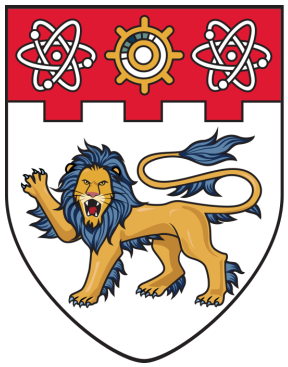

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Nanyang Technological University
School of Computer Science and Engineering

Final Year Project

Docker and Kubernetes: Deploying Speech Recognition System for Scalability

Final Report

By

Ma Xiao (U1722964K)

Supervisor: Associate Professor Chng Eng Siong

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

March 2021

Abstract

This project aims to provide a robust solution for deploying an existing automatic speech recognition (ASR) system. The deployment solutions will enable the system to handle multiple requests concurrently and support dynamically changing workload. The deployment solution is constructed using containerisation deployment technology: Docker and Kubernetes. The solution is implemented on Amazon Web Service (AWS) cloud platform and mainly use Elastic Kubernetes Service (EKS) on the platform.

The solution ensures the auto-scaling of the system and even load balancing between the requests to computational resources to provide higher availability, and ensures the security of the application and the computational resources. This report will present the solution in terms of the designed architecture diagram, detailed illustration of important components and the implementation steps. The report will also demonstrate the experiments result that showing the performance of the solution and evaluate the cost to implement the solution.

Acknowledgements

This project was made possible thanks to the support and guidance of the following people, who have helped me a lot during the project. I would like to express my deepest gratitude and appreciation to them.

First and foremost, I would like to express my appreciation to my supervisor, Associate Professor Chng Eng Siong for his valuable guidance and advice on the specifications of the project solution. He also provided essential resources used during the implementation of the solution. Although he is busy with his research works and projects, he is always patient to hear from the progress of this project and give previous advice.

I would like to express my appreciation to my mentor, Research Staff Vu Thi Ly for her valuable advice on the design and implementation of this project. She also helped me in liaison with the outside client to get more requirements for the solutions.

Last but not least, I would like to thank my family and all my friends who gave me support throughout the project.

Contents

List of Figures

Chapter 1

Overview

There is an existing automatic speech recognition (ASR) system developed based on an open-source toolkit, Kaldi which applies a finite-state transducer to conduct the speech recognition [1]. The ASR system takes the input of audio files or real-time audio streams and outputs the transcripts.

A group of dedicated researchers from NTU has developed some speech recognition models for the ASR system to transcribe audio with different languages to transcripts. They also developed models that could transcribe a mix of languages such as the mix of Mandarin and English. This model has the potential to be widely applied in Singapore, where the majority of people usually speak a mix of Mandarin and English. There are few other code-switching [1] ASR systems developed in the last few years and the ASR system with a language model that could transcribe a mixed of Mandarin and English developed by NTU SpeechLab is the pioneer product in this area. The ASR

[1] Able to transcribe a mix of languages

system is targeted to support various use cases in Singapore. For example, call centres, customer service centres etc.

1.1 Background

The ASR system consists of masters and workers. A master is used to receive the transcribing request from the user and response user with the transcript. A master does not do the transcribing job, instead, it passes transcribing the job to a worker that is connected to it. A worker conducts the transcribing job and communicates only with the master which it is connected to. Figure 1.1 shows the relationships between an ASR master and five ASR workers running with different language models.

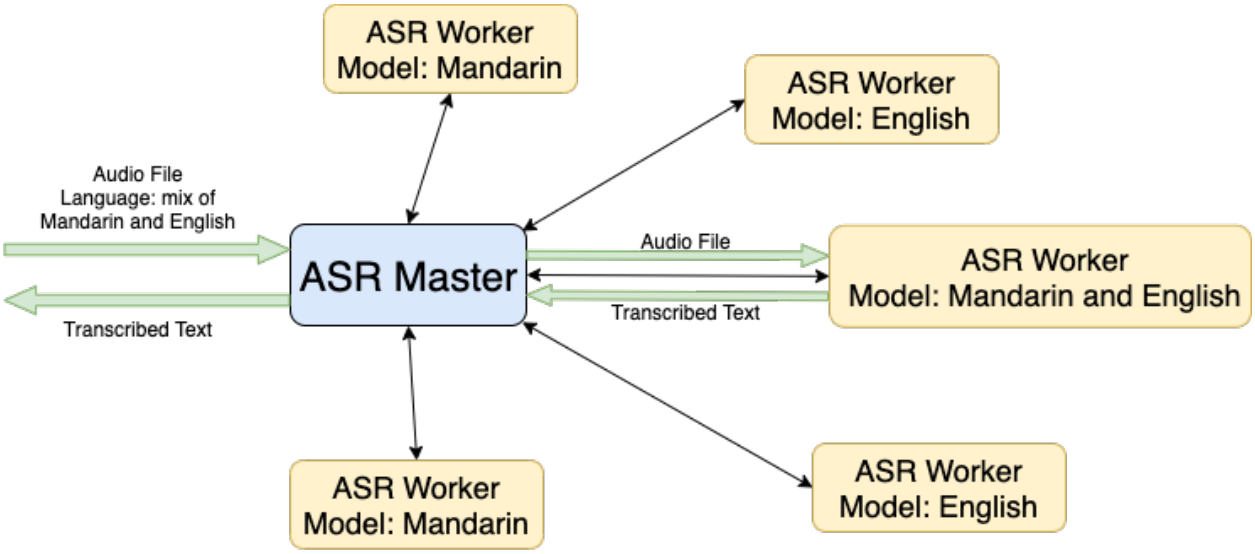

Figure 1.1: ASR Master and Workers

Each ASR worker must load a language model to transcribe and can only do transcribing job of that language. After an ASR worker process starts

running, it will establish a connection with an ASR master. Multiple ASR workers can connect to the same ASR master. A request sent to an ASR master specifies a language and an audio file or stream. When the ASR master receives a request, it delegates the transcribing job to an ASR worker of that language.

1.2 Importance of the Project and Project Objectives

Applying the ASR system will greatly help call centres mitigate the after-call-work and human intervention in the entire workflow. Usually, the call operator has to take notes or transcribe the audio calls into transcripts. These transcripts can be used for the archiving purpose or further downstream tasks such as analytic. Without the ASR system, call centres need to manually transcribe all the call audios to transcripts, which is costly and time-consuming.

However, a limitation of the ASR system is that each ASR worker can only process one transcribing job at the same time. This means if there is only one ASR worker instance available, the system cannot process multiple transcribing requests concurrently. This results in a problem in commercial applications where applications need to support multiple concurrent requests and the number of concurrent requests is usually varying according to the time period. Therefore, it is important to develop a deployment solution of the ASR system to ensure high availability to support commercial usage.

Previous work done by another Final Year Project student, Wong Seng Wee, was successful in the basic scaling of the system. However, the solution has issues with load balancing and results in inefficient usage of the computational resources and bad performance in availability. Another issue with the previous solution is that there few mechanisms to ensure security, while security is one of the top concerns in a commercial application. Last but not least, the solution proposed by the previous student is only implemented on Microsoft Azure. However, there is a high demand to deploy the system on Amazon Web Service (AWS) nowadays. Therefore, a robust deployment solution that ensures security and high availability and implemented on AWS is needed.

1.3 Scope

The objectives of this project are to propose and implement an industrial-level robust deployment solution of the ASR system to ensure auto-scaling with high availability and the security of the computational resources and the application. The solution will utilise the latest containerised technology: Docker and Kubernetes and will be implemented on Amazon Web Service which is currently the most in-demand cloud platform [32].

This project will not explain the details of the ASR system.

1.4 Report Organisation

This report contains six chapters and an overview of each chapter is as follows:

- Chapter 1 provides an overview of the project and explains the motivation and the scope of this project.
- Chapter 2 summarises the technologies applied during the design and implementation of the solution proposed by this project.
- Chapter 3 elaborates the details of the design of the solution.
- Chapter 4 focus on solution implementation.
- Chapter 5 demonstrates the system performance and cost estimation.
- Chapter 6 concludes the project and proposes possible future work.

Chapter 2

Literature review

The solution proposed in this project is designed based on containerisation technology and is implemented by utilising cloud computing. Figure 2.1 shows the system technology stack.

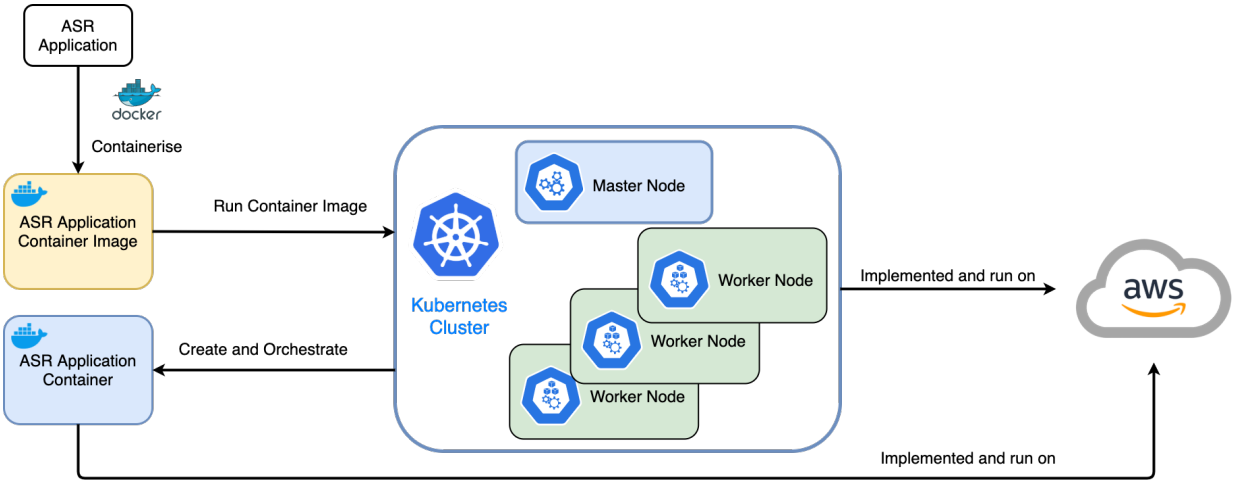

Figure 2.1: System Technology Stack: Docker, Kubernetes, and Amazon Web Service

The solution utilises Docker, a software packaging and delivery platform [33], to containerise the application. The solution utilises Kubernetes, a container orchestration platform [33], to support self-healing and automates the pro-

cess of scaling, managing and updating the containerised application. The solution is implemented on Amazon Web Service (AWS), a cloud computing platform, to simplify the management process of the Kubernetes cluster and ensure the flexibility of the computational resources. The following sections are detailed explanations of the applied technologies.

2.1 Containerisation

2.1.1 Containerisation

During the old days, software were deployed and run on physical machines. When software engineers or system administrators needed to deploy software, there was a lot of manual work to ensure the correct configuration of the software and create the correct running environment for the software. The execution environment includes the operating system, environment variables, dependencies, etc. Therefore, when the software was deployed or transferred to another host, the process often prone to result in bugs and errors. Besides, if multiple software were deployed on the same physical machine, there was no isolation mechanism to ensure software configurations and dependencies do not conflict with others. Furthermore, there were no resources boundaries and some software would under-performed if other software took up most of the resources.

To resolve the above issues, virtualisation was introduced. Virtualisation allows multiple Virtual Machines (VMs) to be run on a single physical server.

Virtualisation provides resource boundaries and ensures environment isolation between software deployed on the same physical machines. However, Each VM runs its own operating system, which is a big overhead to achieve the above functionalities.

Containerisation takes one more step further from virtualisation. Containerisation allows software to be run in containers. Similar to VMs, containers encapsulate software code and all its dependencies so that it can run uniformly and consistently on any infrastructure [10]. However, when multiple containers run on the same machines, they share the same operating system therefore, compared to VMs, containers are lightweight. Containerisation technology is quickly maturing and resulting in benefits for both software developers and operations teams.

2.1.2 Container

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. [26] A running container is a process running on the host that runs the container runtime, and it is completely isolated from both the host and all other process running on that host. Figure 2.2 demonstrates containers and isolation of containers that running on the same machine.

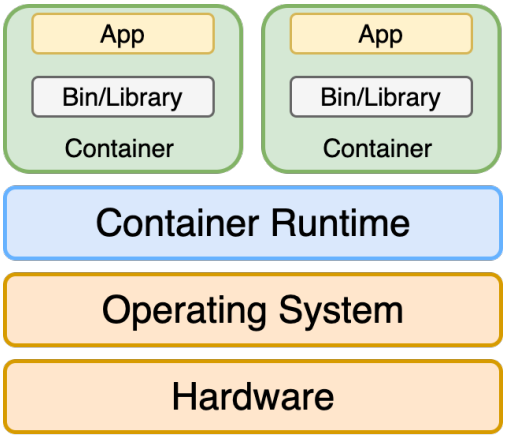

Figure 2.2: Containers and Containers Deployment

2.1.3 Image

A container image is a lightweight, standalone, executable package of software that includes the source and its required environment [26]. It also contains the filesystem that will be available to the application. A container image becomes a container at runtime.

2.1.4 Registry

A registry is a repository that stores the container images. It enables easy sharing of container images between developers and computers. Developers push the images to a registry after building an image from a computer and pull the images from that registry from another computer. Some registries are public where anyone can pull images from them while others are private and only accessible to a certain group of people [28].

2.2 Docker

Docker is a software platform that allows you to build, test, and deploy applications quickly. Docker packages software into containers which include the software and everything the software needs for it to run properly [25]. Docker provides elegant solutions to the common problems faced in software deployment such as installing, removing, upgrading, etc [27]. Currently, Docker is the most popular container runtime and it has good integration with popular cloud platforms such as Amazon Web Service and Microsoft Azure.

2.3 Kubernetes

Kubernetes is an open-sourced container orchestration platform that manages containerized workloads and services facilitating both declarative configuration and automation [9].

Containerisation technology ensures the easy packaging of the software while Kubernetes ensures the easy management of the containers. Kubernetes provides easier solutions for scaling containers, managing the computational resources used by containers, and container self-healing.

2.3.1 Kubernetes Cluster Components

Kubernetes abstract away the actual hardware resources and exposing them as a single platform for developing and running apps [29]. Users can easily manage all the resources in Kubernetes via API. When you deploy your application using Kubernetes, you get a Kubernetes cluster. A Kubernetes cluster is composed of many nodes (a node is a machine within a Kubernetes cluster) at the hardware level, which can be split into two types: [30]:

- Master Node: Hosts the *Kubernetes Control Plane* that manages the whole Kubernetes system.

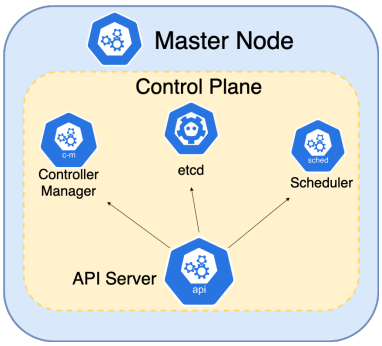

Figure 2.3: Kubernetes Master Node

- Worker Node: Hosts the deployed application.

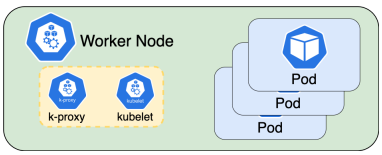

Figure 2.4: Kubernetes Worker Node

Figure 2.5 shows the architecture of a Kubernetes cluster.

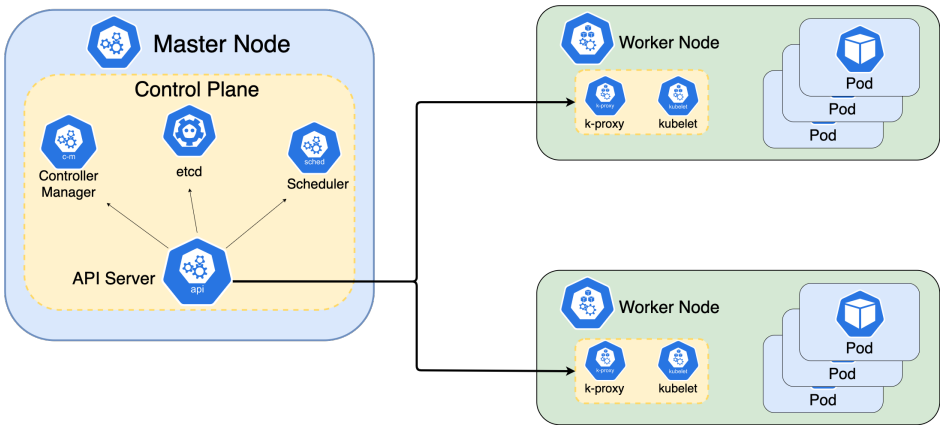

Figure 2.5: Kubernetes Cluster Architecture

The below sub-sections introduces several Kubernetes resources that construct the main part of the deployment solution proposed in this project.

2.3.2 Kubernetes Resources

Pod

A Pod is a group (one or more) of co-located and co-scheduled containers with shared storage and network resources. A Pod represents a basic building block in Kubernetes and it is the smallest deployable units in a Kubernetes cluster [7]. Although it is possible to run multiple containers inside the same Pod, it is recommended that containers run inside the same Pod only if these containers are tightly coupled. The "one-container-per-Pod" model is recommended and it is a common practice when using Kubernetes.

Each Pod must specify how many computational resources to be provisioned to run it. When Kubernetes runs a Pod, it will pick a worker node that has a sufficient amount of available resources to run the Pod and schedule the

Pod onto that worker node. A Pod must be scheduled on a particular worker node. A single Pod is not allowed to run on multiple worker nodes.

Service

Service is a resource to provide a consistent entry to a group of Pods. Expose an application on a set of Pods as a network service is a common practice. Each Pod is assigned to an IP address after it is scheduled to run and before it starts running. In a Kubernetes cluster, Pods are horizontally scaled and each Pod has its own IP address, and a Pod can sometimes down and be replaced by another Pod. Therefore, it is impossible to keep track of all the currently available IP addresses to access the Pods because those IP addresses are dynamically changing within a cluster. Using Services to manage access to every Pod in real-time and expose the access to Pods as one single endpoint is a more practical choice for a large scale system [5] [31]. There are four types of Service and each of them has different use cases, the default Service type is **ClusterIP**.

- **ClusterIP**: Exposes the Service on a cluster internal IP. This type of Service is only reachable from within the cluster.
- **NodePort**: Kubernetes reserve a port on all the worker nodes. By creating a **NodePort** Service, a **ClusterIP** Service is automatically created. **NodePort** Service can be accessed within the cluster using the internal cluster IP and also can be accessed outside the cluster using the <NodeIP>:<NodePort>.

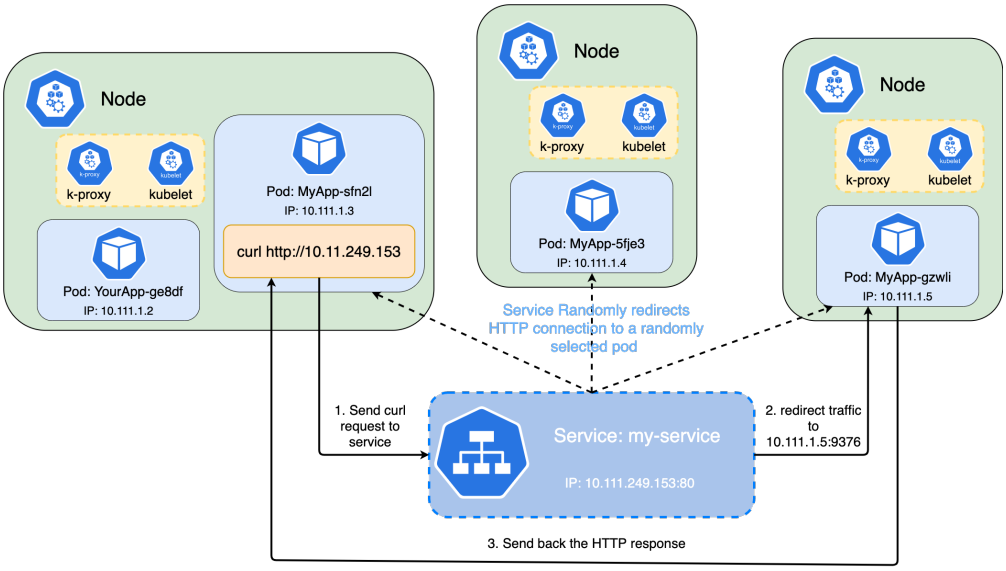

Figure 2.6: Cluster IP Service

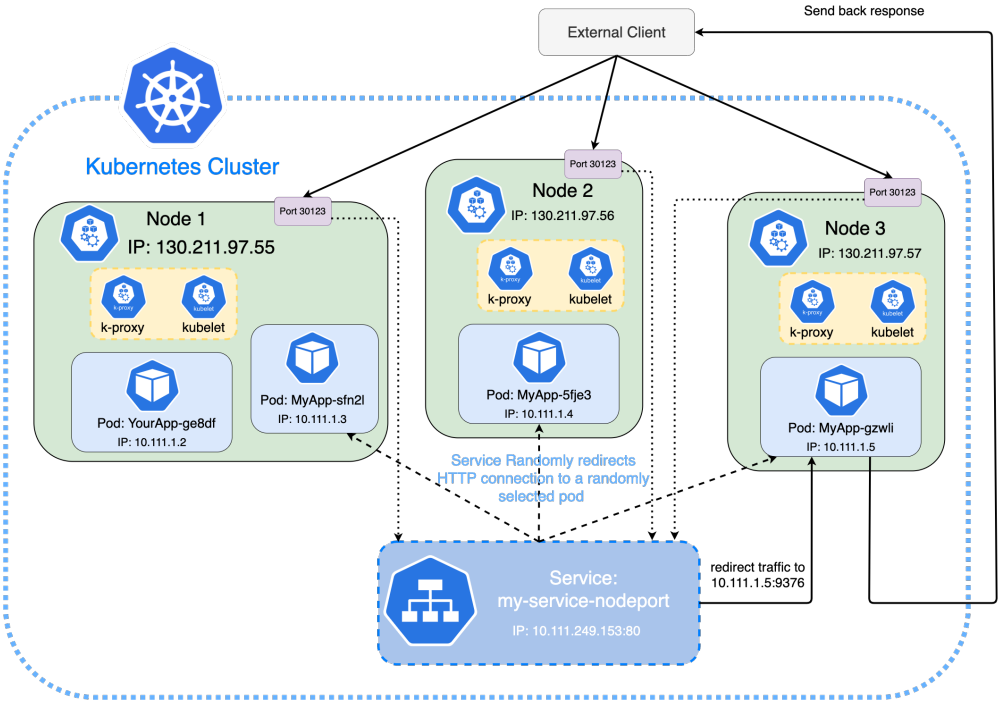

Figure 2.7: NodePort Service

- **LoadBalancer**: Exposes the Service externally using a cloud provider's load balancer. The load balancer will be assigned to a unique, publicly accessible IP address and will redirect all connections to the Service.

NodePort and **ClusterIP** Services are automatically created.

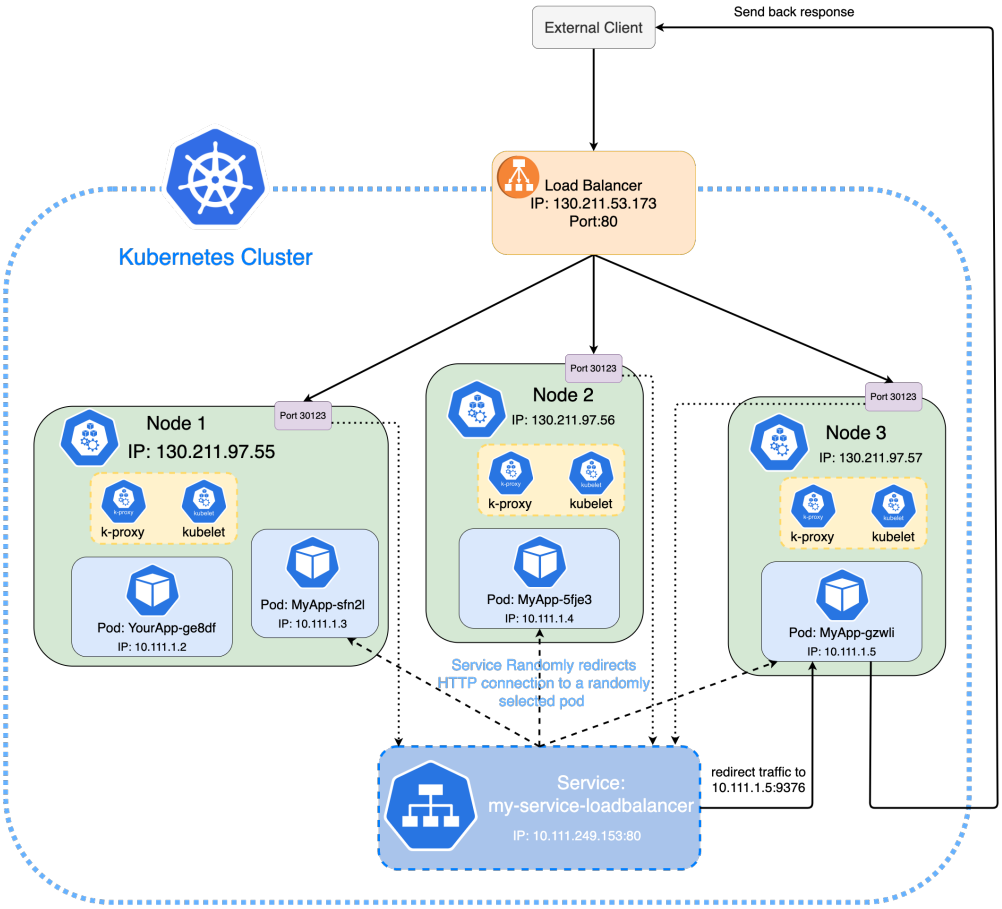

Figure 2.8: LoadBalancer Service

- **ExternalName**: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a *CNAME* record with its value. No proxying of any kind is set up.

Ingress

To provision external access to the cluster, creating NodePort and LoadBalancer Services are two methods to adopt but another method exists, that is to create an Ingress resource. Ingress is an API object that manages external

access to the Services in a cluster, typically HTTP [3]. An Ingress Controller needs to be running in the cluster to use Ingress.

When using multiple LoadBalancer Services to provide external access, each Service requires to create a load balancer, which leads to resources wastage. The advantage of Ingress is that when an Ingress providing access to dozens of Services, only one load balancer is created. When a client sends an HTTP request to the Ingress, the Ingress determines which Service the request is forwarded to base on the host and path specified in the request.

Ingress also plays an important role in a Kubernetes cluster to ensure the load balancing of a Service. Ingress Controller does not forward the request to the Service. It only used Endpoint resources attached to the Service to select a Pod. How the Ingress Controller selects a Pod depends on the load balancing algorithm of the Ingress Controller. Therefore, the load balancing algorithms can be configured. Figure 2.9 shows an Ingress resource providing access to three Services.

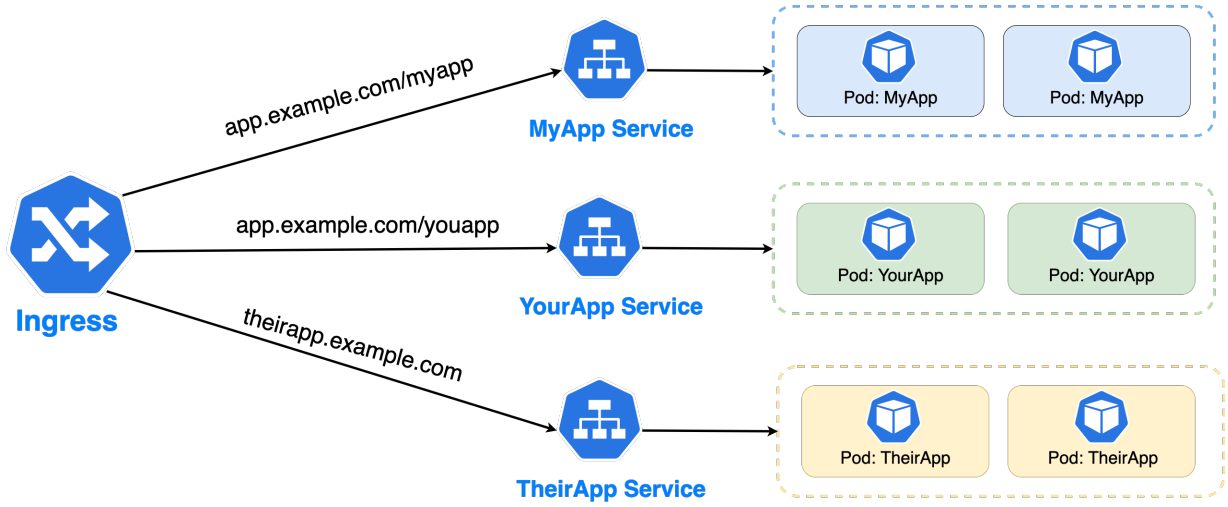

Figure 2.9: Kubernetes Ingress Demonstration

Persistent Volumes

Pods are ephemeral in nature, which means if a Pods is terminated because of some reasons, the Volume inside the Pod is deleted as well. A PersistentVolume (PV), similar to Volume [8], is a piece of storage in the cluster but have a lifecycle independent of any individual Pod that uses the PV. Therefore, PersistentVolume resource is a solution to persistent data that beyond Pods' lifecycles and PersistentVolume resources are used to store data shared by a group of Pods.

A PersistentVolumeClaim (PVC) is a request for PersistentVolume resources and are used by Pods. A PVC can request specific storage size and access modes (ReadWriteOnce, ReadOnlyMany or ReadWriteMany) [6]. When Pods need to consume storage in a PV, it will refer to a PVC that is bound to a PV, instead of referencing the PV directly. The purpose is to abstract the storage implementation from Pods. Figure 2.10 shows the process of using Persistent Volume Claim and Persistent Volume resources for Pods.

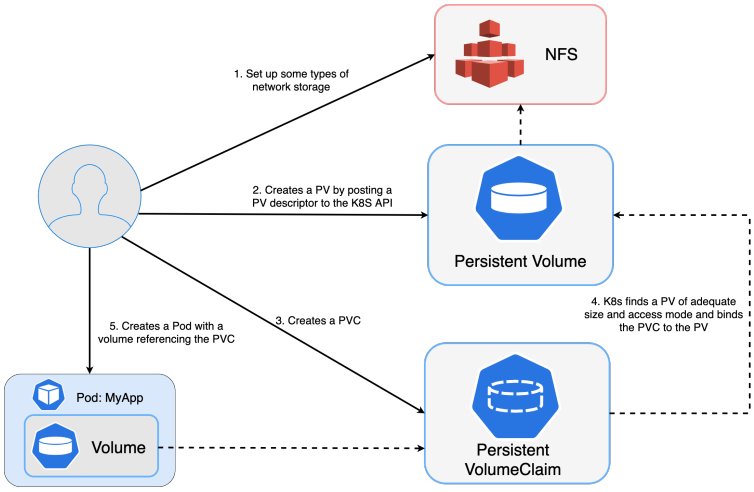

Figure 2.10: Kubernetes Persistent Volume Usage Process

Jobs

Job is a type of workload resources in a Kubernetes cluster. A Job creates one or more Pods and ensures them successfully run and completes. Pods created by a Job resource are configured to complete or terminate after it has finished running the container process. Jobs are usually useful in facing unexpected increasing workloads of the system.

2.4 Cloud Computing

Cloud computing is the delivery of computing services with pay-as-you-go pricing over the internet. Instead of owning and maintaining physical data centres and servers, users can access computational resources like servers, storage, networking, databases, etc. Other than the advantage of cost-saving by reducing the capital expenses of infrastructure setup, cloud computing offers the following benefits to users [24]:

- **Agility**: Instead of spending time on setting up and configuring infrastructure, the cloud enables users to spin up a wide range of computing services such as compute, storage, and database quickly. This feature provides agility and freedom for users to test and experiment with various technologies with their solutions.
- **Elasticity**: Cloud computing does not over-provision resources in order to handle the peak workload of a hosted system. The resource provi-

sioned by the cloud could be elastically scaled up and shrink down as the workload changes.

- **Deploy globally in minutes**: Cloud computing offers users computational resources host globally, therefore, users could expand the system to new geographic regions quickly. Putting the deployed system closer to end-users will reduce latency and improve user experience.

2.5 Amazon Web Service

Amazon Web Service (AWS) is a cloud service provider. AWS provides a comprehensive range of services that help users build applications faster and solve common IT production problems.

2.5.1 Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) provides user scalable computing capacity. EC2 offers various computations resources including but not limited to:

- Virtual computing environments (instances).
- Pre-configured templates for the instances known as Amazon Machine Images.

- Various configurations of CPU, memory, storage, and networking capacity.
- Secure login information using key pairs.
- Virtual firewalls that control the network traffic to the instances.

2.5.2 Virtual Private Cloud

Amazon Virtual Private Cloud (VPC) is a service that enables users to define a virtual network and launch resources into the network. Amazon VPC is the networking layer of Amazon EC2 [22]. The following are key concepts for VPCs:

- Virtual private cloud: A virtual network defined by a user and is logically isolated from other virtual networks in the AWS cloud [18].
- Subnet: A range of IP addresses in a VPC. AWS resources could be launched into a specific subnet. Launch resources that must be connected to the internet to a public subnet, and Launch resources that will not be connected to the internet to a private subnet [18].
- Route table: A set of routes, which are rules determining where network traffic is directed.
- Internet gateway: A gateway attached to a VPC to enable communication between resources in the VPC and the internet.

Chapter 3

Designed Approach

Figure 3.1 shows the architecture of the proposed solution. The architecture diagram illustrates the Kubernetes components and Kubernetes resources that are crucial to achieving the major functionality of the system. The diagram does not include all the Kubernetes resources and all the detailed interactions between different resources. Kubernetes resources used in the designed solution and interactions between them will be demonstrated in the following sections in this chapter.

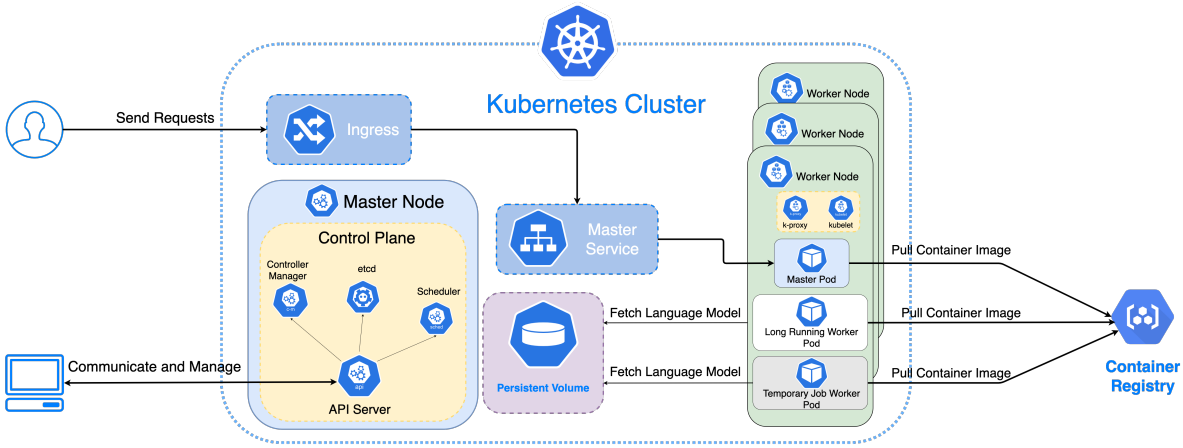

Figure 3.1: Structural architecture diagram of the proposed solution

3.1 Basic Structural Components

3.1.1 Master Pods and Worker Pods

The ASR application contains two parts, masters and workers. In the proposed solution, the ASR masters and ASR workers are deployed separately by creating different kinds of Pods in a Kubernetes cluster: master Pods and worker Pods. Master Pods run the ASR master process while the worker Pods run the ASR worker process, i.e. transcribing process. In the real situation, considering Kubernetes abstracts the underlying hardware as uniform computational resources. Therefore, it is possible that multiple master Pods and worker Pods can be running on the same node, see figure 3.2.

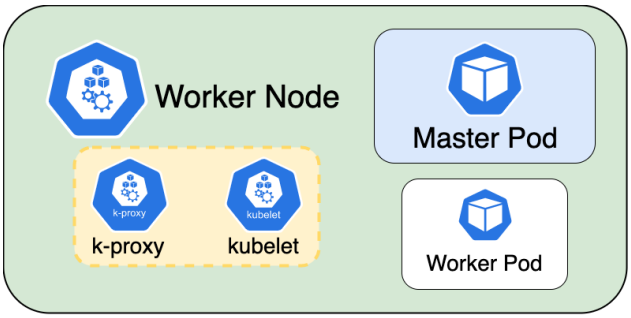

Figure 3.2: A worker node runs a master Pod and a worker Pod

Do not confuse the master Pods and worker Pods with the master node and worker nodes in the cluster. Master Pods and worker Pods are resources in the deployed ASR system, while master nodes and worker nodes are components of a generic Kubernetes cluster. All the master Pods and worker Pods are running on the worker nodes of the cluster [1].

[1]Since the worker node group is the only computational resource in the Kubernetes cluster, the worker node is the only place where all the Pods run.

3.1.2 Persistent Volumes

Persistent Volumes is used to host the language models used by the ASR application. Every ASR worker in the application needs a language model to do the decoding work. Language models are very large in size, usually above 2GB, therefore it will take a long time if the worker Pods download the language model when starting.

Persistent Volume is set up and host all the models needed in the system. As a property of the Persistent Volume shows: the Persistent Volume can exist beyond the worker nodes' life-cycles. Therefore, the content in the Persistent Volume will always be there regardless of node shuts down or spins up. The Persistent Volume is mounted to every Pod, which provides worker Pods quick access to the models and worker Pods spin up speed is improved. Figure 3.3 demonstrates this process.

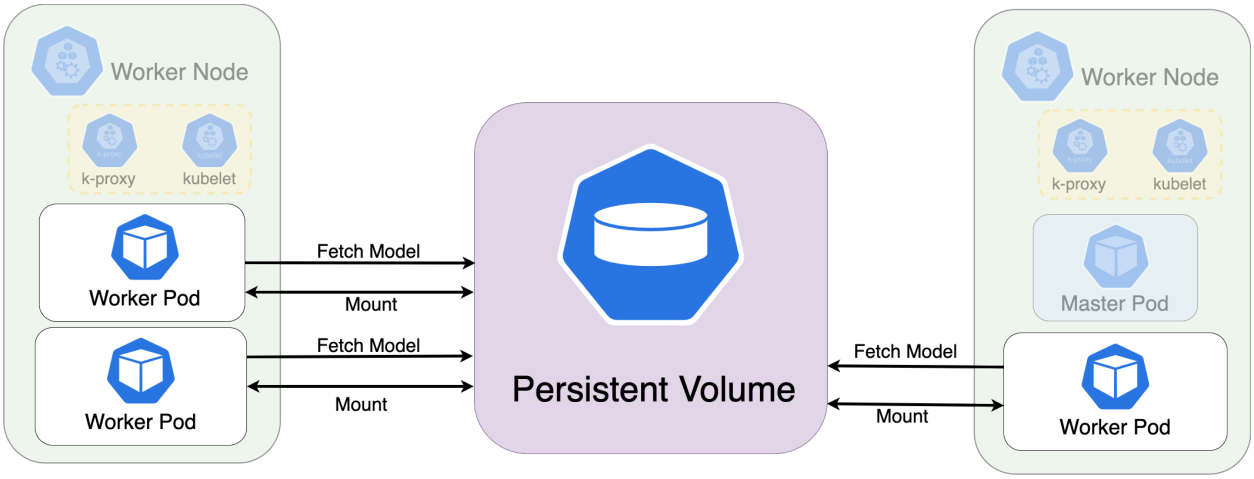

Figure 3.3: Persistent Volume and Pods Mount

3.1.3 Container Registry

Pods are running containers. To get a container, you need to run a container image. Therefore, the solutions should include the source of the image, that is the container registry.

In this deployed system, the ASR application is packaged into a Docker image. The master process and worker process are included in the same image. Therefore master Pods and worker Pods run the same container image but with different run command. The Docker image is uploaded to a central Container Registry. All the Pods within the created Kubernetes Cluster can pull the Docker image from the Container Registry. Figure 3.4 demonstrates the process of Pods pulling the image from a container registry.

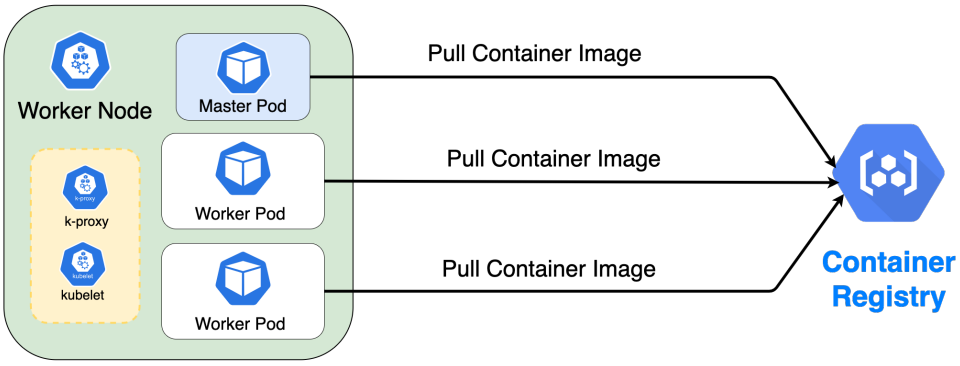

Figure 3.4: Pods pull images from a central container registry

3.2 Resolve Load Balancing Issues with Ingress

The solution proposed by the previous student has issues with load balancing. The previous solution used the **LoadBalancer** Service type. By default, a

Service in a cluster will distribute traffic **randomly** to Pods behind it. This can lead to a problem in our system: when multiple master Pods are running in the cluster, the master Service may direct incoming traffic to a master Pod that have no available worker Pods connected, while other master Pods have available worker Pods and ready to receive requests. This can result in low utilization of the resource and even worse, harm the availability of the system.

3.2.1 Demonstration of the Load Balancing Issue

This demonstrates the situation where no proper load balancer is set up. In this case, the request will sometimes be rejected even there are available workers.

System Set-up

In this experiment, two running master Pods and three running worker Pods are deployed. Master Pods are exposed as one master Service. Figure 3.5 shows the relationships between master Pods and worker Pods: 3 workers join 2 masters, one of the masters have 1 worker attached, the other has 2 workers attached. Note that in the system, each master Pod and each worker Pod run a single container, which means one worker Pod is one running decoder worker instance.

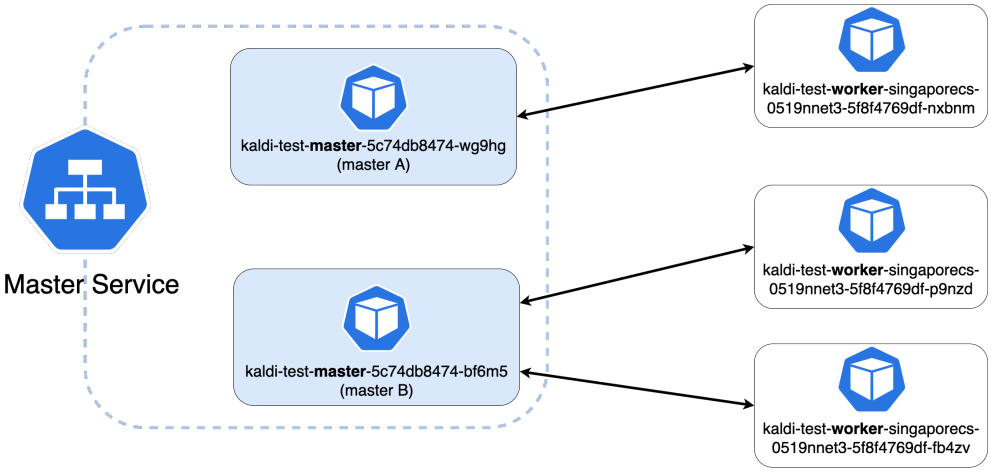

Figure 3.5: Relationships between master Pods and worker Pods

Experiment

In this experiment, 3 requests will be sent to the system and the behaviour (which Pod accept the incoming request) of the system is examined. All requests are sent to the master Service. The below figures show the status of worker Pods before the requests and whether the request is successful or not.

Before the first request, all workers are available. The first request was successful because the transcribing job is processed by a worker Pod. Figure 3.6 demonstrates the result of the first request.

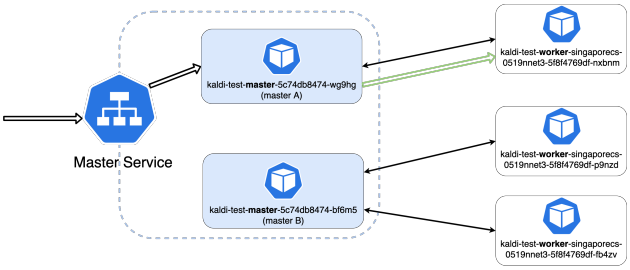

Figure 3.6: The first request is successful

Before the second request, a worker Pod is occupied because it is decoding

the audio sent by the first request. The second request was failed because the master Service directs the traffic to Pod "master A" but it only has 1 attached worker Pod and that worker Pod is occupied. Figure 3.7 demonstrates the failed result of the second request.

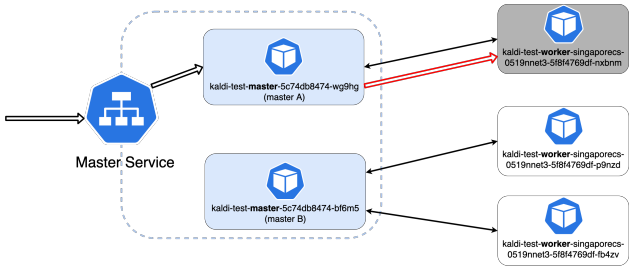

Figure 3.7: The second request is failed

Retry the second request, the request was successful, as the master Service direct the traffic to Pod "master B" which has 2 available workers. Figure 3.8 demonstrates the successful result of the retried second request.

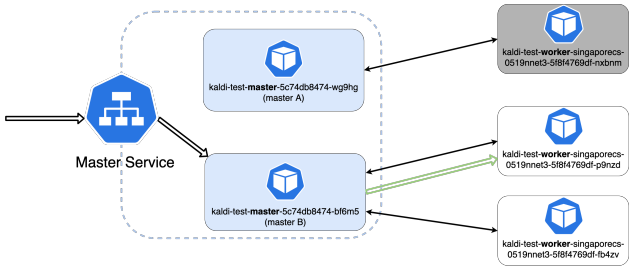

Figure 3.8: The second request retrial is successful

Before the third request, worker Pods are occupied because they are processing the previous two requests. The third request was failed because the master Service directs the traffic to Pod "master A" but it only has 1 attached worker Pod and that worker Pod is occupied. Figure 3.9 demonstrates the failed result of the third request.

Retry the third request, the request was successful, because the master Ser-

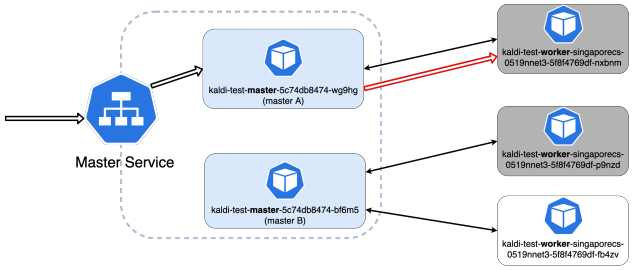

Figure 3.9: The third request is failed

vice direct the traffic to Pod "master B" which have 1 available worker and 1 occupied worker. Figure 3.10 demonstrates the successful result of the retried third request.

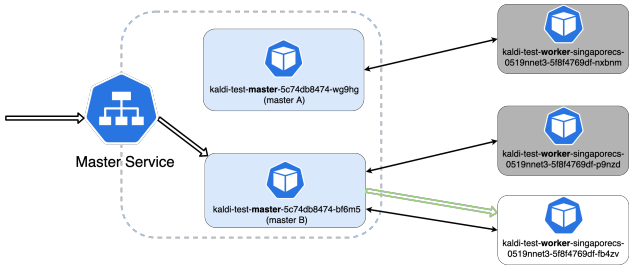

Figure 3.10: The third request retrial is successful

The above experiment demonstrates the missing of an appropriate load balancing mechanism will lead to low availability and high request rejection rate of the system. Setting up an appropriate load balancing mechanism inside the master Service is important to improve the availability of the system.

3.2.2 Ingress

Ingress is implemented here to expose the Service and resolve the load balancing issue. Ingress manages the Service in a way that the Ingress does not direct traffic to Service, instead, with the help of the Ingress Controller,

Ingress direct traffic to Endpoints of the Service with a specific load balancing mechanism with a configurable load balancing algorithm.

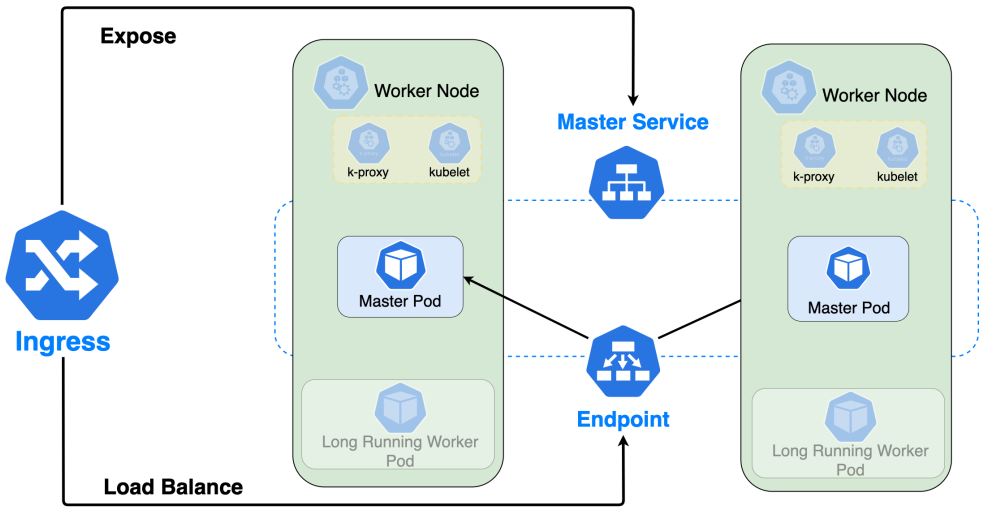

Figure 3.11: Ingress expose Service with load balancing

3.3 Auto-scaling

To ensure high availability of the system, auto-scaling is an important part to be taken into consideration.

The below sections introduce auto-scaling of two different types of worker Pods: static long-running worker Pods and temporary worker Pods. Note that a static long-running refers to a normal worker Pod, which means it is always running unless crash, and able to receive requests when it is idling. Temporary worker Pods are spawn by Jobs in the Kubernetes cluster.

3.3.1 Auto-scaling of Static Long-running Workers

Requests sent to the deployed ASR system may vary in different periods. For some of the use cases, the workload varies drastically during different times of the day. For example, if the system is used in a call centre, there tend to be more requests happens during the working hours, i.e. from 8 am to 6 pm, and there tend to be almost no requests happens during the off-peak hours, i.e. from 11 pm to 5 am. If a fixed number of worker instances are deployed in the system, this situation cannot be handled properly. For example, if the number of deployed workers matches the daily average workload, during the peak hours, the request rejection rate will be high because not enough workers are available and during the off-peak hours, there will be unutilised resources.

It is necessary to scale the system dynamically to support this use case. An appropriate solution is to estimate the number of requests sent at different period of the day according to the historical data and scale the number of static long-running worker Pods beforehand. Table 3.1 shows the number of static long-running worker Pods deployed by a customer for their ASR system. The number of static long-running worker Pods deployed at a different period time of a day reflects the number of requests the customer needs to handle at a different period time of a day.

In this system, a scheduled script is running every hour inside the cluster to manually scale the number of worker Pods to the expected size.

Number of static long-running worker Pods in the system deployed by a customer			
Time Period	Number of Worker Pods	Time Period	Number of Worker Pods
1am - 2am	2	1pm - 2pm	140
2am - 3am	1	2pm - 3pm	140
3am - 4am	1	3pm - 4pm	140
4am - 5am	1	4pm - 5pm	140
5am - 6am	1	5pm - 6pm	140
6am - 7am	2	6pm - 7pm	110
7am - 8am	2	7pm - 8pm	2
8am - 9am	55	8pm - 9pm	2
9am - 10am	130	9pm - 10pm	2
10am - 11am	140	10pm - 11pm	2
11am - 12pm	140	11pm - 12am	2
12pm - 1pm	140	12am - 1am	2

Table 3.1: Number of worker Pods in the system for a customer

3.3.2 Auto-scaling of Temporary Worker Pods with Jobs

Sometimes when an unexpected public emergency issue happens, the number of requests to a related agent's call centre will be drastically higher than normal. If an ASR system is deployed to serve this call centre, it is required to maintain high availability facing the unexpectedly large amount of requests.

To deal with it, the system design includes the Kubernetes Job resource. A Job can spawn temporary Pods in a cluster. In this system, it is configured to spawn temporary ASR worker Pods. A temporary worker Job is different from a static long-running worker Pod because it terminates after finishing a decoding process.

Figure 3.12 illustrates the process of spawning Job. When a master Pod found there is a need for new workers, it will communicate with the cluster control plane via the Kubernetes API server to create a Job resource. After

a Job resource is created, it will spawn a temporary worker Pod accordingly. The temporary worker Pod may be scheduled to a worker node where there is a static long-running worker Pod running. When there is not enough worker node to schedule the temporary worker Pod, the auto-scaling group will spin up a new worker node to host the Pods.

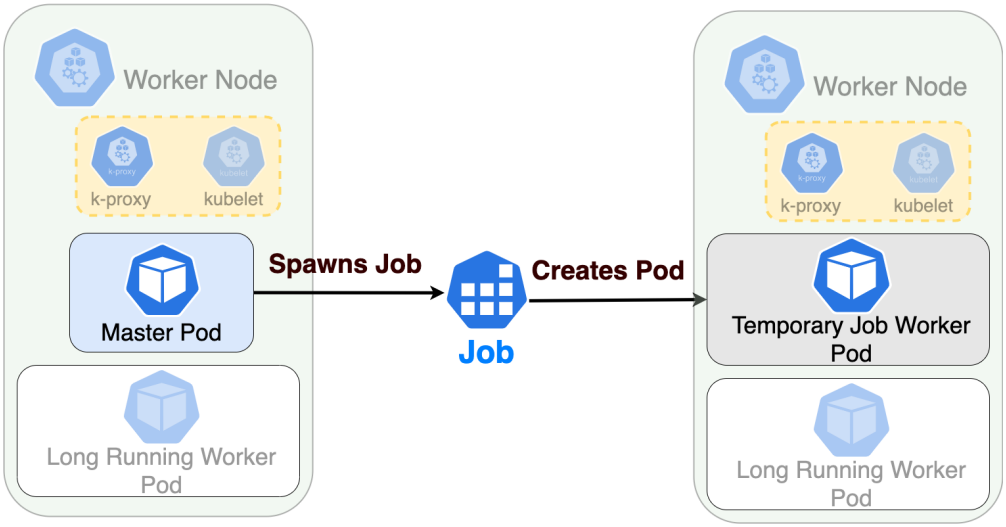

Figure 3.12: Master Pod spawn Job

Chapter 4

Implementation

This chapter will elaborate on the implementation of the designed solution. The solution is implemented on Amazon Web Service (AWS) using Elastic Kubernetes Service (EKS) and other supporting services. Figure 4.1 shows the architecture diagram of the implemented solution on AWS.

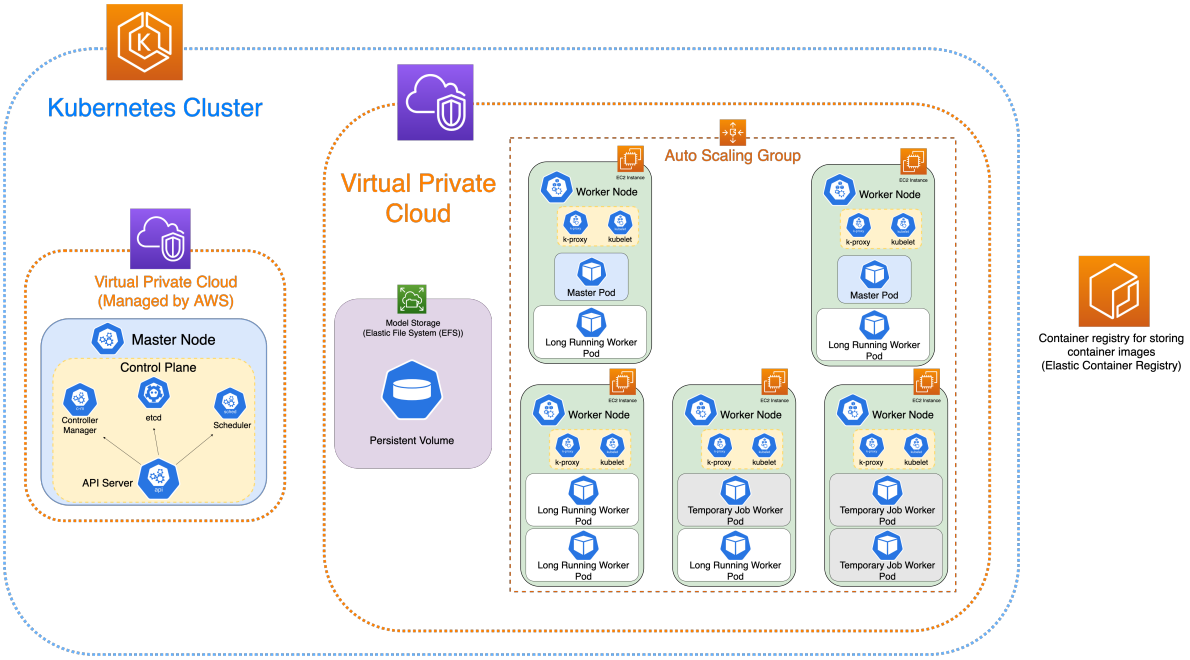

Figure 4.1: Structural architecture diagram of the implemented solution

4.1 Infrastructure Set-up

The infrastructure setup process follows the instructions from a document on the AWS platform: Getting started with Amazon EKS with the AWS Management Console and AWS CLI [17]. The following sub-sections demonstrate important steps to set up the cluster infrastructure.

4.1.1 Networking Set-up

When hosting a Kubernetes cluster on a cloud platform, a good practice is to create a Virtual Private Cloud (VPC). A VPC is considered as a virtual network [22], all the resources inside the VPC can communicate with their private IP address and resources in a VPC need an internet gateway to communicate resources outside VPC. This provides a layer of security to the Kubernetes Cluster.

There are 3 types of VPCs, VPCs with public subnets only, VPCs with private subnets only and VPCs with a mix of public subnets and private subnets. In this project, a VPC with a mix of public subnets and private subnets is chosen because Kubernetes computational resources will be created in private subnets to ensure security while public subnets are also needed for outside clients to access the deployed application.

When setting up the network, manually setup is not required as the AWS platform has already prepared a network stack catering to most Kubernetes

clusters [17]. Follow the instructions and use the AWS CloudFormation [23] to create the required network stack.

4.1.2 Cluster Creation

The next step is to create a cluster using Elastic Kubernetes Service (EKS), note that the cluster should be created inside the VPC created just now. After around 15 minutes, the cluster creation process is finished. The creation of the cluster spin up the cluster master node where the cluster control plane is running. The control plane runs the Kubernetes API server, where you can use a client to communicate with it to manage the cluster.

The kubectl is the official Kubernetes command-line tool [4]. Download it on the local computer and set up the configuration to specify the Kubernetes server you are connecting to is the cluster created just now. Make sure the version of the kubectl and version of the cluster have at most 1 minor difference [4]. For example. if you created a cluster with version 1.18, your kubectl command-line interface (CLI) should be no older than 1.17 and no later than 1.19.

At this step, a cluster is created with only the cluster control plane running. Applications are unable to be deployed on the cluster because there are no available computational resources in the cluster.

4.1.3 Node Group Set-up

The next step is to add computational resources, i.e. node, into the cluster. Elastic Compute Cloud (EC2) service is often used to create computational resources on AWS. EC2 instances are considered virtual machines, which will act as computational resources in the cluster [20].

The computational resources are mainly used to run Pods. Since each Pod must be run on a single node, and the scheduler will only schedule a Pod to a node that has a sufficient amount of vCPUs and memory to run the Pod, it is crucial to choose node type to have sufficient vCPUs and memory.

The ASR system to be deployed contains two kinds of Pods, master Pods and worker Pods. Table 4.1 shows the resource usage of each master Pod and worker Pod.

Pod Type	vCPU Required	Memory Required	vCPU Limit	Memory Limit
Master Pod	1	1GB	1.5	2GB
Worker Pod	0.6	5GB	0.7	6GB

Table 4.1: Master Pods and Worker Pods Resource Usage

Consider the resource used by the Pods and set up EC2 instances with an appropriate amount of vCPU and memory. In the implementation, t3.large EC2 instance type is chosen. Each t3.large instances has 2vCPUs and 8GB memory. Create an auto-scaling group, with node group size set to 1 to 5 for testing purpose.

Auto-scaling of the Node Group

Since the Pod deployment in a Kubernetes cluster need to be scaled up and down to tackle dynamically changing request loads, the computational resources in the cluster are required to be able to scale automatically. AWS Cluster Autoscaler enables the node group to have the property of scaling up and down automatically according to the demand of the computational resources [16]. Deploy an AWS Cluster Autoscaler inside the auto-scaling node group created in the last step to make the node group auto-scalable.

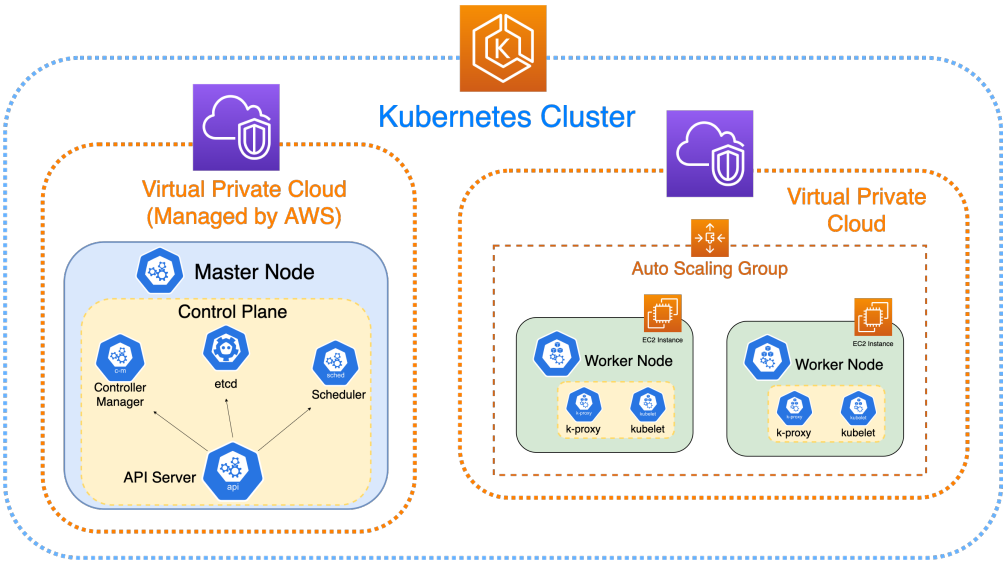

Figure 4.2: Infrastructure Setup

The above are all steps required to set up the infrastructure. Figure 4.2 shows the infrastructure of the system.

4.1.4 Utilise eksctl to Simplify Deployment Process

It is complicated and error-prone to deploy the system following the steps on the AWS management console because every low-level resource needs to be created manually. ekstcl is a simple CLI tool for creating clusters on AWS Elastic Kubernetes Service. It utilises a declarative yaml file to create the cluster just like creating Kubernetes resources. The solution utilises eksctl to deploy the system and cluster creating code snippet is attached in the appendix.

4.2 Container Registry

Before deploying the containerised application, a container registry is needed to host container images. EKS provides the Elastic Container Registry (ECR) where container images are hosted. Create a private repository to host the ASR application container. Use docker to build the container, and push it to the ECR repository created.

Figure 4.3 shows the container registry of the system where Pods pull container image from the images registry. Note that no Pod has been deployed in the system yet, Pods in the figure are just for demonstration.

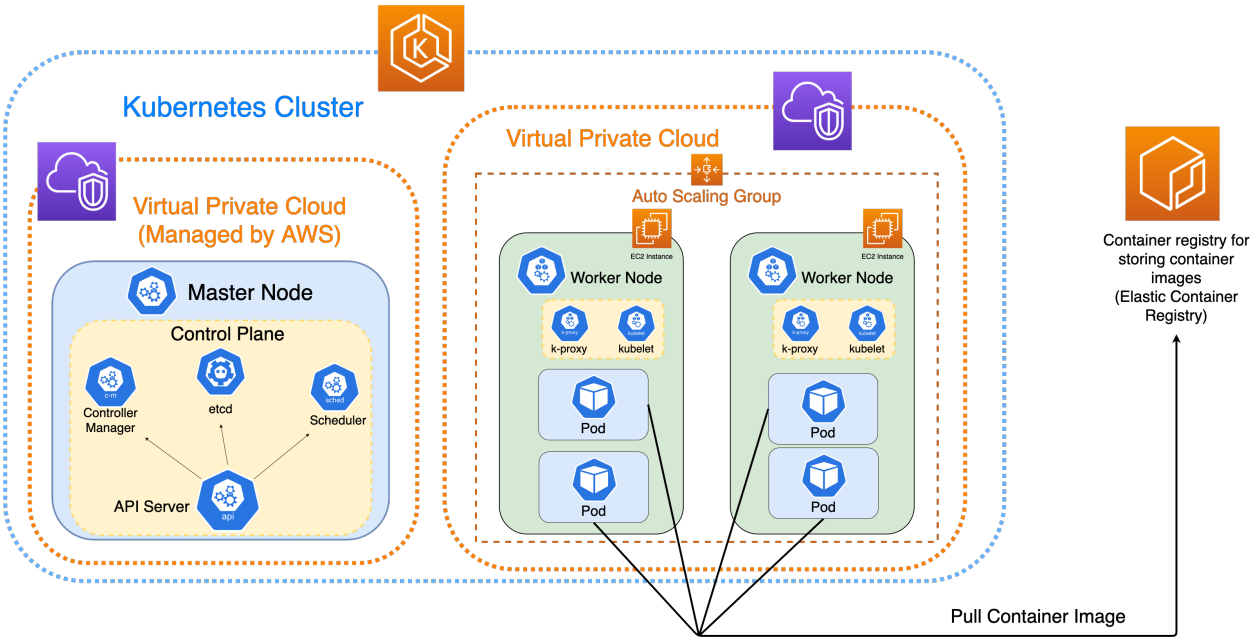

Figure 4.3: Container Registry Setup

4.3 Set up Resources and Deploy the Application

After setting up the infrastructure and the container registry, the next steps are to set up application-related Kubernetes resources in the cluster and deploy the application.

4.3.1 Persistent Volume

When using a cloud platform, a specific storage service is needed to implement the Persistent Volume. network file system (NFS) is a common storage technology used to implement Persistent Volumes in Kubernetes.

The NFS services on the AWS platform is the Elastic File System (EFS). EFS can create a network file system that can be mounted to multiple nodes within

a network. Furthermore, instead of restricting a specific size, EFS can scale in demand [21]. This is a very ideal property to store the languages models for an ASR system since the number of languages models is dynamically changing.

A Persistent Volume requires a path to mount to the worker node. In this system, the Persistent Volume is mounted into the path:

"/home/appuser/opt/models".

Therefore, every worker node can access the models from that path. When a Pod is scheduled to a worker node, it can fetch the required language model from that path.

For a Pod to use a Persistent Volume, a Persistent Volume Claim resource is needed. Persistent Volume Claims abstract away the Persistent Volume from the Pod so that a Pod would not aware of which storage technology is used for the Persistent Volume. Figure 4.4 shows the relationship of the Persistent Volume mounting:

After the Persistent Volume is set up, deploy the application using Helm script.

4.3.2 Deploy application

Using Helm Chart to deploy, several related code snippet can be found in the appendix.

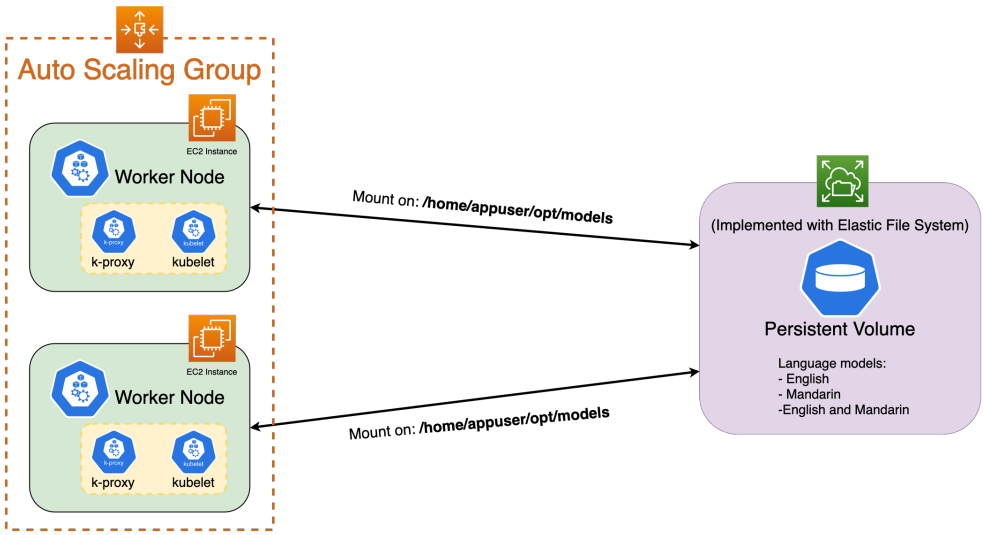

Figure 4.4: Persistent Volume mount to worker nodes

After the Helm Chart is deployed, the system is running in the cluster. A Service of ClusterIP type is created for the master Pods. However, the system so far still could not receive requests from the Internet because the cluster is running inside a virtual private cloud (VPC) and cannot access from outside the VPC. The next step is to create an Ingress resource to expose the Service to the external internet and configure the load balancing.

4.3.3 Ingress with Load Balancer Configuration

In a Kubernetes cluster, every Ingress resources have to run with an Ingress Controller [3]. Deploy the Ingress Controller first. Use the Nginx Ingress Controller, developed by the Nginx community [11]. The Nginx Ingress Controller has options to configure the load balancing algorithm. Change the ConfigMap of the Nginx Ingress Controller and update the load balancing algorithm to "round_robin".

The following code snippet shows the ConfigMap of the Ingress Controller:

```
kind: ConfigMap
apiVersion: v1
metadata:
    name: nginx-config
    namespace: nginx-ingress
data:
    proxy-protocol: "True"
    real-ip-header: "proxy_protocol"
    set-real-ip-from: "0.0.0.0/0"
    lb-method: "round_robin"
```

After the Ingress Controller is deployed in the cluster, create an Ingress resource. The following shows the code snippet of the Ingress resource:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: nginx-ingress
  namespace: ntu-sgdecoding-online-scaled
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
```

```
- host: test-asr.speechlab.sg
  http:
    paths:
    - backend:
        serviceName: kaldi-test-master
        servicePort: 80
- host: test-asr-monitor.speechlab.sg
  http:
    paths:
    - backend:
        serviceName: grafana
        servicePort: 80
```

After setting up the Ingress resource, the ASR application can be accessed from "test-asr.speechlab.sg" and the monitoring dashboard of the system can be accessed from "test-asr-monitor.speechlab.sg".

4.3.4 Auto-scaling of Static Long-running Workers

As the number of expected required worker Pods is known for each period after analysing the daily usage data, a scheduled task to do the scaling is a feasible implementation.

Use cronjob and Kubernetes API to implement the scheduled scaling of the static long-running worker Pods.

To make sure the worker Pods start successfully and ready to do transcribing worker, a buffer time of 30 minutes is added. For example, to scale 140 worker Pods for incoming request starts from 8 am, the cronjob is be scheduled at 7:30 am. In this case, there is sufficient time for Pods and additional required computations resources (node) to be spin up.

4.3.5 Auto-scaling of Temporary Workers

Since the workers are attached to masters in the ASR application. In our deployment solution, a group of worker Pods (one or more) are attached to a master Pod. A master Pod keep track of the number of worker Pods that are attaching to them and what is the language model each of the worker Pod is using. A master Pod at the same time keeps track of which worker Pods are doing the transcribing process and which worker Pods are idling and available to receive a new transcribing request. Therefore, each master Pod is aware of the number of available worker Pod for each language.

Implement a solution that, when a master Pod receive a transcribing request from a client, if it finds that there is less than a certain number of worker Pods available for the requesting language, it will spawn Jobs to create additional worker Pods. The particular threshold number can be configured to match different use cases.

Therefore, in the system, ideally, there will always be a certain number of back up worker nodes available to deal with the unexpectedly large amount

of requests.

Check the appendix to see the detailed spawn Job script for the master Pod.

4.4 Implementation with AWS Fargate Infrastructure

So far in the cluster, the implemented computational resources are Elastic Compute Cloud (EC2) instances, which are virtual machines. Each of the EC2 instances has a fixed amount of available number of vCPUs [1], and a fixed amount of memory once it has started.

Since each Pod must be run on a single node, if the cluster scheduler wants to schedule a Pod to a node, the node must have sufficient vCPU and memory. Meanwhile, after the Pods is running on the node, and there is an extra amount of vCPU and memory left in the node, those vCPU and memory will not be utilized and will be wasted. For example, a master Pod requires 2GB of memory and a worker Pod requires 5GB of memory. To host one master Pod and worker Pod, an instance that with more than 7GB memory need to be chosen. One of the available EC2 instances types is t3.large [12], which has 8GB memory (there is no EC2 instances type that has 7GB memory). However, this choice of instances type will lead to 1GB of unutilised memory, which is a significant amount.

Therefore, it is necessary to choose an appropriate amount of vCPU and memory for the node to ensure higher resource utilization, that is, to choose

[1] Virtual CPUs, used to refer to CPUs inside a virtual machine

an appropriate EC2 instances type to gain higher utilisation of computational resources. However, the process of choosing the optimized EC2 instances type is tedious. Therefore, the inflexibility of EC2 instances may lead to low utilisation of the computational resource.

Fargate is a serverless compute engine for containers and can be used along with Elastic Kubernetes Service (EKS) [15]. Fargate Node is a kind of managed node groups when choosing the worker nodes for the Kubernetes cluster. Other than paying price by the number of nodes you use in your node group, the Fargate Node abstract away the concept of node and directly charge by the number of resources used. For example, to run a master Pod that costs 2GB memory and a worker Pod that costs 5GB memory, will cost resources of 7GB memory, instead of the price of an 8GB virtual machine. Figure 4.5 shows the comparison between the Fargate Node and EC2 instances in terms of cost. Using Fargate Node is a promising method to reduce the cost of the project.

Figure 4.5: Fargate Node Comparing with EC2 instances[15]

To use the Fargate Node as the computational resource of a cluster, specify a Fargate Profile inside the yaml file that used to create the cluster:

```
fargateProfiles:
  - name: fp-ntu-sgdecoding-online-scaled
    selectors:
      # All workloads in the namespace
      # "ntu-sgdecoding-online-scaled"
      # will be scheduled onto Fargate Nodes
      - namespace: ntu-sgdecoding-online-scaled
```

4.5 Ensuring Cluster Security

Security is one of the top concerns for a commercial application. This section will discuss ensuring the security of the application.

When setting up the network of the cluster, a virtual private cloud is created and corresponding subnets are also created. In this project, both public subnets and private subnets are created. The public subnets are only used to run the internet gateways to accept application requests while all the computational resources like EC2 instances and Fargate Nodes are created in private subnets. Resources in a private subnet are not accessible from the Internet but can only be accessed within the private subnet. This mechanism provides a robust solution to computational resources security.

Furthermore, application security is also an important concern since the application should not be access by any unauthorised person or application from the Internet. To deal with this issue, the system can be improved to

be a fully private cluster in which no inbound Internet access is allowed from outside the virtual private cloud (VPC). In a fully private cluster, the deployed application and the Kubernetes control plane are only accessible from within the VPC internal network [19]. Therefore the application could only be accessed within the VPC and not from the Internet.

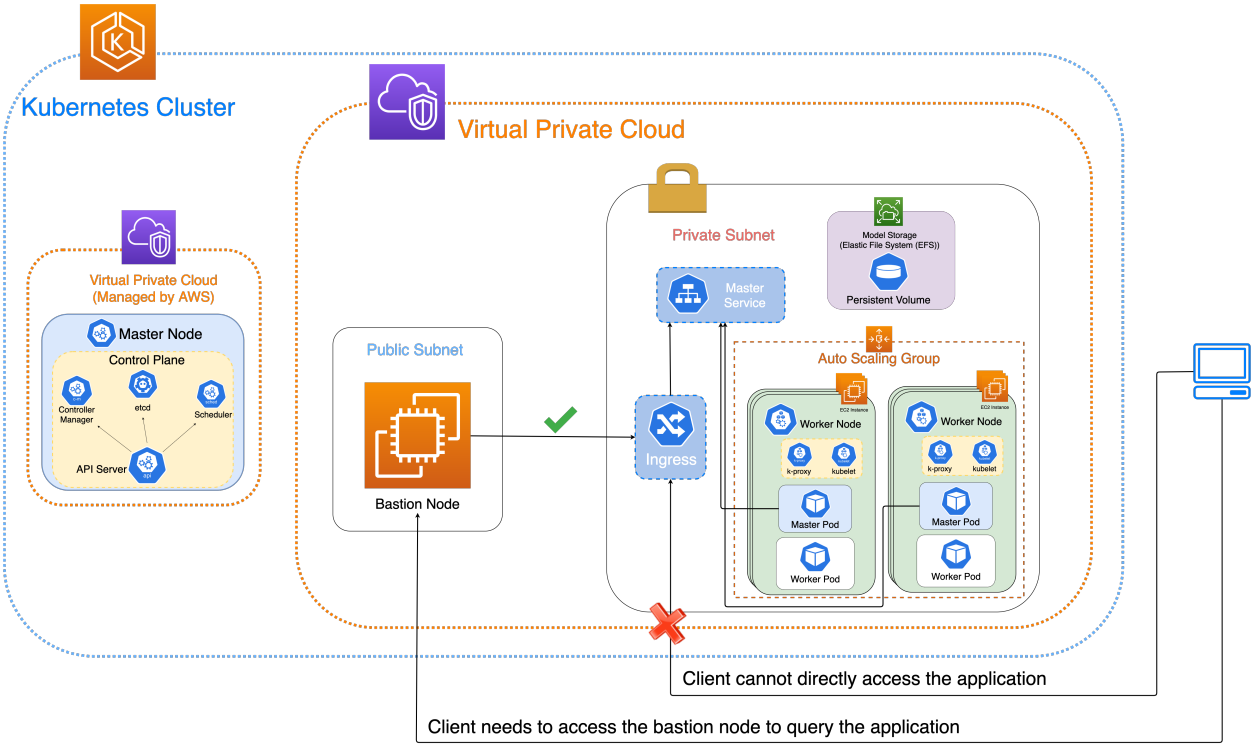

Figure 4.6: Security Mechanism on AWS Implementation

Chapter 5

System Performance and Cost

This chapter will elaborate on the performance of the solution and discuss the estimated cost of the solution. The performance of the solution will be mainly evaluated by the amount of time taken to spin up a Pod.

5.1 Pods Spin-up Time

In the proposed solution, when the system encounters an unexpectedly large number of requests concurrently, new Jobs will be created and new Pods will be spin up to server requests and ensure high availability. Pods spin-up time is an important metrics to evaluate the system performance as it reflects the waiting time for a client request during the peak period. To discuss the time taken to spin-up a Pod, the process of spinning-up Pods needed to be considered. This section will discuss the process of spinning-up Pods in 2 cases, i.e. when using Elastic Compute Cloud (EC2) instances as the

underlying infrastructure and when using Fargate Node as the underlying infrastructure.

5.1.1 Elastic Compute Cloud Infrastructure

This section elaborates on the detailed process of spinning-up a Pod with EC2 as the underlying infrastructure. The following flow charts show the process of spinning up a new Pod.

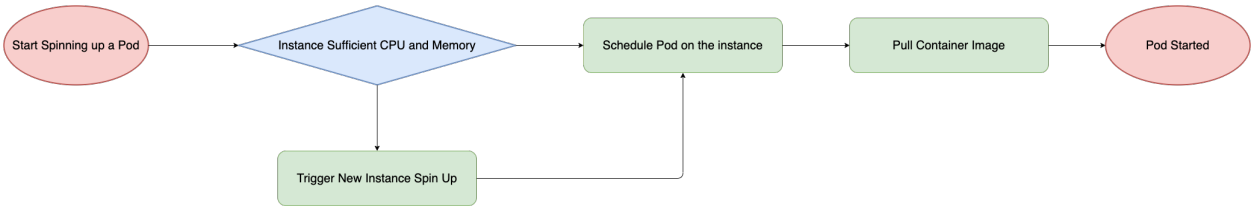

Figure 5.1: Pod creation process on EC2 instances

When the solution is implemented using EC2 instances as the computational resource, as shown in Figure 5.1, there are two possible cases.

If the newly created Pod can be scheduled on any of the current nodes, i.e. at least one of the current nodes in "Ready" states have sufficient available vCPU and memory to hold the Pod, the Pod will be scheduled to that node and no new node will be created. In this case, the spin-up time is relatively short, with an average of **1 minute**.

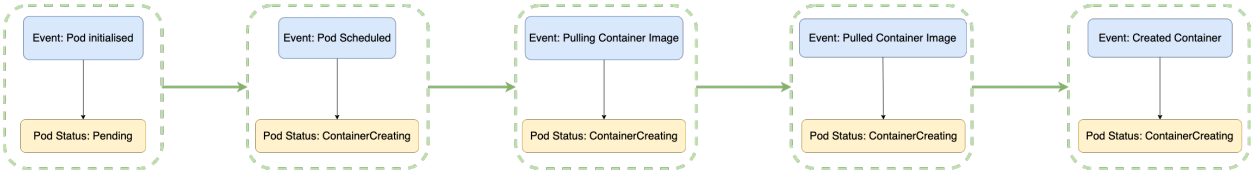

Figure 5.2: Pod creation process on EC2 instances when not triggering node group scale-up

When no available worker node to schedule the Pod, i.e. none of the worker nodes has enough CPU and memory to run the Pod, cluster auto-scaler will trigger a new node to be spin-up. After the new node is ready, the cluster scheduler will schedule the Pod onto the newly created node. The process is shown in figure 5.3. In this case, the spin-up time is relatively long, with an average of **3 minutes**.

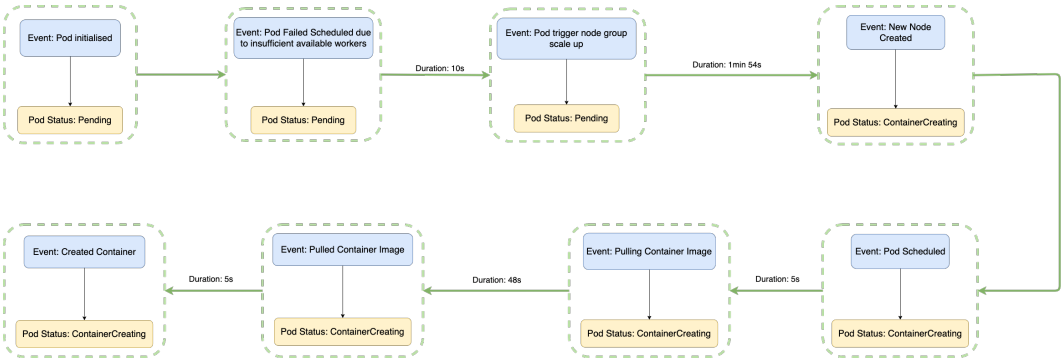

Figure 5.3: Pod creation process on EC2 instances when triggering node group scale-up

From the above analysis, Pod spin-up time when using EC2 instances will be around **2 minutes** on average.

5.1.2 Fargate Node Infrastructure

Each Fargate Node will only host one Pod, therefore every time a new Pod is created, a new Fargate Node will be created first and after the Fargate Node's status become Ready, the newly created Pod will be scheduled onto the Fargate Node and start running. The process is shown in figure 5.4. In this case, the spin-up time is longer than using EC2 instances, with an average of around **3 minutes 40 seconds**. Compared to EC2 instances infrastructure, Fargate Node has around **83.3%** longer Pod spin-up time.

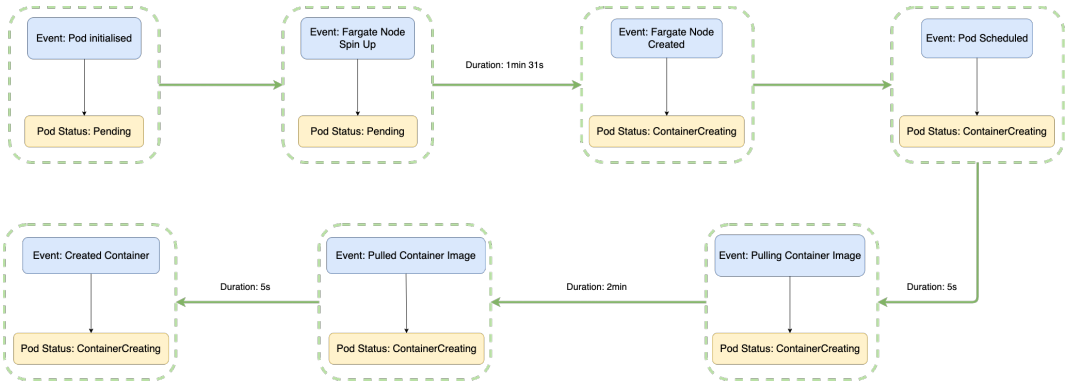

Figure 5.4: Pod creation process on Fargate Nodes

5.2 System Costs

The sections will analyse the cost of the implemented solution. In this project, two methods of deploying Pods are explored, i.e. using Elastic Compute Cloud (EC2) instances as the underlying computational resources and using Fargate Node as underlying computational resources.

5.2.1 Elastic Compute Cloud Infrastructure

When using EC2 instances as the underlying computational resources, an EC2 node group needs to be created within the cluster as computational resources. When the Kubernetes cluster spins up a Pod, the scheduler will pick up an EC2 node to run the Pod on. In this case, the cloud platform charges the price of used EC2 instances.

In this project, when using EC2 as the underlying computational resources, t3.large type is used. Each t3.large worker Node could only run one worker Pod and a master Pod and other supporting Pods. Each t3.large type EC2

instances costs 0.1056 USD per hour [13]. Since every worker Pod need to have one t3.large to run on, every worker Pod costs 0.1056 USD per hour. Therefore, the estimated cost in USD per hour when using EC2 infrastructure is 0.1056 USD $\times$ number of worker Pods.

5.2.2 Fargate Node Infrastructure

Fargate Node solution has higher resource utilization as it provides an exact amount of computational resource required by a Pod. In this project, the implemented solution mainly contains 2 types of Pods, master Pods and worker Pods. When using the Fargate Node to deploy a Master Pod, a Fargate Node of 1vCPU 3GB will be created to provision resources to the Pod. When using the Fargate Node to deploy a Worker Pod, a Fargate Node of 1vCPU 6GB will be created to provision resources to the Pod.

Fargate Node pricing is based on requested vCPU and memory resources and these two dimensions are charged independently. According to Amazon Web Service Fargate Node pricing in Singapore [14]: per vCPU per hour is 0.05056 USD and per GB per hour is 0.00553 USD. From the above information, the cost of each master Pod and each worker Pod can be deduced:

A master Pod costs $0.05056 \times 1 + 0.00553 \times 3 = 0.06719$ USD per hour.

A worker Pod costs $0.05056 \times 1 + 0.00553 \times 6 = 0.08378$ USD per hour.

Besides the Fargate Nodes created for worker Pods and master Pods, there

are also system Pods and monitoring Pods [1], those Pods need some other EC2 instances to host. Consider that those Pods will not consume many resources, spin up two t3.small virtual machines should be sufficient. Each t3.small EC2 instances cost 0.0264 per hour. Therefore the EC2 instances cost for system Pods and monitoring Pods when using Fargate Nodes is $0.0264 \times 2 = 0.0528$ USD.

The total estimated cost when using Fargate Node infrastructure based on the number of worker Pods (suppose there are only two master Pods deployed in the system) is demonstrated in figure 5.5.

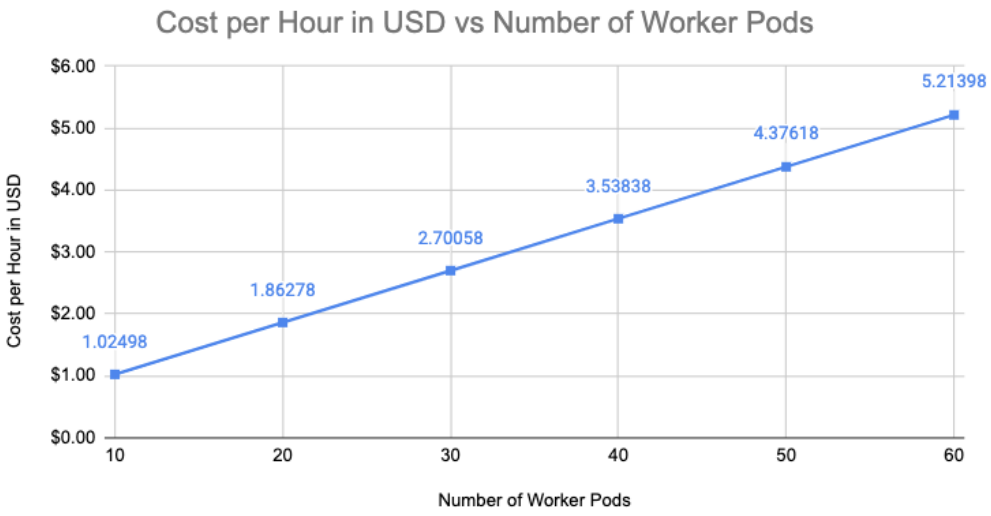

Figure 5.5: Fargate Infrastructure Cost According to Number of Deployed Worker Pods

5.3 Estimated System Cost in a Real Use Case

An ASR application workload profile from a customer can be seen on the table 3.1. Sum up the number of workers required at each period of a day

[1] Prometheus and Grafana are deployed in the system

and get the total number of worker required each day, which is 1437 unit hours, that is, when ASR workers deploy to Pods, each day, there are 1437 Pods hours. The following discusses the cost to deploy ASR workers when using EC2 and Fargate Node.

EC2 cost per day is: $0.1056 \times 1437 = 151.75$ USD Fargate Node cost per day is: $0.08378 \times 1437 + 0.0528 \times 24 = 121.66$ USD

Suppose each month contains 30 days and every day the system will receive roughly the same workloads, the cost for this customer will be:

EC2 per month is: $151.75 \times 30 = 4552.50$ USD Fargate Node per month is: $121.66 \times 30 = 3649.80$ USD

Infrastructure	vCPU Hour Consumed	Memory Hour Consumed	Cost (USD)
EC2 Instances	86220	344880GB	4552.50
Fargate Nodes	44550	258660GB	3649.80

Table 5.1: Monthly Cost Comparison Between EC2 Infrastructure and Fargate Infrastructure

Table 5.1 demonstrates the detailed monthly cost breakdown [2] for deploying ASR workers for this customer. The table shows the monthly cost comparison when deploying using EC2 instances or Fargate Nodes. As shown in the table, deploying the ASR system can be a large amount of cost but if adopt Fargate Node as infrastructure, compare to EC2 solutions, this customer will save 4552.50 - 3649.80 = 902.7 USD each month which will save (4552.50 - 3649.80) / 4552.50 = **19.8%**.

[2]Usages of vCPU hours and memory hours

5.4 Comparison between EC2 and Fargate

This section discusses the trade-off between choosing EC2 instances and Fargate Node.

Using EC2 instances generally has a shorter spin-up time. If EC2 instances type which has a larger amount of vCPU and memory is chosen, each EC2 instances could hold multiple worker Pods and therefore reduce the need to scale worker node when spinning up new worker Pod.

Using Fargate Nodes has a lower cost since this implementation optimise the resource utilisation because only the amount of vCPU and memory required by Pods is provisioned and reduce the resource waste. However, since every time a Pod is spin up, a Fargate Node needed to be created, the Pod spin up time is generally longer than using EC2 instances.

Chapter 6

Conclusion and Future Work

This chapter summarizes the achievement of the project and discusses some of the possible future improvements of the project.

6.1 Summary of Achievements

The objective of this project is to deploy the existing ASR system for high availability and high scalability for commercial usage.

During the project, a Docker and Kubernetes-based solution was designed and implemented on the Amazon Web Service platform. The ASR system is functioning well on the AWS cloud platform and at the same time, availability and scalability are improved.

Traffic load balancing within the system is optimised. The traffic to the cluster would be distributed evenly to the ASR masters, which contributes

to the high availability of the application.

Auto-scaling of the application has been set up. Two auto-scaling methods are achieved: auto-scaling of long static running ASR workers periodically and auto-scaling of temporary ASR workers when facing an unexpectedly large amount of concurrent requests, i.e. if there are not enough application instances or computational resources, the system would be able to spin up new application instances and computational resources automatically.

Methods to optimise the system cost are explored and implemented. Apart from using the Elastic Compute Cloud (EC2) instances as underlying computational resources, the project also explored the Fargate Node as the underlying computational resources. After the experiment, compared to traditional deployment with EC2 instances, using Fargate Node could save a significant amount of money.

Mechanisms to ensure the cluster security and application security are implemented. When deploying the system using Kubernetes, all the computational resources (EC2 instances and Fargate Nodes) are created inside private subnets in the virtual private cloud, which prevents unauthorised user to access the resources. Furthermore, deploying all the Kubernetes resources in private subnets is experimented and it is proved to contribute to the application security.

6.2 Future Improvements

Some improvements are needed to make the system more robust and better performed. This section introduces the possible improvements to the project.

6.2.1 Pre-package Docker Container into Worker Nodes

In the current solution, when a Pod is created, the container image is pulled from a centralised image registry on Amazon Web Service. The image pulling process is network communication. As demonstrated in Chapter 5, the image pulling process takes around 1 to 2 minutes which is a large part of Pod spinning up time.

Therefore, image pulling time is a bottleneck of the Pod spinning up process. It is necessary to further optimise the image pulling process if possible. A possible method to tackle this issue is to move the Docker container image to each worker nodes. If the container image is pre-cached inside a worker node, when a Pod is scheduled to that node, there is no need to pull the container image from a remote container registry via the network.

The support of pre-cache container images depends on cloud platforms. The solution proposed by this project is implemented on AWS. However, currently, the pre-caching image feature still does not have a stable implementation on the AWS cloud [2]. The AWS cloud may provide stable support for the pre-caching image feature.

6.2.2 Fully Private Cluster

Most of the use case of the ASR system will be used internally by a company to process their phone calls. Security is an important concern in such use case since they do not want the system to be abused by the public.

Although during the project, deploying all resources inside private subnets is experimented, the implementation is still not completed. Therefore, the completed solution and implementation for a fully private cluster needed to be done in the future.

6.2.3 Continuous Delivery

Currently, the solution does not have continuous delivery (CD) mechanism, that is, when a new language model is added, the system administrator needs to manually upload the language model to the Persistent Volume and manually update the cluster resources via Kubernetes client CLI, which is sub-optimal.

The solution should add a CD mechanism with the help of CI/CD tools such as GitHub Actions. The CD mechanism should be able to sense the language model update from the code repository and upload the language model automatically into the Kubernetes cluster Persistent Volume and deploy worker Pods with the new language model.

Bibliography

[1] Daniel Povey et al. "The Kaldi Speech Recognition Toolkit". In: (2011).

[2] pilgrim2go et al. *Pre-caching images*. URL: https://github.com/aws/containers-roadmap/issues/358. [Accessed 10 March 2021].

[3] The Kubernetes Authors. *Ingress*. 28 August 2020. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/. [Accessed 20 January 2021].

[4] The Kubernetes Authors. *Install and Set Up kubectl*. 22 October 2020. URL: https://kubernetes.io/docs/tasks/tools/install-kubectl/. [Accessed 19 January 2021].

[5] The Kubernetes Authors. *Kubernetes Service*. 15 January 2021. URL: https://kubernetes.io/docs/concepts/services-networking/service/. [Accessed 19 January 2021].

[6] The Kubernetes Authors. *Persistent Volumes*. 4 February 2021. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/. [Accessed 14 March 2021].

[7] The Kubernetes Authors. *Pods*. 12 January 2021. URL: https://kubernetes.io/docs/concepts/workloads/pods/. [Accessed 19 January 2021].

[8] The Kubernetes Authors. *Volumes*. 29 January 2021. URL: https://kubernetes.io/docs/concepts/storage/volumes/. [Accessed 14 March 2021].

[9] The Kubernetes Authors. *What is Kubernetes?* 22 October 2020. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 19 January 2021].

[10] IBM Cloud Education. *What is containerization*. 15 May 2019. URL: https://www.ibm.com/sg-en/cloud/learn/containerization. [Accessed 18 January 2021].

[11] NGINX Inc. *NGINX Ingress Controller*. 20 January 2021. URL: https://github.com/nginxinc/kubernetes-ingress#nginx-ingress-controller. [Accessed 20 January 2021].

[12] Amazon Web Service Inc. *Amazon EC2 Instance Types*. URL: https://aws.amazon.com/ec2/instance-types/. [Accessed 11 March 2021].

[13] Amazon Web Service Inc. *Amazon EC2 On-Demand Pricing*. URL: https://aws.amazon.com/ec2/pricing/on-demand/. [Accessed 11 March 2021].

[14] Amazon Web Service Inc. *AWS Fargate Pricing*. URL: https://aws.amazon.com/fargate/pricing/. [Accessed 11 March 2021].

[15] Amazon Web Service Inc. *AWS Fargate: Serverless compute for containers*. URL: https://aws.amazon.com/fargate/. [Accessed 20 January 2021].

[16] Amazon Web Service Inc. *Cluster Autoscaler*. URL: https://docs.aws.amazon.com/eks/latest/userguide/cluster-autoscaler.html. [Accessed 19 January 2021].

[17] Amazon Web Service Inc. *Getting started with Amazon EKS – AWS Management Console and AWS CLI*. URL: https://docs.aws.amazon.com/eks/latest/userguide/getting-started-console.html. [Accessed 19 January 2021].

[18] Amazon Web Service Inc. *How Amazon VPC works*. URL: https://docs.aws.amazon.com/vpc/latest/userguide/how-it-works.html. [Accessed 6 February 2021].

[19] Amazon Web Service Inc. *Private clusters*. 26 March 2020. URL: https://docs.aws.amazon.com/eks/latest/userguide/private-clusters.html. [Accessed 18 January 2021].

[20] Amazon Web Service Inc. *What is Amazon EC2?* URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. [Accessed 18 January 2021].

[21] Amazon Web Service Inc. *What Is Amazon Elastic File System?* URL: https://docs.aws.amazon.com/efs/latest/ug/whatisefs.html. [Accessed 19 January 2021].

[22] Amazon Web Service Inc. *What is Amazon VPC?* URL: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html. [Accessed 20 January 2021].

[23] Amazon Web Service Inc. *What is AWS CloudFormation?* URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html. [Accessed 13 March 2021].

[24] Amazon Web Service Inc. *What is cloud computing?* URL: https://aws.amazon.com/what-is-cloud-computing/. [Accessed 13 March 2021].

[25] Amazon Web Service Inc. *What is Docker?* URL: https://aws.amazon.com/docker/. [Accessed 17 January 2021].

[26] Docker Inc. *What is a Container?* URL: https://www.docker.com/resources/what-container. [Accessed 18 January 2021].

[27] Stephen Kuenzli Jeff Nickoloff. "Docker in Action (Second Edition)". In: Manning Publications, 2019. Chap. 1 Welcome to Docker, p. 2.

[28] Marko Lukša. "Kubernetes in Action". In: Manning Publications, 2017. Chap. 1.2 Introducing container technologies, p. 13.

[29] Marko Lukša. "Kubernetes in Action". In: Manning Publications, 2017. Chap. 1.1 Understanding the need for a system like Kubernetes, p. 7.

[30] Marko Lukša. "Kubernetes in Action". In: Manning Publications, 2017. Chap. 1.3.3 Understanding the architecture of a Kubernetes cluster, p. 18.

[31] Marko Lukša. "Kubernetes in Action". In: Manning Publications, 2017. Chap. 5 Services: enabling clients to discover and talk to pods, p. 121.

[32] Katy Stalcup. *AWS vs Azure vs Google Cloud Market Share 2020: What the Latest Data Shows*. 12 November 2020. URL: https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/. [Accessed 21 January 2021].

[33] Guillermo Velez. *Kubernetes vs. Docker: A Primer*. 14 January 2019. URL: https://containerjournal.com/topics/container-ecosystems/kubernetes-vs-docker-a-primer/. [Accessed 2 February 2021].

Appendix A

Code Snippets

This appendix includes important code snippets used to achieve a robust deployment solution.

A.1 Cluster Creation Script

This is cluster.yaml file that is used to create the cluster using eksctl

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: kaldi-test-public-fargate-basic
  region: ap-southeast-1

vpc:
  subnets:
    private:
      apse1-az1:
        id: subnet-01e07c3d32e6af1d0
      apse1-az2:
        id: subnet-00bcf997659e0f197
```

```
    public:
      apse1-az1:
        id: subnet-03626987f42d129d3
      apse1-az2:
        id: subnet-0cefb9de02f61fb8e

nodeGroups:
  - name: ng-ap1a-workers
    labels: { role: workers }
    instanceType: t3.small
    desiredCapacity: 2
    minSize: 1
    maxSize: 2
    volumeSize: 10
    privateNetworking: true
    availabilityZones: ["ap-southeast-1a", "ap-southeast-1b"]
    ssh:
      publicKeyName: EC2AccessKey
    iam:
      withAddonPolicies:
        autoScaler: true
        externalDNS: true
        certManager: true

fargateProfiles:
  - name: fp-ntu-sgdecoding-online-scaled
    selectors:
      - namespace: ntu-sgdecoding-online-scaled
```

A.2 Application Deployment Scripts

These are yaml files to create the application deployment and service.

A.2.1 Deployments

This is deployment.yaml file that is used to create the deployment workload resource in the Kubernetes cluster to deploy application.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "ntuspeechlab.master.name" . }}
  labels:
    app.kubernetes.io/name: {{ include "ntuspeechlab.master.name" . }}
    helm.sh/chart: {{ include "ntuspeechlab.chart" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
    app.kubernetes.io/managed-by: {{ .Release.Service }}
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: {{ include "ntuspeechlab.master.name" . }}
      app.kubernetes.io/instance: {{ .Release.Name }}
  template:
    metadata:
      annotations:
        prometheus.io/scrape: 'true'
        prometheus.io/port: '8081'
      labels:
        app.kubernetes.io/name:
        {{ include "ntuspeechlab.master.name" . }}
        app.kubernetes.io/instance: {{ .Release.Name }}
    spec:
      containers:
        - name:
        {{ include "ntuspeechlab.master.name" . }}
          image:
          "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          command: [{{ include "ntuspeechlab.master.command" . }}]
          envFrom:
            - secretRef:
                name: environment-variables-master-secret
          ports:
            - name: http
```

```
            containerPort: 8080
            protocol: TCP
          - name: prometheus
            containerPort: 8081
        resources:
          limits:
            cpu: 2
            memory: "2Gi"
          requests:
            cpu: 1
            memory: "2Gi"
        volumeMounts:
          - name: models-efs
            mountPath: /home/appuser/opt/models
        livenessProbe:
          httpGet:
            path: /
            port: 8080
          periodSeconds: 60
        readinessProbe:
          httpGet:
            path: /
            port: 8080
          periodSeconds: 60
      volumes:
        - name: models-efs
          persistentVolumeClaim:
            claimName: models-efs-claim


{{- range $model_name, $replicas := .Values.models }}
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "worker.name" $ }}
  {{ printf "-%s" $model_name | lower | replace "_" "-"  }}
  labels:
    app.kubernetes.io/name:
    {{ include "worker.name" $ }}
    {{ printf "-%s" $model_name | lower | replace "_" "-" }}
    helm.sh/chart: {{ include "ntuspeechlab.chart" $ }}
    app.kubernetes.io/instance: {{ $.Release.Name }}
    app.kubernetes.io/managed-by: {{ $.Release.Service }}
spec:
```

```
replicas: {{ $replicas }}
selector:
  matchLabels:
    app.kubernetes.io/name: {{ include "ntuspeechlab.worker.name" $ }}
    {{ printf "-%s" $model_name  | lower | replace "_" "-"}}
    app.kubernetes.io/instance: {{ $.Release.Name }}
template:
  metadata:
    annotations:
      prometheus.io/scrape: 'true'
      prometheus.io/port: '8081'
    labels:
      app.kubernetes.io/name: {{ include "ntuspeechlab.worker.name" $ }}
      {{ printf "-%s" $model_name | lower | replace "_" "-" }}
      app.kubernetes.io/instance: {{ $.Release.Name }}
  spec:
    containers:
      - name: {{ include "ntuspeechlab.worker.name" $ }}
      {{ printf "-%s" $model_name | lower  | replace "_" "-"}}
        image: "{{ $.Values.image.repository }}:{{ $.Values.image.tag }}"
        imagePullPolicy: {{ $.Values.image.pullPolicy }}
        command: [{{ include "ntuspeechlab.worker.command" $ }}]
        envFrom:
          - secretRef:
              name: environment-variables-workers-secret
        ports:
          - name: prometheus
            containerPort: 8081
        resources:
          limits:
            cpu: 0.7
            memory: "6Gi"
          requests:
            cpu: 0.6
            memory: "5Gi"
        env:
        - name: MODEL_DIR
          value: {{ $model_name }}
        volumeMounts:
        - name: models-efs
          mountPath: /home/appuser/opt/models
    volumes:
      - name: models-efs
        persistentVolumeClaim:
          claimName: models-efs-claim
```

```
{{- end }}
```

A.2.2 Services

This is service.yaml file that is used to create the Service resource in the Kubernetes cluster to expose the application.

```
apiVersion: v1
kind: Service
metadata:
  name: {{ include "ntuspeechlab.master.name" . }}
  labels:
    app.kubernetes.io/name: {{ include "ntuspeechlab.master.name" . }}
    helm.sh/chart: {{ include "ntuspeechlab.chart" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
    app.kubernetes.io/managed-by: {{ .Release.Service }}
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/port: '8081'
spec:
  type: {{ .Values.service.type }}
  ports:
    - protocol: TCP
    {{- if (eq .Values.service.type "ClusterIP") }}
      nodePort: null
    {{- end }}
      port: 80
      targetPort: 8080
      name: http
  selector:
    app.kubernetes.io/name: {{ include "ntuspeechlab.master.name" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
```

A.3 Auto-scaling of Temporary Worker Pods

```
def create_job(MODEL):
    api = client.BatchV1Api()
```

```
body = client.V1Job(
    api_version="batch/v1",
    kind="Job"
)
name = 'speechlab-worker-job-{}-{}'.format(
    MODEL.lower().replace("_", "-"), id_generator()
)
body.metadata = client.V1ObjectMeta(
    namespace=NAMESPACE,
    name=name
)
body.status = client.V1JobStatus()
template = client.V1PodTemplate()
template.template = client.V1PodTemplateSpec()
template.template.metadata = client.V1ObjectMeta(annotations={
        "prometheus.io/scrape": "true",
        "prometheus.io/port": "8081"
})
efs_volume_claim = client.V1PersistentVolumeClaimVolumeSource(
    claim_name='models-efs-claim'
)
volume = client.V1Volume(
    name='models-efs',
    persistent_volume_claim=efs_volume_claim
)
env_vars = {
    "MASTER": MASTER,
    "NAMESPACE": NAMESPACE,
    "RUN_FREQ": "ONCE",
    "MODEL_DIR": MODEL,
}

env_list = []
if env_vars:
    for env_name, env_value in env_vars.items():
        env_list.append(client.V1EnvVar(
            name=env_name,
            value=env_value
        ))

container = client.V1Container(
    name='{}-c'.format(name),
    image=IMAGE,
    image_pull_policy="IfNotPresent",
    command=[
```

```
            "/home/appuser/opt/tini",
            "--",
            "/home/appuser/opt/start_worker.sh"
        ],
        env=env_list,
        ports=[client.V1ContainerPort(
            container_port=8081,
            name="prometheus"
        )],
        resources=client.V1ResourceRequirements(
            limits={"memory": "6G", "cpu": "0.7"},
            requests={"memory": "5G", "cpu": "0.6"}
        ),
        volume_mounts=[client.V1VolumeMount(
            mount_path="/home/appuser/opt/models",
            name="models-efs",
            read_only=True
        )]
    )

    template.template.spec = client.V1PodSpec(
        containers=[container],
        restart_policy="OnFailure",
        volumes=[volume]
    )
    body.spec = client.V1JobSpec(
        ttl_seconds_after_finished=100,
        template=template.template
    )

    try:
        logging.info('trying to create job')
        api_response = api.create_namespaced_job(NAMESPACE, body)
        return True
    except ApiException as e:
        logging.exception('error spawning new job: ' + str(e))
```