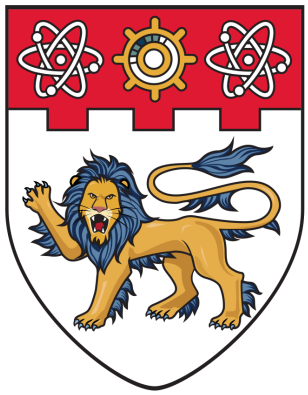

NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Final Year Project

SCSE21-0066

DEPLOYING ASR SYSTEM FOR SCALABILITY AND ROBUSTNESS ON AWS

Final Report

by

Lee Kai Shern

U1820793J

Supervisor: Associate Professor Chng Eng Siong

March 2022

Submitted in Partial Fulfilment of the Requirements for the Degree of Bachelor of

Computer Engineering of the Nanyang Technological University

Abstract

The project aims to provide a robust solution for deploying automatic speech recognition (ASR) system on Cloud. The solutions will enable the system to be provisioned at a lower cost and also simplify the process of deploying systems on Amazon Web Service(AWS).

The solutions are implemented based on the concept of Infrastructure as Code (IaC). This enables the process of building and destroying speech recognition system to be completed in minimum steps which come in convenient at the development stage.

This report will introduce the solutions in terms of the architecture diagram, comparison over different services, frameworks, and tools. The report will demonstrate the provision of the infrastructure of ASR on AWS using Terraform.

Acknowledgements

I have committed efforts to this project. However, the project would not have been possible without the kind support of many individuals. I would like to extend my sincere thanks to all of them.

Firstly, I would like to express my gratitude to Professor Chng Eng Siong for his guidance and advice on the specifications of the solution. Although he was occupied with his research works and projects, he was always willing to listen to the difficulties that I faced and suggested resources that could help with the problem.

Next, I am highly indebted to Research Engineer Vu Thi Ly for her guidance and constant supervision as well as for providing necessary resources and information regarding the project. I appreciate her patience in guiding me through all the technical problems raised to her.

Last but not the least, my appreciations go to my peers who worked on a similar project, Joshua and Zhi Wei for their willingness to lend me a hand with their ability.

It was my pleasure to have these incredible mentors and peers along the way. I had learnt a lot through this project under their guidance and assistance.

Contents

List of Figures

Chapter 1

Overview

Automatic speech recognition (ASR) is a system developed to take in real-time audio streams or audio files and output the corresponding transcripts. NTU Speechlab had developed models which could transcribe speech with a mix of Mandarin and English [1]. ASR is an essential framework in natural language processing, it is being used across different applications in the industry such as intelligent phone assistants, automatic documentation and speech translation [2].

To ensure that the product can be rolled out to clients or users, the system will need to be deployed on a larger scale platform. The development will be backed by the fast-growing cloud computing capability to compose a system that is scalable, robust and accessible.

1.1 Background

From a high-level perspective, the ASR system is constructed from one master node and a few worker nodes. Each worker node is loaded with

one language model and can only transcribe jobs in the same language as the loaded model. The master node is responsible in identifying the language and assigning the job to worker nodes loaded with the language model. Figure 1.1 illustrates how the master and worker nodes work together.

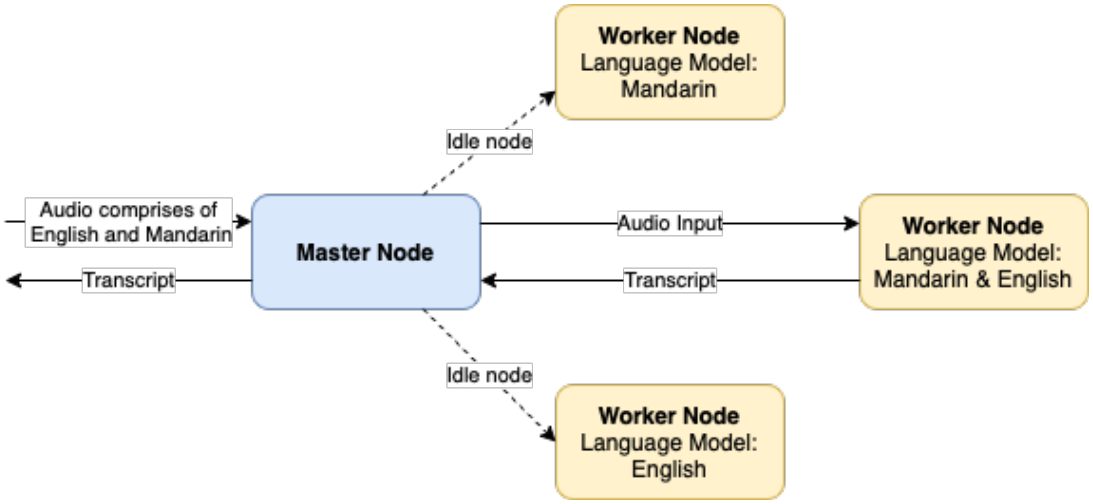

Figure 1.1: ASR Master and worker nodes

ASR system is not used widely in different fields due to some limitations. A big challenge will be the robustness and scalability.

In terms of scalability, if the number of requests sent exceeds the number the worker nodes with the particular language, then incoming jobs will need to wait for free worker nodes to be served. The system needs to be scaled up effectively to handle such situations. In order to scale up effectively, there are two main aspects to be taken into account - time and cost.

Besides, building the infrastructure of the system on the Cloud required a deep understanding of multiples frameworks and services. Infrastructure was provisioned using a combination of scripts and manual pro-

cesses on the AWS console, making the steps to provision the infrastructure disjoint and scattered. This led to the issue of which creation of new environments being repetitive and not always repeatable, reliable and consistent.

1.2 Importance of Study

This project was a continuation of previous projects working on the architecture design of the same system. Students Ma Xiao and Wong Seng Wee had implemented some features under their final year project [3] [4].

Wong had successfully implemented the system on Azure whereas Ma contributed to building the infrastructure on AWS. This project extended the effort of Ma and researched possible improvements for the system specifically on the AWS environment.

Ma's work had achieved scaling in containers using Kubernetes, Fargate and Ingress, but the process of building the infrastructure was complicated. It required a number of changes on the configurations across different files to set up the system on AWS.

ASR System still requires a lot of developments and research to reach the goal of deploying the system to multiple organizations while having a low maintenance cost. However, the issue was that the process of development was costly and slow. It took a long time for researchers or students to spin up and tear down the system in the development stage. This study was important as it simplified the process of provisioning infrastructure and enabled updates to be done efficiently.

The objective of this project was to design a reliable and pragmatic solution to provision and un-provision the infrastructure on the Cloud. While ensuring the flexibility to adjustments, the steps required to build the infrastructure were streamlined and the required inputs from users were minimised.

1.3 Scope

This project focused on the improving robustness and scalability of the ASR system. The project would utilise Amazon Web Service (AWS) as the cloud service provider. The choice of using AWS as the cloud service provider for this project was due to the fact that AWS had a larger community and had introduced more features that fit ASR use cases better. [5]

The solution was implemented using Terraform by Hashicorp which enabled the provision of infrastructure as code, consolidating configurations in source control and bringing transparency and replayability to a previously manual workflow. [6]

Alongside Terraform, the solution also used Helm which would implement software built for Kubernetes. Helm streamlined the installation and management of Kubernetes applications.

Throughout the project, the ASR language models were treated as a black-box operation without any detailed explanation of the functionalities.

1.4 Report Organisation

The report consists of 5 chapters as follow:

Chapter 1: Introduction and overview of the project

Chapter 2: Summary of technologies and terminologies

Chapter 3: Design of the proposed solution

Chapter 4: Detailed implementation of the solution

Chapter 5: Conclusion and suggestions on possible improvements

Chapter 2

Literature Review

The solution proposed in this project was designed based on containerisation technology and was implemented by utilising AWS cloud service. The concept of Infrastructure as Code (IaC) was achieved using Terraform as the tool.

2.1 Containerization

Containerization is a technique that encapsulates software codes and dependencies so that it can run independently and uniformly. Applications are run in isolated user spaces, called containers while using the same shared operating system. [7]

Containerization is a thriving trend in industries when compared to the usage of virtual machines (VM) as it has multiple advantages over VM. Compared to VM, containers are easier to set up, the ease of use of containers accelerates the development and deployment of projects. Besides, containers have significantly lower overhead and generally bet-

ter performance than VM. Another valuable benefit of containerization would be the high portability of containers as each container is abstracted from the host OS and runs the same in any location [8]. Figure 2.1 [9] depicts the architectural differences between containers and virtual machines.

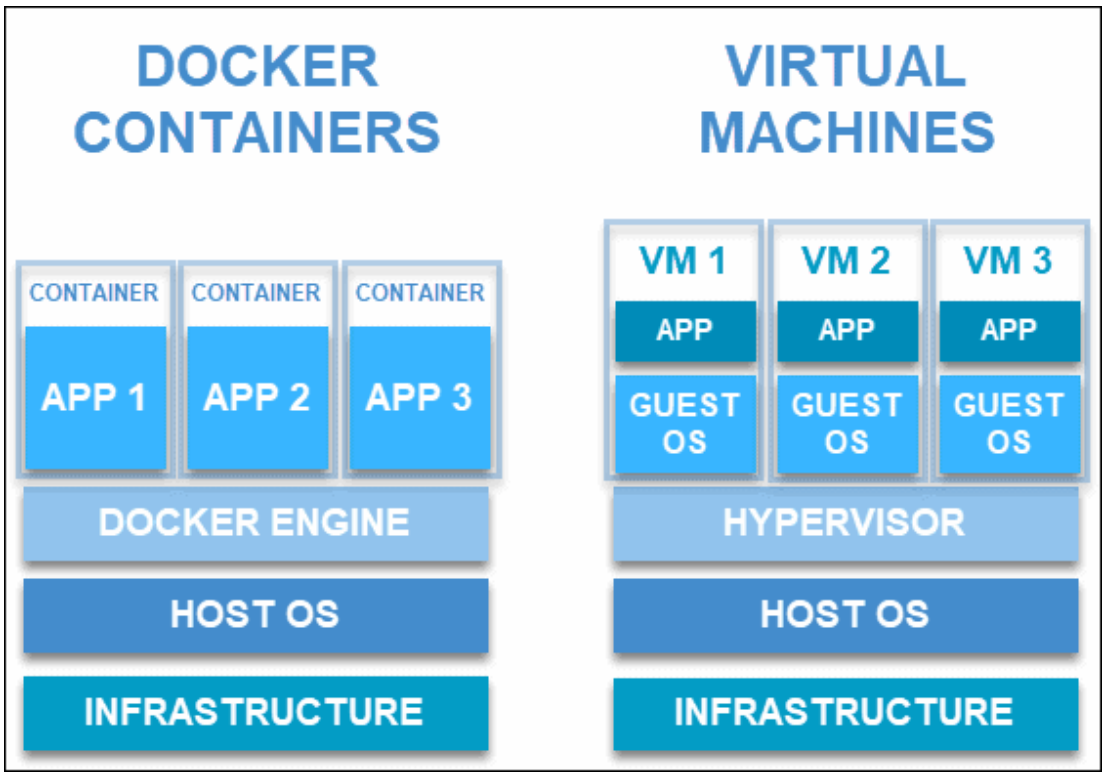

Figure 2.1: Architecture diagrams of Container and Virtual Machines

Containerization makes it straightforward for developers to develop, deploy and operate applications by simplifying the packaging and deployment process.

2.1.1 Container

A container can be interpreted as a fully packaged and portable computing environment. Container packages up all the essential components required to run an application such as binary codes, libraries, configu-

ration files and dependencies. The container itself is abstracted away from the host OS, with only limited access to underlying resources. As a result, the containerized application can be run on various types of infrastructure — on bare metal, within VMs, and in the cloud without needing to refactor it for each environment. [8]

Image, much like a container is an immutable file that contains the fundamental elements needed to run an application. Images consist of a series of layers, each represents a change on an image or an intermediate stage of the image. Every command used to build the image will add another image layer.

A container can be interpreted as a running image.

2.1.2 Docker

Docker is an open platform for developing, shipping, and running applications. Docker provides the ability to package an image and run an application in a container. 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. [10]

2.1.3 Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. Kubernetes excels in container orchestration and maintenance. The list below listed the important features of Kubernetes. [11]

- **Automated rollout and rollback**

Kubernetes progressively rolls out changes to the applications and its configuration. While implementing these changes, Kubernetes will ensure the availability of the system, there will be minimum downtime for the system. If the update fails, Kubernetes will roll-back those changes automatically. This feature is suitable for the growing ecosystem of deployment of solutions.

- **Horizontal scaling**

  Kubernetes Horizontal pod auto-scaler is able to scale the number of pods available in a cluster to cope with the computational work-load requirements of an application. It determines the number of pods needed based on CPU and RAM usage. [12]

- **Self-healing and maintenance**

  Kubernetes will restart the containers that fail, replaces and reschedules containers when nodes fail. Besides, Kubernetes will kill containers that do not meet the health check, ensuring high quality of services at times.

Kubernetes also has a large community of developers with services, support and tools widely available and well-integrated to different cloud service providers. The list below briefly explains the resources of Kubernetes used in this project.

- **Pod**

  A group (one or more) of containers with shared storage and network resources. A Pod is the smallest building unit of deployment in a Kubernetes cluster. Each pod will be assigned to a node with computational resources.

- **Node**

  A virtual or physical worker machine. Kubernetes runs workload by placing the containers into Pods to run on Nodes. Each node is managed by the control plane and contains the services necessary to run Pods.

- **Service**

  An abstract way to expose an application running on Pods as a network service. This guarantees that any changes on Pod is abstracted away and will not affect the functionality of other Pods even when they are working together.

- **Ingress**

  An API object that manages external access to the services in a cluster. Traffic routing is controlled by rules defined on this resource. Ingress can be configured to give Services externally reachable URLs and load balancing.

- **Persistent Volume**

  An API object that captures the details of the implementation of the storage. It abstracts details of how storage is provided from how it is consumed.

2.2 Infrastructure as Code

Infrastructure as Code (IaC) is a fundamental principle of DevOps in which infrastructure is treated as code. With the prevalent cloud-native platforms and consumption of their provisioned infrastructure, the concept of IaC have been popularised. The method to provision and main-

tain critical infrastructure has revolutionized and shifted towards a fully automated era.

Infrastructure building codes have a defined format and syntax depending on the tools used. These tools provide a version control mechanism that logs the history of code development and changes. When updates are being deployed onto the existing system, the tool will be able to detect the changes and only apply them to the relevant frameworks. With IaC, the build is repeatable, reliable and pragmatic. [13]

The common characteristic of the Helm and Terraform is that they only require a few lines of command to build and destroy the infrastructure. A lot of the complexities are handled at the backend requiring no human intervention and allowing users to focus on developing other key features instead.

2.2.1 Terraform

Terraform is an open-source infrastructure as a code software tool that provides a consistent workflow to manage various resources on cloud services. Terraform codifies cloud APIs into declarative configuration files using HashiCorp Configuration Language (HCL). With infrastructure codes, Terraform generates an execution plan for configurations and apply them to Cloud to reach the desired state. [14].

Terraform stores the state of the infrastructure on the local computer. This state file is a custom JSON format that serves as a map for Terraform, describing which resources it manages, and how those resources should be configured. With the state file, Terraform can identify in-

tended updates to infrastructure and apply the applicable changes. [15]

Terraform handles the underlying logic for resource creation. Terraform will plan and build a resource graph to determine resource dependencies and creates or modifies non-dependent resources in parallel, allowing Terraform to provision resources efficiently.

The procedures to demolish the infrastructure will be handled by Terraform and it will ensure that all the resources built are fully destroyed, leaving the environment neat and clean.

However, while Terraform provides support for multiple providers, it is not cloud-agnostic. There is no way to write generic code that can be translated into provider-specific resources. Besides, Terraform would not hide the complexity of underlying Cloud providers. The user would need to understand the services of Cloud providers to use the Terraform modules.

Figure 2.2 [14] shows how Terraform works.

2.2.2 Helm

Helm is an open-source package manager for Kubernetes. It provides the ability to share and use software built for Kubernetes.

Helm uses a packaging format called charts. A chart is a collection of manifest files that describe a related set of Kubernetes resources [16] or it can be viewed as a template to create Kubernetes resources. Alongside the templates, Helm has a configuration file - `values.yaml`. This file contains all the values to be injected into the templates and it can be

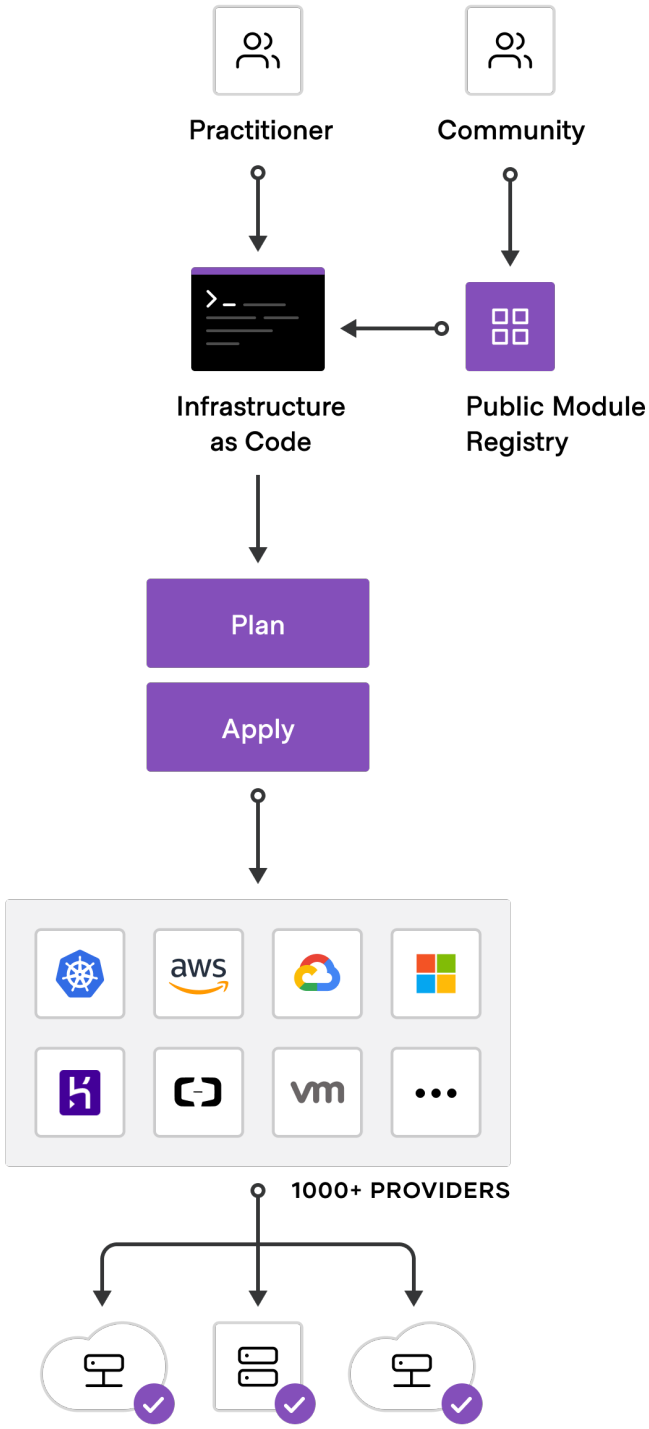

Figure 2.2: How Terraform works

customised by the developers.

The Helm artifactory has a range of charts available for frequently used packages [17], making the process of provisioning these frameworks

clean and easy.

Rather than writing manifest files for each and every Kubernetes resource to configure Kubernetes deployment, developers can use readily available Helm charts to deploy them. Helm chart simplifies the process of application management without compensating on the ability to make customization.

Figure 2.3 [18] shows how Helm works.

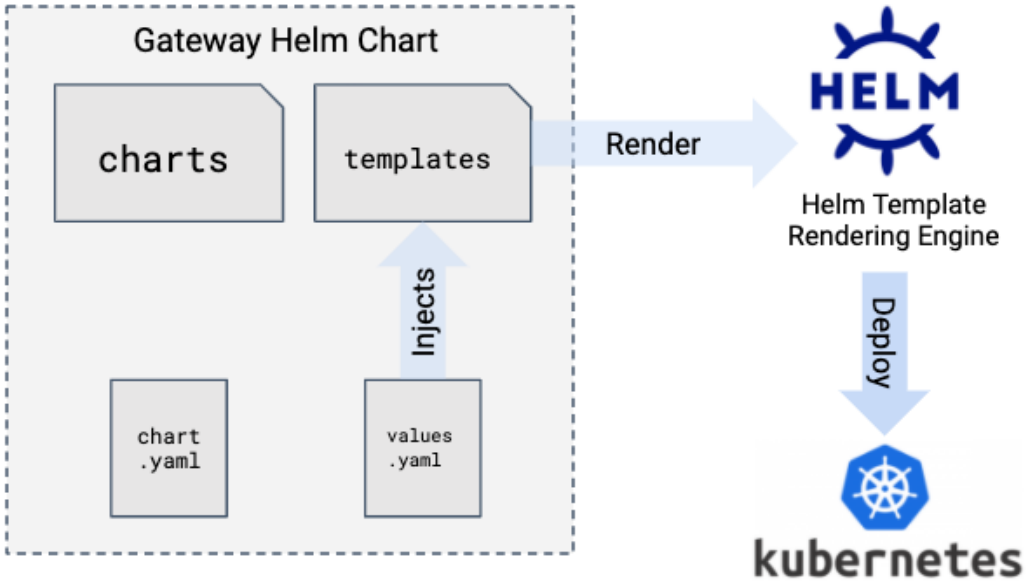

Figure 2.3: How Helm works

2.3 Amazon Web Services (AWS)

Amazon Web Services (AWS) is a cloud service provider which offers scalable solutions for computing, storage, databases and more.

AWS provides IT resources over the internet with pay-as-you-go pricing. Instead of buying and maintaining physical data centres and servers,

developers can access technology services such as computing power and storage on an as-needed basis.

Compared to other cloud service providers, AWS has well-organised documentation and detailed guides for each feature and tool. Besides, AWS continuously introduces new API to enhance the experience of users in implementing solutions on the Cloud. AWS is also backed by a large and strong community and ecosystem compared to others. [5]

2.3.1 Elastic Cloud Computing (EC2)

Elastic Compute Cloud (EC2) service is one of the computational resources on AWS. EC2 instances can be considered as virtual machines, which provide computational power in the cluster [19]. With Amazon EC2, users can set up and configure the operating system and applications that run on the instance.

EC2 are also crafted to have virtual firewalls to control the network traffic in instances. To access the EC2 from the external network, the user must use the key-value pairs which are validated by the instance. This ensures that the security of the instance is paramount. For data or volumes used by EC2, EC2 supports the mounting of the file system, allowing sharing of data across multiple EC2 in the same availability zone.

2.3.2 Virtual Private Cloud (VPC)

Amazon Virtual Private Cloud (VPC) is the networking layer for Amazon EC2. VPC allows users to launch AWS resources into a virtual network [20] and implement top-level managements on all the resources

in the virtual network.

It contains a range of IP addresses and route tables responsible to determine how the network traffic is directed. VPC works similarly to the network that operates in traditional data centre.

2.3.3 Elastic Kubernetes Service (EKS)

Amazon Elastic Kubernetes Service (EKS) is a managed service that simplifies the process of running Kubernetes on AWS [21]. The cluster is managed by AWS and acts as a container orchestration tool with exactly the same capability as Kubernetes mentioned in Section 2.1.3.

2.3.4 Elastic Container Registry (ECR)

Amazon Elastic Container Registry (ECR) is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images [22]. ECR is used to store the image of worker and master pods in this project.

2.3.5 Elastic File System (EFS)

Amazon EFS provides file storage for Amazon EC2 instances. With Amazon EFS, a file system can be created and mounted to the EC2 instances. The mount allows read and write data between EC2 instances and the file system [23]. The purpose of having EFS is to store language models used by worker pods. EFS is the implementation of a network file system (NFS) by AWS to materialize distributed file system protocol.

Chapter 3

Designed Solution

This chapter will briefly explain the implementation of the designed solution. The solution is implemented on Amazon Web Service (AWS) using Terraform as the IaC tool. The system architecture is represented in Figure 3.1.

The architecture may look complicated and overwhelming, but the command required to build it is fairly simple: `terraform init | terraform apply`.

However, the process of setting up is not completed, the system is not fully functional at this stage. There are a few steps that should not or could not be provisioned using Terraform:

1. **Image building and pushing**
   Image is not considered as part of the architecture and images are dynamic.

2. **Mounting of file system**

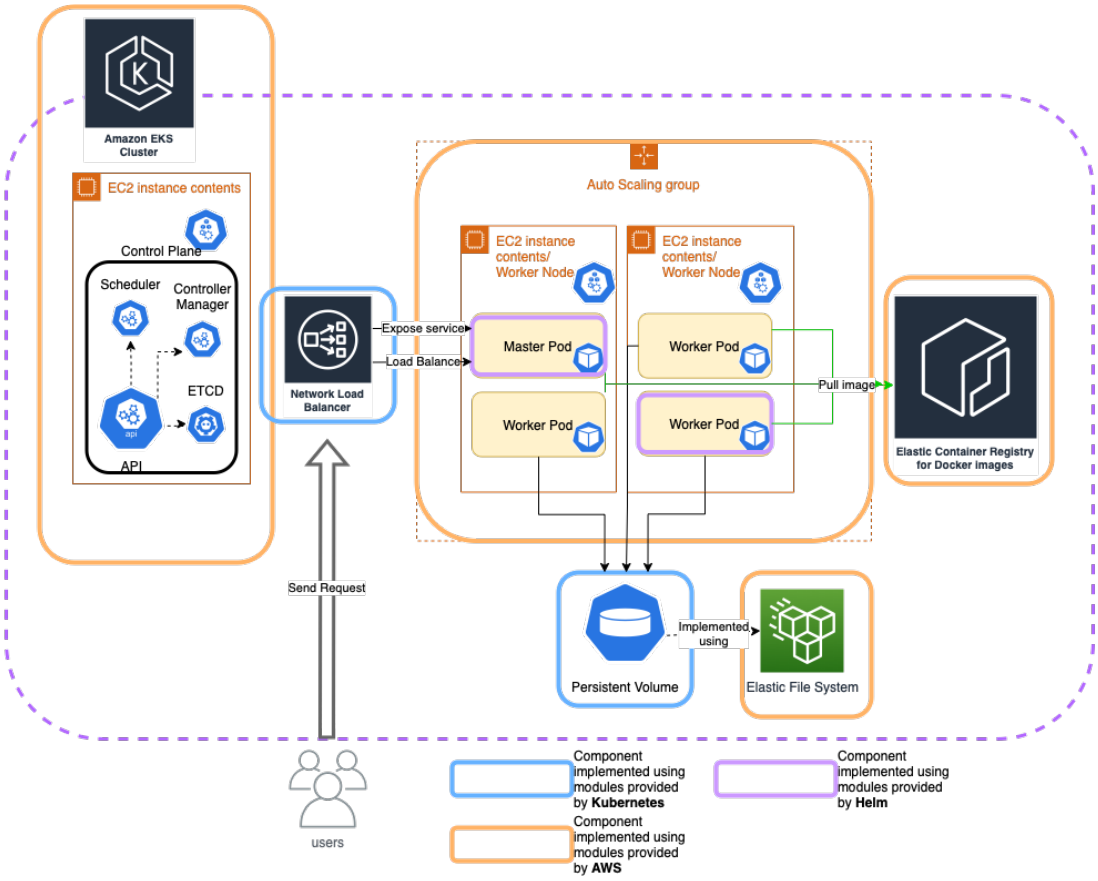

Figure 3.1: Architecture diagram of ASR system on AWS

Terraform does not provide resources to mount the file system to the EC2 instances.

3. **Uploading of models**

   The models are prone to adjustments and are also not part of the architecture.

The steps which are not included in the building plan of Terraform are realised using shell commands. Even though the steps are not provisioned by Terraform, the parameters from the resources built by Terraform are used in the shell commands. There is no argument that requires user input and the variables used are coherent with Terraform.

These steps can also be run in parallel while Terraform is building other

resources as long as the required AWS resources for the step are ready.

The image building and pushing process will need to wait until the resources Elastic Container Registry (ECR) and Elastic Kubernetes Service (EKS) are ready. The reasons are that ECR is needed to store the image pushed whereas the image building process involves Kubernetes configuration generated from EKS.

For mounting of file system and uploading of models, it needs to wait for the EC2 instance and Elastic File System (EFS) mount targets to be created.

The proposed solution can provision the infrastructure reliably in less than 20 minutes.

The command `terraform destroy` will destroy all the infrastructures created by Terraform. There is no need to revert any steps that are not included by the building plan of Terraform.

All the commands needed to construct the infrastructure can be found in the appendix.

Chapter 4

Detailed Implementation

This chapter will elaborate on the implementation of the designed solution. The detailed implementation of each of the components in the framework will be explained. The system architecture is represented in Figure 3.1

The implementation is divided into 4 sections, organised by their respective providers on Terraform. The fifth section will explain on the components that is not provisioned by Terraform. The modules and documentations can be found on Terraform Registry. [24]

Terraform supports references to values from other resources to minimised the number of arguments needed from the developers [25]. Parameters used in resources implemented using Terraform can also be output as values to make information about resources available on the command line. [26]

For example when an AWS EC2 instance is created using the following code:

```
resource "aws_instance" "example" {
    ami           = "ami-abc123"
    instance_type = "t2.micro"
}
```

The id of the instance is needed by other resources. In the Terraform file (.tf), the id attribute can be read by other resources using the syntax `aws_instance.example.id`.

If the attribute is needed in another environment e.g. Shell, it can be output using:

```
output "aws_instance_id" {
    description = "EC2 instance ID"
    value = aws_instance.example.id
}
```

From Shell, the value of it can be retrieved using the command `terraform output -raw aws_instance_id`. This feature ensures that the values used across Terraform and Shell are accurate and consistent.

4.1 Amazon Web Service

Modules provided by Amazon Web Services (AWS) on Terraform registry is used to interact with many resources supported by AWS.

Credentials must be provided for this to configure the resources managed by AWS. This solution uses static credential which is the access key and secret key for authentication of AWS account. The credentials are the only necessary input from user throughout the process.

Figure 4.1 shows the components that are implemented using modules provided by AWS.

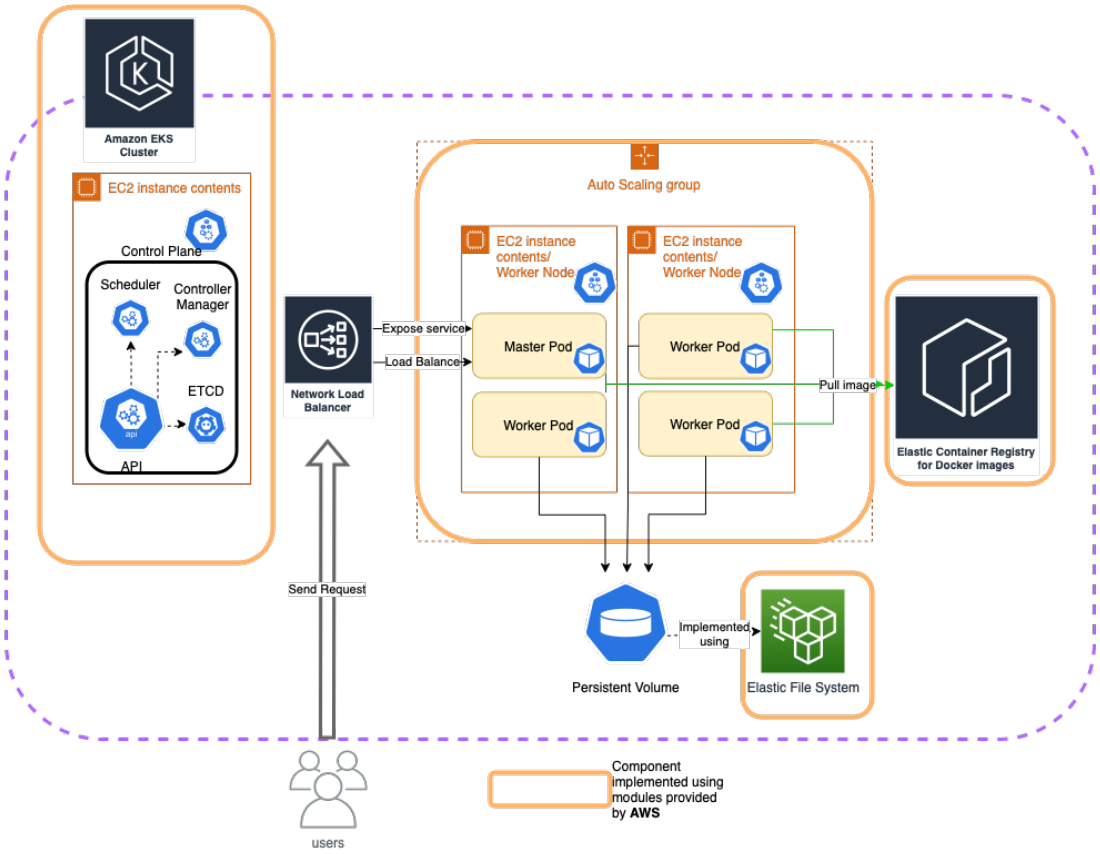

Figure 4.1: Components implemented using modules provided by AWS

AWS provider supports default tag and it is highly recommended to utilise this feature. The tag set in this module will be propagated, and all the subsequent resources created will also have the same tag. The tag can be used to identify the owner of the user and the environment that the developer is working on. With these tags, the development process will be traceable and manageable.

4.1.1 Virtual Private Cloud (VPC)

The Terraform module `vpc` is used to create a VPC and all the subsequent resources should be launched into this VPC. This provides a layer

of security, segregation of resources, and simplify the management.

This project uses a mix of public and private subnets. Private subnets are used for the resources within the VPC to communicate with each other, whereas the public subnets serve the purpose of communicating with external clients. The number of public and private subnets should be the same as the number of availability zones.

4.1.2 Elastic Kubernetes Services (EKS)

The Terraform module `eks` is used to create an Elastic Kubernetes (EKS) cluster and associate worker instances on AWS [27].

The specifications and configurations of the node groups are also declared in this module. The node group is a Kubernetes abstraction for a group of nodes within the cluster. Node group created will be an auto-scaling group.

Creation of EKS cluster will spin up the cluster master node which is responsible for managing the cluster. Control Plan within the master node coordinates all the activities in the cluster such as maintaining, scaling and scheduling of applications.

The information of the EKS cluster created is applied to the Terraform Kubernetes provider to configure Kubernetes for the creation of Kubernetes resources in Section 4.2.

The creation of the EKS cluster takes the most significant amount of time in the whole process of infrastructure setup. After the complete creation of the EKS cluster, the configuration of Kubernetes should be

exported as it will be copied into the image upon build.

4.1.3 Elastic Cloud Computing (EC2)

The computational resources EC2 are used to run Pods. The scheduler will only allocate a Pod to an EC2 instance that has a sufficient amount of vCPUs and memory to run the Pod.

The number of EC2 instances to be created will be auto-scaled based on the workload by Cluster Autoscaler discussed in Section 4.1.2.

4.1.4 Elastic File System (EFS)

The creation of EFS and mount targets were implemented using Terraform infrastructure code [28].

When using a cloud platform, a specific storage service is needed to implement the Persistent Volume [Section4.2.1] of Kubernetes. Network file system (NFS) is a common storage technology used to implement Persistent Volumes in Kubernetes. NFS is a distributed file system protocol, allowing a user on a client computer to access files over a computer network much like local storage is accessed [29].

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

Elastic File System (EFS) is used to store the language models. The EFS will be shared by the worker pods and it is being mounted to the computing nodes, EC2 on path `/efs`.

Figure 4.2 shows how the components EC2, EFS and mount target work together [30]:

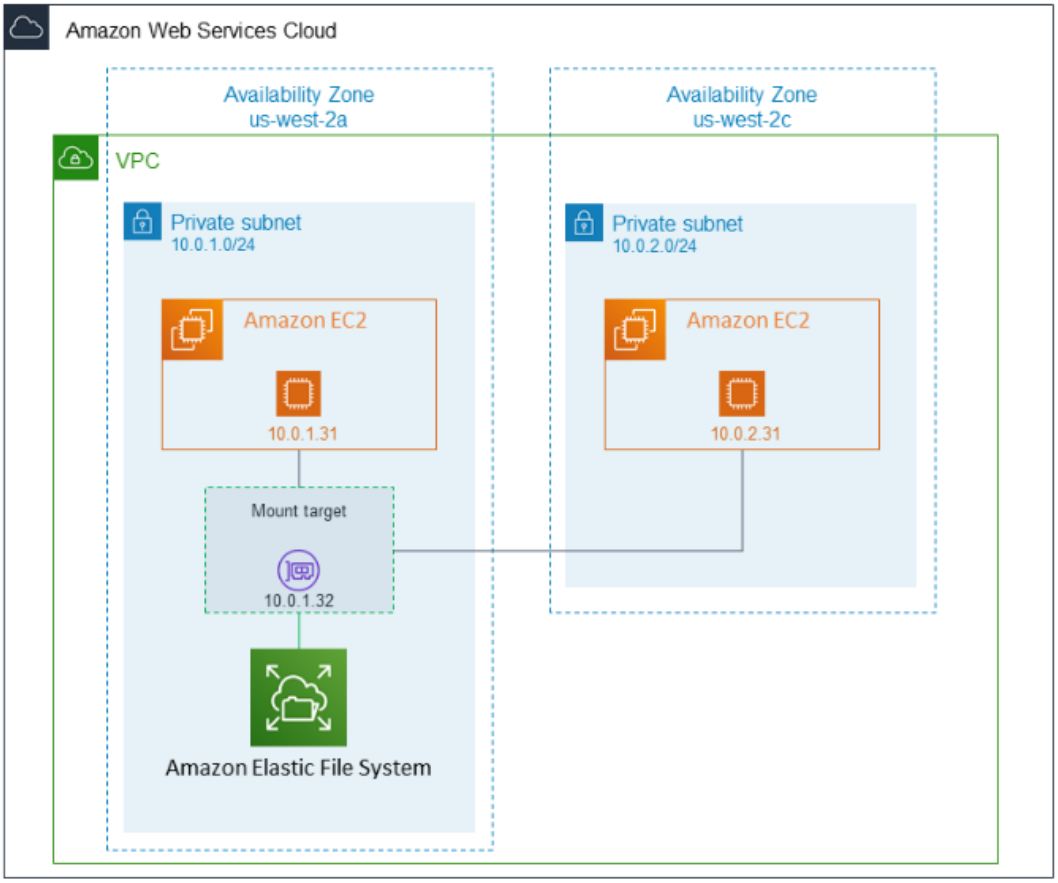

Figure 4.2: Relationship between EFS, EC2 and mount target

However, the mount action is done by using Linux command `mount` at the current stage and the upload of models to the file system is not being achieved using Terraform because the models are dynamic and not considered as part of the infrastructure. The recommended method of uploading is using `scp` command to transfer the models to the instances.

Figure 4.3 shows how worker pods and EFS are connected.

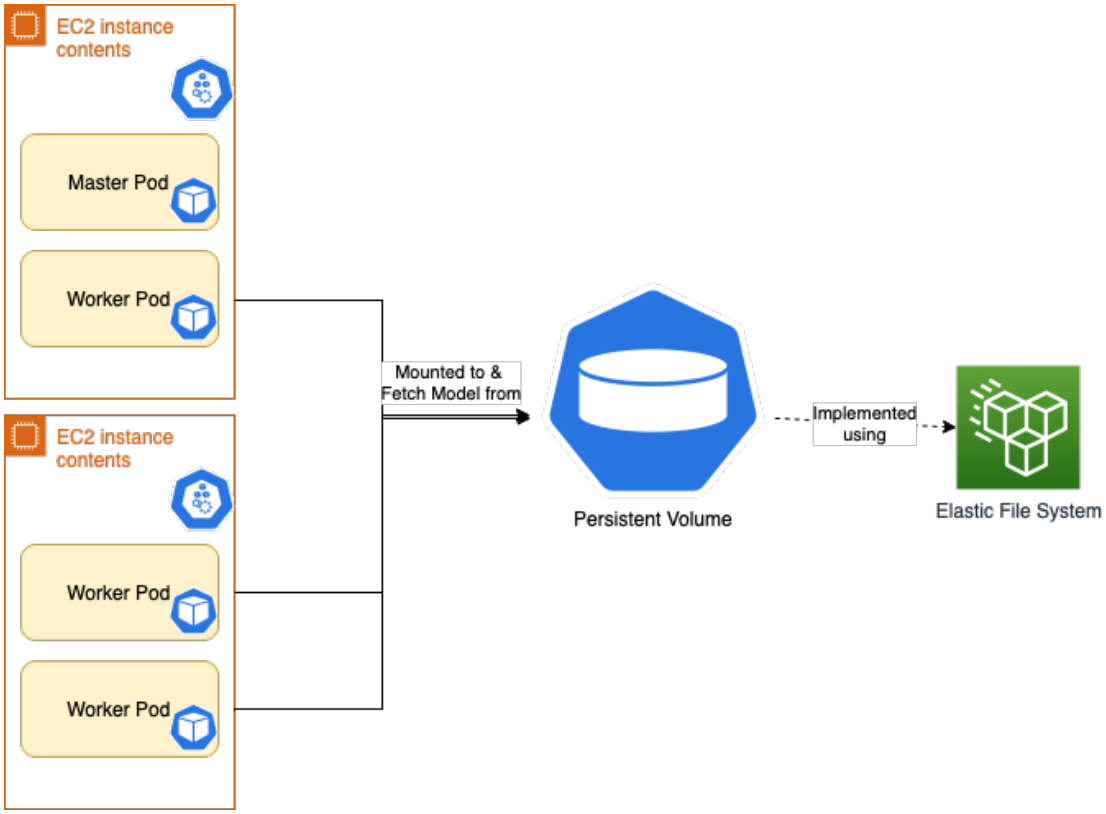

Figure 4.3: Relationship between EFS and worker pods

4.1.5 Key Pair

The module `aws_key_pair` will generate an EC2 key pair resource. A key pair is used to control login access to EC2 instances [31].

This key pair is generated using module `tls_private_key` by Terraform and encoded using Transport Layer Security (TLS) protocol. [32]

This key pair's public key will be registered with the EC2 created to allow logging-in using ssh and transferring of models files. The private key will be downloaded to the local computer as a certificate. The certificate is required for the ssh to the instance and uploading of the models. The certificate cannot be regenerated, it must be kept secured on the local machine.

Figure 4.4 illustrates the process of generating keys to log in to EC2 instances.

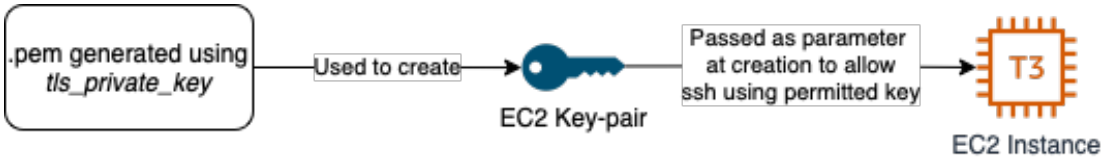

Figure 4.4: Process of key pair generation

4.1.6 Elastic Container Registry (ECR)

The module `aws_ecr_repository` creates and provides an Elastic Container Registry Repository (ECR) [33]. ECR is used to store images of worker and master pods.

The module only creates the repository, the image building and pushing process is done using Docker command because the image is not considered as part of the infrastructure and is dynamic.

Figure 4.5 shows the relationship between ECR, pods and Docker

4.2 Kubernetes

Provider Kubernetes is linked to the EKS cluster mentioned in Section 4.1.2. With such connection, Kubernetes resources that are absent in AWS EKS module can be implemented. Figure 4.6 shows the components that are implemented using modules provided by Kubernetes

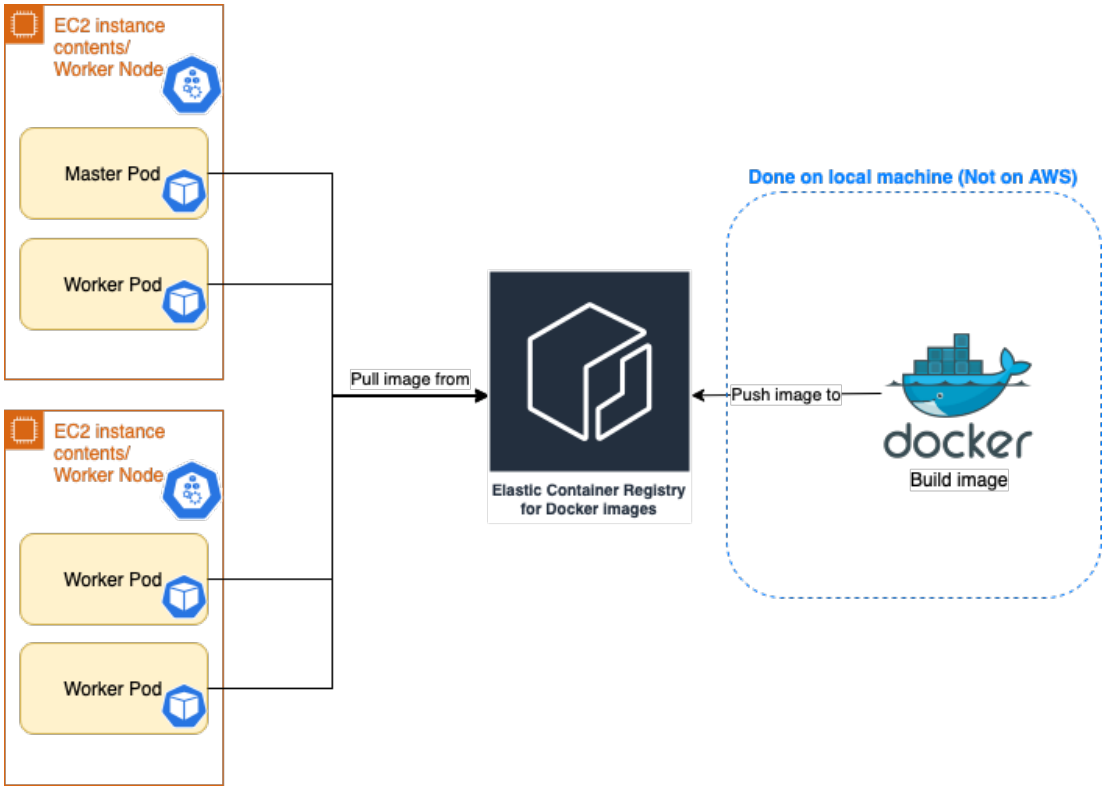

Figure 4.5: Relationship between ECR, pods and Docker

4.2.1 Persistent Volume

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.

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 is not aware of which storage technology is used for the Persistent Volume [34].

A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources.

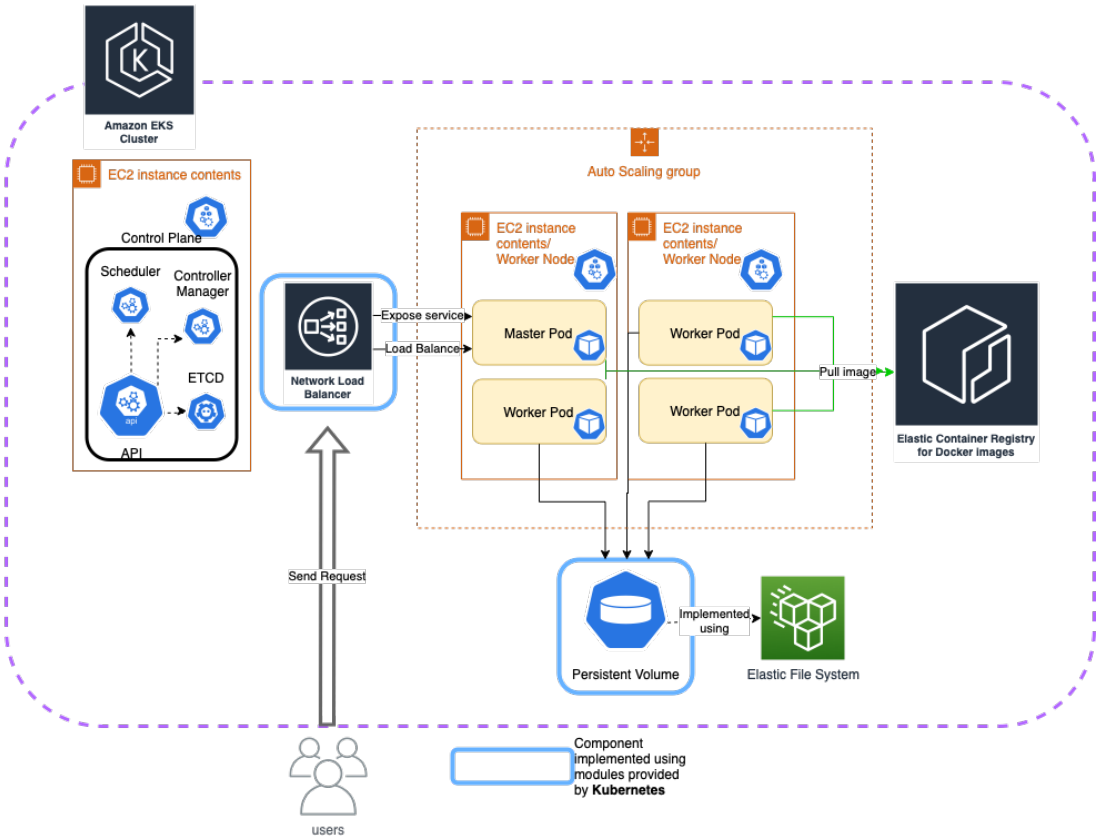

Figure 4.6: Components implemented using modules provided by Kubernetes

A Storage Class provides a way to provision Persistent Volume dynamically when Persistent Volume Claim claims it. [35]

Persistent Volume, Persistent Volume Claim and Storage Class are being built using Terraform infrastructure code using modules provided by Kubernetes. Figure 4.7 illustrates the relationship between the 3 resources.

4.2.2 Ingress

Ingress is a collection of rules that allow inbound connections to reach the endpoints defined by a backend which is the master pod of the ASR system. This project uses Ingress to have an externally-reachable URL

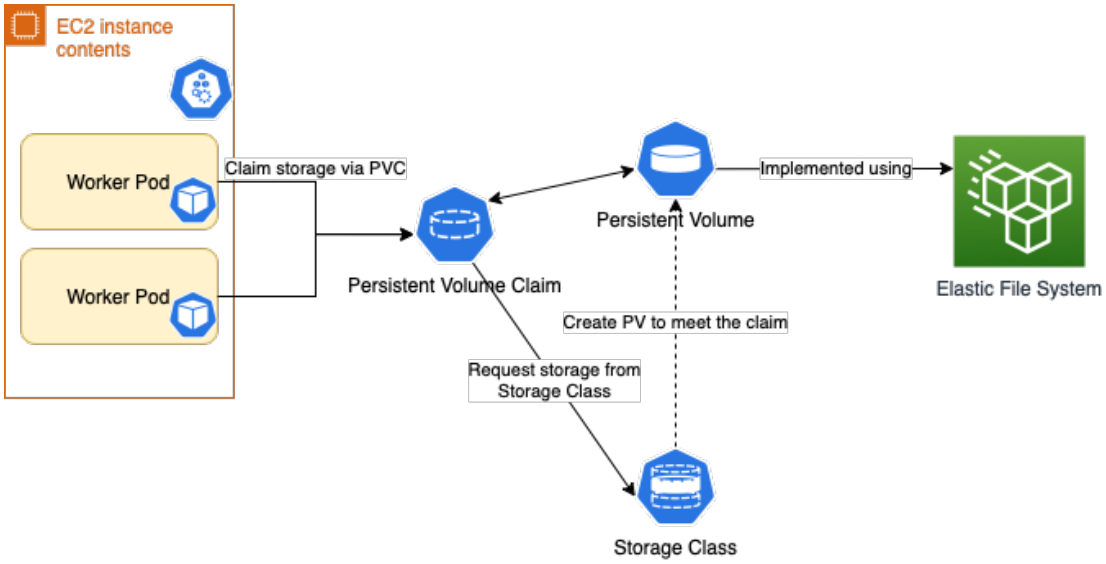

Figure 4.7: Relationship between PV, PVC and Storage Class

and to load balance network traffic. [36]

Ingresses can be implemented by different controllers, often with a different configurations. More detail about the ingress controller will be explained in the Section 4.3.3

4.3 Helm

The module provides the capability to manage installed Charts in the Kubernetes cluster, in the same way as Helm, through Terraform [37]. Similar to Section 4.2, this module requires configuration on Kubernetes to deploy the Helm Chart on the Kubernetes cluster.

Helm artifactory hub has diverse actively maintained Charts and each of the charts will create all the Kubernetes resources needed for the applications, making the deployment process smooth and error-free. With the `helm_release` module, Terraform are able to provision more Kubernetes resources with complexity hidden.

Even though the Helm chart is installed through Terraform, the pre-configured values of the Helm Chart can be overwritten, allowing users to have some extent of flexibility on the chart being deployed.

Figure 4.8 shows the components that are implemented using modules provided by Helm.

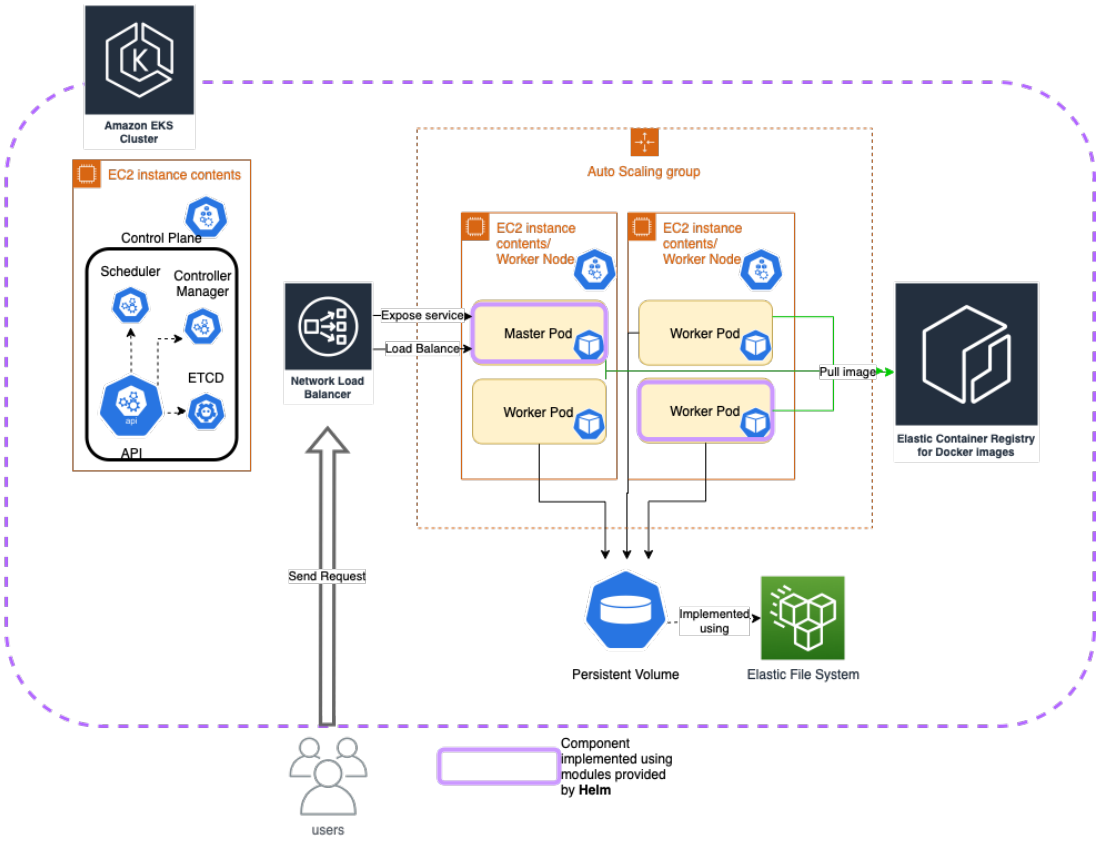

Figure 4.8: Components implemented using modules provided by Helm

4.3.1 Master and Worker pods

The local chart to deploy master and worker pods is being installed to the Kubernetes cluster using Helm through Terraform.

The values in the local chart include information about the repository of Docker image (ECR) and also the command that should be run at the

start of container. Besides, it also specifies the number of worker pods to be spun up initially for each language model.

4.3.2 EFS CSI Driver

The Amazon Elastic File System Container Storage Interface (CSI) Driver provides a CSI interface that allows Kubernetes clusters running on AWS to manage the lifecycle of an EFS system. It is a complementary service that is mandatory for EKS to manage the attachment of EFS volumes to Kubernetes pods.

AWS does not have Terraform modules to implement the driver, but it is available on Helm and is maintained by Kubernetes. EFS CSI Driver is provisioned using the `helm_release` modules.

4.3.3 Ingress Controller

Ingress controller is a specialized load balancer for Kubernetes. It is accountable to accept traffic from outside the Kubernetes platform and load balancer it to Kubernetes pods running inside the platform.

There is multiple ingress controllers in the market. This project uses Nginx Ingress Controller as it supports features such as dynamic reconfiguration and web application firewall, protecting the Kubernetes services from attacks.

Load balancer type used previously was the legacy classic load balancer. It has been updated to Network Load Balancer (NLB) when implementing Ingress using Terraform. AWS EC2 will drop support for classic load balancer soon [38].

AWS is lacking of trustworthy Terraform modules to implement the controller, it is provisioned using Helm modules on Terraform.

4.4 Non-official provider

4.4.1 Cluster Autoscaler

Cluster Autoscaler on AWS utilizes EC2 Auto Scaling Groups to implements Kubernetes node groups. It is granted with the permissions to examine the node groups and launch new EC2 instances to meet the computational demand.

Cluster Autoscaler uses the auto-discovery setup to find the EC2 Auto Scaling groups tagged with the cluster name. [39].

There are two options in the usage of Terraform modules to implement the cluster autoscaler.

1. **Helm**

   Cluster Autoscaler is also available as Helm chart on the artifactory. However, the Helm chart solely install the auto-scaler without the permission and service roles needed to control the auto-scaling group. This method requires extra effort to create the AWS service roles.

2. **Non-official Terraform module**

   The module is managed by Lablabs. Under the table, the module still uses Helm to install the auto-scaler, but it also helps to create the required roles and service account, resulting in a fully

functional Cluster Autoscaler [40]. The concern of using a non-official module is that it might be discontinued for updates and lack of documentation.

This project uses the second option in spite of the concerns that it possesses because the maintainer of the module, Lablabs is well-established with a number of other Terraform modules in development. The module was also provisioned by a huge number of Terraform developers, therefore it should be stable in terms of usage.

4.5 Resources not covered by Terraform

The actions mentioned in this section are the steps not being included in the Terraform building plan. They need to be run using shell commands, check the appendix for the commands.

Despite that Terraform provides a resource known as `null_resource` using `local-exec` provisioner which invokes local executable after a resource is created or in other words, runs a command. Terraform can not guarantee that the command run will result in the creation of an operable resource. In most situations, the usage of this resource is to be avoided as it violates the initial aim of adopting Terraform for its robustness and manageability. [41]

4.5.1 Build and push image

Image is built using Docker and it needs to wait for the completion of EKS creation as the Kubernetes configuration file is needed to run the

containers. There is no module in Terraform to achieve it. Another reason being images are dynamic when compared to other components in the infrastructure. It should not be considered as part of the architecture, the dynamic property of images make the Terraform provision step inconsistent.

After the image is built, it needs to be pushed to ECR to be run by Kubernetes as a container within a pod. Before pushing the image to ECR, the image should be tagged according to the standard of ECR. Using the credential to log in to ECR, Docker will push the image to the corresponding ECR.

4.5.2 Mount file system and upload models

The mount target address is one of the output values from Terraform, it will be written to a text file and be copied into the EC2 instance for mounting.

Using the certificate generated and saved to local machine, the user have access to the EC2 instance. In the EC2 environment, a directory `./efs` is created. The name of the directory should not be changed as it shares the same name as stated in the Dockerfile. After that, the directory is mounted to the mount target of the EFS. All the files written to this directory will be stored in the EFS.

After the directory `./efs` is created and successfully mounted to EFS, the models can be uploaded into the instance using the same certificate saved in the local environment.

These steps have an upstream dependency on EFS and EC2 instance

creation.

Chapter 5

Conclusion & Future Work

This chapter will wrap up the project by summarising the achievement and propose possible improvements to the project.

5.1 Conclusion

The objective of this project is to design a reliable and pragmatic solution to provision and un-provision the ASR system on Cloud. While ensuring the flexibility to changes, the steps required to build the infrastructure were streamlined and simplified.

The project revolved around Terraform, a tool that made the concept of Infrastructure as Code (IaC) accessible and viable. Terraform established a highly replayable workflow to build the system. The hassles of using multiple tools such as terminal and AWS console to provision

the infrastructure had been eliminated. The complexity of scripts was greatly reduce, accelerating the process of development and would benefit developers working on the system. Developers can speedily provision the elementary infrastructure and start working on introducing new features to the system. Upon the completion of experiments, developers can destroy the infrastructure gracefully, leaving the Cloud environment neat and clean.

The benefits of using Terraform goes beyond the development stage, it provides traceability and manageability for the infrastructure created, which are the key characteristics for any industrial standard application. With Terraform states, updates and the dependencies between components are fully handled by Terraform. Terraform is able to detect the changes made and roll out the updates effectively onto the existing infrastructure, resulting in better robustness of the system.

A big part of the infrastructure of the ASR system can now be built or updated using a simple terraform command - `terraform apply`. The time taken to boot up and tear down the infrastructure was significantly reduced as the resource dependencies are handled by Terraform, achieving parallelism in the workflow.

In conclusion, this project has proven that the concept of Infrastructure as Code (IaC) should be the standard for architecture development in the future. With IaC, the build is repeatable, reliable and consistent.

5.2 Future Work & Possible Improvements

5.2.1 Wider usage of Terraform

The combination of Terraform and Helm provides numerous packages that can be deployed. Most of the industrial standard tools or frameworks such as Prometheus and Vault have their corresponding official modules in Terraform. If not Terraform, usually there will be a Helm chart to use.

Unfortunately, Terraform is not cloud-agnostic, meaning that most infrastructure codes are not reusable when switching to another Cloud service providers such as Azure and Google Cloud Platform. However, Terraform also has modules specific for those cloud service providers.

Future developments should use Terraform as an approach to implements these frameworks if possible.

5.2.2 Version Control of Terraform

Integrating with GitHub may help showcase the value of version-controlled infrastructure with Terraform. In addition to providing a single, familiar view where Terraform users can see the status and impact of their changes, the integration also brings about continuous integration and testing for infrastructure changes. The consistent GitHub workflow pairs well with HashiCorp's goals of providing a workflow for provisioning, securing, and running any infrastructure for any application. [42]

With version-controlled Terraform configuration files, the old architec-

ture can be fetched and use for rollback if the newly introduced architecture faces any issue or error.

5.2.3 Automated Terraform

It is desirable to orchestrate Terraform to run in automation in order to ensure consistency between runs. This can be achieved using continuous integration/continuous delivery (CI/CD) tools to automate the provision of infrastructure.

A simple CICD workflow using Github action could be:

1. Check whether the configuration is formatted properly
2. Generate a plan for every pull request
3. Apply the configuration when pull request merged to the main branch

5.2.4 Modifications on Helm Chart capture by Terraform

Terraform implementation is not perfect, there is a flaw when using the `helm_release` module. Changing of value to an existing resources managed by Helm could not be captured by Terraform, so the changes would not be applied. [43]

There are some workarounds on it but the providers are also proactively testing this functionality and should be released in future updates.

Bibliography

[1] Wong Cassandra. *Speech Recognition system can transcribe Singapore lingo in real time*. Sept. 2018. URL: https://sg.news.yahoo.com/speech-recognition-system-can-transcribe-singapore-lingo-real-time-131406725.html.

[2] Matthew Zajechowski. *Automatic Speech Recognition (ASR) Software - An Introduction*. URL: https://usabilitygeek.com/automatic-speech-recognition-asr-software-an-introduction/.

[3] Ma Xiao. *Docker and Kubernetes : deploying speech recognition system for scalability*. 2021. URL: https://hdl.handle.net/10356/148041.

[4] Wong Seng Wee. *Deploying speech recognition system using high availability and scalability kubernetes cluster with kubernetes and docker*. 2020. URL: https://hdl.handle.net/10356/137980.

[5] Jaya Sharma. *AWS vs. Azure: Which One Is Better and Which One Should You Choose*. Jan. 2021. URL: https://www.naukri.com/learning/articles/aws-vs-azure/.

[6] Lenke Hannes. *Monitoring as Code with Terraform Cloud and Checkly*. Apr. 2021. URL: https://www.hashicorp.com/

blog/monitoring-as-code-with-terraform-cloud-and-checkly.

[7] IBM Cloud Education. *Containerization*. May 2019. URL: https://www.ibm.com/sg-en/cloud/learn/containerization#toc-what-is-co-r25Smlqq.

[8] Citrix. *What is containerization?* URL: https://www.citrix.com/solutions/app-delivery-and-security/what-is-containerization.html.

[9] phoenixNAP. *Docker Image vs Container: The Major Differences*. Oct. 2019. URL: https://phoenixnap.com/kb/docker-image-vs-container.

[10] docker docs. *Docker Overview*. URL: https://docs.docker.com/get-started/overview/.

[11] kubernetes. *Production-Grade Container Orchestration*. URL: https://kubernetes.io.

[12] *Understanding Kubernetes Autoscaling*. URL: https://blog.scaleway.com/understanding-kubernetes-autoscaling/.

[13] AWS Whitepaper. *Infrastructure as Code*. URL: https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/infrastructure-as-code.html.

[14] Terraform. *Overview of Terraform*. URL: https://www.terraform.io.

[15] Terraform. *Terraform State*. URL: https://www.terraform.io/language/state.

[16] Helm. *Helm Docs*. URL: https://helm.sh/docs/.

[17] *Artifact Hub*. URL: https://artifacthub.io/packages/search?kind=0&sort=relevance&page=1.

[18] BroadCom. *Anatomy of a Container Gateway Helm Chart*. URL: https://techdocs.broadcom.com/content/dam/broadcom/techdocs/us/en/assets/docops/gateway/gwhelmchartflow.png.

[19] Amazon. *Amazon EC2*. URL: https://aws.amazon.com/ec2/.

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

[21] Amazon Web Service. *Amazon Elastic Kubernetes Service Documentation*. URL: https://docs.aws.amazon.com/eks/?id=docs_gateway.

[22] Amazon Web Service. *Amazon Elastic Container Registry Documentation*. URL: https://docs.aws.amazon.com/ecr/?id=docs_gateway.

[23] Amazon Web Service. *Amazon Elastic File System Documentation*. URL: https://docs.aws.amazon.com/efs/?id=docs_gateway.

[24] Terraform. *Terraform Registry*. URL: https://registry.terraform.io.

[25] Terraform. *References to Named Values*. URL: https://www.terraform.io/language/expressions/references.

[26] Terraform. *Output Values*. URL: https://www.terraform.io/language/values/outputs.

[27] Terraform AWS Modules. *eks*. URL: https://registry.terraform.io/modules/terraform-aws-modules/eks/aws/latest.

[28] Amazon. *Step 3: Mount the file system on the EC2 instance and test*. URL: https://docs.aws.amazon.com/efs/latest/ug/wt1-test.html.

[29] Wikipedia. *Network File System*. URL: https://en.wikipedia.org/wiki/Network_File_System.

[30] Amazon Web Service. *Creating and managing mount targets*. URL: https://docs.aws.amazon.com/efs/latest/ug/accessing-fs.html.

[31] Terraform AWS Provider. *Resource: aws_key_pair*. URL: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/key_pair.

[32] Hashicorp. *Terraform Provider TLS*. URL: https://registry.terraform.io/providers/hashicorp/tls/latest.

[33] AWS. *Resource: aws_ecr_repository*. URL: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/ecr_repository.

[34] Kubernetes. *Persistent Volumes*. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[35] TechWorld with Nana. *Kubernetes Volumes explained — Persistent Volume, Persistent Volume Claim Storage Class*. URL: https://youtu.be/0swOh5C3OVM.

[36] Terraform Kubernetes Provider. *kubernetes_ingress*. URL: https://registry.terraform.io/providers/hashicorp/kubernetes/latest/docs/resources/ingress.

[37] Terraform Helm Provider. *Helm Provider*. URL: https://registry.terraform.io/providers/hashicorp/helm/latest/docs.

[38] AWS. *Migrate your Classic Load Balancer*. URL: https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/migrate-classic-load-balancer.html#migrate-step-by-step-classiclink.

[39] Amazon Web Service. *Cluster Autoscaler on AWS*. URL: https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/README.md.

[40] Lablabs. *eks-cluster-autoscaler*. URL: https://registry.terraform.io/modules/lablabs/eks-cluster-autoscaler/aws/latest.

[41] Terraform. *local-exec Provisioner*. URL: https://www.terraform.io/language/resources/provisioners/local-exec.

[42] Seth Vargo. *Version-Controlled Infrastructure with GitHub Terraform*. URL: https://www.hashicorp.com/blog/version-controlled-infrastructure-with-github-and-terraform.

[43] Hashicorp. *Values modified outside of terraform not detected as changes*. URL: https://github.com/hashicorp/terraform-provider-helm/issues/372.

Chapter 6

Appendices

6.1 Script to provision the infrastructure

```
# cd to /terraform file
# configure AWS: aws configure
terraform init
terraform validate
terraform apply

export AWS_REGION=$(terraform output -raw region)
export AWS_ACCOUNT=$(terraform output -raw awsaccount)
export CLUSTER_NAME=$(terraform output -raw cluster_name)
export PUBLIC_DNS=$(terraform output -raw public_dns)
export MOUNT_TARGET_ADDR=$(terraform output -raw mount_target_addr)
export NAMESPACE=$(terraform output -raw namespace)
export HELM_CHART=$(terraform output -raw helmchart)
export PRIVATE_KEY=$(terraform output -raw private_key_name)
```

```
export IMAGE_NAME=$(terraform output -raw image_name)

aws eks --region $AWS_REGION update-kubeconfig \
    --name $CLUSTER_NAME

echo $MOUNT_TARGET_ADDR > /tmp/mount_target_addr.txt
chmod 400 "$PRIVATE_KEY_NAME.pem"
scp  -i "$PRIVATE_KEY_NAME.pem" /tmp/mount_target_addr.txt \
    ec2-user@$PUBLIC_DNS:./
ssh -i "$PRIVATE_KEY_NAME.pem" ec2-user@$PUBLIC_DNS

# commands to run within instance
mkdir ~/efs
sudo mount -t nfs -o nfsvers=4.1,rsize=1048576,wsize=1048576, \
    hard,timeo=600,retrans=2,noresvport \
    $(cat mount_target_addr.txt):/ ~/efs
sudo chmod go+rw ~/efs/
exit
# Exited from the instance

# Upload image
scp -r /fyp2021/models/* ec2-user@$PUBLIC_DNS:./efs

# build and push container image
sudo cp ~/.kube/config ../docker/secret/
docker build -t $IMAGE_NAME ../docker/
```

```
docker tag $IMAGE_NAME:latest \
    $AWS_ACCOUNT.dkr.ecr.$AWS_REGION.amazonaws.com \
    /$IMAGE_NAME:latest

# push image to registry
aws ecr get-login-password --region $AWS_REGION | \
    sudo docker login --username AWS --password-stdin\
    $AWS_ACCOUNT.dkr.ecr.$AWS_REGION.amazonaws.com
docker push $AWS_ACCOUNT.dkr.ecr.$AWS_REGION.amazonaws.com\
    /$IMAGE_NAME:latest
```