**A NETWORK SCIENCE BASED APPROACH FOR OPTIMAL MICROSERVICE GOVERNANCE**

Saranga S.A.G

IT17016230

B.Sc. (Hons) Degree in Information Technology

Specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

September 2020

**A NETWORK SCIENCE BASED APPROACH FOR OPTIMAL MICROSERVICE GOVERNANCE**

Saranga S.AG.

IT17016230

Dissertation submitted in partial fulfillment of the requirements for the Bachelor of

Science specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

September 2020

**Declaration**

I declare that this is my own work and this proposal does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any other university or institute of higher learning and to the best of our knowledge and belief, it does not contain any material previously published or written by another person except where the acknowledgment is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the nonexclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

**Signature:**

|  |  |  |
| --- | --- | --- |
| **IT17016230** | **Saranga S.A.G** |  |

The above candidates are carrying out research for the undergraduate Dissertation under my supervision.

**Signature of the Supervisor: ……………………………………**

**Date: …………………………….**

**Abstract**

With the popularity of microservices and containerization, usage of the platforms which orchestrate the container eco-system like Kubernetes was increased with a significant amount within a brief period of time. Even though Kubernetes claims to make a complex microservice system more simplify, it is not the case when it comes to monitoring and maintaining a system with more than fifty different microservices.

However, Kubernetes take care of the deployment of the microservice, but Kubernetes only consider the size and the hardware requirement of the microservice. The word Dependency is often paired with microservices as microservices need to communicate with one another, making a dependency between them. Nevertheless, dependence has a considerable amount of impact on the performance of the overall application.

This research thereby aims to address the problems of monitoring and maintaining a Kubernetes cluster and consider the dependency between microservices to create an optimal deployment plan.

**Keywords:** Network Mesh, Microservice Mesh, Kubernetes, Microservice

**Acknowledgment**

I would like to thank my supervisor, Dr. Dharshana Kasthurirathna, for the guidance and motivation provided to make this research a success. I would also like to thank the Department of Software Engineering of the Sri Lanka Institute of Information Technology as well as the CDAP lecturers and staff for providing the opportunity to conduct this research.

**Table of Contents**

[**Declaration 1**](#_Toc51617684)

[**Abstract 2**](#_Toc51617685)

[**Acknowledgment 3**](#_Toc51617686)

[**Table of Contents 4**](#_Toc51617687)

[**List of Figures 6**](#_Toc51617688)

[**List of Tables 7**](#_Toc51617689)

[**List of Abbreviations 8**](#_Toc51617690)

[**1. INTRODUCTION 9**](#_Toc51617691)

[**1.1 Background and Literature 16**](#_Toc51617692)

[**1.2 Research Gap 18**](#_Toc51617693)

[**2. RESEARCH PROBLEM 19**](#_Toc51617694)

[**3. OBJECTIVES 21**](#_Toc51617695)

[**3.1 Main Objective 21**](#_Toc51617696)

[**3.2 Specific Objectives 21**](#_Toc51617697)

[**4. METHODOLOGY 22**](#_Toc51617698)

[**4.1 Requirement Gathering 22**](#_Toc51617699)

[**4.1.1 Past research analysis 23**](#_Toc51617700)

[**4.1.2 Identifying existing systems 23**](#_Toc51617701)

[**4.2 Feasibility Study 23**](#_Toc51617702)

[**4.2.1 Technical Feasibility 23**](#_Toc51617703)

[**4.2.2 Schedule Feasibility 25**](#_Toc51617704)

[**4.2.3 Economic Feasibility 25**](#_Toc51617705)

[**4.3 Requirement Analysis 26**](#_Toc51617706)

[**4.4 Software Solution 26**](#_Toc51617707)

[**4.4.1 Dependency Network 27**](#_Toc51617708)

[**4.5 System Development and Implementation 28**](#_Toc51617709)

[**4.5.1 Building the Dependency Network 30**](#_Toc51617710)

[**4.5.2 Setting up the Kubernetes Cluster 36**](#_Toc51617711)

[**4.5.3 Setting up and Initializing Istio 38**](#_Toc51617712)

[**4.5.4 Setting up Kiali and Prometheus 39**](#_Toc51617713)

[**4.5.5 Implementation of the Middle Tier Server 41**](#_Toc51617714)

[**4.5.6 Deployment of Middle Tier Server 41**](#_Toc51617715)

[**4.6 Project Requirements 42**](#_Toc51617716)

[**4.6.1 Functional requirements 42**](#_Toc51617717)

[**4.6.2 Non-Functional Requirements 42**](#_Toc51617718)

[**4.7 Commercialization 43**](#_Toc51617719)

[**5. RESULTS AND DISCUSSION 44**](#_Toc51617720)

[**5.1 Testing 44**](#_Toc51617721)

[**5.1.1 Deployment of a set of sample microservices 44**](#_Toc51617722)

[**5.1.2 Deployment of The Developed Node Server 44**](#_Toc51617723)

[**5.1.3 Data Collection 45**](#_Toc51617724)

[**5.2 Test Results – Final Solution 46**](#_Toc51617725)

[**5.3 Research Findings 47**](#_Toc51617726)

[**5.4 Discussion 49**](#_Toc51617727)

[**6. CONCLUSION 50**](#_Toc51617728)

[**References 50**](#_Toc51617729)

**List of Figures**

[Figure 1.1: Monolithic Architecture 9](#_Toc51617743)

[Figure 1.2: Server-Client Architecture 10](#_Toc51617744)

[Figure 1.3: Microservice architecture sample 11](#_Toc51617745)

[Figure 1.4: Virtualization vs. Containerization 13](#_Toc51617746)

[Figure 1.5: Containerization with Docker 14](#_Toc51617747)

[Figure 1.6: Kubernetes Logo 15](#_Toc51617748)

[Figure 1.7: Kubernetes architecture 18](https://mysliit.sharepoint.com/sites/2020_021_ANetworkScienceBasedApproachforoptimalMicroserviceG/Shared%20Documents/General/thesis/IT17016230_2020-021_Final_Report_new.docx#_Toc51617749)

[Figure 2.1: Current challenges in Kubernetes 20](#_Toc51617750)

[Figure 4.1: Solution Architecture 27](#_Toc51617751)

[Figure 4.2: Developed system architecture 28](#_Toc51617752)

[Figure 4.3: Detailed system architecture 29](#_Toc51617753)

[Figure 4.4: Istio Architecture for a single cluster 31](#_Toc51617754)

[Figure 4.5: Istio Architecture for a multi-cluster 32](#_Toc51617755)

[Figure 4.6: Prometheus Architecture 33](#_Toc51617756)

[Figure 4.7: Sample Grafana Dashboard 34](#_Toc51617757)

[Figure 4.8: Sample Kiali Dashboard 34](#_Toc51617758)

[Figure 4.9: Sample Jaeger Dashboard 35](#_Toc51617759)

[Figure 4.10: Creation of a cluster in Azure Kubernetes Services 37](#_Toc51617760)

[Figure 4.11: Selection of the number of nodes in the cluster 37](#_Toc51617761)

[Figure 4.12: Installation of Istio 38](#_Toc51617762)

[Figure 4.13: Enabling auto-injection for the side cart 38](#_Toc51617763)

[Figure 4.14: Deployment of the sample application 39](#_Toc51617764)

[Figure 4.15: Deployment of Ingress gateway 39](#_Toc51617765)

[Figure 5.1: Generated quantified dependency CSV 46](#_Toc51617766)

[Figure 5.2: Generated CPU load CSV 46](#_Toc51617767)

[Figure 5.3: Generated node-edge graph based on the dependency 47](#_Toc51617768)

[Figure 5.4: Variation of dependency against the time 48](#_Toc51617769)

[Figure 5.5: Fluctuation of the CPU usage against time 48](#_Toc51617770)

**List of Tables**

[Table 4.1: Tools and Languages Used 27](#_Toc51574608)

**List of Abbreviations**

|  |  |
| --- | --- |
| Abbreviation | Description |
| AKS | Azure Kubernetes Services |
| API | Application Programming Interface |
| AWS | Amazon Web Service |
| CSV | Comma Separated Value |
| DB | Database |
| HPA | Horizontal Pod Autoscaler |
| HTTP | HyperText Transfer Protocol |
| IP | Internet Protocol |
| JS | JavaScript |
| K8 | Kubernetes |
| OS | Operating System |
| REST | REpresentational State Transfer |
| SQL | Structured Query Language |
| TCP | Transform Control Protocol |
| VM | Virtual Machine |
| WSL | Windows Subsystem for Linux |
| YAML | Yet Another Markup Language |

# INTRODUCTION

With the advancement of computer science and software engineering, the complexity of computer programs was too much to be handled within a monolithic architecture [1].

A screenshot of a cell phone

Description automatically generated

Figure 1.1: Monolithic Architecture

Therefore, the software programs were moved to client-server based architecture, which divides the application into two significant parts [2].

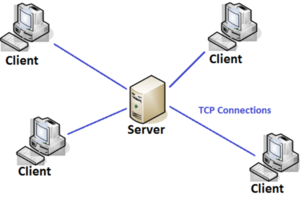


Figure 1.2: Server-Client Architecture

The communication between the two regions was maintained with the TCP/IP protocol [2]. TCP/IP is a connection-oriented protocol, and the connection is established until both parties have finished exchanging messages [3]. The client-server architecture brought advantages such as making it easier to protect and access controlling data and the ability to deploy and maintain the application in different environments as the network handles the connection. However, having too many clients requesting data concurrently from the server is one of the primary disadvantages of the client-server model. Alternatively, the complexity of the client and server as individual components was rising, making the maintaining and debugging difficult again.

In 2005, Dr. Peter Rogers mentioned the term "micro web services" for the first time during a conference on Cloud Computing [4]. The term "microservices" was first introduced in 2011 and was considered as a specialized implementation of Service-Oriented Architecture (SOA), coined to denote the common architectural approach of decomposing applications into smaller self-contained, loosely coupled services.

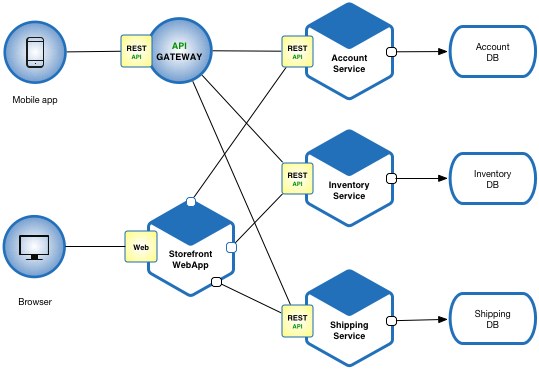


Figure 1.3: Microservice architecture sample

The microservice architectural style was later widely adopted in place of the traditional monolithic architecture by many leading companies such as Amazon, Netflix, LinkedIn, and SoundCloud due to the capability to develop loosely coupled services possessing the ability to be independently deployed, versioned, and scaled while ensuring in benefits such as faster delivery, more excellent performance, and greater autonomy.

The basic idea behind the microservice architecture is, divide, and conquer, which means to divide a more central problem to manageable and achievable smaller problems and implement them separately. The microservice architecture, which is loosely coupled, allows developers to develop individual microservices with any programming language, framework, and any platform that suits the most. This is possible as the HTTP protocol handles the communication between microservices and HTTP being both synchronous and asynchronous protocol. If there are two microservices called 'A,' 'B.' The microservice 'A' sends a request, and it expects and waits for a response from 'B.' The synchronousness of the application is decided by the microservice' A.' It can either wait for a response or continue the execution. That scenario is entirely independent of the execution of 'B.'

The facts mentioned above allows microservices to be

* Loosely Coupled
* Maintainable and Testable
* Independently deployable

Even though microservices solves many problems, microservices also have several disadvantages.

* The complexity of the communication of microservices
* The number of resources needs to allocate
* The difficulty of global testing
* Challenges in deployment because of varied environments and operating systems

To overcome the above difficulties, the technology "Containerization" came to play. The idea of containers was started way back in 1979 with UNIX Chroot [5]. It is a UNIX operating-system system call for transforming the root directory of a process, and it is children to a new position in the filesystem, which is only visible to a given process. Most of the time, containers compared to virtual machines since they both allow the software to execute in a contained environment. However, they are different in the structure and the process of execution.

VMs are an abstraction of the hardware layer, meaning that each VM simulates a physical machine that can run the software. VM technology can use one physical server to run the equivalent of many servers, each of which is called a VM. So, while multiple VMs run on one physical machine, each VM has its copy of an Operating System, applications and their related files, libraries, and dependencies.

Containers are an abstraction of the application layer, meaning that each container simulates a different software application. Though each container runs isolated processes, multiple containers share a universal Operating System.

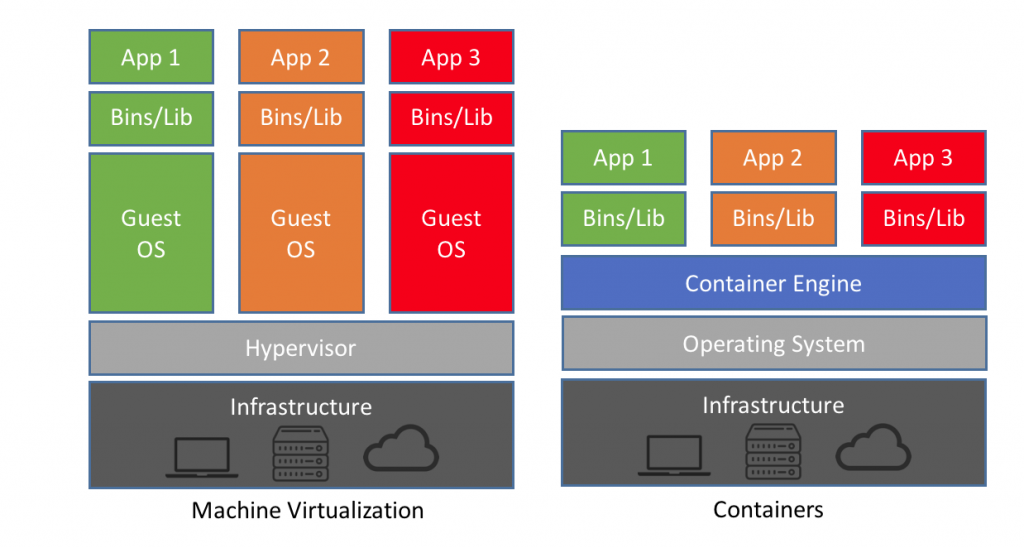


Figure 1.4: Virtualization vs. Containerization

With the popularity of containerization, there are several platforms to provide the services of containerization, including Docker, VirtualBox, Vagrant, and Wox [6]. All these platforms have their ups and downs, but Docker holds the position of the leading platform when it comes to the containerization.

There are five main advantages of Docker, which are speed, portability, scalability, rapid delivery, and density [7]. Furthermore, since all required dependencies, libraries, and all other components needed for the microservice packaged up in the container itself, the software is isolated from its environment, ensuring the performance and the functionality of the software remain the same regardless of the underlying platform [8].

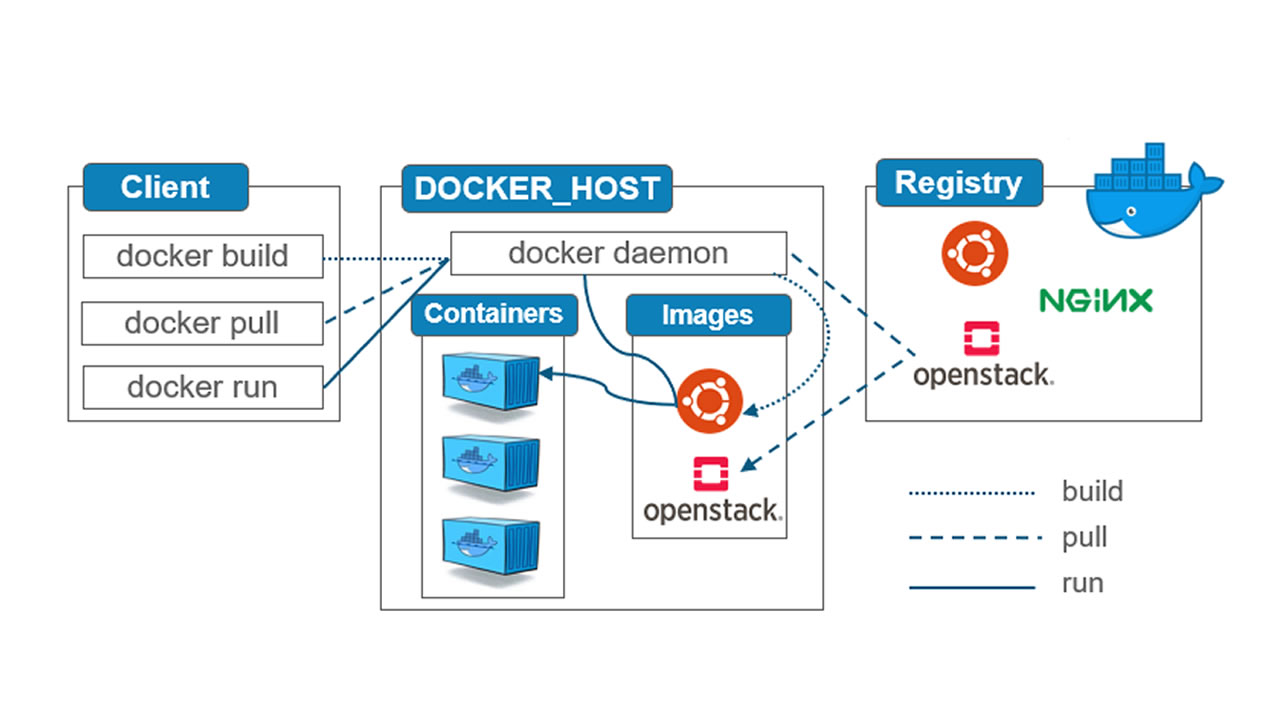


Figure 1.5: Containerization with Docker

Nevertheless, with the use of docker containers in production, practical problems become visible such as which container runs on which node, how to improve the performances, how to add more replications to add redundancy, and how to communicate with each node. Kubernetes [9] is designed to overcome all the above-mentioned practical problems and many more.

Kubernetes is an open-source API that can control the deployments and connection of multiple containers [10]. In the long term, Kubernetes is an open-source container orchestration platform that automates the manual process of developing, deploying, and maintaining containerized microservice applications.

The introduction of Kubernetes brought about a revolutionary approach to the deployment of microservice applications. Kubernetes provides an effective solution to the orchestration of containerized microservices. As a result, in conjunction with the vast array of services and features such as service discovery and load balancing, storage orchestration, and self-healing mechanisms offered through the Kubernetes framework, many organizations throughout the world are increasingly adopting Kubernetes as an effective solution to govern their microservice deployments, thereby establishing Kubernetes among the foremost microservice deployment tools currently available.

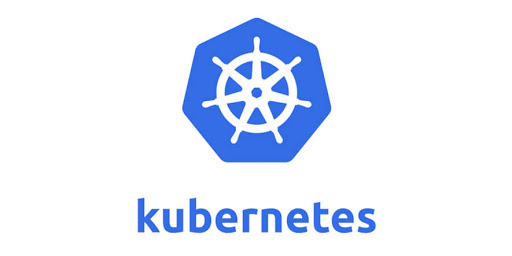


Figure 1.6: Kubernetes Logo

The proceeding sections in this document discuss critical aspects of the development of the solution mentioned above. In this regard, this document provides a thorough analysis of critical topics which include the background, research gap as well as the research problem addressed by the developed solution, along well an in-depth discussion and analysis of results obtained. Lastly, the document also discusses future work and suggested improvements to the developed solution, along with an overall conclusion of the research conducted.

## Background and Literature

As previously stated, Kubernetes provides all in one solution for making a complicated application more manageable, maintainable, and scalable. By 2020, the projected market of containerized and orchestrated application is 4.3 billion dollars [11]. 63% of enterprises are adopting Microservices as the primary architecture. 58% of companies are using Kubernetes in production, while 32% of companies evaluate Kubernetes for future use.

When it comes to the history of Kubernetes, it was started as a large scale cluster management at Google with the name "Borg" [12] in 2003. It initially was a small project with the collaboration of 3-4 people

In 2013, the system "Borg" was converted to project "Omega," a flexible, scalable scheduler for large computer clusters. Google introduced Kubernetes in 2015 as an open-source project.

Kubernetes introduces several new vocabularies, and it is necessary to understand that terminology to get a clear understanding of Kubernetes and how it works.

* Pods

In simplest terms, a pod can be identified as a group of containers. Moreover, a pod is known as the smallest unit of a Kubernetes cluster administration. Computational resources such as memory and storage are shared among the containers within a single pod, which allows the containers inside a single pod to behave as a single application. However, the most common practice is to have a single container per pod.

* Deployments

Deployment decides and defines the scale of the application, which is running inside the Kubernetes cluster. It describes the number of identical pods to be deployed in the cluster to increase the reliability of the application. Kubernetes takes care of the pod health and takes necessary action to keep the application up and running.

* Services

A service is an interpretation of a set of pods. As the pods can be replaced, the IP addresses of the pods can be changed. A service exposes a single endpoint for a particular service, which can be consumed by other microservices, and it is mapped with pods. Therefore, for the outside network, there is no visible change.

* Node

In Kubernetes, there are two main types of nodes.

1. Master Node
2. Worker (slave) nodes

A node controls and manages pods. It acts as the host machine, which can either be physical or virtual. There can be multiple pods in one node, and the node should accommodate all the resource requirements of the pod.

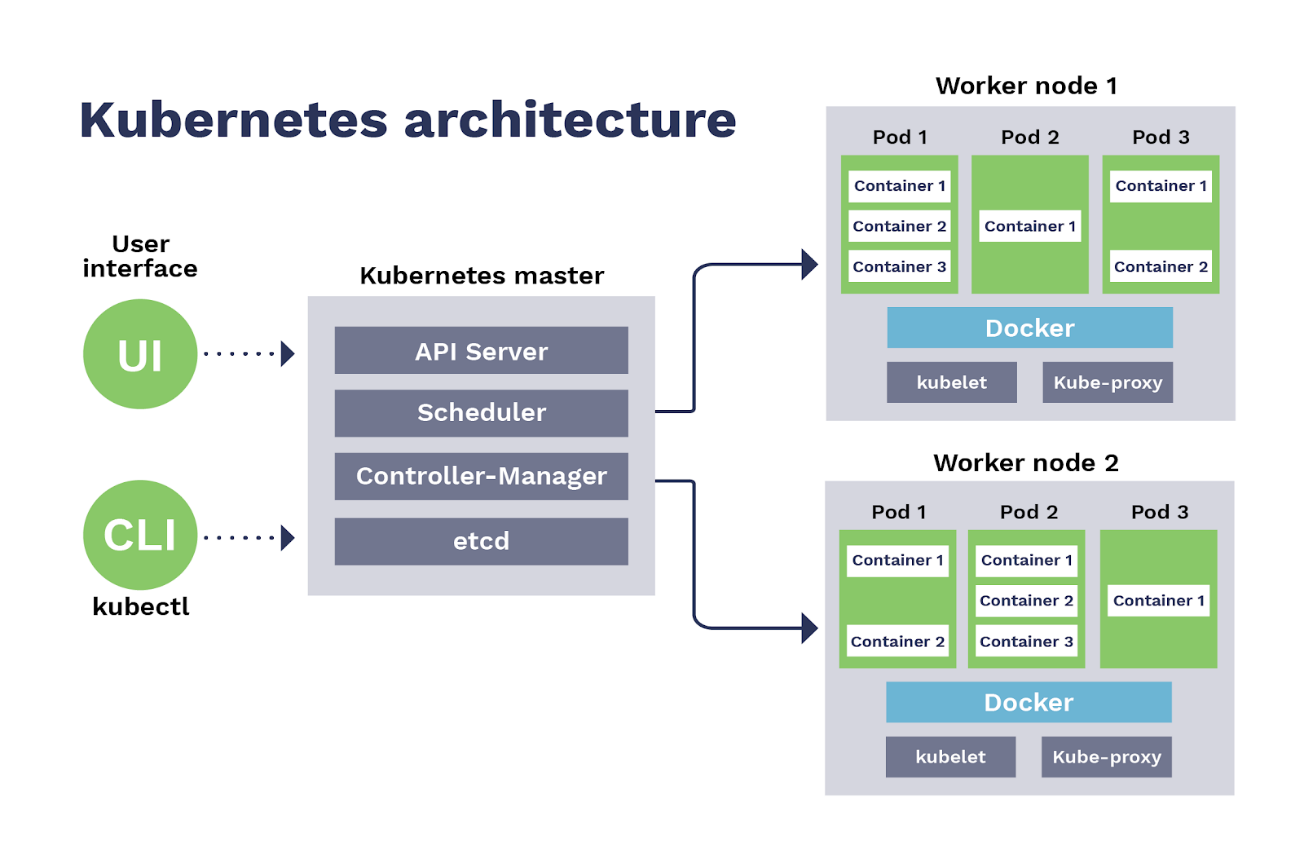


Figure .: Kubernetes architecture

## Research Gap

The primary research gap that this research aims at fulfilling is the gap that exists in current methodologies when considering the dependency level between microservices, particularly concerning Kubernetes. A thorough analysis of published research papers throughout the years has managed to highlight very low of the issues that are present in current microservice deployment methodologies and thereby enabled in the identification of the research gap, as mentioned in the previous statement.

The initial inspiration for the identification of this research problem and the objectives of this research can found in [13]. This publication clearly describes some of the critical challenges faced in the deployment of microservices and the need for Application Performances Monitoring tools, especially those deployed in containers, to include additional measures to monitor microservices such that they could use as input for resilience mechanisms and creation of auto-scaling policies.

Even though there are numerous researches about Kubernetes and service mesh, there were no publications considering the dependency between microservices as a whole. The available publications go on to describing the auto-deployment facilities of Kubernetes [14]; however, they do not describe how that can affect the network latency between microservice are deployed automatically in the available nodes.

However, when it comes to the deployment of the optimal deployment strategy, generated at the end of the solution, support for auto-deployment is crucial. Currently, the developers use the Yaml language [15] to write configurations as code to do the necessary deployments. Nevertheless, when it comes to handling the deployments of a vast number of microservices, it can be a headache for the development team.

According to publication [16, 17, 18], there is automated deployment support in Kubernetes, as depicted in figure 1.2.1. However, still, there is no consideration given towards the dependency level, resilience analysis, and load prediction, which are the key pillars of this research.

# RESEARCH PROBLEM

Even though Kubernetes is quite successful and widely used nowadays, it is not without its unique challenges, which are mostly true concerning some of the current processes in the governance of microservices through Kubernetes, particularly considering processes involving in monitoring, resource utilization, and performance optimization.

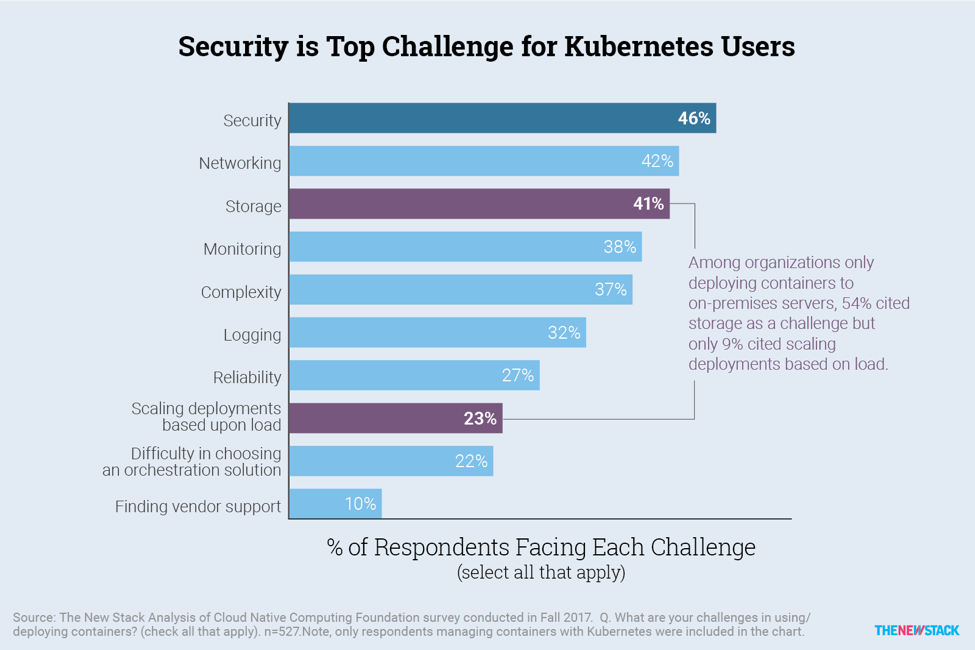


Figure 2.1: Current challenges in Kubernetes

Some of the known problems to Kubernetes [19]:

* Complexity
  + Kubernetes itself is quite complex to install, configure and manage
  + With the increased number of nodes, pods, and services, it is more challenging to get a holistic idea about the microservice architecture.
  + Monitoring the network traffic and the resource utilization for computational purposes have to be done by installing several metric servers separately.
* Unknowingly over or under-allocating the available resources with negative results
  + Pods might not start at all
  + Pods might not be ready to cater to a higher load without any preparation.
  + Pods might crash at terrible times under high workload
* Using/configuring external load balancers to access the applications via the internet – yet another technology to learn and manage
* Creating health checks for every component
* Integration into the build pipelines
* Learning YAML
* Monitoring

This research aims to address all the problems mentioned above by developing a model to consider the dependency levels between microservices to design a more improved deployment strategy and support automated deployment and provide a monitoring solution to get a better idea about the whole microservice architecture to get business-wise decisions.

# OBJECTIVES

## Main Objective

The main objective of this research to generate a dependency network based on the metrics gathered from the Kubernetes cluster to provide a better understanding of the whole microservice architecture. This aids in making decisions regarding the deployment strategy of the microservice architecture.

## Specific Objectives

The following are the sub-objectives of conducting this research.

* To configure the Kubernetes cluster to get the metrics of the microservices and nodes.
* To develop a scheduler to save metrics for a point of time by taking a snapshot of Kubernetes metrics.
* To generate a dataset on demand.
* To analyze the network and calculate the network traffic
* To support the auto-deployment by generating **Yaml** files and applying them to the cluster.
* To provide a monitoring solution with the real-time metrics

# METHODOLOGY

## Requirement Gathering

Requirement gathering was through performing an extensive analysis of past research conducted throughout recent years, identification and analysis of the existing systems, as well as reading through a variety of online resources. Kubernetes' official documentation and the Istio documentation were the primary resources when gathering the requirements.

### Past research analysis

When it comes to Past Research Analysis, there are quite a lot of research papers and publications under the topic of Service mesh. However, there was a smaller number of publications considering the dependency levels between microservices. Key topics of interest included Service Mesh, Load balancing, load prediction, and network monitoring.

During the past research analysis, the main focus was to identify the methodologies and the tools used to build the existing tools and platforms. Moreover, it helped to identify the problems that the past researchers faced.

### Identifying existing systems

There are several existing APM tools to monitor applications running on top of Kubernetes [20, 21]. However, those APMs do not provide any suggestions or support for an optimal deployment strategy. Moreover, most of the APMs are running independently, and it is harder to monitor every monitoring solution. There is a void in the current market for a monitoring solution to combine existing ones to provide a complete, more detailed platform and suggestions for a better deployment strategy and support for auto-deployment. Furthermore, there were no existing systems that consider the dependency between microservices and suggest an optimal deployment strategy.

## Feasibility Study

### Technical Feasibility

In order to develop the optimization model, all researchers should have basic knowledge of Kubernetes and its relevant components. Members should be able to perform basic configuration and should have sufficient knowledge and practical experience on how to deploy microservices through Kubernetes.

Also, the researchers should have a basic idea about Istio service mesh as this research highly rely on Istio service mesh metric services.

#### Knowledge on Kubernetes

In order to develop the optimization model, all members should have basic knowledge of Kubernetes and its relevant components. Members should be able to perform basic configuration and should have sufficient knowledge and practical experience on how to deploy microservices through Kubernetes.

#### Knowledge of APM tools

In order to develop the optimization model, all researchers should have quite an in-depth understanding of the existing APM tools and the features and drawbacks present. The members should also have sufficient knowledge of configuring and APM tools selected for this research as well as knowledge on how to integrate the selected APM tools with the optimization model.

#### Knowledge in Machine Learning

In order to develop the optimization model, all researchers should have quite basic knowledge of machine learning basics as well as time series analysis. Members should be aware of the time series prediction models as well as knowledge on how to integrate the relevant models with machine learning and develop basic algorithms. Furthermore, members should also have a basic understanding of the Python programming language and related python machine learning and time series libraries.

#### Knowledge in Microservices

The topics like containerization, dockers, Kubernetes, and service mesh are all based on microservices. Therefore, having a good idea and understanding of the concepts and the techniques of the microservice architecture is highly essential.

#### Knowledge in YAML

When it comes to auto-deployment in Kubernetes, the configuration language, Yaml [15], plays a vital role. Since one of the outcomes of the research is to come up with an auto-deployment solution, knowing Yaml is an added advantage.

### Schedule Feasibility

As the output of this research is an input for the researches in other team members, the solution should implement within at least four months. The auto-deployment part of the solution can implement at the final stage of the research as it relies on the generated optimal deployment strategy.

### Economic Feasibility

The cost of the project should be as minimal as possible for it to be included and accepted in the existing APM tool market in Kubernetes because most APM tools and solutions offered currently concerning Kubernetes are often opensource.

## Requirement Analysis

The requirement analysis phase was one of the keys phases in this research project since it enabled in the identification of a variety of factors that should be considered in the implementation process of this research.

During this process, the information gathered from the various sources during the requirement gathering phase was analyzed. As a result, the critical factors related to the possible challenges that may be encountered as well as insight into the methodology and also a clear understanding of the use of possible and tools and technology, were also able to be easily identified.

Furthermore, a clear idea of the scope of the research, as well as the feasibility of the project, was also able to be identified during this phase. Requirement analysis also helped in the determination of the existing research gaps as well as provide insight into the identification underlying research problem as the research.

## Software Solution

The overview of the system is as follows. Its structure is mainly composed of the following components.

* Create a Kubernetes cluster on Azure AKS [22]
* Configure Istio [23]
* Track and Monitor Network traffic and dependency
* Store metrics using a scheduler
* Generate datasets on demand
* Build the dependency network
* Support Auto deployment

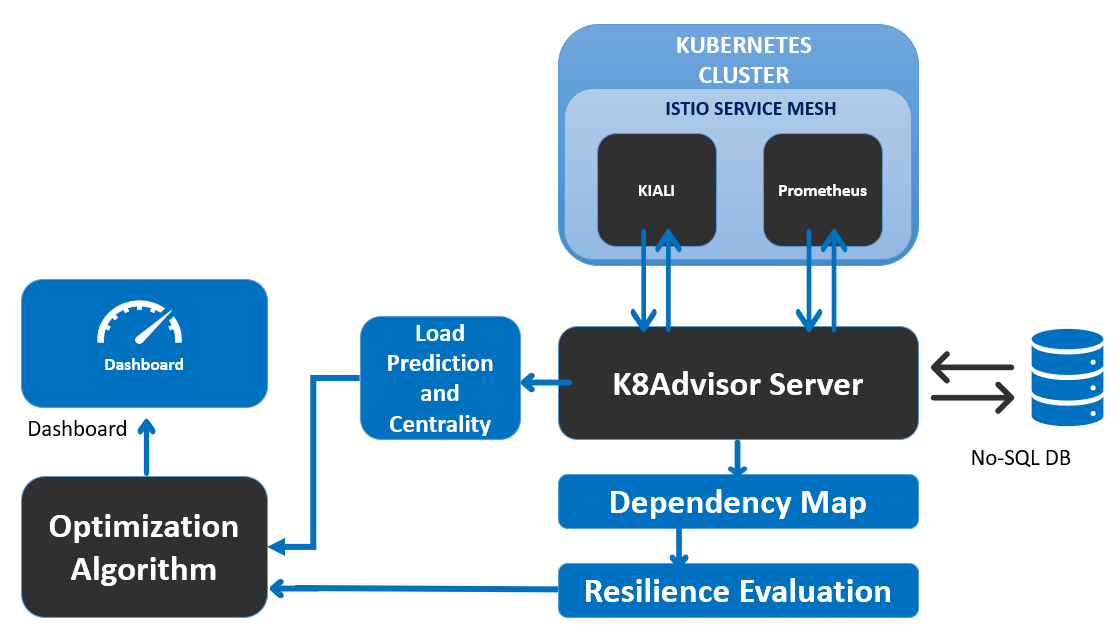


Figure 4.1: Solution Architecture

### Dependency Network

The microservice dependency network is a map that depicts the architecture of the microservice cluster based on the level of dependency among microservices. This model uses a series of tools to capture the network requests and responses and reverse engineer the metrics collected to build a dependency network based on the timestamps.

## System Development and Implementation

The implementation process of the model performs as per Figure 4.2 given below.

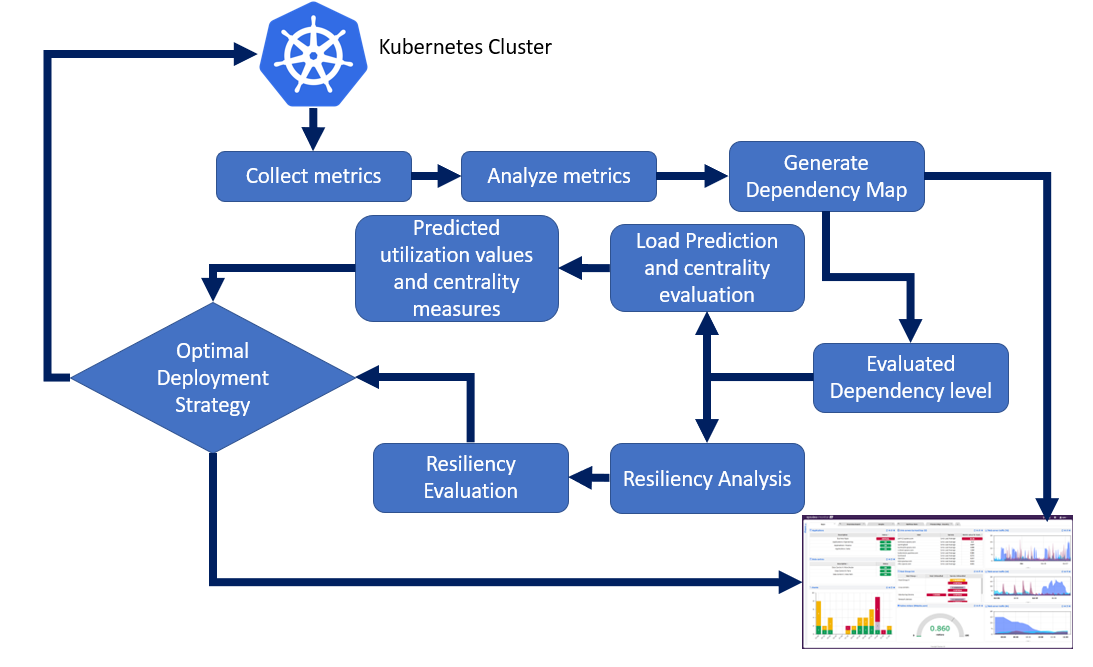


Figure 4.2: Developed system architecture

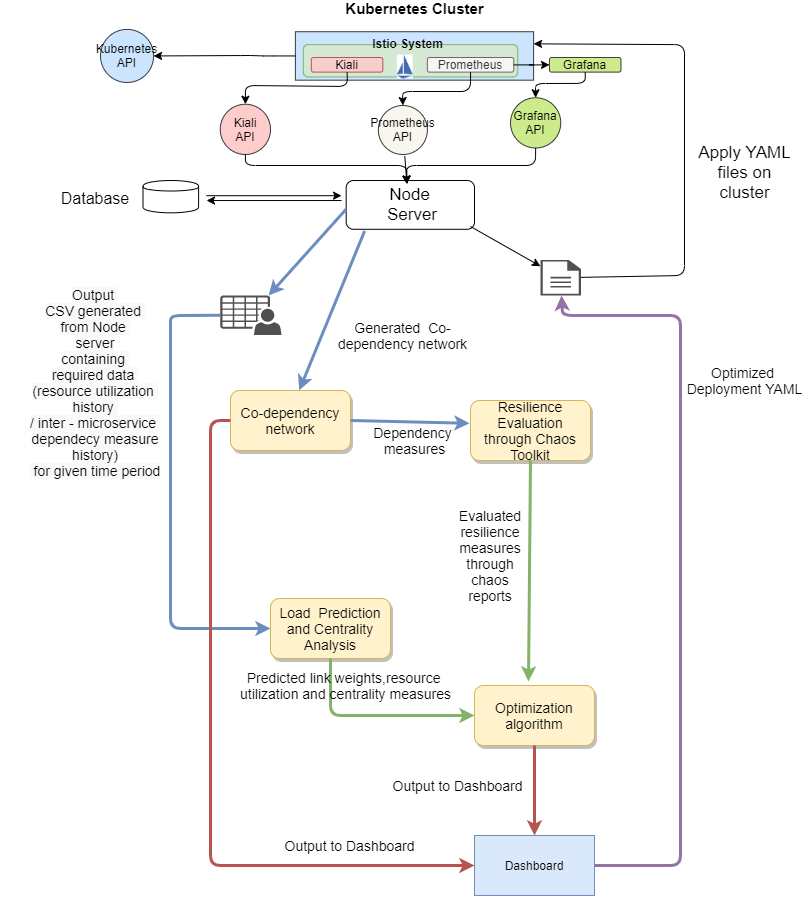


Figure 4.3: Detailed system architecture

The steps followed to reach the solution are as given below.

1. Created a Kubernetes cluster in Azure AKS
2. Configured Istio and deploy it to the created AKS cluster.
3. A proxy on top of each node is created to capture the network requests and responses in and out.
4. Came up with the algorithm to backtrack the captured metrics with the timestamps to quantify the dependency levels between microservices of the cluster.
5. Created and optimized a dataset to use in the load prediction and resiliency analysis part of the research.
6. Visualized the generated dependency network in the format of a chart and displayed it in the business intelligence dashboard.

### Building the Dependency Network

The word **Service Mesh**is a frequently seen word when talking about the microservice architectures. A service mesh is a configurable, low‑latency infrastructure layer

designed to handle a high volume of network-based inter-process communication among application infrastructure services using application programming interfaces (APIs) [24]. A service mesh ensures that communication among containerized and often ephemeral application infrastructure services is fast, reliable, and secure. The mesh provides critical capabilities, including service discovery, load balancing, encryption, observability, traceability, authentication and authorization, and support for the circuit breaker pattern.

The service mesh is usually executed by providing a proxy instance, called a sidecar*,* for each service instance. Sidecars handle interservice communications, monitoring, and security-related concerns – indeed, anything that can be abstracted away from individual services. This way, developers can handle development, support, and maintenance for the application code in the services; operations teams can maintain the service mesh and run the app [25]. Istio, backed by Google, IBM, and Lyft, is currently the best-known service mesh architecture. Kubernetes, which was initially designed by Google, is currently the only container orchestration framework supported by Istio.

Istio [23] makes it easy to convert a microservice application to a service mesh enabling load balancing, service authentication, routing, monitoring, and many more.

There are several core features bundled with Istio,

* Traffic Management
* Security
* Policies
* Observability

In the developed research, the feature observability is used heavily.

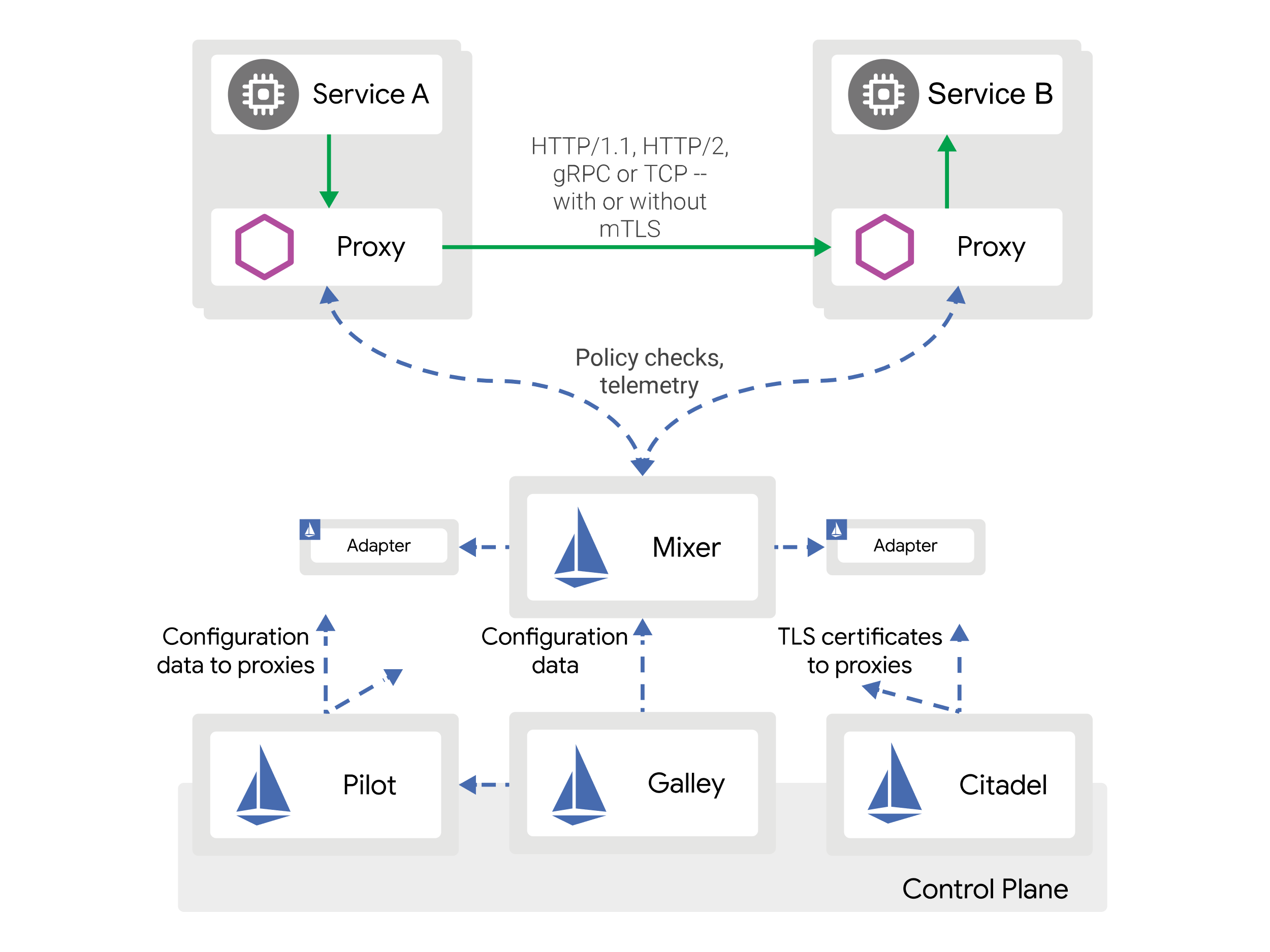


Figure 4.4: Istio Architecture for a single cluster

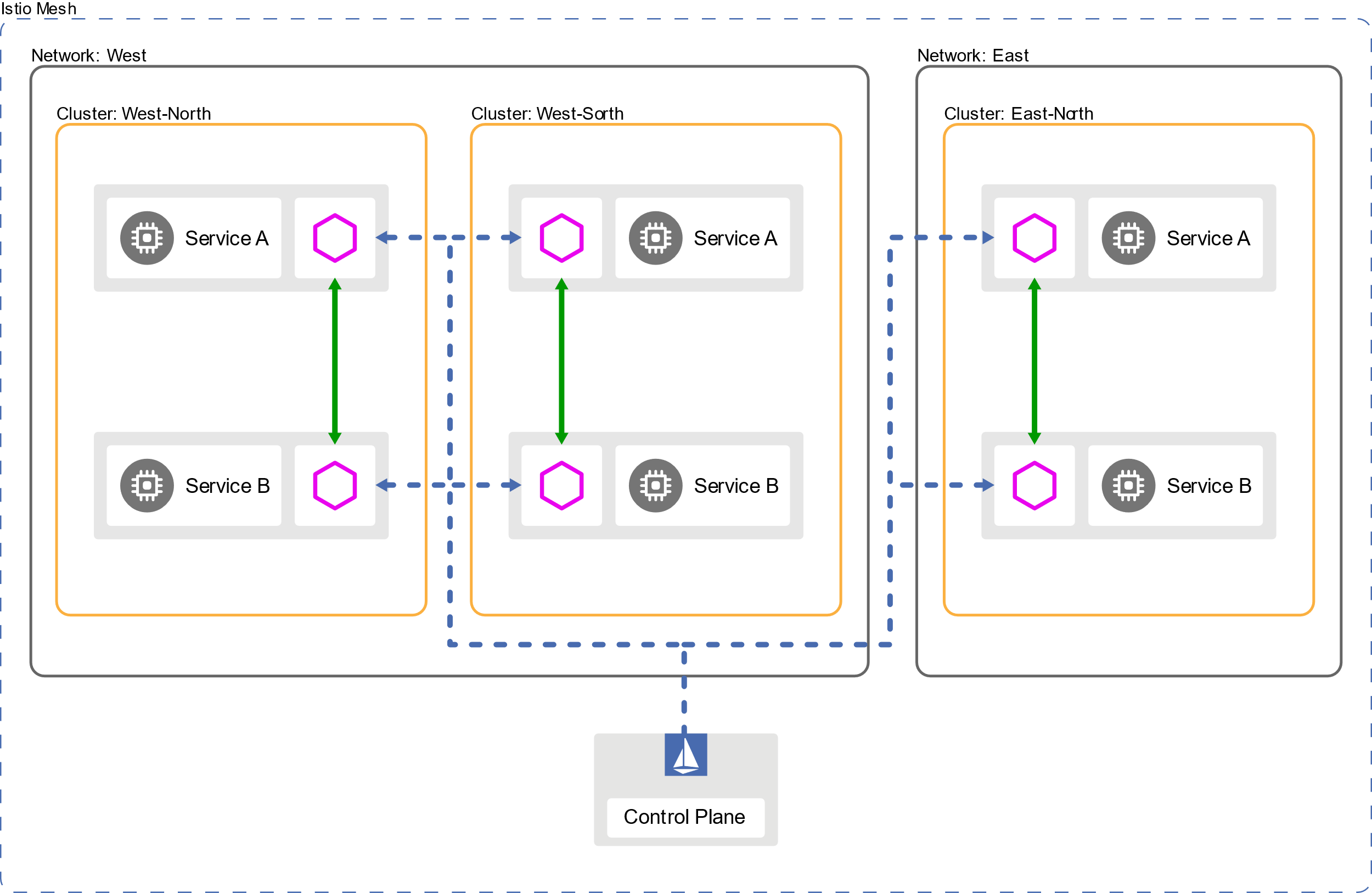


Figure 4.5: Istio Architecture for a multi-cluster

Configuration of Istio with the Kubernetes cluster exposes a set of dashboards to the user allowing the user to visualize a set of metrics of the nodes and pods inside the Kubernetes cluster.

**Prometheus** [26]

Prometheus is an open-source system monitoring and alerting toolkit originally built at SoundCloud. Many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company.

When it comes to recording any purely numeric time series, Prometheus works like a charm. Prometheus is well known for its' reliability. It fits with monitoring dynamic service-oriented architectures, as shown in figure 3.4.4.

A screenshot of a cell phone

Description automatically generated

Figure 4.6: Prometheus Architecture

**Grafana** [27]

Grafana is a dashboard that enables developers to query, visualize, and understand the metrics of a configured Kubernetes cluster in the format of multiple configurable charts, as depicts in figure 3.4.5. Grafana allows creating customized dashboards to capture and monitor relevant values to a particular subject.

Grafana is compatible with an array of data sources. In the case of this research, Prometheus mentioned above can act as a data source for the Grafana model.

A picture containing indoor, wall, shelf, table

Description automatically generated

Figure 4.7: Sample Grafana Dashboard

**Kiali**

Kiali is an observability console for Istio with service mesh configuration capabilities [28]. It helps developers to understand the structure of their service mesh by inferring the topology, and also provides the health of the mesh. Kiali provides detailed metrics, and a fundamental Grafana [27] integration is available for advanced queries. Distributed tracing is provided by integrating Jaeger.

A screenshot of a computer screen

Description automatically generated

Figure 4.8: Sample Kiali Dashboard

**Jaeger**

Jaeger [29] is used heavily in production level microservice architectures to find the network traces. It is an open-source system that can use for monitoring and troubleshooting a microservice architecture, including distributed context propagation, root cause analysis, service dependency analysis, and many more.

A screenshot of a social media post

Description automatically generated

Figure 4.9: Sample Jaeger Dashboard

The research is to get the necessary details from the dashboards mentioned above and tools using the APIs exposed from the Istio. In summary, the following tools and languages and frameworks are used for the implementation of the model.

|  |  |
| --- | --- |
| Programming Language s | Python  JavaScript |
| Configuration Languages | YAML |
| Frameworks | React JS  Node JS  Spring Boot |
| Tools | Istio  Grafana  Kiali  Prometheus  Jaeger |
| Libraries | NetworkX (Python) |

*Table 4.1: Tools and Languages Used*

### Setting up the Kubernetes Cluster

For the development and testing of this research, Azure Kubernetes Services (AKS) [30] was chosen as the Kubernetes platform provider. However, the model discussed in the research was developed in such a way that it works with the Kubernetes core services regardless of the platform provider.

Azure CLI was mainly used when initializing the cluster in AKS. Also, the Azure portal can be used to create a Kubernetes cluster and deploy it in Azure.

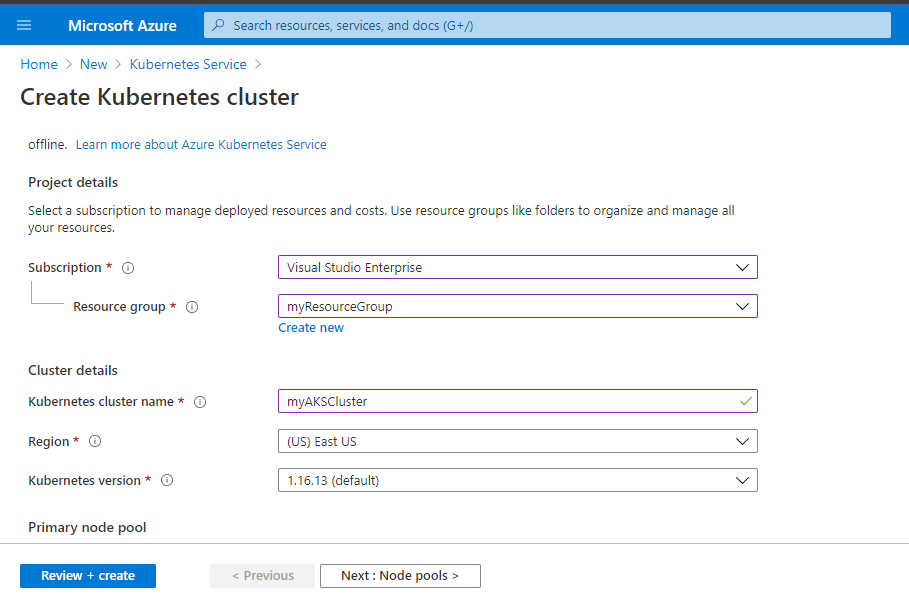


Figure 4.10: Creation of a cluster in Azure Kubernetes Services

When creating the Kubernetes cluster, the node count can be defined and can be scaled with the requirement in the future. For the purpose of developing and testing the research objective, three nodes were used in the Kubernetes cluster.

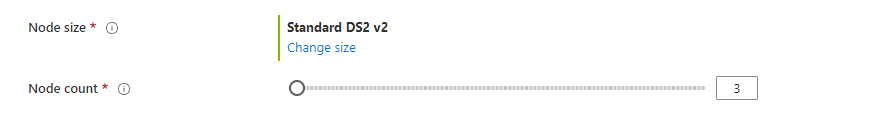


Figure 4.11: Selection of the number of nodes in the cluster

### Setting up and Initializing Istio

As previously stated, Istio, which is an open-source service mesh that provides all the primary tools need to manage a Kubernetes cluster, is used in the development of the research. Istio works well with the Linux terminal. Therefore, the Windows Subsystem for Linux (WSL) [31] is used to work with Istio.

Istio can be downloaded by navigating to the Istio release page, and it can easily be installed using the Linux terminal. The installation contains a sample application that will be used to test the research outcomes and the "Istioctl" client binaries. Adding the Istioctl client to the path variable allows Istioctl commands to be called from the terminal.

Preconfigured Istioctl client can be used to install Istio to the Kubernetes cluster setup earlier.

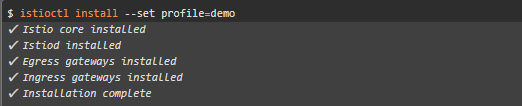


Figure 4.12: Installation of Istio



Figure 4.13: Enabling auto-injection for the side cart

As mentioned previously, Istio comes with a pre-configured sample application [32] called "BookInfo." It contains four microservices that have connections with each other.

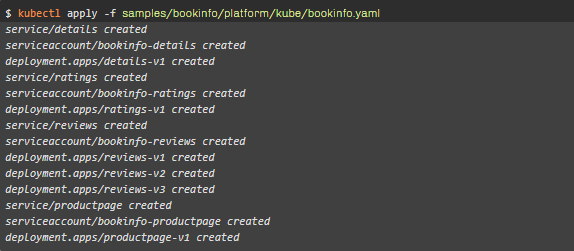


Figure 4.14: Deployment of the sample application

The sample application is deployed in the Kubernetes cluster. However, it is not accessible from the outside network. An Istio Ingress Gateway is created to map the paths and routes.



Figure 4.15: Deployment of Ingress gateway

### Setting up Kiali and Prometheus

#### Kiali Service Mesh

Kiali is a management console for istio based service mesh [33]. Kiali is a service that provides the ability to observe a service mesh and visualize the network traffic. Furthermore, Kiali provides metrics on the health of pods and services. Kiali uses Jaeger as a data source to find the network traces of the mesh.

Kiali can be installed as an add-on to the istio configured cluster.



Once Kiali is installed successfully, the dashboard can be accessed as a service.



Kiali user interface can be opened using the "**istioctl"** client library.



Furthermore, Kiali exposes a rigid and detailed set of REST APIs [34], which can be used to extract information of the service mesh programmatically. This set of REST APIs will be used in this research to get the dependency levels of the microservices.

However, when executing the above command to access the Kiali UI, the port listening as the Kiali server can be changed with the execution time. Simple port-forwarding is used to avoid that and keep a static port number.

#### Prometheus

Prometheus is built In 2012 at Sound Cloud and released as an open-source monitoring toolkit. Prometheus provides a multi-dimensional data model with time-series data of the hardware metrics of a server.

Prometheus can be used as a data source for monitoring dashboards such as Grafana to visualize the metrics.

Same as Kiali, Prometheus also can be installed as an add-on to the istio configured cluster using the "istioctl" client library,





Prometheus also provides a set of REST APIs [35] that can be used to query metrics programmatically.

### Implementation of the Middle Tier Server

A central server to execute the logic for metrics gathering and storing those in a database is crucial to the research as the stored data is used to generate time-series datasets, which is used to train the machine learning models.

To keep the simplicity of the implementation and the lightweight of the execution, Node JS is used as the primary framework for the middle tier server.

The specialty of this research is that using a customer-specific dataset to train the models. A cloud mongo database is hosted inside the cluster to save the data related to the dependency and hardware metrics.

### Deployment of Middle Tier Server

As mentioned in the previous section, the middle-tier server is developed in Node JS, and the implementation is containerized with Docker. The built docker image is used to deploy a container inside the cluster. The deployed cluster can run both Kiali and Prometheus servers within the container and consume respective REST APIs.

A separate scheduler is implemented using Python to call an endpoint which queries the cluster for metrics and save them inside the mongo database with the timestamp. The scheduler is set up to run every hour to create a consistent dataset. However, the customer can change the period based on the cluster and the application preference.

Another endpoint is created to read the last 100 records of the database and create separate CSV files for dependency metrics and CPU metrics. The number of records is allowed to change using a parameter to the endpoint.

## Project Requirements

### Functional requirements

The primary functional requirements aimed at fulfilling during the implementation process in this research are as follows.

* Quantified dependency metrics should be able to capture.
* CPU metrics should be able to capture.
* Captured metrics should be saved in a database
* A dependency map should be created
* Separate time-series datasets should be able to generate

### Non-Functional Requirements

The following are the non-functional requirements that are primarily being focused on during this research.

* Availability – The system should be able to function throughout the day without any restrictions.
* Efficiency – The system should be as efficient as possible and make use of minimal resources in the prediction process such that it does not affect the performance of the overall cluster.
* Performance – The system should handle the vast amount of data it receives and processes it without affecting the system performance.
* Interoperability – The system should interact and communicate with the other components developed in this research and receive inputs and forward outputs to the desired components.

## Commercialization

Even though this is developed as a component of the primary governance model, this component and the dependency map could be developed as a standalone monitoring application for Kubernetes because this provides all the necessary information about the

1. The health of the cluster
2. Quantified dependency between microservices
3. The CPU load of each pod

in one single location.

Furthermore, this research component can be used as follows.

1. A data source for load prediction
2. A data source for dependency prediction
3. A visualizer for dependency
4. A visualizer for load deviation
5. All in one monitoring solution.

The primary target audience for this is System Administrators, Cloud Engineers, and DevOps Engineers, who work with a cloud-based Kubernetes platform. The management level of a company is identified as the secondary level audience as the management decisions such as the resource allocation can be taken considering the output of the developed component.

A freemium [36] model is decided to use for marketing as there are several free and open-source monitoring solutions available in today's market, which creates competition.

# RESULTS AND DISCUSSION

## Testing

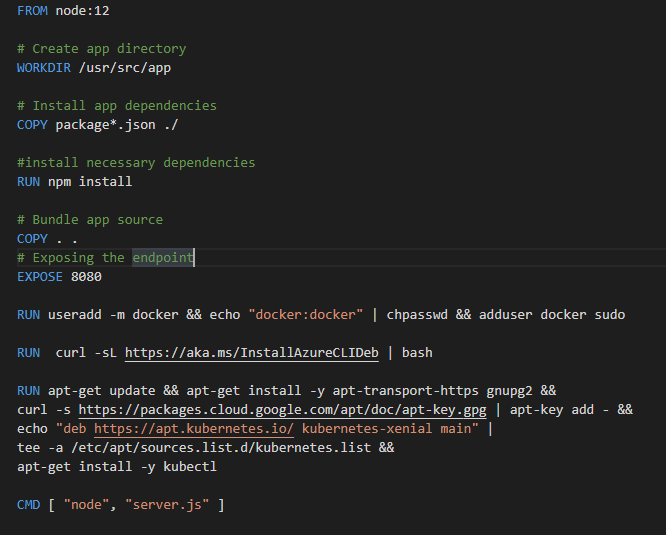
### Deployment of a set of sample microservices

Testing of the research outcome plays a crucial role in the success of the research. A set of sample microservices is created for testing purposes. The created microservices are deployed to a Kubernetes cluster located in Azure using deployment YAML files.

### Deployment of The Developed Node Server

A cloud mongo database deployed in mongo atlas is used to store the queried metrics. The developed Node JS server is containerized using Docker to maintain an isolated environment.

However, a set of client libraries should be installed inside the Linux environment for the execution of the server. It is handled by installing all the necessary dependencies when creating the container with the docker image.



The built docker image is deployed in the same cluster with the help of YAML configuration, which creates an app service and load balancer inside a new namespace. The load balancer exposes a public IP address that can be accessed from outside of the cluster.

The node server then can be accessed from the public IP and the exposed port from the docker container.

### Data Collection

A scheduler is set up to call an API exposed from the Node server in specific time intervals. Each time that API is called, it queries the entire Kubernetes cluster and adds a database record on the quantified dependency and the CPU load for the timestamp. Another API is exposed to read the database and generate a CSV file, and the generated files are used as a data source for the developed dashboard.

## Test Results – Final Solution

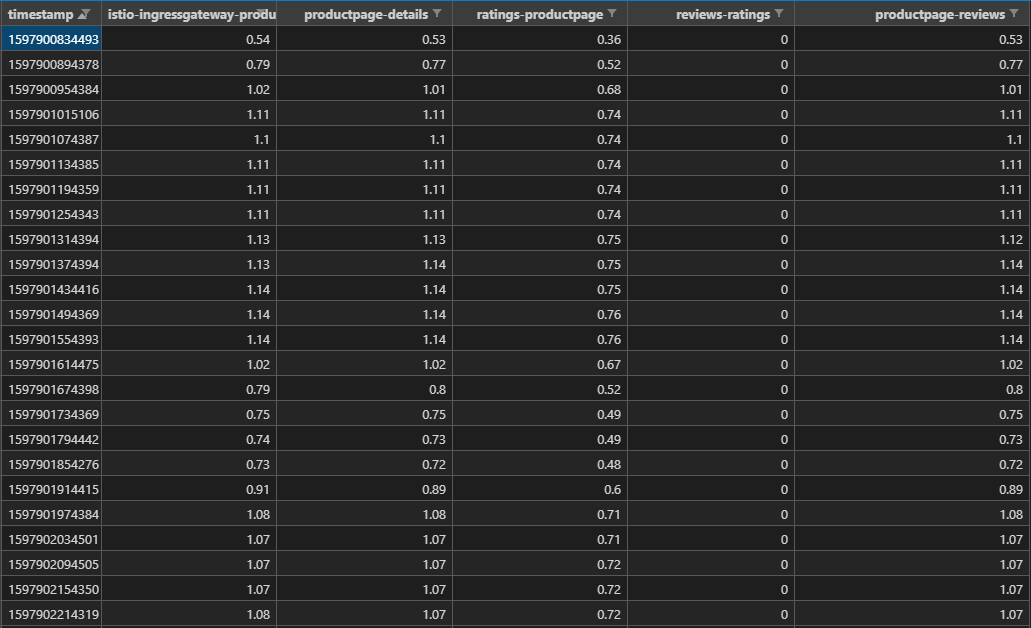


Figure 5.1: Generated quantified dependency CSV

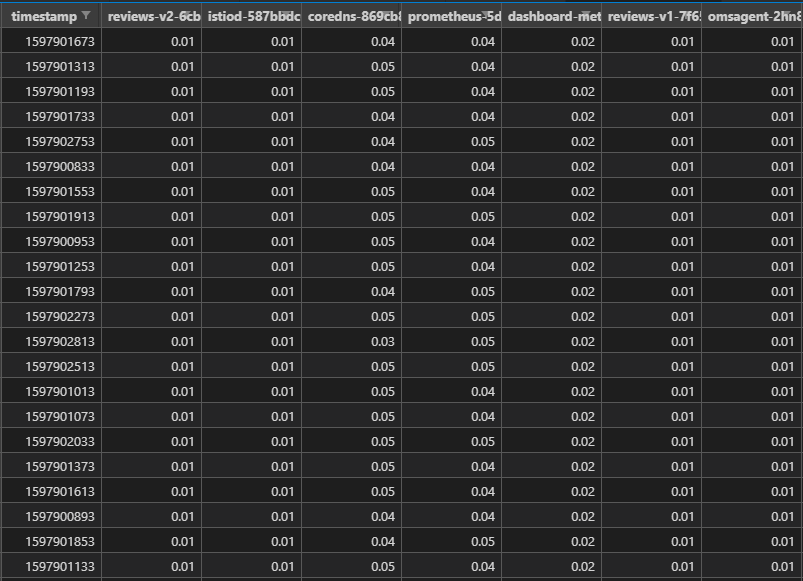


Figure 5.2: Generated CPU load CSV

When it comes to the test results, the gathered metrics evidently display a quantified dependency between microservices. Furthermore, a simple node graph is generated to visualize the direction and the dependency level.

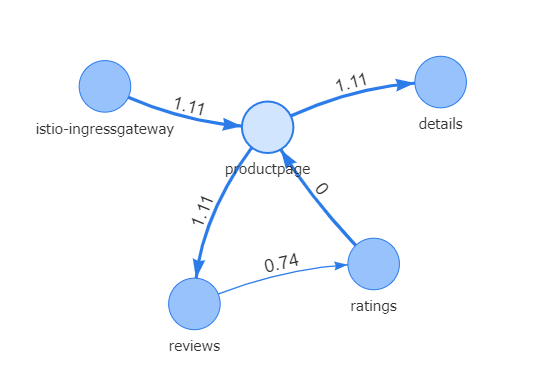


Figure 5.3: Generated node-edge graph based on the dependency

## Research Findings

As the data is gathered in a manner of a time series, the fluctuation of the data against the time is visualized.

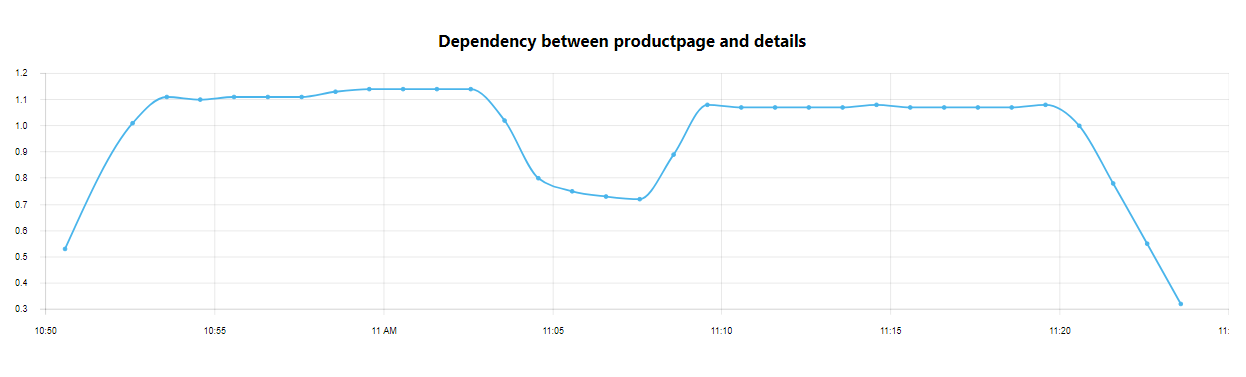


Figure 5.4: Variation of dependency against the time

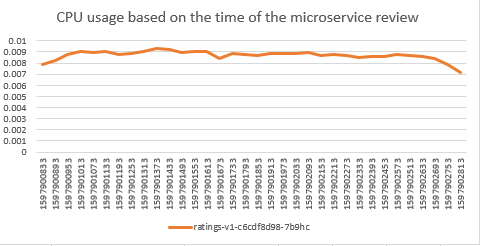


Figure 5.5: Fluctuation of the CPU usage against time

According to Figure 5.4 and Figure 5.5, it is evident that the usage of the microservice has a significant impact on dependency and CPU usage. When It comes to the deployment strategy of the microservices, if the dependency is considered, the efficiency of the cluster can be increased drastically.

The testing of the research is done on a set of simple microservices. However, the fluctuation of the dependency and the CPU load has a significant amount of values in a production-grade cluster.

## Discussion

Even though microservices hold the position of most popular and most used software architecture, the complexity of a system creates many problems. The technologies like Docker and Kubernetes were developed to overcome the complexity of the microservice applications. However, decreasing the performance of the entire system is a critical problem when it comes to microservice deployment.

A microservice deployment is often compared with a network mesh. In network science, several fundamental principles are used to increase the performance of the whole network. However, when it comes to microservice deployment, the reliability and availability of the services should also be considered.

Dependency between microservices is a crucial metric as it can be used to get an idea of how vital one service to another. Furthermore, that idea can be considered when developing the deployment strategy. However, it is not considered in the current Kubernetes deployment.

When it comes to scaling an application, Kubernetes provides horizontal pod auto-scaling, but it does not consider the run time dependency of microservices as well as the CPU load. Since this research component query and collects the metrics mentioned above, those metrics can be used to generate time-series datasets which are used to train machine learning models to predict the future loads and dependencies.

Predicted dependencies and CPU loads are then used to generate an optimal deployment model for the next twenty-four hours. That way, the microservice application can maintain reliability and availability and keep the performance simultaneously.

# CONCLUSION

According to the findings of the research, it is evident that the dependency between microservices plays a significant role when it comes to the performance of the entire application even though it is not considered in current Kubernetes deployments.

Furthermore, by creating a time-series can help in generating a prediction which can be used to be prepared for auto-scaling and then considered in developing an optimal deployment strategy.

Nevertheless, the importance of all in one monitoring solution is identified because even though there are several monitoring solutions available in the current market and the lack of tools that provides data and information to make management decisions.

**References**

|  |  |
| --- | --- |
| [1] | C. Richardson, "Monolithic Architecture," [Online]. Available: https://microservices.io/patterns/monolithic.html. |
| [2] | M. Rouse and J. Sullivan, "Client-server model (client-server architecture)," SearchNetworking, [Online]. Available: https://searchnetworking.techtarget.com/definition/client-server#:~:text=denial%20of%20service.-,Client%2Dserver%20protocols,end%20have%20finished%20exchanging%20messages.. |
| [3] | Cloud Flare, "What is TCP/IP?," Cloud Flare, [Online]. Available: https://www.cloudflare.com/learning/ddos/glossary/tcp-ip/. |
| [4] | L. Mauersberger, "Microservices: What They Are and Why Use Them," 14 08 2017. [Online]. Available: https://www.leanix.net/en/blog/a-brief-history-of-microservices. |
| [5] | J. Strotmann, "Infographic: A Brief History Of Containerization," PLESK, 13 12 2016. [Online]. Available: https://www.plesk.com/blog/business-industry/infographic-brief-history-linux-containerization/. |
| [6] | Educuba, "Docker Alternatives," Educuba, [Online]. Available: https://www.educba.com/docker-alternatives. |
| [7] | B. B. Rad, H. J. Bhatti and M. Ahmadi, "Advantages of Docker Container," An Introduction to Docker and Analysis of its Performance," vol. 16, pp. 230-232, 2017. |
| [8] | J. Turnbull, The Docker Book: Containerization Is the New Virtualization, 2019. |
| [9] | Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. |
| [10] | Azure, Microsoft, "What is Kubernetes?," [Online]. Available: https://azure.microsoft.com/en-us/topic/what-is-kubernetes/. |
| [11] | "2019 CNCF Survey results are here: Deployments are growing in size and speed as cloud native adoption becomes mainstream," Cloud Native Computing Foundation, [Online]. Available: https://www.cncf.io/blog/2020/03/04/2019-cncf-survey-results-are-here-deployments-are-growing-in-size-and-speed-as-cloud-native-adoption-becomes-mainstream/. |
| [12] | "Large-scale cluster management at Google with Borg," Google Cooperation, [Online]. Available: https://research.google/pubs/pub43438/. |
| [13] | W. Li, Y. Lemieux, J. Gao, Z. Zhao and Y. Han, "Service Mesh: Challenges, State of the Art, and Future Research Opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, San Francisco East Bay, CA, USA, USA, 2019. |
| [14] | P. Bakker, "One year using Kubernetes in production: Lessons learned," TechBeacon, [Online]. Available: https://techbeacon.com/devops/one-year-using-kubernetes-production-lessons-learned. |
| [15] | R. Gibb, "What is YAML?," StackPath, 17 7 2019. [Online]. Available: https://blog.stackpath.com/yaml/. [Accessed 22 2 2020]. |
| [16] | D. Vohra, "Why Kubernetes ?," in *Kubernetes Microservices with Docker*, pp. 41-42. |
| [17] | "Kuberners," Kubernetes.io, [Online]. Available: https://kubernetes.io/. [Accessed 14 01 2020]. |
| [18] | "How to automate Kubernetes workflows," [Online]. Available: https://buddy.works/guides/how-optimize-kubernetes-workflow. [Accessed 24 02 2020]. |
| [19] | J. Hirschaurer, "Problems solved and problems created by Kubernetes," Instana, 10 12 2018. [Online]. Available: https://www.instana.com/blog/problems-solved-and-problems-created-by-kubernetes/. [Accessed 23 02 2020]. |
| [20] | A. Murty, "Kubernetes: How New Relic Sets You Up for Success," [Online]. Available: https://blog.newrelic.com/engineering/monitoring-application-performance-in-kubernetes/. [Accessed 18 02 2020]. |
| [21] | D. Berman, "Top 10 Open Source Monitoring Tools for Kubernetes," logz.io, 4 10 2019. [Online]. Available: https://logz.io/blog/open-source-monitoring-tools-for-kubernetes/. [Accessed 12 02 2020]. |
| [22] | "Azure Kubernetes Service (AKS)," Microsoft Azure, [Online]. Available: https://azure.microsoft.com/en-us/services/kubernetes-service/. [Accessed 24 02 2020]. |
| [23] | 2019 Istio Authors, "What is Istio?," Istio, [Online]. Available: https://istio.io/docs/concepts/what-is-istio/. [Accessed 21 02 2020]. |
| [24] | F. Smith and O. Garrett, "What Is a Service Mesh?," NGINX, 03 04 2018. [Online]. Available: https://www.nginx.com/blog/what-is-a-service-mesh/. [Accessed 12 02 2020]. |
| [25] | C. Fred Moyer, "Comprehensive Container-Based Service Monitoring with Kubernetes and Istio". |
| [26] | Prometheus Authors, "What is Prometheus," [Online]. Available: https://prometheus.io/. [Accessed 21 02 2020]. |
| [27] | Grafaana Labs, "What is Grafana ?," [Online]. Available: https://grafana.com/. [Accessed 22 02 2020]. |
| [28] | G. Chandra, "Istio Service Mesh Observability with Kiali," 29 7 2019. [Online]. Available: https://itnext.io/istio-service-mesh-observability-with-kiali-c2ded8a413c4. [Accessed 22 02 2020]. |
| [29] | "Introduction to Jaeger," Jaeger, [Online]. Available: https://www.jaegertracing.io/docs/1.6/. [Accessed 22 02 2020]. |
| [30] | "Azure Kubernetes Services," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/services/kubernetes-service/. |
| [31] | "Windows Subsystem for Linux Installation Guide for Windows 10," Microsoft Docs, [Online]. Available: https://docs.microsoft.com/en-us/windows/wsl/install-win10. |
| [32] | "Bookinfo Application," Istio Documentation, [Online]. Available: https://istio.io/latest/docs/examples/bookinfo/. |
| [33] | Kiali Developer Group, "About Kiali," [Online]. Available: https://kiali.io/. |
| [34] | Kiali Developer Group, "Developer API Kiali," [Online]. Available: https://kiali.io/documentation/latest/developer-api/. |
| [35] | Prometheus Developer Group, "Prometheus HTTP API," [Online]. Available: https://prometheus.io/docs/prometheus/latest/querying/api/. |
| [36] | V. Kumar, "Making “Freemium” Work," Harvard Business Review, [Online]. Available: https://hbr.org/2014/05/making-freemium-work. |

**Appendix**

## Appendix A: Docker File for containerization

FROM node:12

# Create app directory

WORKDIR /usr/src/app

# Install app dependencies

COPY package\*.json ./

#install necessary dependencies

RUN npm install

# Bundle app source

COPY . .

# Exposing the endpoint

EXPOSE 8080

RUN useradd -m docker && echo "docker:docker" | chpasswd && adduser docker sudo

RUN  curl -sL https://aka.ms/InstallAzureCLIDeb | bash

RUN apt-get update && apt-get install -y apt-transport-https gnupg2 && curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add - && echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | tee -a /etc/apt/sources.list.d/kubernetes.list &&  apt-get update && apt-get install -y kubectl

# RUN ./kubernetes\_login.sh

CMD [ "./kubernetes\_login.sh" ]

## Appendix B: Yaml file used to deploy the container in the cluster

apiVersion: apps/v1

kind: Deployment

metadata:

  name: k8-server

  namespace: cdap

spec:

  replicas: 1

  selector:

    matchLabels:

      bb: web

  template:

    metadata:

      labels:

        bb: web

    spec:

      containers:

      - name: k8-server

        image: gihanrcg/k8-server:9191

---

apiVersion: v1

kind: Service

metadata:

  name: node-server-entry

  namespace: cdap

spec:

  type: LoadBalancer

  selector:

    bb: web

  ports:

  - port: 8080

    targetPort: 8080

## Appendix C: Mongo models for database

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const CpuUtilizationSchema = new Schema({

    timestamp: {

        type: Date

    },

    podName: {

        type: String,

        required: true,

    },

    value: {

        type: Number,

        required: true,

    }

});

module.exports = CpuUtilization = mongoose.model('CpuUtilization', CpuUtilizationSchema);

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const EdgeSchema = new Schema({

    timestamp: {

        type: Date

    },

    dependencyName: {

        type: String,

        required: true,

    },

    sourceId: {

        type: String,

        required: true,

    },

    sourceService: {

        type: String

    },

    targetId: {

        type: String,

        required: true,

    },

    targetService: {

        type: String,

        required: true,

    },

    requestsPerSecond: {

        type: Number,

    }

});

module.exports = Edge = mongoose.model('Edge', EdgeSchema);

## Appendix D: Yaml used to deploy auto scaler and load prediction component

apiVersion: apps/v1

kind: Deployment

metadata:

  name: load-prediction

  namespace: cdap

spec:

  replicas: 1

  selector:

    matchLabels:

      bb: web

  template:

    metadata:

      labels:

        bb: web

    spec:

      containers:

      - name: load-prediction

        image: nishitha97/load\_pred\_v2:v1

---

apiVersion: v1

kind: Service

metadata:

  name: load-prediction-entry

  namespace: cdap

spec:

  type: LoadBalancer

  selector:

    bb: web

  ports:

  - port: 5000

    targetPort: 5000

## Appendix E: Kiali Routes

const express = require('express');

const { authenticateKiali, getKialiStatus } = require('./KialiFunctions');

const { createServicesGraphJson, generateCsv } = require('./KialiUtils');

const router = express.Router();

const Dependency = require('../../models/Edge')

const path = require('path');

router.get('/auth', (req, res) => {

    authenticateKiali().then(auth => {

        res.status(200).send(auth)

    }).catch(err => {

        console.error(err);

        res.status(502).send(err);

    });

});

router.get('/status', (req, res) => {

    getKialiStatus().then(status => {

        res.status(200).send(status)

    }).catch(err => {

        console.error(err);

        res.status(502).send(err);

    });

});

router.get('/services', async (req, res) => {

    let x = await createServicesGraphJson();

    res.status(200).send(x);

});

router.get('/getServiceDataSet', async (req, res) => {

    await generateCsv();

    res.status(200).download(path.resolve(\_\_dirname + '../../../generatedCsv/dependency.csv'));

})

module.exports = router;

## Appendix F: Prometheus Routes

const express = require('express');

const router = express.Router();

const { CreateCpuUsageGraph, AddDataBaseRecordCpu, generateCsv } = require('./PrometheusUtils');

const path = require('path');

router.get('/getcpu', async (req, res) => {

    let x = await CreateCpuUsageGraph();

    res.status(200).send(x);

});

router.get('/adddatabaserecord', async (req, res) => {

    let x = await AddDataBaseRecordCpu();

    res.status(200).send(x);

});

router.get('/generatecsv', async (req, res) => {

    let x = await generateCsv();

    res.status(200).send(x);

});

router.get('/getcpudataset', async (req, res) => {

    await generateCsv();

    res.status(200).download(path.resolve(\_\_dirname + '../../../generatedCsv/cpu.csv'));

})

module.exports = router;

## Appendix G: Kubernetes Util Routes

const express = require('express');

const router = express.Router();

const exec = require('child\_process').exec;

const { createServicesGraphJson } = require('../kilai/KialiUtils');

const { AddDataBaseRecordCpu } = require('../prometheus/PrometheusUtils');

const { UpdatePodNames, ReadYamlFiles } = require('./KubernetesUtils');

const path = require('path');

router.post('/startservers', function (req, res) {

    exec('sh api/utils/shells/test.sh', (err, stdout, stderr) => {

    console.log('Done')

    if (err) {

      console.error(err)

      res.status(200).send("Server starting failed...!!!");

    } else {

      console.log(`stdout: ${stdout}`);

      console.log(`stderr: ${stderr}`);

    }

  });

  res.status(200).send("Server is starting and good to go...!!!");

});

router.post('/adddbentries', (req, res) => {

  let i = 1;

  setInterval(() => {

    console.log(`Adding record ${i} at ${new Date().toLocaleTimeString()}`);

    createServicesGraphJson();

    AddDataBaseRecordCpu();

    i++;

  }, 60000);

  res.send('200');

});

router.get('/update\_pod\_names', async (req, res) => {

  await UpdatePodNames();

  res.send("Pod Names updated successfully..!");

});

router.get('/get\_pod\_limits', async (req, res) => {

  await UpdatePodNames();

  await ReadYamlFiles();

  res.status(200).download(path.resolve(\_\_dirname + '../../../generatedFiles/pod\_cpu\_limits.json'));

});

module.exports = router;