniently map each service name defined in the metadata to a URL value that can be provided as an environment variable. For example, to connect the `ta-server` service with the `ta-redis` service, simply provide `ta-redis-service` inside of the server YAML file[3], and the redis URL can be accessed inside of the server application as an environment variable.

All of the Kubernetes YAML files were applied on Google Kubernetes Engine clusters. Each of the clusters was created using the `gcloud` command as shown in **Code** 4.1 and each having the configuration shown in Table 4.1.

[2]`https://github.com/jojonicho/skripsi`
[3]`https://github.com/jojonicho/skripsi/blob/master/server.yaml`

```
export PROJECT_ID=`gcloud config get-value project`
export M_TYPE=e2-standard-4
export ZONE=asia-southeast1-a
export CLUSTER_NAME=${PROJECT_ID}-${RANDOM}

gcloud services enable container.googleapis.com

gcloud container clusters create $CLUSTER_NAME \
--cluster-version latest \
--machine-type=$M_TYPE \
--num-nodes 4 \
--zone $ZONE \
--workload-pool=$PROJECT_ID.svc.id.goog \
--project $PROJECT_ID \
```

Code 4.1: Google Kubernetes Engine cluster creation script.

Table 4.1: Cluster configurations.

Configuration	Value (s)
Version	1.25.7-gke.1000
Nodes	4
Machine type	e2-standard-4

In order to implement a geo-distributed cluster configuration, regardless of whether it's an MCI with MCI or Istio / ASM implementation, we first registered each cluster to the same Fleet. In addition, workload identity was enabled to access Google Cloud services, which can be done by providing a workload pool, as shown in **Code** 4.2.

```
gcloud container clusters update $CLUSTER_NAME \
    --zone=$ZONE \
    --workload-pool=$PROJECT_ID.svc.id.goog

gcloud container fleet memberships register $CLUSTER_NAME \
   --gke-cluster $ZONE/$CLUSTER_NAME \
   --enable-workload-identity \
   --project $PROJECT_ID
```

Code 4.2: Fleet registration script.

4.2 MCS with MCI

In an MCS with MCI scenario, a config cluster was chosen amongst the available clusters, as shown in **Code** 4.3, where `CLUSTER_NAME` is the cluster name of the chosen config cluster. The **Code** 4.3 script enabled the deployment of custom resources for MultiClusterService and MultiClusterIngress. The final step was to apply the MCS and MCI resources as shown in `MCS.yaml`[4] and `MCI.yaml`[5] using `kubectl` inside of the config cluster.

```
# only execute on config cluster
gcloud container hub ingress enable \
  --config-membership=$CLUSTER_NAME
```

Code 4.3: Config cluster multi-cluster script.

Deploying a MultiClusterService resource on the config cluster created a derived service resource in each of the clusters registered to a Fleet while deploying the MultiClusterIngress resource created a Virtual IP Address which can be attained using the `kubectl describe mci ta-server-ingress` command. To access the URL, we turned off any form of VPN as Google Cloud has connection problems when using them.

4.3 Istio / ASM

In the Istio / ASM scenario, Anthos Service Mesh was installed in each cluster using `asmcli`. Once installed in each cluster, we restarted each deployment pod to enable automatic proxy injection by running the `kubectl rollout restart deployment` command. Finally, we applied the DestinationRule resource, as shown in **Code** 4.4, to enable locality load balancing.

[4]`https://github.com/jojonicho/skripsi/blob/master/mcs.yaml`
[5]`https://github.com/jojonicho/skripsi/blob/master/mci.yaml`

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: ta-server-destionationrule
spec:
  host: ta-server-service.sharedvpc.svc.cluster.local
  trafficPolicy:
    loadBalancer:
      localityLbSetting:
        enabled: true
        failover:
        - from: asia-southeast1-a
          to: australia-southeast1-a
        - from: australia-southeast1-a
          to: asia-southeast1-a

    outlierDetection:
      splitExternalLocalOriginErrors: true
      consecutiveLocalOriginFailures: 10

      consecutive5xxErrors: 1
      interval: 1s
      baseEjectionTime: 2s
```

Code 4.4: DestinationRule configuration for locality load balancing.

In addition, we used the DestinationRule resource to configure locality failover, which allowed traffic to be routed to other clusters when a cluster isn't healthy. To enable this, outlier detection was configured, which enabled sidecar proxies to determine if a service is unhealthy.

4.4 Performance Testing

Load testing and stress testing were done using Vegeta, an open-sourced load tester, which we chose for its set of features, mainly the ability to create HTTP requests and report metrics that aligns with the evaluation metrics defined in the methodology of this research. To simulate requests from different regions, we deployed the load testing tool as a containerized Docker image and ran it on each region, allowing the request to originate from a chosen region. We then repeat the same test for every request per second (RPS) configuration, which can be seen in `vegeta.sh`[6] and `asm-vegeta.sh`[7] for the MCS with MCI configuration and Istio / ASM configuration, respectively.

[6]https://github.com/jojonicho/skripsi/blob/master/tests/vegeta.sh
[7]https://github.com/jojonicho/skripsi/blob/master/tests/asm-vegeta.sh

Finally, we used the `vegeta report` command to transform the collected data into human-readable metrics. It can also produce a plot-friendly percentile table by supplying `-type=hdrplot` to the report command, as seen in **Code** 4.5, which allows further analysis of latency percentile distribution.

```
RPS_LIST=(10 50 100)
OUTPUT_DIR=$1

for RPS in "${RPS_LIST[@]}"
do
  vegeta report -type=hdrplot $OUTPUT_DIR/results.${RPS}rps.bin >
    $OUTPUT_DIR/${RPS}.hgrm
  python3.8 plot.py $OUTPUT_DIR/${RPS}.hgrm $OUTPUT_DIR/${RPS}.png
done
```

Code 4.5: Service and Deployment configuration for ta-redis application.

CHAPTER 5
RESULTS AND ANALYSIS

This chapter discusses the test results and analysis. The discussion includes reliability analysis and performance analysis.

5.1 Reliability

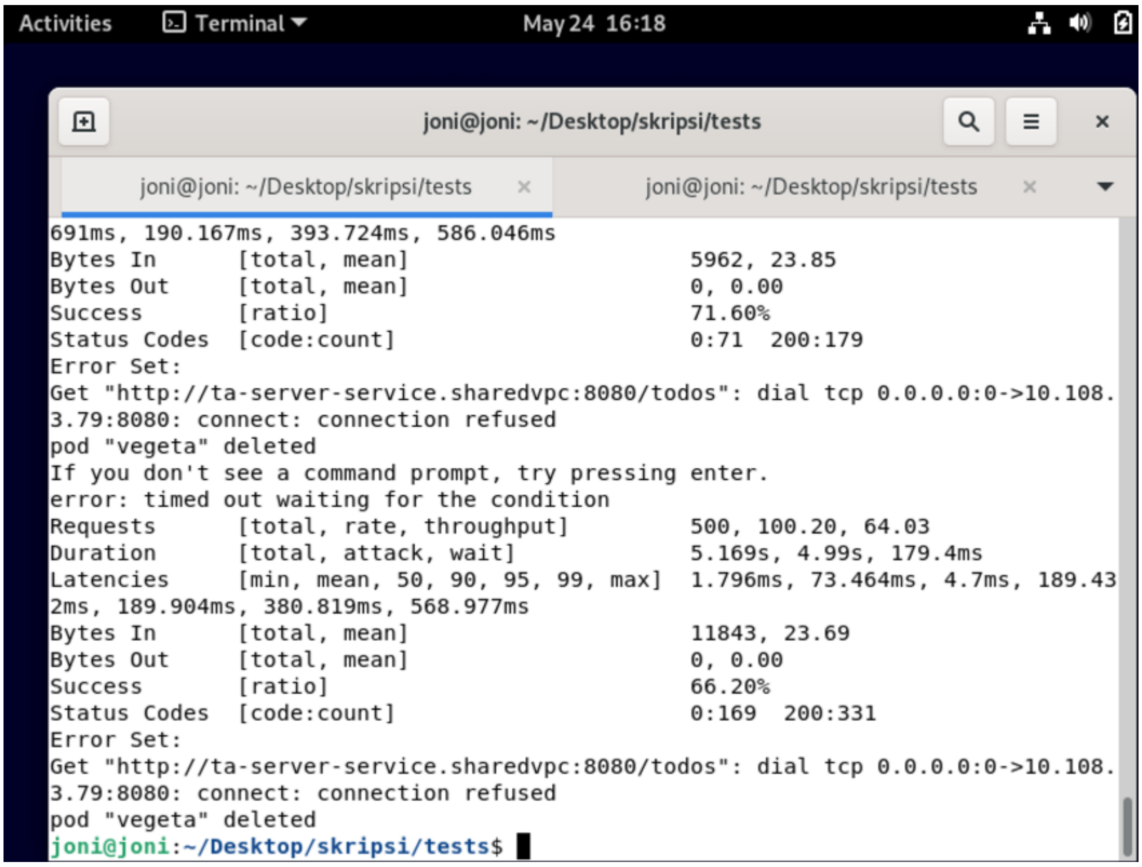
To evaluate the reliability of geo-distributed clusters, we tested the single-cluster variation as well for comparison. In addition, analyzing the single-cluster results also helps us understand the behavior of each geo-distributed cluster method, mainly its recovery and failover mechanism.

After configuring and testing each scenario, the results can be seen in Table 5.1 and Table 5.2 for single cluster and multi-cluster test results, respectively. The main result that determines reliability is the success ratio for each method.

Table 5.1: Single Cluster Reliability Test Results

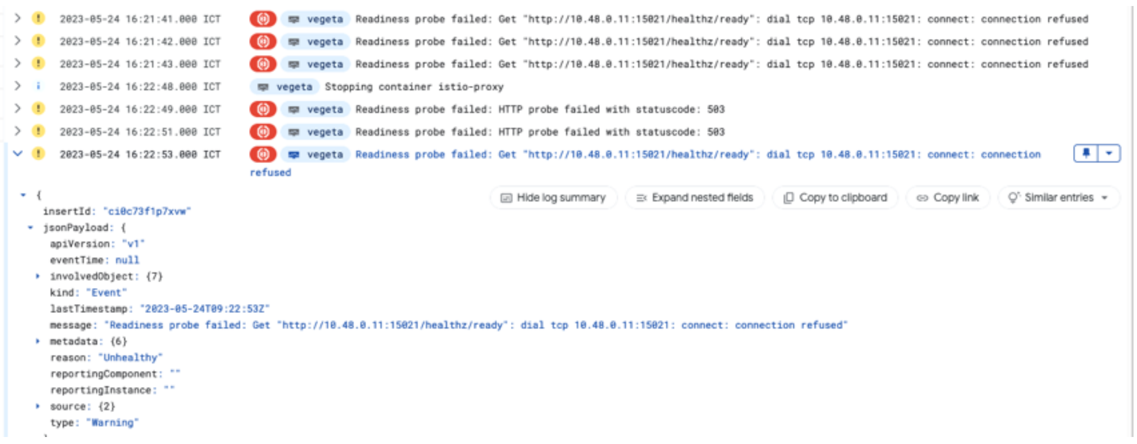
Method	Location	RPS	Success Ratio
MCS with MCI	southeast-asia	10	100.00%
		50	100.00%
		100	100.00%
	australia	10	100.00%
		50	100.00%
		100	100.00%
Istio / ASM	southeast-asia	10	62.67%
		50	71.87%
		100	71.93%
	australia	10	33.33%
		50	34.67%
		100	30.07%

From the single cluster test results, it can be seen that MCS with MCI performs better than Istio / ASM as it is able to handle up to 100 requests per second. This can be explained by the added complexity that Istio brings inside of a Kubernetes architecture, mainly having to inject the Envoy sidecar in each pod. This results in more resources and time required, especially in the case of server recovery during high traffic. In addition, the error response set, as seen in Figure 5.1, confirms this, as all of the errors have a response code of 0, which means that the requests weren’t sent at all, as the server was busy restarting. Server restarts are an expected behavior, as it happens when a server is overloaded, as indicated by an error response code, and are necessary to return a server to a healthy state. Furthermore, the internal state of the pod can be seen in Figure 5.2, where a readiness probe, which is done to check if a pod is healthy, failed as it has not recovered. This results in the connection refused error received by the client.


Activities
Terminal
May 24 16:18
joni@joni: ~/Desktop/skripsi/tests
joni@joni: ~/Desktop/skripsi/tests
joni@joni: ~/Desktop/skripsi/tests

```
691ms, 190.167ms, 393.724ms, 586.046ms
Bytes In      [total, mean]                     5962, 23.85
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           71.60%
Status Codes  [code:count]                      0:71  200:179
Error Set:
Get "http://ta-server-service.sharedvpc:8080/todos": dial tcp 0.0.0.0:0->10.108.
3.79:8080: connect: connection refused
pod "vegeta" deleted
If you don't see a command prompt, try pressing enter.
error: timed out waiting for the condition
Requests      [total, rate, throughput]         500, 100.20, 64.03
Duration      [total, attack, wait]             5.169s, 4.99s, 179.4ms
Latencies     [min, mean, 50, 90, 95, 99, max]  1.796ms, 73.464ms, 4.7ms, 189.43
2ms, 189.904ms, 380.819ms, 568.977ms
Bytes In      [total, mean]                     11843, 23.69
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           66.20%
Status Codes  [code:count]                      0:169  200:331
Error Set:
Get "http://ta-server-service.sharedvpc:8080/todos": dial tcp 0.0.0.0:0->10.108.
3.79:8080: connect: connection refused
pod "vegeta" deleted
joni@joni:~/Desktop/skripsi/tests$ 
```

Figure 5.1: Testing report with HTTP code 0 connection refused response.



```
2023-05-24 16:21:41.000 ICT   vegeta Readiness probe failed: Get "http://10.48.0.11:15021/healthz/ready": dial tcp 10.48.0.11:15021: connect: connection refused
2023-05-24 16:21:42.000 ICT   vegeta Readiness probe failed: Get "http://10.48.0.11:15021/healthz/ready": dial tcp 10.48.0.11:15021: connect: connection refused
2023-05-24 16:21:43.000 ICT   vegeta Readiness probe failed: Get "http://10.48.0.11:15021/healthz/ready": dial tcp 10.48.0.11:15021: connect: connection refused
2023-05-24 16:22:48.000 ICT   vegeta Stopping container istio-proxy
2023-05-24 16:22:49.000 ICT   vegeta Readiness probe failed: HTTP probe failed with statuscode: 503
2023-05-24 16:22:51.000 ICT   vegeta Readiness probe failed: HTTP probe failed with statuscode: 503
2023-05-24 16:22:53.000 ICT   vegeta Readiness probe failed: Get "http://10.48.0.11:15021/healthz/ready": dial tcp 10.48.0.11:15021: connect: connection
                              refused
{
 insertId: "ci0c73f1p7xvw"
 jsonPayload: {
  apiVersion: "v1"
  eventTime: null
  involvedObject: {7}
  kind: "Event"
  lastTimestamp: "2023-05-24T09:22:53Z"
  message: "Readiness probe failed: Get "http://10.48.0.11:15021/healthz/ready": dial tcp 10.48.0.11:15021: connect: connection refused"
  metadata: {6}
  reason: "Unhealthy"
  reportingComponent: ""
  reportingInstance: ""
  source: {2}
  type: "Warning"
```

Figure 5.2: Pod logs showing readiness probe fail.

There is also a discrepancy between the cluster's success ratio on Istio / ASM, where southeast-asia's has at least 2 times higher success ratio than its australia counterpart. As there are virtually no differences between the cluster's methods, this is perhaps an issue on the Google Cloud Platform's part, whether intended or not.

Table 5.2: Multi-Cluster Reliability Test Results

Method	Location	RPS	Success Ratio
MCS with MCI	southeast-asia	10	100.00%
		50	100.00%
		100	100.00%
	australia	10	100.00%
		50	100.00%
		100	100.00%
Istio / ASM	southeast-asia	10	100.00%
		50	100.00%
		100	100.00%
	australia	10	100.00%
		50	100.00%
		100	100.00%

In contrast to single-cluster performance, multi-cluster performance seems to have fixed the reliability issue of Istio / ASM, where it can withstand 100 requests per second without a single failure. At the same time, the MCS with MCI maintains its perfect response ratio.

In addition, the server inside of a MCS with MCI method is able to handle 10 requests per second without restarting and only needs server recovery for 50 requests per second and above. In contrast, the server inside the Istio / ASM method needed to restart even during the lowest request per second configuration. This implies that a single cluster

Istio / ASM may not be the most suitable geo-distributed approach for production-level applications, as it is unable to handle pedestrian-level traffic without inducing downtime on the servers. Therefore, it appears that MCS with MCI is more reliable than Istio / ASM for a single cluster configuration. It is advisable for Istio / ASM adopters to maximize its reliability by deploying services to multiple clusters and enabling the failover feature. Furthermore, there seems to be a problem with the Istio / ASM approach, where the failover feature of locality load balancing won't work with clients without a Service resource. This results in only one cluster responding to all of the requests without utilizing the other clusters.

In conclusion, the geo-distributed Kubernetes architecture is an effective way to increase the reliability of web applications, as it provides a connected network of clusters that serves as a failover when a cluster is experiencing downtime. There are also performance benefits to this, which we will see in section 5.2.

5.2 Performance

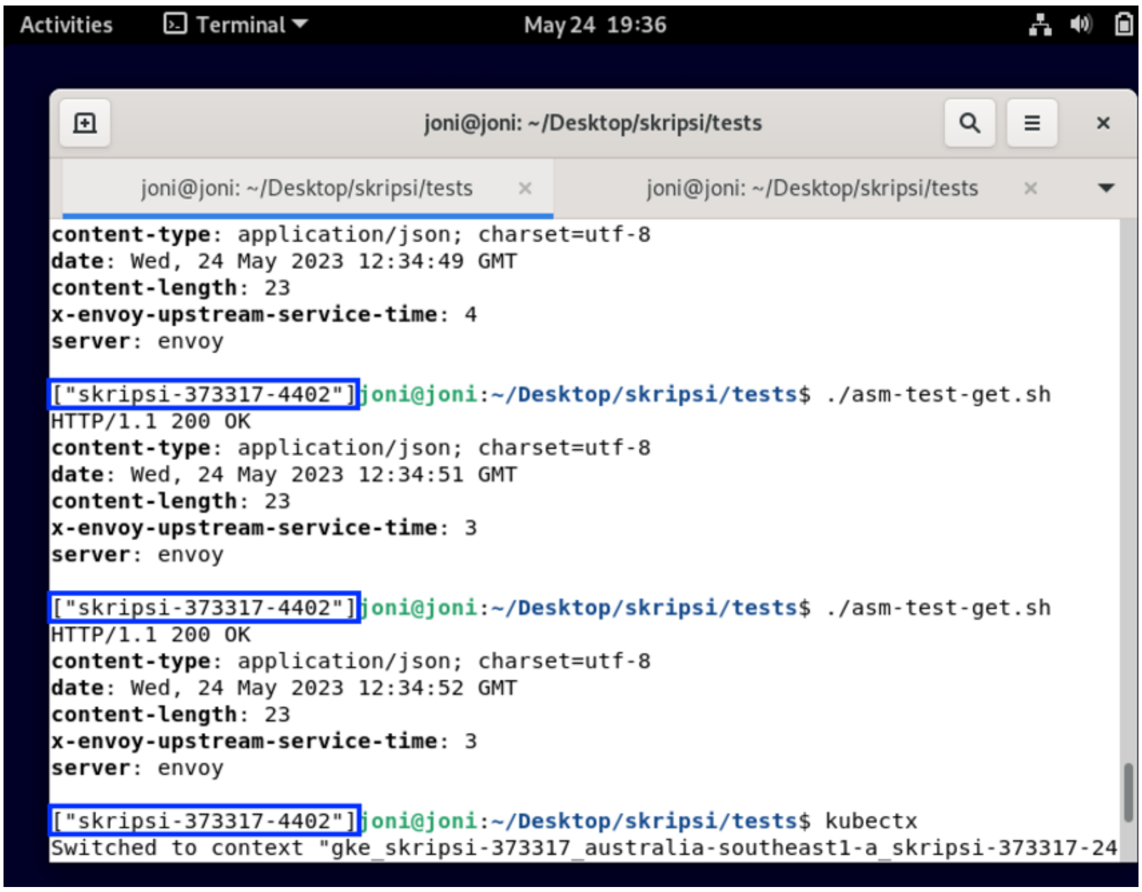
Istio / ASM's results may seem to be better for the single cluster configuration, especially if you compare the minimum latencies of each configuration from each method. However, this is simply the result of requests not being sent, resulting in it having a latency of under 2 milliseconds.

By comparing the minimum, average, and maximum latency of both configurations, as shown in Table 5.3, it can be seen that MCS with MCI has lower latency overall. Another important observation is the latency deviation produced by the Istio / ASM approach, where it may fluctuate to just under 300 milliseconds. This is a 171% to 4385% increase compared to MCS with MCI with a much more stable percentage increase between 136% to 255%.

In addition, both Istio / ASM and MCS with MCI approach correctly route traffic to the nearest cluster, as seen in Figure 5.3 and Figure 5.4, where the server response, which contains the cluster name of the server that handled the request, corresponds to the cluster where the request originated from. This is a contrast to the study done by Andrew (2023), where it was stated that the Google Cloud Load Balancer didn't work correctly. It is to be noted that the MCS with MCI approach may simply be better integrated with Google Cloud Load Balancer compared to the DaemonSet resource used by Andrew (2023).

Table 5.3: Multi-Cluster Performance Test Results

Location	**RPS**	**Latency (ms)**									
		Min		**Median**		**Mean**		**95th**		**Max**	
		MCS	**ASM**	**MCS**	**ASM**	**MCS**	**ASM**	**MCS**	**ASM**	**MCS**	**ASM**
southeast-asia	10	**2.841**	3.497	**3.37**	4.2	**3.688**	4.343	5.593	**4.826**	**7.262**	9.034
southeast-asia	10	**3.085**	97.064	**3.593**	97.476	**4.096**	97.694	**7.588**	98.198	**8.219**	103.985
southeast-asia	10	**3.088**	3.972	**3.47**	4.42	**3.726**	4.591	5.068	**4.908**	**6.589**	12.504
southeast-asia	50	**2.487**	3.656	**3.017**	4.529	**3.143**	4.566	**4.111**	5.047	**6.367**	10.978
southeast-asia	50	**2.82**	3.594	**3.322**	4.464	**3.41**	23.077	**4.033**	97.65	**6.898**	103.524
southeast-asia	50	**3.185**	3.614	**4.052**	4.499	**4.418**	4.528	6.846	**5.402**	**8.992**	10.787
southeast-asia	100	3.836	**3.438**	4.286	**4.26**	**4.37**	11.211	**4.861**	96.766	**8.859**	293.551
southeast-asia	100	**2.434**	3.158	**4.033**	4.134	**3.974**	4.353	4.808	**4.606**	**12.033**	103.519
southeast-asia	100	**2.734**	3.243	**3.471**	3.969	**3.535**	3.998	**4.084**	4.532	14.286	**10.165**
australia	10	**3.56**	3.74	4.492	**3.951**	**4.744**	5.114	6.693	**4.686**	**8.014**	58.857
australia	10	3.608	**3.541**	4.042	**3.9**	4.182	**4.097**	5.603	**4.371**	**7.02**	11.12
australia	10	4.406	**3.509**	4.745	**3.902**	5.242	**4.05**	6.946	**4.554**	**8.819**	9.119
australia	50	3.938	**3.191**	4.8	**3.649**	5.05	**3.692**	7.248	**4.089**	10.318	**8.716**
australia	50	3.282	**2.967**	**3.799**	3.977	**4.018**	4.03	5.355	**4.471**	**7.916**	9.789
australia	50	**3.261**	3.377	**3.794**	3.831	4.029	**3.881**	5.399	**4.253**	**7.937**	10.506
australia	100	3.426	**2.937**	3.958	**3.582**	4.203	**3.707**	5.943	**4.074**	**13.548**	34.545
australia	100	3.268	**3.065**	3.919	**3.676**	4.116	**3.751**	5.767	**4.236**	14.572	**9.319**
australia	100	3.178	**2.789**	3.797	**3.621**	**3.973**	15.747	**5.156**	96.294	**8.512**	291.411


Activities
Terminal
May 24 19:36
joni@joni: ~/Desktop/skripsi/tests
joni@joni: ~/Desktop/skripsi/tests
joni@joni: ~/Desktop/skripsi/tests

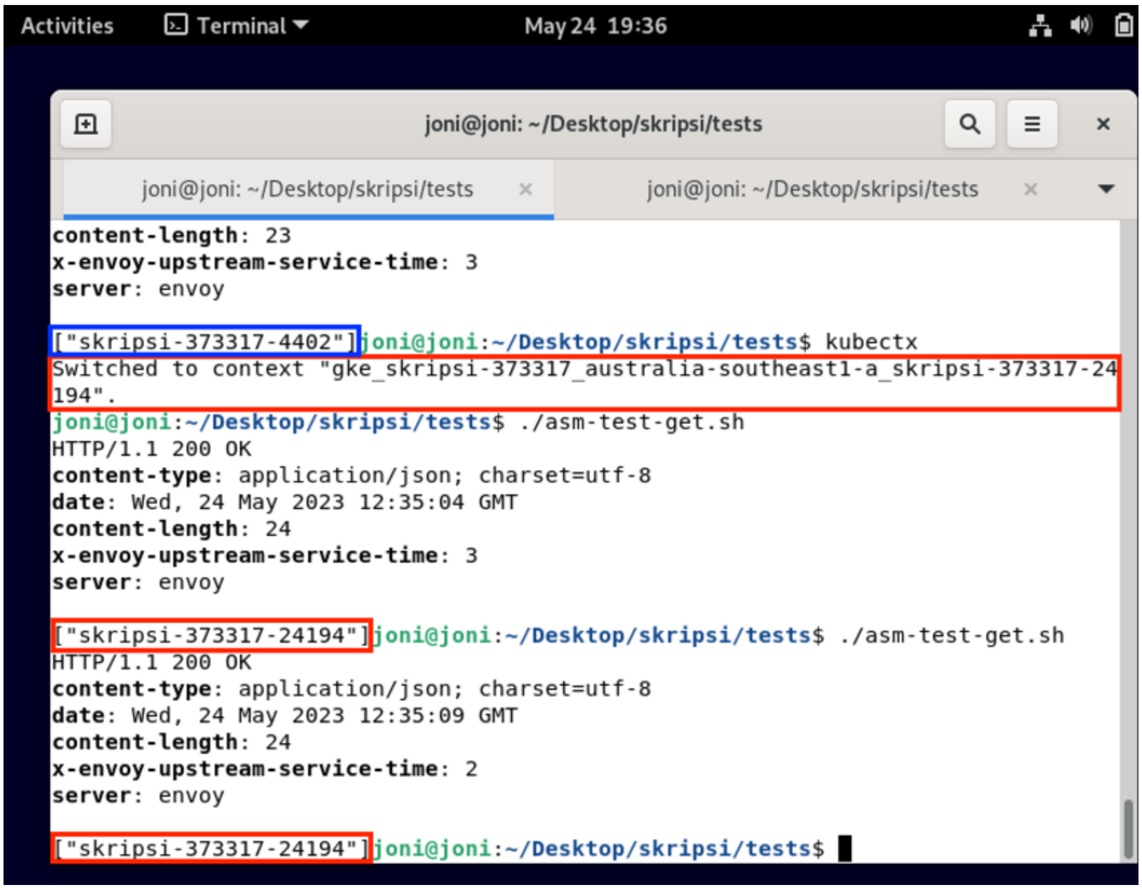
```
content-type: application/json; charset=utf-8
date: Wed, 24 May 2023 12:34:49 GMT
content-length: 23
x-envoy-upstream-service-time: 4
server: envoy

["skripsi-373317-4402"]joni@joni:~/Desktop/skripsi/tests$ ./asm-test-get.sh
HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
date: Wed, 24 May 2023 12:34:51 GMT
content-length: 23
x-envoy-upstream-service-time: 3
server: envoy

["skripsi-373317-4402"]joni@joni:~/Desktop/skripsi/tests$ ./asm-test-get.sh
HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
date: Wed, 24 May 2023 12:34:52 GMT
content-length: 23
x-envoy-upstream-service-time: 3
server: envoy

["skripsi-373317-4402"]joni@joni:~/Desktop/skripsi/tests$ kubectx
Switched to context "gke_skripsi-373317_australia-southeast1-a_skripsi-373317-24
```

Figure 5.3: Geo-aware routing from southeast-asia cluster.


Activities
Terminal
May 24 19:36
joni@joni: ~/Desktop/skripsi/tests
joni@joni: ~/Desktop/skripsi/tests
joni@joni: ~/Desktop/skripsi/tests

```
content-length: 23
x-envoy-upstream-service-time: 3
server: envoy

["skripsi-373317-4402"]joni@joni:~/Desktop/skripsi/tests$ kubectx
Switched to context "gke_skripsi-373317_australia-southeast1-a_skripsi-373317-24
194".
joni@joni:~/Desktop/skripsi/tests$ ./asm-test-get.sh
HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
date: Wed, 24 May 2023 12:35:04 GMT
content-length: 24
x-envoy-upstream-service-time: 3
server: envoy

["skripsi-373317-24194"]joni@joni:~/Desktop/skripsi/tests$ ./asm-test-get.sh
HTTP/1.1 200 OK
content-type: application/json; charset=utf-8
date: Wed, 24 May 2023 12:35:09 GMT
content-length: 24
x-envoy-upstream-service-time: 2
server: envoy

["skripsi-373317-24194"]joni@joni:~/Desktop/skripsi/tests$ 
```

Figure 5.4: Geo-aware routing from australia cluster.

Furthermore, the results from Table 5.3 might indicate that MCS with MCI simply outperforms Istio / ASM, as it has a lower mean and lower maximum latency overall. However, it can be seen from Figure 5.5 and Figure 5.6 that the latency over time of Istio / ASM is comparable to MCI with MCS.

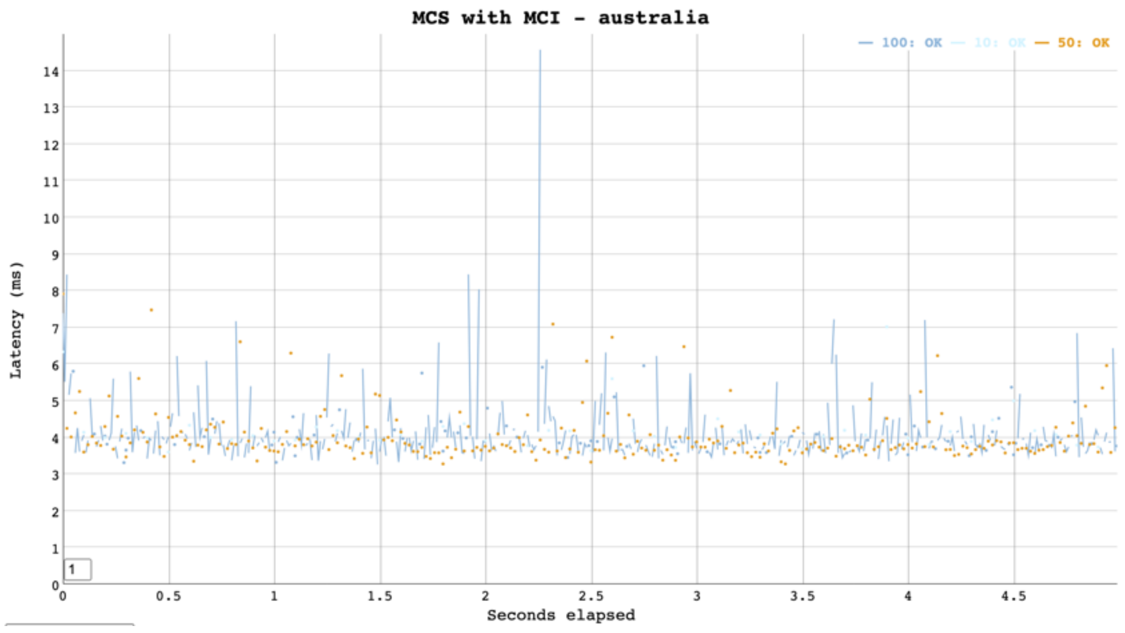


Figure 5.5: Latency over time of MCS with MCI australia 3.

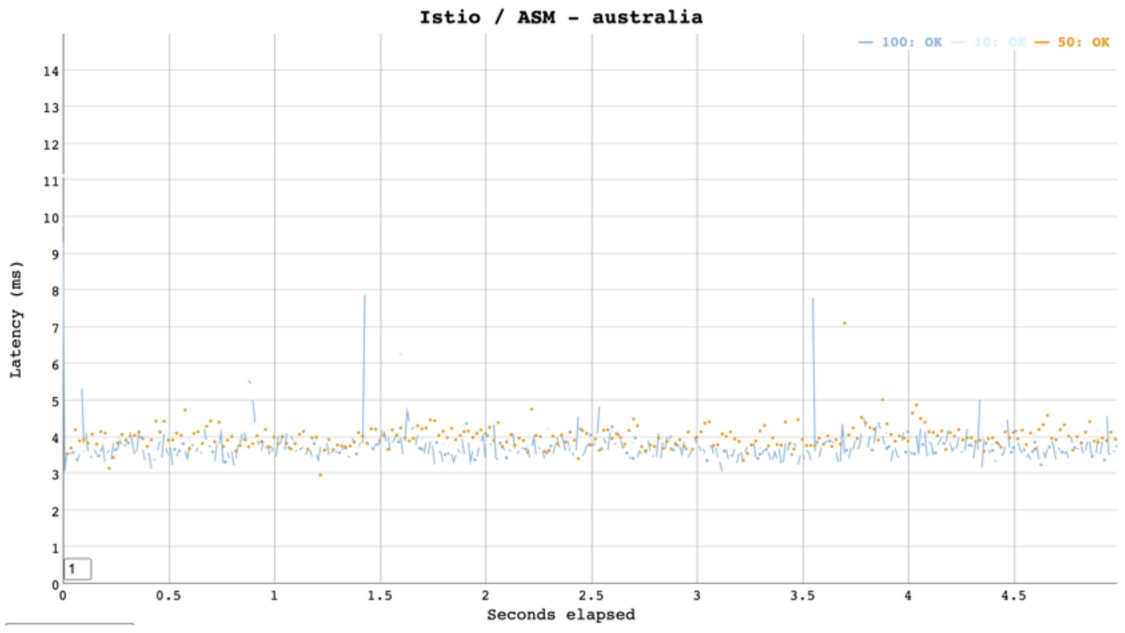


Figure 5.6: Latency over time of Istio / ASM australia 2.

Further testing shows that the reason Istio / ASM method has a worse overall result is the occurrence of performance outliers, as seen in Figure 5.7 and Figure 5.8, the latency over time for the southeast-asia and australia region, respectively. These outliers produced responses with upwards of 300 milliseconds latency and indicate that the Istio / ASM method is prone to under-performance during high-traffic situations. This can be explained by the slow server recovery time shown in Table 5.1, where the failed responses must be redirected to a server with a nonoptimal geographic distance.

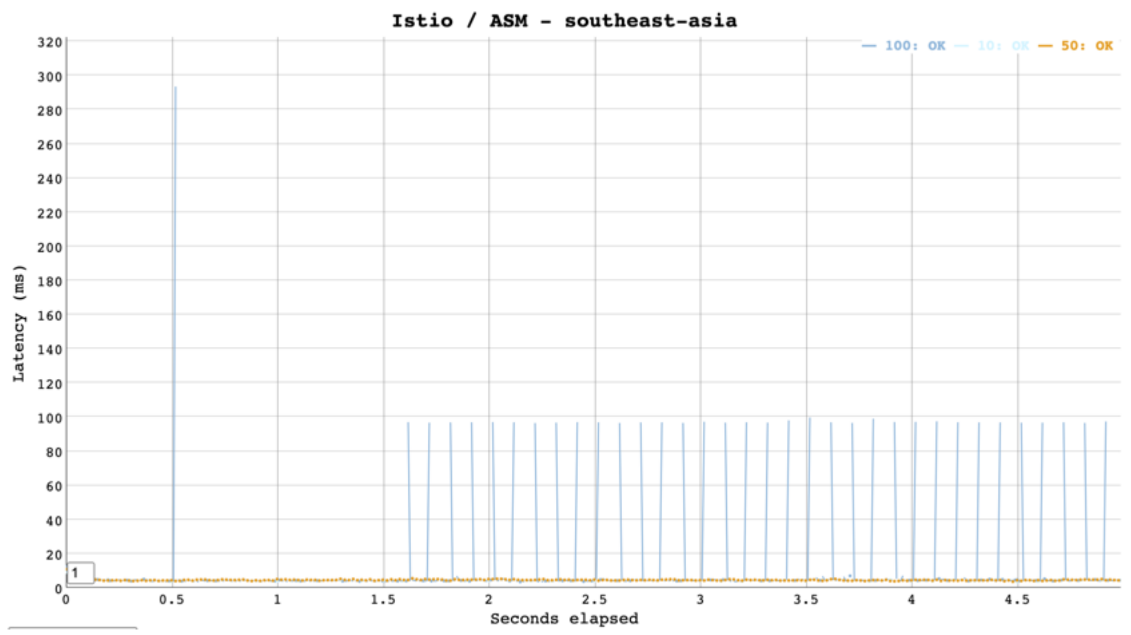


Figure 5.7: Latency over time of Istio / ASM southeast-asia 1.

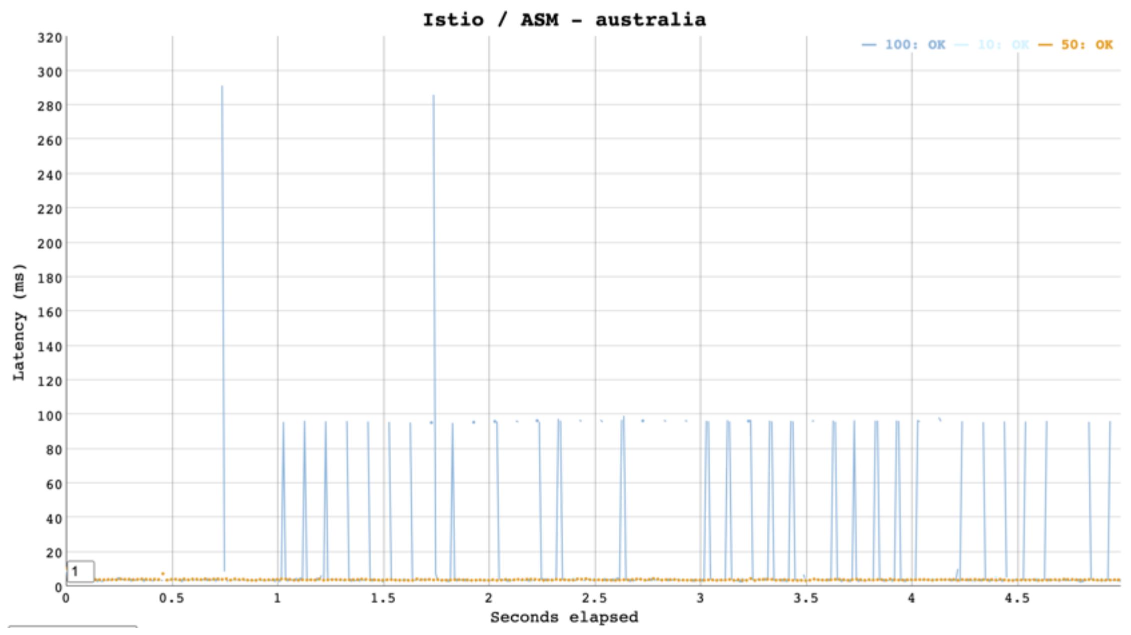


Figure 5.8: Latency over time of Istio / ASM australia 3.

Conversely, test results from MCS with MCI as seen in Figure 5.9 and Figure 5.10 show its consistency as it is able to maintain its performance through multiple experiments. There hasn't been an occurrence where MCS with MCI takes longer than 15 milliseconds to respond. This can be explained by MCS with MCI's excellent single-cluster performance, as its server is able to recover from failure in a short amount of time. Furthermore, a single-cluster MCS with MCI outperforms a multi-cluster Istio / ASM configuration in overall latency with an equivalently flawless success ratio.

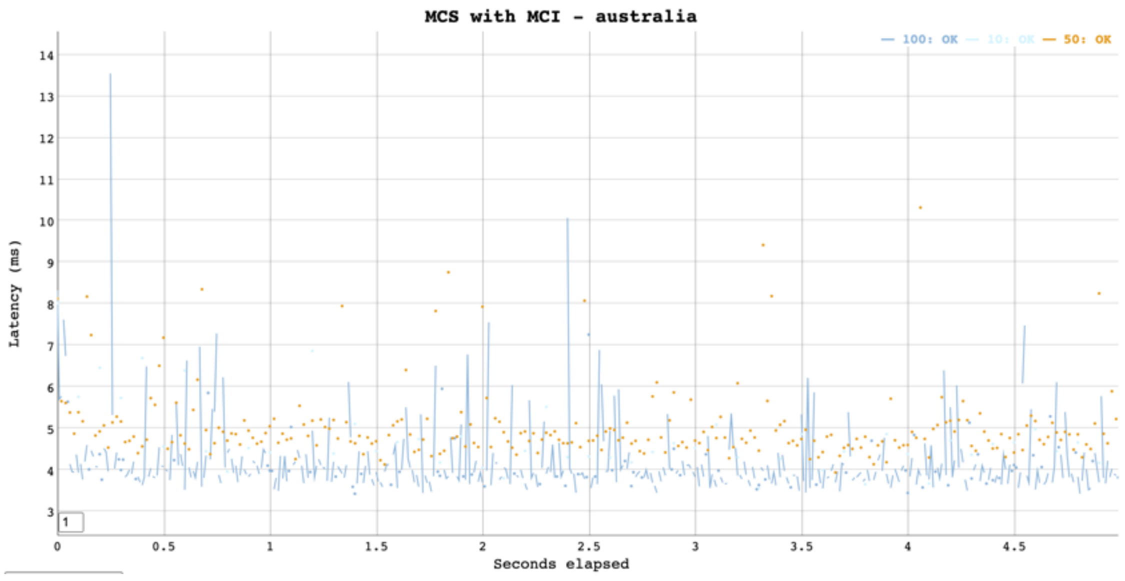


Figure 5.9: Latency over time of MCS with MCI australia 1.

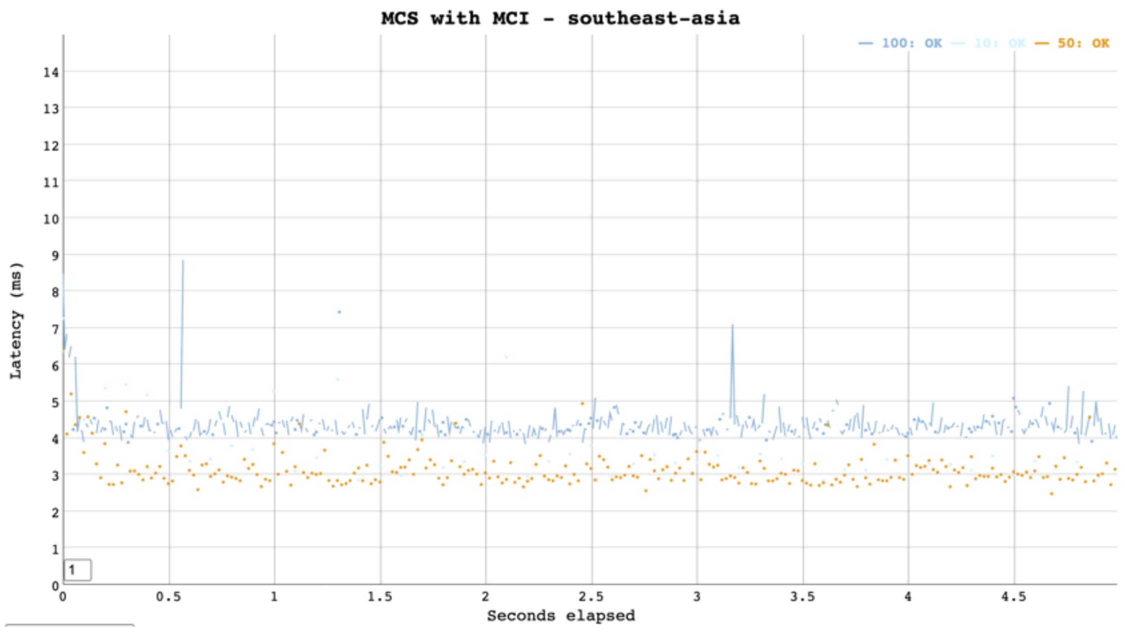


Figure 5.10: Latency over time of MCS with MCI southeast-asia 1.

To further compare these two approaches, we can compare their latency percentile distribution in order to draw a conclusion. From Figure 5.11 and Figure 5.12, it can be seen that Istio / ASM has a lower latency when comparing their 50th and 95th percentile.

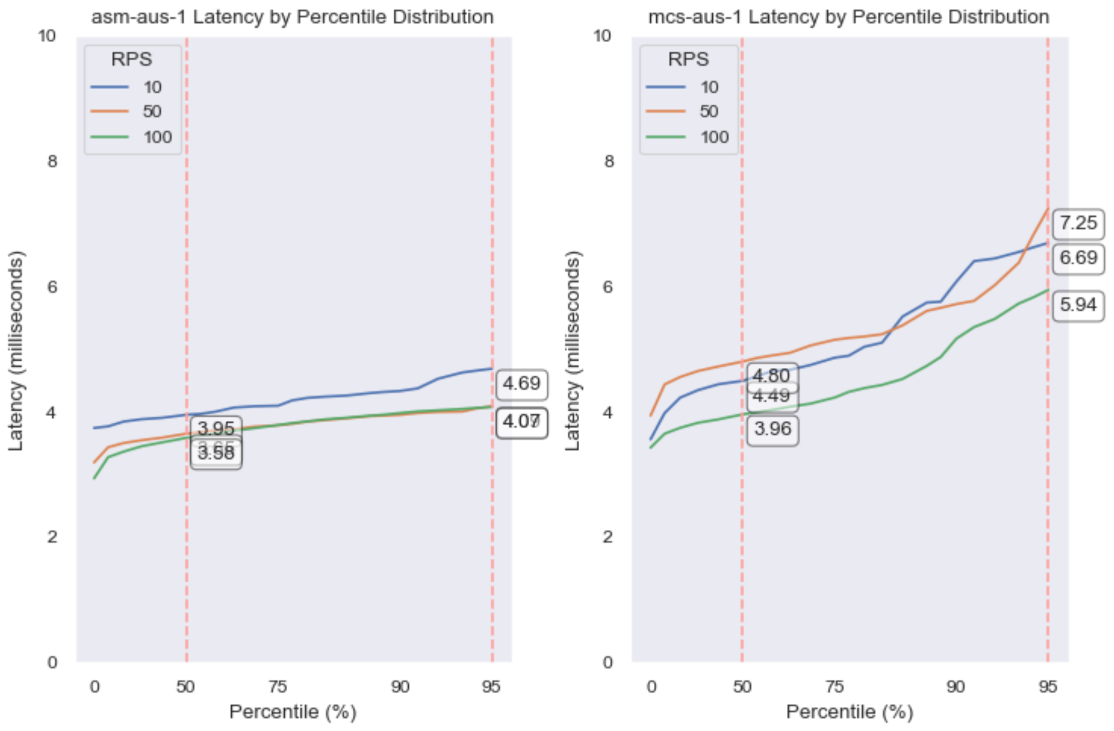


Figure 5.11: Latency percentile distribution of Istio / ASM and MCS with MCI australia 1.

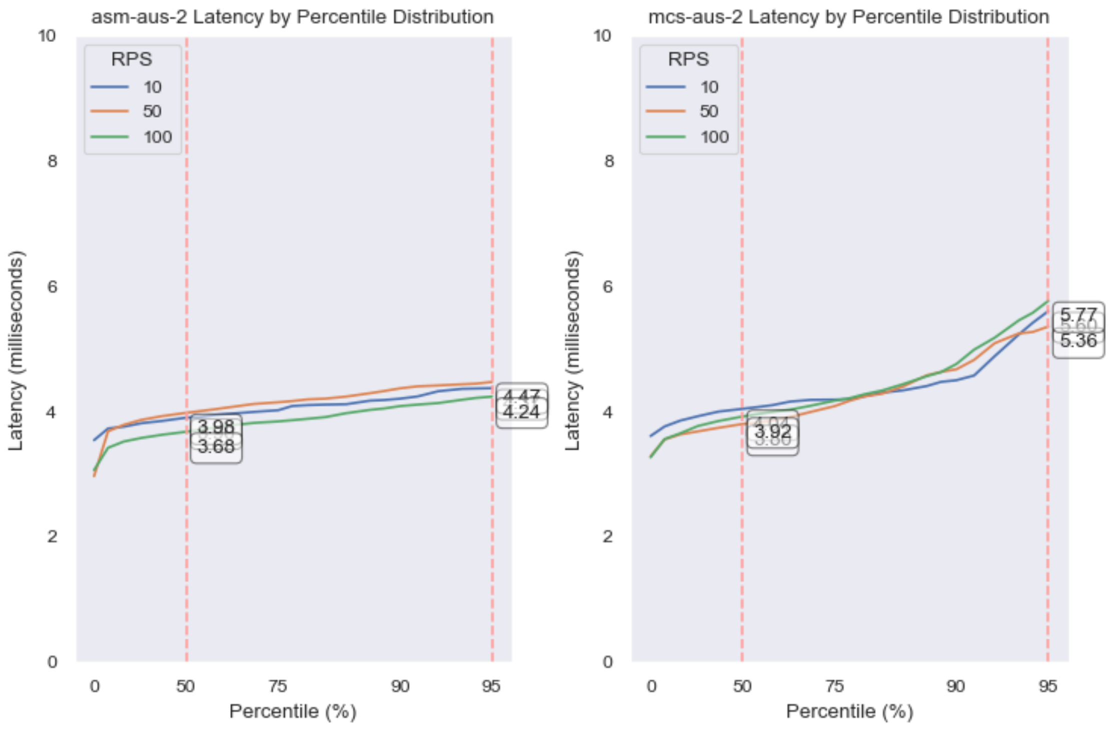


Figure 5.12: Latency percentile distribution of Istio / ASM and MCS with MCI australia 2.

The same result can be seen in the southeast region, where comparing Figure 5.13 shows that Istio / ASM has a lower overall latency. We compare the 50th percentile (median) as it is useful to represent what the majority of the response latency looks like. On the other hand, the lower 95th latency percentile and an overall more stable increase compared to the 50th percentile in Istio / ASM tell us that even the small extreme-end of responses that performed worse than normal is still within an acceptable latency.

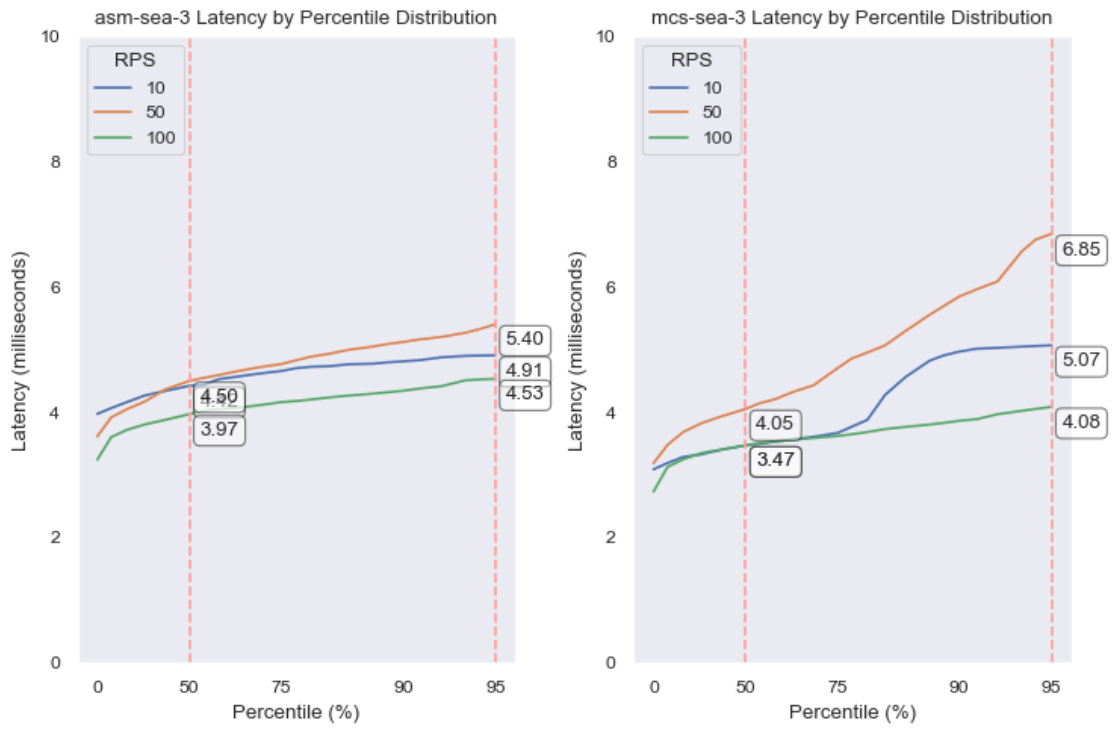


Figure 5.13: Latency percentile distribution of Istio / ASM and MCS with MCI southeast-asia 3.

However, it can't be denied that Istio / ASM suffers from its occasional under-performance, as seen in Figure 5.14, where the jump from the 50th percentile latency to the 95th percentile latency is enormous. This appears to be caused by a combination of Istio / ASM's slow recovery time and slow failover mechanism, which is not a problem inside an MCS with MCI method.

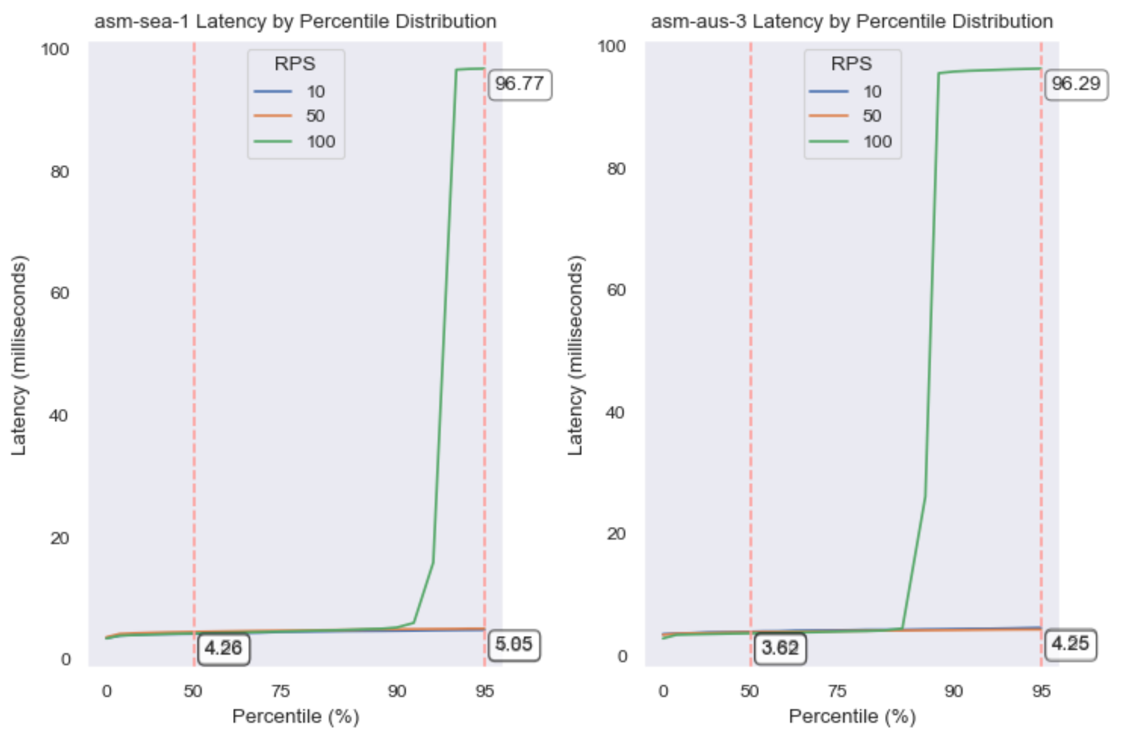


Figure 5.14: Latency percentile distribution of Istio / ASM southeast-asia 1 and Istio / ASM australia 3.

Moreover, comparing the latency's 50th and 90th percentile, as seen in Table 5.3, shows that Istio / ASM has the better overall latency, as it has a lower 95th percentile latency in 12 out of 18 experiments with a 1.4-millisecond difference on average compared to the MCS with MCI approach. The MCS with MCI approach has the advantage in 50th percentile performance, however, as it outperforms Istio / ASM 10 out of 18 times with a 10-millisecond difference on average.

In conclusion, Istio / ASM performs better than MCS with MCI as it is able to handle a wider number of requests with slightly lower latency, despite its outliers. It is advisable to have multiple replicas of each server to avoid long recovery times. Despite this, the difference in median latency is negligible, making MCS with MCI a viable alternative. In addition, an improvement to the failure mechanism can be done by having more clusters to reduce the physical distance between a cluster and its failover destination.

Additionally, it can be seen from Table 5.3 that Istio / ASM has a lower latency in the australia region while the same can be said with the MCS with MCI method on the southeast-asia region. This is possibly the effect of timezone differences, as the tests were done one after another, which results in different actual local time for each of the region.

CHAPTER 6
CONCLUSION

This chapter discusses the conclusions that resulted from the research conducted, which answers . In addition, this chapter also gives suggestions to further this research.

6.1 Conclusion

From this research, there are several conclusions that can be made. First of all, the geo-distributed Kubernetes clusters architecture improves the reliability of worldwide applications. Experiment results show that both MCS with MCI and Istio / ASM methods reliably perform at 100% success rate, making either one a viable choice for application reliabilty. For limited resources, however, MCS with MCI shows better reliability keeping its flawless success rate even with just a single cluster deployed. It is advisable to use the Istio / ASM method with multiple clusters deployed to take advantage if its failover mechanism, as it suffers from slow recovery time, which appears to be the bottleneck for single-cluster reliability.

In addition, Istio / ASM improves application performance. Both MCS with MCI and Istio / ASM methods correctly and efficiently route requests to the cluster with minimal physical distance, which minimizes the overall latency. However, it appears that Istio / ASM is the better choice for application performance as it outperforms MCS with MCI in the 95th percentile of latency on most experiments, which means that its a much more predictable performance, as it is able to deliver lower latency to the vast majority of requests. Despite this, Istio / ASM suffers from outlier performances due to its slow recover and failover time.

Overall, both geo-distributed methods are viable options for worldwide application usage in terms of both reliability and performance. MCS with MCI has its upsides of consistently performing within normal latency ranges while Istio / ASM has a slightly better performance for the majority of the requests but suffers from latency spikes in small cases.

6.2 Suggestions

From this research, there are several suggestions that can be done in further studies, one of which, is to study the effects of increasing the replicas of each server. Since Istio / ASM adds complexity to a Kubernetes architecture, it would be useful to research about the effects of horizontal scaling on its overall performance. In addition, further research should be done on other cloud platforms as well, as Google Cloud Platform is only one of many other available cloud service providers such as AWS and Azure. Comparisons between different cloud service providers can be useful to share knowledge about prices between providers as well as overall ease of use for the average developer. In addition, a wider variety of cloud services can be used as different client origins to study the effects of intra-cloud and inter-cloud web performance. Furthermore, future researches should also explore different and/or more regions to measure reliability and performance on a wider geographical scale.

In addition, it would be beneficial to do research on other open-sourced service mesh such as Linkerd and Consul to help understand the benefits and drawbacks of each service mesh implementation and its various features. Furthermore, studies on geo-distributed databases should be explored, as it is one of the most important part of a web application. It is, however, a complex topic that introduces geo-distributed partitioning and replica replacement.

REFERENCES

Andrew. (2023, 1). *Analisis performa web service dalam geo-distributed kubernetes cluster* (Tech. Rep.).

Bhalerao, D. M., & Ingle, D. (2019, 1). Novel Technique for Atomic Web Services Reliability for Service Oriented Architecture. *Social Science Research Network*. Retrieved from `https://doi.org/10.2139/ssrn.3372226` doi: 10.2139/ssrn .3372226

Cisco Annual Internet Report (2018–2023) White Paper (Tech. Rep.). (2022, 1). Retrieved from `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`

Google. (2023a, 3). *About Anthos Service Mesh.* Retrieved from `https://cloud.google.com/service-mesh/docs/overview`

Google. (2023b, 3). *Multi Cluster Ingress.* Retrieved from `https://cloud.google.com/kubernetes-engine/docs/concepts/multi-cluster-ingress`

Jeffery, A., Howard, H. C., & Mortier, R. (2021, 4). Rearchitecting Kubernetes for the Edge.. Retrieved from `https://doi.org/10.1145/3434770.3459730` doi: 10.1145/3434770.3459730

Rossi, F. A., Cardellini, V., Lo Presti, F., & Nardelli, M. (2020, 6). Geo-distributed efficient deployment of containers with Kubernetes. *Computer Communications*, *159*, 161–174. Retrieved from `https://doi.org/10.1016/j.comcom.2020.04.061` doi: 10.1016/j.comcom.2020.04.061

Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., ... Mattis, P. L. (2020, 5). CockroachDB: The Resilient Geo-Distributed SQL Database.. Retrieved from `https://doi.org/10.1145/3318464.3386134` doi: 10.1145/3318464.3386134

Xie, X., & Govardhan, S. (2020, 5). A Service Mesh-Based Load Balancing and Task Scheduling System for Deep Learning Applications. *Cluster Computing and the Grid*. doi: 10.1109/ccgrid49817.2020.00009

Yu, J. (2019, 3). Exploration on Web Testing of Website. *Journal of physics*. Retrieved from https://doi.org/10.1088/1742-6596/1176/2/022042 doi: 10.1088/1742-6596/1176/2/022042

LAMPIRAN