



**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

**Nanyang Technological University
School of Computer Science and Engineering**

Final Year Project

On-premise Cloud Deployment of ASR System

Final Report

by

Yap Zher Hao, Jonathan (U1922422K)

Supervisor: Associate Professor Chng Eng Siong

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

Abstract

This project aims to provide a bare metal deployment to the Automatic Speech Recognition (ASR) system. The deployment is made on a bare metal system to ensure that there is an option for an on-premise deployment for the ASR. This deployment solution is constructed using containerisation technology, Docker and Kubernetes. The solution is then implemented on on-premise servers in Nanyang Technological University.

This solution ensures the auto-scaling of the system and has an on-premise load balancer to ensure that the system is able to meet the needs of the incoming requests. This report presents the deployment solution in terms of the core technologies used, the design of the architecture, implementation of the solution as well as the multitude of errors faced in the implementation of the solution.

Acknowledgements

This project would not have been possible without the invaluable support and guidance of the following individuals, whose unwavering assistance was instrumental to its success. I wish to express my sincerest gratitude and appreciation to these individuals.

First and foremost, I would like to express my appreciation to my supervisor, Associate Professor Chng Eng Siong for allowing me the chance to take up this project even though I had little knowledge on the topic of the project. Throughout the course of this project, I have acquired a wealth of knowledge and gained experience in areas that were previously inaccessible to me.

Secondly, I would like to express my gratitude to Research Staff Vu Thi Ly for her invaluable mentorship during the design and implementation of this project. Her guidance was instrumental in helping me navigate through difficult moments and in directing my efforts towards achieving the project's objectives. Her insights and advice on identifying and addressing the root causes of issues were invaluable in enhancing my problem-solving skills.

Lastly, I would like to thank my family and friends for their support throughout the project.

Contents

| | |
|---|----------|
| Abstract | i |
| Acknowledgements | ii |
| Contents | v |
| List of Figures | vi |
| 1 Introduction | 1 |
| 1.1 Background | 1 |
| 1.2 Objectives and Aims | 2 |
| 1.3 Scope | 4 |
| 1.4 Report Structure | 4 |
| 2 Literature Review | 5 |
| 2.1 Basics of Cloud Computing | 5 |
| 2.2 Introduction to technology stack used | 7 |
| 2.3 Virtualisation | 8 |
| 2.3.1 Virtual Machines (VMs) | 8 |
| 2.3.2 Containerisation | 9 |
| 2.3.3 VMs vs Containers | 9 |
| 2.4 Docker | 10 |
| 2.4.1 Docker architecture | 11 |
| 2.5 Kubernetes (K8s) | 13 |
| 2.5.1 Kubernetes Infrastructure Components | 13 |
| 2.5.2 Kubernetes Resources | 15 |
| 2.6 Helm | 18 |
| 2.6.1 Helm Basic Concepts | 18 |
| 2.7 Monitoring and Visualisation tools | 19 |
| 2.7.1 Grafana | 19 |
| 2.7.2 Prometheus | 19 |
| 2.7.3 Compatibility of Grafana and Prometheus | 20 |

| | | |
|----------|--|-----------|
| 3 | Designed Approach | 21 |
| 3.1 | Components in the ASR System | 21 |
| 3.2 | Basic Components in the K8s deployment | 22 |
| 3.2.1 | Server and Worker Pods | 22 |
| 3.2.2 | Volumes | 23 |
| 3.2.3 | Ingress | 23 |
| 3.2.4 | Services | 24 |
| 3.2.5 | Load Balancer | 24 |
| 3.3 | Monitoring components in K8s deployment | 24 |
| 3.3.1 | Prometheus | 24 |
| 3.3.2 | Grafana | 25 |
| 3.4 | Helm Charts | 25 |
| 3.4.1 | Benefits of Helm | 25 |
| 3.4.2 | Structure of Helm Charts | 25 |
| 4 | Implementation | 27 |
| 4.1 | Docker Deployment | 27 |
| 4.2 | Deploying on cloud providers | 28 |
| 4.3 | On-Premise Deployment | 29 |
| 4.3.1 | On-Premise Infrastructure Set Up | 29 |
| 4.3.2 | ASR Components Deployment | 30 |
| 4.3.3 | Monitoring | 31 |
| 4.4 | Helm Packaging | 32 |
| 4.5 | Major errors | 34 |
| 4.5.1 | Deploying MetalLB onto on-prem cluster | 34 |
| 4.5.2 | Crashing of worker's kube-flannel and kube-proxy pod | 35 |
| 4.5.3 | Upgrading the worker node's OS version | 35 |
| 5 | Conclusion | 36 |
| 5.1 | Summary of project | 36 |
| 5.2 | Future Works | 36 |
| | Appendices | 42 |
| A | Code Snippets | 42 |
| A.1 | Testing Scripts | 42 |
| A.1.1 | client3.py | 42 |
| A.2 | Docker Configuration Files | 48 |
| A.2.1 | docker-compose.yml | 48 |

| | | |
|-------|---|----|
| A.3 | Setup Scripts | 49 |
| A.3.1 | network.sh | 49 |
| A.3.2 | kube.sh | 50 |
| A.4 | On-premise Deployment Configuration Files | 51 |
| A.4.1 | deployments/server.yaml | 51 |
| A.4.2 | deployments/worker.yaml | 52 |
| A.4.3 | ingresses/server-resource.yaml | 53 |
| A.4.4 | services/server.yaml | 54 |
| A.4.5 | volumes/worker-pv.yaml | 54 |
| A.4.6 | volumes/worker-pvc.yaml | 55 |
| A.4.7 | volumes/worker-storageclass.yaml | 55 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | Differences between On-Premise and Cloud Hosting environments | 3 |
| 2.1 | Resource management between levels of Cloud Services | 6 |
| 2.2 | VMs vs Containers | 10 |
| 2.3 | Docker architecture diagram | 11 |
| 2.4 | Kubernetes architecture diagram | 14 |
| 2.5 | Ingress Overview | 17 |
| 2.6 | Helm architecture diagram | 19 |
| 3.1 | ASR system architecture diagram | 22 |
| 3.2 | Overview of Master and Worker nodes in the cluster | 23 |
| 3.3 | Structure of Helm Chart | 26 |
| 4.1 | Output of ASR system | 28 |
| 4.2 | ASR system on cloud provider architecture | 29 |
| 4.3 | Structure of K8s manifest files for ASR system | 31 |
| 4.4 | Successful On-premise Cluster Status | 31 |
| 4.5 | Example of a Dashboard for the ASR system on Grafana | 32 |
| 4.6 | Structure of Helm Chart for ASR system | 33 |
| 4.7 | PodSecurityPolicy Deprecation | 34 |
| 4.8 | Crashing of worker's kube-flannel and kube-proxy pod | 35 |

Chapter 1

Introduction

1.1 Background

The consumption of online video media has experienced a significant surge in demand in recent years, particularly during the pandemic. As the situation stabilises and the world faces the aftermath of Covid, many people have become interested in accessing media over the internet. Online video streaming has become one of the most popular ways to meet this demand, offering a wide range of entertainment, educational, and informative content. As a result, it has become one of the most sought-after leisure activities.

One of the biggest players in this up-and-coming space is YouTube, which currently has 2.6 billion viewers [1] and more than a billion hours of video is being consumed on YouTube daily [2]. YouTube is currently available in 91 countries [1] and supports 80 different languages [3]. With such a diverse and large user base, situations which viewers come across videos of which they find hard to understand are quite common. Viewers might have difficulty trying to understand different accents even if the video is in their native language. One notable example of this is when the video author is speaking in their second language or is unfamiliar with the language, leading to potential communication difficulties or misunderstandings. This is when accurate subtitles would go a long way in helping the viewer's understanding of the content that the author of the video is trying to deliver. However, not all content creators on YouTube may have the time or resources to manually create subtitles for their videos. Therefore, a speech recognition system is necessary to automate the process. This speech recognition system should be capable of recognising the wide range of accents and speech patterns

present within a language, including those specific to a particular region or locale. This leads to the main program to be deployed in this project, the Automatic Speech Recognition (ASR) system [4].

The ASR system utilised in this project was developed by AISG Speech Lab, and it is built on Kaldi, an open-source speech recognition toolkit. Kaldi is written in C++ and designed specifically for speech recognition and signal processing, utilising a finite-state transducer for the purpose of speech recognition [5]. The main function of the system, in its current state, is to take in an audio input in either English, Mandarin, or a mix of the two, and transcribe it to text using machine learning algorithms to achieve high accuracy. This would save content creators time as the subtitles can be auto-generated with high accuracy using the ASR system. Since the current iteration of the ASR system is based in Singapore, this style of transcribing both English and Mandarin would be extremely useful as many of the locals would often converse in a mix of the two languages. Further development is currently being worked on for the system to be able to transcribe other languages in order for the ASR system to be used in other countries.

1.2 Objectives and Aims

The eventual objective of this project is to have it as an online accessible service for relevant organisations in Singapore. The ideal production level solution would be a deployment of the ASR system on a cloud environment as this would allow for high availability and high scalability [6] of the system to cater for surges in usage rates across different timings of the day. However, the organisations that are interested in the ASR system would likely already have their preferred cloud infrastructures and services which they use for their operations. Some of the popular cloud service providers include AWS, GCP, and Azure. These cloud service providers use common underlying technology like Docker and Kubernetes (K8s) for their services. However, because of the layer of abstraction provided by these cloud providers, they would have slightly different variations of services which would be useful in similar use cases. Thus, deploying the ASR system on different cloud infrastructures would each require a different approach. Therefore, keeping in mind the different needs of each organisation, the ASR system must have a deployment for all the existing infrastructures to suit the needs of these organisations.

Although existing work has been made for the deployment of the ASR system on popular cloud infrastructures mentioned above, there is still an apparent lack of research and development regarding organisations that prefer using on-premise cloud servers. The main focus of this project would be to fill in this research and development gap by having the ASR system deployed on bare metal servers without the help of cloud providers. My research aims to find ways to automate the set up of bare metal servers and the deployment of the ASR project on such systems. This would also allow for the deployment to be dynamic such that server administrators could change the default values to their server configuration without the need to get too technical with the architecture of the ASR system.

| Parameters | On-Premises (Self-Hosted) | Cloud Hosting |
|--------------------------------------|--|--|
| Level of Support | In-house IT staff | Vendor-provided support |
| Customisability (including security) | High | Limited by vendor |
| Reliability | Depends on IT staff | Guaranteed by vendor Service Level Agreement |
| Scalability | Limited by hardware capacity | Elastic scaling |
| Cost | High initial investment, ongoing maintenance costs | Pay-per-use model, lower initial investment |

Figure 1.1: Differences between On-Premise and Cloud Hosting environments

The above table shows the differences between on-premise and cloud hosting environments. Although there seems to be many factors in favour of cloud hosting, in terms of reliability, scalability and cost, on-premise hosting would still be preferred in certain situations. In the case of highly sensitive or classified data or applications, on-premise hosting would allow for greater control over the security measures as there would not be a third party involved. There is also the major benefit of having large customisability options for on-premise deployments where the customisation options would be limited to what the vendor could provide in cloud hosting deployments.

The eventual deployment of the on-premise ASR system should also have features similar to those provided in popular cloud infrastructures mentioned

above. Some examples of these features include load balancers [7], to provide congestion control, and storage services such as relational database services [8], to provide persistent storage. This would allow organisations who are interested in having a private cloud deployment of the ASR system to have the same benefits as the more established cloud providers. My project would therefore be able to ease the deployment process of the ASR system on bare metal on-premise servers.

1.3 Scope

The scope of this project is bounded by the deployment of the system onto bare metal systems, more specifically, deployment of the system into Nanyang Technological University (NTU) school servers. Although the scope of the project is limited to NTU school servers, related processes and methods of deployment can be first conceptualised and tested on some of the services provided by cloud providers in order to have an easier time transiting to bare metal systems. This project will not cover the details of the speech recognition models or the toolkit used in the system.

1.4 Report Structure

This report is divided into five chapters and an overview of each chapter is as follows:

Chapter 1, **Introduction**: provides an introduction of the project and a summary of its motivation and the scope.

Chapter 2, **Literature Review**: discusses the various technologies that are used in this project.

Chapter 3, **Designed Approach**: provides an overview of the concepts/ideas used in the design of the deployment of the ASR system

Chapter 4, **Implementation**: addresses the implementation of the approach as well as a few of the more major errors faced.

Chapter 5, **Conclusion**: concludes the report with a summary and future research direction.

Chapter 2

Literature Review

This chapter reviews the currently available technologies out in the market that are used as the backbone to this project. This chapter would help identify the key features of the deployment and provide any details on the currently available services that are used to host the ASR system on a bare metal system.

2.1 Basics of Cloud Computing

Cloud computing [9] is a field in computing where computing resources are delivered on demand, over the internet, to allow for faster innovation, flexible resources, and economies of scale. This would abstract the management of these services away from the user and would allow a more cost effective approach as users would only pay for what they use. In today's world, many organisations would prefer to scale their services to meet the demands of their clients, it is thus imperative that they utilise cloud computing. Cloud computing, despite being a recent innovation in the field of computing, has been gaining popularity as an increasing number of Information Technology (IT) professionals are recognising its potential to enhance their workflow.

Modern day cloud computing systems also offer another benefit, the model of distributed systems [10]. Large clouds would often be in a distributed system, with data centres located in multiple locations across the globe. This feature allows many organisations to deploy their systems in the microservices architecture that is commonly seen nowadays. The distributed systems model would host different microservices on different devices of their network and effectively split up the work, coordinating with each other to complete jobs

more efficiently than if a single device had been responsible for the tasks. This allows different levels of cloud services such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS) [11], just to name a few. These distributed systems would require a multitude of networking protocols to coordinate the distributed devices in order to effectively complete the jobs assigned to them.

The level of cloud service that is the focus of this project is the bare metal or on-premise server deployment. Differences between the different services are shown below:

| Resource | SaaS | PaaS | IaaS | On-Premise |
|----------------|------------------|------------------|------------------|------------|
| Application | Service Provider | User | User | User |
| Data | Service Provider | User | User | User |
| Runtime | Service Provider | Service Provider | User | User |
| Middleware | Service Provider | Service Provider | User | User |
| OS | Service Provider | Service Provider | User | User |
| Virtualisation | Service Provider | Service Provider | Service Provider | User |
| Networking | Service Provider | Service Provider | Service Provider | User |
| Servers | Service Provider | Service Provider | Service Provider | User |
| Storage | Service Provider | Service Provider | Service Provider | User |

Figure 2.1: Resource management between levels of Cloud Services

The differences between the services are the level of control and complexity of what the developer has to manage.

As seen from the diagram, a SaaS service has the least control over anything in the application but this also allows the service to be very simple to the user. Users of SaaS softwares would not need to worry about the hosting, operating systems or the language used to write the software in. This software offers off-the-shelf convenience and ease of use, which would be very suitable for consumers without any technical knowledge.

A PaaS service would be one step up from SaaS, where PaaS would provide a framework for application creation and deployment. PaaS allows developers

to focus on building their applications without having to consider the operating systems, storages or infrastructure. The difference between PaaS and SaaS is that PaaS allows the user to build the application itself on the framework provided and the PaaS service would provide the rest to be able to get the application up and running.

IaaS is a type of cloud computing service that enables users to access compute, storage, and networking resources on-demand, typically on a pay-per-use basis. With IaaS, users can obtain virtualised resources like servers, disks, networks, and IP addresses. However, users are responsible for administering the operating system (OS), data, applications, middleware, and runtimes. Users can have complete control over the entire infrastructure through a dashboard or API.

On-Premise environment [12] is when resources are deployed in-house and within an enterprise's IT infrastructure. This allows complete control over the resources and infrastructure but also adds a layer of complexity with needing to set up and maintain the whole infrastructure as well as the application.

2.2 Introduction to technology stack used

When doing an on-premise deployment, it is important to keep in mind that the deployment should take advantage of various virtualisation technologies. The reasoning of why this is the case can be found in the next section on virtualisation.

That being said, there are a few essential virtualisation [13] and orchestration [14] technologies available in the market now which would help with achieving the goal of this project. The technologies used in this project are, Docker as the container runtime engine, K8s for the container orchestration, Helm for templating, MetalLB for loadbalancing, Prometheus for monitoring and Grafana for visualisation.

There may be a few reasons why relevant organisations would choose to have an on-premise deployment of the ASR system as opposed to a deployment on the cloud. Specifically to this project, relevant organisation might want to keep their audio data or the data generated by the model within the organisation. They might also have certain configurations which are not

provided by external cloud hosting vendors. Another reason is that they might already have existing private infrastructure hosting their other projects and would like to integrate the ASR system into their infrastructure.

2.3 Virtualisation

Before virtualisation technology [13], software applications were deployed and ran on physical servers in a data centre. This could be a time-consuming and resource-intensive process, requiring dedicated IT staff to oversee hardware maintenance, updates, and security. During which, there is a need for a multitude of manual work to ensure the correct configuration of the software and the correct environment for that software to run on. This environment comprises the OS, environment variables, and other dependencies, including packages and libraries that are required for the application to run properly. Therefore, when software is deployed or migrated to another host, the process is often prone to result in bugs or errors. Furthermore, if several software applications were deployed on the same physical server, there would not be any isolation mechanism to ensure configurations and dependencies avoid conflicting with each other. Resource segregation is also not in place and would result in certain software underperforming if others took up more resources than they should.

2.3.1 Virtual Machines (VMs)

In order to resolve the above issues, virtualisation was introduced. Virtualisation allows for multiple Virtual Machines (VMs) [15] to be present and run on a single physical server. This would allow for each VM to act as an isolated environment and provide resource and environment segregation. However, one downside to VMs is that they each run their own OS, which would be considered a major overhead.

VM Images

A VM image [16] refers to a file containing a virtual disk that has an OS installed and is bootable. This file represents the state of a virtual machine, encompassing various components such as memory and device registers.

2.3.2 Containerisation

Containerisation technology [17] is still relatively new in the IT industry but is starting to gain popularity after their introduction. Containerisation is similar to VMs whereby they encapsulate software applications and their dependencies so they could be isolated from one another. They also allow software applications to be able to run uniformly and consistently on any infrastructure. However, the difference is containerisation would use the same OS. This would allow containerisation technology to be extremely lightweight when compared to VMs. Although containerisation technology is still in its infancy, it is quickly maturing and growing a vibrant community to provide support for both software developers and operations teams.

Container

A container is a self-contained and portable unit of software that includes the application code, its dependencies, and the runtime environment required to run the application consistently and reliably across different physical servers. A running container is a process which runs on the host machine and completely isolates software from its environment to ensure that it works uniformly despite differences in infrastructures and OS environments. Containerised software will therefore always run in the same way, regardless of the external infrastructure in which the container is run from.

Container Image

A container image comprises all the essential components required to execute an application, including the code, runtime, system tools, system libraries, and configurations, and can be easily packaged as a lightweight and autonomous executable package. The container images are transformed into containers when executed.

2.3.3 VMs vs Containers

The primary distinction between VMs and containers lies in their virtualisation approach: VMs virtualise an entire machine, including hardware components, while containers virtualise software layers above the OS level. This allows containers to overcome the main overhead of running VMs which is very useful when trying to develop and deploy a scalable and reliable infrastructure to host code on.

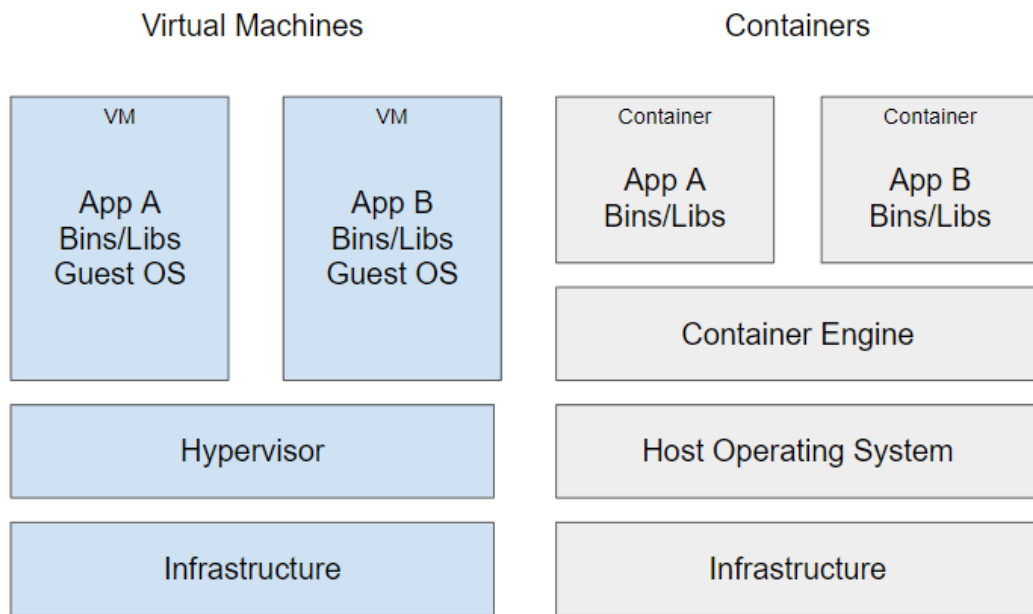


Figure 2.2: VMs vs Containers

2.4 Docker

Docker [18] is a platform, available as an open-source tool, designed for the purpose of developing, shipping, and executing applications. Docker uses the containerisation technology to allow the developer to separate their applications from their infrastructure so that their software could be delivered smoothly. Docker also allows the management of infrastructure at a software level, allowing developers to write code as to how they would like their virtualised infrastructure's environment to be like. Docker packages software into containers which include the software and everything the software needs for it to run properly. This would provide a reusable and programmable way to deal with common problems faced in software deployment such as installing, upgrading and removing on different host environments. Currently, Docker is the most popular container runtime and it has good integration and community support with many other popular cloud technologies as well as major cloud providers. In an on-premise environment, Docker allows developers to build and test applications in a local environment and then deploy them to a production environment with minimal effort, reducing the burden on IT staff.

2.4.1 Docker architecture

Docker has a client-server architecture [18]. The Docker daemon is responsible for building, running, and distributing Docker containers, which run software applications. It communicates with the Docker client, which can either reside on the same system as the daemon or on a different system. Communication between the client and daemon occurs through REST APIs, UNIX sockets, or network interfaces.

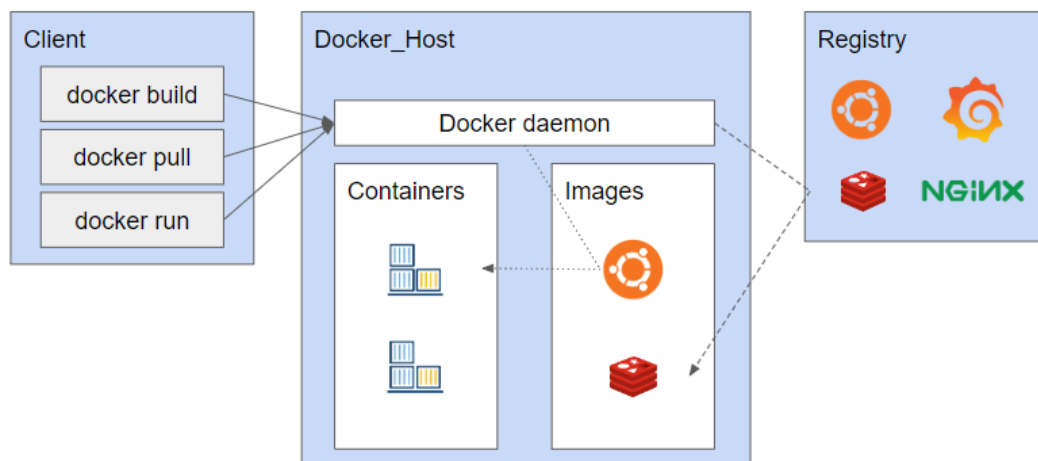


Figure 2.3: Docker architecture diagram

Docker daemon

The Docker daemon, dockerd [18], is responsible for the management of Docker objects such as images, containers, networks, and volumes. It receives Docker API requests from the docker client, which is issued by the developer, to be able to manage the docker objects. A Docker daemon can also interact with other daemons to manage Docker services.

Docker client

The Docker client [18], is the entry point in which developers interact with the Docker architecture. Commands are interpreted in the Docker client, and sent to dockerd through Docker API requests, which would execute those commands. Docker clients can communicate with more than one Docker daemon.

Docker registries

A Docker registry [18] is a storage of Docker images. It allows the developer

to pull images from the configured registry for use. The developer could also push an image of their own onto the registry for others to pull.

Docker Hub is a publicly accessible registry that is available for use by anyone, and by default, Docker is set up to search for container images on Docker Hub. Additionally, Docker Hub allows for the creation of private repositories for developers who have images containing sensitive information that should not be shared publicly.

Docker objects

The actual virtual environment created when using Docker is composed of different objects [18]. Some of the more prominent Docker objects are Dockerfiles, images, containers, volumes and network.

Dockerfile: The Dockerfile [19] is a text file consisting of a series of instructions or commands that are executed sequentially to perform specific actions on the base image, resulting in the creation of a new Docker image. With each instruction in the Dockerfile, a new layer is added to the Docker image.

Docker Images: Docker images [18] are essentially container images. Docker images can be generated in two way, namely, by using a Dockerfile, or by modifying an existing Docker image by altering the container environment and then saving the updated state as a new image. When creating an image with a Dockerfile, each instruction results in a layer that builds on top of the previous one. The order of the layers is crucial for efficient lifecycle management, as changing a layer in the image requires rebuilding all subsequent layers built from it. As a result, a series of intermediate images is created, with each layer being dependent on the layer immediately below it. Therefore, a good rule of thumb to follow is to have the layer that changes the most to be at the top of the stack of layers.

Docker Containers: Docker containers [18] are the structural units of Docker, which contain the entire package needed to run the application. Docker containers is a virtualised runtime environments of Docker images. This would create a clear distinction between Docker containers and images where containers are the instances of the images which can be thought of as templates for the containers.

Docker Volumes: Docker volumes [20] are entities which allow for persistent data to be stored and sharing of container's data. This is done by mounting the Docker container's host directory of choice to the inside of that Docker container, resulting in the ability of the container to read and write from that particular host directory.

Docker Networking: Docker networking [21] would allow Docker containers to be linked to the networks required, while providing isolation for these containers. Docker network contains the following drivers: Bridge, Host, None, Overlay and Macvlan. Bridge is the default network driver and is used when multiple containers communicate with the same host. Host is used when isolation is not required between the container and the host. None disables all networking capabilities. Overlay enables containers to run on different hosts. Finally, Macvlan is used when MAC addresses are required to be assigned to containers.

2.5 Kubernetes (K8s)

Kubernetes (K8s) [22] is a container orchestration tool [14], being able to automate the lifecycle of containers, including provisioning, deployment and scaling. This allows for developers to be able to enjoy the benefits of containerisation at scale without incurring additional maintenance overhead. Containerisation technology ensures the easy packaging of the software while K8s ensures the easy management of the containers. K8s provides a way to automate the deployment, scaling and management of containerised applications. This can help to simplify the process of deploying and managing large-scale applications in an on-premise environment. This helps the organisation easily manage their own on-premise infrastructure as everything is made configurable in software.

2.5.1 Kubernetes Infrastructure Components

K8s infrastructure components [23] consist of the cluster itself, a control plane, and nodes.

Control Plane

The control plane [23] is commonly known as the master node. This is where the developer would issue the relevant commands to configure the K8s

cluster. The following are the components that make up the control plane. First, kube-apiserver [23] is the component that exposes the K8s API. Next is the etcd [23] which is a consistent and highly-available key value store used as K8s' backing store for all cluster data. Kube-scheduler [23] assigns newly created Pods [24] to a node for them to run on. A pod is the smallest deployable unit in K8s and would be discussed more extensively in the next section, K8s Objects. Kube-controller-manager [23] runs controller processes. Finally, cloud-controller-manager [23] embeds cloud-specific logic.

Nodes

Nodes or worker nodes [23] also have components that run on them. This is where the heavy lifting of running the actual application is done. The K8s runtime environment is maintained by various components that are responsible for managing the running Pods. The worker nodes in particular run several components, including the kubelet [23], which is an agent that ensures healthy container runtime in each Pod. Additionally, the kube-proxy [23] runs on each node in the cluster and maintains network rules, enabling both internal and external communication with the Pods. Finally, a container runtime [23] is responsible for running the containers, which can be the aforementioned containerd.

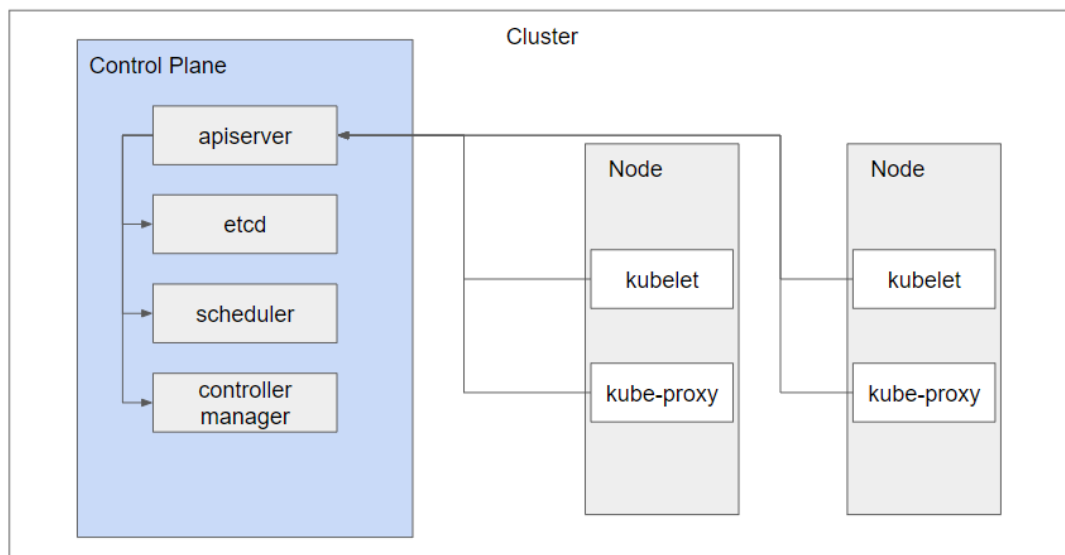


Figure 2.4: Kubernetes architecture diagram

2.5.2 Kubernetes Resources

K8s resources are persistent entities in the K8s system that represent the state of your cluster. By creating resources, the cluster would know what its desired state should look like. The following are the resources which are relevant to this project:

Pods

A Pod, as mentioned above, is a basic building block in K8s and it is the smallest deployable unit in a K8s cluster. A pod is able to run one or more containers but it is recommended that only containers that are tightly coupled to run on the same pod. Otherwise, the “one pod per container” rule is recommended and is the default practice when using K8s. Each pod must specify the amount of computational resources to be provisioned for it to run. When K8s chooses a worker node to run the pod, it would choose a worker node that has sufficient resources to run the pod and schedule the pod to run on that worker node. Pods are ephemeral and are not designed to run forever. Pods might stop working during its lifetime, this leads to a problem where the internal IP of a particular application is not consistent when the Pod is restarted.

ReplicaSets

A ReplicaSet [25] is a K8s object that helps maintain a stable set of replica Pods running at any given time. It ensures the availability of a specified number of identical Pods, making it possible to maintain the desired level of application availability and scalability.

Deployments

Deployments [26] are essentially blueprints for creating Pods or ReplicaSets. It acts as another layer of abstraction above Pods, where a developer would mostly work with deployments instead of Pods. Deployments tell K8s how to create or modify instances of the pods that hold a containerised application. Deployments can help to scale the number of pods, roll out updated code of the containerised application in a controlled manner and roll back to an earlier deployment version if necessary.

Services

In K8s, services [27] are logical abstractions for a deployed group of Pods in a cluster. Services act as an IP endpoint so that the IP to reference an

application would stay the same even if the Pods are recreated. There are internal and external services. Internal services are used by the cluster internally and external services expose the application to the outside network. There are several types of services available in K8s:

ClusterIP [27]: A ClusterIP creates a virtual IP inside the cluster to enable communication between different services. ClusterIP would allow the pods running the same type of service to be grouped together and would allow for communication between pods to be abstracted to just a single interface. ClusterIP is also the default service type for K8s.

NodePort [27]: NodePorts allow an internal port to be accessible on a port in a node. K8s would reserve a port on the node which ranges from 30000 to 32767. Creating a NodePort service would also automatically create a ClusterIP service. NodePort service with pods in different nodes still works the same way as if the pods are all in the same node. This would allow the service to be accessed within and outside the K8s cluster.

LoadBalancer [27]: This provisions a load balancer for the application in supported cloud providers. This would allow the service to be externally exposed using the cloud provider's native load balancer. The load balancer provisioned would be assigned to a unique and publicly accessible IP address and will redirect all networking traffic to the service. NodePort and ClusterIP service would automatically be created.

Ingress

The K8s Ingress component [28] provides a way to expose HTTP and HTTPS routes from outside the cluster to the services running inside the cluster. It achieves this by controlling the traffic routing using rules that are defined on the Ingress resource. Ingress is preferred over using multiple LoadBalancer Services to provide external access as each of the services would then require one load balancer to be provisioned leading to more resources being used. An Ingress would only require one load balancer to be provisioned even if it is providing access to multiple services. When a client sends an HTTP request to the Ingress, the Ingress decides on the Service the request is forwarded to by the host and path specified in the request. An Ingress Controller must exist first to satisfy an Ingress. Creating only an Ingress resource without the corresponding controller has no effect. The diagram below shows an Ingress resource providing access to a service.

Ingress Controller

An Ingress Controller [29] acts as a reverse proxy and load balancer. It is the implementation of a K8s ingress resource. It recognises the configurations from Ingress resources and converts them into routing rules that reverse proxies can recognise and implement. The Ingress Controller adds a layer of abstraction to traffic routing, taking traffic from outside the K8s cluster and load balancing it to Pods running inside the cluster.

Difference between Load Balancer and Ingress Controller

A load balancer type service is used to expose services externally using cloud provider's load balancer or in the case of an on-premise deployment, a load balancer like MetalLB [30]. Ingress controller is used to route traffic from external sources to the appropriate service within the K8s cluster based on rules defined in the Ingress resource

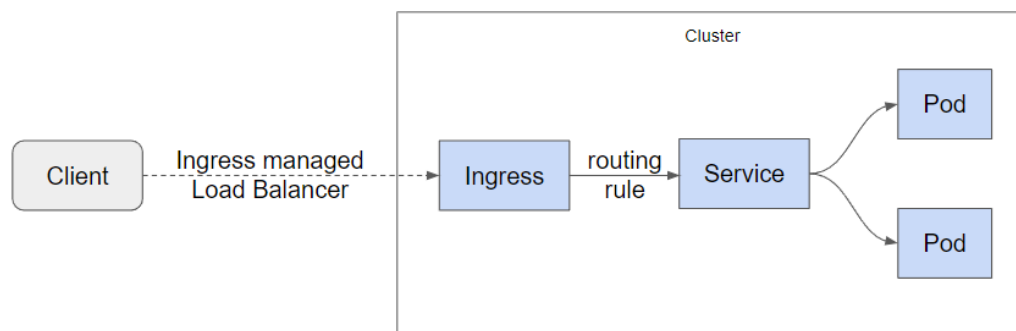


Figure 2.5: Ingress Overview

Storage

Data that is generated or being used in pods would not stay persistent upon the restarting of the pod. This poses a problem where data that needs to be persistent cannot be stored within the containers. K8s storage components would be the answer to this issue. Storage in or outside the K8s cluster would keep the data persistent. This section of the report will examine Persistent Volumes and their components, which are relevant to the requirements of this project.

Persistent Volumes (PV) [31]: PVs are similar to Volumes, whereby they live in a cluster but have lifecycles which are independent to the Pods which are

using the PV. This makes PVs ideal for storing data which are required to persist beyond the lifecycle of any Pod in the cluster. PVs are also used to store shared data between the different Pods in the cluster as they are able to be accessed by multiple Pods in the same namespace.

Persistent Volume Claims (PVC) [31]: PVCs are requests for PV resources and are used by Pods to claim the need for a PV. A PVC can request specific storage sizes and access modes namely, ReadWriteOnce, ReadOnlyMany or ReadWriteMany. When Pods require persistent storage in a PV, they would use a PVC to claim a PV for their needs instead of referencing the PV directly. This provides an abstraction of the implementation of storage from the Pods.

Storage Classes [32]: A Storage Class allows administrators to describe types of storage they offer. These types are known as “classes”. Different types of classes might offer different levels of quality-of-service or backup policies or other policies which the administrators choose.

2.6 Helm

Helm [33] is a package manager for K8s, which makes it more efficient in deploying large systems with many microservices than just using K8s manifest files. This makes it very popular in the community and it is widely used along with K8s.

2.6.1 Helm Basic Concepts

Helm has three basic components [33] and all operations of Helm are basically centred around these concepts.

Chart

A Chart [33] is a collection of files that describe a set of related K8s resources and is the packaging format in Helm. Helm Charts can be used to deploy a part of the whole architecture or a complex system as a whole.

Repository

Helm repositories [33] are similar to Docker repositories where Helm Charts are stored. They can be pulled from the repository to be deployed immediately or downloaded as a package onto the local system. A developer could also push their own Charts to the repository. Like Docker repositories, Helm repositories could also be made private to the user or organisation.

Release

Releases [33] are instances of Charts running in K8s. It is possible for a Chart to be installed multiple times in the same K8s cluster, creating a new Release for every installation.

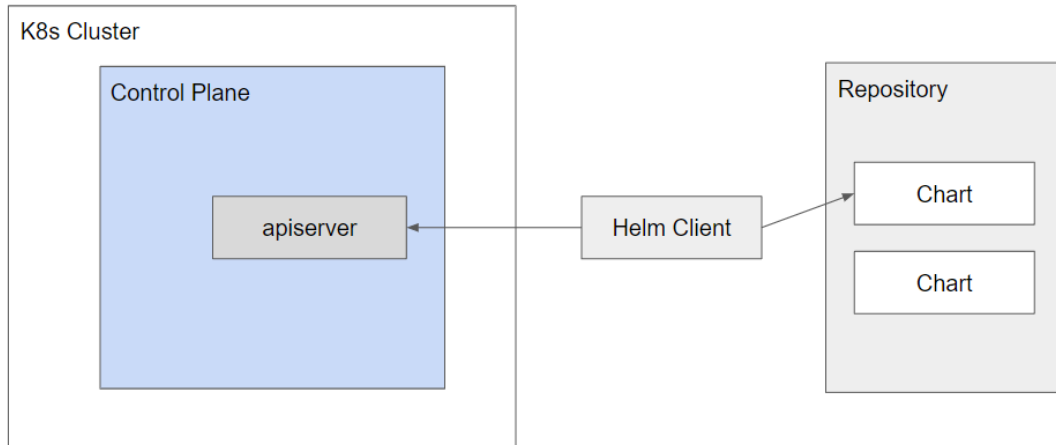


Figure 2.6: Helm architecture diagram

2.7 Monitoring and Visualisation tools

2.7.1 Grafana

Grafana [34] is an open source interactive data-visualisation tool which allows users to see data from various data sources in the form of charts and graphs which are presented in a unified dashboard. Grafana also allows users to query or set alerts on the data and metrics from wherever that information is stored including a K8s cluster. This allows anyone using the dashboard to be able to more easily analyse the data, identify trends and inconsistencies, and ultimately improve on the maintainers' ability to understand the state of the system which Grafana is visualising.

2.7.2 Prometheus

Prometheus [35] is a popular open-source alerting toolkit, designed for monitoring and alerting cloud-native environments including K8s. It collects metrics from various sources in the cloud environment and stores them along

with a timestamp, in a time-series database which can then be used for further analysis.

2.7.3 Compatibility of Grafana and Prometheus

Grafana and Prometheus [36] are often used together to have a complete monitoring and visualisation solution for cloud-native environments.

Grafana provides a wide range of built-in visualisation options, such as graphs, gauges, and heatmaps, as well as allowing users to create their own custom dashboards to monitor and analyse metrics.

Prometheus is primarily used for collecting and storing metrics from various sources, and has a powerful query language for analysing and querying these metrics. However, unlike Grafana, Prometheus does not offer built-in features for data visualisation or dashboarding.

By combining Prometheus and Grafana, organizations can leverage the strengths of both tools to create a complete monitoring and visualisation solution. Prometheus can be used to collect and store metrics from various sources, and Grafana can be used to visualise and analyse these metrics in real-time, providing insights into the health and performance of the monitored systems.

Chapter 3

Designed Approach

The proposed solution would consist of a simple cluster with 2 nodes, the master and worker nodes. The following diagram would illustrate the K8s components and resources which are essential to achieving the main functionality of the ASR system.

3.1 Components in the ASR System

To comprehend the intended deployment, it is essential to have an understanding of the components that constitute the ASR system. The ASR system is composed of two primary parts - a server process and a worker process. The ASR server process receives and manages the client requests while the ASR worker process carries out the actual transcription process. Furthermore, it is necessary to have a volume mount to enable the worker process to access the transcription models.

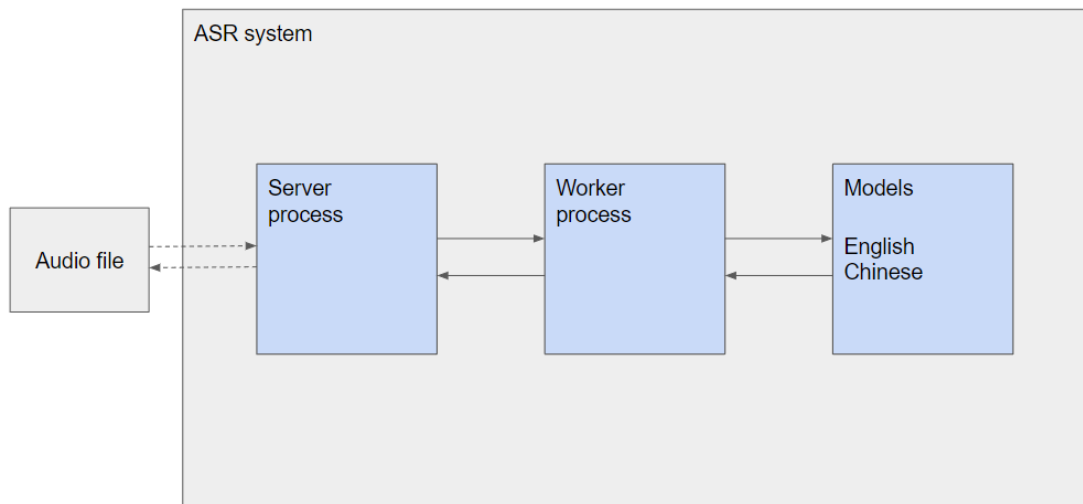


Figure 3.1: ASR system architecture diagram

3.2 Basic Components in the K8s deployment

3.2.1 Server and Worker Pods

As mentioned above, there are 2 main processes to the ASR system, the server and the worker process. This would lead to the proposed solution having two different kinds of Pods in order to separate the processes involved. These Pods are simply the server and worker Pods. The server Pods would be responsible for running the ASR server process while the worker Pods would be responsible for running the ASR worker process. It is possible to have multiple server and worker Pods running on the same node, depending on the configurations and the workload that is required.

Server and worker Pods are different from master and worker nodes. Pods are the smallest unit of execution in a K8s cluster while nodes are the physical servers or VMs that comprise a K8s Cluster. That being said, all the server and worker Pods would be running on the worker nodes of the cluster. This is due to the fact that worker nodes are the only computational resource in the K8s cluster, the worker node is the only place where all the Pods run. The master node would host the K8s control plane components. It is used as the entry point to the cluster, allowing the cluster administrator to issue commands to change the configurations of the deployments in the cluster.

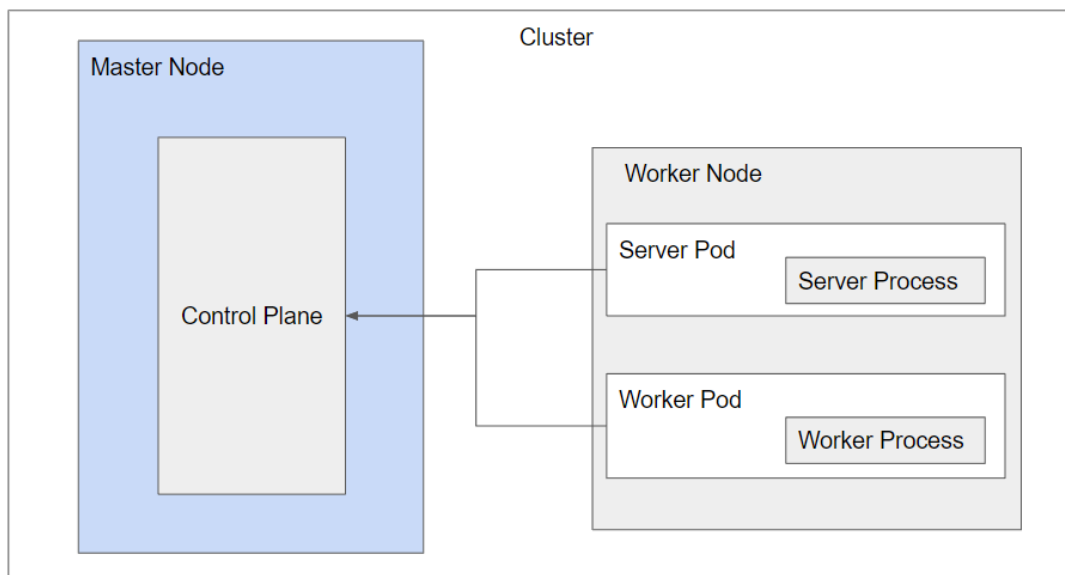


Figure 3.2: Overview of Master and Worker nodes in the cluster

3.2.2 Volumes

The worker pod of the ASR system would have to make use of persistent volume mounts for it to be able to access the ASR models used for the speech to text transcription. For this purpose, the cluster has to support a storage class to be able to define a "kubernetes.io/no-provisioner" as the provisioner, because this is a deployment for an on-premise cluster without any cloud providers as the provisioner. There should also be a PVC as well as the actual PV itself. Both the PVC and the PV would have their storage class set to the storage class that was created previously. The PVC is used by the application to request for a persistent storage from the cluster. This would allow K8s to dynamically provision a PV that meets the requested criteria from the PVC.

3.2.3 Ingress

An ingress resource is needed to define the set of rules that would route traffic to the K8s service created above based on the path of the incoming request. However, an ingress controller is needed to be able to interpret the rules that are written in the ingress resource and actually do the network traffic routing. For this cluster, the standard Nginx ingress controller is used [37].

3.2.4 Services

In order for a user to be able to access the application, a service has to be used to expose the ASR application running on the server and worker pods as a network service. In this deployment, a load balancer service is used so that it could distribute network traffic evenly across a set of pods, and would be able to easily scale up to meet more demand in the future.

3.2.5 Load Balancer

The difficulty of deploying the ASR system on-premise is that there is no load balancer service being provided by traditional cloud providers like AWS, GCP or Azure. Thus, there has to be an alternative load balancer to provide the same functionality. MetalLB is a popular choice for an on-premise load balancer [38].

Even though MetalLB provides load balancing service, it still requires a service of type load balancer deployed in the K8s cluster. This is due to that fact that MetalLB operates at the network layer and only provides a way to assign an external IP address to a service. It does not actually create or manage the load balancer type service, which is responsible for distributing traffic to the pods in the K8s cluster.

3.3 Monitoring components in K8s deployment

3.3.1 Prometheus

Prometheus has several components which has to be deployed in a K8s cluster in order for it to work. Below are the components which are deployed for Prometheus to work in the K8s cluster.

Prometheus Server: This is the main component which collects and stores the metrics data. It periodically scrapes data from exporters.

Exporters: These are agents that obtains metrics data in a format that Prometheus can understand. K8s comes with built-in exporters for various components, such as kubelet and kube-proxy but custom exporters can also be written for specific use cases.

Alertmanager: This is a component that receives alerts from Prometheus server when a certain metric exceeds a threshold or when a certain event occurs and sends notifications to various channels.

3.3.2 Grafana

Grafana would be used in the cluster to provide a web interface for dashboard creation, querying and visualisation. Grafana would require a data source, which in this case would be Prometheus. Grafana would use the metrics from which Prometheus gathers from its exporters and displays them in a visual form for the user.

3.4 Helm Charts

Helm Charts can be used to package a whole K8s deployment and make it easier for the cluster administrators to change configurations and easily scale the system. K8s components are deployed using manifest files which are yaml files that hold the configurations of each component which is going to be deployed in the K8s cluster. Helm converts templates to manifest files by using its templating engine.

3.4.1 Benefits of Helm

Helm allows the maintainer to abstract this by providing a the maintainer reusable templates for K8s resources. This makes it easier to manage and deploy complex applications with multiple resources. Helm also allows versioning of the application releases, allowing for rollbacks. Helm also allows for easy management and deployment of applications that rely on multiple external components. Another important benefit with Helm is that it allows for simpler deployments. With Helm, an application with all the required K8s components could be deployed with a single command as compared to having multiple command for manifest files.

3.4.2 Structure of Helm Charts

The basic components in a Helm Chart are:

Chart.yaml contains basic information about the Chart, such as its name, version, description, and maintainers.

values.yaml contains the default values for the template configuration settings used in the Chart which would be used to create the manifest files to deploy the application in the K8s cluster. Users can choose to override these values during installation or upgrade.

The **templates directory** contains the templated K8s manifests for the resources that the Chart deploys. The templates use Go templating language to inject configuration values from the values.yaml file into the templated K8s manifest files when generating the final manifest files used for the deployment.

helpers.tpl contains helper functions that can be used in the templates to reduce code duplication and improve readability.

NOTES.txt contains a message that is displayed after the Chart is installed or upgraded. It is often used to provide additional instructions or information to the user.

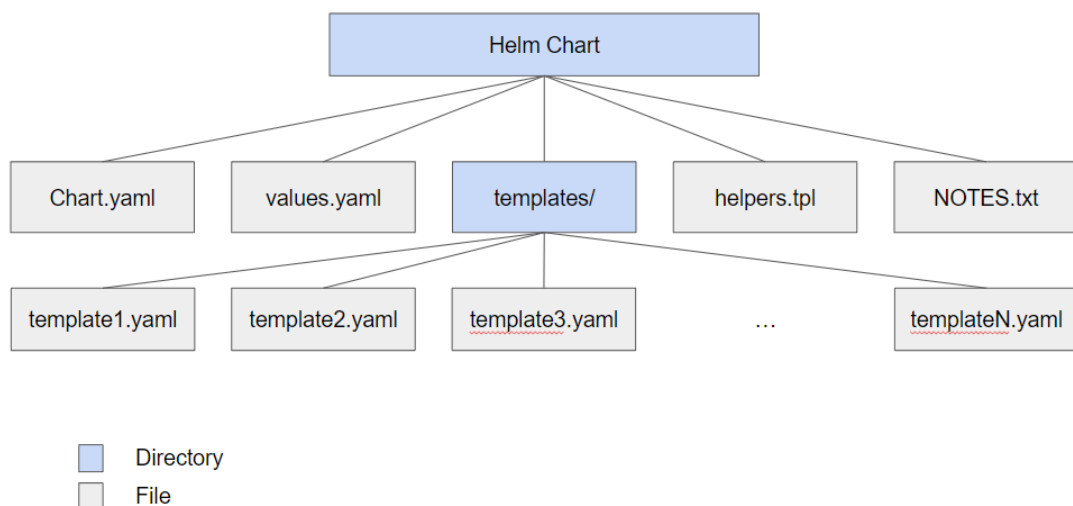


Figure 3.3: Structure of Helm Chart

Chapter 4

Implementation

This chapter would discuss the implementation of the designed approach to the problem. As the objective is an on-premise deployment, the solution would be implemented on NTU's school servers. Testing the deployment of the ASR system on easier and more manageable environments would allow better build up to the actual deployment. The different environments that were tested on is a Docker deployment and deployment on cloud providers, mainly Azure and AWS. To test the reachability of the ASR system, the WebSocket protocol [39] and a Python script called "client3.py" will be utilised. Refer to Appendix A.1.1 for the "client3.py" Python script.

Throughout the deployment process of the ASR system, several errors were encountered at different stages. This section will highlight some of the major errors encountered during the deployment process, in addition to discussing the implementation of the deployment.

4.1 Docker Deployment

The first stage of getting to understand the deployment is to be able to set up a Dockerised version of the application. As mentioned in section 3.1, the ASR system mainly comprises of a server and a worker process as well as a volume mount for the models. This would mean that the deployment would require 2 containers, one for the server process and one for the worker process, as well as a volume mount to a directory in the host machine that is mounted onto the worker container so that the worker process is able to access the models. In this case, docker-compose.yml, a file which is used by Docker Compose to bring up multiple containers, is used. Refer to Appendix

A.1.2 for the "docker-compose.yml"

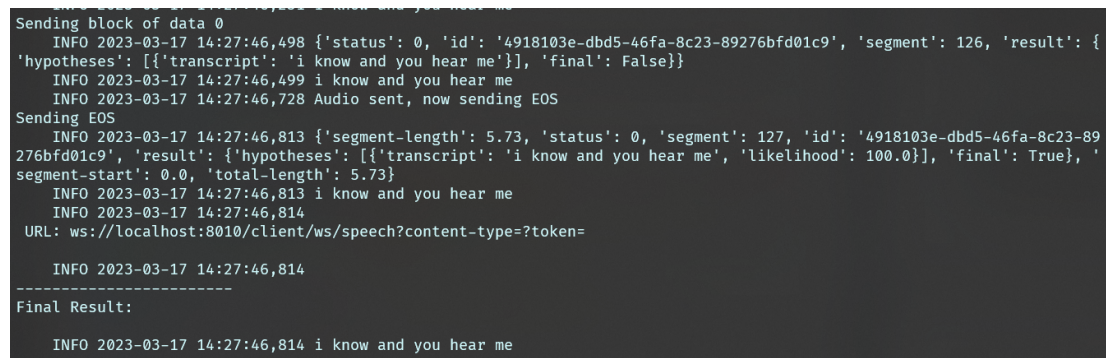
In order for the docker-compose.yml file to bring up the containers, Docker must be installed on the machine [40]. Once installed, simply execute the command:

```
docker compose up -d
```

to get Docker to bring all the containers up. Once the containers are running, sending a websocket command to localhost through port 8010 is enough to reach the ASR system deployed on the Docker containers. This can be done using the Python script, "client3.py" in Appendix A.1.1 through the command:

```
python [path to Python script] -u ws://localhost:8010/client/ws/speech  
-m [model to use] [path to audio file]
```

Upon successful execution, an output similar to the following should be observed:



```
INFO 2023-03-17 14:27:46,498 {'status': 0, 'id': '4918103e-dbd5-46fa-8c23-89276bfd01c9', 'segment': 126, 'result': {'  
'hypotheses': [{'transcript': 'i know and you hear me'}], 'final': False}}  
INFO 2023-03-17 14:27:46,499 i know and you hear me  
INFO 2023-03-17 14:27:46,728 Audio sent, now sending EOS  
Sending EOS  
INFO 2023-03-17 14:27:46,813 {'segment-length': 5.73, 'status': 0, 'segment': 127, 'id': '4918103e-dbd5-46fa-8c23-89  
276bfd01c9', 'result': {'hypotheses': [{'transcript': 'i know and you hear me', 'likelihood': 100.0}], 'final': True}, '  
segment-start': 0.0, 'total-length': 5.73}  
INFO 2023-03-17 14:27:46,813 i know and you hear me  
INFO 2023-03-17 14:27:46,814  
URL: ws://localhost:8010/client/ws/speech?content-type=?token=  
  
INFO 2023-03-17 14:27:46,814  
-----  
Final Result:  
  
INFO 2023-03-17 14:27:46,814 i know and you hear me
```

Figure 4.1: Output of ASR system

4.2 Deploying on cloud providers

To have a better understanding of how the ASR system is going to be deployed onto the on-premise servers in NTU, having a deployment for cloud providers is essential and would be the first step. This is because cloud providers allow more room for error as many of the considerations would be abstracted away from the user, such as the load balancer. For this purpose, the ASR system has been deployed on Azure VMs and EC2 instances on AWS. The deployment for both is similar. Due to the unavailability of the servers for Azure VMs nearing December 2022, transitioning to AWS' EC2 is necessary to continue learning about the deployment process.

The deployments would follow the structure of the Docker deployment. However, K8s has another layer of abstraction. Instead of interacting with the containers directly, each process running in a container which would be running in a pod in the worker node. Volume mounts would need to have persistence in a K8s cluster and as such would require a PV and a PVC in the cluster. A service is also required to expose the server pod in the worker node to the end user for them to use the same Python script mentioned above, client3.py to send an audio file in. Load balancers and ingresses are not required as the cloud providers would have their own native load balancers for routing traffic. The diagram below shows the architecture of the ASR system's deployment on cloud-providers.

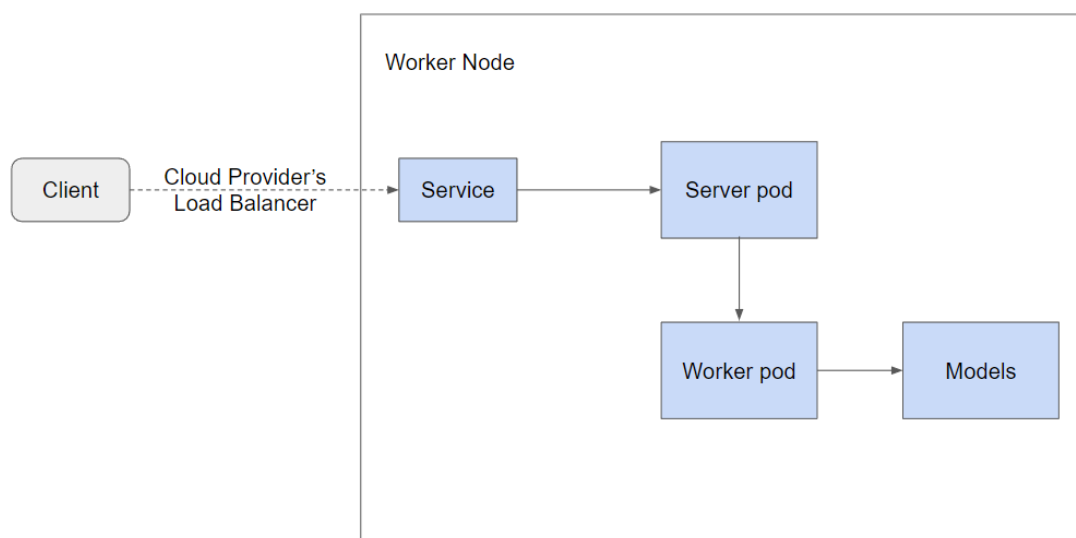


Figure 4.2: ASR system on cloud provider architecture

The steps for setting up and deploying on cloud providers are similar to that of the on-premise deployment and as such it would be covered in the next section.

4.3 On-Premise Deployment

4.3.1 On-Premise Infrastructure Set Up

The setup of infrastructure that is used in this project is as follows: Two servers, both with 20 cpu cores and 248 Gb of RAM. Both the master and

worker node should be running Ubuntu 18.04.

K8s has a container runtime dependency, meaning it requires a container runtime to be installed first. Docker would be used as the container runtime in this project.

Initially, the installation of K8s [41] is required on both the master and worker nodes, similar to the deployment process on cloud providers. The installations should consist of kubectl, kubeadm and kubelet. kubectl is only needed for the master node as it would be the command line tool used to set up and update the configurations of the cluster.

In order to set up the cluster, the master and worker nodes would have to have the correct network configurations. Refer to "network.sh" script in Appendix A.3.1 to set up the network configurations for both master and worker nodes.

The cluster can then be initialised in the master node and joined by the worker node. The "kube.sh" script in Appendix A.3.2 would take in the parameter "master" or "worker" in order to execute the correct commands to either create or join the cluster using kubeadm. The script also combines the installation of kubelet, kubeadm and kubectl.

4.3.2 ASR Components Deployment

Taking reference from the Docker deployment, the manifest files of the on-premise K8s deployment can be written. Refer to the configuration files under Appendix A.4 for the components which are deployed in the cluster, some values might need to be tweaked according to the setup of the specific infrastructure the files are being deployed on. For K8s clusters which are on cloud providers, the cloud provider's load balancer would be sufficient to do the routing of network traffic such that the system would work. However, in the case of an on-premise deployment, just deploying based on the files in Appendix A.4 is not enough. A load balancer and ingress controller is still needed. As mentioned earlier in the report, MetalLB load balancer and Nginx ingress controller are used in the deployment [42].

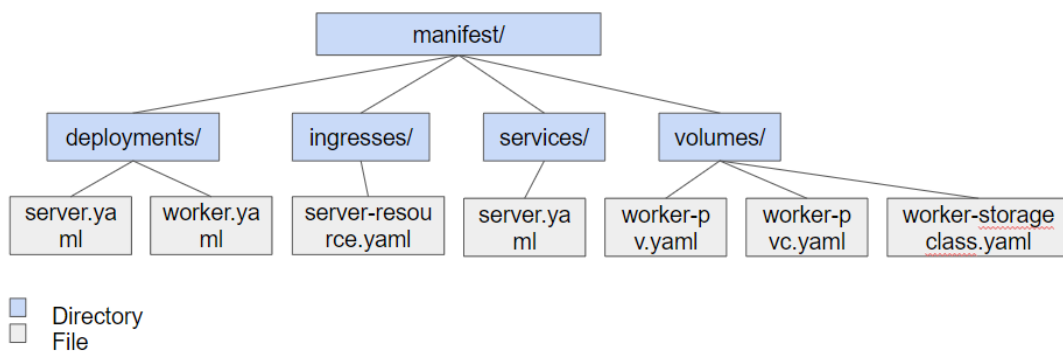


Figure 4.3: Structure of K8s manifest files for ASR system

The output of the status of a successful deployment of the ASR system onto on-premise servers is shown below:

```

fyp-jonathan@t1ntu-ESC4000-G2-Series:~/master-dir/one-node/manifest$ kubectl get all
NAME                                READY    STATUS    RESTARTS   AGE
pod/on-prem-server-548bc8d76f-g55wz 1/1      Running   0           33d
pod/on-prem-worker-6d74496ff5-tzwbw 1/1      Running   0           33d

NAME                                TYPE                CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
service/kubernetes                  ClusterIP           10.96.0.1     <none>          443/TCP          66d
service/on-prem-server-service      LoadBalancer       10.98.70.205  172.21.46.1    8010:32522/TCP   33d

NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
deployment.apps/on-prem-server      1/1      1              1             33d
deployment.apps/on-prem-worker      1/1      1              1             33d

NAME                                DESIRED    CURRENT    READY    AGE
replicaset.apps/on-prem-server-548bc8d76f 1          1          1        33d
replicaset.apps/on-prem-worker-6d74496ff5 1          1          1        33d
  
```

Figure 4.4: Successful On-premise Cluster Status

The IP address to access the ASR system is shown under "service/on-prem-server-service"s "EXTERNAL-IP". In this case the IP address is 172.21.46.1. The same testing method as the one shown in the section 4.1 Docker deployment can be used changing the IP address from "localhost" to "172.21.46.1".

4.3.3 Monitoring

In order to deploy Grafana, Prometheus has be deployed first to be used as a data source for Grafana to pull data from [43]. To monitor the K8s cluster and

obtain useful metrics, deployment of Kube State Metrics [44], Alertmanager [45], and Node Exporter [46] is necessary. Next, Grafana can then be deployed and used to view the metrics of the cluster [47]. Dashboards could be created or imported in order to have visuals on the metrics of the ASR cluster. Below is an example of such a dashboard using the metrics obtained by Prometheus on the ASR cluster deployed.

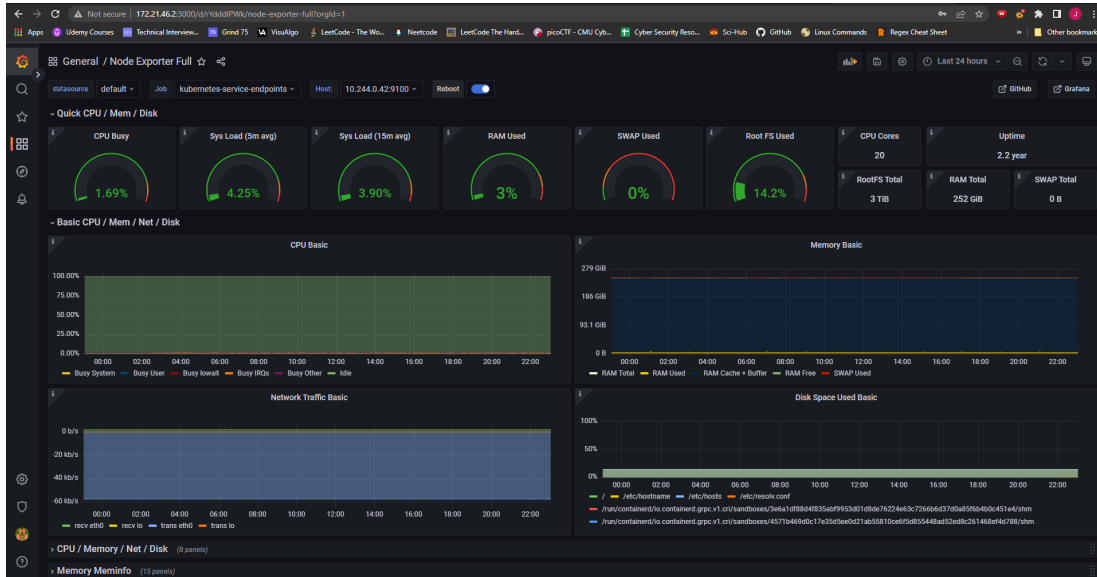


Figure 4.5: Example of a Dashboard for the ASR system on Grafana

4.4 Helm Packaging

To package the deployment, it is necessary to create a customized helm chart and populate the necessary templates, values, and helper.tpl files based on the deployment's manifest files. The structure of the Helm chart is as follows:

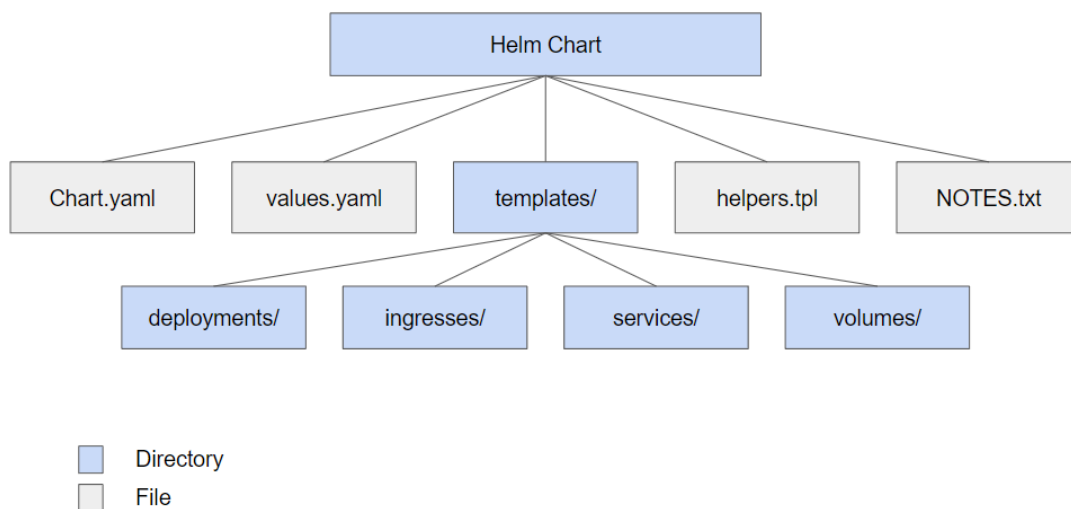


Figure 4.6: Structure of Helm Chart for ASR system

The names of each K8s component as well as the variables used are defined in helper.tpl file which pulls the actual values from the values.yaml file. The values.yaml file would contain the default parameters which could be overwritten with other values to fit the use cases of the deployment. In general, helpers.tpl file should include things that are being reused many times in the helm templates. Below shows an example of how the chart works.

helpers.tpl file snippet:

```

# container deployment Image
{{- define "on-prem.server.image.repo" -}}
  {{ default "lyvt/decoding-sdk" .Values.server.image.repository }}
{{- end -}}

```

values.yaml file snippet:

```

server:
  # container deployment Image
  image:
    repository: lyvt/decoding-sdk

```


templates/deployments/server.yaml file snippet:

containers:

- **image:** `{{ include "on-prem.server.image.repo" . }}`:1.2

Actual manifest file generated snippet:

containers:

- **image:** `lyvt/decoding-sdk`:1.2

The variable, "on-prem.server.image.repository" is defined in helpers.tpl and its value, "lyvt/decoding-sdk" is pulled from values.yaml. In the template file, templates/deployments/server.yaml, "on-prem.server.image.repository" is referenced, and "lyvt/decoding-sdk" is substituted in place of the reference during the construction of the manifest files by the templating engine.

4.5 Major errors

4.5.1 Deploying MetalLB onto on-prem cluster

An error was encountered during the deployment of MetalLB onto the cluster, as MetalLB requires the use of "PodSecurityPolicy", which has been deprecated in K8s v1.25 onwards [48]. As a result, it was necessary to roll back kubeadm, kubectl and kubelet to version v1.24.

```
lyp-jonathan@tintu-ESC4000-G2-Series:~$ kubectl delete -f metallb.yaml
serviceaccount "controller" deleted
serviceaccount "speaker" deleted
clusterrole.rbac.authorization.k8s.io "metallb-system:controller" deleted
clusterrole.rbac.authorization.k8s.io "metallb-system:speaker" deleted
role.rbac.authorization.k8s.io "config-watcher" deleted
role.rbac.authorization.k8s.io "pod-lister" deleted
role.rbac.authorization.k8s.io "controller" deleted
clusterrolebinding.rbac.authorization.k8s.io "metallb-system:controller" deleted
clusterrolebinding.rbac.authorization.k8s.io "metallb-system:speaker" deleted
rolebinding.rbac.authorization.k8s.io "config-watcher" deleted
rolebinding.rbac.authorization.k8s.io "pod-lister" deleted
rolebinding.rbac.authorization.k8s.io "controller" deleted
daemonset.apps "speaker" deleted
deployment.apps "controller" deleted
resource mapping not found for name: "controller" namespace: "" from "metallb.yaml": no matches for kind "PodSecurityPolicy" in version "policy/v1"
ensure CRDs are installed first
resource mapping not found for name: "speaker" namespace: "" from "metallb.yaml": no matches for kind "PodSecurityPolicy" in version "policy/v1"
ensure CRDs are installed first
```

Figure 4.7: PodSecurityPolicy Deprecation

4.5.2 Crashing of worker's kube-flannel and kube-proxy pod

During the on-premise deployment process of the ASR system, there were repeated crashes of the worker's kube-flannel and kube-proxy pods. Further investigation revealed that the issue was caused by a configuration problem in the Nvidia GPU's configuration. The Nvidia K8s plugin had differing documentation on Nvidia.com [49] compared to that on GitHub [50], making it challenging to diagnose and resolve the issue as it was beyond the scope of the project. Nonetheless, a solution that proved effective, entailing the inclusion of "SystemdCgroup = true" in /etc/containerd/config.toml.

```
Every 1.0s: kubectl get all --all-namespaces
```

| NAMESPACE | NAME | READY | STATUS | RESTARTS | AGE |
|--------------|---|-------|------------------|-------------|-------|
| kube-flannel | pod/kube-flannel-ds-v9gsq | 0/1 | CrashLoopBackOff | 5 (22s ago) | 9m25s |
| kube-flannel | pod/kube-flannel-ds-w7rxd | 1/1 | Running | 0 | 11m |
| kube-system | pod/coredns-6d4b75cb6d-9j9gx | 1/1 | Running | 0 | 13m |
| kube-system | pod/coredns-6d4b75cb6d-rwkp4 | 1/1 | Running | 0 | 13m |
| kube-system | pod/etcd-tlntu-esc4000-g2-series | 1/1 | Running | 4 | 13m |
| kube-system | pod/kube-apiserver-tlntu-esc4000-g2-series | 1/1 | Running | 3 | 13m |
| kube-system | pod/kube-controller-manager-tlntu-esc4000-g2-series | 1/1 | Running | 3 | 13m |
| kube-system | pod/kube-proxy-trbsl | 1/1 | Running | 0 | 13m |
| kube-system | pod/kube-proxy-x26vx | 0/1 | CrashLoopBackOff | 5 (56s ago) | 9m25s |
| kube-system | pod/kube-scheduler-tlntu-esc4000-g2-series | 1/1 | Running | 3 | 13m |

| NAMESPACE | NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|-------------|--------------------|-----------|------------|-------------|------------------------|-----|
| default | service/kubernetes | ClusterIP | 10.96.0.1 | <none> | 443/TCP | 13m |
| kube-system | service/kube-dns | ClusterIP | 10.96.0.10 | <none> | 53/UDP,53/TCP,9153/TCP | 13m |

| NAMESPACE | NAME | DESIRED | CURRENT | READY | UP-TO-DATE | AVAILABLE | NODE SELECTOR | AGE |
|--------------|--------------------------------|---------|---------|-------|------------|-----------|------------------------|-----|
| kube-flannel | daemonset.apps/kube-flannel-ds | 2 | 2 | 1 | 2 | 1 | <none> | 11m |
| kube-system | daemonset.apps/kube-proxy | 2 | 2 | 1 | 2 | 1 | kubernetes.io/os=linux | 13m |

| NAMESPACE | NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|-------------|-------------------------|-------|------------|-----------|-----|
| kube-system | deployment.apps/coredns | 2/2 | 2 | 2 | 13m |

| NAMESPACE | NAME | DESIRED | CURRENT | READY | AGE |
|-------------|------------------------------------|---------|---------|-------|-----|
| kube-system | replicaset.apps/coredns-6d4b75cb6d | 2 | 2 | 2 | 13m |

Figure 4.8: Crashing of worker's kube-flannel and kube-proxy pod

4.5.3 Upgrading the worker node's OS version

Towards the end of the December break, it was communicated that the worker node would undergo an OS upgrade from Ubuntu 18.04 to Ubuntu 22.04. Following the upgrade, numerous deployment errors arose, resulting in repeated crashes of both the master and worker nodes. Suspecting a hidden network problem, it was suggested to switch to a single node cluster to complete the project. As the master node had successfully hosted all deployments previously on Ubuntu 18.04, this was the chosen configuration.

Chapter 5

Conclusion

Overall, the project met the objectives to deploy a working ASR system onto on-premise NTU school servers. The deployment came with many challenges, but a reasonable result is still achieved. This chapter summarises the project and discusses some of the possible future developments that the project could move towards.

5.1 Summary of project

In the course of the project, a Docker and K8s solution was conceptualised and realised on AWS and EC2 as well as the actual NTU servers. The ASR system which was deployed was functioning as expected, and there is room for scaling according to traffic workload.

Monitoring tools like Prometheus and Grafana was also deployed on the same cluster giving the cluster administrator a more visual cue on the state of the cluster.

Helm Charts are used in the project to improve the workflow by providing packaging and versioning. This allows better scalability if there are additional components which are needed in the K8s cluster in the future.

5.2 Future Works

The current state of the project involves a single node cluster due to compatibility issues with the OS version. However, there are several potential

areas for future development that could improve the scalability, maintainability, and security of the system.

One possible area for improvement is to expand the cluster to a multiple node configuration. This would allow for increased resilience and redundancy in case of hardware or software failures.

Another potential area for improvement is to include Continuous Delivery into the solution, making it easier to add new components into the cluster. This could involve automating the deployment process, enabling automatic testing and integration of new code, and providing feedback loops for developers.

Automated ways of provisioning the nodes could also be explored, such as using tools like Terraform or Ansible. This would help to reduce the manual effort required for setting up new nodes, and would also provide a more repeatable and consistent approach to infrastructure management.

Finally, it may be worth considering ways to enhance the security of the cluster, since security is not currently a major part of the project. This could involve implementing authentication and access controls, securing network communication channels, and regularly updating and patching the underlying software components.

Bibliography

- [1] G. Blogger, "Youtube user statistics 2022," Available at <https://www.globalmediainsight.com/blog/youtube-users-statistics/> (2022).
- [2] L. Ceci, "Hours of video uploaded to youtube every minute 2007-2020," Available at <https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/> (2023).
- [3] M. Mohsin, "10 youtube stats every marketer should know in 2022 [infographic]," Available at <https://sg.oberlo.com/blog/youtube-statistics> (2022).
- [4] D. K. L. Yeo, "Offline web subtitle editor," Available at https://dr.ntu.edu.sg/bitstream/10356/148184/2/desmond_fyp.pdf (2021).
- [5] D. P. et al., "The kaldi speech recognition toolkit," in *IEEE 2011 workshop on automatic speech recognition and understanding*, 2011.
- [6] C. Kan, "Doccloud: An elastic cloud platform for web applications based on docker," in *2016 18th international conference on advanced communication technology (ICACT)*, 2016, pp. 478–483.
- [7] R. Ashalatha and J. Agarkhed, "Evaluation of auto scaling and load balancing features in cloud," 2015.
- [8] R. B. B. Beach, S. Armentrout and E. Tsouris, "Relational database service," in *Pro Powershell for Amazon Web Services: Springer*, 2019, pp. 237–274.
- [9] K. Zettler, "What is cloud computing? an overview of the cloud," Available at <https://www.atlassian.com/microservices/cloud-computing>.
- [10] devasishakula503, "What is a distributed system?" Available at <https://www.geeksforgeeks.org/what-is-a-distributed-system/> (2022).

- [11] A. Aravamudhan, "Saas vs paas vs iaas: Examples, differences, & how to choose," Available at <https://www.eginnovations.com/blog/saas-vs-paas-vs-iaas-examples-differences-how-to-choose/>.
- [12] T. Cleo, "On premise vs. cloud: Key differences, benefits and risks," Available at <https://www.cleo.com/blog/knowledge-base-on-premise-vs-cloud>.
- [13] C. Hopping, "What is virtualisation?" Available at <https://www.itpro.co.uk/612016/what-is-virtualisation> (2022).
- [14] "What is orchestration?" Available at <https://www.redhat.com/en/topics/automation/what-is-orchestration> (2019).
- [15] R. Sheldon, "virtual machine (vm)," Available at <https://www.techtarget.com/searchitoperations/definition/virtual-machine-VM>.
- [16] T. O. et al., "Virtual machine image," Available at <https://www.sciencedirect.com/topics/computer-science/virtual-machine-image>.
- [17] "What is containerization?" Available at <https://www.ibm.com/sg-en/topics/containerization>.
- [18] "Docker overview," Available at <https://docs.docker.com/get-started/overview/>.
- [19] "Dockerfile reference," Available at <https://docs.docker.com/engine/reference/builder/>.
- [20] "Volumes," Available at <https://docs.docker.com/storage/volumes/>.
- [21] "Networking overview," Available at <https://docs.docker.com/network/>.
- [22] "Kubernetes overview," Available at <https://kubernetes.io/docs/concepts/overview/>.
- [23] "Kubernetes architecture: 11 core components explained," Available at <https://spot.io/resources/kubernetes-architecture/11-core-components-explained/>.
- [24] "Kubernetes pods," Available at <https://kubernetes.io/docs/concepts/workloads/pods/>.
- [25] "Kubernetes replicaset," Available at <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>.

- [26] "Kubernetes deployments," Available at <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>.
- [27] "Kubernetes service," Available at <https://kubernetes.io/docs/concepts/services-networking/service/>.
- [28] "Kubernetes ingress," Available at <https://kubernetes.io/docs/concepts/services-networking/ingress/>.
- [29] "Kubernetes ingress controllers," Available at <https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>.
- [30] "MetalLB," Available at <https://metallb.universe.tf/> (2021).
- [31] "Kubernetes persistent volumes," Available at <https://kubernetes.io/docs/concepts/storage/persistent-volumes/>.
- [32] "Kubernetes storage classes," Available at <https://kubernetes.io/docs/concepts/storage/storage-classes/>.
- [33] D. Merron and T. Idowu, "Introduction to kubernetes helm charts," Available at <https://www.bmc.com/blogs/kubernetes-helm-charts/> (2022).
- [34] "What is grafana?" Available at <https://www.redhat.com/en/topics/data-services/what-is-grafana> (2022).
- [35] "Prometheus monitoring: Use cases, metrics, and best practices," Available at <https://www.tigera.io/learn/guides/prometheus-monitoring/>.
- [36] "Prometheus vs grafana: Knowing the difference," Available at <https://www.opsramp.com/guides/prometheus-monitoring/prometheus-vs-grafana/>.
- [37] "Nginx ingress controller - how it works," Available at <https://kubernetes.github.io/ingress-nginx/how-it-works/>.
- [38] "Bare-metal considerations," Available at <https://kubernetes.github.io/ingress-nginx/deploy/baremetal/>.
- [39] "What is a websocket?" Available at <https://www.wallarm.com/what-a-simple-explanation-of-what-a-websocket-is>.
- [40] "Install docker engine on ubuntu," Available at <https://docs.docker.com/engine/install/ubuntu/>.

- [41] “Install tools — kubernetes,” Available at <https://kubernetes.io/docs/tasks/tools/>.
- [42] K. Cottart, “Ingresses and load balancers in kubernetes with metallb and nginx-ingress,” Available at [https://www.adaltas.com/en/2022/09/08/kubernetes-metallb-nginx/\(2022\)](https://www.adaltas.com/en/2022/09/08/kubernetes-metallb-nginx/(2022)).
- [43] B. Wilson, “How to setup prometheus monitoring on kubernetes cluster,” Available at [https://devopscube.com/setup-prometheus-monitoring-on-kubernetes/\(2023\)](https://devopscube.com/setup-prometheus-monitoring-on-kubernetes/(2023)).
- [44] —, “How to setup kube state metrics on kubernetes,” Available at [https://devopscube.com/setup-kube-state-metrics/\(2022\)](https://devopscube.com/setup-kube-state-metrics/(2022)).
- [45] —, “Setting up alert manager on kubernetes – beginners guide,” Available at [https://devopscube.com/alert-manager-kubernetes-guide/\(2022\)](https://devopscube.com/alert-manager-kubernetes-guide/(2022)).
- [46] devopscube, “How to setup prometheus node exporter on kubernetes,” Available at [https://devopscube.com/node-exporter-kubernetes/\(2021\)](https://devopscube.com/node-exporter-kubernetes/(2021)).
- [47] B. Wilson, “How to setup grafana on kubernetes,” Available at [https://devopscube.com/setup-grafana-kubernetes/\(2022\)](https://devopscube.com/setup-grafana-kubernetes/(2022)).
- [48] T. Sable, “Podsecuritypolicy deprecation: Past, present, and future,” Available at [https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/\(2021\)](https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/(2021)).
- [49] N. Corporation, “Installation guide,” Available at [https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html\(2023\)](https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html(2023)).
- [50] “Installation guide,” Available at <https://github.com/NVIDIA/k8s-device-plugin>.

Appendix A

Code Snippets

A.1 Testing Scripts

A.1.1 client3.py

```
import argparse
from ws4py.client.threadedclient import WebSocketClient
import time
import threading
import sys
import urllib.parse
import queue
import json
import time
import os
import datetime
import pyaudio
import ssl

FORMAT = pyaudio.paInt16
CHANNELS = 1
RATE = 16000
CHUNK = int(RATE / 10) # 100ms

import logging
import logging.handlers
# create logger
logger = logging.getLogger('client')
```

```

logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)
logfh = logging.handlers.RotatingFileHandler('client.log', maxBytes
      =10485760, backupCount=10)
logfh.setLevel(logging.DEBUG)

# create formatter
formatter = logging.Formatter(u'%(levelname)8s %(asctime)s %(message)s
      ')
logging._defaultFormatter = logging.Formatter(u"%(message)s")

# add formatter to ch
ch.setFormatter(formatter)
logfh.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)
logger.addHandler(logfh)

def rate_limited(maxPerSecond):
    minInterval = 1.0 / float(maxPerSecond)
    def decorate(func):
        lastTimeCalled = [0.0]
        def rate_limited_function(*args,**kwargs):
            elapsed = time.perf_counter() - lastTimeCalled[0]
            leftToWait = minInterval - elapsed
            if leftToWait>0:
                time.sleep(leftToWait)
            ret = func(*args,**kwargs)
            lastTimeCalled[0] = time.perf_counter()
            return ret
        return rate_limited_function
    return decorate

class MyClient(WebSocketClient):

```

```

def __init__(self, mode, audiofile, url, protocols=None, extensions=
None, heartbeat_freq=None, byterate=32000,
            save_adaptation_state_filename=None, ssl_options=None,
            send_adaptation_state_filename=None):
    super(MyClient, self).__init__(url, protocols, extensions,
        heartbeat_freq)
    print(url)
    self.final_hyps = []
    self.audiofile = audiofile
    self.byterate = byterate
    self.final_hyp_queue = queue.Queue()
    self.save_adaptation_state_filename =
        save_adaptation_state_filename
    self.send_adaptation_state_filename =
        send_adaptation_state_filename

    self.ssl_options = ssl_options or {}

    if self.scheme == "wss":
        # Prevent check_hostname requires server_hostname (ref #187)
        if "cert_reqs" not in self.ssl_options:
            self.ssl_options["cert_reqs"] = ssl.CERT_NONE

    self.mode = mode
    self.audio = pyaudio.PyAudio()
    self.isStop = False

@rate_limited(4)
def send_data(self, data):
    self.send(data, binary=True)

def opened(self):
    logger.info("Socket opened! " + self.__str__())
    def send_data_to_ws():
        if self.send_adaptation_state_filename is not None:
            logger.info("Sending adaptation state from %s" % self.
                send_adaptation_state_filename)
            try:
                adaptation_state_props = json.load(open(self.
                    send_adaptation_state_filename, "r"))

```

```

        self.send(json.dumps(dict(adaptation_state=
                                adaptation_state_props)))
    except:
        e = sys.exc_info()[0]
        logger.info("Failed to send adaptation state: %s" % e
                    )

    logger.info("Start transcribing...")
    if self.mode == 'stream':
        stream = self.audio.open(format=FORMAT, channels=CHANNELS
                                ,
                                rate=RATE, input=True,
                                frames_per_buffer=CHUNK)
        while not self.isStop:
            data = stream.read(int(self.byterate / 8),
                               exception_on_overflow=False)
            self.send_data(data) # send data

        stream.stop_stream()
        stream.close()
        self.audio.terminate()
    elif self.mode == 'file':
        with self.audiofile as audiostream:
            for block in iter(lambda: audiostream.read(int(self.
                byterate/4)), ""):
                print("Sending block of data " + str(len(block)))
                self.send_data(block)
                if (len(block) == 0):
                    break

    logger.info("Audio sent, now sending EOS")
    print("Sending EOS")
    self.send("EOS")

    t = threading.Thread(target=send_data_to_ws)
    t.start()

def received_message(self, m):
    response = json.loads(str(m))

```

```

logger.info(response)
if response['status'] == 0:
    if 'result' in response:
        trans = response['result']['hypotheses'][0]['transcript']
        if response['result']['final']:
            #print >> sys.stderr, trans,
            self.final_hyps.append(trans)

            #print("\033[H\033[J") # clear console for better
            output
            logger.info('%s' % trans)
        else:
            print_trans = trans
            if len(print_trans) > 80:
                print_trans = "... %s" % print_trans[-76:]

            #print("\033[H\033[J") # clear console for better
            output
            logger.info('%s' % print_trans)
    if 'adaptation_state' in response:
        if self.save_adaptation_state_filename:
            logger.info("Saving adaptation state to %s" % self.
                save_adaptation_state_filename)
            with open(self.save_adaptation_state_filename, "w")
                as f:
                    f.write(json.dumps(response['adaptation_state']))
        else:
            logger.info("Received error from server (status %d)" %
                response['status'])
            if 'message' in response:
                logger.info("Error message: %s" % response['message'])

def get_full_hyp(self, timeout=60):
    return self.final_hyp_queue.get(timeout)

def closed(self, code, reason=None):
    #print "Websocket closed() called"
    #print >> sys.stderr
    self.final_hyp_queue.put(" ".join(self.final_hyps))

```

```

def main():

    parser = argparse.ArgumentParser(description='Command line client
        for kaldigstserver')
    parser.add_argument('-o', '--option', default="file", dest="mode",
        help="Mode of transcribing: audio file or streaming")
    parser.add_argument('-u', '--uri', default="ws://localhost:8888/
        client/ws/speech", dest="uri", help="Server websocket URI")
    parser.add_argument('-r', '--rate', default=32000, dest="rate", type
        =int, help="Rate in bytes/sec at which audio should be sent to
        the server. NB! For raw 16-bit audio it must be 2*samplerate!")
    parser.add_argument('-t', '--token', default="", dest="token", help=
        "User token")
    parser.add_argument('-m', '--model', default=None, dest="model",
        help="model in azure container")
    parser.add_argument('--save-adaptation-state', help="Save adaptation
        state to file")
    parser.add_argument('--send-adaptation-state', help="Send adaptation
        state from file")
    parser.add_argument('--content-type', default='', help="Use the
        specified content type (empty by default, for raw files the
        default is audio/x-raw, layout=(string)interleaved, rate=(int)<
        rate>, format=(string)S16LE, channels=(int)1")
    parser.add_argument('audiofile', nargs='?', help="Audio file to be
        sent to the server", type=argparse.FileType('rb'), default=sys.
        stdin)
    args = parser.parse_args()

    if args.mode == 'file' or args.mode == 'stream':
        content_type = args.content_type
        if content_type == '' and args.audiofile.name.endswith(".raw")
            or args.mode == 'stream':
            content_type = "audio/x-raw, layout=(string)interleaved,
                rate=(int)%d, format=(string)S16LE, channels=(int)1" %(
                args.rate/2)

    ws = MyClient(args.mode, args.audiofile, args.uri + '?%s' % (
        urllib.parse.urlencode([("content-type", content_type)])) + '

```

```

&%s' % (urllib.parse.urlencode([("token", args.token)])) + '
&%s' % (urllib.parse.urlencode([("token", args.token)])) + '
&%s' % (urllib.parse.urlencode([("model", args.model)])),
byterate=args.rate,
    save_adaptation_state_filename=args.
        save_adaptation_state,
        send_adaptation_state_filename=args.
            send_adaptation_state)

ws.connect()
result = ws.get_full_hyp()

logger.info("\n URL: " + str(args.uri + '?%s' % (urllib.parse.
    urlencode([("content-type", content_type)])) + '?%s' % (
        urllib.parse.urlencode([("token", args.token)]))) + "\n")
logger.info("\n-----\nFinal Result: \n")
logger.info(result)
else:
    print('\nTranscribe mode must be file or stream!\n')

if __name__ == "__main__":
    main()

```

A.2 Docker Configuration Files

A.2.1 docker-compose.yml

```

---
version: '2.1'
services:
    decoding-sdk-server:
        image: lyvt/decoding-sdk:1.2
        restart: unless-stopped
        ports:
            - "8010:8010"
        command: /home/speechuser/start_master.sh -p 8010

```

```

decoding-sdk-worker:
  image: lyvt/decoding-sdk:1.2
  restart: unless-stopped
  volumes:
    # mapping volumes
    - ../models/online-models/:/opt/models/
  environment:
    - MASTER=decoding-sdk-server:8010
    - MODEL_DIR=english
    - INSTANCE_NUM=2

  command: /home/speechuser/start_worker.sh

```

A.3 Setup Scripts

A.3.1 network.sh

```

#!/usr/bin/env bash

sudo apt-get update
sudo modprobe br_netfilter

cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
br_netfilter
EOF

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF

sudo sysctl --system
cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf
overlay
br_netfilter
EOF

```



```

sudo modprobe overlay
sudo modprobe br_netfilter

# Setup required sysctl params, these persist across reboots.
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

# Apply sysctl params without reboot
sudo sysctl --system

```

A.3.2 kube.sh

```

#!/usr/bin/env bash

lsb_release -a
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg
     https://packages.cloud.google.com/apt/doc/apt-key.gpg
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg
     ] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc
     /apt/sources.list.d/kubernetes.list

sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl

if [ "$1" = 'master' ]; then
    sudo apt-mark hold kubelet kubeadm kubectl
    sudo systemctl enable --now kubelet
    sudo kubeadm init --pod-network-cidr=10.244.0.0/16 --cri-socket=/run/
        containerd/containerd.sock > kube_details.txt

    mkdir -p $HOME/.kube
    sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
    #vim ~/.kube/config
    sudo chown -R $(id -u):$(id -g) $HOME/.kube/config

```

```

kubect1 taint nodes --all node-role.kubernetes.io/control-plane- #
    kubect1 taint nodes --all node-role.kubernetes.io/master-
kubect1 get nodes
kubect1 apply -f https://raw.githubusercontent.com/coreos/flannel/
    master/Documentation/kube-flannel.yml
kubect1 get nodes

elif [ "$1" = 'worker' ]; then
    sudo systemctl enable --now kubelet
    # joining command have to get from the master node

fi

```

A.4 On-premise Deployment Configuration Files

A.4.1 deployments/server.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: on-prem-server
spec:
  replicas: 1
  selector:
    matchLabels:
      podName: on-prem-server-pod
  template:
    metadata:
      labels:
        podName: on-prem-server-pod
    spec:
      containers:
        - name: on-prem-server-container
          image: lyvt/decoding-sdk:1.2
          imagePullPolicy: Always
          ports:
            - containerPort: 8010

```

```

    args:
      - /home/speechuser/start_master.sh
      - -p
      - "8010"
    resources:
      limits:
        memory: 6Gi
      requests:
        cpu: 1000m
        memory: 6Gi
    restartPolicy: Always
status: {}

```

A.4.2 deployments/worker.yaml

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: on-prem-worker
spec:
  replicas: 1
  selector:
    matchLabels:
      podName: on-prem-worker-pod
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        podName: on-prem-worker-pod
    spec:
      containers:
        - name: on-prem-worker-container
          image: lyvt/decoding-sdk:1.2
          imagePullPolicy: Always
          args:
            - /home/speechuser/start_worker.sh

```

```

env:
  - name: MASTER
    value: on-prem-server-service:8010
  - name: MODEL_DIR
    value: english
  - name: INSTANCE_NUM
    value: "2"
volumeMounts:
  - mountPath: /opt/models/
    name: on-prem-worker-pvc
resources:
  limits:
    memory: 6Gi
  requests:
    cpu: 1000m
    memory: 6Gi
restartPolicy: Always
volumes:
  - name: on-prem-worker-pvc
    persistentVolumeClaim:
      claimName: on-prem-worker-pvc
status: {}

```

A.4.3 ingresses/server-resource.yaml

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: on-prem-server-ingress-resource
spec:
  ingressClassName: nginx
  rules:
    - host: asr-project.com
      http:
        paths:
          - path: /
            pathType: Prefix

```

```
    backend:
      service:
        name: on-prem-server-service
        port:
          number: 8010
```

A.4.4 services/server.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: on-prem-server-service
spec:
  selector:
    podName: on-prem-server-pod
  ports:
    - name: "8010"
      port: 8010
      targetPort: 8010
  status:
    loadBalancer:
      ingress:
        - ip: 172.21.47.111 # ip of the endpoints
```

A.4.5 volumes/worker-pv.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: on-prem-worker-pv
spec:
  storageClassName: on-prem-worker-storageclass
  claimRef:
    name: on-prem-worker-pvc
    namespace: default
  capacity:
```

```

    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  local:
    path: /home/fyp-jonathan/worker1-dir/models
  # Assign this pod to the worker-node
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - micl-server03

```

A.4.6 volumes/worker-pvc.yaml

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: on-prem-worker-pvc
spec:
  storageClassName: on-prem-worker-storageclass
  volumeName: on-prem-worker-pv
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
status: {}

```

A.4.7 volumes/worker-storageclass.yaml

```

kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:

```

name: on-prem-worker-storageclass
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer