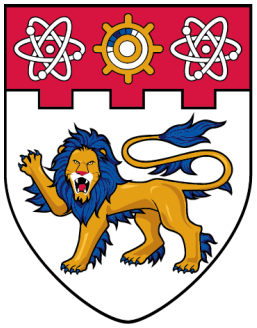

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SCSE21-0060: Docker and Kubernetes – Enhance the securities of the live-stream ASR system deployment in the cloud

Joshua Lee Jun Xiang

U1821231B

Project Supervisor: Assoc. Prof Chng Eng Siong

Examiner: Ast. Prof Zhao Jun

School Of Computer Science and Engineering

2021/2022

NANYANG TECHNOLOGICAL UNIVERSITY

SCSE21-0060:
Docker and Kubernetes – Enhance the securities of the live-stream ASR system deployment in the cloud

Submitted in Partial Fulfilment of the Requirements for the Degree of Bachelor of Engineering in Computer Science of the Nanyang Technological University

by

JOSHUA LEE JUN XIANG

School Of Computer Science and Engineering

2021/2022

Abstract

The goal of this project is to employ relevant and current security solutions to increase the robustness and reliability of an existing automatic speech recognition (ASR) system deployment on the cloud. The implemented solutions will harden the ASR deployment, reducing both the attack surface and exploitable vulnerabilities on the system. The security solutions utilises Nginx Ingress Controller and the secret management tool, HashiCorp Key Vault. They are applied on the ASR system hosted on the Azure cloud platform using Azure Kubernetes Service (AKS) and on the Amazon Web Service (AWS) cloud platform using Elastic Kubernetes Service (EKS). An Infrastructure as Code (IaC) tool, Terraform, is also implemented to improve the deployment of the ASR system.

The security solutions better secure the system's endpoints and carry out appropriate encryption of secret data. This ensures the performance and availability of the ASR deployment services, as well as prevents potential leakage of sensitive information. This will be illustrated in the report through architectural diagrams, figures, and tables, detailing important elements of the solution and how they are implemented. The report will also include the experiments carried out to demonstrate the effectiveness of the solution.

Acknowledgement

I wish to offer my deepest thanks to the following people who have provided guidance and support to me during this project. I would like to express my gratitude and appreciation to them.

I would like to thank Associate Professor Chng Eng Siong for setting his expectations and providing guidance and advice for this project. This allowed me to gain essential insight and understanding of the project's background and scope, as well as what are the suitable approaches to be taken for the project's solution. Despite his busy schedule, he is always patient and willing to assist in various matters concerning the project.

I am also grateful to my mentor, Research Staff Vu Thi Ly, for providing valuable feedback and resources on what solutions should be implemented for this project. She helped in identifying the appropriate design approach for this project and ensured the project was well on track and progressing smoothly through regular updates and meetings.

Finally, I would like to thank my friends and family who have provided me support throughout this project.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

There is an automatic speech recognition (ASR) system developed based on Kaldi, an open-source toolkit used for speech recognition. The system utilises finite-state transducers to execute the speech recognition process [1], accepting input in the form of an audio file or a live audio stream and transcribing it to produce a transcript of what was said. A committed team of NTU researchers developed speech recognition models that allow the ASR system to transcribe audio containing words spoken in multiple different languages. The system is able to output a transcript detailing what was spoken in their respective languages. The code-switching ASR system can potentially support many current and relevant use cases such as transcribing calls made to customer service centers for documenting and archiving, as well as generating subtitles for lectures and recordings.

The ASR system is containerised into a container image using Docker, a popular containerisation tool. The Docker container image is then used to deploy the system as a container application using Kubernetes, a container orchestration tool. Previous students have con-

tributed to the project by deploying the ASR system on the Azure cloud platform and the Amazon Web Services (AWS) cloud platform using Azure Kubernetes Service (AKS) and Elastic Kubernetes Service (EKS) respectively. The deployment of the application on cloud platforms allows for high availability and high scalability [2], as the resources are provisioned with redundancies and scaled according to the experienced load.

The Kubernetes service available on cloud platforms provides numerous benefits, including load balancing according to the experienced load and computing costs based on the resources consumed [3]. However, the default Kubernetes implementation possesses security vulnerabilities, such as exposing each service on its own public-facing endpoint, providing a larger attack surface. Furthermore, Kubernetes Secret objects are stored in base64 encoding which can be easily decoded, potentially compromising data confidentiality and integrity [4]. This project aimed to overcome these limitations by focusing on implementing additional security measures to harden and provide more holistic security for the existing deployments.

1.2 Project Objectives and Aims

The main objective of this project was to employ additional security implementations on top of the existing deployments on the various cloud platforms. This would harden the present ASR system deployments on both AWS and AKS, improving security, decreasing the number of exploitable vulnerabilities, and making the system less susceptible and more resilient to cyber threats. Preventing such attacks will fulfill the basic security requirements of data confidentiality, data integrity, and availability of the deployment services. The goals of the project are listed and described below:

- **Reduce the attack surface on the deployment:** Implement a solution to reduce the total number of public-facing endpoints on the deployments, hence, reducing the attack surface of the deployments.
- **Transport Layer Security (TLS) end-to-end encryption:** Utilise TLS to provide end-to-end encryption, allowing the deployment to be more resilient against passive cyber attacks that monitor and scan network traffic.
- **Provide stronger encryption for Kubernetes Secret objects:** Explore current solutions that provide stronger encryption of Kubernetes Secret objects as they are stored in base64 encoding and can be easily decoded without needing any additional software.

- **Reduce visibility of secrets used in deployment:** Identify a strategy that can limit the visibility or hide the secret objects and their values, preventing unauthorised parties from being aware and obtaining said secret objects and values.
- **Regularly update encryption keys used for encrypting Kubernetes Secrets:** Devise a procedure that allows for the encryption keys, used in Kubernetes Secret encryption, to be regularly updated such that the encrypted secrets are still secure should older encryption keys be compromised.

1.3 Project Scope

The scope of this project was to apply security implementations on top of the existing ASR system deployments. Throughout the course of the project, some of the implementations brought benefits extending beyond the security element of the project, including enhancing the provisioning of the deployment by using Terraform, an Infrastructure as Code (IaC) tool and improving load balancing with the deployed reverse proxy.

1.4 Report Organisation

This section provides a list of the 6 chapters in this report. The synopsis of each chapter is detailed below:

- **Chapter 1 (Introduction):** Provides an overview of the project and describes its objectives, aim and scope.
- **Chapter 2 (Literature Review):** Elaborate on the technology used in provisioning the ASR deployments and the project solutions.
- **Chapter 3 (Analysis and Design Approach):** Illustrate the design of the solutions and why they are used in conjunction with the current ASR deployments.
- **Chapter 4 (Implementation):** Detail the solutions implemented by this project and its applications.
- **Chapter 5 (Experiments and Results):** Describes the experiments done to test the operation of the solutions and their respective results.
- **Chapter 6 (Conclusion and Future Work):** Concludes the project and offer suggestions for potential future work.

Chapter 2

Literature Review

The ASR system is deployed onto cloud platforms using containerisation technology. Figure 2.1 below provides a basic overview of the technology stack used in deploying the ASR system on both the Azure cloud platform and AWS cloud platform.

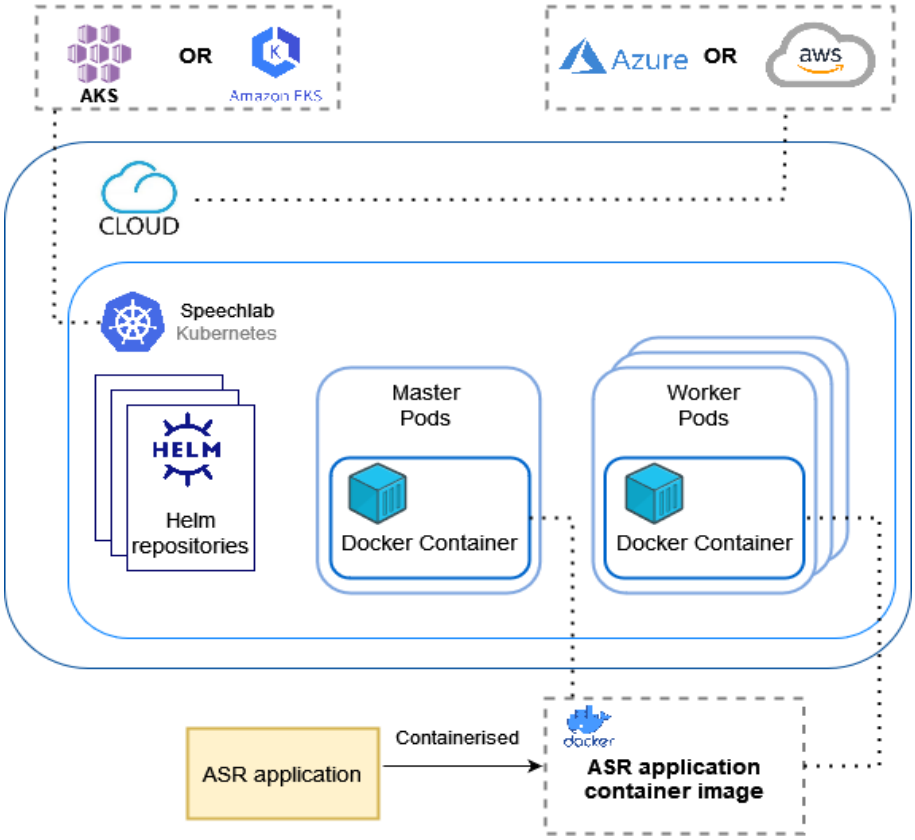

Figure 2.1: ASR deployment Overview

As seen in figure 2.1 above, the ASR application is containerised into a Docker container image. The ASR application container image is then used to run Docker containers within the Master and Worker Pods which are created and managed by the container orchestration tool, Kubernetes. The ASR system runs on both Azure and AWS cloud platforms, using their respective Kubernetes services, AKS and EKS.

Helm repositories are used to manage and install the ASR deployment together with other relevant tools and applications. The deployment, tools, and application are provision through Helm charts, which are a group of files detailing linked Kubernetes resources [5].

This chapter will continue to further discuss the various technologies used in deploying the ASR system on the cloud. It will also elaborate on the tools and services which are used in implementing the project solution.

2.1 Containerisation

Containerisation is the process of encapsulating or packaging software code and all its relevant dependencies such that it can operate reliably and uniformly on any infrastructure [6]. Containerisation consists of multiple components, namely being containers, container images, and container registries. More details on each component will be elaborated on in the subsequent sections. It serves as an alternative to virtualization, which is the process of creating a layer

of abstraction over a computer's resources such that they can be shared among multiple virtual computers [7]. These virtual computers, also known as virtual machines (VMs), can act as independent computers and run simultaneously, sharing a single physical computer's resources.

Figure 2.2 below illustrates the structural differences between a VM and a container.

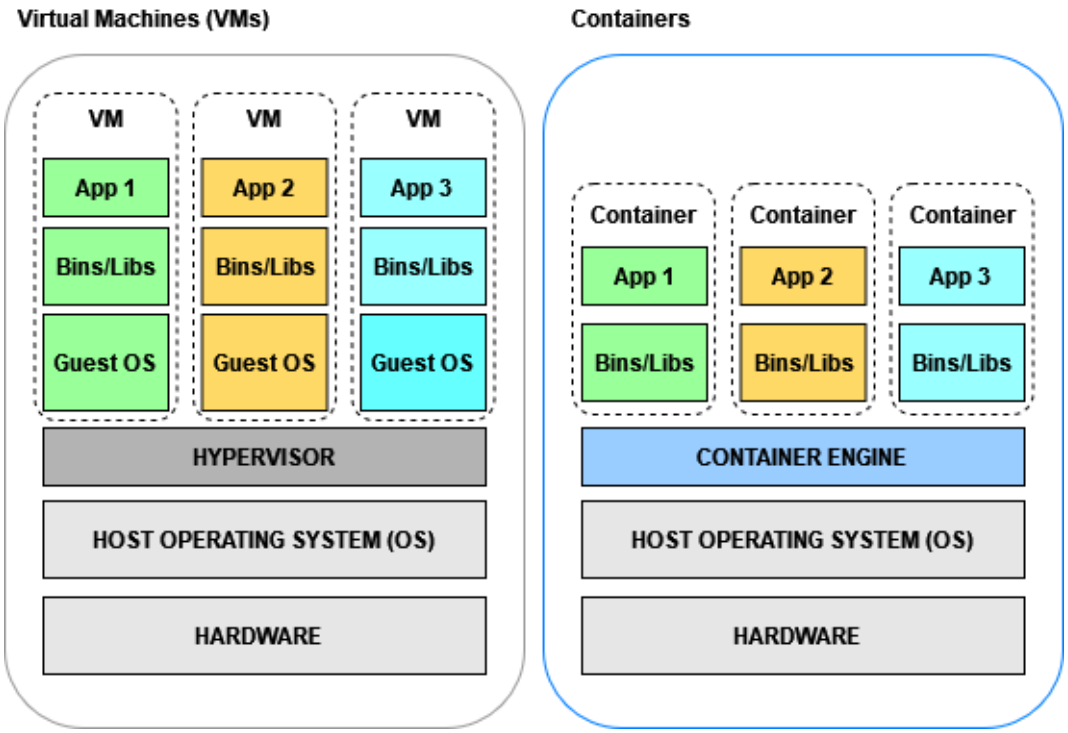

Figure 2.2: Virtualisation vs Containerisation

A key benefit that containerisation has over virtualisation is that its container runtime engine is installed on the host system's operating system (OS) as seen above. As such, it shares the machine's OS kernel and does not incur additional overheads from associating an OS

with each application, unlike VMs [8]. This allows containers to be more lightweight and require lesser computational resources as compared to VMs, increasing resource efficiency and lowering costs.

Containers

Containers are a standardised unit of software. It includes the application code and dependencies required to run software services reliably and efficiently in different computing environments [9]. Containers are lightweight as they consume lesser computational resources [10]. The lightweight nature of containers allows them to be workload portable. Furthermore, as each container is a standalone software, it also benefits from separation of responsibility and application isolation.

Container Images

Container images are lightweight, standalone, and executable packages that can run on different computing environments that have a suitable container runtime engine [9]. They are immutable and can be deployed consistently across different computing environments. During runtime, the container image is used to create the container which hosts the software services [9].

Container images consist of layers that are added on from a base image [10]. This enables users to develop and optimise container images according to their specific requirements and use cases.

Container Registry

A container registry is a repository used for storing and retrieving container images [11]. There are public container registries that allow users to share and access publicly available container images, as well as private container registries where access to the stored container images are restricted to a select group of people [12].
Container registries are commonly used to support container-based application development. As such, many popular cloud service providers such as Azure cloud platform and AWS cloud platform provide container registry services.

2.2 Docker

Docker is a free and open platform for building, delivering, and operating applications. It packages software into Docker container images and uses a container runtime engine, Docker Engine, to create containers hosting the software services [9].
The Docker platform aids developers in simplifying, organizing, and optimizing their workflow through appropriate solutions such as the use of Dockerfile, a configuration file used to assemble a container image [13]. Hence, it is widely adopted and hosted by popular cloud platforms such as Azure and AWS.

2.3 Kubernetes

Kubernetes is an open-source platform used for maintaining containerised applications and software services. It supports both declaration configuration, as well as automation [14]. As containerisation technology becomes increasingly popular amongst developers, Kubernetes serves as a suitable solution, providing a platform that allows for easy management of containerised applications.

Kubernetes infrastructure is a group of resources, also known as a cluster, used to generate a Kubernetes environment [15]. The cluster consists of a set of machines, termed nodes, that host created Kubernetes components and computing resources. There are two types of Kubernetes nodes: master nodes and worker nodes. Figure 2.3 below shows a Kubernetes cluster containing both a master node and a worker node.

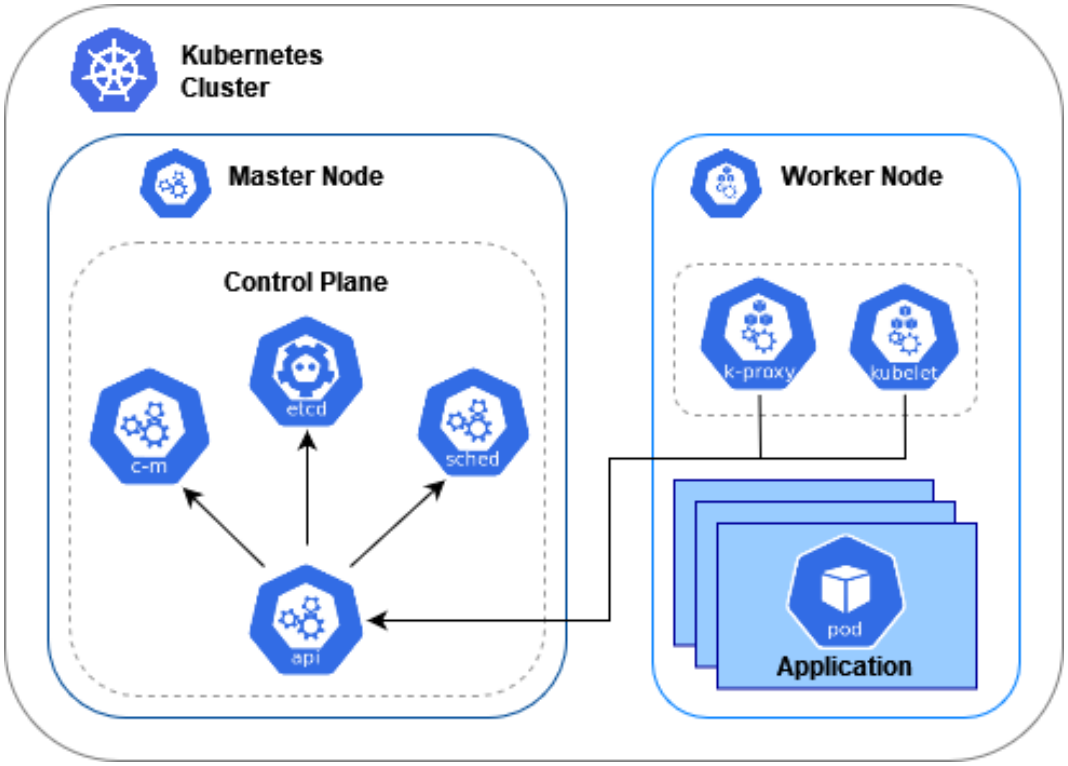

Figure 2.3: Kubernetes Cluster

As seen in Figure ref 2.3 above, the master node hosts the Kubernetes Control Plane which contains processes and objects used in managing the Kubernetes environment. The worker nodes contain the kubelet and kube-proxy, which manage the deployed applications and communicate with the Kubernetes Control Plane [16]. Containerised applications managed by the Kubernetes cluster are deployed onto the worker nodes.

Kubernetes provides a variety of services such as load balancing, service discovery, and self-healing of containers [14]. These services are implemented through the provisioning of Kubernetes resources. In

the following section, I will further explain the various Kubernetes resources that are related and relevant to the solutions implemented for this project.

2.3.1 Kubernetes Resources

The Kubernetes resources that will be detailed in this section are listed below:

- Pod
- Deployment
- Service
- Ingress
- Secret
- CronJob

Yaml configuration files are often used to define and maintain Kubernetes resources [17]. The specifications listed in the files are used to dictate and modify the behavior of the resources.

Pod

Pods are the smallest deployable Kubernetes computing resource that can be instantiated and maintained. It contains one or more containers that share storage and network resources, as well as the

specifications for how the container is run [18]. Besides the application containers themselves, a pod can also contain init containers that operate during its startup. When a pod is created, the kube-scheduler in the Kubernetes Control Plane will identify and deploy the pod onto a node that has sufficient resources to host it.

Deployment

A Kubernetes deployment is used to provision and alter instances of pods that contain a containerised application. It can scale the number of pods, systematically update the pods, and roll back to earlier deployment versions if necessary [19].
After a deployment has been instantiated, its specifications can be modified and the Deployment Controller will update the actual state to the desired state at a controlled rate.

Service

A Kubernetes service is a logical abstraction used to provide access to an application as a network service [20]. Kubernetes assigns each pod their own Internet Protocol (IP) address and a Domain Name System (DNS) name for a set of pods. When the Kubernetes service resource is created, any received request will be sent to the pods that possess the label matching the service's selector specification as shown below:

```
[...]
spec:
  selector:
    app: MyApp
[...]
```

Four different service types can be specified, each fulfilling different use cases. Table 2.1 below details each service type and their respective functions:

Service Type	Description
ClusterIP	This is the default service type and exposes the service on a cluster-internal IP. This service can only be reached from within the cluster.
NodePort	This service can be accessed from both within and outside the cluster. A ClusterIP service will be automatically created for the service to be accessed from within the cluster. The service can be accessed externally through the following: ⟨NodeIP⟩:⟨NodePort⟩
LoadBalancer	A ClusterIP and NodePort will be created automatically for the service to be accessed internally and externally. The received requests will be load-balanced to the pods behind the service.
ExternalName	The service will be mapped to the externalName field which is returned by a CNAME record with its value. This service does not require any kind of proxying.

Table 2.1: Kubernetes Service Types

Ingress

A Kubernetes ingress resource is an API object that can manage external access to the services in a cluster. This is typically done over a HTTP connection [21]. Ingress can also be configured to provide load balancing services, SSL termination, and name-based virtual hosting. For an ingress resource to be running within a cluster, an Ingress Controller that is responsible for reading and processing the ingress resource information has to be running in a cluster as well [22].

The ingress resource can be configured to direct traffic to different services through a specified hostname and subpaths. Figure 2.4 below shows a Kubernetes ingress resource directing traffic to different services according to their specified subpath.

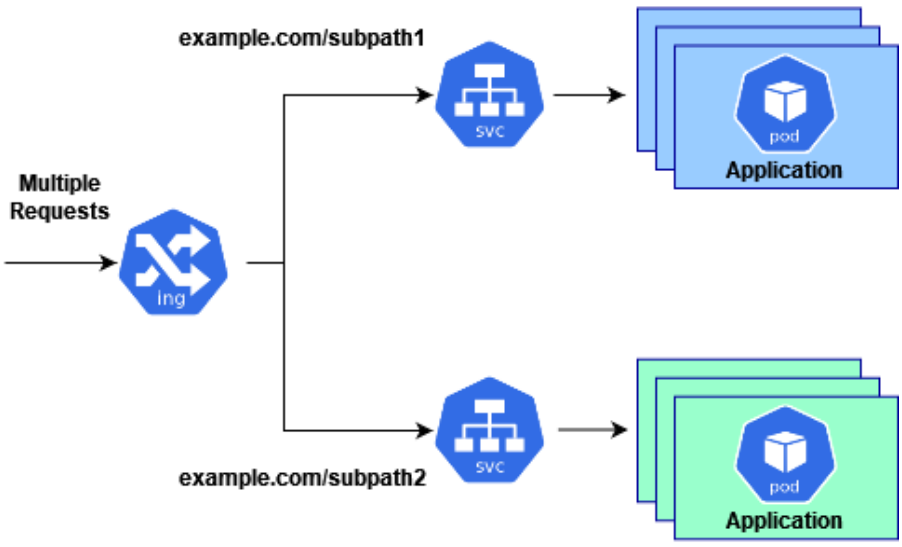

Figure 2.4: Kubernetes Ingress example

Secret

A Secret is a Kubernetes object that holds confidential data such as a password, a token, or a key. The use of Kubernetes secret objects, enables users to exclude sensitive data from the application code [4]. There are various types of Kubernetes secrets that are listed in Table 2.2 below:

Secret Type	Usage
Opaque	Data defined by the user
kubernetes.io/service-account-token	Hold service account tokens
kubernetes.io/dockercfg	Hold serialised /.dockercfg files
kubernetes.io/dockerconfigjson	Hold serialised /.docker/config.json file
kubernetes.io/basic-auth	Hold credentials for basic authentication
kubernetes.io/ssh-auth	Hold credentials for SSH authentication
kubernetes.io/tls	Hold client or server TLS data
bootstrap.kubernetes.io/token	Hold bootstrap token data

Table 2.2: Kubernetes Secret Types

Kubernetes secret objects are stored in the Kubernetes Control Plane object, etcd [4]. The secret values are encoded in base64 format which can be easily decoded. Therefore, it is recommended that better encryption solutions are implemented for Kubernetes secret objects.

CronJob

A CronJob is a Kubernetes resource that creates jobs at a set interval [23]. Jobs are a type of workload that completes or termin-

ates after finishing its process [24]. CronJobs are useful as they can execute specific processes at a set interval, removing the need for manual intervention for processes that need to be run periodically. The CronJob schedule is specified in the Cron format which is detailed in Figure 2.5 below:

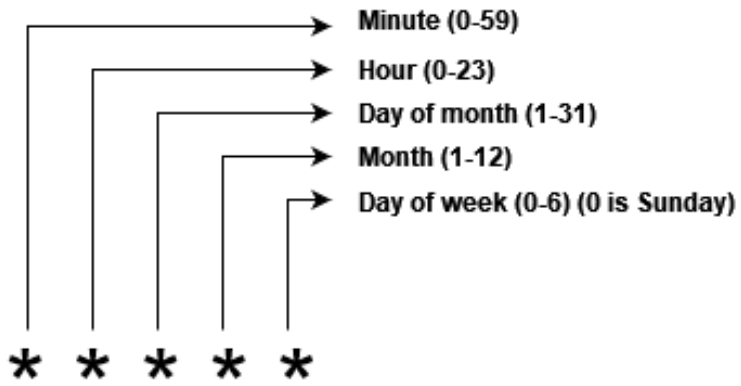

Figure 2.5: Cron format

2.4 Cloud Computing

Cloud computing describes a centralised system that guarantees Quality of Service (QoS) environments while offering flexible, dynamic, and configurable services and resources according to the client's demands [25]. Cloud computing allows applications to meet their infrastructural requirements without the need to own and maintain physical computing resources and infrastructure. The cloud computing resources are managed and maintained by the cloud providers.

This reduces the overhead costs of maintaining physical infrastructure. Furthermore, cloud computing adopts a pay-per-use payment model where a user is only charged for the resources that are being used, further increasing cost efficiency.
Two popular cloud platforms that offer services supporting Kubernetes are used to deploy the ASR system. The AKS and the EKS are used to implement the ASR system on the Azure cloud platform and AWS cloud platform respectively.

2.5 Helm

Helm is a package manager that can be used to find, publish and utilise Kubernetes applications. The applications are shared using Helm repositories that contain Helm charts, a group of files detailing a related set of Kubernetes resources [5]. The use of Helm helps to simplify the process of installing and managing Kubernetes applications as repositories can be easily installed and updated.

Chapter 3

Analysis and Design Approach

In this chapter, I will first examine the current ASR deployments and identify existing security vulnerabilities present on the deployments. Following this, I will discuss suitable solutions and improvements that can be implemented, as well as the design approach taken to mitigate these vulnerabilities.

3.1 Existing Azure ASR deployment

The following diagram shows the existing ASR deployment on the Azure cloud platform. The diagram illustrates the Kubernetes components which are relevant to security vulnerabilities and the solution that will be implemented on the deployments.
As seen in Figure 3.1 below, each service present on the cluster has its own separate public-facing endpoint that users query in order to access that service. In addition, the master and worker pods obtain environmental values from data stored as Kubernetes Secret objects. These objects are also referenced by the pods to obtain credentials used to retrieve data stored on the cloud.

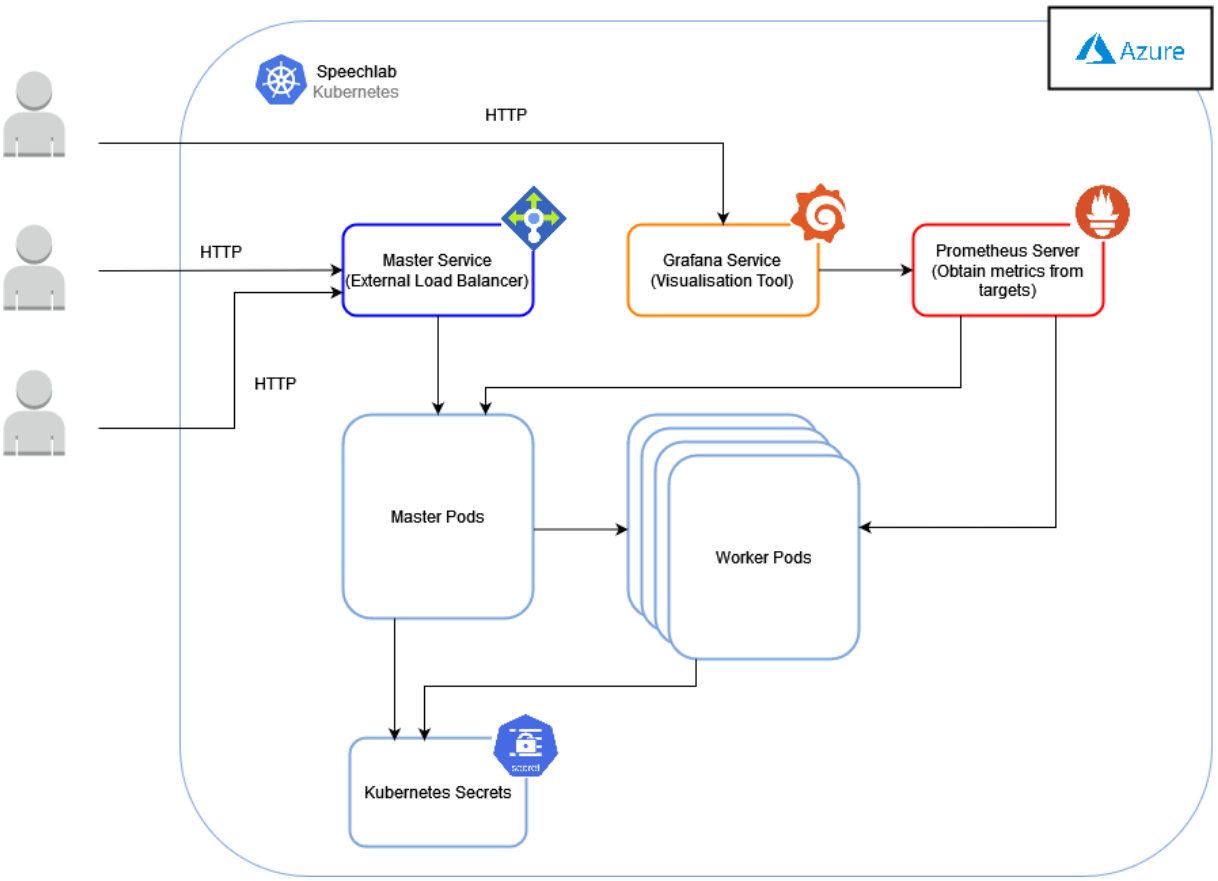

Figure 3.1: AKS existing structure

3.2 Existing AWS ASR deployment

Figure 3.2 below shows the existing AWS deployment. In this deployment, user queries pass through a single public-facing endpoint, the Bastion Host, before it is routed to the relevant services in the cluster. Similar to the Azure ASR deployment, master and worker pods also obtain environmental values from Kubernetes Secret objects.

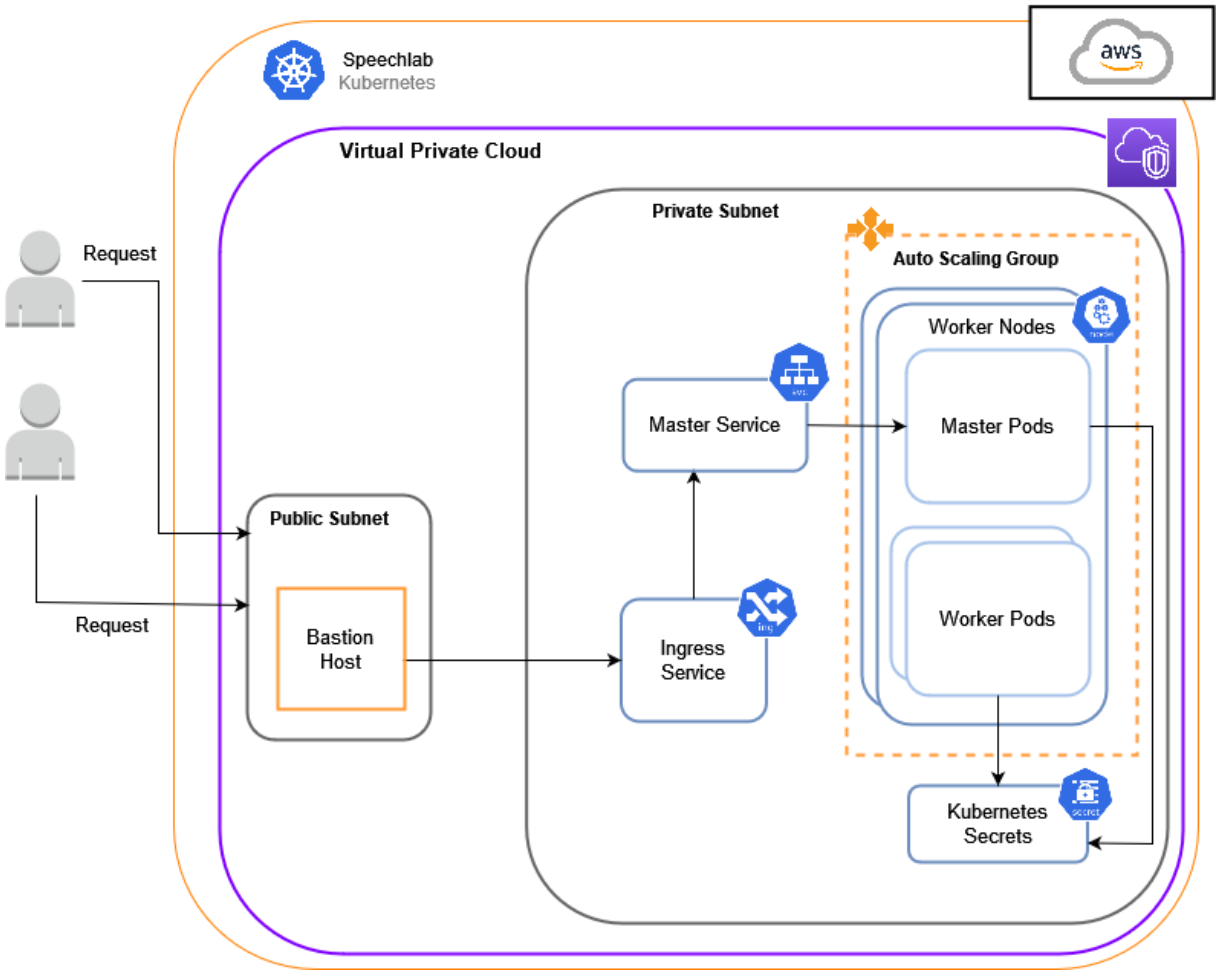

Figure 3.2: EKS existing structure

3.3 Larger attack surface and Unencrypted Communications

Having multiple public-facing endpoints on a single deployment is not ideal from a security point of view as it provides a larger attack surface [26]. Furthermore, communication between the user and the

deployment services are carried out using HTTP requests which are unencrypted and allow for a multitude of security vulnerabilities such as leakage of information [27].
A reverse proxy is a suitable solution to this vulnerability as it can filter all requests through a single public-facing endpoint before being load-balanced to the services behind it, reducing the attack surface. Furthermore, it can be configured to provide various features such as end-to-end encryption, preventing unencrypted communication between the user and the deployment [28].

3.3.1 Cluster Reverse Proxy Solutions

As seen in Figure 3.2 above, the AWS ASR deployment routes the requests through a single public-facing endpoint and already provides more secure communications. As such, this solution will only be implemented on the Azure ASR deployment.
There are various reverse proxy solutions that can be employed for an Azure deployment, with notable ones being Application Gateway Ingress Controller (AGIC) and the Nginx Ingress Controller. Both are able to meet the solution requirements of allowing traffic to be routed through a single public-facing endpoint and providing end-to-end encryption. However, one drawback of AGIC is that its deployment values cannot be modified when it is installed through AKS as an add-on [29]. Furthermore, the Nginx Ingress Controller is

heterogeneous as it can be deployed on multiple different cloud platforms [30]. Therefore, the Nginx Ingress Controller was chosen to be implemented. Figure 3.3 below illustrates how the Nginx Ingress Controller routes traffic within the cloud deployment.

Nginx Ingress Controller

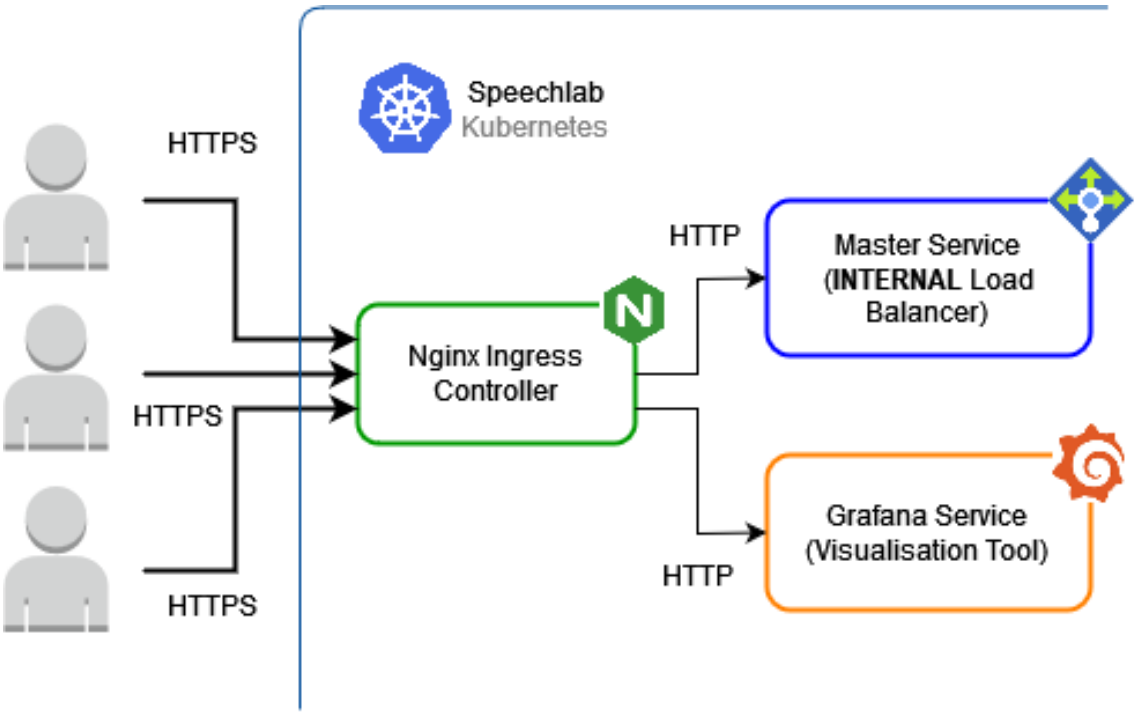

Figure 3.3: Nginx Ingress Contoller routing requests

Nginx Ingress Controller allows for TLS communication between clients and the application. It also provides the useful feature of TLS termination, which allows for HTTP communications within the cluster [31]. Initiating and maintaining a HTTPS connection re-

quires more processing and can result in higher latency within the network [32]. Therefore, by switching to a HTTP connection within the private network, low latency and fast response from the deployment are maintained. Figure 3.4 below illustrates the process of TLS termination.

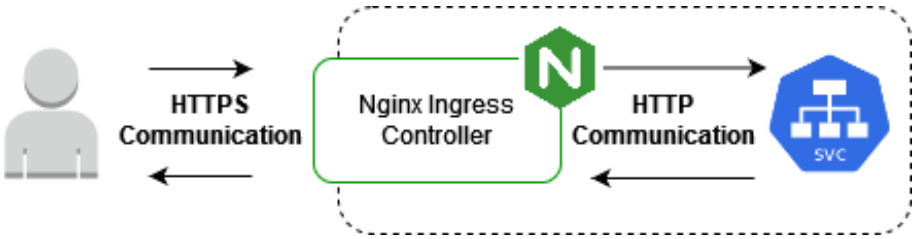

Figure 3.4: TLS Termination illustration

3.4 Vulnerable stored Secret Values

Kubernetes Secret objects store secret data in base64 format [4] which can be easily decrypted should an attacker obtain the object. This could potentially lead to the leakage of sensitive information and account credentials. As a result, the secret values require better encryption such that should the secret object be leaked, an attacker is unable to easily obtain the secret data.

3.4.1 Providing stronger Secret Encryption

There are various solutions available for providing better encryption for secret values. The official Kubernetes documentation provides a solution through encrypting secret data at rest [33]. This can be

done through specific configurations or through a Key Management Service (KMS) that manages the encryption process [34]. A key consideration is that the ASR system is deployed on the cloud, where the provider manages the Kubernetes system objects. This makes it difficult to modify and configure the system objects which is required to encrypt secret data at rest.
Another available solution is HashiCorp Vault which can be used as a third-party secret management tool. It encrypts all stored data using an encryption key and ciphers all leaving data, allowing for more secure storage and communication of secret data [35]. Furthermore, it provides heterogeneity, as it can be supported on various cloud platforms [36]. Thus, HashiCorp Vault was incorporated into the project solution.

HashiCorp Vault

HashiCorp Vault enforces secure secret data communication by having no mutual trust between the client and the Vault server [35]. A client has to set up a secure communication channel with the server by providing a client token for every request made. If a request is made without a token, only login requests are permitted. Figure 3.5 below demonstrates how HashiCorp Vault will communicate stored secret values in the cluster.

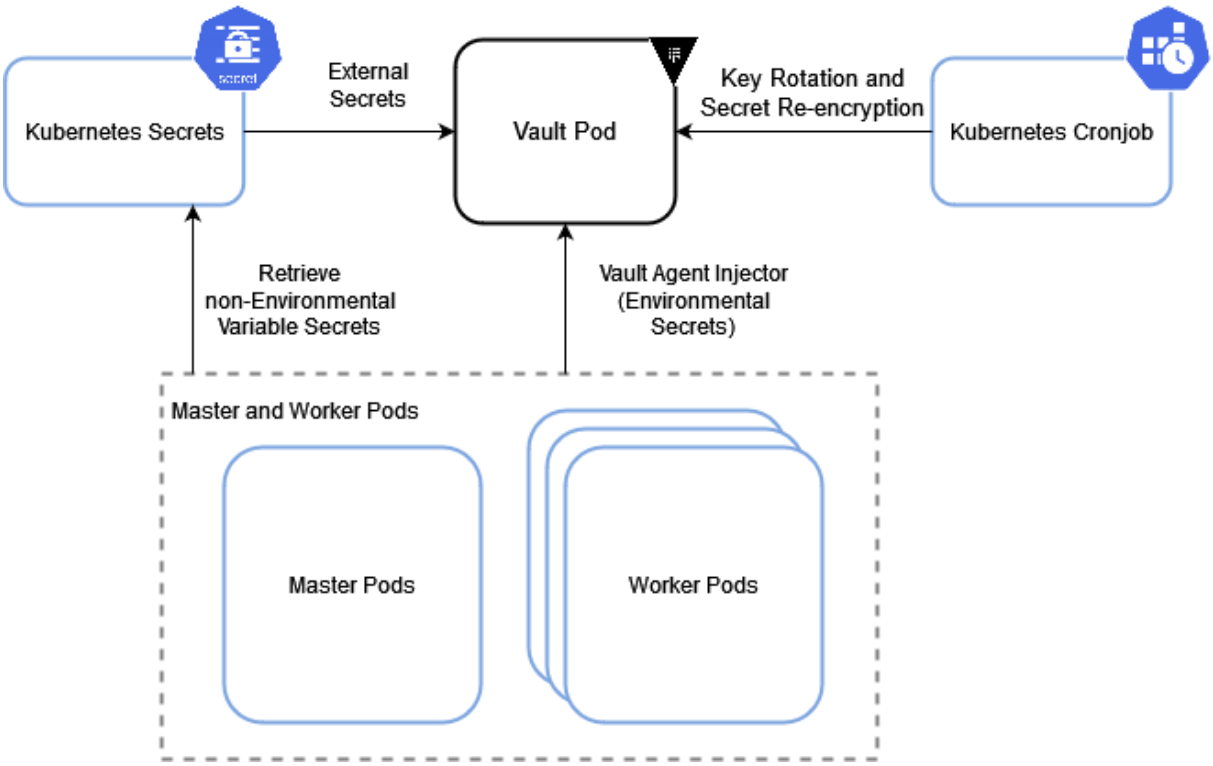

Figure 3.5: Secrets stored and managed by Vault

As shown in Figure 3.5 above, environmental secret values are directly injected into the master and worker pods from the Vault pod using Vault Agent Injector. External secrets are used to synchronise secret values that need to be referenced through Kubernetes Secrets. The Vault CSI Secrets Store driver is an alternative way to retrieve secret values from Vault and pass them onto Kubernetes Secrets. However, it has conflicting authentication configurations with Vault Agent Injector that do not allow both solutions to be used concurrently. Therefore, the External Secrets solution was chosen to be used in conjunction with the Vault Agent Injector.

3.5 Infrastructure Provisioning with Terraform

Currently, a deployment script containing a list of commands is used to create the ASR deployment. This is not ideal as the user deploying the cluster has to inspect the list of commands whenever an error occurs, or the command used is outdated and has to be updated.
Terraform, an Infrastructure as Code (IaC) tool, is a suitable solution that can quickly provision the infrastructure used in deploying the cluster. This is done using configuration files written in Terraform's high-level configuration language [37] that determine what infrastructures are created, data to be retrieved, and respective plugins to be utilised. It provides convenient functionalities such as allowing quick clean-up by tearing down the provisioned infrastructure using a single command [38].
Terraform's ability to easily provision, manage, and take down infrastructure, makes it an effective tool in testing and developing the ASR deployment. Hence, it was implemented into the project's solution as well.

3.6 Proposed Solution Workflow

After analysing the current ASR deployments and available solutions, the following applications were selected as the project's solutions:

1. Nginx Ingress Controller
2. HashiCorp Vault
3. Terraform

The proposed solution workflow will first create the deployment infrastructure using Terraform. After which, yaml configuration files will be used to create Kubernetes resources needed to deploy the relevant applications on the cluster. These applications will be installed using Helm, together with the ASR application itself. Lastly, yaml configuration files will create the remaining Kubernetes objects available through the previously installed applications.
Figure 3.6 below details the workflow of the proposed solutions:

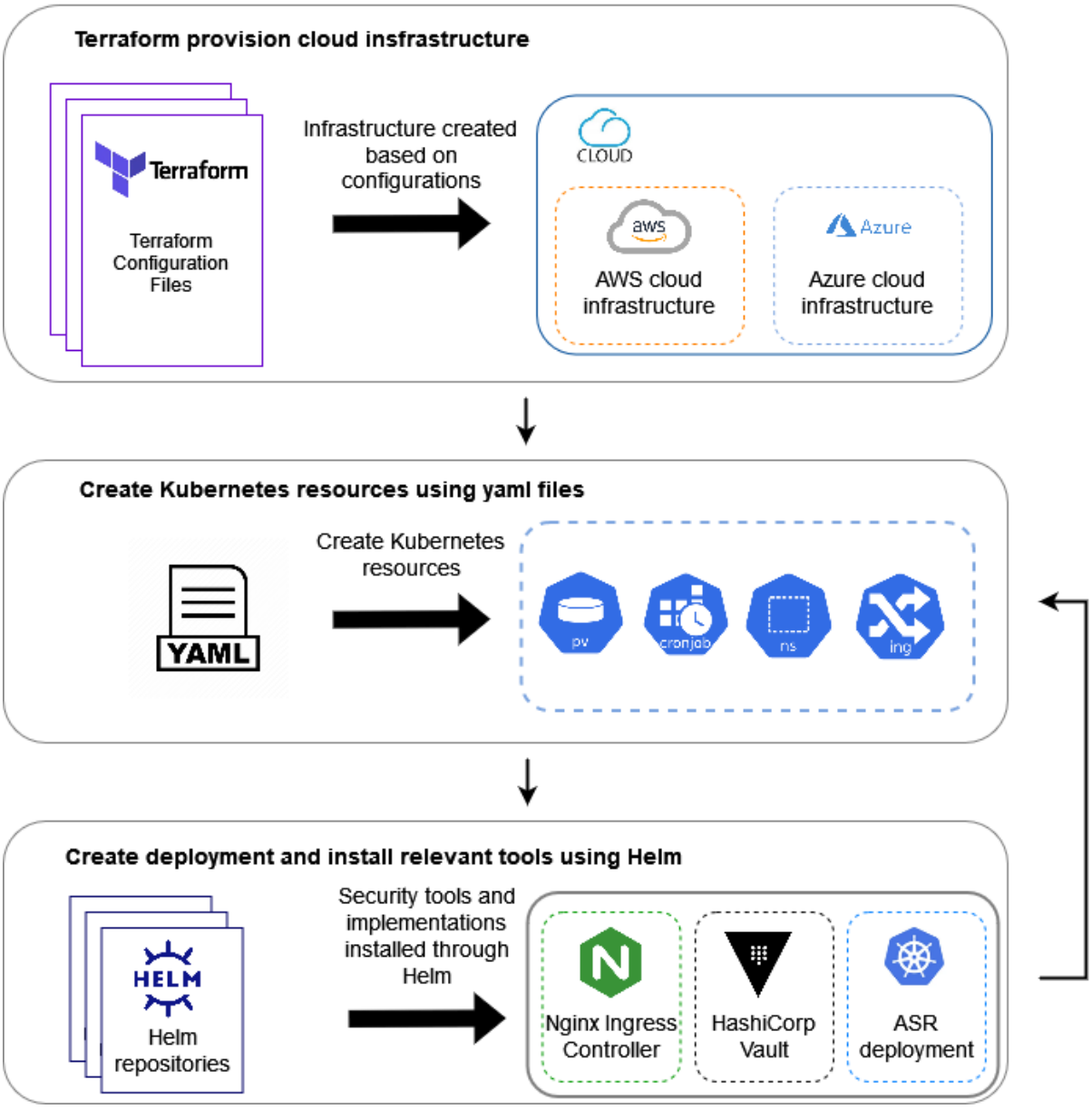

Figure 3.6: ASR solution workflow

The subsequent chapter will describe how the proposed solutions, Nginx Ingress Controller, HashiCorp Vault, and Terraform, are implemented onto the ASR deployments.

Chapter 4

Implementation

In this chapter, I will be elaborating on the project solutions and how they are implemented in the ASR deployments. The Helm tool was used to obtain the relevant files necessary to create and install the respective Kubernetes objects for these solutions.

4.1 Azure AKS Solutions

Figure 4.1 below shows the Azure ASR deployment after the project solutions have been implemented. The solutions that will be implemented are enumerated below:

1. Nginx Ingress Controller
2. HashiCorp Vault

Nginx Ingress Controller is used to create a single public-facing endpoint that will route user requests to their respective services. Vault has been implemented to store and manage secret data. The data is injected into master pods and worker pods, as well as synchronised into Kubernetes Secrets.

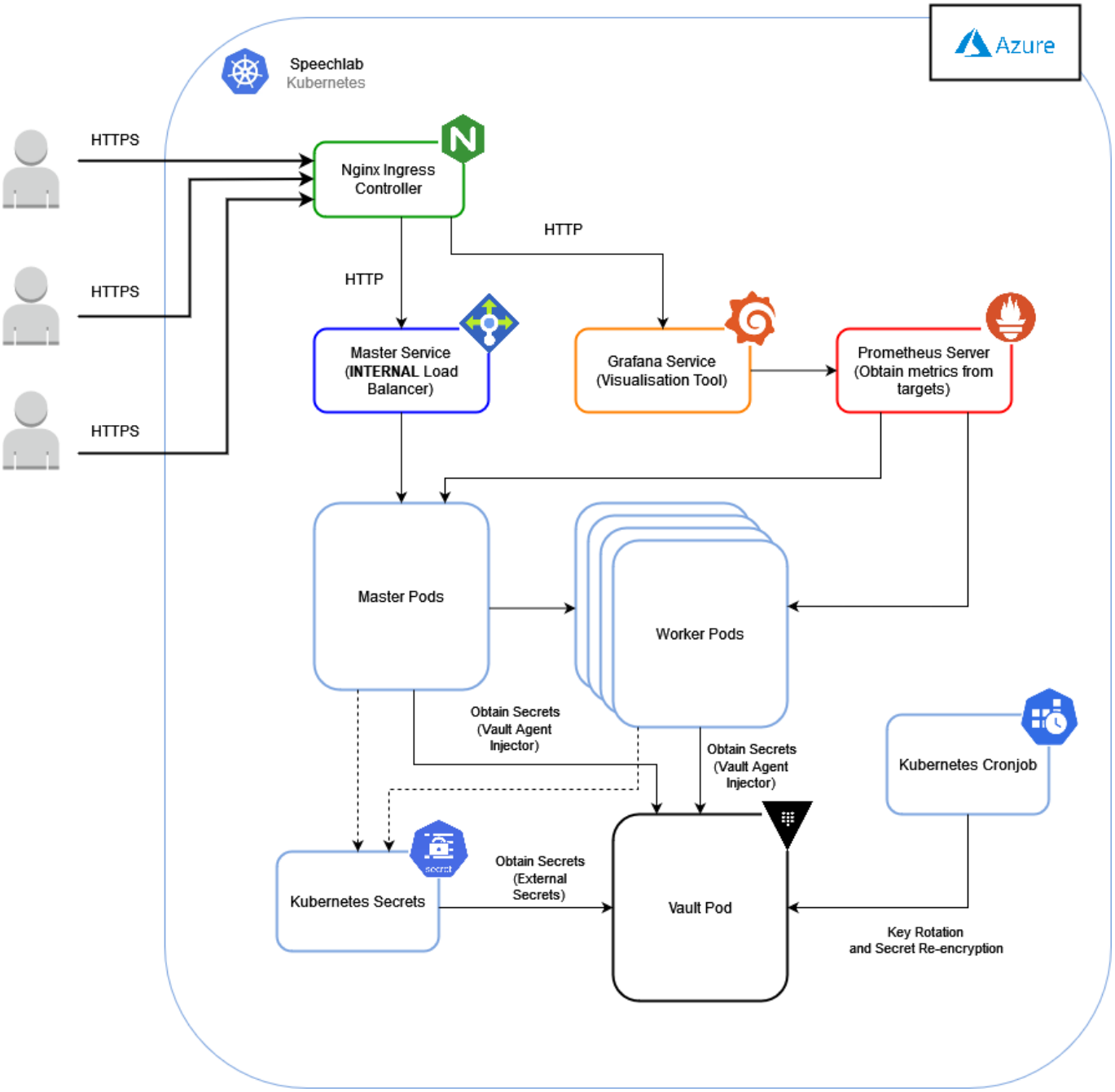

Figure 4.1: Azure solution Architecture

4.2 AWS EKS Solutions

Figure 4.2 below illustrates the AWS ASR deployment after the project solutions have been applied to it. The applied solutions are listed below:

1. HashiCorp Vault

The Vault installation onto the AWS deployment does not require the synchronising of secret data into Kubernetes Secret objects as the current deployment does not reference Kubernetes Secrets aside from environment variables. Therefore, the implementation of Vault in the EKS cluster only utilises the Vault Agent Injector to directly inject environmental secrets into the pods and does not utilise External Secret objects.

Further elaborations on the Nginx Ingress Controller and HashiCorp Vault solution implementations will be carried out in the subsequent sections.

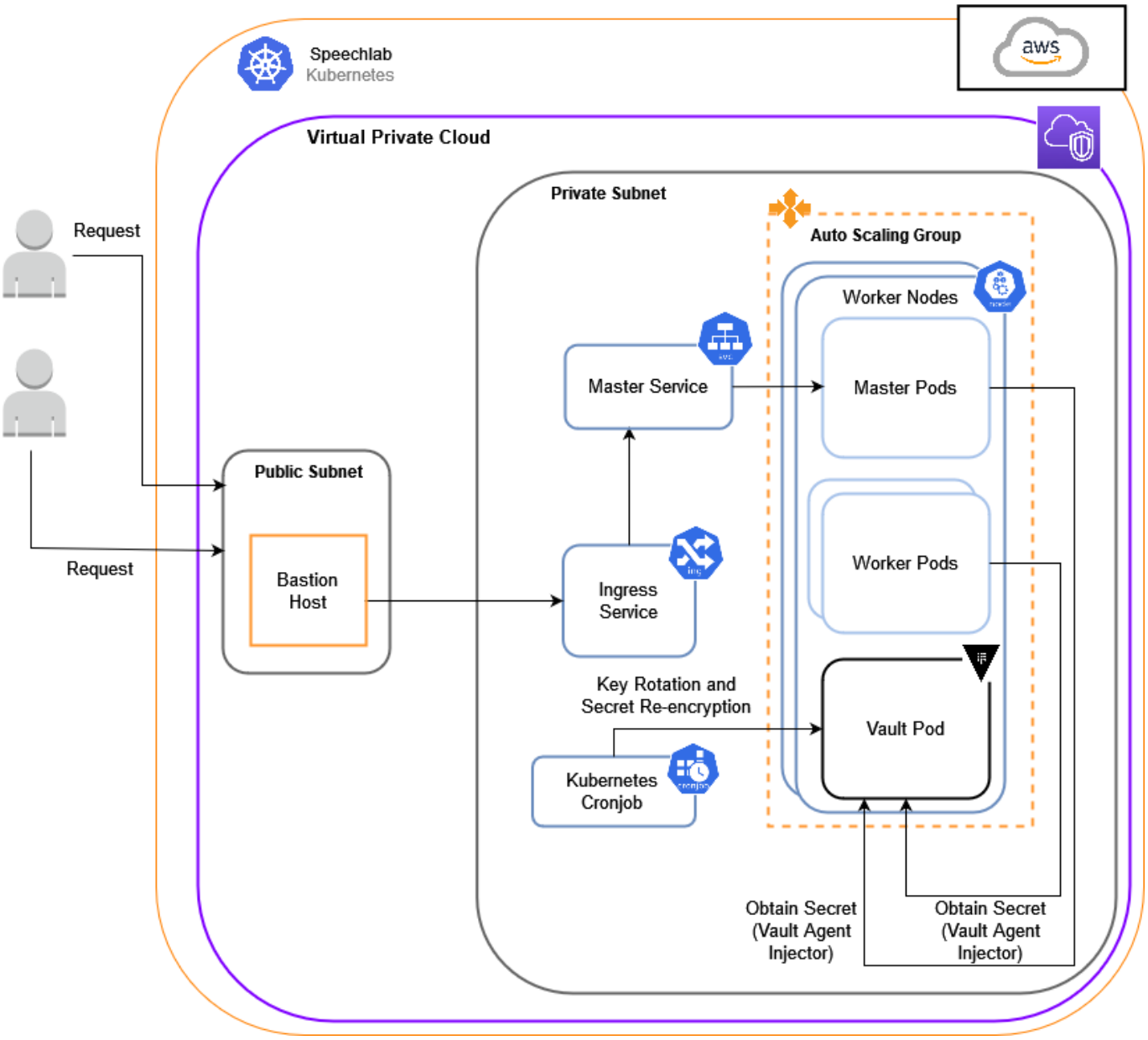

Figure 4.2: AWS solution architecture

4.3 Nginx Ingress Controller

4.3.1 Initial Set-up

The steps involved and commands used in implementing the Nginx Ingress Controller were retrieved from the Azure official documentation article titled: Create an ingress controller with a static public IP address in Azure Kubernetes Service (AKS) [39]. It includes steps and commands that were used to install the Nginx Ingress Controller on the Azure ASR deployment.

A separate namespace was created in the cluster for the Nginx Ingress Controller Kubernetes resources. Following this, the "ingress-nginx" helm repository was added and installed into the Kubernetes cluster. A static IP address Azure network resource, public-ip, was established previously in the deployment with a DNS name attached to it. The created namespace, public-ip network resource, and the attached DNS name were specified as parameters in the command used to install the Nginx Ingress Controller. By specifying the public-ip network resource and DNS name, the created Nginx Ingress Controller service will be assigned the static IP address and the DNS name. The installation of the Nginx Ingress Controller through helm created the following Kubernetes resources on the created namespace:

- nginx-ingress-ingress-nginx-controller Pod
- nginx-ingress-ingress-nginx-controller Service

- nginx-ingress-ingress-nginx-controller-admission Service
- nginx-ingress-ingress-nginx-controller Deployment
- nginx-ingress-ingress-nginx-controller Replicaset

The subsequent sections will explain the solutions implemented from installing the Nginx Ingress Controller.

4.3.2 Routing through a single endpoint

With the installation of the Nginx Ingress Controller, all external queries will now be routed through a single public-facing endpoint. Previously, the Master Service and Grafana Service each had their own external loadbalancer services as shown in Figure 4.3 below.

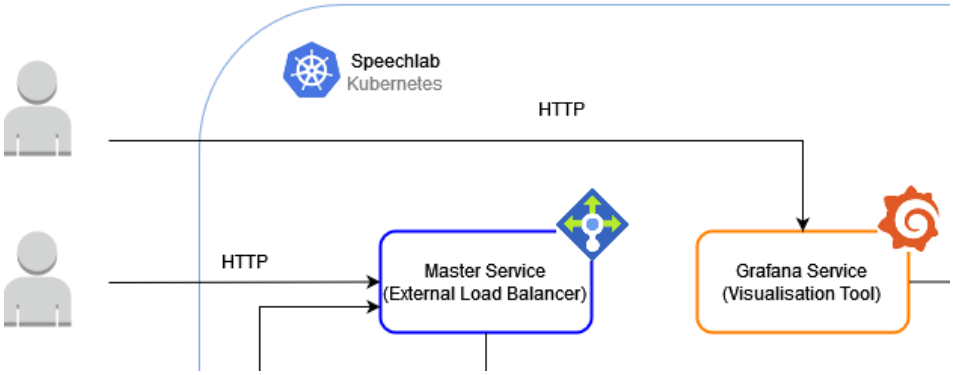

Figure 4.3: Screenshot of external loadbalancer services

With the implementation of the Nginx Ingress Controller, all queries will be filtered through the created ingress resource and routed to their respective services as seen in Figure 4.1.

Master Service Internal LoadBalancer

The Master Service was using an external loadbalancer service in order to receive queries from users. Following the Azure official documentation, an additional annotation can be included into the service creation file to configure it into an internal loadbalancer [40]. This removes the public-facing endpoint and renders the Master Service accessible only from within the cluster. A code snippet containing the annotation to be included is shown below:

```
[...]
metadata:
  name: internal-app
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
[...]
```

An ingress file was created such that requests sent to the ingress resource can be routed to the Master Service. The ingress file is configured to route requests to the Master Service when it is sent to the DNS name root path. The following code snippet shows the annotation used to set the root path:

```
[...]
metadata:
  name: master-svc-ing
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /
[...]
```

Grafana Service Ingress Routing

The Grafana Service was previously configured to use a public-facing loadbalancer to accept external queries. This was done by including the following line of code into the deployment script:

```
kubectl patch svc grafana \
     --namespace "$NAMESPACE" \
     -p '{"spec": {"type": "LoadBalancer"}}'
```

This line was removed from the deployment script for the service to remain internal and within the cluster. Referencing the Grafana official documentation to route the service behind a reverse proxy [41], the "grafana-values.yaml" file was configured such that it will allow requests to the DNS name subpath "/grafana" to be routed to the Grafana Service.

A separate ingress file was created to filter requests sent to the ingress resource and route them to the Grafana Service. A different annotation was set to allow for requests to the "/grafana" subpath to be routed to the Grafana Service. The following code snippet shows the annotation set in the grafana ingress file:

```
[...]
metadata:
  name: grafana-ing
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /$1
[...]
```

As a result of routing both the Master Service and Grafana Service through the ingress resource, all requests made to the Azure ASR deployment are routed through a single public-facing endpoint. This effectively reduces the attack surface on the deployment as well as allows for easier monitoring of network traffic into the deployment.

4.3.3 TLS end-to-end encryption

End-to-end encryption was set up on the Nginx Ingress Controller by utilising a cert-manager that manages and automatically provisions Let's Encrypt certificates [39]. Let's Encrypt is a free automated open certificate authority that provisions certificates used in TLS encryption. In order to carry out this process, the "jetstack" helm repository was added and installed onto the Azure ASR deployment. After "jetstack" had been installed using helm, a ClusterIssuer Kubernetes object was deployed within the previously created namespace using a yaml file. The ClusterIssuer object was created to issue certificates for the ingress resources. The following annotation was added into both the Master Service and Grafana Service ingress file to provision the certificates and provide TLS encryption when sending requests to the ingress resource:

```
[...]
  annotations:
    kubernetes.io/ingress.class: nginx
    cert-manager.io/cluster-issuer: letsencrypt-prod
[...]
```

TLS Termination

The provisioned ClusterIssuer Kubernetes object also provides TLS termination. TLS termination is enabled in the Nginx Ingress Controller through the utility of a cert-manager. External requests to the deployment is done over a HTTPS connection. By using TLS termination, once the request is within the private network, it is propagated using a HTTP connection as shown in Figure 4.1 above.

The ingress files were configured and updated accordingly such that a HTTPS connection, together with TLS termination, is established for the deployment services.

4.4 HashiCorp Vault

4.4.1 Initial Set-up

The official documentation for HashiCorp Vault specifies the steps to be taken for Vault to be installed and deployed on both the Azure cloud platform [42] and the AWS cloud platform [43].
The helm repository "hashicorp" was added and installed for Vault to be installed onto the respective ASR deployments. Once the repository was installed, the following Kubernetes resources were created within the clusters:

- vault-0 Pod
- vault-agent-injector Pod

- vault Service
- vault-agent-injector-svc Service
- vault-internal Service
- vault-agent-injector Deployment
- vault-agent-injector ReplicaSet

The vault-0 pod contains the Vault instance and is sealed upon installation of the Vault. The initiation command can be run to obtain the unseal keys and login token that will be used to access the Vault application. Once the Vault application is accessed, it will be configured to create the secret store and the secret data will be stored onto the Vault instance.
A Read Access Policy will be written onto the Vault application that will allow for the reading of Vault secret values. A service account name will be authenticated with the Read Access Policy to allow for Pods with the service account name attached to it to possess the Read Access Policy's listed capabilities. The last step in the initial set-up is to create the specified Kubernetes ServiceAccount on the cluster.

The following sections will describe how Vault provides secret management and better encryption for the secret data used within the cluster.

4.4.2 Inject Vault Secrets into Kubernetes Pods using Vault Agent Injector

HashiCorp Vault possesses a Vault Agent Injector functionality that can insert Vault Agent containers into pods. These containers use Vault Agent Templates to provide Vault secret values through a shared memory volume mounted on the pod. This allows other pod containers to retrieve secret values from Vault without being Vault aware [44].

The volume is mounted to the path "/vault/secrets/" in the container, with the secret values being determined by the templates specified in the container annotations as seen in the code snippet below:

```
[...]
    annotations:
      vault.hashicorp.com/agent-inject: 'true'
      vault.hashicorp.com/role: 'vault_role'
      vault.hashicorp.com/agent-inject-secret-config: 'secret/data/vault_path'
      vault.hashicorp.com/agent-inject-template-config: |
        {{ with secret "secret/data/vault_path" -}}
          export ENV_SECRET_KEY="{{ .Data.data.VAULT_SECRET_KEY }}"
        {{- end }}
      vault.hashicorp.com/agent-pre-populate-only : "true"
[...]
```

These annotations are specified in the master and worker pods' deployment file. The Kubernetes ServiceAccount authenticated with

the Vault's Read Access Policy will also be specified within the pods' deployment file to allow the pod to read the respective Vault values. The following command will be included in the container's command specifications:

```
[...]
        command: ['bash', '-c', "source /vault/secrets/config && ..."
[...]
```

By injecting the template into the container through the shared memory volume, the source command can be used to read and execute the template file. This directly instantiates the environment variables into the containers [45], allowing the container to use these values without referencing them through Kubernetes Secrets.

4.4.3 Synchronise Vault Secrets onto Kubernetes using External Secrets

The External Secrets tool uses an Operator that can consume data from a third-party service such as Vault and inject the read data as Kubernetes Secrets. The External Secrets tool is only implemented on the Azure ASR deployment as the AWS ASR deployment only references Vault secret values through environmental variables, which is done using Vault Agent Injector.

External Secrets are installed onto the cluster using Helm. The "external-secrets" helm repository is added and installed with the Vault internal IP address and authentication service account being

specified as fields in the installation command. ExternalSecret objects are subsequently created using yaml specification files. The specification files include the type of information retrieved by the ExternalSecret object, the Vault role used to retrieve the data, and the path where the secret is stored within Vault. A code snippet detailing the required specifications is shown below:

```
[...]
spec:
  backendType: vault
  vaultMountPoint: kubernetes
  vaultRole: vault_role
  dataFrom:
    - secret/data/secret_path
[...]
```

The created ExternalSecrets Kubernetes objects are used to synchronise Vault secret values into Kubernetes Secrets. This enables pods to reference necessary secret values from Vault through Kubernetes Secrets.

4.4.4 Rotating encryption keys and Re-encrypting Secrets

HashiCorp Vault has a functionality for automatically rotating secret encryption key [46]. When the key is rotated, the previously-stored secrets are not re-encrypted and can still be decrypted with the old encryption key.

HashiCorp Vault does not have a functionality for the re-encryption of secrets after a key has been rotated. However, this can be carried out using internal HTTP API calls to obtain the secret, delete the secret data, and recreate the secret using the newly rotated encryption key. This process can be automated by using Kubernetes CronJobs that can carry out specified commands within the cluster at a set interval. Table 4.1 below shows the interval that can be set for the automatic Vault key rotation and Kubernetes CronJobs.

Rotation functionality	Tool	Set Intervals
Vault key rotation	HashiCorp Vault	Rotate after a fixed number of hours (minimal value of 24h)
Kubernetes CronJobs	Kubernetes	Rotate by set minute, hour, day of the month, month, and day of the week

Table 4.1: Comparing Vault key rotation and Kubernetes CronJobs

The Vault key rotation intervals are set by a number of hours [47] while Kubernetes CronJob intervals are denoted by specified periods in a month [48]. The re-encryption of secret data has to be done at the same interval after the encryption key has been rotated. As the set intervals for Vault key rotation and Kubernetes CronJobs differ, they cannot be executed within the same fixed regular intervals.
The Vault key rotation functionality can also be executed using HTTP API calls, allowing it to be specified as a command and executed through Kubernetes CronJobs as well. By having HTTP API calls for both automatic key rotation and secret re-encryption, the

key can be rotated at a regular interval with the secrets being re-encrypted by the new key at the same fixed interval.

4.5 Additional Implementations

Following the implementation of the Nginx Ingress Controller and HashiCorp Vault solutions, additional configurations and implementations were made to improve security as well as streamline the deployment process. The additional implementations are listed below:

- Terraform
- Encryption of Elastic File System (EFS) at rest
- Azure host-based encryption

4.5.1 Terraform

4.5.1.1 Initial Set-up

Terraform is installed following the installation article from its official documentation titled: Install Terraform [49]. The IaC tool is installed using the apt-get command-line tool.
First, the relevant Gnu Privacy Guard (GPG) packages are installed using apt-get. GPG keys are used to provide secure transmission of data and ensure that it arrives from a genuine source. After which, apt-get is used to obtain the HashiCorp GPG key together

with the official HashiCorp Linux repository. Once the repository is added and updated, Terraform Command-Line Interface (CLI) is installed.

4.5.1.2 Deploying Infrastucture as Code (IaC)

The Terraform configuration files are written and placed within a "/terraform" directory. The main Terraform commands [50] used in executing the tool's functionalities are listed below:

- **terraform init** - Used to set the Terraform working directory
- **terraform validate** - Ensure the configuration files are valid
- **terraform plan** - Display the changes from applying the configuration files
- **terraform apply** - Update or create the relevant infrastructure
- **terraform destroy** - Tear down the provisioned infrastructure

Through the use of the Terraform configuration files, the specified cloud infrastructures can be easily managed, created, and taken down after sufficient testing and development have been done on the deployment.

4.5.2 Encryption of Elastic File System (EFS) at rest

The Elastic File System (EFS) service available on the AWS cloud platform is a network file system (NFS) used to store the speech recognition models used in the ASR application. The current implementation of the AWS ASR deployment has the EFS unencrypted at rest. Having an encrypted file system will automatically encrypt both data and metadata before it is written to the EFS, reducing the risk of compromising data confidentiality [51].
The process of encrypting the EFS is carried out during the deployment of the ASR system. As the EFS is created using Terraform, the following specification is included into the Terraform configuration file to encrypt the EFS at rest:

```
[...]
# create efs resource
resource "aws_efs_file_system" "speechefs" {
  tags = {
    Name = "speech_efs"
  }
  encrypted = true
}
[...]
```

Including the specification "encrypted=true" will create an encrypted EFS file system when Terraform is used to deploy the ASR infrastructure. The encryption process is handled by the AWS cloud

provider which will encrypt the data and metadata stored within the EFS using industry-standard AES-256 encryption algorithm [51]. The AWS cloud provider will also manage and automatically decrypt the data when it is presented to the application.

4.5.3 Azure host-based encryption

The Azure cloud platform supports host-based encryption, which allows data stored in created VMs on the AKS cloud platform to be encrypted at rest. It also encrypts the data that is communicated from the VM hosts to storage services [52].
Azure host-based encryption is carried out during the deployment phase of the Azure ASR application. In order to apply host-based encryption onto the Azure hosts, the Azure feature "EncryptionAtHost" has to be added. The following command is included in the deployment script to apply the feature:

```
az feature register --namespace "Microsoft.Compute" --name "EncryptionAtHost"
```

When the command to create the Kubernetes cluster using AKS is executed, the following parameter has to be included for host-based encryption to be applied:

```
az aks create \
    --resource-group $RESOURCE_GROUP \
    --name $KUBE_NAME \
    --node-vm-size Standard_B4ms \
    --kubernetes-version $KUBE_VERSION \
    --enable-encryption-at-host \
    --zones 1 2 3 --load-balancer-sku standard
```

The included "–enable-encryption-at-host" parameter creates the specified AKS cluster with host-based encryption applied. The encryption process is done using platform managed keys [52], as such no further implementations were required. Encrypting communicated data and data at rest better enforces data confidentiality and reduces the risk of losing sensitive data.

Chapter 5

Experiments and Results

The security solutions were implemented to make the deployment more secure, however, we still needed to guarantee that other aspects of the deployment, such as the ASR system's low latency and quick response to client queries, were maintained. HTTPS connections have higher overhead costs than HTTP connections, which was one concern over introducing TLS [32]. This could result in an increased latency when processing a client's request.

The TLS termination feature available in the Nginx Ingress Controller solution is thus implemented such that a HTTPS connection is established when a client sends a request to the public-facing endpoint but the communication within the secure private network itself is done over a HTTP connection. This helps to reduce the overall overhead costs resulting from establishing a HTTPS connection.

An experiment was conducted to examine the impact experienced by the application when TLS and TLS termination is implemented onto the ASR deployment.

5.1 TLS Termination resultant latency

Experiment

The experiment was conducted by sending client requests over two separate ingress resources. The first ingress resource uses a normal HTTP connection when communicating with the client while the second ingress resource uses TLS to establish a HTTPS connection with the client and TLS termination to have HTTP communications within the private network.

The "send_multi_ws_audio_files.sh" script was used to send multiple client requests to the application simultaneously. Both ingress resources were tested with increasing amounts of load in order to determine what is the latency difference experienced by the application under different loads for the two different connections. The time difference from when the socket is first opened to when the last request is processed was collected from both resources, and the period between the two results was taken to be the time needed to process the client requests.

The client request loads that were used to test the applications are 1, 10, 25, 50, and 100 client requests. The client requests were sent to the application under the assumption that each client request is sent 5 seconds after the previous request has been sent to simulate real requests being received by the application.

Results

Table 5.1 below shows the various timestamps gathered from when the server socket is first opened to when the last request has been processed. The timestamps collected are from each connection type and its load experienced as seen below:

Number of requests/Connection type	Start Time	End Time
1 / HTTP	06:07:39,610	06:07:45,704
1 / HTTPS	06:13:03,570	06:13:09,673
10 / HTTP	06:54:56,669	06:55:47,749
10 / HTTPS	06:56:00,896	06:56:52,012
25 / HTTP	06:59:19,338	07:01:25,459
25 / HTTPS	07:19:06,589	07:21:12,751
50 / HTTP	07:38:14,042	07:42:25,201
50 / HTTPS	07:53:16,090	07:57:27,296
100 / HTTP	08:00:09,673	08:08:30,916
100 / HTTPS	08:19:13,843	08:27:35,172

Table 5.1: Start to end timestamps under different loads

The following Table 5.2 illustrates the time taken, and the time difference for each connection to finish processing their respective loads:

Number of requests	HTTP	HTTPS	Latency Difference
Single request	6s 094ms	6s 103ms	9ms
10 requests	51s 080ms	51s 116ms	36ms
25 requests	2mins 6s 121ms	2mins 6s 162ms	41ms
50 requests	4mins 11s 159ms	4mins 11s 206ms	47ms
100 requests	8mins 21s 243ms	8mins 21s 329ms	86ms

Table 5.2: Load processing time and connection latency difference

Observations

It can be observed that there is a slight latency increase of 9ms when a single request is transcribed over a HTTPS connection instead of a HTTP connection. For the subsequent loads of 10, 25, 50, and 100 requests, the latency differences were within a similar range of 30 to 90ms as seen in Table 5.2 above.

It was inferred that the latency difference between the different request loads is similar because the time taken for a request to be sent over HTTP or HTTPS is a few milliseconds, which is much quicker than the period between successive requests. As such, the additional latency incurred by sending larger loads of requests over a HTTPS connection does not contribute to the overall time taken to receive and process the respective loads of requests. Furthermore, as TLS termination is enabled, the time needed to transcribe requests for both the HTTP and HTTPS connections are the same. Hence, due to the two aforementioned factors, the latency differences for the lar-

ger loads of requests are within a similar range.

Following these findings, it can be seen that although there is a slight latency increase when requests are sent over HTTPS, it is of a very small value that has very little effect on the overall time taken to process the respective loads of requests. Therefore, setting up TLS for the application through Nginx Ingress Controller meets the application's requirements of maintaining low latency and quick response to client queries.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The objective of this project was to employ hardening techniques to improve the security of the ASR systems deployed on the cloud.
The architecture of the previous ASR systems was illustrated and analysed to identify exploitable vulnerabilities, as well as suitable solutions that could be applied to the deployments. The Nginx Ingress Controller and HashiCorp Vault applications were implemented as they were the most appropriate solutions that meet the security requirements of the ASR deployment.
Nginx Ingress Controller fulfilled the objectives of reducing the attack surface on the deployment and providing TLS end-to-end encryption between the clients and the application. This allowed the ASR deployments to be less susceptible and more resilient against external cyber attacks.
HashiCorp Vault met the objectives of providing better encryption for secret values and reducing the visibility of confidential data. As a result, it is harder to identify secret data and better ensures data confidentiality.

During the course of the project, requirements beyond the scope of security were also met through the utility of applications such as Terraform, which streamline and increased the efficiency of creating and tearing down ASR deployments on cloud and testing platforms.

6.2 Future Work

Following the solutions that are applied in this project, other security implementations can be employed in the future to further secure the ASR deployments.

6.2.1 Red Hat OpenShift

Red Hat OpenShift is an enterprise-grade Kubernetes platform that provides services enhancing security and performance [53]. Currently, the ASR system image is built using the "debian:10.1" base image. By utilising Red Hat OpenShift base images, the created container image layers can be encrypted and decrypted using private-public key pairs [54]. This decreases the image's information vulnerability as the data cannot be retrieved without possessing the appropriate decryption keys.
Red Hat OpenShift provides many other security features that can further enhance the deployment's security such as vulnerability and configuration management, risk profiling, and detection and response measures [55].

6.2.2 Service Mesh

A service mesh is a flexible, dependable, and quick communication infrastructure layer that can regulate how various components of an application communicate data to one another [56]. The security solutions currently employed on the ASR deployments are focused on encrypting communications outside of the application and encrypting data stored at rest. Through the provisioning of a service mesh, additional security measures can be applied to the data communicated within the application as well. The service mesh can provide security features such as encrypting data communicated within the application and authentication and authorisation for communications between application components [56].

One consideration that needs to be accounted for is that adding security features within the application may result in an increase in latency for processing client requests. Appropriate testing should be conducted to determine whether this solution meets the application requirements before applying it to the ASR deployments.

Bibliography

[1] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer and Karel Vesely. "The Kaldi speech recognition toolkit". In: *IEEE 2011 workshop on automatic speech recognition and understanding*. CONF. IEEE Signal Processing Society. 2011.

[2] Chuanqi Kan. "DoCloud: An elastic cloud platform for Web applications based on Docker". In: *2016 18th international conference on advanced communication technology (ICACT)*. IEEE. 2016, pp. 478–483.

[3] Nguyen Nguyen and Taehong Kim. "Toward highly scalable load balancing in Kubernetes clusters". In: *IEEE Communications Magazine* 58.7 (2020), pp. 78–83.

[4] The Kubernetes Authors. *Secrets*. 2021. URL: `https://kubernetes.io/docs/concepts/configuration/secret/` (visited on 26th December 2021).

[5] Helm Authors. *Charts*. 2021. URL: `https://helm.sh/` (visited on 11th January 2022).

[6] IBM Cloud Education. *What is containerization*. 2019. URL: `https://www.ibm.com/sg-en/cloud/learn/containerization/` (visited on 21st February 2022).

[7] IBM Cloud Education. *What is virtualization*. 2019. URL: https://www.ibm.com/sg-en/cloud/learn/virtualization-a-complete-guide/ (visited on 21st February 2022).

[8] Mathijs Jeroen Scheepers. "Virtualization and containerization of application infrastructure: A comparison". In: *21st twente student conference on IT*. Vol. 21. 2014.

[9] Docker Inc. *What is a Container?* 2022. URL: https://www.docker.com/resources/what-container/ (visited on 23rd February 2022).

[10] Claus Pahl, Antonio Brogi, Jacopo Soldani and Pooyan Jamshidi. "Cloud container technologies: a state-of-the-art review". In: *IEEE Transactions on Cloud Computing* 7.3 (2017), pp. 677–692.

[11] Red Hat Inc. *What is a container registry?* 2020. URL: https://www.redhat.com/en/topics/cloud-native-apps/what-is-a-container-registry (visited on 23rd February 2022).

[12] Steve Buchanan, Janaka Rangama and Ned Bellavance. "Container registries". In: *Introducing Azure Kubernetes Service*. Springer, 2020, pp. 17–34.

[13] Docker Inc. *Dockerfile reference*. 2022. URL: https://docs.docker.com/engine/reference/builder/ (visited on 23rd February 2022).

[14] The Kubernetes Authors. *What is Kubernetes?* 2021. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 24th February 2022).

[15] VMWare Inc. *What is Kubernetes infrastructure*. 2022. URL: https://www.vmware.com/topics/glossary/content/kubernetes-infrastructure.html (visited on 25th February 2022).

[16] The Kubernetes Authors. *Kubernetes Components*. 2022. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 25th February 2022).

[17] Red Hat Inc. *What is YAML?* 2021. URL: https://www.redhat.com/en/topics/automation/what-is-yaml (visited on 6th March 2022).

[18] The Kubernetes Authors. *Pods*. 2021. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 25th February 2022).

[19] The Kubernetes Authors. *Deployments*. 2022. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visited on 25th February 2022).

[20] The Kubernetes Authors. *Service*. 2022. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 25th February 2022).

[21] The Kubernetes Authors. *Ingress*. 2022. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/ (visited on 25th February 2022).

[22] The Kubernetes Authors. *Ingress Controllers*. 2022. URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/ (visited on 25th February 2022).

[23] The Kubernetes Authors. *CronJob*. 2021. URL: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/ (visited on 25th February 2022).

[24] The Kubernetes Authors. *Jobs*. 2022. URL: https://kubernetes.io/docs/concepts/workloads/controllers/job/ (visited on 25th February 2022).

[25] Lizhe Wang, Gregor Von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao and Cheng Fu. "Cloud computing: a perspective study". In: *New generation computing* 28.2 (2010), pp. 137–146.

[26] Pratyusa K Manadhata and Jeannette M Wing. "An attack surface metric". In: *IEEE Transactions on Software Engineering* 37.3 (2010), pp. 371–386.

[27] Suphannee Sivakorn, Iasonas Polakis and Angelos D Keromytis. "The cracked cookie jar: HTTP cookie hijacking and the exposure of private information". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 724–742.

[28] Peter Sommerlad. "Reverse Proxy Patterns." In: *EuroPLoP*. 2003, pp. 431–458.

[29] Microsoft Azure. *Installation Guide*. 2022. URL: https://docs.microsoft.com/en-us/azure/application-gateway/ingress-controller-overview (visited on 12th January 2022).

[30] NGINX Inc. *Installation Guide*. 2022. URL: https://kubernetes.github.io/ingress-nginx/deploy/ (visited on 12th January 2022).

[31] Microsoft Azure. *Overview of TLS termination and end to end TLS with Application Gateway*. 2022. URL: https://docs.microsoft.com/en-us/azure/application-gateway/ssl-overview (visited on 12th January 2022).

[32] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki and Peter Steenkiste. "The cost of the" s" in https". In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 2014, pp. 133–140.

[33] The Kubernetes Authors. *Encrypting Secret Data at Rest*. 2021. URL: https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/ (visited on 31st December 2021).

[34] The Kubernetes Authors. *Using a KMS provider for data encryption*. 2022. URL: https://kubernetes.io/docs/tasks/administer-cluster/kms-provider/ (visited on 12th January 2022).

[35] HashiCorp. *Security Model*. 2022. URL: https://www.vaultproject.io/docs/internals/security (visited on 12th January 2022).

[36] HashiCorp. *Extend Vault across your IT environment*. 2022. URL: https://www.hashicorp.com/products/vault (visited on 12th January 2022).

[37] HashiCorp. *Terraform Language Documentation*. 2022. URL: https://www.terraform.io/language (visited on 9th January 2022).

[38] HashiCorp. *Command: destroy*. 2022. URL: https://www.terraform.io/cli/commands/destroy (visited on 9th January 2022).

[39] Microsoft Azure. *Create an ingress controller with a static public IP address in Azure Kubernetes Service (AKS)*. 2021. URL: https://docs.microsoft.com/en-us/azure/aks/ingress-static-ip (visited on 3rd January 2022).

[40] Microsoft Azure. *Use an internal load balancer with Azure Kubernetes Service (AKS)*. 2021. URL: https://docs.microsoft.com/en-us/azure/aks/internal-lb (visited on 4th January 2022).

[41] Grafana Labs Team. *Run Grafana behind a reverse proxy*. 2022. URL: https://grafana.com/tutorials/run-grafana-behind-a-proxy/ (visited on 4th January 2022).

[42] HashiCorp. *Vault Installation to Azure Kubernetes Service via Helm*. 2022. URL: https://learn.hashicorp.com/tutorials/vault/kubernetes-azure-aks?in=vault/kubernetes (visited on 5th January 2022).

[43] HashiCorp. *Vault Installation to Amazon Elastic Kubernetes Service via Helm*. 2022. URL: https://learn.hashicorp.com/tutorials/vault/kubernetes-amazon-eks?in=vault/kubernetes (visited on 5th January 2022).

[44] HashiCorp. *Agent Sidecar Injector*. 2022. URL: https://www.vaultproject.io/docs/platform/k8s/injector (visited on 10th January 2022).

[45] HashiCorp. *Vault Agent Injector Examples*. 2022. URL: https://www.vaultproject.io/docs/platform/k8s/injector/examples (visited on 10th January 2022).

[46] HashiCorp. *Key Rotation*. 2022. URL: https://www.vaultproject.io/docs/internals/rotation (visited on 12th January 2022).

[47] HashiCorp. *Configure Automatic Key Rotation*. 2022. URL: https://www.vaultproject.io/api-docs/system/rotate-config (visited on 9th January 2022).

[48] The Kubernetes Authors. *CronJob*. 2022. URL: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/ (visited on 9th January 2022).

[49] HashiCorp. *Install Terraform*. 2022. URL: https://learn.hashicorp.com/tutorials/terraform/install-cli (visited on 9th January 2022).

[50] HashiCorp. *Basic CLI Features*. 2022. URL: https://www.terraform.io/cli/commands (visited on 9th January 2022).

[51] Amazon Web Services Inc. *Encrypting data at rest*. 2022. URL: https://docs.aws.amazon.com/efs/latest/ug/encryption-at-rest.html (visited on 26th February 2022).

[52] Microsoft Azure. *Enable host-based encryption on Azure Kubernetes Service (AKS)*. 2022. URL: https://soft.com/en-us/azure/aks/enable-host-encryption (visited on 26th February 2022).

[53] Red Hat Inc. *What is Red Hat OpenShift*. 2022. URL: https://cloud.redhat.com/learn/what-is-openshift (visited on 6th March 2022).

[54] IBM Cloud Kubernetes Service. *Deploy Containers From Encrypted Images in Red Hat OpenShift on IBM Cloud Clusters*. 2021. URL: https://www.ibm.com/cloud/blog/deploy-containers-from-encrypted-images-in-red-hat-openshift-on-ibm-cloud-clusters (visited on 6th March 2022).

[55] Red Hat Inc. *Red Hat Advanced Cluster Security for Kubernetes*. URL: https://www.redhat.com/en/technologies/cloud-computing/openshift/advanced-cluster-security-kubernetes (visited on 6th March 2022).

[56] Red Hat Inc. *What's a service mesh?* 2018. URL: https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh (visited on 6th March 2022).