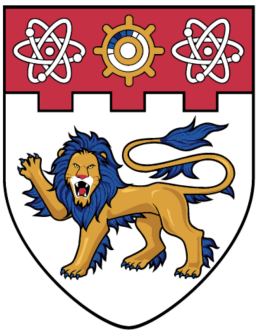

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

School of Computer Science and Engineering

Docker and Kubernetes: Deploying Speech Recognition System

Final Year Project

Chen Benjamin

Supervisor: Associate Professor Chng Eng Siong

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

Abstract

This project aims to provide a Continuous Integration and Continuous Deployment (CI/CD) solution for the automatic speech recognition (ASR) system. The CI/CD solution will automate much of the manual work when deploying new versions of the ASR system such as building, testing and pushing images to the container registry, thus saving time and allowing our engineers to focus more on developing the ASR system. The solution is implemented on both Gitlab and Jenkins to compare 2 of the most popular CI/CD tools, and with the help of Terraform, not only are we deploying code changes, we are also able to deploy infrastructure changes as well.

The solution also aims to continuously deploy new versions of the ASR system onto multiple cloud platforms at once, thus giving developers a single point of interaction when working with so many cloud providers. This report will present the solution with the help of architectural diagrams, and detailed guides to implement the solution. Lastly, the report will compare the benefits of the 2 CI/CD tools used for this project.

Acknowledgments

This project was made possible by the support and guidance of the following people who helped me during the project. I would like to express my gratitude and appreciation to them.

Firstly, I would like to express my appreciation to Research Associate Vu Thi Ly, for her help and guidance when I was really lost and had no clue what was going on. Thanks to her patience and willingness to help, I was able to pick up a lot of skills along the way and learned a lot about Kubernetes applications.

Next, I would like to express my appreciation to my supervisor, Professor Chng Eng Siong for allowing me to focus my time on school work and internship during the first semester of this project. It allowed me to gain inspiration and think of new ideas for this project.

Lastly, my friends and family who provided support throughout the project.

Contents

List of Figures

1 Introduction

1.1 Background

The automatic speech recognition (ASR) system is based on an open-source toolkit, Kaldi [1], which applies a finite-state transducer to conduct the speech recognition. The ASR system is able to take in audio files or real-time audio streams and return transcripts of the audio data received.

Researchers from NTU have developed speech recognition models for the ASR system to transcribe various languages to transcripts. They even developed models that could transcribe audio data containing a mix of languages (e.g Mandarin and English), which could potentially be very useful in Singapore due to the very "Singaporean" habit of mixing English with their mother tongue when conversing. Once the system has been deployed, it is targeted to support various services around Singapore such as call centres, customer service centers and more.

1.2 Importance of Project

While the ASR system is still in its early stages of development, the goal of the project is to eventually deploy the system to various organizations around Singapore once the entire operation of the system is self sustainable. These organizations will have existing infrastructures in different cloud platforms such as Azure, AWS, Google Cloud or others, and they will require us to deploy our system into their infrastructure. Once the system

is deployed for these organizations, it is our job to maintain the system and provide software updates when necessary.

Imagine having more than 100 organizations all using our ASR system each deployed into their own compute clusters across various cloud platforms, and our engineers are tasked to deliver a software update to the system periodically due to the ever changing behaviour of the human speech. We need a way to safely and efficiently deliver the new software update to all these organizations who are using our system.

To update the ASR application for one organization, we need to connect to their compute clusters and execute certain commands, and to update the system infrastructure, we need to log into their cloud console and perform manual configurations. While this is manageable when the ASR system is being used by a few organizations, it can quickly get out of hand when more organizations start using the system.

1.3 Scope

The objective of this project is to develop and implement a continuous integration and continuous deployment (CI/CD) pipeline which can safely and reliably deliver software and infrastructure updates [2] of the ASR application to multiple compute clusters across various cloud platforms, namely Azure, Google Cloud and Amazon Web Services.

The solution will be implemented using two of the best CI/CD tools in the industry, Gitlab and Jenkins [23] with Terraform integrated into them to help deliver infrastructure updates.

At the end of the project, we will be comparing the 2 CI/CD tools in terms of how easy it is to use, how easy it is to learn, how well does it support CI/CD utilities and lastly it's cost of use.

1.4 Report Organisation

This report consists of 5 chapters, each chapter explains the following:

- Chapter 1 gives a high level overview of the entire project and explains the importance and scope of this project.
- Chapter 2 provides the details about the technologies used in designing and implementing this project.
- Chapter 3 illustrates the design of the solution of this project with the use of architectural diagrams.
- Chapter 4 elaborates on the implementation of the solution.
- Chapter 5 concludes the project and discuss possible future improvements to this project.

2 Literature Review

The project is implemented using Gitlab and Jenkins as the CI/CD tools. While the CI/CD tools used for this project are different, they will still be managing the same underlying architecture which is hosting our ASR application. They will also both be using Terraform and Helm to deploy architectural changes and software changes respectively.

The figure below illustrates how the different technology stacks selected for this project is utilized to build our CI/CD solution with Gitlab. The technology stack used for the Gitlab CI/CD solution includes the following:

- Gitlab - Forms base of our architecture by providing basic tools for our implementation to work such as a git repository, terraform remote backend, CI/CD servers, container registry and more.
- Azure - Infrastrucure for our ASR application.
- Google Cloud - Infrastrucure for our ASR application.
- AWS - Infrastrucure for our ASR application.
- Terraform - Manages infrastructure which hosts our ASR application.
- Helm - Manages deployment of our ASR application.

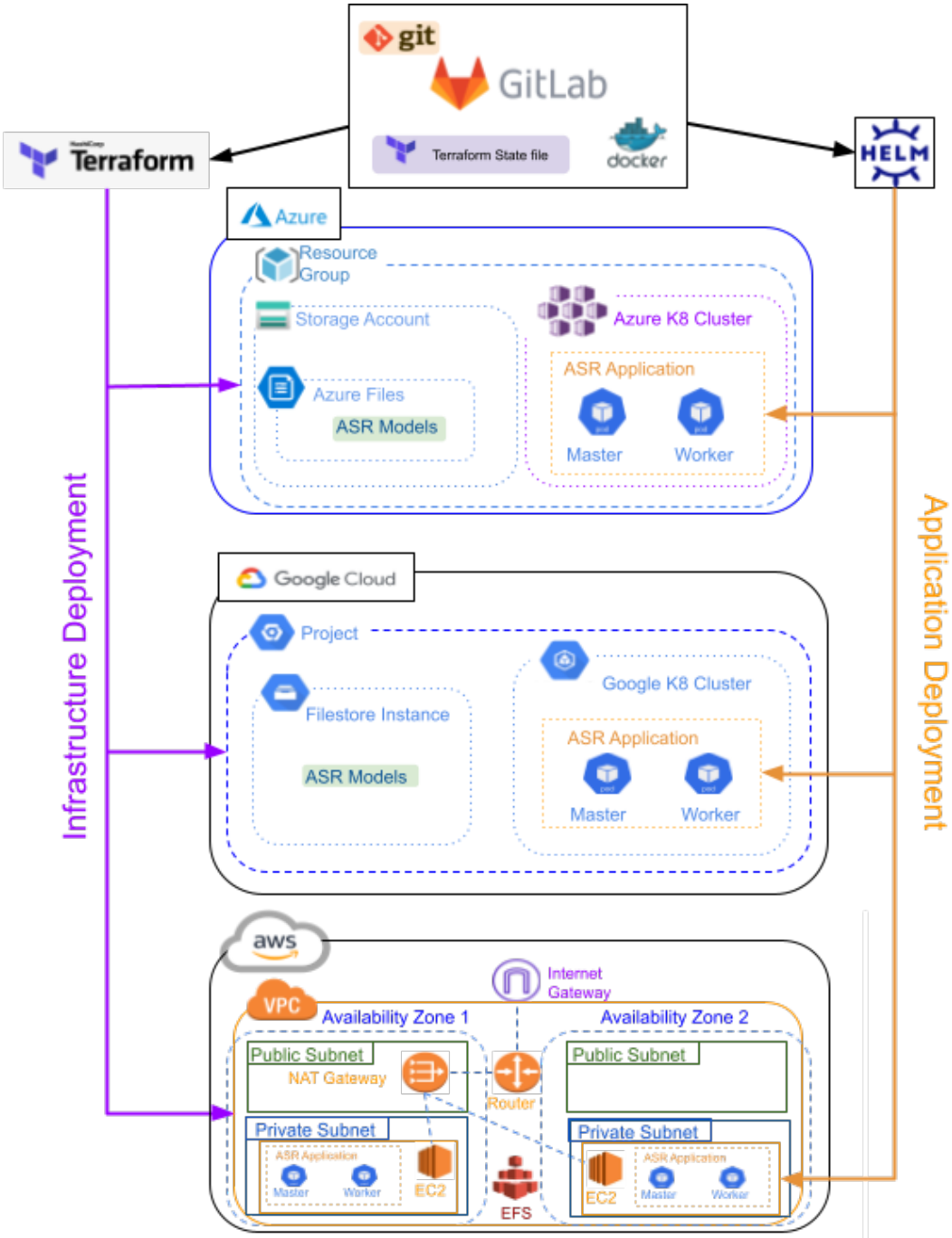

Figure 1: Gitlab CI/CD Architecture

For Gitlab, a lot of components required for the CI/CD pipelines are provided on the Gitlab platform, such as a remote backend for our Terraform State file, a docker container registry to store our ASR images and a Git repository, for management of our source code.

Next, figure below illustrates the architecture using Jenkins. Note that we are not changing anything about our application or the infrastructure of the application. Hence, majority of the Figure is similar to Figure 1 shown above.

When using Jenkins, we require lots of external services as Jenkins mainly focuses of CI/CD. It does not have any helper services to support the CI/CD pipelines. Therefore, we require the following technology stack this time:

- Github - Git repository which manages our ASR project.
- Jenkins - CI/CD server that runs continuous deployment and delivery jobs for us.
- Docker hub - Private container registry to manage our ASR application image.
- Terraform Remote Backend - Stores our terraform state file, for this project, we configured a storage account in Azure to host the terraform backend.
- Azure - Infrastrucure for our ASR application.
- Google Cloud - Infrastrucure for our ASR application.
- AWS - Infrastrucure for our ASR application.
- Terraform - Manages infrastructure which hosts our ASR application.
- Helm - Manages deployment of our ASR application.

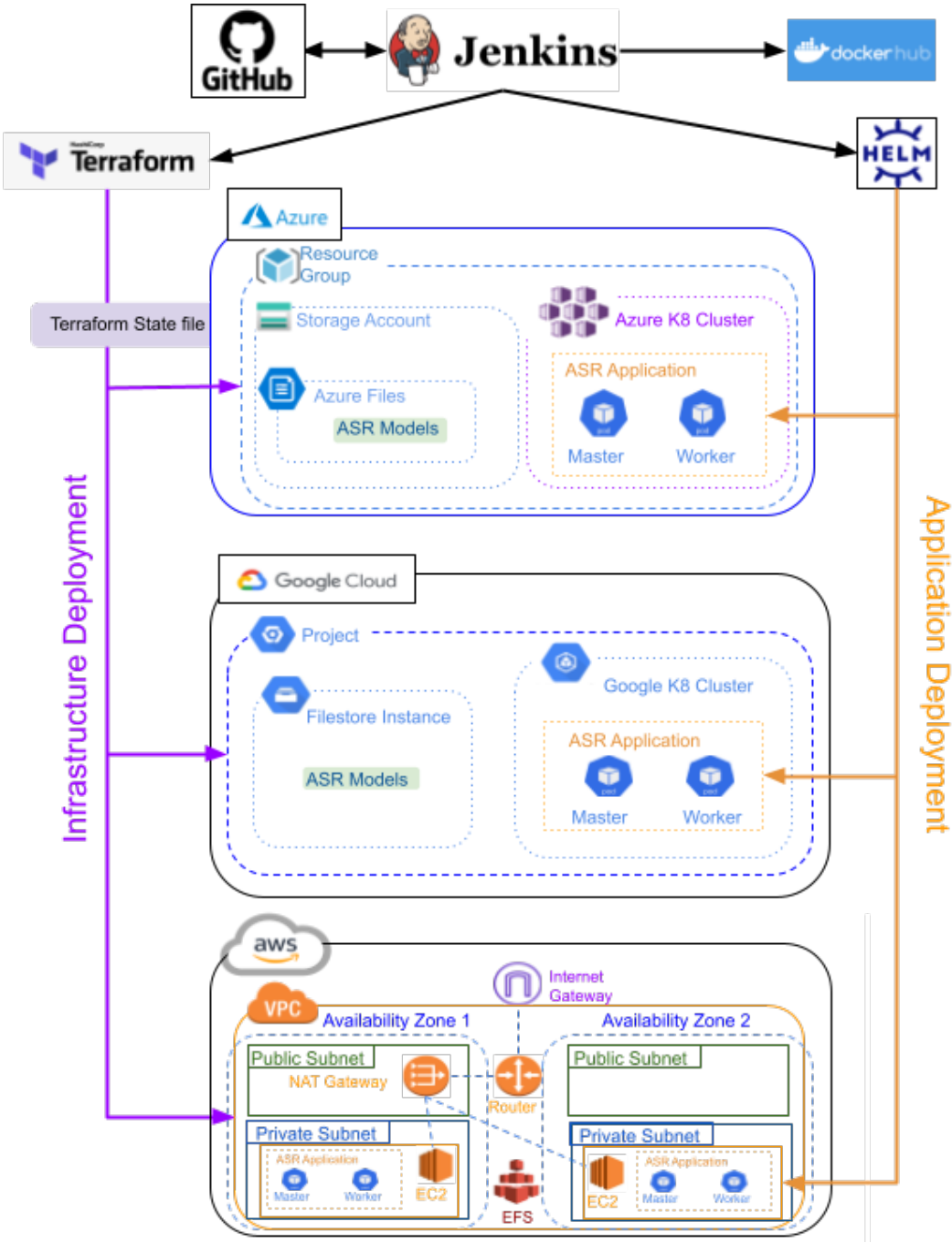

Figure 2: Jenkins CI/CD Architecture

The following sections below will provide a more detailed explanation of the technology stacks used in this project.

2.1 Docker and Containerization

Docker is an open source containerization tool used to package applications into a container image which can then be used to spin up container instances to run your application anywhere [14]. Containers are increasingly becoming a more popular vessel to run applications than VMs.

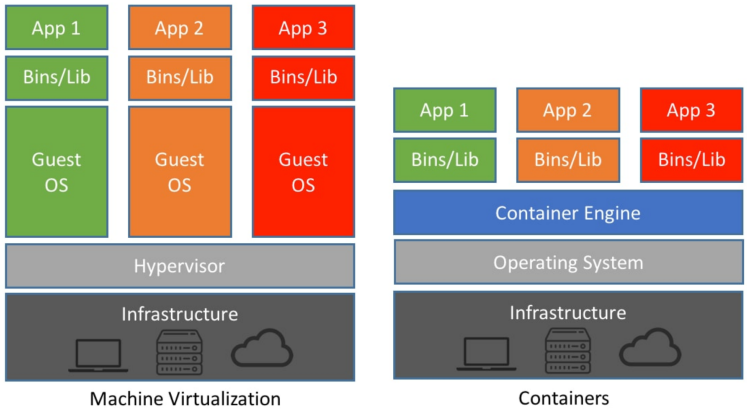

Figure 3: VM vs Containers [19]

This is due to containers being more lightweight than VMs as they share one operating system while each VM requires its own operating system [12].

2.2 Kubernetes

Kubernetes is an open source container orchestration engine that automates the deployment, scaling, and management of containerized applications [6]. Containerization technology and Kubernetes go hand in hand. While containers focus on running applications, Kubernetes will work on supporting the containers. The orchestration tool comes with many useful features to facilitate container operation such as the following:

- Scheduling - Containers are run in pods, which are the smallest unit

of deployable workload that can be created in Kubernetes. Each time a new pod is created, the Kubernetes scheduler will look for the best possible worker node to assign the pod to. The decision is based on the amount of available resources in that node for the pod. This helps to ensure that workload among the worker nodes are evenly distributed and avoids the situation where certain nodes are overloaded [5].

- Service Exposure - Kubernetes use a feature called service to expose containers running in pods as a network service to the public. When running a Kubernetes application, pods are constantly being created and destroyed and each time the lifecycle of a pod is renewed, it gets a new IP address assigned to it. Kubernetes services help to expose a static IP address to the public despite the constantly changing IP addresses of the pods.

- Persistent Storage - When a pod's lifecycle is renewed, all the data in the pod is lost. Kubernetes solves this problem by using a feature called Persistent Volumes to provide an abstract way to mount a shared file system onto these pods when they are created. This is useful for stateful applications which require client data to be saved. It is also useful for applications which require consistent access to large amounts of data to run its processes.

- Storage of Sensitive Data - Many applications require access to private resources, such as databases, container registries, storage blobs or file systems. When pods are accessing these resources, they are required to provide credentials in order to authenticate themselves. Storing of sensitive date in the container or the configurations is very risky, hence Kubernetes provides a feature called Secrets which creates objects to contain sensitive data for the pods to consume, thus allowing them to gain access to private resources [9].

2.3 Cloud Computing

Cloud computing is the delivery of computing services such as servers, storage, networking, intelligence and more, through the Internet. Cloud computing is able to accelerate innovation, due to its low barrier of entry, flexible resources, and economies of scale. You only pay for the cloud services you use, thus helping you lower operational costs. Cloud computing also helps you to manage your infrastructure more easily and efficiently as your business goes through constant change and growth [11]. Cloud computing provides the following benefits for projects and businesses:

- Cost - With the existence of cloud computing, it eliminates the need for organizations to spend large amounts of capital to purchase hardware and software and set up a datacenter. While the hassle of running a datacenter is handled by the cloud provider, organizations are able to focus on developing their ideas and only paying for cloud services they use [21].
- Global Scale - Cloud resources have the ability to scale on demand, and they are available from multiple geographical locations. As your business grows and your need for resources grows, Cloud computing is able to meet these demands due to its capability to scale.
- Speed - With just a few clicks of a mouse or running a few commands, vast amounts of resources can be provisioned for your business needs. Therefore taking the load off of capacity planning.

2.4 Terraform

Terraform is an infrastructure as code (IaC) tool, meaning you define your Cloud infrastructures using code. This allows you to create, configure, and version your infrastructure more efficiently and reliably. Terraform is able to define low-level components such as individual VMs, storage, networking all the way up to high-level components such as DNS entries, software application features and more [28]. Terraform can also help to manage a wide variety of resources due to its capability of importing different service providers, such as Kubernetes, Helm, Azure etc. [29]

2.5 CI/CD

The CI/CD pipeline is one of the best practices for DevOps teams to push code changes more regularly and reliably. The goal of CI is to create a more consistent and automated way to build, package, and test applications. With a consistent integration process, teams are able to commit code changes more frequently, resulting in better collaboration and code quality [20]. Continuous delivery picks up where continuous integration ends. CD automates the delivery of applications to desired infrastructure environments. Most teams work with multiple non-production environments, such as development and staging environments, and continuous delivery ensures that there is an automatic way to make code changes to them.

2.5.1 Gitlab

GitLab is an online DevOps lifecycle tool that provides a Git-repository manager and CI/CD pipeline features, using an open-source license, developed by GitLab Inc [32]. Gitlab provides an online git repository, container registry, Terraform backend and CI/CD pipelines and more all in one platform.

2.5.2 Gitlab Managed Kubernetes Cluster

Gitlab allows you to integrate your Kubernetes cluster into your Gitlab project. By integrating the cluster, this enables Gitlab to deploy applications to your clusters through Gitlab CI/CD pipelines. To see the steps on how to add an existing Kubernetes Cluster onto Gitlab, see Appendix - A at the bottom of this report.

2.5.3 Gitlab Container Registry

Each GitLab project can have its own storage space for its Docker images. You can view the Container Registry for a project or group:

1. Go to your project or group.
2. Go to Packages and Registries> Container Registry.

Only project members or teams can access the Private Project Container Registry. If the project is publicly available, so is the Container Registry.

2.5.4 Gitlab Managed Terraform State

To keep track of the state of our infrastructure, Terraform uses a state file which maps real world resources, e.g Kubernetes clusters, File stores, Databases, Container Registries etc, to the infrastructure configurations you defined in your Terraform code [30]. Before Terraform makes any operations to the infrastructure, it will first perform a "Refresh" to update the state file to the real infrastructure. After that, it compares the configurations defined in your code to that of the state file. By doing so, Terraform will know what infrastructures to create or destroy so that the real resources match what you defined in your code.

For individuals still learning to use Terraform, the default local backend is recommended, which requires no configuration and stores the Terraform

state in your local computer. However, If you and your team are using Terraform to manage infrastructure, a remote backend with Terraform Cloud or Terraform Enterprise should be used so each member has access to the same backend [24].

GitLab has its own Terraform state backend that can store your Terraform state and spares you from setting up external remote resources like Amazon S3 or Google Cloud Storage. Its features include [17]:

- Remote Terraform plan and apply execution
- Locking and unlocking state.
- Versioning of Terraform state files.
- Supporting encryption of the state file both in transit and at rest.

2.5.5 Jenkins

Jenkins is one of the leading open-source CI/CD tools out there [3]. Unlike Gitlab, which is accessed through an online platform, Jenkins is a self-contained, open source automation server, meaning that it is hosted on a server provisioned by you. With Jenkins you are able to configure and automate many of the manual tasks related to building, testing, and deployment of software.

These tasks can be triggered in multiple different ways, namely:

- Webhooks that gets triggered by a code change in a version control system (e.g Github, Gitlab)
- Scheduling mechanisms set up in Jenkins are able to trigger these tasks to run periodically
- Other tasks completed successfully can trigger the next task in line

3 Designed Solution

This chapter will elaborate on the design of the CI/CD configurations for Gitlab and Jenkins. It will also be explaining the workflow of the CI/CD configurations and how it delivers and deploys software onto production.

3.1 Gitlab CI/CD Pipeline

The Gitlab CI/CD pipeline consists of different stages, every job in each stage must be executed successfully in order for the pipeline to proceed to the next stage. Each stage will perform the following operations:

- Init - Initializes Terraform, verifies Terraform backend is working.
- Validate - Checks Terraform code for errors.
- Plan - Lists out order of tasks to be executed in order to configure infrastructure to the desired state.
- Apply - Executed the tasks planned in the previous stage.
- Build - Builds the ASR application image.
- Deploy Staging - Deploys the ASR application to staging environment.
- Test Staging - Sends audio sample to the staging environment for testing.
- Deploy Production - Deploys the ASR application to production environment.

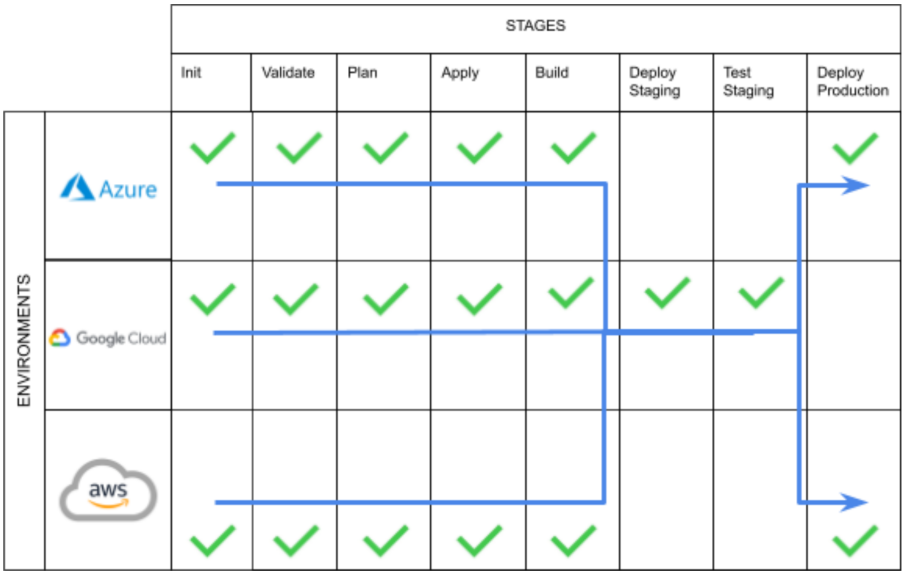

Figure 4: Gitlab CI/CD Pipeline Workflow

3.2 Jenkins CI/CD Pipeline

For the Jenkins CI/CD pipeline, a slightly different approach was used. This implementation groups the Init, Validate, Plan and Apply into one stage called 'Deploy Infrastructure'. Figure 8 shows the Jenkins CI/CD Pipeline workflow which consists of the following jobs:

- Build - Builds the ASR application image and pushes it to our container registry.
- Deploy Infrastructure - Configures our infrastructure based on the Terraform code in our Github repository.
- Deploy Staging - Deploys ASR application to staging environment.
- Test Application - Sends audio sample to the staging environment for testing.
- Deploy Production - Deploys the ASR application to the production environment.

Each job is configured to have access to our Github repository where it can use necessary files from our project to run scripts. The jobs are also configured to have access to required credentials or secret files from the Jenkins Credential Manager. Scripts which modify or push containers to private resources are able to authenticate using the secrets from the Credential Manager.

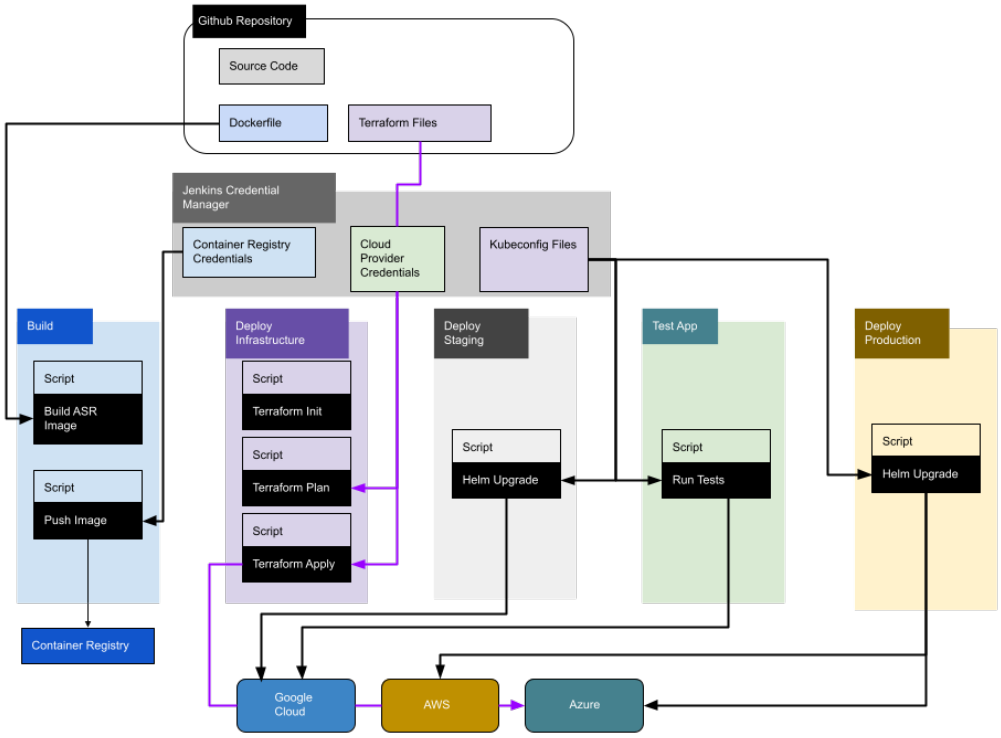

Figure 5: Jenkins CI/CD Pipeline

4 Implementation

This chapter we will be going through how to set up the ASR application infrastructure with the use of Terraform. After that, we will discuss how the ASR models are uploaded to each cloud platform.

Next, we will look at how to deploy the necessary kubernetes resources for our application and lastly, we will discuss how the CI/CD solution is set up using Gitlab and Jenkins.

4.1 Infrastructure Setup

For the infrastructure setup, we will be utilizing Terraform to help setup and manage our infrastructures. By using Terraform, much of the manual setup will be automated for us as it minimises the need for us to log onto the cloud consoles and configure the infrastructure.

Another practice to automate the infrastructure setup process is to write shell scripts which will configure your infrastructure for you all in one go, while that practice is convenient to set up your infrastructure, it can't manage it for you as the scripts only defines the instructions to perform in order to set up the infrastructure, whereas Terraform code defines the infrastructure itself.

4.1.1 Terraform Commands

The following are the four main Terraform commands needed to perform infrastructure configurations using Terraform:

- Terraform init
- Terraform validate
- Terraform plan
- Terraform apply

What Terraform init does is it initializes a working directory for your Terraform configurations [25]. You can use a local backend to store your Terraform configurations, but for our case, we will use a remote backend. Using a remote backend allows teams to collaborate on the infrastructure configurations and also coordinate state changes using lock/unlock files. If a team member is making changes to the infrastructure, Terraform will use lock/unlock files to lock other members from making changes to the infrastructure as it will disrupt any operations which are occurring [31].

Terraform validate will check your Terraform code for syntax errors and configuration to see if it is valid [27]. Terraform plan will "Refresh" the Terraform state file and get the latest updates on the state of the infrastructure from Azure and plan what operations it should execute [26].

Lastly, Terraform apply command will apply all these changes to Azure.

4.1.2 Azure

Before setting up Azure infrastructure, ensure that Azure and Terraform CLI tools are installed in your computer. For Terraform to interact with Azure, we will set azurerm as our provider. By setting azurerm as our provider, Terraform will import all the necessary libraries to manage Azure resources. After setting our provider, we will define the infrastructure we need for our application. For our Automated Speech Recognition application we require the following:

- azurerm_resource_group: A container to hold our project resources
- azurerm_storage_account: A container for storage objects
- azurerm_storage_share: Network File system for our ASR models
- azurerm_kubernetes_cluster: Managed Kubernetes cluster to deploy our application

Next, we will log into our Azure account by running the 'az login' command. If you have multiple subscriptions in your account, set the account to one of those subscriptions. Once logged into Azure through CLI, Terraform will know which Azure account to make infrastructure configurations on and also have the necessary permissions to make these changes. After that, go to the directory where the Terraform files are located and run:

- Terraform init
- Terraform validate
- Terraform plan
- Terraform apply

4.1.3 Google Cloud

Before configuring the infrastructure on Google Cloud, first ensure that gcloud and Terraform CLI tools are installed. After that, log onto your Google cloud console and create a new project, e.g "ntu asr project". For Terraform to interact with our Google Cloud project, we will set google as our provider and the project field as our Google project id. By setting google as our provider, Terraform will import all the necessary libraries to manage Google Cloud resources. After setting our provider, we will define the infrastructure we need for our application. For our Automated Speech Recognition application we require the following:

- google_service_account: A service account for Terraform to make authenticated configurations to Google Cloud
- google_container_cluster: Google Kubernetes Engine (GKE)
- google_container_node_pool: A group of nodes within the Google container cluster
- google_filestore_instance: Network File system for our ASR models

Next, we will log into our Google Cloud account by running 'gcloud auth application-default login' Once logged into Google Cloud through CLI, Terraform will have the necessary authentications to make infrastructure configurations. After that, go to the directory where the Terraform files are located and run:

- Terraform init
- Terraform validate
- Terraform plan
- Terraform apply

4.1.4 AWS

To set up the infrastructure for AWS, first ensure that aws CLI, Terraform CLI and eksctl tools are installed. After that run 'aws configure' command and enter your AWS Access Key ID and AWS Secret Access Key so you're authenticated and are able to interact with AWS. Once aws is configured, Terraform will be able to set up our infrastructure for us. The configuration for AWS is much more complex as compared to Azure and Google due to the nature of how AWS provides its services. To simplify the Terraform configuration, we would be using VPC and EKS modules to set up the infrastructure. With the help of those modules we won't have to figure out the security rules, IAM policies, IAM roles and networkings needed for our VPC and EKS to work. Also our Terraform code is cleaner and easier to read. For our ASR application to work on AWS, we require the following resources:

- VPC (Virtual Private Cloud): A container for our application related resources
- EKS (Elastic Kubernetes Service): A Kubernetes Cluster
- EFS (Elastic File System): File system to store our ASR models

Now we have defined all the infrastructures needed for our application, head to the directory where the Terraform files are located and run the following commands:

- Terraform init
- Terraform validate
- Terraform plan
- Terraform apply

4.2 Uploading Models

File stores are network file systems which can store data for our cloud applications. In our case, we are using it to store ASR models so our containers can access these models. Each cloud provider will require its own file store for the ASR models as they are unable to access each other's. This is due to compatibility issues between the cloud providers.

4.2.1 Azure File Share

To upload the ASR models onto Azure file share you can log onto the Azure console to upload the files for the ASR models individually until all the files have been uploaded. A more convenient way to upload is by using Azure CLI to upload the files. Using the 'az storage file upload-batch' command, we are able to upload all the files in the ASR models in one batch. This command requires the following arguments for it to work:

- destination: The name of the directory in the file store to upload the files
- source: The directory where the files are located in your local computer
- account-key: Azure file share storage key
- account-name: Account name of the storage account

4.2.2 Google File Store

Uploading the ASR models onto Google File Store requires a more complicated approach as compared to Azure as Google Cloud does not provide a command to upload files directly onto the file store. They also don't provide a way to upload files through the Google cloud console. One of the methods to upload to Google cloud File store is to first ssh into any one of the VMs in our Google node pool. Once connected to

the VM, we will mount the File system onto our VM by running 'sudo mount <ip-address>:/<file-share> <mount-point-directory>' Afterwards, we will upload the models onto the fileshare through the VM from our local computer by running 'gcloud compute scp <local-data-path> <client-name>:<mount-directory> –project=<project-id> –zone=<zone>'

4.2.3 AWS Elastic File System

Uploading the models onto AWS EFS is similar as compared to Google's File store. The first step is to create a Key pair. This Key pair is required in order to SSH into a VM. Next, create a subnet for a mount target in the EFS. Once created, launch an EC2 instance in that subnet and add the File system into the EC2 configuration. Remember to edit the route table settings for the subnet so that it is connected to the public internet gateway. Otherwise, your SSH request will not be able to reach the EC2 instance. After launching the EC2 instance, wait for it to pass all the status checks and connect to it. After we are connected, we are able to transfer our ASR models onto the EFS through our EC2 instance using SCP.

4.3 Kubernetes Resource Setup

The last step before we can deploy the application onto our Kubernetes clusters is to set up our application-related Kubernetes resources.

4.3.1 Container Registry

The Container Registry is where the Docker image for our application will be stored [13]. When pods are running in our Kubernetes cluster, they will pull the container image from the Container Registry. Since we are deploying our application onto multiple cloud platforms, we will use a centralised container registry. Gitlab provides a container registry along with a git repository for our project. The container image can be built

locally using a Dockerfile and pushed onto the container registry on Gitlab where it will be accessed by pods from different Kubernetes platforms.

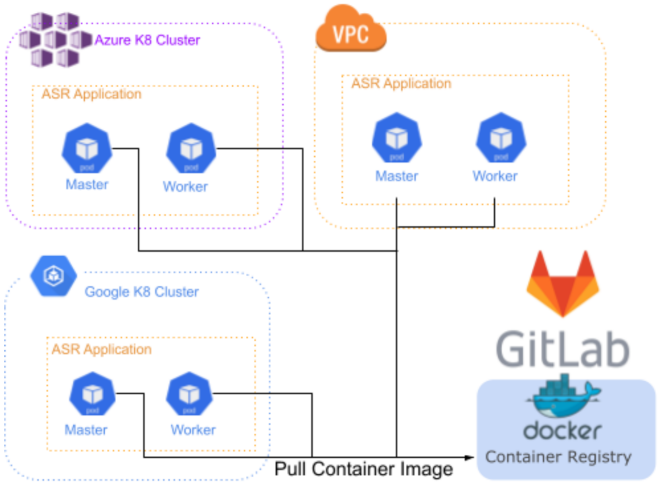

Figure 6: Central Container Registry

4.3.2 Persistent Volumes

Persistent Volumes is a storage resource in a cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. Persistent Volumes captures the implementation details of cloud provider storage systems and provides it as a volume to the cluster just like VMs are provided as nodes to the cluster as a compute resource [7]. Since the implementation details of each cloud provider storage system is different, the pv.yaml file which is used to describe the implementation will be different when provisioning PVs for each cloud provider.

4.3.2.1 Azure For our cluster to access Azure's File share, our PV.yaml file requires 2 important details:

- share name: Name of the file share in Azure's Storage Account
- secret name: Name of Kubernetes Secrets which contains Storage Account Name and Storage Account Key

4.3.2.2 Google Cloud For our cluster on Google Cloud to access Google's Network File System, requires 3 important details:

- storage capacity: Size of the file share on the Google File Store instance
- file share path: Name of the file share on the Google File Store instance
- server IP address: IP address of the Google File Store instance

4.3.2.3 AWS For our EKS cluster, our Persistent Volumes will be dynamically provisioned using Storage Classes instead. For our cluster to be able to find the storage resources, we just have to provide the following detail:

- file system ID: The ID of the EFS (e.g fs-xxxxxxxx)

4.3.3 Persistent Volume Claim

Once Persistent Volumes is set up, Persistent Volume Claims are used to request for storage resources from Persistent Volumes. This is very similar to how a Pod will request and consume resources (CPU and Memory) from nodes in the Kubernetes cluster. While Pods are able to request for a certain amount of CPU and Memory resources from nodes, Claims can request for specific capacities and access modes such as ReadWriteOnce, ReadOnlyMany or ReadWriteMany [8]. For our ASR application, we will require ReadOnlyMany access mode as the application only needs to access the language models, but not modify it.

4.3.4 Kubernetes Secrets

A Secret is a kubernetes resource that contains sensitive data (e.g passwords, tokens, keys). Otherwise, such information would be placed into the Pod specification or in a container image. But using a Secret means that

you don't need to include sensitive information in your application code or Kubernetes specifications. Since secrets can be created independently of the Pods that use them, the risk of disclosing the secret (and its data) during the workflow of creating, viewing, and editing Pods is reduced [10]. For our ASR application, the worker and master pods will be pulling a docker image from a private container registry, hence, they will require certain credentials (Password, Username, Email, etc) in order to be authenticated to the container registry. These credentials will be provided to these Pods through Secrets.

4.4 Deploy Application

To deploy our application, we will be using Helm which is a tool used for managing even the most complicated Kubernetes applications [4]. By running 'helm install sgdecoding-online-scaled' where sgdecoding-online-scaled is the name of our ASR application, the master and worker pods should be deployed into our Kubernetes clusters. If you were to check the workload of the clusters, you should see a certain number of master and worker pods, depending on the number of replicas specified in your configurations.

4.5 Gilab CI/CD Setup

4.5.1 Kubernetes Integration

To set up the CI/CD for Gitlab, we first integrate our existing Kubernetes Clusters which had been set up using Terraform on our previous sections. To see how to add Kubernetes Clusters onto Gitlab, see Appendix - A for details. Once all the Clusters have been added, you should see this on your projects Infrastructure > Kubernetes clusters page:

The page will display the number of Nodes, CPUs and Total memory available in those clusters. If there are no Nodes available, it will display

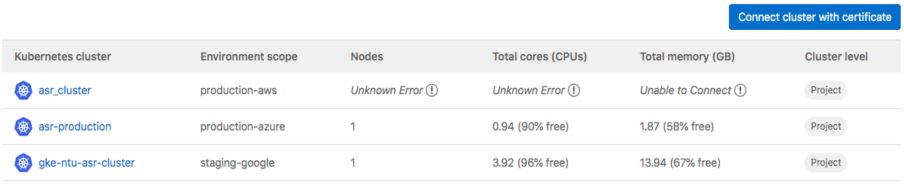

Figure 7: Gitlab Integrated K8 Clusters

Unknown Error. For our CI/CD environment configuration, we will be using Azure and AWS as production environments and Google Cloud as our staging/testing environment. Having just one cloud provider as the staging environment is able to test for any application level bugs, but will not test for bugs in the infrastructure except for the infrastructure used for staging (Google cloud for this case). To achieve full coverage for infrastructure issues, it is recommended to have a staging environment in each cloud provider, though this will be more expensive.

4.5.2 Terraform Authentication

The CI/CD scripts are run remotely in a docker container, so we need to find a way to automate the login process for each cloud provider, because commands such as 'az login' will not work in the docker container as they require user interactions. Logging into the cloud provider will allow Terraform to acquire the necessary authentications and permissions to configure our infrastructure, and since our CI/CD scripts utilize Terraform to manage our infrastructure, this step is extremely important. To see more details on the process of automating the cloud authentication go to Appendix - C to E.

4.5.3 Gitlab CI/CD Pipeline

The CI/CD pipeline is a sequence of steps needed to deliver a new version of a software [18]. With the help of Gitlabs CI/CD pipeline, we are able to automate these steps to ensure that the delivery process is consistent throughout the team, and the team is able to focus more time on developing the application. In the next few section, we will be discussing the various features of the CI/CD pipeline built for the ASR project and how it automates any changes made at the infrastructure or the application level. A .gitlab-ci.yaml file is used to build and configure the pipeline, see Appendix - B for more details on how the .gitlab-ci.yaml file works.

4.5.4 Gitlab Pipeline

Our pipeline is made of multiple stages and for each stage there are certain jobs which need to be completed. To progress to the next stage, all the jobs must pass. For example, on the plan stage, 'terraform plan' command is executed, and if the command returns no errors, the pipeline will mark this stage as pass and continue on to the next stage. If any stage fails, the pipeline will stop running and Gitlab will notify the developer that the pipeline has failed (via email). Some stages have identical jobs running in different environments. If there are enough Gitlab runners, these jobs will be executed in parallel.

Figure 8: Gitlab Pipeline

Our setup uses Google cloud as the staging environment, where we test the application before deploying it onto our production environments (Azure & AWS). The pipeline first starts with Init, Validate, Plan and Apply stages, which will check and apply any changes made to the infrastructure. Build stage builds the docker image for our application and pushes the new image onto our container registry. Next, the Deploy Staging stage deploys the application onto the staging environment. Once complete, the Test Staging stage will test the application by sending an audio sample for the application to transcribe. Below is a screenshot of the test logs taken from this stage.

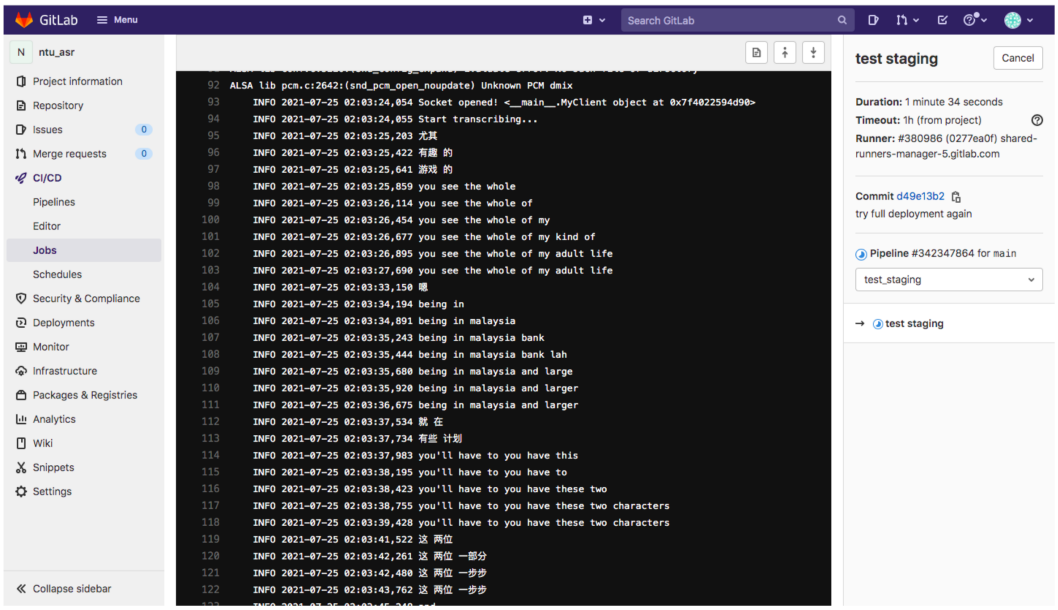

Figure 9: Gitlab Test Log

After the tests have passed, the application is ready to be deployed to production. The last stage, Deploy Production, will deploy the application to our production environments.

4.6 Jenkins CI/CD Setup

4.6.1 Jenkins Server

For Jenkins to run your CI/CD jobs it requires computing resources to run them. This resource can come in many forms, such as a container running in a Kubernetes cluster, a VM running on a cloud platform, a small server you own or even a raspberry pi [15]. The resource you choose to run the Jenkins server on depends on your requirements. For example, if the project is on a large scale with a hundred team members with thousands of code commits per day, then it is recommended to run an autoscaling group of Jenkins containers on a Kubernetes cluster. For our project, as it is still new and in its early stages of development at the time of this writing, we will be using a small VM running on Azure to host our Jenkins server. This Jenkins server serves as an interface for developers to configure CI/CD jobs to automate their software delivery tasks such as building, testing and deployments.

4.6.2 Setting Up Jenkins VM

In order for our VM to run CI/CD jobs for us, we first need to provide it with the necessary tools and software to build, test and deploy our project. To do so, we must first ssh [22] into the VM. From here, we are able to use the command line interface to install the following:

- Azure CLI
- Dockers
- Helm
- Terraform
- Kubernetes
- Docker-compose

4.6.3 Managing Secret Credentials

Jenkins provides an easy to use secret management service which allows you to upload sensitive data such as keys, credentials, passwords or files to be accessed by your CI/CD jobs. Each job is able to access these secret resources through environment variables. For this project we need to upload the following secrets onto our Jenkins Server:

- Container Registry Credentials
- Github Credentials (Required if project registry is private)
- Azure Kubeconfig File
- AWS Kubeconfig File
- Google Cloud Kubeconfig File
- ARM_CLIENT_ID
- ARM_CLIENT_SECRET
- ARM_SUBSCRIPTION_ID
- ARM_TENANT_ID
- Google Service Account Credentials File
- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY

Cloud Provider Credentials are used by Terraform scripts to authenticate themselves to each cloud provider, thus giving the scripts necessary permissions to make infrastructure configurations. Kubeconfig files are used by the Deploy Application job to locate and authenticate with the different kubernetes clusters running our applications so that we can deploy updated versions of our ASR application to them.

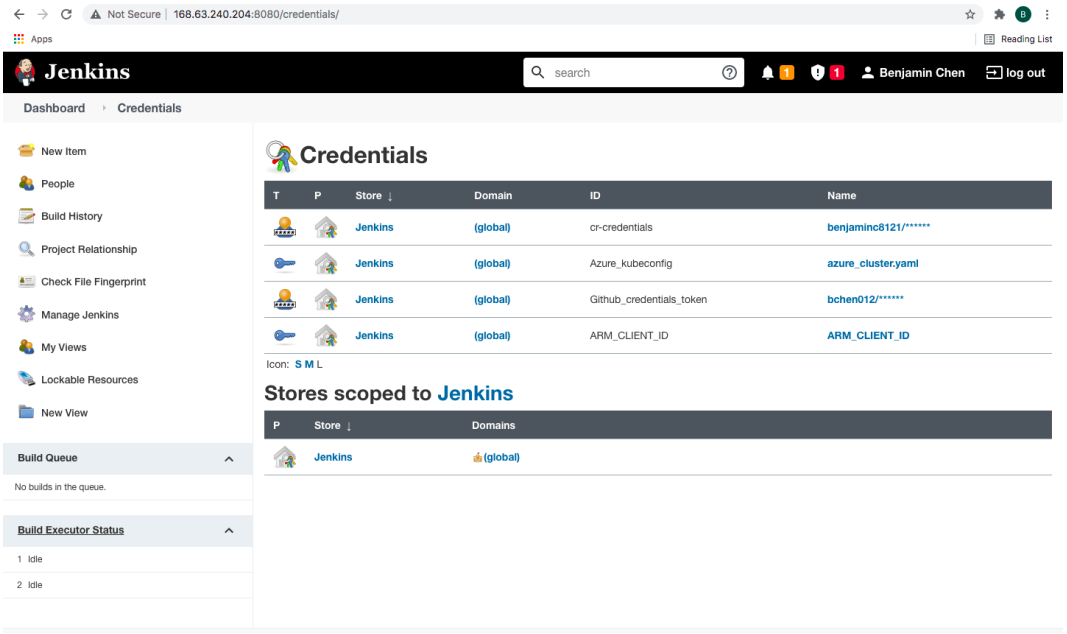

Figure 10: Jenkins Managed Credentials

4.6.4 Jenkins Pipeline

The pipeline is made of 5 different stages. The first stage, which is the Build stage, is triggered using a Github hook. Whenever a push is made to the project repository on Github, the pipeline will start to execute. As soon as the Build stage succeeds, it will trigger the next one and so on. If any of the stages fail, it will block the pipeline from executing any further, thus preventing any bugs from ending up in production. The fourth job

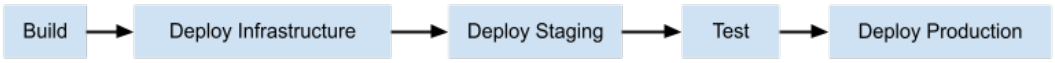

Figure 11: Jenkins Pipeline

will test the application by sending a sample audio to the one of the clusters (Depends on which provider is your Staging Cluster in) and the results of the transcribed audio will be send back to the Jenkins VM. They can then be viewed in the Jenkins Job logs.

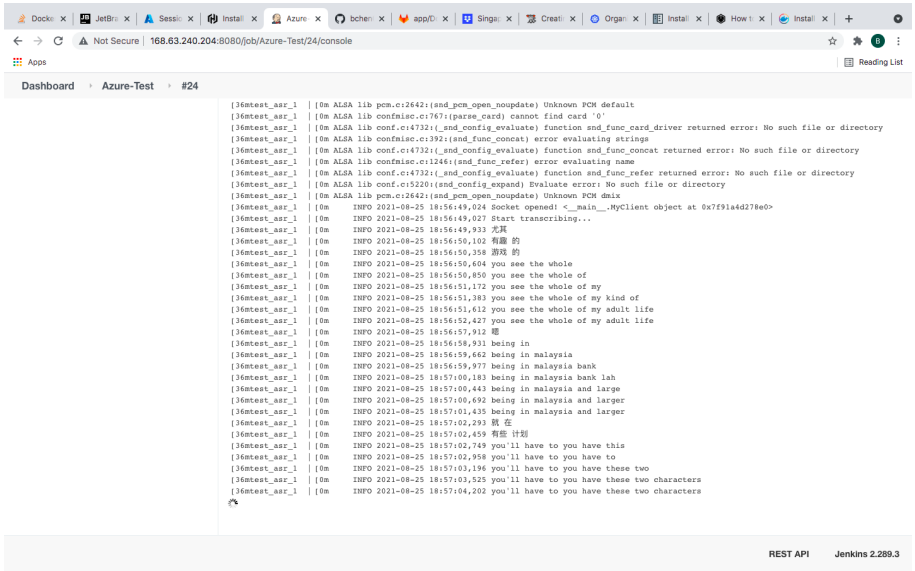

Figure 12: Jenkins Test Logs

5 Conclusion and Future Improvements

5.1 Gitlab vs Jenkins

Gitlab	Jenkins
Steep learning curve	Easy to pick up
Easy to set up	Complicated to set up
Uses code to configure Pipelines	Uses web page to configure Pipelines
Flexible configurations	Standard configurations
Provides Git Repository on Gitlab	External Git Repository on Github
Provides Terraform Backend	External Terraform Backend
Provides Container Registry	External Container Registry
Uses HashiCorp Vault to store secrets	Provides an easy way to store secrets
Free 400 minutes per month of CI/CD minutes to run jobs	Host your own resource to run jobs

After going through this entire journey of learning Gitlab CI/CD followed by Jenkins CI/CD and implementing a CI/CD solution to the ASR project using both platforms, the table above compares my experiences and analysis of the two platforms.

While Gitlab was very difficult to pick up, I would say that it is worth the trouble due to the vast amount of support the Gitlab platform provides for CI/CD operations. Once the user gets the hang of using the .gitlab-ci.yaml to configure the CI/CD pipelines, it can be a very powerful tool to set up CI jobs due to its vast amount of capabilities to configure finer details of the pipeline. On the other hand Jenkins is a much easier tool to pick up due to its user friendly front-end which helps in assisting users to configure their CI/CD pipelines.

5.2 Future Improvements

5.2.1 Version Control

Currently, the versioning of the ASR container images are done manually by modifying the version number in the CI/CD scripts. This may cause some mix-ups in the version number when teams are working on the project at the same time. A versioning system that can automatically bump up the version and help also teams stay aligned on version number could be a possible improvement to this project.

5.2.2 HashiCorp Vault Integration to Gitlab

Currently the cloud provider credentials for Gitlab are stored in Gitlab settings as protected variables. A more secure way will be to use HashiCorp Vault [16] to store our project Secrets.

5.3 Conclusion

The objective of this project is to develop a CI/CD pipeline which is able to safely and reliably deliver software updates of the ASR application to multiple compute clusters in various cloud platforms using 2 popular CI/CD tools (Gitlab and Jenkins) and compare the utility of the 2 tools.

During this project, CI/CD pipelines were built using both Gitlab and Jenkins. The pipelines are triggered to run whenever a developer pushes new code changes to the remote git repository. The pipelines will automatically build a new version of the ASR application and push the image to our container registry upon the trigger.

Terraform has been used to manage the infrastructure needed for our ASR application. Terraform code has been written for this project and is stored in the projects git repository. The code defines the infrastructure configurations for our ASR application in 3 different cloud platforms (Azure, Google Cloud and AWS). The CI/CD pipelines will automatically configure the real world infrastructure, which hosts the ASR application, to match the configurations defined in our Terraform code whenever a developer modifies the Terraform code in our git repository.

The pipelines will automatically deploy the new version of our ASR application to all the kubernetes clusters once it has successfully built the ASR application, pushed it to the container registry and ensured that the infrastructure configurations correct and stable.

Lastly, the pipelines will send a audio sample to one of the kubernetes clusters for testing.

References

[1] Daniel Povey et al. *The Kaldi Speech Recognition Toolkit*. 2011.

[2] ambassador. *CI/CD and progressive delivery - Safely shipping code*. 2021. URL: https://www.getambassador.io/docs/argo/latest/concepts/cicd/.

[3] Creative Commons Attribution-ShareAlike. *Jenkins*. URL: https://www.jenkins.io.

[4] Helm Authors. *The package manager for Kubernetes*. 2021. URL: https://helm.sh.

[5] The Kubernetes Authors. *kube-scheduler*. 2021. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/.

[6] The Kubernetes Authors. *Kubernetes Documentation*. 2020. URL: https://kubernetes.io/docs/home/.

[7] The Kubernetes Authors. *Persistent Volumes*. 2021. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[8] The Kubernetes Authors. *Persistent Volumes*. 2021. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[9] The Kubernetes Authors. *Secrets*. 2021. URL: https://kubernetes.io/docs/concepts/configuration/secret/.

[10] The Kubernetes Authors. *Secrets*. 2021. URL: https://kubernetes.io/docs/concepts/configuration/secret/.

[11] Microsoft Azure. *What is cloud computing?* URL: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/#benefits.

[12] Roderick Bauer. *What's the Diff: VMs vs Containers*. 2018. URL: https://www.backblaze.com/blog/vm-vs-containers/.

[13] Google Cloud. *Container Registry*. URL: https://cloud.google.com/container-registry.

[14] IBM Cloud Education. *What is Docker?* 2021. URL: https://www.ibm.com/cloud/learn/docker.

[15] RASPBERRY PI FOUNDATION. *Products*. 2021. URL: https://www.raspberrypi.org/products/.

[16] GitLab. *Authenticating and reading secrets with HashiCorp Vault*. URL: https://docs.gitlab.com/ee/ci/examples/authenticating-with-hashicorp-vault/.

[17] GitLab. *GitLab managed Terraform State*. URL: https://docs.gitlab.com/ee/user/infrastructure/terraform_state.html.

[18] Redhat Inc. *What is a CI/CD pipeline?* 2019. URL: https://www.redhat.com/en/topics/devops/what-cicd-pipeline.

[19] Doug Jones. *Containers vs. Virtual Machines (VMs): What's the Difference?* 2018. URL: https://www.netapp.com/blog/containers-vs-vms/.

[20] Isaac Sacolick. *What is CI/CD? Continuous integration and continuous delivery explained*. URL: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-conti%20nuous-delivery-explained.html.

[21] inc. Salesforce.com. *12 Benefits of Cloud Computing*. 2021. URL: https://www.salesforce.com/ap/products/platform/best-practices/benefits-of-cloud-computing/.

[22] SSH.COM. *SSH Protocol – Secure Remote Login and File Transfer*. 2021. URL: https://www.ssh.com/academy/ssh/protocol.

[23] Cprime Studios. *Top CI/CD Tools in 2021: The Most Complete Guide with 33 Best Picks for DevOps*. 2021. URL: https://cprimestudios.com/blog/top-cicd-tools-2021-most-complete-guide-33-best-picks-devops.

[24] Hashicorp Terraform. *Backends*. URL: https://www.terraform.io/docs/language/settings/backends/index.html.

[25] Hashicorp Terraform. *Command: init*. URL: https://www.terraform.io/docs/cli/commands/init.html.

[26] Hashicorp Terraform. *Command: plan*. URL: https://www.terraform.io/docs/cli/commands/plan.html.

[27] Hashicorp Terraform. *Command: validate*. URL: https://www.terraform.io/docs/cli/commands/validate.html.

[28] Hashicorp Terraform. *Introduction to Terraform*. URL: https://www.terraform.io/intro/index.html.

[29] Hashicorp Terraform. *Providers*. URL: https://www.terraform.io/docs/language/providers/index.html.

[30] Hashicorp Terraform. *State*. URL: https://www.terraform.io/docs/language/state/index.html.

[31] Hashicorp Terraform. *State Locking*. URL: https://www.terraform.io/docs/language/state/locking.html.

[32] Wikipedia. *GitLab*. URL: https://en.wikipedia.org/wiki/GitLab.

Appendices

A Adding Existing Kubernetes Cluster

To add a Kubernetes cluster to your project:

1. Go to project's Infrastructure > Kubernetes clusters page.
2. Click Add Kubernetes cluster.
3. Click on Add existing cluster tab and fill the following details:
    a. Kubernetes cluster name (required) - Name of Kubernetes cluster to be displayed on Gitlab.
    b. Environment scope (required) - The associated environment of this cluster.
    c. API URL (required) - Base URL of your existing kubernetes cluster. Kubernetes exposes several APIs, we only want the "base" URL that is common to all of them. To get the API URL, run this command after connecting to your Kubernetes cluster:

    ```
    kubectl cluster-info | grep -E 'Kubernetes master|
    Kubernetes control plane'| awk '/http/ {print $NF}'
    ```

    d. CA certificate (required) - A Kubernetes certificate is required to authenticate to the cluster. Kubernetes creates a default CA certificate and we will be using that.
    e. Token - GitLab authenticates with the use of a service token with cluster-admin privileges.
    f. GitLab-managed cluster - Checking this will allow GitLab to manage the cluster's service accounts and namespaces.

g. Project namespace prefix (optional) - Leaving it blank, GitLab creates one for you. Filling up this field will cause apps to be deployed to <namespace prefix>-<environment> namespace in your cluster.

h. Finally, click the Create Kubernetes cluster button.

B Building CI/CD pipeline with .gitlab-ci.yml

The .gitlab-ci.yml file is a YAML file where you configure instructions for the GitLab CI/CD pipeline. In this file, you will define the structure and order of the jobs to be performed by the runner and decisions a runner must make when certain conditions are met. To set up the file, create a YAML file with the name .gitlab-ci.yml and store it in the root directory of your project. In this YAML file, you can define the structure of your pipeline by configuring stages. This example shows a simple set of pipeline stages:

```
stages:
  - build
  - test
  - deploy
```

This pipeline has 3 stages, build, test and deploy. Each stage must succeed in order to progress to the next stage. So if the build stage succeeds, but the test stage fails, the pipeline will stop running, and Gitlab will indicate that the pipeline has failed. Once the stages have been defined, each stage can have 1 or more jobs. The example below shows 4 different jobs. The build stage has one job named “build-job”, while the test stage has 2 jobs named “test-job1” and “test-job2”. Lastly the deploy stage has 1 job named “deploy-prod”.

```
build-job:
```

```
  stage: build
  script:
    - echo "Welcome, $GITLAB_USER_LOGIN!"
test-job1:
stage: test
  script:
    - echo "I will be testing something"
test-job2:
  stage: test
  script:
    - echo "I will also be testing something
      but I will take longer"
    - echo "Let me sleep for 20 seconds first"
    - sleep 20
deploy-prod:
  stage: deploy
  script:
    - echo "Now I'm ready to deploy once
      all tests have passed"
```

When code is committed to the Gitlab repository, the pipeline will be triggered and the following image can be seen on the Gitlab console at your projects CI/CD > Pipelines page. Multiple jobs can be run in parallel if they belong to the same stage, for example, test-job1 and test-job2 are both from the test stage, and the Gitlab runners will run them in parallel.

Gitlab runners are resources that run your CI/CD job. Gitlab provides its own set of runners for you to use. The first 2000 minutes of run-time are free, after that you have to pay for more runner minutes. You can also install Gitlab runner into your Kubernetes clusters integrated into the project or install Gitlab runner locally on your computer. To use the runner in your Kubernetes Cluster or on your computer, go to Settings >

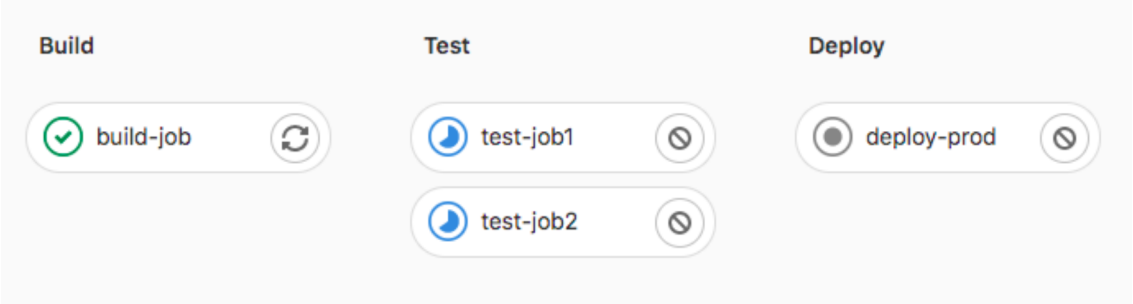

Figure 13: Pipeline Jobs

CI/CD > Runners > Expand and ensure that the runners installed have been enabled. Notice that on this page, each runner has a set of tags, the tags are for you to specify which runner to use when executing a job. In this example, the build-job will be executed by runner #157328.

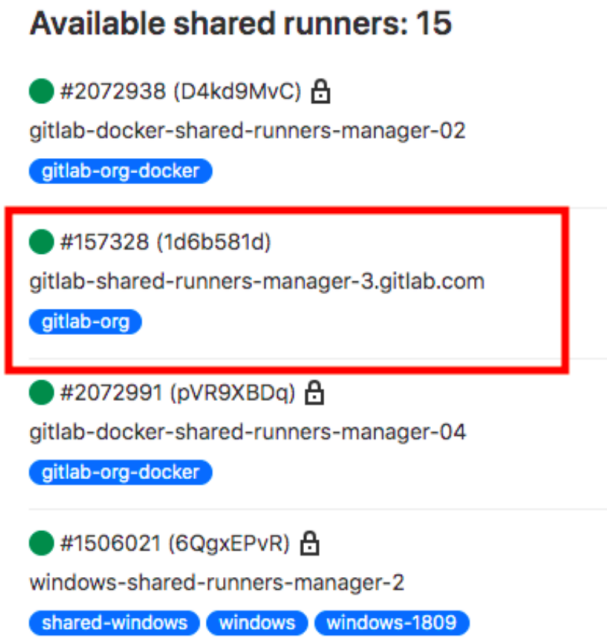

Figure 14: Gitlab Runners

Clicking on one of the jobs will allow you to view the logs from the jobs. Below is an example from the build job we created in the earlier example.

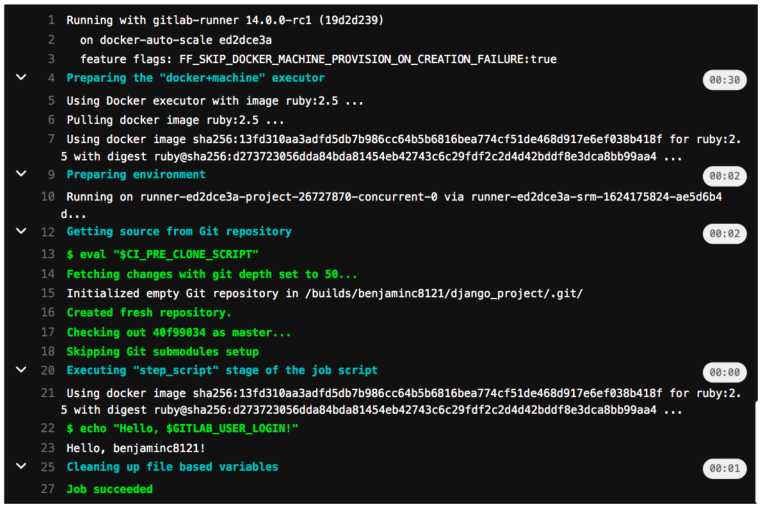

Figure 15: Gitlab Job Log

C Automating Azure Authentication (Gitlab)

To automate our Azure login, we first create a service principal with the following azure CLI command:

```
az ad sp create-for-rbac --name TerraformServicePrincipal
```

This will produce an output like the following:

```
{
  "appId": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "displayName": "TerraformServicePrincipal",
  "name": "http://TerraformServicePrincipal",
  "password": "xxxxxxxxxxxxxxxxxxxxxxxxx",
  "tenant": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx"
}
```

These outputs will be saved as environment variables for our CI/CD script and they will be used by Terraform to authenticate with our Azure account in order to make configurations to our Azure infrastructure. To save them as environment variables securely, you can use Vault by Hashicorp or if you are using Gitlab, go to Settings > CI/CD > Variables > Expand on your projects page and save them here as protected variables: The following

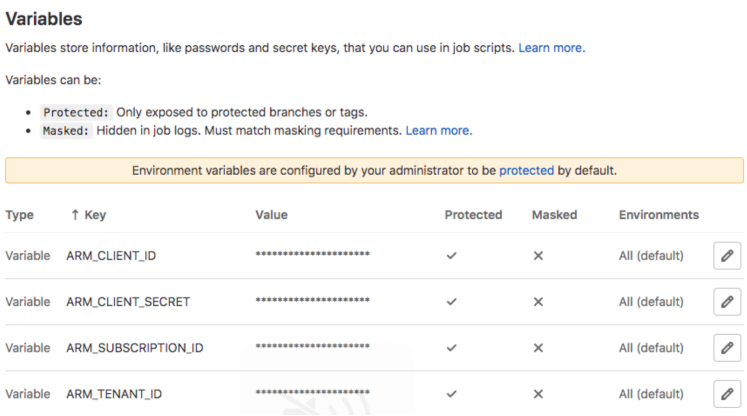

Figure 16: Protected Azure Credentials on Gitlab

keys will correspond to the following values from your output produced earlier:

- ARM_CLIENT_ID: appID
- ARM_CLIENT_SECRET: password
- ARM_SUBSCRIPTION_ID: azure subscription ID (Can be found on Azure console)
- ARM_TENANT_ID: tenant

D Automating Google Cloud Authentication (Gitlab)

To automate our Google Cloud login, we first go to IAM & Admin > Service Accounts > terraform-sa (created using terraform) > Keys on our Google Cloud console and create a new JSON type key. Once created, a file containing the service account credentials will automatically be downloaded. Next we will create a new role called Terraform_role with the following command:

```
gcloud iam roles create Terraform_role \
--file=Terraform_role.yaml \
--project $PROJECT_ID
```

The Terraform_role.yaml file contains the permissions which will be added to the role. Below is a list of all the permissions required for our application:

- compute.instanceGroupManagers.get
- container.clusters.create
- container.clusters.delete
- container.clusters.get

- container.clusters.list
- container.clusters.update
- file.instances.create
- file.instances.delete
- file.instances.get
- file.instances.list
- iam.serviceAccounts.get
- container.operations.get

Once Terraform_role has been created we will bind this role to our service account (terraform-sa) created earlier, thus giving it all the necessary permissions to configure our infrastructure. This can be achieved with the following command:

```
gcloud projects add-iam-policy-binding ntu-asr-317615 \
    --member=serviceAccount:sa@xxxx.iam.gserviceaccount.com \
    --role=projects/<project ID>/roles/Terraform_role
```

Where member is the email of our service account which can be found on the "client_email" field in the service account credentials file downloaded earlier. Lastly, we just need to provide the path to our service account credentials file as an environment variable for our CI/CD scripts. For example:

```
export GOOGLE_APPLICATION_CREDENTIALS=ntu-asr-35440.json
```

E Automating AWS Authentication (Gitlab)

To automate our AWS login, first create access keys for an IAM user. Upon creation you should see the following:

- Access key ID: XXXXXXXXEXAMPLE
- Secret access key: XXXXXX/XXXXXX/XXXXXEXAMPLEKEY

Next, we just have to save these credentials as environment variables for our CI/CD script and they will be used by Terraform to authenticate with our AWS account in order to make configurations to our AWS infrastructure. To save them as environment variables securely, you can use Vault by Hashicorp or if you are using Gitlab, go to Settings > CI/CD > Variables > Expand on your projects page and save them here as protected variables:

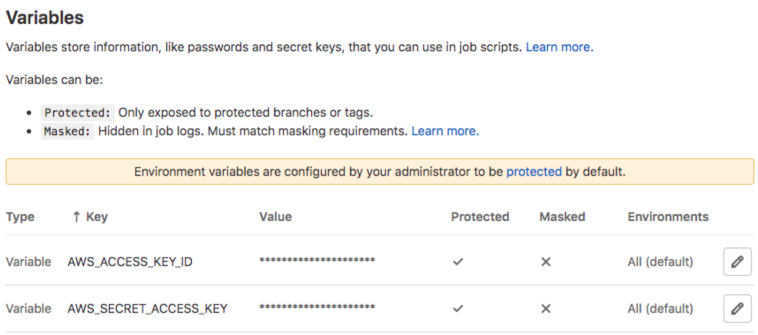

Figure 17: Protected AWS Credentials on Gitlab