NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

FINAL YEAR PROJECT

PROJECT TITLE:

Deploying Speech Recognition System using High Availability and Scalability Kubernetes Cluster with Kubernetes and Docker

PROJECT ID: SCSE19-0003

FINAL REPORT

Done by: WONG SENG WEE
MATRICULATION NUMBER: U1620607H

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

Table of Contents

List of Figures

Abstract

This project aims to improve the scalability of the existing speech recognition system such that it can support dynamic increase in workload (phone calls) requesting for its service. This project aims to use containerisation technology such as *Docker* [1] and container orchestration tool such as *Kubernetes* [2] to orchestrate the use of resources in the cloud server to process the incoming speech-to-text decoding requests efficiently and effectively.

The report will present the proposed architecture to increase the availability and scalability of the speech recognition system and detail its characteristics. In addition, the report will also discuss the use of a dashboard to present the metrics monitoring the health of the Kubernetes cluster supporting the speech recognition system. The cost and performance of the proposed system architecture will be evaluated against the objectives of this project.

Acknowledgements

This project was made possible due to the support and guidance provided by the following groups of people. The author would like to express his gratitude and appreciation to the following people,

- Associate Professor Chng Eng Siong whom is the project supervisor for his time and support. He provided valuable advice on the specifications of the Kubernetes cluster.
- Research Associate Ly Thi Vu, for her help in liaison with the relevant personnel at the AISG as well as her help in procuring the required resources to make this project possible.
- Teoh Cheewei from AISG whom provided useful insights to the current deployment of the speech recognition system. He aided in the initial understanding of the entire system from a greater perspective.
- Last but not least, friends and family who provided support throughout the project.

Chapter 1: Introduction

Docker [1] is a very popular containerization tool which allows an application to be packaged into a container easily. The application container can then be deployed on a production server to serve its users rapidly and effectively with little hassle. Docker allows an application to be ported from a development environment to a production environment easily. With the same Docker image, the application may be deployed on multiple virtual machines to serve a large user base, and thus making it very scalable.

On the other hand, Kubernetes [2] is a container orchestration tool which is used to scale, run and monitor an application. As an application grows to serve more users, the server needs to manage its resources well to handle more incoming requests. Kubernetes acts as an intelligent ‘agent’ to manage the server resources i.e. number of virtual machines. It monitors the statuses of containers operating on the virtual machines and thus optimizing the use of the virtual machines on the server. Docker and Kubernetes work hand in hand to enable scalable deployment of application.

1.1 Background Information

The speech recognition system takes in input in the form of an audio file or a live audio stream. The system predicts the dialog output based on the character sequences from the input source. It makes use of the open-source Kaldi [3] toolkit based on finite-state transducers to carry out speech recognition. A user may pass in an audio file or open a live audio stream to the server and obtain a transcribed output of the audio data.

A dedicated team of staff is working on producing speech recognition models that would be able to decode audio of various language mediums e.g. Chinese, English etc. These models would then be loaded into the speech recognition system to complete the decoding requests from the users. The speech recognition system aims to support a wide variety of uses cases where the system can transcribe caller's content in real time.

1.2 Importance of Project

While the speech recognition system is fully functional and can transcribe audio files with a high accuracy, it is limited in the number of concurrent users using the system. The speech recognition application deployed on the server can only serve **one** user at a time. The system does not have the ability to scale according to the number of user requests. Thus, the use of server resources is inefficient and does not maximise the bandwidth of the server. In most commercial use cases, multiple users are expected to access the speech recognition system at the same time. Users may have to wait for the server to be available when another user is using the speech recognition system. Hence, this project is important as it allows the speech recognition system to scale dynamically and be able to serve multiple users concurrently.

Previous work done by another FYP [1] student, Heng Weiliang was successful in ensuring that the system can be accessed by multiple users concurrently however, the maximum number of concurrent users is limited to the number of static workers preassigned within the Kubernetes cluster. It is inefficient as the number of static workers in the Kubernetes cluster does not scale according to the number of incoming user requests.

[1] Abbreviation for Final Year Project

1.3 Scope of Project

The scope of this project includes devising a reliable system architecture to deploy the speech recognition application such that it can serve multiple users concurrently and dynamically scale according to the workload of the server. The system architecture will include the details of how the master pods manage the worker pods within the Kubernetes cluster to create a system capable of reusing the server resources efficiently. This project **will not** describe the details of the speech recognition models or the toolkit used by the system. The objectives of the project are to deploy the existing speech recognition system with a **robust system architecture** which can support dynamic increase in workload requesting for its service. The system should also be able to accept multiple user requests at the same time.

Chapter 2: Design and Implementation

The design of the system architecture is deliberated with the features of the Kubernetes [2] technology in mind. Kubernetes is used in conjunction with Docker which supplies the speech recognition application in the form of a container image.

2.1 Concepts/Technologies used within the System Architecture

2.1.1 Docker

Docker is a containerisation tool which can be used to package an application into a container image which can then be turned into a container at run time. A docker image is a lightweight, standalone piece of software where applications can be ported over from different development environments easily. [1]

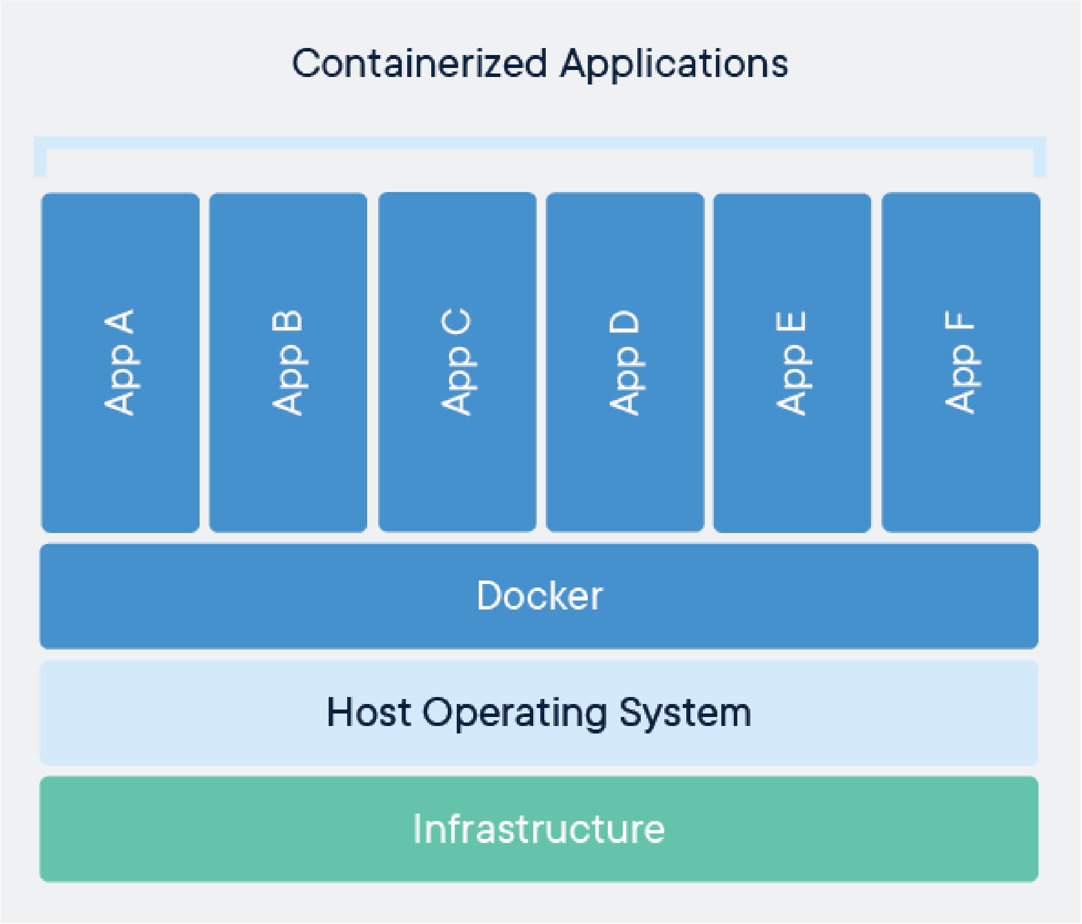


Figure 1 Docker architecture [1]

Docker creates the standard for all operating systems running the Docker engine to run the container image smoothly. The Docker technology packages the speech recognition source code into a container image where it can be loaded into the Kubernetes infrastructure to be deployed on the production server.

2.1.2 Kubernetes

Kubernetes is a container orchestration tool for managing containerized workloads and services which allows for declarative configuration and automation. [4] In a production environment, Kubernetes is an excellent tool to manage the containers that run the speech recognition container and ensure as little downtime as possible. For example, when an app container is out of order, another container starts up to guarantee availability. Kubernetes have several useful functionalities such as,

1. **Service Discovery and Load Balancing**

   It can expose a service of an app container to the public and allow users to access the container via the service protocol. It can also load balance incoming user requests to ensure a pod does not get overwhelmed and cause service disruption. [4]

2. **Autoscaling**

   Kubernetes have the feature to scale the number of virtual machines or nodes[2] to provide the necessary resources i.e. CPU computing cores and memory to start the containers dynamically according to the number of user requests to the service. [4]

3. **Self-healing**

   Kubernetes restarts failed containers, replaces non-functional containers, remove containers that failed user-defined health checks and only show the containers to the clients in their ready state. [4]

[2] In this project, a node refers to a virtual machine instance

Pods

A pod is a group of one or more containers with shared storage or network within the same Kubernetes cluster. [5] It is the basic unit for deploying an application within the Kubernetes cluster. The Kubernetes controller plane manages the pods according to the status of the cluster in line with the configuration files applied at the cluster set up.

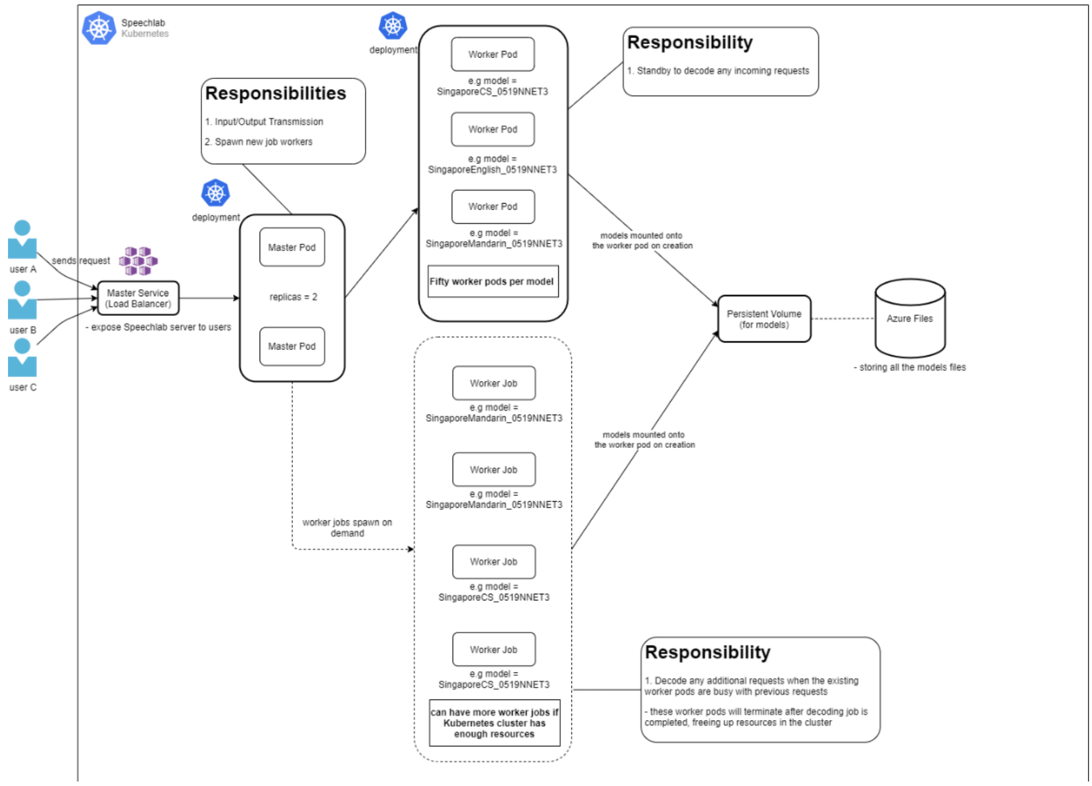
In this case, the speech recognition master server script is run by a container in a master pod loaded with the Docker image where its main role is to maintain connection with the user and the worker pod for the input and output of the transcribing job. Similarly, a worker pod runs the Docker image in a container where it also interfaces with the persistent volume storing the model files.

Persistent Volume

Persistent Volume (PV) is a storage mechanism which allows data to be stored within the Kubernetes cluster beyond the lifecycle of a pod or container. In many cases, a pod or container may be recycled when it finished performing its designated task; new pods may be initiated to provide its function. Persistent volume usually last for as long as the Kubernetes cluster itself as it contains the static files necessary for the pods to mount upon pod creation. Pod configuration can be modified to ensure that the pod mount the persistent volume and is thus able to fetch files stored in the persistent volume during its operations.

In this case, the speech recognition server makes use of a persistent volume to store all the model files e.g. *SingaporeCS_0519NNET3* so that a worker pod can put them into the decoder for its speech-to-text operation.

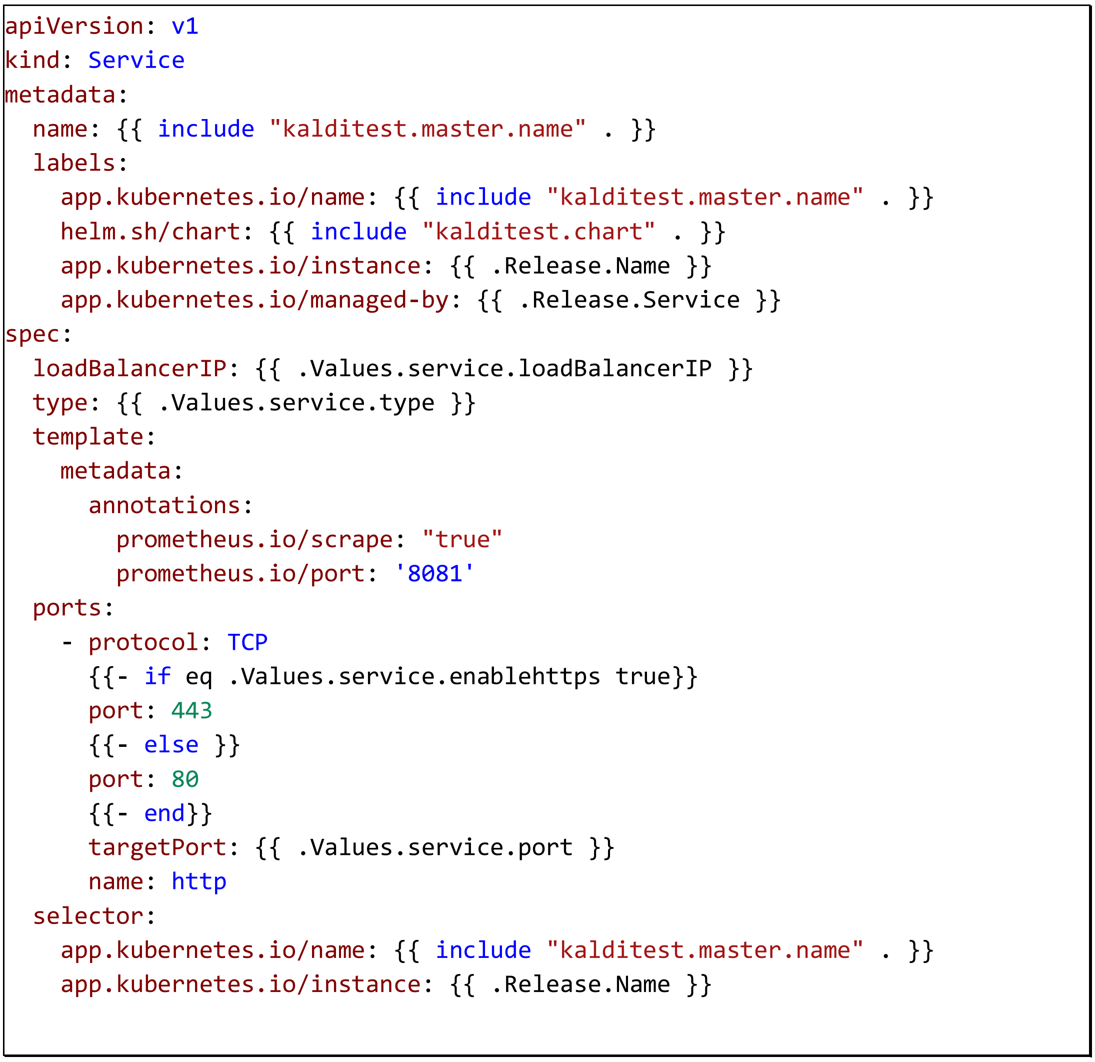
2.3 System Architecture



Speechlab
Kubernetes
deployment
Worker Pod
e.g model = SingaporeCS_0519NNET3
Worker Pod
e.g model = SingaporeEnglish_0519NNET3
Worker Pod
e.g model = SingaporeMandarin_0519NNET3
Fifty worker pods per model
Responsibility
1. Standby to decode any incoming requests
Responsibilities
1. Input/Output Transmission
2. Spawn new job workers
deployment
user A
sends request
user B
user C
Master Service (Load Balancer)
- expose Speechlab server to users
Master Pod
replicas = 2
Master Pod
models mounted onto the worker pod on creation
Persistent Volume (for models)
Azure Files
- storing all the models files
Worker Job
e.g model = SingaporeMandarin_0519NNET3
Worker Job
e.g model = SingaporeMandarin_0519NNET3
models mounted onto the worker pod on creation
worker jobs spawn on demand
Worker Job
e.g model = SingaporeCS_0519NNET3
Worker Job
e.g model = SingaporeCS_0519NNET3
can have more worker jobs if Kubernetes cluster has enough resources
Responsibility
1. Decode any additional requests when the existing worker pods are busy with previous requests
- these worker pods will terminate after decoding job is completed, freeing up resources in the cluster

Figure 2 Overall System Architecture

The overall system architecture shows the way the master pods and worker pods are arranged to communicate with each other within the Kubernetes cluster. A master service pod will be exposed to external users with an IP address where users can access the speech recognition system with various methods e.g. performing a cURL command and using Python client script. The master service will direct all the user requests to the master pods where they will load balance the workload and distribute it evenly to all the worker pods.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ include "kalditest.master.name" . }}
  labels:
    app.kubernetes.io/name: {{ include "kalditest.master.name" . }}
    helm.sh/chart: {{ include "kalditest.chart" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
    app.kubernetes.io/managed-by: {{ .Release.Service }}
spec:
  loadBalancerIP: {{ .Values.service.loadBalancerIP }}
  type: {{ .Values.service.type }}
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: '8081'
  ports:
    - protocol: TCP
      {{- if eq .Values.service.enablehttps true}}
      port: 443
      {{- else }}
      port: 80
      {{- end}}
      targetPort: {{ .Values.service.port }}
      name: http
  selector:
    app.kubernetes.io/name: {{ include "kalditest.master.name" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
```

Figure 3 Code Snippet for deploying Master Service pod

(refer to Appendix A for its values)

The Kubernetes cluster consists of several static long-running worker pods and transient job workers. The number of static long running worker pods is fixed upon deployment of the Kubernetes cluster. The purpose of the static long running worker pods is to ensure the availability of the system **at all times**, since transient job workers require some time to start up upon user request. For the reference of the following trains of thought, only one language model is loaded – "*SingaporeCS_0519NNET3*" into the Kubernetes cluster. For a single language model, there are 50 static long running workers capable of transcribing English and Mandarin based audio input.[3]

2.3.1 Master Pods

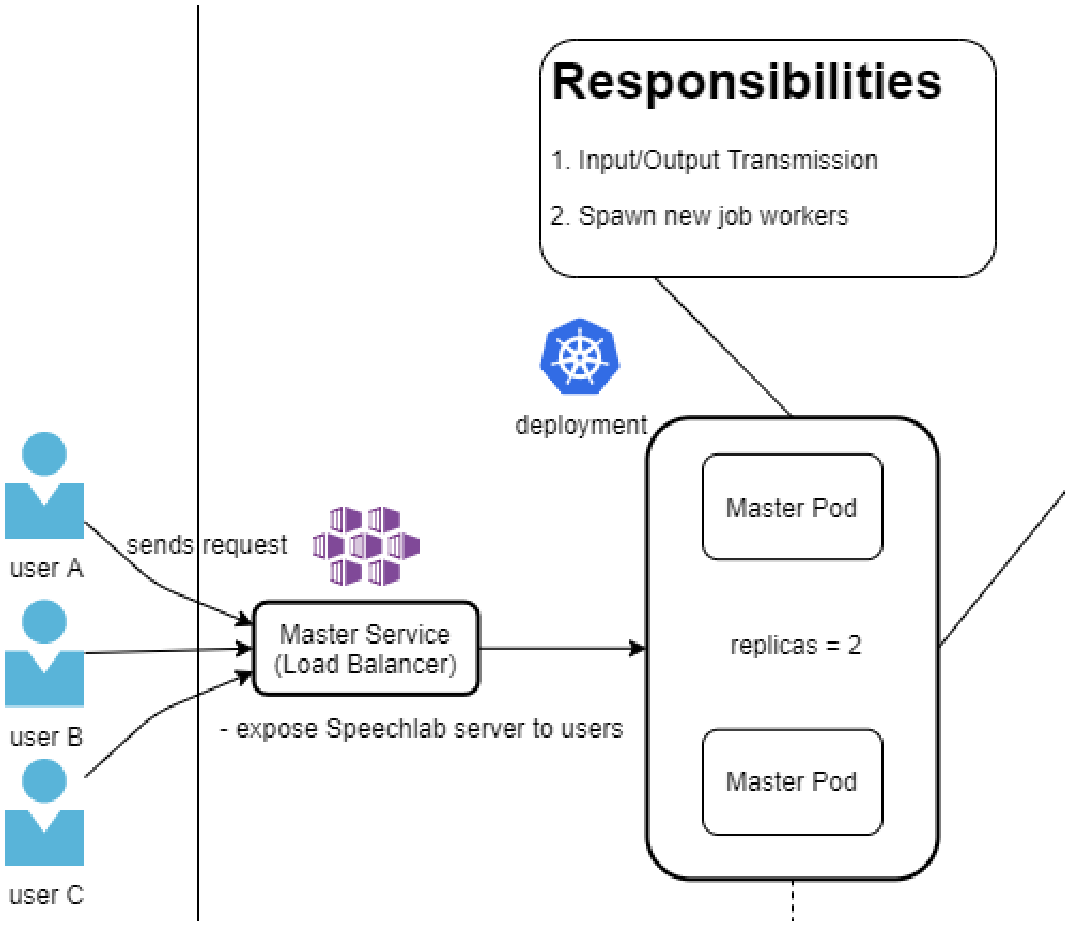


Figure 4 Cropped Image of System Architecture detailing the Master pods

The master pods function as the interface between the master service accepting user requests and the worker pods that carry out the decoding jobs. The master pods control the containers which run the script – `start_master.sh` that starts the master servers. The `start_master.sh` script is run when the master pod is created and ready for operation.

[3] Number of long-running worker pods are not fixed at 50 and can be changed flexibly according to the actual user demand

```
#!/bin/bash

export PYTHONIOENCODING=utf8

# schedule delete completed job
python3 /home/appuser/opt/cronjob.py &

if [ "$ENABLE_HTTPS" == "true" ] ; then
    python3 /home/appuser/opt/kaldi-gstreamer-
server/kaldigstserver/master_server.py --
certfile=/home/appuser/opt/ssl/fullchain.cer  --
keyfile=/home/appuser/opt/ssl/dev.aisingapore.org.key --
port=8080 2>&1 | tee /home/appuser/opt/master.log
else
    python3 /home/appuser/opt/kaldi-gstreamer-
server/kaldigstserver/master_server.py 2>&1 | tee /home/appuser/opt/master.log
fi
```

*Figure 5 Code snippet of **`start_master.sh`** script*

Roles and Responsibilities

The master pods have 2 main responsibilities,

1. Input and Output Transmission
2. Spawn new job workers

Input and Output Transmission

Considering that the speech-to-text decoding system is expected to handle 100 concurrent users, the master pods have at least two replicas to direct the incoming requests to the worker pods so that the system is not overwhelmed. Having two replicas of the master pods, it ensures that when one master pod is down, the other master pod can continue to accept user requests. In the event that the system is expected to handle more traffic, the number of master pod replicas can be increased accordingly.

Secondly, the master pod also runs the `master_server.py` script (which can be found in Appendix B: Code Snippet of `master_server.py`). The script opens up a Tornado[4] [6] server stream between the master pod and the worker pod. Any voice input data from the user request is passed from the master service to the master pod which is then passed on to the worker pod for decoding. The Tornado web server allows the voice data stream to be transmitted to the worker pod for decoding asynchronously and efficiently. Tornado can scale to tens of thousands of open connections, which is very effective for long polling, WebSockets and applications which require a long-lived connection between the user and the server. [6] In this project, the speech-to-text system has several use cases,

1. Allow users to transmit audio input and receive the text output in real time
2. Allow users to send audio files and receive text output

In the case of the aforementioned use case 1, the Tornado framework facilitates the transmission of non-blocking audio input to the worker pod for real time decoding into text output which is then transmitted back to the user via the master pod. Such bidirectional data transmission is made efficient with the use of the Tornado web framework.

[4] Tornado web server is a web framework that uses non-blocking network IO to allow multiple concurrent data streaming channels between the client and the server. [6]

Spawn new Job workers

All request traffic passes through the master service pod and the master pods. Hence, the master pods which contain the master server script have the data of all incoming user requests. Different requests vary in terms of the language model used e.g. *SingaporeCS_0519NNET3*, since there is only a limited number of static long running pods available, the Kubernetes cluster may need to spawn more job workers to meet the increase in demand for decoding jobs. The master pod will keep track of the number of available worker pods for each language model and trigger the spawn of new job workers conditionally. The conditions to spawn new worker jobs are either lack of workers that support the requested language model or the number of available workers for the requested language model is less than **5**. In other words, there will always be at least 5 worker pods scheduled to pre-empt any future incoming requests.

```
try:
    spawn_worker = (model not in self.application.available_workers) or len(
    self.application.available_workers[model]) <= 5
    if spawn_worker:
        logging.info('no available workers for model: {}, spawning new worker'
.format(model))
        SpawnWorker(model=model).start()

        self.worker = self.application.available_workers[model].pop()
        self.application.send_status_update()
        logging.info("%s: Using worker %s" % (self.id, self.__str__()))
        self.worker.set_client_socket(self)

        content_type = self.request.headers.get("Content-Type", None)
        if content_type:
            content_type = content_type_to_caps(content_type)
            logging.info("%s: Using content type: %s" %
(self.id, content_type))
            self.worker.write_message(json.dumps(dict(id=self.id, content_type
=content_type, user_id=self.user_id, content_id=self.content_id)))
except KeyError:
   logging.warn("%s: No worker available for client request" % self.id)
   logging.exception("no available worker error message")
   self.set_status(503)
   self.finish("No workers available, please re-try 60 seconds later")
```

Figure 6 Code snippet of ***master_server.py*** *showing the master pod keeping track of the number of available workers for each language model*

When a new worker job is spun up by the Kubernetes system, the worker pod will attempt to open a web socket connection with the master pods. If the Kubernetes credentials are configured correctly, the worker pod would be able to successfully connect to the master pod. Subsequently, the master pods will note the increase the number of available worker pods.

Resources Used

Each master server pod is allocated **2 vCPU cores** and **2GiB of memory**. Given the roles and responsibility of the master pods are mainly to facilitate input, output transmission and to spawn new worker jobs, the resources allocated to the master pods are sufficient.

2.3.2 Long-running static Worker Pods

The system architecture is designed to have high availability and long-running static worker pods are deployed to ensure that there are always worker pods available to carry out decoding jobs. When a worker pod is transcribing audio input from a user, it cannot accept request from another user. One worker pod can only service one user request at a point in time. Long-running static worker pods will carry out the decoding jobs and revert to a ‘standby’ mode and wait for the next incoming request.

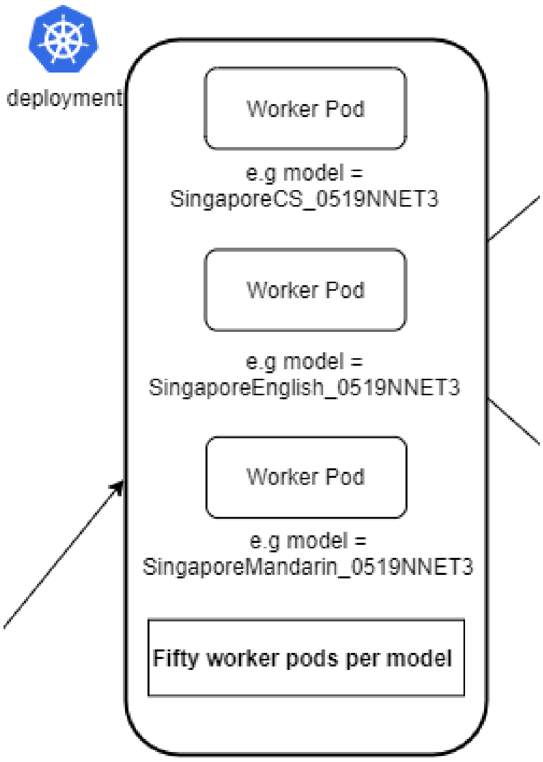


Figure 7 Cropped Image of System Architecture detailing the Worker pods

The worker pods control the containers that contain the worker script i.e. `start_worker.sh` to initialise the decoding jobs. The `start_worker.sh` script is run when the worker pod is created and ready for operation.

```bash
#!/bin/bash

echo "start_worker.sh: MASTER=$MASTER, MODEL_DIR=$MODEL_DIR RUN_FREQ=$RUN_FREQ"

#start worker and connect it to the master
export GST_PLUGIN_PATH=/home/appuser/opt/gst-kaldi-nnet2-online/src/:/home/appuser/opt/kaldi/src/gst-plugin/
export PYTHONIOENCODING=utf8

# automatically use engine template file if the model does not have engine.yaml file
FILE=/home/appuser/opt/models/$MODEL_DIR/engine.yaml
if test -f "$FILE"; then
    echo "$FILE exist"
    export USE_WHICH_ENGINE_FILE=$FILE
else
    echo "$FILE does not exist, use engine template"
    sed -i 's/{{MODEL_DIR}}/'"$MODEL_DIR"'/g' /home/appuser/opt/engine_template.yaml
    export USE_WHICH_ENGINE_FILE=/home/appuser/opt/engine_template.yaml
fi

echo "decided to use engine file: $USE_WHICH_ENGINE_FILE"

if [ "$ENABLE_HTTPS" == "true" ] ; then
    python /home/appuser/opt/kaldi-gstreamer-server/kaldigstserver/worker.py -c $USE_WHICH_ENGINE_FILE -u wss://$MASTER/worker/ws/speech 2>&1 | tee /home/appuser/opt/worker.log
else
    python /home/appuser/opt/kaldi-gstreamer-server/kaldigstserver/worker.py -c $USE_WHICH_ENGINE_FILE -u ws://$MASTER/worker/ws/speech 2>&1 | tee /home/appuser/opt/worker.log
fi
```

Figure 8 Code Snippet of ***start_worker.sh*** script

`worker.py` is the script that contains the logic to process all the incoming user decoding requests. Upon receiving a new user request, the script identifies the language model used and fork a Tornado sub process to start the decoding job. The worker.py script takes in few parameters namely,

Engine file (which can be found in

1. Appendix C: Code Snippet of `engine_template.yaml`)
2. Web socket URI between the master pod and worker pod

The engine file provides a set of configuration values for the processor in the worker pod to use during decoding. This file is part of the Kaldi toolkit and GStreamer framework used to implement a real-time full-duplex speech recognition server. They contain the language model properties which are used by the worker pods to carry out the decoding job.

All the relevant Kaldi toolkit files including `worker.py` and `master_server.py` are packaged into a Docker container image which are then run in the containers of the master and worker pods.

Resources Used

Each worker pod is allocated **1 vCPU core** and **5GiB of memory**. A worker pod is recommended to have at least 4GiB of RAM to ensure smooth decoding process. [7]

2.3.3 Worker Jobs

Worker jobs are similar to the long-running static workers mentioned earlier with the subtle difference that they are **recycled** after completing the decoding jobs. The worker pods which are running the decoding operations are **resource intensive**, requiring at least 4GiB of RAM to carry out the speech-to-text operation. An efficient way of using these resources would be to **free up** these pods after they are done with the decoding job. The worker jobs will be automatically deleted **100 seconds** it completed the decoding job. Also, the worker job will terminate itself after 15 minutes of inactivity and be deleted from the Kubernetes cluster.

Freed-up resources are returned to the Kubernetes cluster where the master pods will decide to spin up more worker jobs when necessary. There is no hard limit on the number of worker jobs that can be created; as long as the Kubernetes cluster has enough virtual machine instances to meet the computing resources required, the worker jobs may be created.

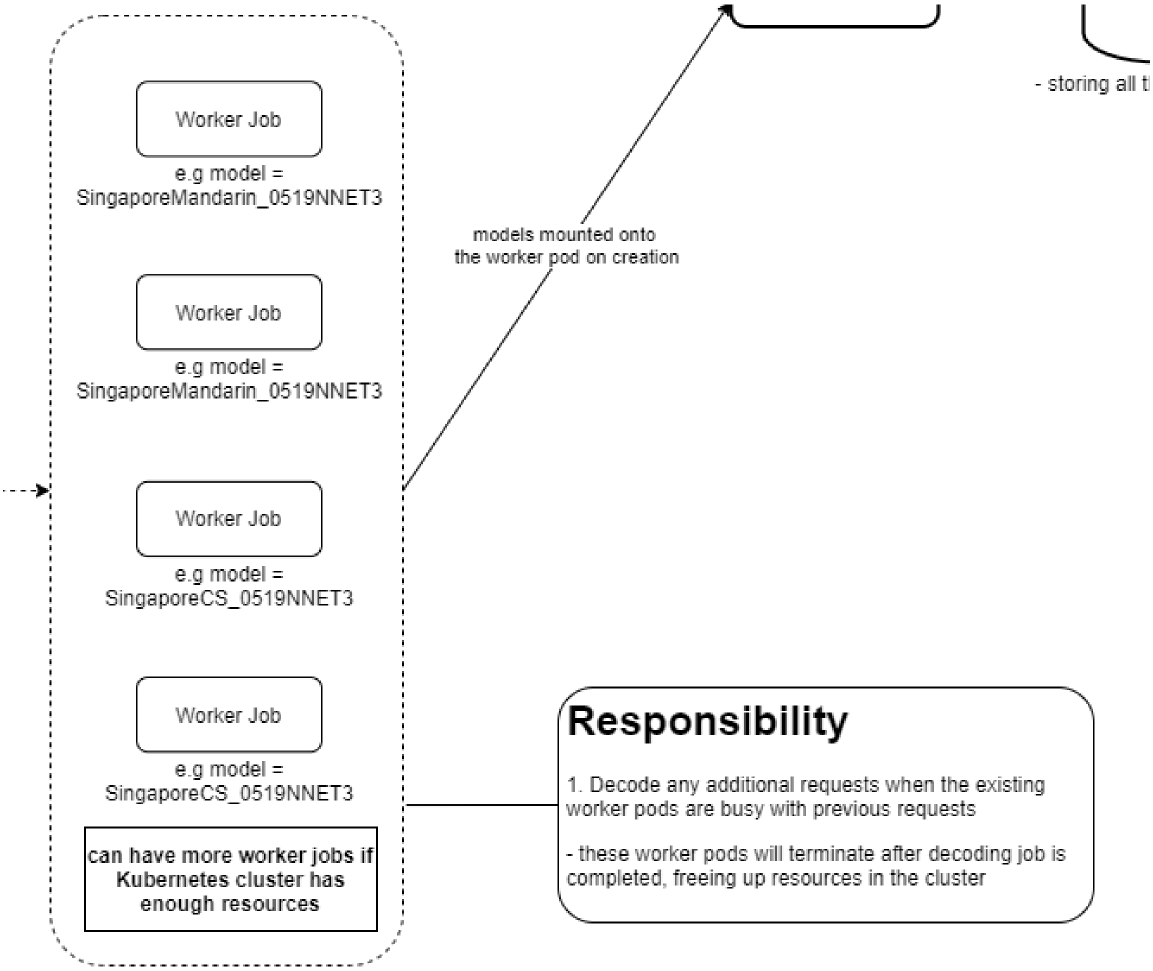

- storing all t
models mounted onto
the worker pod on creation
Worker Job
e.g model =
SingaporeMandarin_0519NNET3
Worker Job
e.g model =
SingaporeMandarin_0519NNET3
Worker Job
e.g model =
SingaporeCS_0519NNET3
Worker Job
e.g model =
SingaporeCS_0519NNET3
can have more worker jobs if Kubernetes cluster has enough resources
Responsibility
1. Decode any additional requests when the existing worker pods are busy with previous requests
- these worker pods will terminate after decoding job is completed, freeing up resources in the cluster

Figure 9 Cropped Image of System Architecture detailing the Worker Jobs

Resources Used

Each worker job pod is allocated **1 vCPU core** and **5GiB of memory**. A worker job has the same resource allocation as the long-running static worker pods.

2.3.4 Persistent Volume

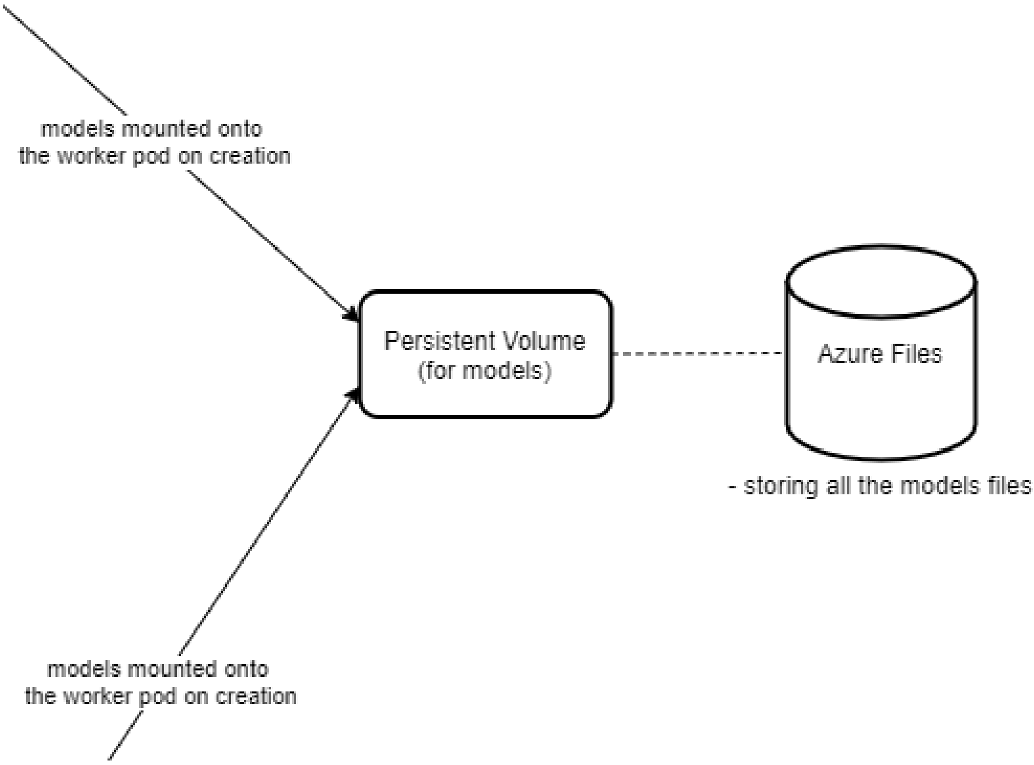

models mounted onto
the worker pod on creation
Persistent Volume
(for models)
Azure Files
- storing all the models files
models mounted onto
the worker pod on creation

Figure 10 Cropped Image of System Architecture detailing the persistent volume

Persistent volume is the mechanism used to hold static files stored in the Kubernetes cluster. These files stored in the persistent volume will persist beyond the lifecycle of the pods in the Kubernetes cluster. In this project, worker jobs will be recycled once they complete decoding a user's request, but the persistent volumes stay in the Kubernetes cluster. [8]

The persistent volume created is used to **store the language models** e.g. *SingaporeCS_0519NNET3*. The language models are mounted on every worker pod at the path - `/home/appuser/opt/models` when the pods are first initialised. The speech-to-text recognition system require these language model files to carry out the decoding jobs. These language model files can always be reused when the user request the same language model for decoding hence these files are expected to persist beyond the lifecycle of the worker pods.

In this project, the type of file volume used for the persistent volume is Azure Files. Azure Files is a file share service offered by Microsoft Azure which uses the industry standard Server Message Block (SMB) protocol to manage a file storage system on the cloud server. Azure Files allows for shared access to the files hosted on the Kubernetes cluster easily through provisioning of persistent volume. [9] Language models may be uploaded to the Azure File storage via a command line e.g.

```
az storage file upload-batch -d $MODEL_SHARE --account-
key $STORAGE_KEY --account-name $STORAGE_ACCOUNT_NAME -
s models/
```

or by logging into the Azure portal to manually upload the files. To add a new language model, the developer may first upload the files to the Azure Files storage then edit the `values.yaml` file (refer to Appendix A: Code Snippet of `values.yaml` for cluster deployment) to specify the number of long-running worker pods needed for the new language model.

2.4 Kubernetes Cluster Autoscaling

The Azure Kubernetes Service (AKS) which is used in this project facilitates the cluster autoscaling feature. Master pods and worker pods consume computing resources i.e. virtual CPU and RAM and the demand for these resources fluctuate according to the number of user requests. In this project, we emphasize on high availability and scalability of the speech-to-text recognition system available to the users.

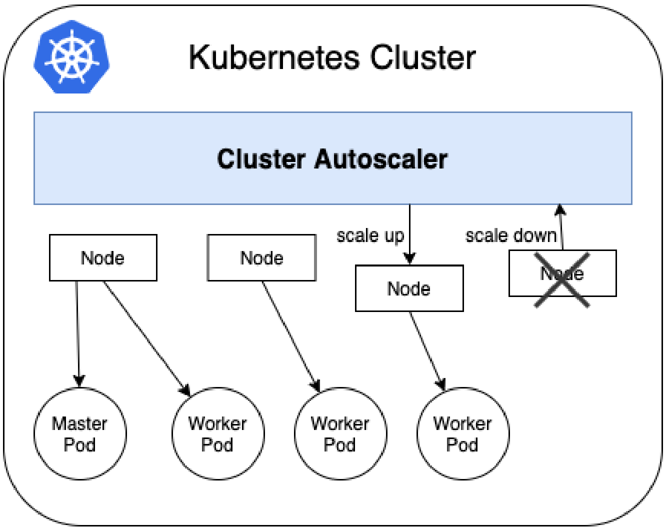


Figure 11 Visual Illustration of Kubernetes Cluster Autoscaler (not drawn to actual deployment)

The cluster autoscaler component monitors the Kubernetes cluster for pods which cannot be scheduled due to **resource constraints**. Such issue arises when the master pods need to spin up new worker jobs to meet the demand but the nodes [5] i.e. virtual machines (VMs) in the cluster do not have enough resources to create new worker jobs. The nodes are regularly checked for unscheduled pods and also for nodes that are idling. If the nodes are not running any pods, the cluster will scale

[5] A Kubernetes cluster has a fixed number of nodes which are virtual machine instances available to create new pods. When demand increases, the cluster autoscaler helps to initialize new nodes to increase the amount of computing resources available in the pool.

down and terminate the node to minimise utilisation of nodes. [10] The ability to scale the number of nodes up and down automatically makes the deployment of the speech-to-text recognition system efficient and cost-effective.

2.5 Resources Computation

The virtual machine instance used in the Kubernetes cluster is the **Standard B4ms.** A Standard B4ms virtual machine instance has properties as follows, [11]

Virtual CPU(s)	**RAM**	**Temporary Storage**
4	16GiB	32GiB

The objective of the project is to support a peak concurrent decoding job of **100** for a single language model. Assuming that on average, less than 50 concurrent connections are maintained to the server hence a minimum of 50 long-running workers is always needed to be kept online in Kubernetes cluster.[6]

[6] The number of long-running worker pods can be adjusted according to the actual user demand. 50 is only a rough gauge to support 100 concurrent connections to the server.

2.5.1 Breakdown of Minimum Resource Requirements

Component	Quantity	vCPU required	RAM required
Long-running worker pods	50	1 worker pod takes up 1vCPU 50 worker pods take up **50vCPU**	1 worker pod takes up 5GiB of RAM 50 worker pods take up **250GiB of RAM**
Master pods	2	1 master pod takes up 2vCPU 2 master pods take up **4vCPU**	1 master pod takes up 2GiB of RAM 2 master pods take up **4GiB of RAM**
Prometheus	1	**~0.5vCPU**	**~4GiB of RAM**
Grafana	1	**~0.03vCPU**	**~33MiB of RAM**
Total minimum amount of resources required[7]		**~55vCPU**	**~260GiB of RAM**

$$Min.num.of\ nodes\ based\ on\ vCPU\ requirements = \frac{55vCPU}{4vCPU} \approx \mathbf{14}$$

$$Min.num.of\ nodes\ based\ on\ RAM\ requirements = \frac{260GiB}{16GiB} \approx \mathbf{17}$$

Since one Standard B4ms virtual machine instance can provide 4vCPU and 16GiB of RAM, minimum number of nodes required in the Kubernetes cluster to support 50 long-running worker pods is **17**.

[7] Minimum amount of resources required are computed based on the major components in the Kubernetes cluster.

In order to achieve 100 concurrent connections to the server, there should be at least **50** long-running worker pods and **50** worker jobs ready in the Kubernetes cluster.

2.5.2 Breakdown of Maximum Resource Requirements

Component	Quantity	vCPU required	RAM required
Long-running worker pods	50	1 worker pod takes up 1vCPU 50 worker pods take up **50vCPU**	1 worker pod takes up 5GiB of RAM 50 worker pods take up **250GiB of RAM**
Worker Jobs	50	1 worker job takes up 1vCPU 50 worker jobs take up **50vCPU**	1 worker job takes up 5GiB of RAM 50 worker jobs take up **250GiB of RAM**
Master pods	2	1 master pod takes up 2vCPU 2 master pods take up **4vCPU**	1 master pod takes up 2GiB of RAM 2 master pods take up **4GiB of RAM**
Prometheus	1	**~0.5vCPU**	**~4GiB of RAM**
Grafana	1	**~0.03vCPU**	**~33MiB of RAM**
Total minimum amount of resources required[8]		**~105vCPU**	**~510GiB of RAM**

[8] Minimum amount of resources required are computed based on the major components in the Kubernetes cluster.

$$Min.num.of\ nodes\ based\ on\ vCPU\ requirements = \frac{105vCPU}{4vCPU} \approx \mathbf{27}$$

$$Min.num.of\ nodes\ based\ on\ RAM\ requirements = \frac{505GiB}{16GiB} \approx \mathbf{33}$$

Since one Standard B4ms virtual machine instance can provide 4vCPU and 16GiB of RAM, minimum number of nodes required in the Kubernetes cluster to support a total of 100 workers is **33**.

2.6 Monitoring System

A monitoring dashboard is also implemented as part of the design of a high availability and scalability deployment of the speech-to-text recognition system. In order to transform the speech-to-text recognition system into a commercial solution, the data derived from the real-time usage helps the deployment to become more efficient and cost-effective. Knowing the usage patterns of the users allow the Dev-Ops engineer to fine tune the Kubernetes configuration to better suit the needs of the users. The number of long-running worker pods can be adjusted to optimise the use of computing resources.

2.6.1 Prometheus

Prometheus is an open-source monitoring and alert system which can be integrated into the Kubernetes cluster easily. It extracts the metrics [9] from the Kubernetes cluster and pass them to Grafana to display on the dashboard. It can make powerful queries to the Kubernetes cluster that slices time series data in order to generate ad-hoc graphs, tables and alerts. [12]

[9] Metrics available from the Kubernetes cluster include the CPU usage of the pods, memory usage of the pods etc

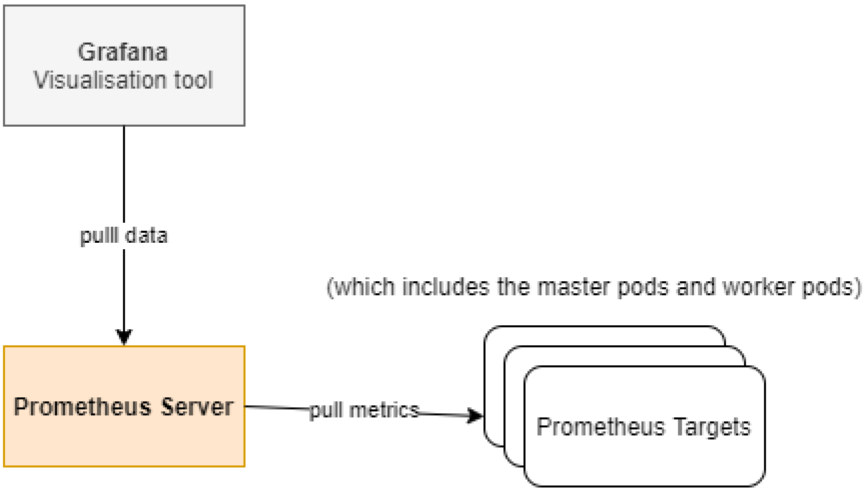


Figure 12 Prometheus Sub-Architecture [13]

Prometheus allows for custom metrics to be added into the application to create a better visualisation of the system's performance.

The following table shows the custom metrics implemented for the speech-to-text recognition system,

Custom Metric(s)	**Description**
`number_of_request_received_by_master`	This metric keeps track of the total number of incoming user requests.
`number_of_worker_available`	This metric keeps track of the current number of **available workers** for each language model.
`number_of_request_reject`	This metric keeps track of the total number of user request **rejected**.
`worker_status`	This metric keeps track of the **status** of the workers, 1. Idle 2. Running (decoding in progress)
`worker_model`	This metric keeps track of the **language model** which the worker pod is using

In this project, Prometheus is set up to scrape data from the master and worker pods every **10 seconds**. Custom metrics are obtained from self-defined targets which are the master pods and worker pods. Scrape interval timing was set to be 10 seconds considering the duration of the audio input transmitted by the user is expected to longer than 10 seconds.

The scrape interval was chosen to optimise the performance of the monitoring system while keeping the system data storage efficient. A shorter scrape interval would allow the system to capture more data points but at the same time generate more data for storage i.e. take up more persistent storage space, which may not be helpful in evaluating the system performance anyway.

2.6.2 Grafana

Grafana is an open source analytics and monitoring solution which helps to visualise the data in useful manner. In this project, Grafana is used to translate the empirical data extracted from the Prometheus server into something laymen can understand. It helps the developers understand the state of the speech-to-text recognition system and its performance over time.

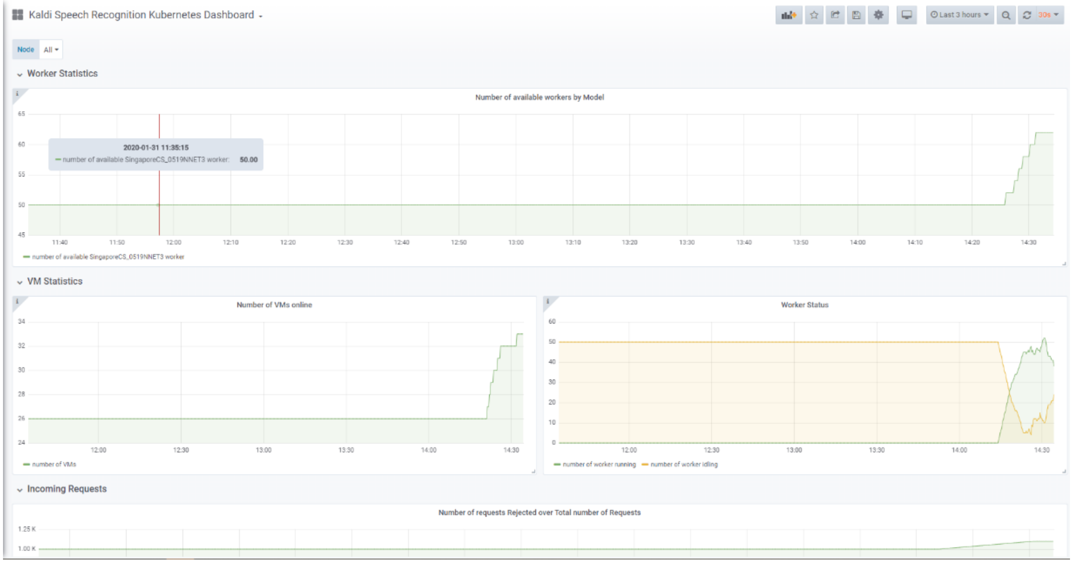


Figure 13 Monitoring Dashboard powered by Grafana and Prometheus

The figure above shows the overall monitoring dashboard of the speech-to-text recognition system.

There are a few metrics which indicates the status of the system namely,

1. **Worker Statistics**

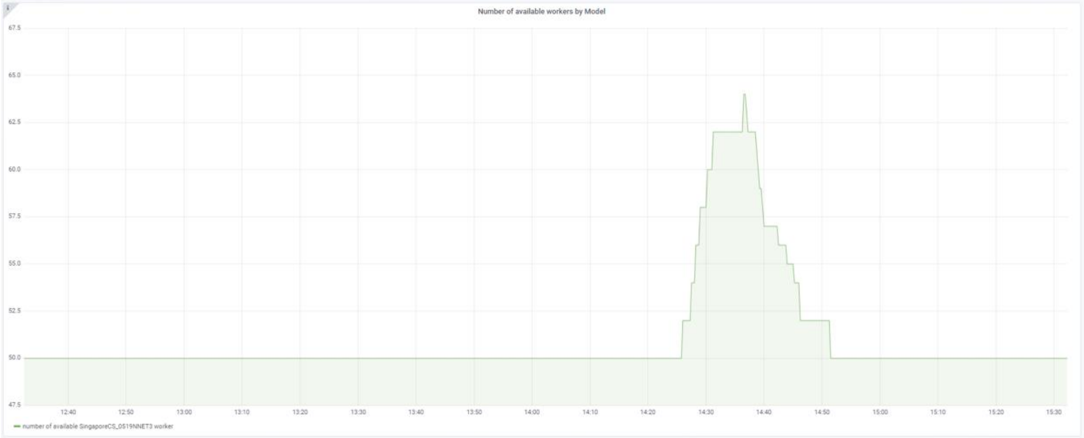


Figure 14 Graph showing the number of available workers by language models [10]

The graph above shows the number of available workers categorised by the language model over time. The spike in number of available workers is due to the increase in number of user requests at the specific timing as the master pods trigger the initialization of new worker jobs.

[10] For the sake of clarity, only **one** language model's worker statistics are shown. Multiple language models' worker statistics can be shown overlapping each other in a single graph.

2. Number of Virtual Machines online

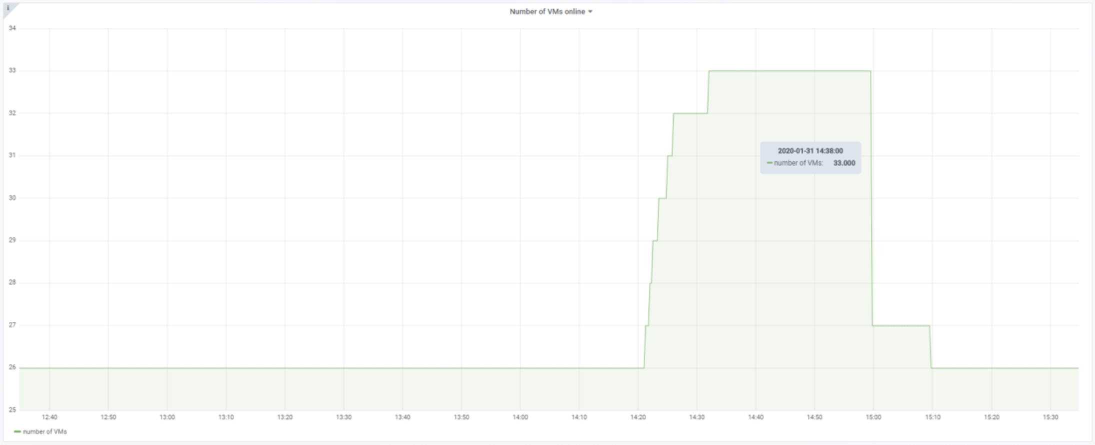


Figure 15 Graph showing the number of virtual machine online

As the demand to create more worker jobs increases, the number of virtual machines available to supply the computing resources may have to increase as well. As seen in the graph above, the number of virtual machines in the Kubernetes cluster fluctuate over time according to the number of user requests.

3. Worker status

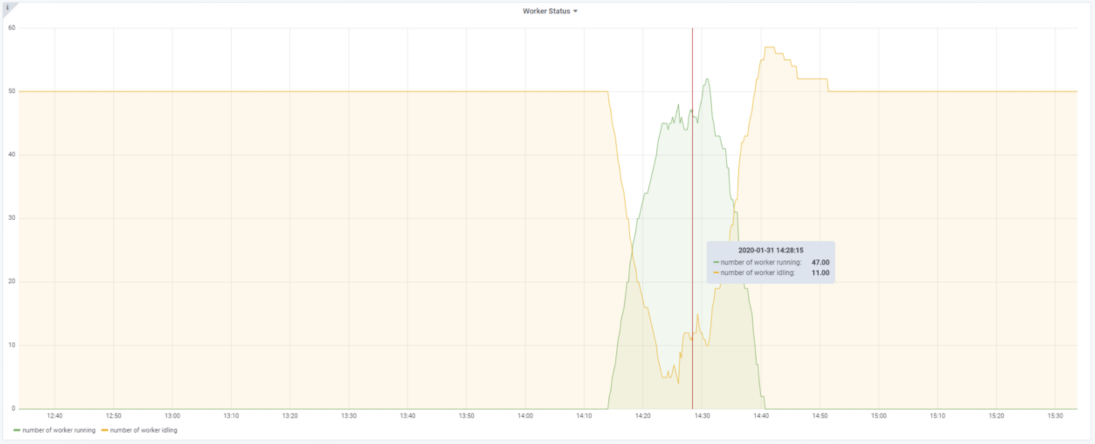


Figure 16 Graph showing the statuses of the worker pods

The graph above shows the number of workers idling or running a decoding job.

4. Number of incoming requests Rejected

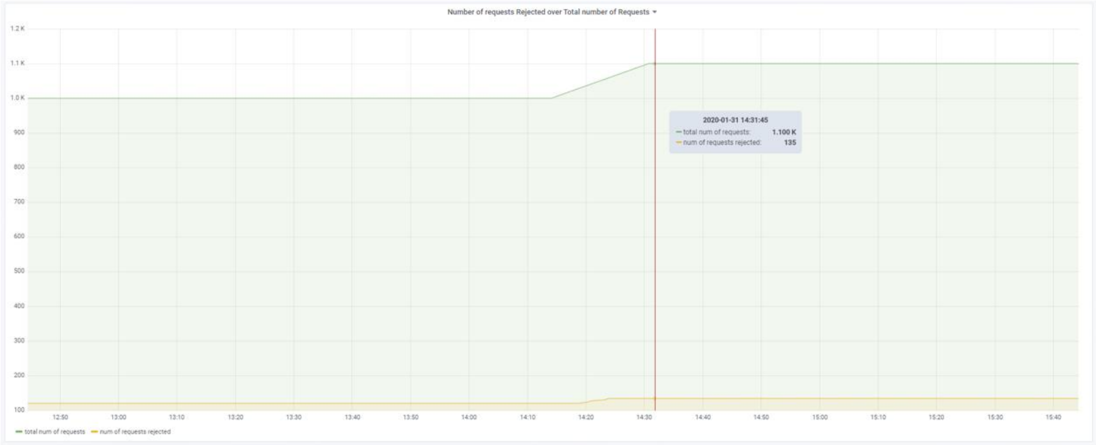


Figure 17 Graph showing the number of incoming request rejected

5. Resource Usage

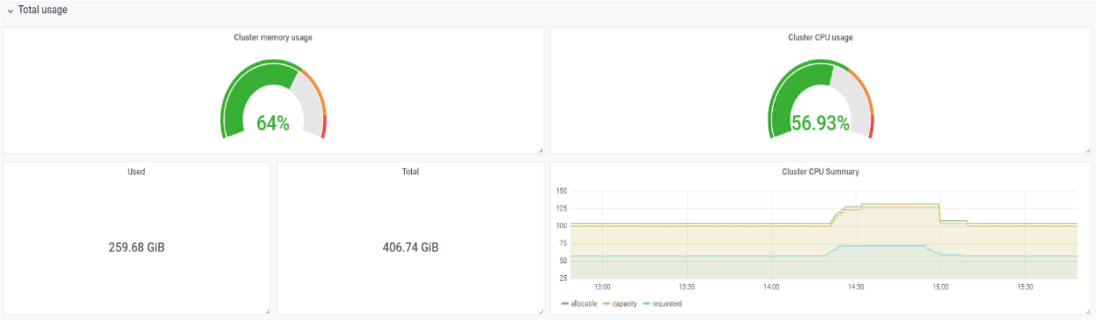


Figure 18 Graph showing the resource usage of the Kubernetes cluster

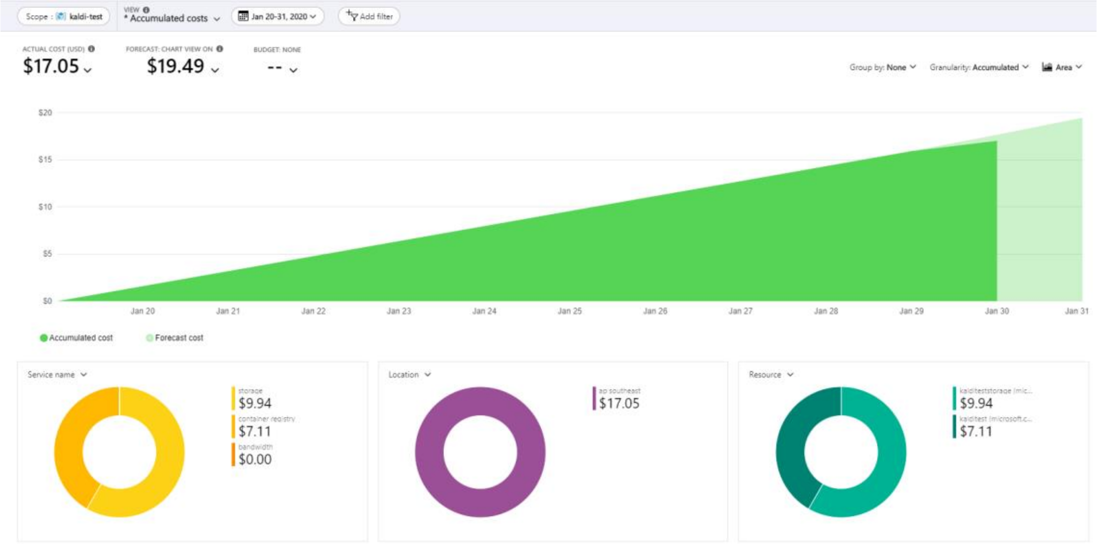
The graph above shows the resource usage of the Kubernetes cluster. From the graph, one can interpret the cluster's memory usage and the cluster's CPU usage at one glance. It allows the developers to quickly diagnose any possible errors.

Chapter 3: Project Costs

Implementation of this project was done using Microsoft Azure’s public cloud services. Following cost estimates are therefore projected based on the rates offered by Microsoft Azure.

3.1 Simulated Costs

A testing script was used to simulate user requests sent to the system. The script scheduled an audio file to be sent to the system 100 times in an interval of 10 seconds. The cost was derived by simulating an average of 200 user requests daily during the period.



Scope : kaldi-test

VIEW: * Accumulated costs

Jan 20-31, 2020

ACTUAL COST (USD): $17.05

FORECAST: CHART VIEW ON: $19.49

BUDGET: NONE: --

Group by: None

Granularity: Accumulated

Area

Figure 19 Overall Cost Analysis shown in the Azure Portal

The figure above shows the overall cost analysis of the proposed system architecture deployment over the course of **12 days**.

The cost breakdown of the system for the period of 12 days is as follows,

Service	**Description**	**Cost (USD$)**
Azure Storage	Azure Storage [14] is static file storage service which is used to store the language model files to be accessed by the Kubernetes cluster.	$9.94
Container Registry	Azure Container Registry [15] is used to store the Docker image containing the Kaldi toolkit and other relevant scripts used in conjunction with the language model files to decode audio input	$7.11
Azure Kubernetes Service (AKS)	The Azure Kubernetes Service is the cluster management service that eases the operations and deployment of Kubernetes. [16]	Free
Virtual Machines Instances	Virtual machine instances are used to start up the worker and master pods for the decoding jobs. These virtual machine instances are managed by the Azure Kubernetes Service.	~$1033.06 [11]
Total Cost (USD$)	**~$1,050**	

[11] Actual cost for the virtual machine instances were not shown in the cost analysis. The estimated cost for the virtual machine instances is derived from the Azure Pricing Calculator. [17]

3.2 Projected Monthly Costs

Service	**Cost (USD$)**
Azure Storage	$24.85
Container Registry	$17.80
Azure Kubernetes Service (AKS)	Free
Virtual Machines Instances	$2582.64
Total Projected Monthly Cost (USD$)	**$2,625.30**

The projected monthly cost is calculated based on the minimum number of virtual machines needed to maintain 50 long-running worker pods i.e. 17 and the associated cost with the type of virtual machine used in the Kubernetes cluster. The number of virtual machine instances used during actual deployment may be higher, subjected to the usage of the system. During the tests, the system briefly scaled to use all 33 virtual machine instances to support a peak of 100 concurrent connections to the server.

Due to the high CPU and RAM requirements of the Kaldi toolkit used to enable the speech-to-text recognition system, the **Standard B4ms** virtual machine instances are used as the nodes in the Kubernetes cluster. The cost of using one Standard B4ms virtual machine instance is $0.211/hour. [17]

Chapter 4: System Performance

The performance of the system architecture is evaluated based on the following determinants,

1. **Time taken to 'spin up' a new worker job**

In the case where the number of available long-running worker pods is less than the number of user requests, new worker jobs have to be initialised to meet the user demand. While the new worker jobs are being initialised, users may have to wait for the worker jobs to get ready.

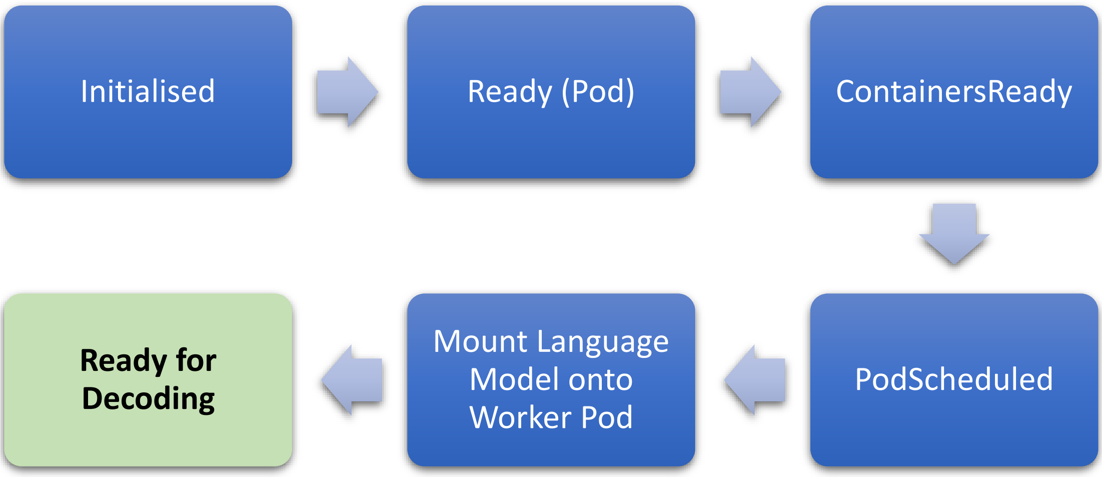


Figure 20 Pod Lifecycle

The whole process of initialising a new worker job takes only **38 seconds** which is two times faster than the current implementation of the speech-to-text recognition system at AISG which takes about **1 minute and 36 seconds** to start up a new worker job. The drastic fall in the time taken to start up a worker job is attributed to the use of Azure Files which has a higher throughput[12] than that of Blob Storage (which is being used at AISG). The worker pod is thus able to mount the language model files quicker and be ready for decoding.

[12] Azure Files has a maximum file throughput of up to 6,204 MiB/s while Blob Storage's maximum file throughput is 300MiB/s. [21]

2. Time to start up a VM instance

In the case where all the existing VM instances are exhausted of vCPU and RAM, the Kubernetes cluster will automatically start up a new VM instance to provide more computing resources i.e. vCPU and RAM to spin up more worker jobs. This is a time-consuming activity.

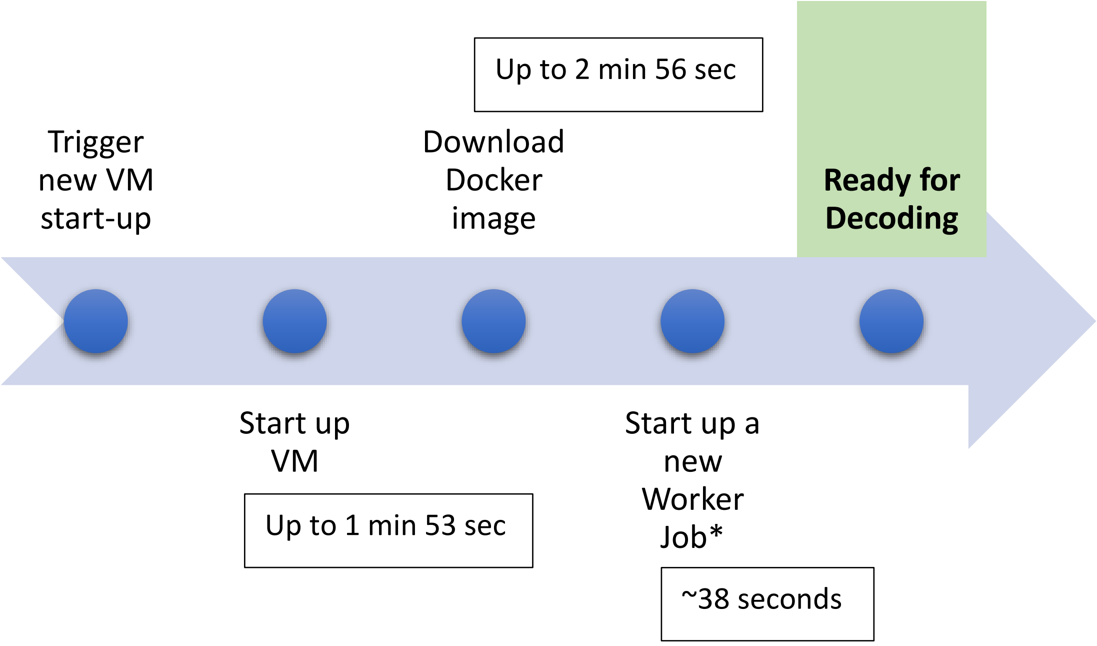


Figure 21 New VM Lifecycle

Although the maximum possible time taken to prepare for a new worker job to be ready may be long, fine tuning to the number of long-running worker pods ready in the Kubernetes cluster will fix the problem.

By anticipating the number of user requests during a certain time period, it is possible to reduce the user’s waiting time to **zero** by increasing the number of long-running worker pods beforehand. Alternatively, by ensuring that there is at least 5 worker pods available at any point in time (refer to Spawn new Job workers), pods will be scheduled early and trigger a node scale up via the cluster autoscaler if necessary so that less time is wasted on waiting for the VM instances to get ready. If any of the long-running worker pods are on standby, a user’s decoding request will be accepted immediately.

Chapter 5: Possible Improvements

5.1 Reducing Docker Image size

Currently, the size of the Docker image size used is about 3.2GiB. This is causing the newly initialised VM instances to take a long time to download the Docker image file, adding delays to decode user's request. The major reason for the huge Docker image size is because of the Kaldi toolkit source code. [18] It might be possible to use other toolkits with smaller file size so that the Docker image would also be smaller in size.

In addition, the dependencies included in the Dockerfile could be revised and redundant packages may be removed so that the size of the Docker image can be reduced.

5.2 Pre-package Docker Image into the VMs

As mentioned in earlier chapter, a new VM instance is initialised when there is not enough vCPU and RAM to spin up new worker jobs, and this takes some time. A way to reduce the time taken to download the Docker image is to package the Docker image file in the VM instance started by the Kubernetes cluster. When the VM instance is initialised, the Docker image is already present in the VM instance and there is no need for the VM instance to download it from the container registry thus reducing the waiting time. It is possible to use Packer to build automated virtual machine images that includes our custom Docker image which can in turn be used in the Kubernetes cluster. [19]

5.3 Using Azure Container Registry Teleportation

A viable option is to use Azure Container Registry Project Teleportation[13] which removes the cost of download and decompression of Docker images by mounting pre-expanded layers from the Azure Container Registry to the Azure container hosts. It is said that the Docker image download time could be reduced to less than 7.6 seconds for a Docker image file size of 5.1GiB. [20]

[13] This feature is still at preview stage and is only available to selected developers.

Chapter 6: Conclusion

The main objective of this project is to devise a way to create a speech-to-text recognition system which has high availability and high scalability. In this project, a Kubernetes-powered system architecture was designed to support concurrent usage of the speech-to-text recognition system by multiple users. In comparison to traditional deployment of the system, only one user may access the system at any point in time. Apart from ensuring high availability and high scalability, the project also aims to produce a cost-efficient system that can deliver the decoding function while keeping the cost of operations low.

While there may be scenarios where the waiting time for worker pods to be available is long, pre-emptive measures such as spawning new worker jobs early, help to reduce the waiting time for the users.

The design of the system architecture is robust and scalable but the key to optimising the system is to adjust the number of long-running workers available for each language model according to the usage patterns of the users. By anticipating the number of user requests in advance, the system can adapt to become even more cost-efficient. The number of VM instances on standby could be reduced further depending on the number of long-running worker pods on standby.

Appendix A: Code Snippet of `values.yaml` for cluster deployment

```yaml
models:
  # SingaporeEnglish_0519NNET3: 1
  SingaporeCS_0519NNET3: 50
  # SingaporeMandarin_0519NNET3: 1

image:
  repository: kalditest.azurecr.io/kalditestscaled
  tag: latest
  pullPolicy: Always
  pullSecrets: azure-cr-secret

nameOverride: ""
fullnameOverride: ""

service:
  type: LoadBalancer
  port: 8080
  resourceGroup: kaldi-test
  loadBalancerIP: 20.43.148.11
  enablehttps: false

commands:
  master:
    - '"/home/appuser/opt/tini"'
    - '"--"'
    - '"/home/appuser/opt/start_master.sh"'
  worker:
  # see _helpers.tpl on how full command constructed
    pre:
      - '"/home/appuser/opt/tini"'
      - '"--"'
      - '"/home/appuser/opt/start_worker.sh"'
fileshare:
  secretName: models-files-secret
  shareName: online-models
```

Appendix B: Code Snippet of `master_server.py`

```
.
.
.
num_req = prom.Counter('number_of_request_receive_by_master',
                       'number of request receive by master')
num_worker = prom.Gauge('number_of_worker_available',
                        'number of worker available')
num_req_reject = prom.Counter(
        'number_of_request_reject', 'number_of_request_reject')

class Application(tornado.web.Application):
    def __init__(self):
        settings = dict(
            cookie_secret="43oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o/Vo=",
            template_path=os.path.join(os.path.dirname(
                os.path.dirname(__file__)), "templates"),
            static_path=os.path.join(os.path.dirname(
                os.path.dirname(__file__)), "static"),
            xsrf_cookies=False,
            autoescape=None
        )

        handlers = [
            (r"/", MainHandler),
            #(r"/.well-known/acme-
challenge/(.*)", tornado.web.StaticFileHandler, {'path': '/home/appuser/opt/ss
l/verify/'}),
            (r"/client/ws/speech", DecoderSocketHandler),
            (r"/client/ws/status", StatusSocketHandler),
            # (r"/client/dynamic/reference", ReferenceHandler),
            (r"/client/dynamic/recognize", HttpChunkedRecognizeHandler),
            (r"/worker/ws/speech", WorkerSocketHandler),
            (r"/client/static/(.*)", tornado.web.StaticFileHandler,
             {'path': settings["static_path"]}),
            # (r"/prepare_job", HttpPrepareJobHandler),

        ]
        tornado.web.Application.__init__(self, handlers, **settings)
        self.available_workers = {}
        self.status_listeners = set()
        self.num_requests_processed = 0

    def send_status_update_single(self, ws):
        status = dict(num_workers_available=[{k: len(v)} for k, v in self.avai
lable_workers.items(
        )], num_requests_processed=self.num_requests_processed)
        ws.write_message(json.dumps(status))
```

Appendix C: Code Snippet of `engine_template.yaml`

```
use-nnet2: True
decoder:
    # All the properties nested here correspond to the kaldinnet2onlinedecoder
 GStreamer plugin properties.
    nnet-mode: 3
    use-threaded-decoder:  true
    model : /home/appuser/opt/models/{{MODEL_DIR}}/final.mdl
    word-syms : /home/appuser/opt/models/{{MODEL_DIR}}/words.txt
    fst : /home/appuser/opt/models/{{MODEL_DIR}}/HCLG.fst
    mfcc-config : /home/appuser/opt/models/{{MODEL_DIR}}/conf/mfcc.conf
    ivector-extraction-
config : /home/appuser/opt/models/{{MODEL_DIR}}/conf/ivector_extractor.conf
    max-active: 10000
    beam: 10.0
    lattice-beam: 6.0
    acoustic-scale: 1
    do-endpointing : true
    endpoint-silence-
phones : "1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20"
    traceback-period-in-secs: 0.25
    chunk-length-in-secs: 0.25
    num-nbest: 10

    frame-subsampling-factor: 3
out-dir: tmp

use-vad: False
silence-timeout: 10

logging:
    version : 1
    disable_existing_loggers: False
    formatters:
        simpleFormater:
            format: '%(asctime)s - %(levelname)7s: %(name)10s: %(message)s'
            datefmt: '%Y-%m-%d %H:%M:%S'
    handlers:
        console:
            class: logging.StreamHandler
            formatter: simpleFormater
            level: DEBUG
        debug_file_handler:
            class: logging.handlers.RotatingFileHandler
            level: DEBUG
            formatter: simpleFormater
            filename: verbose.log
            maxBytes: 10485760 #10Mb
            backupCount: 20
```

Bibliography

[1] Docker Inc., “What is a Container?,” [Online]. Available: https://www.docker.com/resources/what-container. [Accessed 15 January 2020].

[2] The Kubernetes Authors, “Production-Grade Container Orchestration,” [Online]. Available: https://kubernetes.io/. [Accessed 15 January 2020].

[3] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motl´ıcek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer and K. Vesely´, “The Kaldi Speech Recognition Toolkit,” 2011.

[4] The Kubernetes Authors, “What is Kubernetes,” 25 November 2019. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 21 January 2020].

[5] The Kubernetes Authors, “Pods,” 6 August 2019. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod/#what-is-a-pod. [Accessed 22 January 2020].

[6] The Tornado Authors, “Tornado Web Server,” [Online]. Available: https://www.tornadoweb.org/en/stable/. [Accessed 18 February 2020].

[7] jcsilva, “docker-kaldi-gstreamer-server,” [Online]. Available: https://github.com/jcsilva/docker-kaldi-gstreamer-server/blob/master/README.md#practical-example. [Accessed 7 March 2020].

[8] The Kubernetes Authors, “Persistent Volumes,” [Online]. Available: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#lifecycle-of-a-volume-and-claimPersistent Volumes. [Accessed 8 March 2020].

[9] Microsoft Azure, “What is Azure Files?,” [Online]. Available: https://docs.microsoft.com/en-us/azure/storage/files/storage-files-introduction#key-benefits. [Accessed 8 March 2020].

[10] Microsoft Azure, “Automatically scale a cluster to meet application demands on Azure Kubernetes Service (AKS),” [Online]. Available: https://docs.microsoft.com/en-us/azure/aks/cluster-autoscaler. [Accessed 9 March 2020].

[11] Microsoft Azure, “B-series burstable virtual machine sizes,” [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable. [Accessed 11 March 2020].

[12] Prometheus Authors, “Prometheus,” [Online]. Available: https://prometheus.io/. [Accessed 8 March 2020].

[13] Prometheus Authors, “What is Prometheus?,” [Online]. Available: https://prometheus.io/docs/introduction/overview/. [Accessed 9 March 2020].

[14] Microsoft Azure, “Storage,” [Online]. Available: https://azure.microsoft.com/en-us/services/storage/. [Accessed 11 March 2020].

[15] Microsoft Azure, “Container Registry,” [Online]. Available: https://azure.microsoft.com/en-us/services/container-registry/. [Accessed 11 March 2020].

[16] Microsoft Azure, “Azure Kubernetes Service (AKS),” [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/kubernetes-service/. [Accessed 11 March 2020].

[17] Azure Pricing Calculator, “Pricing calculator,” [Online]. Available: https://azure.microsoft.com/en-us/pricing/calculator/. [Accessed 11 March 2020].

[18] kaldi-asr, “Kaldi Speech Recognition Toolkit,” [Online]. Available: https://github.com/kaldi-asr/kaldi. [Accessed 12 march 2020].

[19] Packer, “Azure Virtual Machine Image Builders,” [Online]. Available: https://packer.io/docs/builders/azure.html. [Accessed 12 March 2020].

[20] Microsoft Azure, “Project Teleport Demos,” [Online]. Available: https://github.com/AzureCR/teleport. [Accessed 12 March 2020].

[21] Microsoft Azure, “Azure Files scalability and performance targets,” [Online]. Available: https://docs.microsoft.com/en-us/azure/storage/files/storage-files-scale-targets. [Accessed 12 March 2020].