ΠΑΝΕΠΙΣΤΉΜΙΟ ΔΥΤΙΚΉΣ ΜΑΚΕΔΟΝΊΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία

Ενορχήστρωση Πόρων με Χρήση Τεχνικών Μηχανικής Μάθησης

Από τον φοιτητή:

Αποστολάκος Τρύφων

ΑΕΜ: 1137

Επιβλέπων Καθηγητής:

Σαρηγιαννίδης Παναγιώτης

Αναπληρωτής Καθηγητής

Ιούλιος 2021, Κοζάνη

UNIVERSITY OF WESTERN MACEDONIA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Diploma Thesis

Resource Orchestration using Machine Learning Techniques

By:

Apostolakos Tryfon

Supervisor:

Panagiotis Sarigiannidis

Associate Professor

July 2021, Kozani

Περίληψη

Στην παρούσα διπλωματική εργασία μελετώνται οι εφαρμογές μιας νέας τεχνολογίας εικονοποίησης πόρων, οι περιέκτες, και πως μπορούν να ενορχηστρωθούν σε σενάρια εργασιών μηχανική μάθησης. Αρχικά, στην εισαγωγή εξηγούνται εις βάθος οι υπάρχουσες τεχνολογίες εικονοποίησης πόρων, οι εικονικές μηχανές, και στην συνέχεια εισάγεται και εξηγείται η λειτουργία των περιεκτών. Τέλος, συγκρίνονται οι δυο αυτές τεχνολογίες σε όσους κοινούς άξονες έχουν και παρουσιάζονται σύντομα τα πλεονεκτήματα και μειονεκτήματα της καθεμιάς.

Η διπλωματική ξεκινάει με την παρουσίαση της νέας μεθόδου εικονοποίησης πόρων, τους περιέκτες. Στην συνέχεια, αναφέρεται και επεξηγείται ένα ολοκληρωμένο εργαλείο για διαχείριση μοντέλων μηχανική μάθησης, το TensorFlow. Συγκεκριμένα, στο σύστημα που αναπτύσσεται στα πλαίσια της παρούσας, χρησιμοποιείται το TensorFlow σε ένα κατανεμημένο περιβάλλον, επιδεικνύοντας έτσι κυρίως τα πλεονεκτήματα των περιεκτών, μεταξύ των οποίων, η εύκολη ενορχήστρωσή και κλιμάκωσή τους, ανάλογα με τις απαιτήσεις της κάθε εργασίας.

Ο στόχος της παρούσας, είναι η ανάπτυξη ενός συστήματος, το οποίο να μπορεί να επεξεργαστεί μοντέλα μηχανικής μάθησης, από την εκπαίδευση μέχρι την αποθήκευση και την βελτίωση με κατανεμημένο τρόπο, χρησιμοποιώντας ενορχήστρωση περιεκτών. Οι περιέκτες επιλέχθηκαν διότι φέρουν αυξημένες επιδόσεις σε σύγκριση με τις εικονικές μηχανές, ενώ διατηρούν την ευκολία στην μεταφερσιμότητα και την κλιμάκωσή τους. Έτσι, το σύστημα που υλοποιείται έχει επιδόσεις συγκρίσιμες με αυτές ενός φυσικού συστήματος, και δυνατότητες ενορχήστρωσης των εικονικών του πόρων περισσότερες από εκείνες των εικονικών μηχανών.

Στο τρίτο κεφάλαιο, εκπαιδεύονται αρχικά ένα, και στην συνέχεια δυο μοντέλα μηχανικής μάθησης στο σύστημα ταυτόχρονα, καταγράφοντας τις επιδόσεις του εκάστοτε υπολογιστή στο σύστημα. Αφού απεικονιστούν και σχολιαστούν τα αποτελέσματα, συγκρίνονται με ένα σύστημα αναφοράς το οποίο δεν χρησιμοποιεί περιέκτες, και άρα δεν είναι εύκολα κλιμακούμενο, και υπολογίζεται η διαφορά στην απόδοση.

Λέξεις κλειδιά

Εικονοποίηση πόρων, περιέκτες, ενορχήστρωση πόρων, μηχανική μάθηση.

Abstract

This dissertation examines a novel way of resource visualization via containers and orchestration. The method used for simulating a workload is training Machine Learning models. The introduction of this dissertation elaborates on the existing techniques of resource visualization, virtual machines. Sequentially, containers are introduced and elucidated further. Finally, these two technologies are compared on their similarities, and some advantages and disadvantages of each method are presented.

This dissertation starts with a description of containers as a way of resource visualization. Following this description, TensorFlow, a complete machine learning library, is introduced and shortly explained. More specifically, this dissertation aims at developing a distributed testbed, with each computer acting as a TensorFlow worker node. This testbed accentuates the significant benefits of using containers; these benefits include, but are not limited to, ease of orchestration, high scalability and availability, depending on each processes' needs.

This thesis aims at developing a testbed able to handle the complete lifecycle of a machine learning model, from its training and storage on disk to its loading and tuning. All of its lifecycle maintenance will be done distributed, managing the workflow with an appropriate orchestrator software. Containers were selected due to their increased performance compared to virtual machines; while still maintaining their transferability and scalability. The resulting system has performance comparable to a bare-metal system while still having all of the features of virtual machines.

In the third chapter, the testbed is benchmarked on two scenarios; one consisting of a single model and the nodes, and one composed of two models simultaneously. After elaborating on the results, they are compared to a reference system running the same processes natively, not having the advantages of containers. Along with each test case, resource utilization is monitored and visualized of each node in the system. The performance drop is calculated and visualized for each scenario.

Keywords

Virtualization, containers, orchestration, machine learning.

Ευχαριστίες

Θα ήθελα αρχικά να ευχαριστήσω την οικογένειά μου για την υποστήριξη που μου παρείχε κατά την φοίτησή μου.

Ακόμη θα ήθελα να ευχαριστήσω την κοπέλα και τους στενούς μου φίλους, που με υποστήριξαν στην προσπάθειά μου, καθ’ όλη την διάρκεια της φοίτησής μου.

Τέλος, θα ήθελα να ευχαριστήσω τον επιβλέπων κ. Σαρηγιαννίδη, Αναπληρωτή Καθηγητή του Τμήματος, τον κ. Λάγκα Θωμά, Επίκουρο Καθηγητή του Διεθνές Πανεπιστημίου Ελλάδος, καθώς και τους Υποψήφιους Διδάκτορες του Τμήματος Λιατίφη Αθανάσιο και Πλιάτσιο Δημήτριο για την καθοδήγηση τους από την σύλληψη του θέματος έως και την ολοκλήρωση της διπλωματικής.

TABLE OF CONTENTS

Abbreviations

AI	Artificial Intelligence
API	Application Programable Interface
ARP	Address Resolution Protocol
CI/CD	Continuous Integration, Continuous Development/Deployment
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
IPv4	Internet Protocol Version 4
ISA	Instruction Set Architecture
MEC	Mobile Edge Computing
ML	Machine Learning
OS	Operating System
RAM	Random Access Memory
RL	Reinforcement Learning
SDN	Software Defined Networks
TPU	Tensor Processing Units
vCPU	Virtual Central Processing Unit
VM	Virtual Machine
vRAM	Virtual Random Access Memory
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Table of Figures

List of Tables

Chapter One

Introduction

1.1 Virtual machines, a way of virtualizing resources.

Traditionally, users interfaced with a computing system, widely known as a computer, via the Operating System (OS). Thus, the user interacted with the computer's hardware via the operating system's interface (graphical or not).

The primary objective of Virtual Machines (VMs) is to virtualize hardware resources so that the user will be able to program the computer for its desired purpose, not depending on the specifics of the system's hardware [1]. To achieve that purpose, virtual machines abstract physical resources and enable users to access them via a virtual OS, running as an application on the main one. The primary operating system is called host OS, while the OS within the virtual machine is called guest OS. By implementing this abstraction, virtual machines enable users to work on multiple guest operating systems, aside from the guest OS, simultaneously, each for a different purpose and with a different number of resources. This way, users can select how many resources (CPU, RAM, storage) each machine should have, depending on the purpose of each device. An abstract architecture is shown in Figure 1. The aforementioned features increase the availability of the services running on the system, since the failure of one VM does not disrupt the operation of the whole system. Furthermore, VMs enable users to write programs otherwise not compatible with their host, possibly due to a restriction of their operating system. Using VM, users can develop using a device with a different architecture, achieving their desired goal without tweaking their host computer [2], [3].

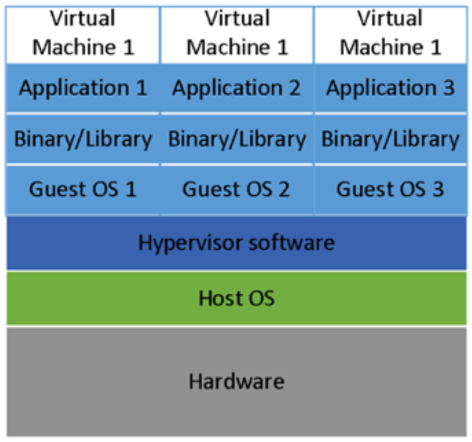


Figure 1 - Virtual Machine Architecture.

Historically, the term VM was first mentioned around 1960-1970. That was followed by an initial spread and adoption of that technology. One of the first instances of virtual machines is in the 1960s; they attempted to time-schedule resource utilization for different tasks [4]. Back in those days, when a user wanted to program a computer to perform a specific job (for example, complex mathematical calculations), they needed to use the provided interface, commonly the OS, to do so. As deduced, the user should know the underlying hardware specifications of the system to program it correctly [5]. To name a few, IBM System 360 and IBM System 370 are some systems of that era [6], [7].

It was only later adopted, around 2000-2005, when the technology matured enough, and processors, memory, and storage devices could handle the amount of multitasking required. Between 2000 and 2010, some major free software suites were released on the internet, enabling everyday users (not only programmers) to use virtual machines at home, with commercial off-the-shelf hardware[1,2]. Nowadays, the virtual machine has a large chunk of the market share, as far as servers are concerned. Specifically, 92% of companies use virtualization on their servers as of 2020 [8].

Despite the multiple advantages of this new technology, there also exist some disadvantages. Firstly, the most important one being the performance impact on the host OS, also called overhead [9]. Consequently, this performance reduction also affects the guest machines, compared to the same machine running on bare-metal. For example, an Ubuntu VM has diminished performance compare to an actual Ubuntu host running on real hardware. This reduction in performance occurs because virtual machines set up an entire OS stack containing the kernel, Instruction Set Architecture (ISA), drivers, libraries, Graphical User Interfaces (GUIs), file system. The decreased performance becomes even more noticeable the more complex the desired task gets. Additionally, every VM is always at risk of getting infected with malware from the internet (as a typical machine would) while also from the host machine; having one more attack vector needs to be taken care of to ensure the virtual machine's healthy state.

In order for VMs to work, the host OS needs to install specialized software. This software is responsible for creating, managing, and scheduling tasks regarding virtual machines. This software is called a hypervisor and can be further identified as type-1 or type -2 hypervisors. The type -1 hypervisor runs on bare metal and manages all the virtual machines it creates. The type-2 hypervisor runs on an existing OS. The critical difference is that type-1 hypervisors have full access to the hardware since they are the closest to it, while type-2 request access to the hardware from the OS they are installed. Most software aimed at consumers or home users is type-2 since home users want a host OS to use their computer. On the other hand, type-1 are aimed at enterprise applications. From now on, when mentioning hypervisors, type-2 hypervisors are implied.

[1] https://www.virtualbox.org/

[2] https://www.vmware.com/products/workstation-pro.html

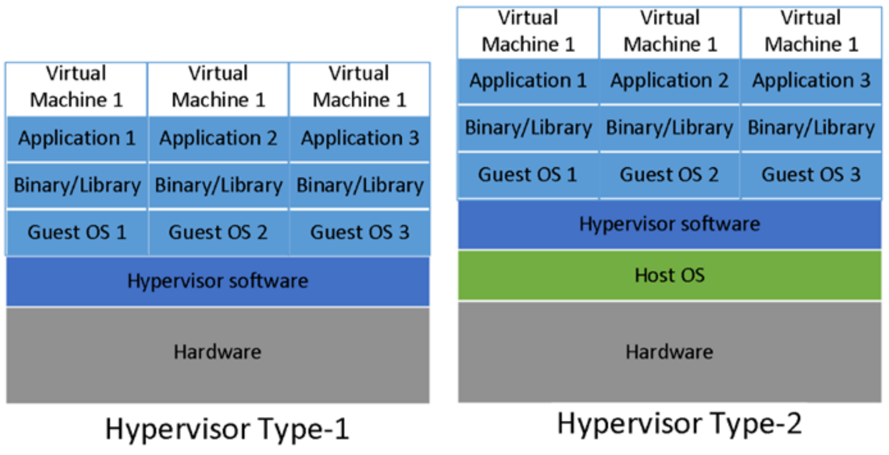


Figure 2 - Hypervisor types.

Hypervisors, as shown in Figure 2, act as a middleman for the hardware and the guest OS. However, hypervisors, being applications on the host OS, need to run on the userspace, a dedicated virtual memory segment for storing non-kernel information, since kernel processes strictly must run on an isolated memory segment for security reasons. Thus, when VMs request resources, this request has to go through the guest OS and its kernel, and the hypervisor, in turn, must request from the host OS and its kernel the resources. For this to be achievable, a portion of RAM has to be reserved. This portion is increasing as the CPU for each VM increases. This reservation is called overhead and affects every virtual machine to this day. Table 1 summarizes the aforementioned remarks.

Type-1 Hypervisors	**Type-2 Hypervisors**
It runs on bare metal, has no OS as a middleman.	Are installed and run inside the OS.
Have direct access to the underlying hardware, thus creating VMs with direct access to it.	VMs created from type-2 have indirect access to the hardware.
Increased speed and security since no host OS is present to be exploited.	Lower speed and security since OS can still be vulnerable.
It should be installed on systems with appropriate hardware and require more configuring.	Easier, install-and-run, supporting the vast majority of commercially available hardware.
Have higher scalability and availability compared to type-2.	Due to OS restrictions, it cannot scale that efficiently.

Table 1 - Hypervisor types.

1.2 Containers, a novel way of virtualizing resources.

In the last decade, a new virtualization method has been under development, aiming to address some of the VM issues. Its purpose is not to replace virtual machines entirely, but to enable users to virtualize resources in a different, less isolated, but much more efficient approach. Though containers are fundamentally different, they share some similarities with virtual machines. With this novel method of virtualizing resources, the guest subsystem, now called a container, shares the same kernel with the host operating system [10]. The resulting container is a semi-isolated system since it has the same kernel and operating system as the host. However, each container creates a new filesystem and a network stack. The significant difference from virtual machines is the lack of isolation between the host and guest OS and kernel. This different approach in their architecture is shown in Figure 3.

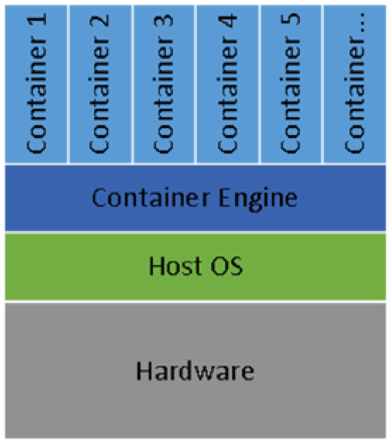


Figure 3 - Container Architecture.

Containers achieve minimal overhead on the system and increase the overall performance, while decreasing the impact on the host OS. However, it is noteworthy that containers' purpose is not just to abstract the underlying hardware from the user to allow them to make their goals more attainable. As their names suggest, containers enable users to easily compile all their source code or libraries into one or more easily manageable, highly scalable and available components. The containers can be deployed, moved, and deleted dynamically and promptly. Users can pack a whole application with its dependencies and libraries as a service and deploy it to many replicas within containers. In addition, users can also decouple the libraries or part of the service, as a whole, and deploy it distributed, the pieces of which are called microservices. Either way, each component is running on a container, which is an isolated environment with a dedicated file system and network stack, sharing the host OS and kernel. As deduced from their purpose, containers are created and removed dynamically and are designed to be easily replaceable.

Summarizing, the leverage of containers can provide several advantages. Firstly, since containers share the underlying OS and kernel, there is no need to boot an entire OS with drivers and libraries. As a result, containers have negligible bootup and shutdown times. Moreover, they usually occupy less disk space, compared to a traditional VM, which can take up a couple of gigabytes. Another significant advantage of containers is the lack of memory overhead on the system. As containers share the same kernel, any request for hardware resources does not have to go through two kernels to be accepted. The container runtime software (the equivalent of the hypervisor software in virtual machines) is usually lightweight with no Graphical User Interface (GUI), thus improving the overall performance of the containers. Each container's prototype contains the instructions and the purpose of the container, and what to run when started. It is stored in an image, similar to .iso files on virtual machines. The only difference is that these images are minimal in size.

1.3 Comparison between virtual machines and containers.

Table 2 contains a summary of the significant advantages of each virtualization method, namely VMs and containers. On the other hand, Table 3 summarizes the corresponding disadvantages [11]-[14]. It is imperative to understand that, due to their fundamental differences in their architecture and their purpose, containers cannot be compared on their entirety of features to VMs [11]. Also, containers were not created to replace or make virtual machines better; instead, they mainly aim at abstracting a problem to solve it more easily. However, both technologies share some similarities. In some cases, a user may have to decide which virtualization method should choose, since both VMs and containers can perform the same tasks. For these reasons, the following comparison is presented [12], [13].

Advantages of Virtual Machines	**Advantages of Containers**
More robust isolation enhanced security.	Very portable, due to their small size on disk.
Ability to create a virtual machine with a different OS architecture of the host (for example, one can make a Unix VM on a Linux host).	Highly portable, since the only thing required to be transferred to another host is an online repository (or a physical medium) and a container runtime engine.
Ideal for a monolithic application that is required to run on one specific host.	Very light on resources, especially RAM, since there is little to no overhead and a small footprint on storage space.
	It can be booted and shut down in a matter of seconds.
	Their CPU performance is closer to bare metal compared to virtual machines.

Table 2 - Advantages of each virtualization method.

Disadvantages of Virtual Machines	**Disadvantages of Containers**
Significant footprint on storage, RAM, and CPU performance. The existence of overhead	Fundamental lack of security due to the lack of isolation between kernel space and userspace

further hinders their performance.	on the host and guest.
Their size on disk can be many gigabytes.	Containers must use the host OS kernel, thus depending on the host OS architecture. For example, cannot create a Windows OS container in a Linux host OS.
Boot up time and shutdown time dependent on the disk speeds.	Harder to store and transfer files from and to host OS. Containers were created to be ephemeral, so extra effort needs to be made to create persistent volumes.

Table 3 - Disadvantages of each virtualization method.

1.4 Overlay networks, a way of virtualizing networks.

In recent years, to overcome specific difficulties of traditional networks, new methods of implementing networks were developed [14]–[16]. Nowadays, it is common for a set of services to run on a network with no direct physical infrastructure, broadly called a Software Defined Network (SDN) [17]–[23]. The idea behind an SDN is to abstract the physical layer of a network and program the desired behaviour on the network. This topic is relevant in containers and orchestration since when a network within a swarm is created, the architect must decide on its driver. The most common driver is called "overlay." An overlay Software Defined Network is created, and the desired services are attached and operate on it.

Firstly, the abstract architecture of SDNs is visualized in Figure 4. It should be noted that the architecture depicted in Figure 4 is not the complete architecture. Some components, such as the Management Layer, that span across all other layers, are not shown since they are not in the scope of this dissertation. As shown in Figure 4, SDNs consist of multiple layers, each having its responsibilities within its scope. Most notably, distinct layers are the Application, Control, and Infrastructure layers. Starting from the bottom, the Data Plane contains the underlying hardware infrastructure such as switches. The Control layer contains a running instance of the software that enables the programming of the underlying infrastructure. This software is called an SDN Controller and is also responsible for translating high-level requests, created from the Application Layer, via the Northbound Application Programming Interface (API), into lower-level commands of the protocols that are supported by the Data Plane. Lastly, the Application Layer commonly contains the high-lever services or applications that communicate with the infrastructure via the SDN Controller [24].

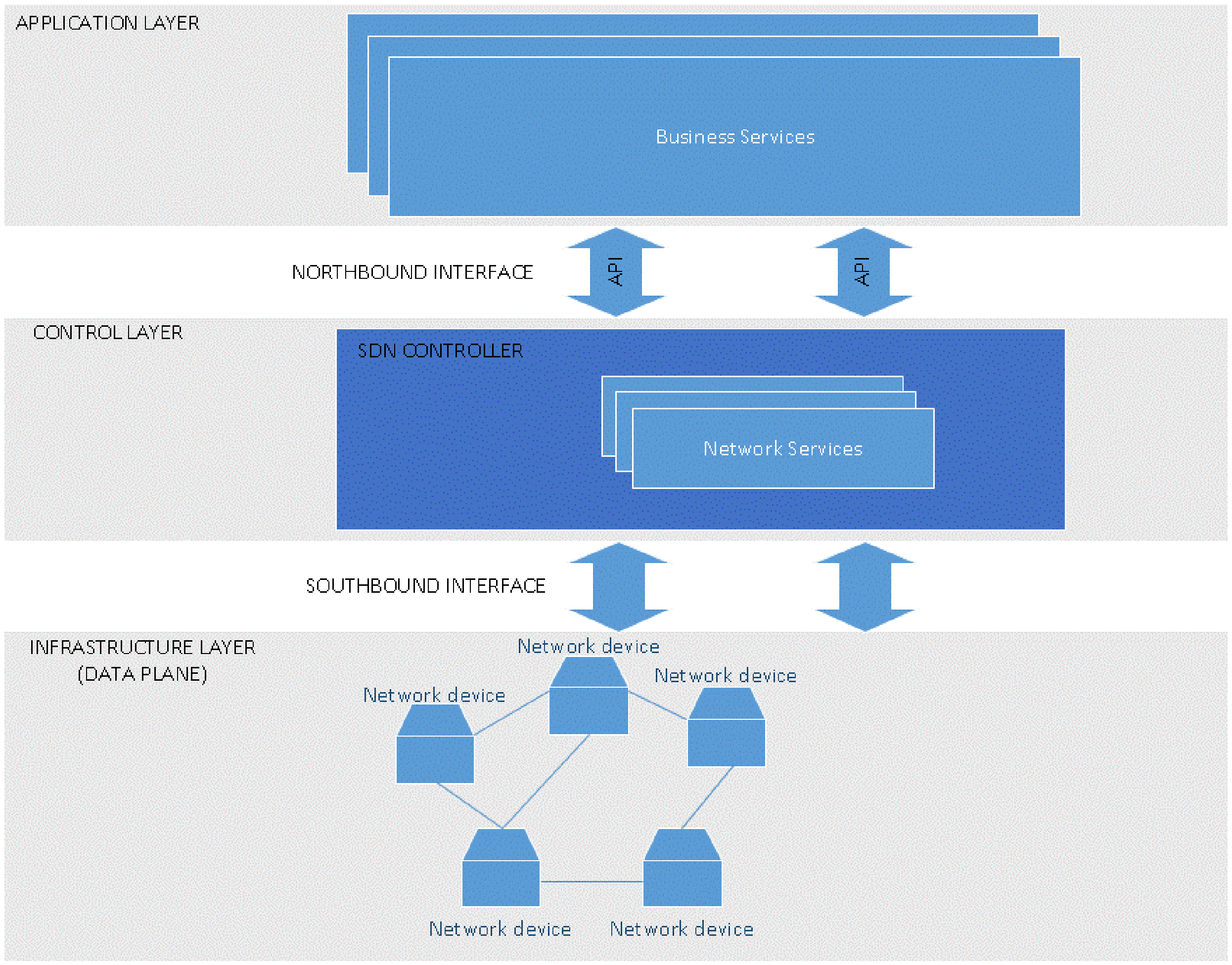


Figure 4 - SDN Architecture.

Next, the incorporation of Overlay networks in Docker Swarm mode is examined. Swarm mode allows the rapid creation of disposable networks. Three drivers are available when creating a network, the default being bridged. The other option is an overlay driver, and the last is a custom network driver. The overlay option that creates an SDN Overlay network is selected and further explained.

Figure 5 depicts an abstracted architecture of an Overlay network. As can be deduced, Overlay networks are a way of virtualizing networks and giving them a programmable logic. Overlay networks hide the physical infrastructure by creating layers of abstractions and installing software called acting as an "agent" that manages the created SDN Overlay.

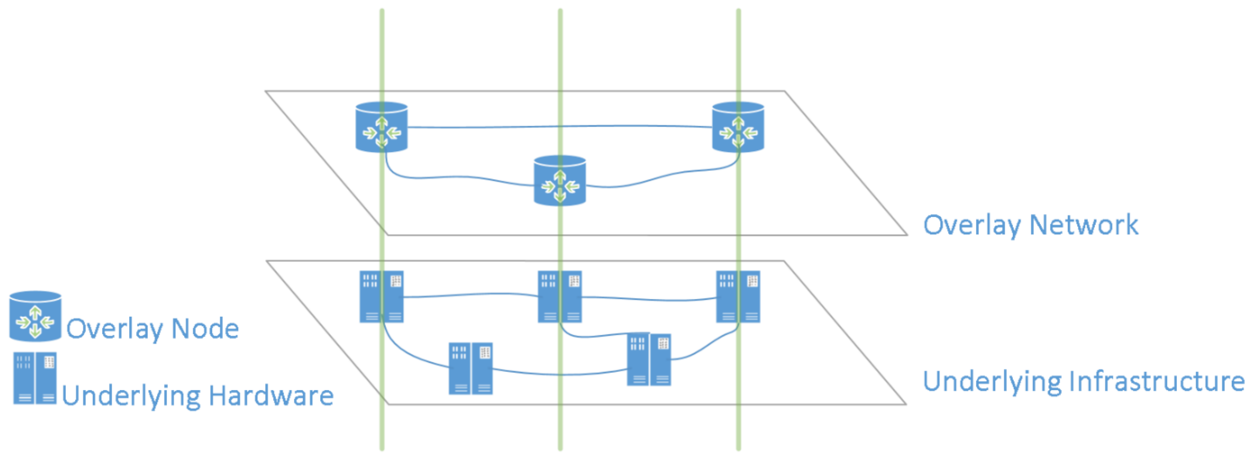


Figure 5 - Abstract architecture of an Overlay network.

Docker Engine further hides this architecture and simplifies the use of SDN Overlays. A stack consists of many services in Docker swarm, where each service can be a business application, a database, a website; each service can be connected to one or more networks. Based on the container architecture, as shown in Figure 6, a set of Swarm workers as three physical machines connected to a cluster in Docker Swarm mode can be seen. Each worker has a set of running services, with each service containing a different number of replicas of a container. Each service has a different color, while services 1, 3, and 5 are considered to be of high importance. These services, for example, could be connected to an overlay network. Thus, the containers these services would spawn would also be connected to an overlay network. Within that network, services 2, 4, and 6 are not known and cannot be reached. This way, within the overlay created for services 1, 3, and 5, a graph of the network is shown in Figure 7.

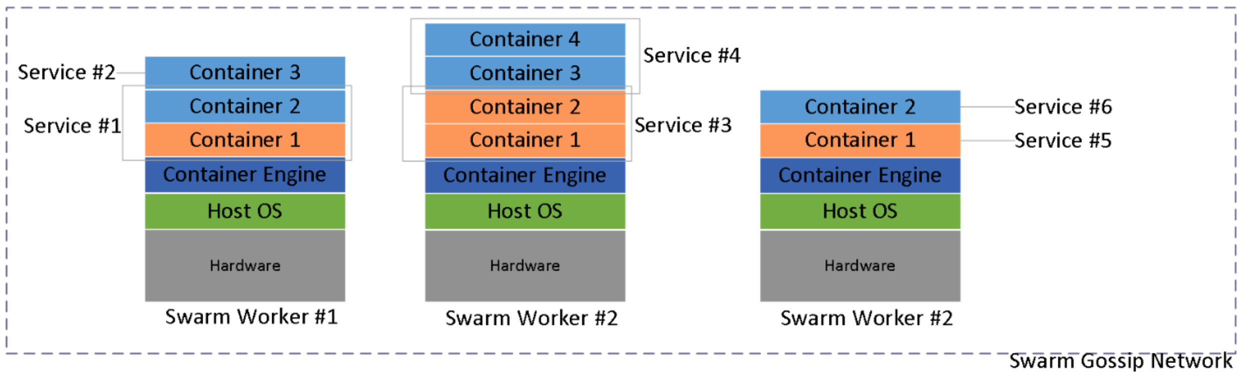


Figure 6 - A cluster with three nodes and some running services.

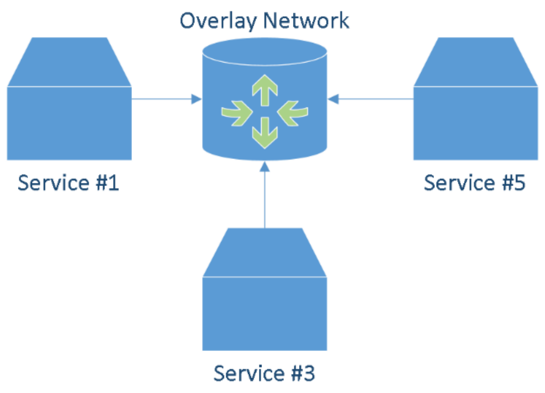


Figure 7 - Inside a Swarm Overlay network.

Docker Swarm allows the creation of a cluster, in which many computers can join as nodes. Every node can have two distinct roles, those being a worker or a manager node. Worker nodes simply take on a workload and process it, while manager nodes have some added responsibilities. These include maintaining the stability of the cluster, ensuring services are running where they should and are responsive, and, lastly, serving HTTP APIs.

Manager nodes use a version of Raft Consensus Protocol [25]. This algorithm enables the high availability of the cluster through fault tolerance. Fault tolerance allows for the existence of many managers in a swarm in case one or more fails. This way, even if some managers fail, the cluster will elect another manager, and its availability will not change. To achieve this functionality, docker uses two distinct networks, listening to different ports. One is for managers to communicate and maintain the state of the cluster; one for the worker nodes to exchange information regarding their running services. The first is called Raft Consensus Group, while the workers' network is called the Gossip network. That is why in Figure 6, the Gossip network encloses all the workers [26].

1.5 Tensorflow, a complete Machine Learning Python library.

Tensorflow[3] is a machine learning library built for the Python[4] programming language [27]–[33]. At the time of writing this thesis, the current version is 2.4.1, which is officially supported on python versions 3.6, 3.7, and 3.8. Tensorflow provides all the needed tools for creating, training, storing, loading, and modifying machine learning models[34]–[38].

A typical TensorFlow workflow is shown in Figure 8. Firstly, the dataset for the specific problem must be acquired. This dataset can further be preprocessed for enhanced training. Such actions could include splitting the dataset into smaller portions, one for training and one for testing.

[3] https://www.tensorflow.org/
[4] https://www.python.org/

The utilized model can be an entirely new model built from scratch or an existing model that has been tuned accordingly. TensorFlow has its repository for trained models[5]. At this point, it should be noted that before TensorFlow 2.0, some problems could be solved with TensorFlow Estimators. However, as per the documentation, new code should ideally be written with the tf.keras API instead. After the model and the dataset are established, a distribution strategy should be selected.

Tensorflow has a built-in API allowing the training of models across different host machines, each with its dedicated resources. The name of its API is tf.distribute.Strategy, containing various types of training strategies, depending on the needs of each scenario. Tensorflow supports Tensor Processing Units (TPUs), as well as Graphics Processing Unit (GPUs) and CPUs. The first has the best performance per watt [39], [40].

The aim of this thesis is not to create a novel machine-learning algorithm or optimize an existing one. Instead, it leverages machine learning workflows to develop a novel, highly performing, and easily scalable (horizontally or vertically) testbed that can orchestrate machine learning workloads using containerization. The dataset and the model trained in the following chapters have been studied extensively, are part of the official documentation of Tensorflow, and are used as an introduction to machine learning. Tensorflow requires pip[6] to be installed and updated. Finally, Ubuntu Desktop 20.04[7] has been selected as the host OS.

[5] https://www.tensorflow.org/hub
[6] https://pypi.org/project/pip/
[7] https://ubuntu.com/download/desktop

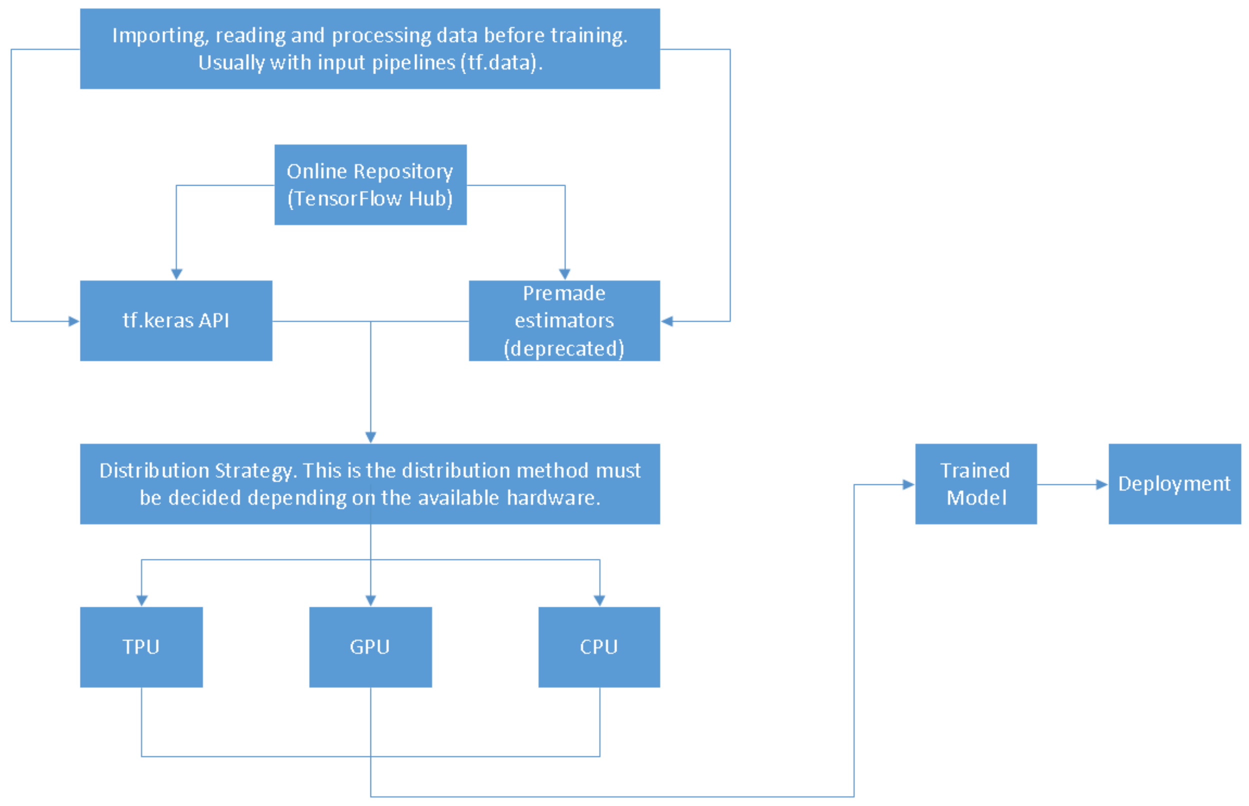

Importing, reading and processing data before training. Usually with input pipelines (tf.data).
Online Repository (TensorFlow Hub)
tf.keras API
Premade estimators (deprecated)
Distribution Strategy. This is the distribution method must be decided depending on the available hardware.
TPU
GPU
CPU
Trained Model
Deployment

Figure 8 - Typical TensorFlow workflow.

1.6 Aim of this Thesis and a short description of the testbed.

This thesis aims to create a novel testbed able to handle machine learning workloads and train models in a distributed way. By leveraging containers, the testbed can scale vertically and horizontally. Vertical scaling implies changing the size of each container CPU, RAM, and disk. In the case of vertical scaling, the number of available instances is modified. In addition to these, the testbed will maintain acceptable performance, near bare-metal, yet still be highly available and easy to orchestrate.

In the scope of this dissertation, multiple files have been developed aiming to assist with the orchestration of the cluster. Namely, the files created from scratch are *parse_ips.py*, *start_cluster.sh*, *docker-compose.yml*, *Dockerfile*, *mystats1_.csv*, *run_workload.sh*. These files were created to automate orchestrating tasks, such as creating, updating, and monitoring the cluster. In addition to these, cAdvisor and docker-stats are utilized to debug issues during the testbed development.

Three virtual hosts are deployed, with each one having two virtual CPUs (vCPUs) and 4GB of virtual RAM (vRAM). All traffic is permitted (inbound and outbound) between these hosts on all ports. Each host, as mentioned above, is running the latest version of Ubuntu Desktop 20.04. Each host has the latest Docker Engine[8] and Docker Compose[9] installed. Each host can support multiple containers, while each container is based on a custom TensorFlow image. The custom image developed adds some more tools to the existing TensorFlow image found in their dockerhub[10] repository.

[8] https://docs.docker.com/engine/
[9] https://docs.docker.com/compose/install/
[10] https://hub.docker.com/r/tensorflow/tensorflow

Chapter Two

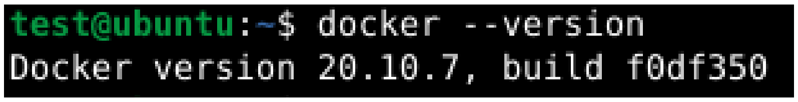
2.1 Installing and configuring Docker engine and verifying Swarm mode operation.

Firstly, the latest version of Docker Engine is installed on all three hosts. Detailed instruction can be found on the official documentation in the following link:

https://docs.docker.com/engine/install/ubuntu/

The testbed uses the apt version of Docker Engine with root privileges given to the daemon, as described in the following link. After installing docker, its correct installation can be confirmed with the command `docker --version`. In Figure 9, the result of that command being run in one of the hosts is seen.

https://docs.docker.com/engine/install/linux-postinstall/



```
test@ubuntu:~$ docker --version
Docker version 20.10.7, build f0df350
```

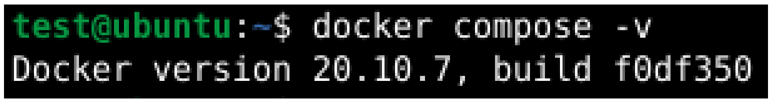
Figure 9 - Docker version.

Next, docker-compose should be installed and verified with the following command.

```
docker-compose -v
```

A successful run of a version command for docker-compose is shown in Figure 10.

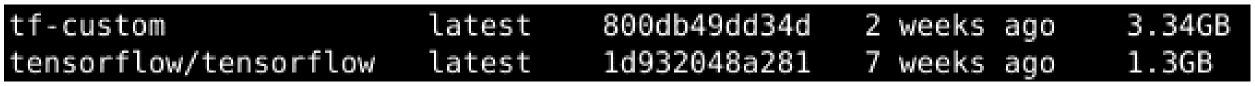
https://docs.docker.com/compose/install/



```
test@ubuntu:~$ docker compose -v
Docker version 20.10.7, build f0df350
```

Figure 10 - docker-compose version.

This project is based on the TensorFlow image, which can be found in the following dockerhub repository. To pull this image, `docker pull TensorFlow/TensorFlow` is issued on each host. The custom TensorFlow image that was created to fit the needs of this testbed is called tf-custom and is shown alongside the TensorFlow image as shown in Figure 11.



```
tf-custom                  latest    800db49dd34d   2 weeks ago    3.34GB
tensorflow/tensorflow      latest    1d932048a281   7 weeks ago    1.3GB
```

Figure 11 - Available Docker images.

The base TensorFlow image has been cloned, while the installation of Docker Engine and docker-compose have been verified. The last step is to create a cluster containing the three hosts discussed earlier. A cluster is defined as a set of nodes (hosts) in swarm mode of docker, running Docker Engine. In a cluster, the first to create it is declared a manager, and the others to join are called workers.

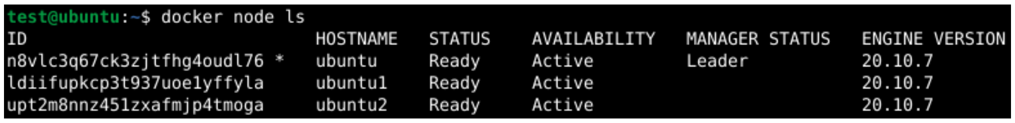
To create a swarm with one node as the manager, the following command is issued to one host.

```
docker swarm init --advertise-addr [IPV4_ADDR]
```

```
docker swarm join \
--token SWMTKN-1-3pu6hszjas19xyp7ghgosyx9k8atbfcr8p2is99znpy26u2lkl-
1awxwuwd3z9j1z3puu7rcgdbx \
[IPv4_ADDR]:2377
```

The init command returns a join token and the appropriate command, which in turn is run to the other two hosts, allowing them to join the cluster as workers. After running the join command in the other two hosts, a three-node cluster has been successfully created. To get some basic information about the nodes and their overall status, `docker node ls` can be issued and get the results shown in Figure 12.

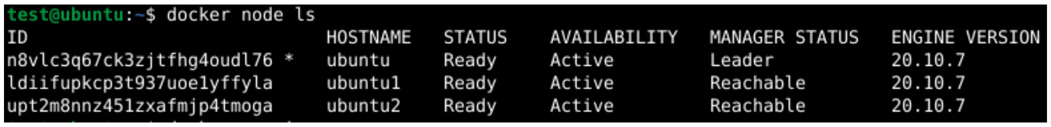
Optionally, running `docker node promote ubuntu1`, and `Docker node promote ubuntu2`



```
test@ubuntu:~$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
n8vlc3q67ck3zjtfhg4oudl76 *   ubuntu     Ready     Active         Leader           20.10.7
ldiifupkcp3t937uoe1yffyla     ubuntu1    Ready     Active                          20.10.7
upt2m8nnz451zxafmjp4tmoga     ubuntu2    Ready     Active                          20.10.7
```

Figure 12 - Available nodes before promotion to managers.

promotes all the nodes to managers. `docker node ls` returns what is shown in Figure 13.



```
test@ubuntu:~$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
n8vlc3q67ck3zjtfhg4oudl76 *   ubuntu     Ready     Active         Leader           20.10.7
ldiifupkcp3t937uoelyffyla     ubuntu1    Ready     Active         Reachable        20.10.7
upt2m8nnz451zxafmjp4tmoga     ubuntu2    Ready     Active         Reachable        20.10.7
```

Figure 13 - Available nodes after promotion to managers.

The next step is to construct a Dockerfile containing all the changes made and compiled to a new image of TensorFlow. The contents are shown below.

```
#Dockerfile
FROM tensorflow/tensorflow
#base image of tensorflow
RUN mkdir script
#script is created under /, being bind mount in the yml file
RUN apt install nmap -y \
    && python -m pip install pip==21.0.1 \
    && pip install -U setuptools \
    && pip install -U wheel \
    && pip install -U numpy \
    && pip install -U matplotlib \
    && pip install -U pandas \
    && pip install -U scipy \
    && pip install -U scikit-learn \
    && pip install -U tensorflow \
    && pip install -U python-mnist \
    && pip install -U Keras
#updating these tools and installing some
#not all of the are needed, however in future update might be utilized, so are ke
pt
RUN pip install python3-nmap \
    && pip install -U psutil
#psutil is a performance monitor tool
WORKDIR /script
#change dir and start wrapper script
CMD /script/start_cluster.sh
```

The base image, as mentioned, is the latest version of TensorFlow/TensorFlow. Then, a directory under root is created. It is named /script, as seen in the first RUN command, and in the subsequent two RUN commands, some additional tools are installed and updated. Finally, the current directory is changed to /script, and the wrapper script that creates the cluster is executed in the final CMD command.

2.2 Creating a Swarm cluster with three hosts, able to process machine learning models.

In this chapter, the orchestration of the cluster is documented, As mentioned in the previous section, containers are a novel way of virtualizing resources more efficiently by breaking down rather large monolithic components into smaller entities, called microservices. Since containers are ephemeral and easily replaceable, their lifecycle from the inception of the cluster to its deployment, its update, and scaling should be closely monitored and strictly defined. Container orchestration is the practice of automating most of the aforementioned tasks and further enabling CI/CD pipelines. Many orchestration tools exist nowadays, including Kubernetes[11], Docker Swarm[12] Apache Mesos[13], RedHat OpenShift[14].

In this dissertation, Docker Swarm is selected for having higher overall performance due to its integration into the Docker engine. Orchestration occurs when creating the cluster description and consecutively when adding a second workload as another cluster. In addition to that, orchestration includes scaling the workload. For example, should more resources become available, the updating of the cluster towards utilizing more resources is considered an orchestration task.

Up to this point, a custom TensorFlow image tailored to the needs of the testbed has been created. The specifications of this image have been discussed in the aforementioned Dockerfile. The cluster has also been initialized, and the node connectivity has been verified. There are two methods to deploy an application to the cluster and orchestrate it, namely the imperative and declarative methods. Using the imperative method, the corresponding commands should be entered into the terminal. On the other hand, using the declarative method, the user provides a description of the cluster's desired state, while the orchestrator carries out the required operations to shift the cluster to that state. The latter method is also called self-healing and is generally the preferred way of orchestrating a cluster and is implemented in this testbed, using a docker-compose.yml file.

YAML[15] is a way of describing the desired state by introducing services. Swarm services are entities containing information regarding the behaviour, networking, file structure, and runtime of a set of containers. Each service effectively is a set of replicate containers with the same description. Each service can be based on a different image. For example, service A could be a

[11] https://kubernetes.io/
[12] https://docs.docker.com/engine/swarm/
[13] https://mesos.apache.org/
[14] https://www.openshift.com/
[15] https://yaml.org/

database, while service B a web server. These descriptions are written onto YAML files that can be passed to docker stack and deployed to a swarm. Docker stack deploy disregards build instructions, so it helps test the system with docker-compose since the same YAML file can be passed to docker stack.

The contents of docker-compose.yml are shown. This file contains both build instructions for the images and a description of services passed to both docker-compose and docker stack for deployment. Firstly, three similar services are created, with the only difference being that each service (and its replicas) is bound to spawn containers on one of the three nodes. This strategy is implemented to maximize the utilization of the resources while controlling the scale of the testbed more easily. If one service with three replicas were used, it would be harder to manage the networking of the containers that would be created. Each service uses the tf-custom image, and it builds it; however, the build commands are skipped when deploying to a stack. In addition, it mounts the script folder created in the Dockerfile to a script folder in the current directory of the node. All services are then connected to a custom overlay network called clust_net, which is specified at the end of the file. Finally, each service is ordered to spawn containers only to one of the nodes.

```yaml
#docker_compose.yml
version: "3.9"
services:
#late version 3 is used for futureproofing and simply because it works
  tf-custom1_0:
    image: tf-custom
    build: .
    #build instructions are ignored by swarm deploy
    volumes:
        - ./script/:/script/
    networks:
      clust_net:
    deploy:
      placement:
        constraints: [node.hostname == ubuntu]
        #each service's containers ought to run onto a different host
        #for easier debugging, management and scalability

  tf-custom1_1:
    image: tf-custom
    build: .
    volumes:
        - ./script/:/script/
    networks:
      clust_net:
```

```
    deploy:
      placement:
        constraints: [node.hostname == ubuntu1]

  tf-custom1_2:
    image: tf-custom
    build: .
    volumes:
        - ./script/:/script/
    networks:
      clust_net:
    deploy:
      placement:
        constraints: [node.hostname == ubuntu2]



networks:
  clust_net:
    driver: overlay
    #the network is fundamentally a sdn network giving many security benefits
    #in addition to the layer of abstraction, it introduces
    ipam:
      config:
        - subnet: 192.168.0.0/24
        #we define the subnetwork to make scanning with nmap easier
```

2.3 Configuring correct IPv4 addressing in the cluster.

One of the challenges of the testbed is setting up the networking of the containers dynamically. As containers are created at runtime, their Internet Protocol (IP) addresses cannot be predefined. To address this issue, each container, at startup, performs a fast one-port Nmap[16] scan and saves the active hosts to an XML file. This file is parsed in a python function. Based on the scan results, the hosts in the same network are passed on to the TensorFlow strategy for the training to occur. Of note, a potential alternative would be to perform an arp table scan or create another container to manage all the networking. Arp is not suitable because arp tables refresh dynamically and cannot be relied upon. Additionally, creating a container dedicated to IP allocation would be resource-intensive and highly inefficient.

[16] https://nmap.org/

The fast execution of the Nmap scan is ensured by only scanning one port on the subnetwork of each potential host, with the scan taking about 3-4 seconds to finish. The port used is 4789 since both managers and workers use it for the overlay network traffic[17]. Firstly, each host scans the subnetwork. Sequentially, each result is saved to a different XML and parsed by a copy of the python script. The scan is performed in the *start_cluster.sh* file, the contents of which are shown below.

```bash
#!/bin/bash
#start_cluster.sh
echo "Starting Nmap scan for docker-engine compatible hosts."
rm -r keras-model/ >/dev/null 2>&1
#remove old model if it exists, and silence all outputs, stderr and stdout
rm ips.xml >/dev/null 2>&1
#remove old xml of hosts if it exists, and silence all outputs, stderr and stdout

sleep 20
#we wait 10s before starting the scan, to ensure all hosts are up.
nmap -p 4789 192.168.0.0/24 -oX ips.xml
#we port scan the 192.168.0.0/24 subnet and save the output to an xml file
echo "Starting cluster."

python main.py
```

The first rm lines remove old residual data from potential previous tests of the cluster. Then the script waits 20 seconds for the containers to start; each container requires approximately 2-3 seconds, and they begin almost simultaneously. Twenty seconds was chosen to err on the side of caution. Then the scan is performed in 3-4 seconds, with each output saved to the file ips.xml. Finally, the main.py script starts the parsing of the XML and then the training.

Parsing takes place in a function defined in *parse_ips.py*. This function first opens the file ips.xml, starts reading it from its root, and searches for tags containing hosts. After finding a host tag, it goes a level deeper and searches for a non-empty hostname tag. If it reads non-empty hostname tags, it saves the IP address of the tag in a temporary variable. Finally, it creates the list with the nodes alive in the network and returns it. The code is shown below.

```python
#parse_ips.py
import xml.etree.ElementTree as ET

def parse_ips():
    #open file and start iterating from root
    mytree = ET.parse('ips.xml')
    myroot = mytree.getroot()
```

[17] https://docs.docker.com/engine/swarm/swarm-tutorial/#open-protocols-and-ports-between-the-hosts

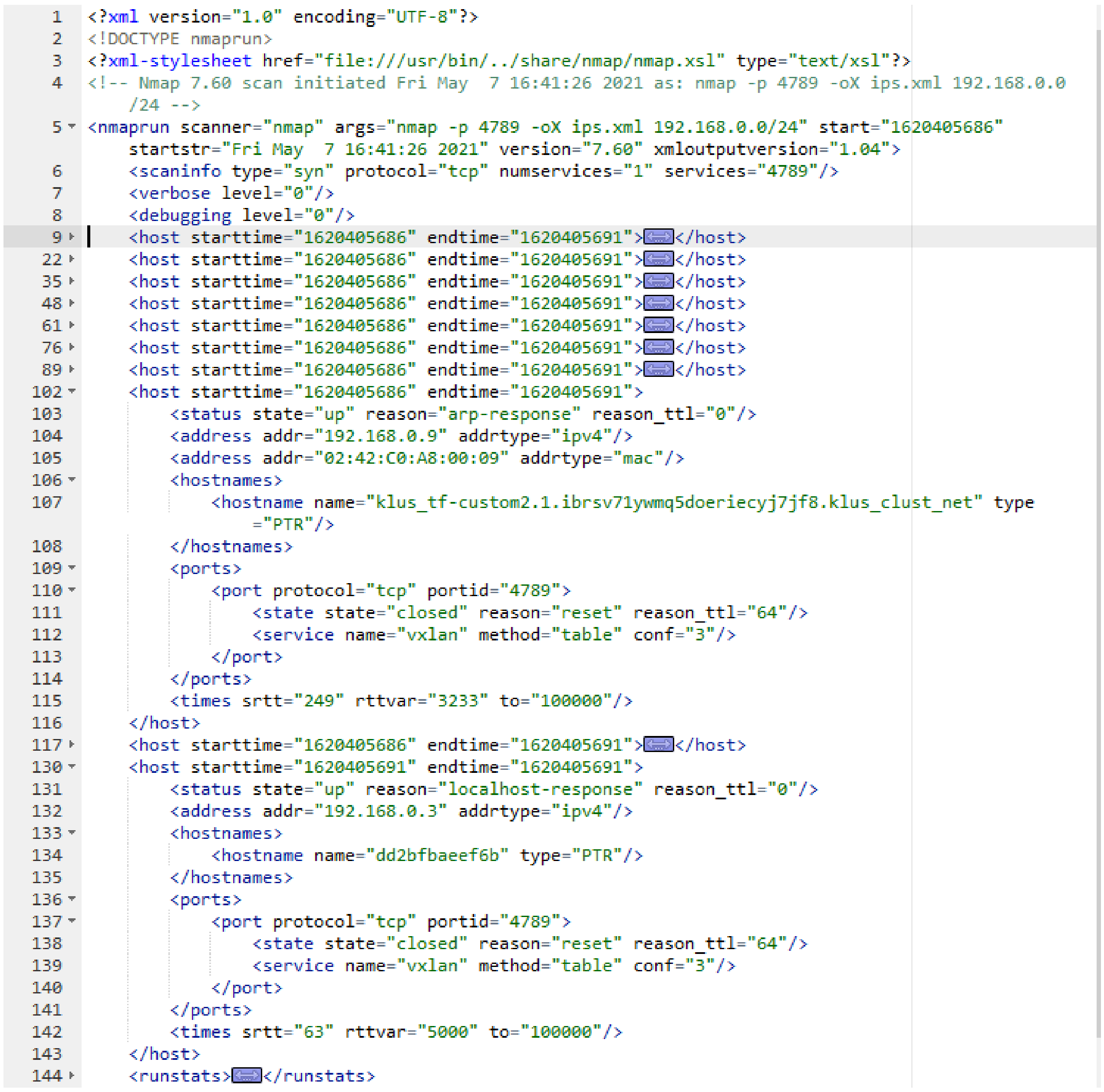
```
    possible_ips=list()
    final_ips=list()
    hosts_found=list()
    #initialize misc variables
    cnt=0
    #cnt will be used to save the index of each host found independently of the o
utside for's
    for i in myroot.findall("host"):
        #take each host
        for j in i:
            if j.tag == "address":
                #for each address that is found, save it at the table possible_ip
s
                possible_ips.append(j.attrib["addr"])
#
#instead of deleting each mac separately, I could possibly only save IPS, modifyi
ng 17-20 lines to check if the type of addr is ipv4 and not mac
#

            if j.tag == "hostnames":
                for k in j:
                    #when we find a non-
empty hostnames body of that tag, we have found a live host
                    #saving its index to the hosts found
                    hosts_found.append(cnt)
        cnt+=1
    print(hosts_found)
    del possible_ips[1::2]

    #since for each host there corresponds one IP and one MAC, we delete every se
cond element of possible_ips
    #to rid it of MACs
    for i in hosts_found:
        final_ips.append(possible_ips[i])
    #finally we the index of every live host, to get its corresponding IP saving
it to final_ips
    #and returing it
    #print(final_ips)

    return final_ips
```

As shown in Figure 13, *ips.xml* contains various hosts, some of which have been expanded. Each host is described with a host tag, and inside contains multiple other tags. The hosts that contain a non-empty body on the hostnames tag are the nodes that are parsed and saved. In Figure 14, two hosts can be seen, with 192.168.0.9 and 192.168.0.3 as their respective IPs.

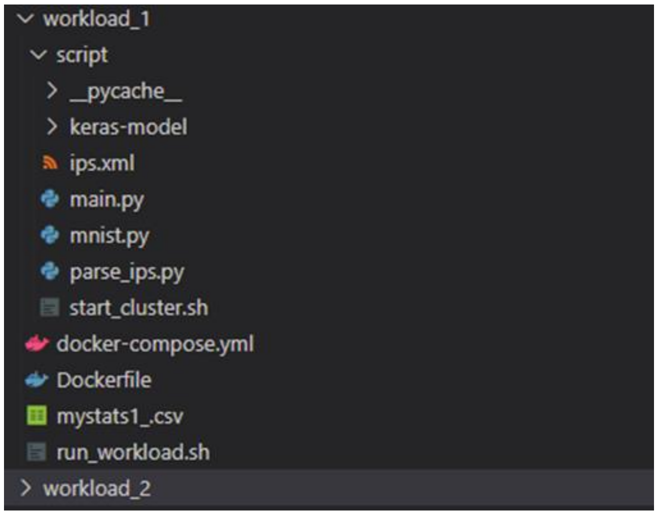


```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE nmaprun>
<?xml-stylesheet href="file:///usr/bin/../share/nmap/nmap.xsl" type="text/xsl"?>
<!-- Nmap 7.60 scan initiated Fri May  7 16:41:26 2021 as: nmap -p 4789 -oX ips.xml 192.168.0.0/24 -->
<nmaprun scanner="nmap" args="nmap -p 4789 -oX ips.xml 192.168.0.0/24" start="1620405686" startstr="Fri May  7 16:41:26 2021" version="7.60" xmloutputversion="1.04">
    <scaninfo type="syn" protocol="tcp" numservices="1" services="4789"/>
    <verbose level="0"/>
    <debugging level="0"/>
    <host starttime="1620405686" endtime="1620405691"></host>
    <host starttime="1620405686" endtime="1620405691"></host>
    <host starttime="1620405686" endtime="1620405691"></host>
    <host starttime="1620405686" endtime="1620405691"></host>
    <host starttime="1620405686" endtime="1620405691"></host>
    <host starttime="1620405686" endtime="1620405691"></host>
    <host starttime="1620405686" endtime="1620405691"></host>
    <host starttime="1620405686" endtime="1620405691">
        <status state="up" reason="arp-response" reason_ttl="0"/>
        <address addr="192.168.0.9" addrtype="ipv4"/>
        <address addr="02:42:C0:A8:00:09" addrtype="mac"/>
        <hostnames>
            <hostname name="klus_tf-custom2.1.ibrsv71ywmq5doeriecyj7jf8.klus_clust_net" type="PTR"/>
        </hostnames>
        <ports>
            <port protocol="tcp" portid="4789">
                <state state="closed" reason="reset" reason_ttl="64"/>
                <service name="vxlan" method="table" conf="3"/>
            </port>
        </ports>
        <times srtt="249" rttvar="3233" to="100000"/>
    </host>
    <host starttime="1620405686" endtime="1620405691"></host>
    <host starttime="1620405691" endtime="1620405691">
        <status state="up" reason="localhost-response" reason_ttl="0"/>
        <address addr="192.168.0.3" addrtype="ipv4"/>
        <hostnames>
            <hostname name="dd2bfbaeef6b" type="PTR"/>
        </hostnames>
        <ports>
            <port protocol="tcp" portid="4789">
                <state state="closed" reason="reset" reason_ttl="64"/>
                <service name="vxlan" method="table" conf="3"/>
            </port>
        </ports>
        <times srtt="63" rttvar="5000" to="100000"/>
    </host>
    <runstats></runstats>
```

Figure 14 - Structure of ips.xml.

2.4 File and folder structure of the testbed.

Having resolved the networking between hosts, the cluster can now start, train a model, and save the output to one of the nodes. Figure 15 depicts the files that need to be replicated to all of the hosts, with the only exception being the docker-compose.yml and the *run_workload.sh* files that only need to be to the swarm manager. The manager runs the swarm, which, in turn, deploys the stack to the cluster, and each node instantiates one or more depending on the workloads, containers.



```
workload_1
  script
    __pycache__
    keras-model
    ips.xml
    main.py
    mnist.py
    parse_ips.py
    start_cluster.sh
  docker-compose.yml
  Dockerfile
  mystats1_.csv
  run_workload.sh
workload_2
```

Figure 15 - File structure of the testbed.

The functionality of each folder and file is described as follows:

- *script/* is a folder containing all necessary files required inside the containers. This folder is bind mount to the container, as described in docker-compose. Files outside that folder are running on the host machine and do not interact with the containers directly.
- *__pycache__/* is a folder containing bytecode to optimize startup. It can safely be ignored or deleted and does not impact the functionality of the system.
- *keras-model/* is a folder containing the trained model created from the dataset. It includes the architecture of the model (layers, connections), the weight values, and other information relevant to the trained model. Each time old models are deleted, and new ones will be instantiated; however, this can change quickly to fit any needs.
- *ips.xml*, *parse_ips.py*, *start_cluster.sh*, *docker-compose.yml*, and *Dockerfile* have been documented in sections 2.1, 2.2, and 2.3, respectively.
- *main.py* and *mnist.py* are created according to the official documentation for TensorFlow's Multi Worker Mirrored Strategy. Expressly, mnist.py is provided as a ready training loop that has been tailored to work in a cluster. More documentation can be found at the following link and its sublinks: https://www.tensorflow.org/api_docs/python/tf/distribute/MultiWorkerMirroredStrategy

- *run_workload.sh* is a wrapper file starting the swarm and monitoring the performance of the node.
- *mystats1_.csv* contains the output of the docker stats command with some special formatting, making it easier to read during debugging. It is primarily used for debugging purposes during orchestration and as such, will not be further discussed in this thesis.

2.5 Starting the testbed, training, and saving a model.

Finally, having prepared the testbed, a model can be trained, and the cluster's performance monitored. To start the cluster, the manager node, in this case, "ubuntu", as shown in Figure 14, has to run its run_workload.sh script, which wraps the deployment of the stack and the monitoring. Its contents are shown below.

```
#!/bin/bash
#run_workload.sh
docker stack deploy --prune -c /media/hdd/edw/workload_1/docker-
compose.yml workload1

#delete previous stats if they exist and start the cluster

while true
do
docker stats -a --no-stream --
format "table {{.Name}}\t{{.NetIO}}" >> /media/hdd/edw/workload_1/mystats1_.csv
echo "Saving stats to file mystats.csv."
echo "Press Ctrl+C to exit."
done
# while the cluster is running, output the results of docker stats onto a mystats
.csv
#note that docker stats allow us
to format the output, and we use it to get only the data needed
```

This file uses the YAML file created earlier to deploy a stack, a sum of services, each running on a different host. In addition, it monitors the network throughput and saves it to the file mystats1_.csv. To start the cluster, one must execute run_workload.sh by running the following command.

```
./run_workload.sh
```

It should also be noted that this file prints directly to the terminal until closed, so it either can be modified to run in the background, or have it manually set as a background job, or open a new terminal. In any case, after starting the cluster, the file can be closed by pressing Ctrl+C. After checking the logs of any of the manager nodes, a folder named keras-model should be created under script/. This folder contains the model that was saved after the training.

Main.py is responsible for initializing each distributed node with the correct IPs, as extracted from *parse_ips.py*. After calling *parse_ips()* as a function and sorting them, it loads the dataset, creates the model, calculates each worker's batch sizes, and starts the distributed training. *Main.py* acts as a wrapper to *mnist.py*, transforming traditional monolithic training workload into a distributed one. Finally, it should be noted that instructions on how to structure *main.py* and the *mnist.py* workload are provided in the official TensorFlow documentation[18,19]. It is also noteworthy, that any workload can be performed in a distributed environment with minimal changes to the existing code; that is the job of the provided *main.py*.

The code for *mnist.py* and *main.py* with comments and documentation is shown below.

```
import os
import tensorflow as tf
import numpy as np

def mnist_dataset(batch_size):
  (x_train, y_train), _ = tf.keras.datasets.mnist.load_data()
  # The `x` arrays are in uint8 and have values in the range [0, 255].
  # You need to convert them to float32 with values in the range [0, 1]
  x_train = x_train / np.float32(255)
  y_train = y_train.astype(np.int64)
  train_dataset = tf.data.Dataset.from_tensor_slices(
      (x_train, y_train)).shuffle(60000).repeat().batch(batch_size)
  return train_dataset

def build_and_compile_cnn_model():
  model = tf.keras.Sequential([
      tf.keras.Input(shape=(28, 28)),
      tf.keras.layers.Reshape(target_shape=(28, 28, 1)),
      tf.keras.layers.Conv2D(32, 3, activation='relu'),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(128, activation='relu'),
      tf.keras.layers.Dense(10)
  ])
  model.compile(
      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
      optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
      metrics=['accuracy'])
```

[18] https://www.tensorflow.org/api_docs/python/tf/distribute/MultiWorkerMirroredStrategy
[19] https://www.tensorflow.org/tutorials/distribute/multi_worker_with_ctl

```
  return model
```

```
#main.py
import json
import mnist
import os
import time
import sys
import socket
import parse_ips as parsr
import tensorflow as tf
from time import sleep


def _is_chief(task_type, task_id):
    return (task_type == 'worker' and task_id == 0) or task_type is None

def _get_temp_dir(dirpath, task_id):
    base_dirpath = 'workertemp_' + str(task_id)
    temp_dir = os.path.join(dirpath, base_dirpath)
    tf.io.gfile.makedirs(temp_dir)
    return temp_dir

def write_filepath(filepath, task_type, task_id):
    dirpath = os.path.dirname(filepath)
    base = os.path.basename(filepath)
    if not _is_chief(task_type, task_id):
        dirpath = _get_temp_dir(dirpath, task_id)
    return os.path.join(dirpath, base)

def run_strat():
    '''This function encapsulates all the functionalities of the main program.

    Firstly, it discovers the IP of the current node and next calls the parse_ips
() function
    from the respective files. Next, it sorts the IPs that are returned, and sets
 up the tf_config
    accordingly. Then it creates the distributed strategy and calls the already c
onfigured mnist.py file. Finally,
    it saves the model on the master node, which is defined as the one with a tas
k index of 0.
    '''
    host_address = socket.gethostbyname(socket.gethostname())
    #socket library is used to get the self IP

    ip_list=parsr.parse_ips()
    #we call the parse_ips and then attemp to sort them

    ips_sorted=list()
    for i,val in enumerate(ip_list):
        ips_sorted.append(ip_list[i].split('.'))
    #in the for above, we split each IP when we encounter a '.' for example 192.1
68.0.3 would be split into four string each containing one octet

    ips_sorted=[[int(y) for y in x] for x in ips_sorted]
    #casting each octet of the IP to string
    ips_sorted = sorted(ips_sorted, key=lambda item: item[-1])
    #sorting each IP based on the last (int) element of the list it is in
    #for example we have the following IPS [[192,168,0,6], [192,168,0,3], [192,16
8,0,11]]
    #each sublist will be sorted according to its last element, in this example 6
,3,11, thus getting the sublist containing 3 first

    #print(ips_sorted)
    tf_config = {
        'cluster': {
            'worker': ['','','']
        },
        'task': {'type': 'worker', 'index': 99}
    }
    #tf_config = {
    #    'cluster': {
    #        'worker': ['', '']
    #    },
    #    'task': {'type': 'worker', 'index': 99}
    #}
    #notice how cluster does not contain valid workers and index, that is to make
 debugging easier
    for i in range(0,len(ips_sorted)):
        tf_config['cluster']['worker'][i]='.'.join(str(x) for x in ips_sorted[i])
+':5001'
        #first we add each worker in the cluster, concatenating the port in the e
nd (:5001)
        if host_address == '.'.join(str(x) for x in ips_sorted[i]):
            #to specify the index of each node dynamically we simply take the sel
f IP we saved
```

```
            #and compare it with the position of that particular IP in the sorted
 IP list.
            #for example if the IP of this node is 192.168.0.6 and the list conta
ins three IPS in this order
            #['192.168.0.3', '192.168.0.6', '192.168.0.11'] the index for this pa
rticular host will be 1
            #since it is in the second position
            tf_config['task']['index'] = i

    #print(tf_config)

    #if tf_cfg['cluster']['worker'][2] == '':
    #        print("ERROR")
    #        return -1

    os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
    #in this specific testbed, not CUDA cores are available so each node only use
s CPU power
    os.environ.pop('TF_CONFIG', None)
    #it is important to clear the operating system
    #environment variable TF_CONFIG before saving the one we created
    if '.' not in sys.path:
        sys.path.insert(0, '.')

    os.environ['TF_CONFIG'] = json.dumps(tf_config)
    batch_worker = 64

    strategy = tf.distribute.MultiWorkerMirroredStrategy()
    #at this point, we initialize the distributed strategy
    #in order for the above line to work, tf_config must have been initialized
    #correctly. Otherwise, when strategy starts, it effectively runs in a single
node system.

    tf_config  = json.loads(os.environ.get('TF_CONFIG'))
    worker_cnt = len(tf_config['cluster']['worker'])
    total_batch = batch_worker * worker_cnt
    dataset = mnist.mnist_dataset(total_batch)

    with strategy.scope():
        #Model building/compiling need to be within `strategy.scope()`.
        model = mnist.build_and_compile_cnn_model()
    model.fit(dataset, epochs=12, steps_per_epoch=70)
```

```python
    current_path = '/script/keras-model'
    task_type, task_id = (strategy.cluster_resolver.task_type,
                          strategy.cluster_resolver.task_id)
    write_path = write_filepath(current_path, task_type, task_id)
    model.save(write_path)

    if not _is_chief(task_type, task_id):
        tf.io.gfile.rmtree(os.path.dirname(write_path))

    #loaded_model = tf.keras.models.load_model(current_path)

    # Restoring the model in case we need to train it again.
    #loaded_model.fit(single_worker_dataset, epochs=2, steps_per_epoch=20)

if __name__ == "__main__":
    run_strat()
    while True:
        print("Finished Training. Remove the docker stack and save the model.")
        time.sleep(50)
```

Chapter Three

3.1 Monitoring and visualizing the performance of the cluster during training.

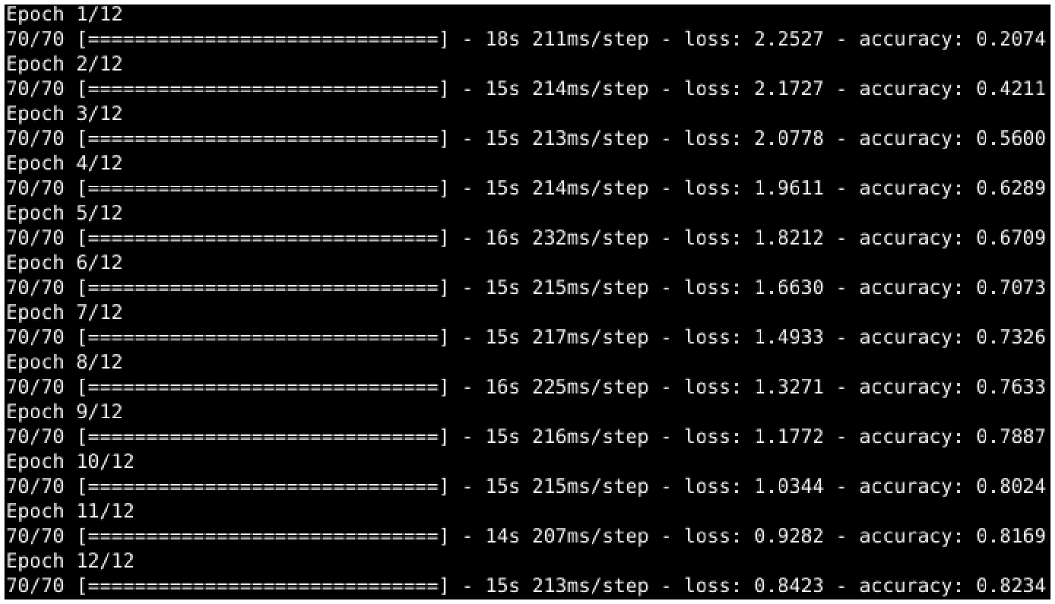
In this chapter, the performance of the testbed is evaluated. In this direction, a comparison with the bare-metal execution of the same workload in a distributed manner will be performed. In the first test case, one workload is run five times to calculate an average run time; a comparison is performed on its performance when running bare-metal versus in the container testbed. The second test case has two workloads run simultaneously on all hosts. Similarly, a comparison of the average times of each epoch and the total average time will be presented. An introduction of a delay is expected when using the docker swarm cluster. However, this delay enables the orchestration of the testbed, which in the long term, will be time-efficient. The benefits of using containers and thus orchestrating them with a tool are described in sections 1.3 and 1.4. It should be noted that the performance in the cluster can be further increased by removing the monitoring tools that were employed (i.e., cAdvisor and docker stats).

In the docker cluster, "docker stats" is used to measure the bandwidth consumed. This variable is expected to remain the same independently of which environment the workload is run. In addition, a container named cAdvisor[20], developed by Google, was used to visualize some metrics dynamically. This docker image, when downloaded, can be used to create a container that automatically sets up a dashboard on the localhost. Some visualization options supported include Grafana[21] and Prometheus[22]. This tool was primarily used to develop the testbed as a debugging tool since it can show advanced metrics such as core utilization and specific network interface throughput. However, it can only serve as a convenient debugging tool unless modified correctly. As a proof of concept, the times of each epoch and the total time calculated from the debugging messages will be shown in the following sections.

[20] https://github.com/google/cadvisor
[21] https://grafana.com/
[22] https://prometheus.io/

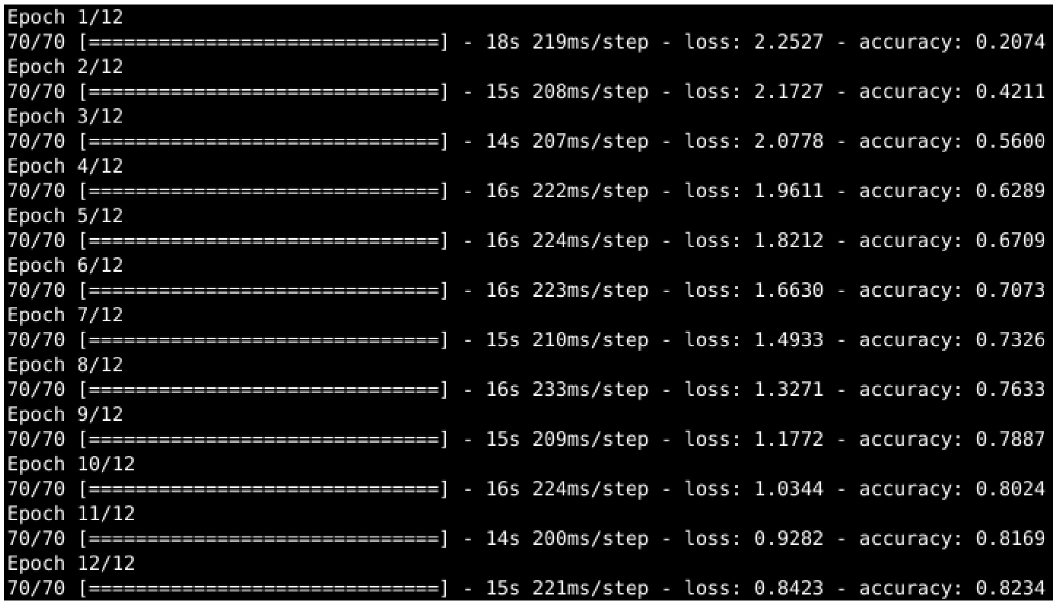
3.2 Training and comparison of a single workload on the cluster.

In this section, there will be a presentation of the resulting training times of the testbed. These results will be compared with the results of training the same model, only this time without the abstraction of the containers.



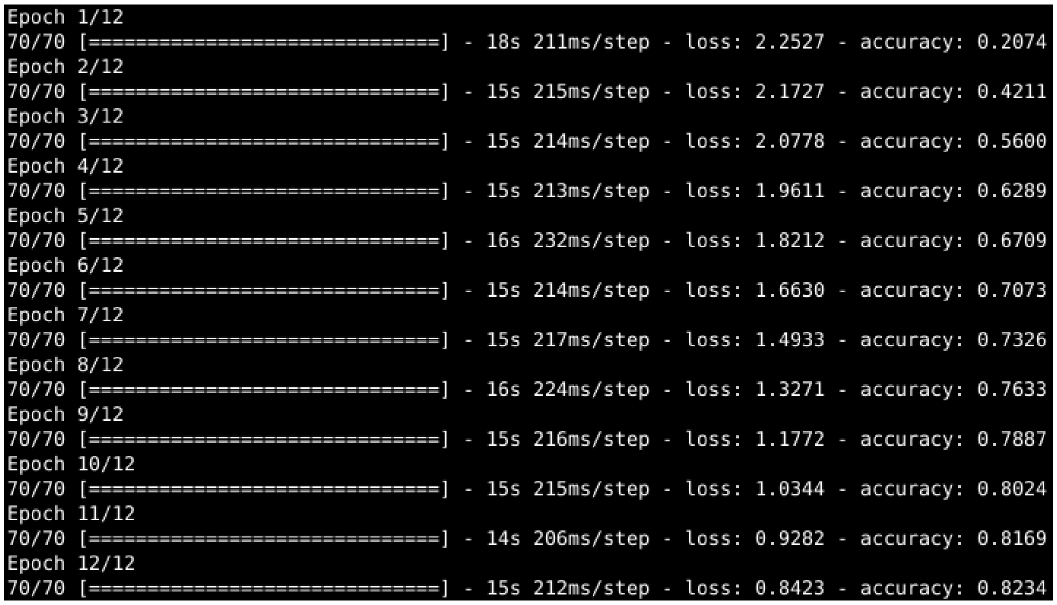
```
Epoch 1/12
70/70 [==============================] - 18s 211ms/step - loss: 2.2527 - accuracy: 0.2074
Epoch 2/12
70/70 [==============================] - 15s 214ms/step - loss: 2.1727 - accuracy: 0.4211
Epoch 3/12
70/70 [==============================] - 15s 213ms/step - loss: 2.0778 - accuracy: 0.5600
Epoch 4/12
70/70 [==============================] - 15s 214ms/step - loss: 1.9611 - accuracy: 0.6289
Epoch 5/12
70/70 [==============================] - 16s 232ms/step - loss: 1.8212 - accuracy: 0.6709
Epoch 6/12
70/70 [==============================] - 15s 215ms/step - loss: 1.6630 - accuracy: 0.7073
Epoch 7/12
70/70 [==============================] - 15s 217ms/step - loss: 1.4933 - accuracy: 0.7326
Epoch 8/12
70/70 [==============================] - 16s 225ms/step - loss: 1.3271 - accuracy: 0.7633
Epoch 9/12
70/70 [==============================] - 15s 216ms/step - loss: 1.1772 - accuracy: 0.7887
Epoch 10/12
70/70 [==============================] - 15s 215ms/step - loss: 1.0344 - accuracy: 0.8024
Epoch 11/12
70/70 [==============================] - 14s 207ms/step - loss: 0.9282 - accuracy: 0.8169
Epoch 12/12
70/70 [==============================] - 15s 213ms/step - loss: 0.8423 - accuracy: 0.8234
```

Figure 16 - Iteration times, with containers, node #1.



```
Epoch 1/12
70/70 [==============================] - 18s 219ms/step - loss: 2.2527 - accuracy: 0.2074
Epoch 2/12
70/70 [==============================] - 15s 208ms/step - loss: 2.1727 - accuracy: 0.4211
Epoch 3/12
70/70 [==============================] - 14s 207ms/step - loss: 2.0778 - accuracy: 0.5600
Epoch 4/12
70/70 [==============================] - 16s 222ms/step - loss: 1.9611 - accuracy: 0.6289
Epoch 5/12
70/70 [==============================] - 16s 224ms/step - loss: 1.8212 - accuracy: 0.6709
Epoch 6/12
70/70 [==============================] - 16s 223ms/step - loss: 1.6630 - accuracy: 0.7073
Epoch 7/12
70/70 [==============================] - 15s 210ms/step - loss: 1.4933 - accuracy: 0.7326
Epoch 8/12
70/70 [==============================] - 16s 233ms/step - loss: 1.3271 - accuracy: 0.7633
Epoch 9/12
70/70 [==============================] - 15s 209ms/step - loss: 1.1772 - accuracy: 0.7887
Epoch 10/12
70/70 [==============================] - 16s 224ms/step - loss: 1.0344 - accuracy: 0.8024
Epoch 11/12
70/70 [==============================] - 14s 200ms/step - loss: 0.9282 - accuracy: 0.8169
Epoch 12/12
70/70 [==============================] - 15s 221ms/step - loss: 0.8423 - accuracy: 0.8234
```

Figure 17 - Iteration times, with containers, node #2.



```
Epoch 1/12
70/70 [==============================] - 18s 211ms/step - loss: 2.2527 - accuracy: 0.2074
Epoch 2/12
70/70 [==============================] - 15s 215ms/step - loss: 2.1727 - accuracy: 0.4211
Epoch 3/12
70/70 [==============================] - 15s 214ms/step - loss: 2.0778 - accuracy: 0.5600
Epoch 4/12
70/70 [==============================] - 15s 213ms/step - loss: 1.9611 - accuracy: 0.6289
Epoch 5/12
70/70 [==============================] - 16s 232ms/step - loss: 1.8212 - accuracy: 0.6709
Epoch 6/12
70/70 [==============================] - 15s 214ms/step - loss: 1.6630 - accuracy: 0.7073
Epoch 7/12
70/70 [==============================] - 15s 217ms/step - loss: 1.4933 - accuracy: 0.7326
Epoch 8/12
70/70 [==============================] - 16s 224ms/step - loss: 1.3271 - accuracy: 0.7633
Epoch 9/12
70/70 [==============================] - 15s 216ms/step - loss: 1.1772 - accuracy: 0.7887
Epoch 10/12
70/70 [==============================] - 15s 215ms/step - loss: 1.0344 - accuracy: 0.8024
Epoch 11/12
70/70 [==============================] - 14s 206ms/step - loss: 0.9282 - accuracy: 0.8169
Epoch 12/12
70/70 [==============================] - 15s 212ms/step - loss: 0.8423 - accuracy: 0.8234
```

Figure 18 - Iteration times, with containers, node #3.

Figures 16 to 18 show the elapsed time for every epoch of training, as well as for every step. The training time of each iteration is measured in seconds (s), while the step in milliseconds (ms). Each figure represents one node of the system, while these figures were selected as representatives of the average time calculated. It should be briefly noted, that each node requires the same time for each iteration since training happens in a distributed manner. The per-step times vary between nodes, and that is the reason these figures are shown.

Figures 19 to 21 show each node's respective times for the same model, only when the workload is run without the cluster. It is noteworthy that in the scenario without the swarm, one must configure every time the IP addresses inside main.py manually since, due to the hosts being in a LAN network, these addresses could change.

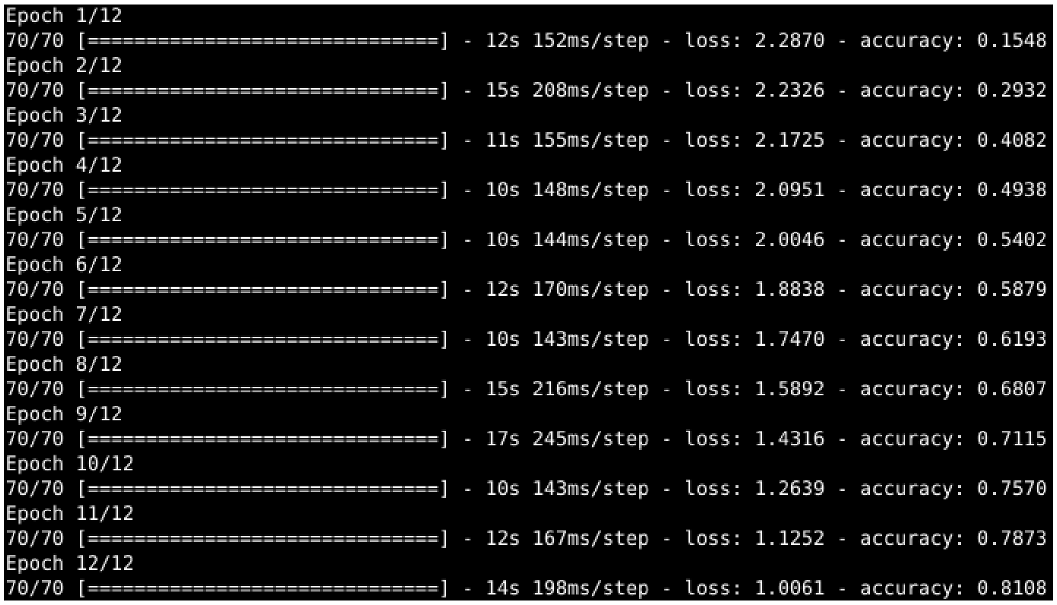

```
Epoch 1/12
70/70 [==============================] - 12s 152ms/step - loss: 2.2870 - accuracy: 0.1548
Epoch 2/12
70/70 [==============================] - 15s 208ms/step - loss: 2.2326 - accuracy: 0.2932
Epoch 3/12
70/70 [==============================] - 11s 155ms/step - loss: 2.1725 - accuracy: 0.4082
Epoch 4/12
70/70 [==============================] - 10s 148ms/step - loss: 2.0951 - accuracy: 0.4938
Epoch 5/12
70/70 [==============================] - 10s 144ms/step - loss: 2.0046 - accuracy: 0.5402
Epoch 6/12
70/70 [==============================] - 12s 170ms/step - loss: 1.8838 - accuracy: 0.5879
Epoch 7/12
70/70 [==============================] - 10s 143ms/step - loss: 1.7470 - accuracy: 0.6193
Epoch 8/12
70/70 [==============================] - 15s 216ms/step - loss: 1.5892 - accuracy: 0.6807
Epoch 9/12
70/70 [==============================] - 17s 245ms/step - loss: 1.4316 - accuracy: 0.7115
Epoch 10/12
70/70 [==============================] - 10s 143ms/step - loss: 1.2639 - accuracy: 0.7570
Epoch 11/12
70/70 [==============================] - 12s 167ms/step - loss: 1.1252 - accuracy: 0.7873
Epoch 12/12
70/70 [==============================] - 14s 198ms/step - loss: 1.0061 - accuracy: 0.8108
```

Figure 19 - Iteration times, without containers, node #1.

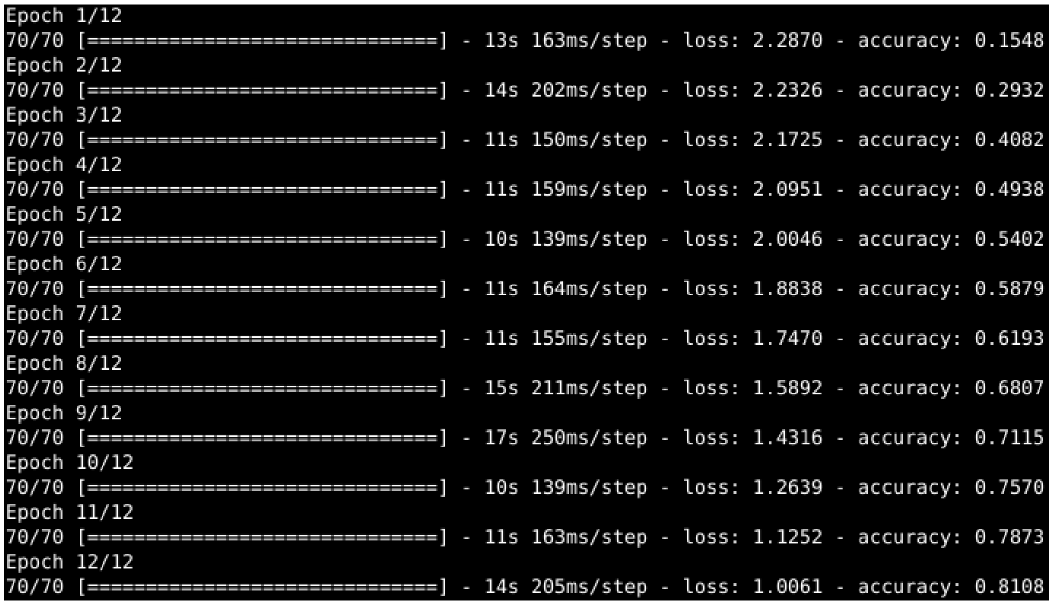
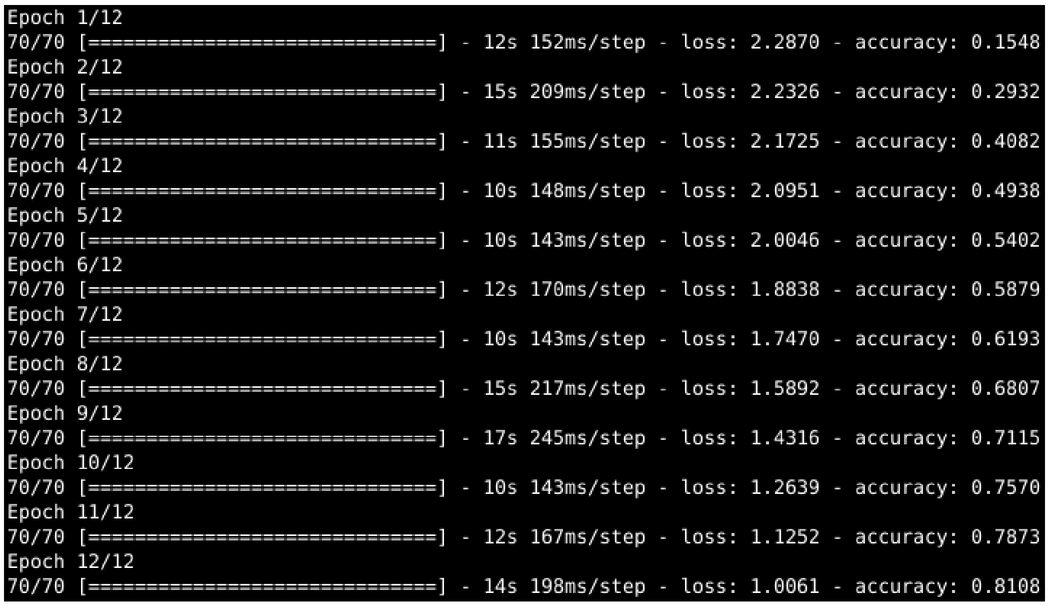

```
Epoch 1/12
70/70 [==============================] - 13s 163ms/step - loss: 2.2870 - accuracy: 0.1548
Epoch 2/12
70/70 [==============================] - 14s 202ms/step - loss: 2.2326 - accuracy: 0.2932
Epoch 3/12
70/70 [==============================] - 11s 150ms/step - loss: 2.1725 - accuracy: 0.4082
Epoch 4/12
70/70 [==============================] - 11s 159ms/step - loss: 2.0951 - accuracy: 0.4938
Epoch 5/12
70/70 [==============================] - 10s 139ms/step - loss: 2.0046 - accuracy: 0.5402
Epoch 6/12
70/70 [==============================] - 11s 164ms/step - loss: 1.8838 - accuracy: 0.5879
Epoch 7/12
70/70 [==============================] - 11s 155ms/step - loss: 1.7470 - accuracy: 0.6193
Epoch 8/12
70/70 [==============================] - 15s 211ms/step - loss: 1.5892 - accuracy: 0.6807
Epoch 9/12
70/70 [==============================] - 17s 250ms/step - loss: 1.4316 - accuracy: 0.7115
Epoch 10/12
70/70 [==============================] - 10s 139ms/step - loss: 1.2639 - accuracy: 0.7570
Epoch 11/12
70/70 [==============================] - 11s 163ms/step - loss: 1.1252 - accuracy: 0.7873
Epoch 12/12
70/70 [==============================] - 14s 205ms/step - loss: 1.0061 - accuracy: 0.8108
```

Figure 20 - Iteration times, without containers, node #2.



```
Epoch 1/12
70/70 [==============================] - 12s 152ms/step - loss: 2.2870 - accuracy: 0.1548
Epoch 2/12
70/70 [==============================] - 15s 209ms/step - loss: 2.2326 - accuracy: 0.2932
Epoch 3/12
70/70 [==============================] - 11s 155ms/step - loss: 2.1725 - accuracy: 0.4082
Epoch 4/12
70/70 [==============================] - 10s 148ms/step - loss: 2.0951 - accuracy: 0.4938
Epoch 5/12
70/70 [==============================] - 10s 143ms/step - loss: 2.0046 - accuracy: 0.5402
Epoch 6/12
70/70 [==============================] - 12s 170ms/step - loss: 1.8838 - accuracy: 0.5879
Epoch 7/12
70/70 [==============================] - 10s 143ms/step - loss: 1.7470 - accuracy: 0.6193
Epoch 8/12
70/70 [==============================] - 15s 217ms/step - loss: 1.5892 - accuracy: 0.6807
Epoch 9/12
70/70 [==============================] - 17s 245ms/step - loss: 1.4316 - accuracy: 0.7115
Epoch 10/12
70/70 [==============================] - 10s 143ms/step - loss: 1.2639 - accuracy: 0.7570
Epoch 11/12
70/70 [==============================] - 12s 167ms/step - loss: 1.1252 - accuracy: 0.7873
Epoch 12/12
70/70 [==============================] - 14s 198ms/step - loss: 1.0061 - accuracy: 0.8108
```

Figure 21 - Iteration times, without containers, node #3.

In Figures 19 to 21, when using containers, a constant increase in the average elapsed training and step time. Similarly to Figures 16 to 21, Table 4 summarizes the comparison results between the two cases, i.e., with or without container orchestration. The total time of training can be calculated by summing each epoch's times. Alternatively, this time can be calculated by obtaining the start time and the stop time of the training, as is provided in the debugging logs at runtime. By checking the records, the total training time without using the orchestrated cluster is **2 minutes and 30 seconds**, while the respective time for the same model with the same resources, only this time with the help of orchestration, is **3 minutes and 5 seconds**. These numbers verify what was already predicted that a minor delay would occur when using orchestration. This delay of 35 seconds in training that lasted 150 seconds is approximately a **20% increase** [41]. Finally, Figures 22 to 23 visualize the training times and step responses.

Epoch #	**Time(s)/Step(ms), without containers**			**Time(s)/Step(ms), with containers**		
	Node #1	Node #2	Node #3	Node #1	Node #2	Node #3
1	12/152	13/163	12/152	18/211	18/219	18/211
2	15/208	14/202	15/209	15/214	15/208	15/215
3	11/155	11/150	11/155	15/213	14/207	15/214
4	10/148	11/159	10/148	15/214	16/222	15/213
5	10/144	10/139	10/143	16/232	16/224	16/232
6	12/170	11/164	12/170	15/215	16/223	15/214
7	10/143	11/155	10/143	15/217	15/210	15/217
8	15/216	15/211	15/217	16/225	16/233	16/224
9	17/245	17/250	17/245	15/216	15/209	15/216
10	10/143	10/139	10/143	15/215	16/224	15/215
11	12/167	11/163	12/167	14/207	14/200	14/206
12	14/198	14/205	14/198	15/213	15/221	15/212

Table 4 - Results with one workload on both setups.

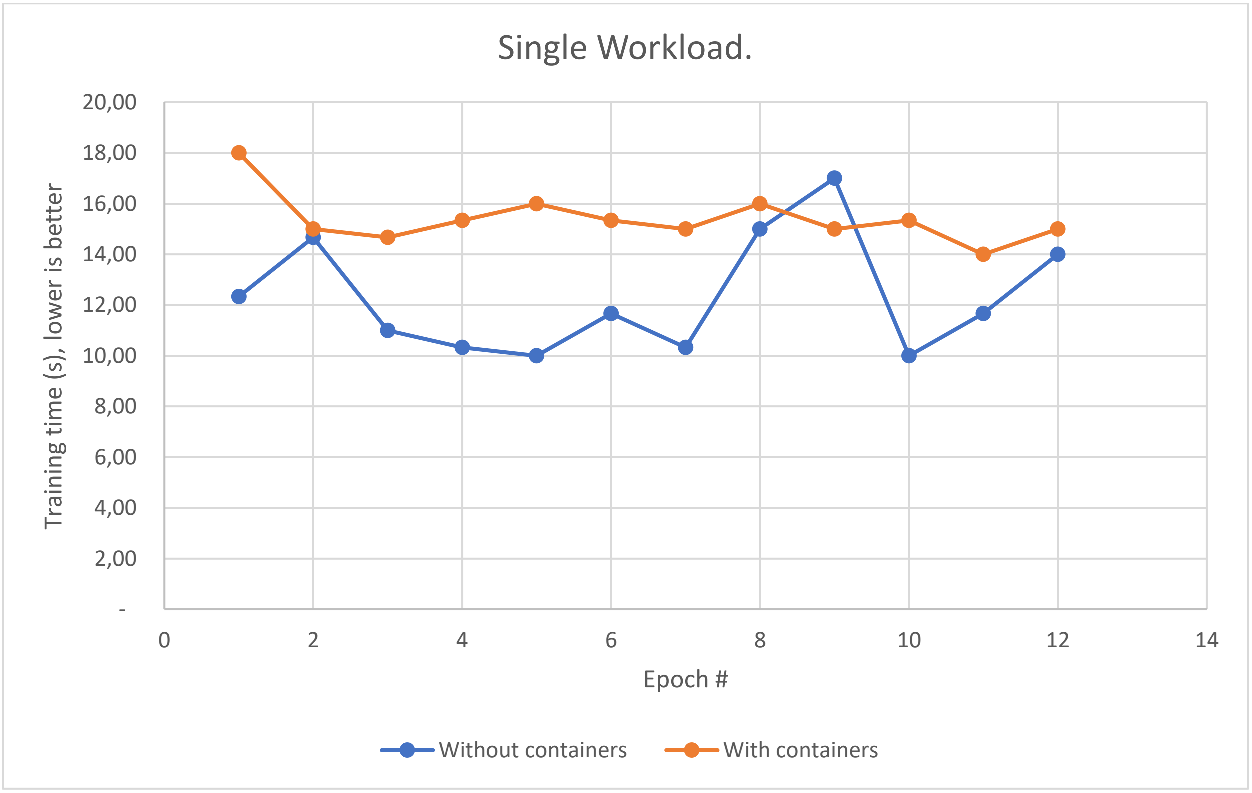


Figure 22 - Single Workload training times.

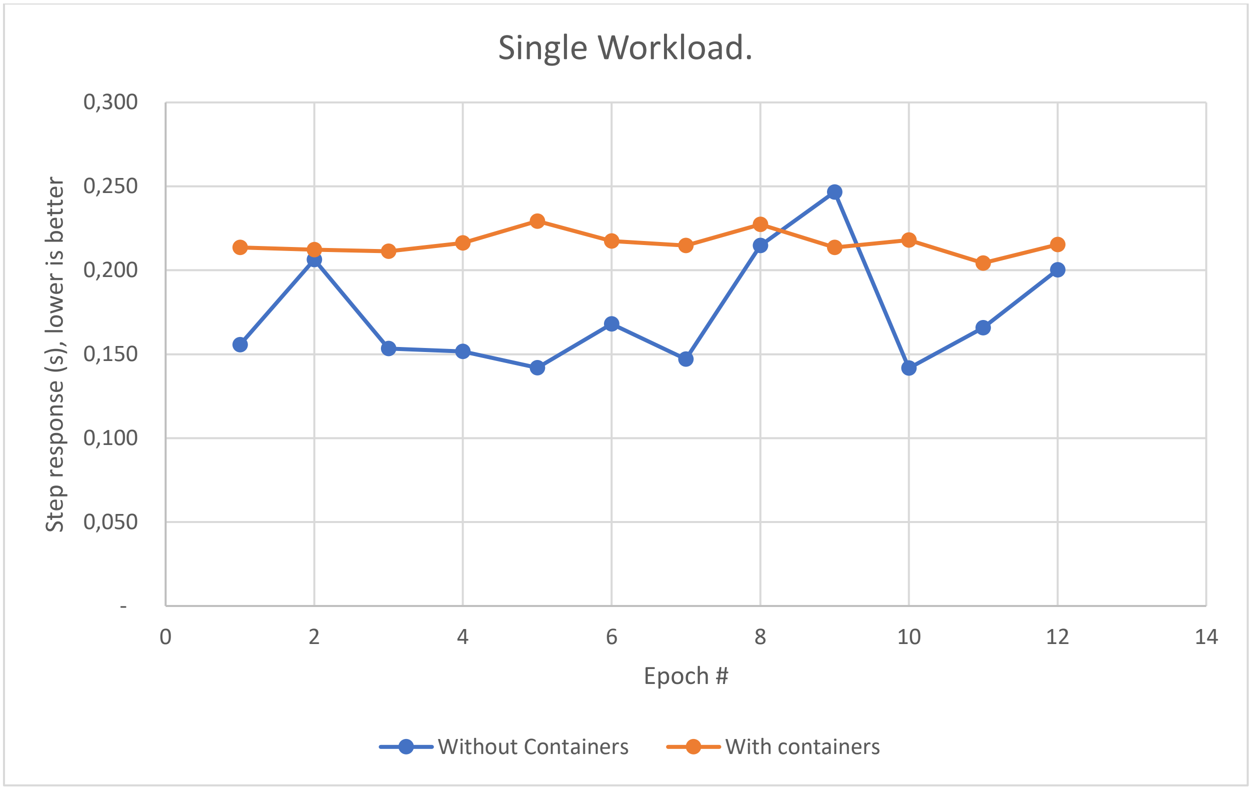
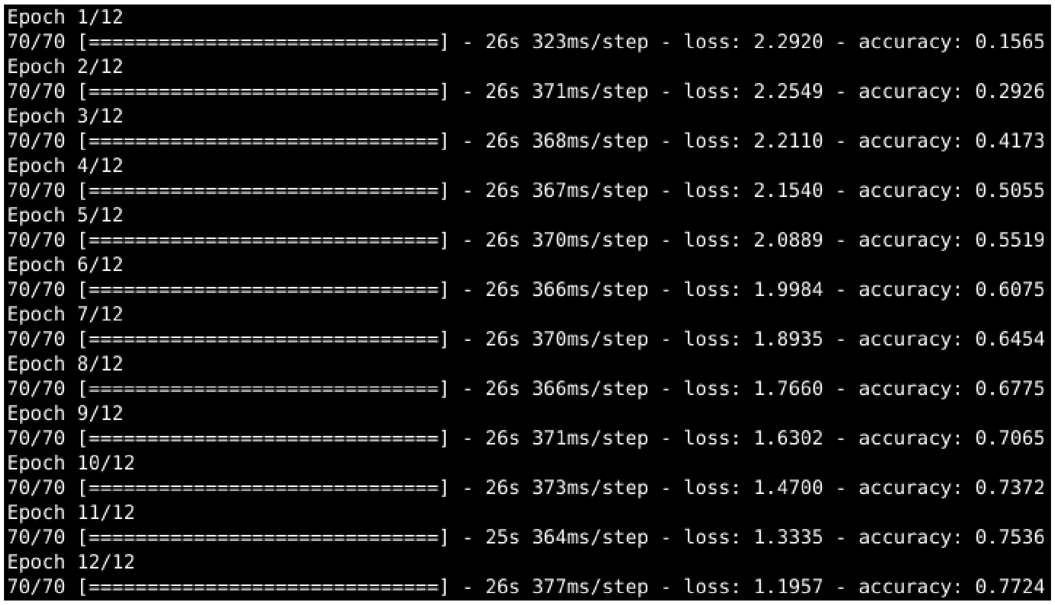


Figure 23 - Single workload, step response.

3.3 Training multiple Machine Learning models simultaneously on the cluster and visualizing the results.

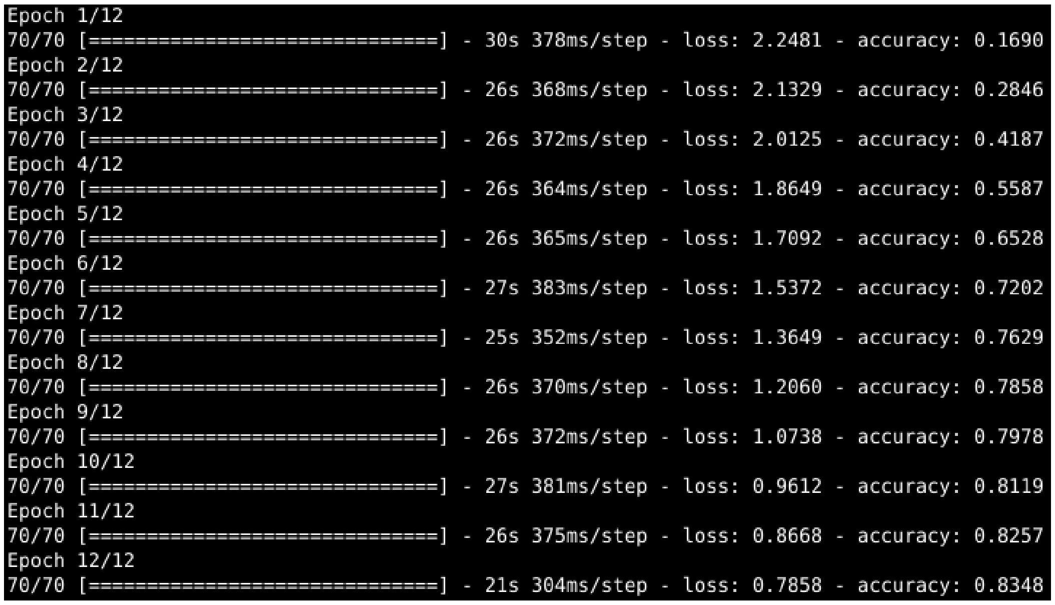
This section presents the results of running and evaluating two models simultaneously. Both models use the *mnist.py* source code, and each model is set to run in a different port. Some benefits of implementing this system with containers were seen in this scenario. Firstly, to orchestrate the container testbed, the only changes to the existing testbed are changing the network address of the new overlay network, for example, from 192.168.0.0/24 to 10.10.0.0/24. Finally, in *main.py*, for clarity reasons, the port of each worker was changed although not needed, since they are on a different network.

On the other hand, the second workload on the bare metal required an alternative port since the first port was bound for the first workload. Some logs were left out to avoid repetitiveness in this section. The corresponding results are summarized in Table 5 and Table 6 and are further elaborated on in Section 3.4. Figures 24 to 27 depict the logs that were included as samples of this scenario.



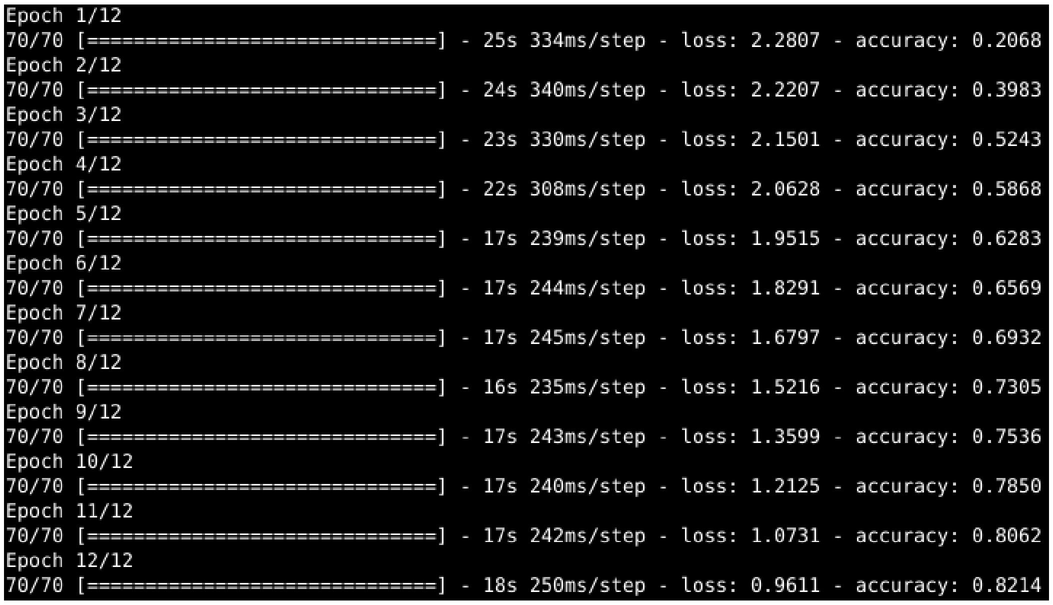
```
Epoch 1/12
70/70 [==============================] - 26s 323ms/step - loss: 2.2920 - accuracy: 0.1565
Epoch 2/12
70/70 [==============================] - 26s 371ms/step - loss: 2.2549 - accuracy: 0.2926
Epoch 3/12
70/70 [==============================] - 26s 368ms/step - loss: 2.2110 - accuracy: 0.4173
Epoch 4/12
70/70 [==============================] - 26s 367ms/step - loss: 2.1540 - accuracy: 0.5055
Epoch 5/12
70/70 [==============================] - 26s 370ms/step - loss: 2.0889 - accuracy: 0.5519
Epoch 6/12
70/70 [==============================] - 26s 366ms/step - loss: 1.9984 - accuracy: 0.6075
Epoch 7/12
70/70 [==============================] - 26s 370ms/step - loss: 1.8935 - accuracy: 0.6454
Epoch 8/12
70/70 [==============================] - 26s 366ms/step - loss: 1.7660 - accuracy: 0.6775
Epoch 9/12
70/70 [==============================] - 26s 371ms/step - loss: 1.6302 - accuracy: 0.7065
Epoch 10/12
70/70 [==============================] - 26s 373ms/step - loss: 1.4700 - accuracy: 0.7372
Epoch 11/12
70/70 [==============================] - 25s 364ms/step - loss: 1.3335 - accuracy: 0.7536
Epoch 12/12
70/70 [==============================] - 26s 377ms/step - loss: 1.1957 - accuracy: 0.7724
```

Figure 24 - Iteration times, with containers, node #1, first workload.



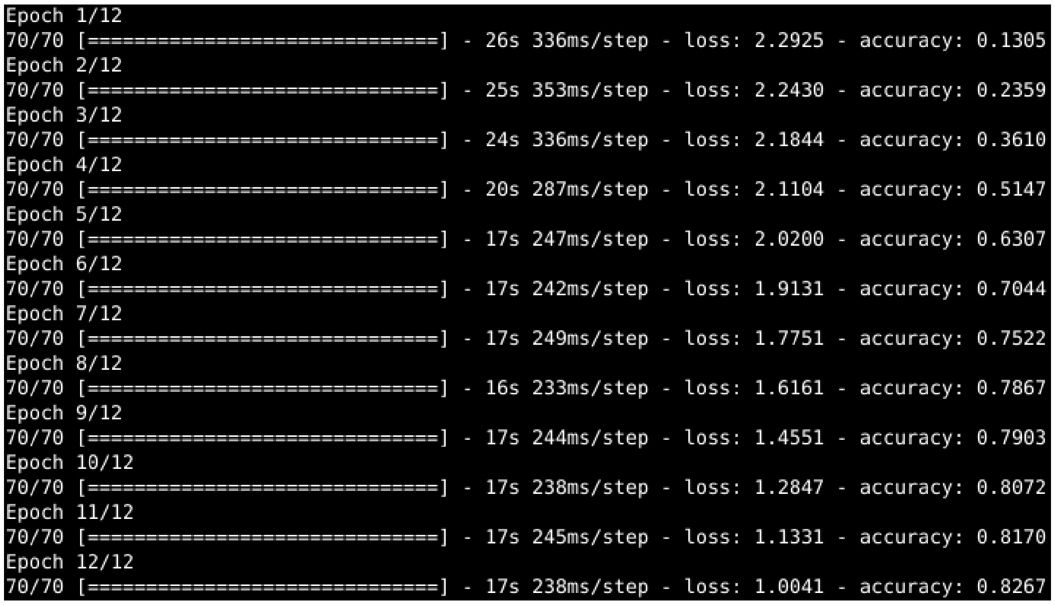
```
Epoch 1/12
70/70 [==============================] - 30s 378ms/step - loss: 2.2481 - accuracy: 0.1690
Epoch 2/12
70/70 [==============================] - 26s 368ms/step - loss: 2.1329 - accuracy: 0.2846
Epoch 3/12
70/70 [==============================] - 26s 372ms/step - loss: 2.0125 - accuracy: 0.4187
Epoch 4/12
70/70 [==============================] - 26s 364ms/step - loss: 1.8649 - accuracy: 0.5587
Epoch 5/12
70/70 [==============================] - 26s 365ms/step - loss: 1.7092 - accuracy: 0.6528
Epoch 6/12
70/70 [==============================] - 27s 383ms/step - loss: 1.5372 - accuracy: 0.7202
Epoch 7/12
70/70 [==============================] - 25s 352ms/step - loss: 1.3649 - accuracy: 0.7629
Epoch 8/12
70/70 [==============================] - 26s 370ms/step - loss: 1.2060 - accuracy: 0.7858
Epoch 9/12
70/70 [==============================] - 26s 372ms/step - loss: 1.0738 - accuracy: 0.7978
Epoch 10/12
70/70 [==============================] - 27s 381ms/step - loss: 0.9612 - accuracy: 0.8119
Epoch 11/12
70/70 [==============================] - 26s 375ms/step - loss: 0.8668 - accuracy: 0.8257
Epoch 12/12
70/70 [==============================] - 21s 304ms/step - loss: 0.7858 - accuracy: 0.8348
```

Figure 25 - Iteration times, with containers, node #1, second workload.



```
Epoch 1/12
70/70 [==============================] - 25s 334ms/step - loss: 2.2807 - accuracy: 0.2068
Epoch 2/12
70/70 [==============================] - 24s 340ms/step - loss: 2.2207 - accuracy: 0.3983
Epoch 3/12
70/70 [==============================] - 23s 330ms/step - loss: 2.1501 - accuracy: 0.5243
Epoch 4/12
70/70 [==============================] - 22s 308ms/step - loss: 2.0628 - accuracy: 0.5868
Epoch 5/12
70/70 [==============================] - 17s 239ms/step - loss: 1.9515 - accuracy: 0.6283
Epoch 6/12
70/70 [==============================] - 17s 244ms/step - loss: 1.8291 - accuracy: 0.6569
Epoch 7/12
70/70 [==============================] - 17s 245ms/step - loss: 1.6797 - accuracy: 0.6932
Epoch 8/12
70/70 [==============================] - 16s 235ms/step - loss: 1.5216 - accuracy: 0.7305
Epoch 9/12
70/70 [==============================] - 17s 243ms/step - loss: 1.3599 - accuracy: 0.7536
Epoch 10/12
70/70 [==============================] - 17s 240ms/step - loss: 1.2125 - accuracy: 0.7850
Epoch 11/12
70/70 [==============================] - 17s 242ms/step - loss: 1.0731 - accuracy: 0.8062
Epoch 12/12
70/70 [==============================] - 18s 250ms/step - loss: 0.9611 - accuracy: 0.8214
```

Figure 26 - Iteration times, without containers, node #1, first workload.



```
Epoch 1/12
70/70 [==============================] - 26s 336ms/step - loss: 2.2925 - accuracy: 0.1305
Epoch 2/12
70/70 [==============================] - 25s 353ms/step - loss: 2.2430 - accuracy: 0.2359
Epoch 3/12
70/70 [==============================] - 24s 336ms/step - loss: 2.1844 - accuracy: 0.3610
Epoch 4/12
70/70 [==============================] - 20s 287ms/step - loss: 2.1104 - accuracy: 0.5147
Epoch 5/12
70/70 [==============================] - 17s 247ms/step - loss: 2.0200 - accuracy: 0.6307
Epoch 6/12
70/70 [==============================] - 17s 242ms/step - loss: 1.9131 - accuracy: 0.7044
Epoch 7/12
70/70 [==============================] - 17s 249ms/step - loss: 1.7751 - accuracy: 0.7522
Epoch 8/12
70/70 [==============================] - 16s 233ms/step - loss: 1.6161 - accuracy: 0.7867
Epoch 9/12
70/70 [==============================] - 17s 244ms/step - loss: 1.4551 - accuracy: 0.7903
Epoch 10/12
70/70 [==============================] - 17s 238ms/step - loss: 1.2847 - accuracy: 0.8072
Epoch 11/12
70/70 [==============================] - 17s 245ms/step - loss: 1.1331 - accuracy: 0.8170
Epoch 12/12
70/70 [==============================] - 17s 238ms/step - loss: 1.0041 - accuracy: 0.8267
```

Figure 27 - Iteration times, without containers, node #1, second workload.

Epoch #	**Time(s)/Step(ms), without containers**			**Time(s)/Step(ms), with containers**		
	Node #1	Node #2	Node #3	Node #1	Node #2	Node #3
1	25/334	24/325	25/334	26/323	26/314	26/322
2	24/340	24/343	24/339	26/371	26/374	26/371
3	23/330	23/335	23/330	26/368	26/372	26/369
4	22/308	21/300	22/308	26/367	26/370	26/367
5	17/239	17/246	17/239	26/370	26/359	26/371
6	17/244	17/237	17/244	26/366	26/370	26/365
7	17/245	18/252	17/245	26/370	26/373	26/370
8	16/235	16/228	17/235	26/366	26/369	26/366
9	17/243	17/236	17/243	26/371	25/358	26/371
10	17/240	17/246	17/239	26/373	26/376	26/373
11	17/242	16/235	17/242	25/364	26/370	26/364
12	18/250	18/256	17/249	26/377	27/379	26/377

Table 5 - Results of Workload #1 on both setups.

Epoch #	**Time(s)/Step(ms), without containers**			**Time(s)/Step(ms), with containers**		
	Node #1	Node #2	Node #3	Node #1	Node #2	Node #3
1	30/378	31/382	30/397	26/336	26/327	26/336
2	26/368	25/357	26/368	25/353	25/357	25/353
3	26/372	26/374	26/372	24/336	24/339	24/335
4	26/364	26/369	26/365	20/287	20/280	20/287
5	26/365	26/369	26/366	17/247	10/253	17/247
6	27/383	26/371	27/385	17/242	17/249	17/243
7	25/352	25/355	25/352	17/249	17/242	17/249
8	26/370	26/373	26/370	16/233	17/241	16/233
9	26/372	26/376	26/371	17/244	17/239	17/245
10	27/381	27/385	27/381	17/238	17/244	17/237
11	26/375	26/365	26/374	17/245	17/237	17/244
12	21/304	22/309	21/304	17/238	17/246	17/238

Table 6 - Results of Workload #2 on both setups.

3.4 Comparison of average training times.

This section will present and comment on the graphs extracted from the tables found in Section 3.2 and Section 3.3. Firstly, Figure 28 shows the training time of workload 1, with and without the abstraction of containers, while Figure 29 shows the respective step response time of workload #1. Similarly, Figure 30 and Figure 31 show the same result for workload #2 in the same manner. It is noteworthy that workload #2 seemed to perform significantly better with the orchestrated container cluster, while workload #1 markedly worse with the cluster.

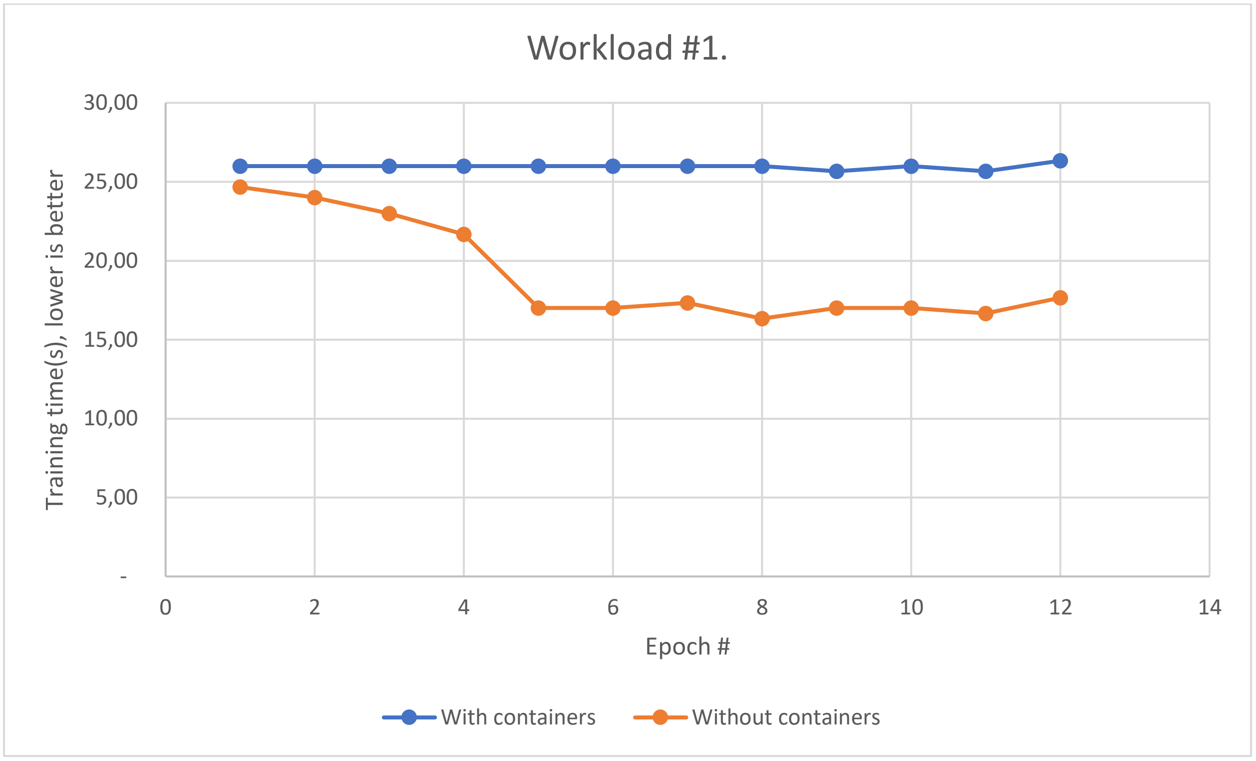


Figure 28 - Workload #1 training time.

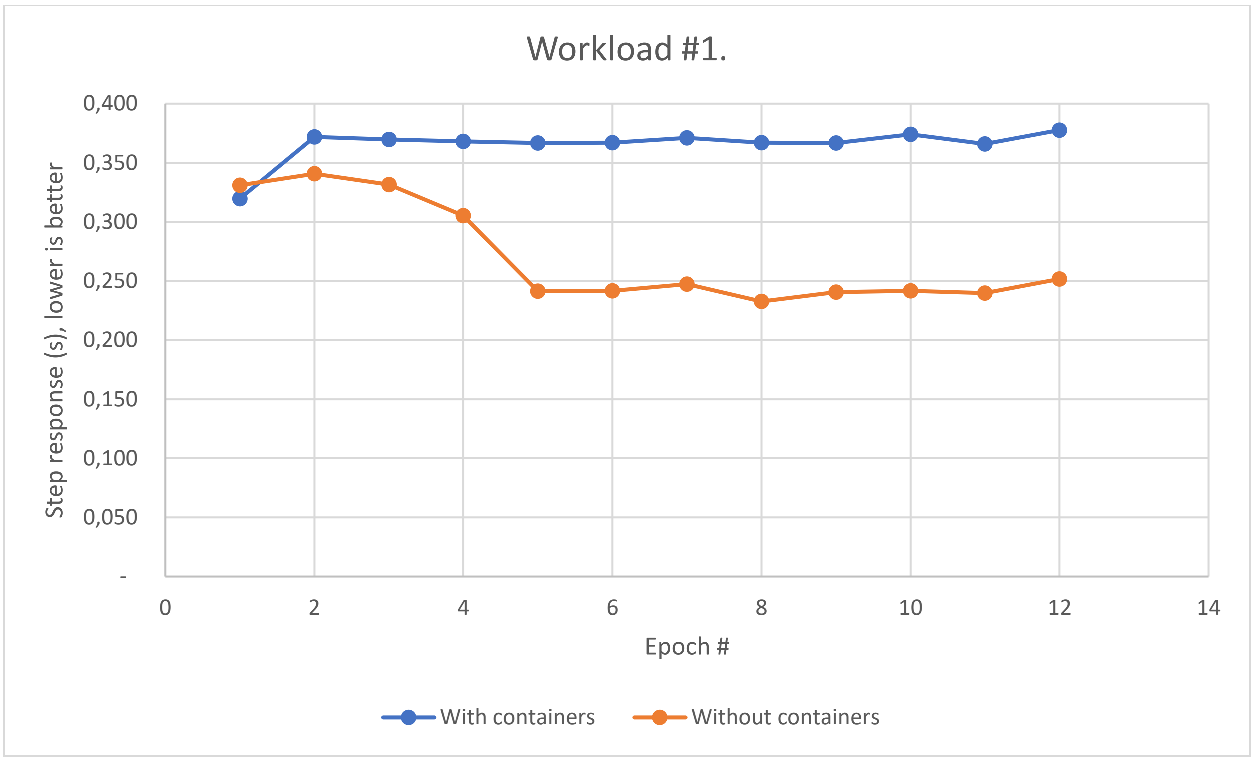


Figure 29 - Workload #1 step response.

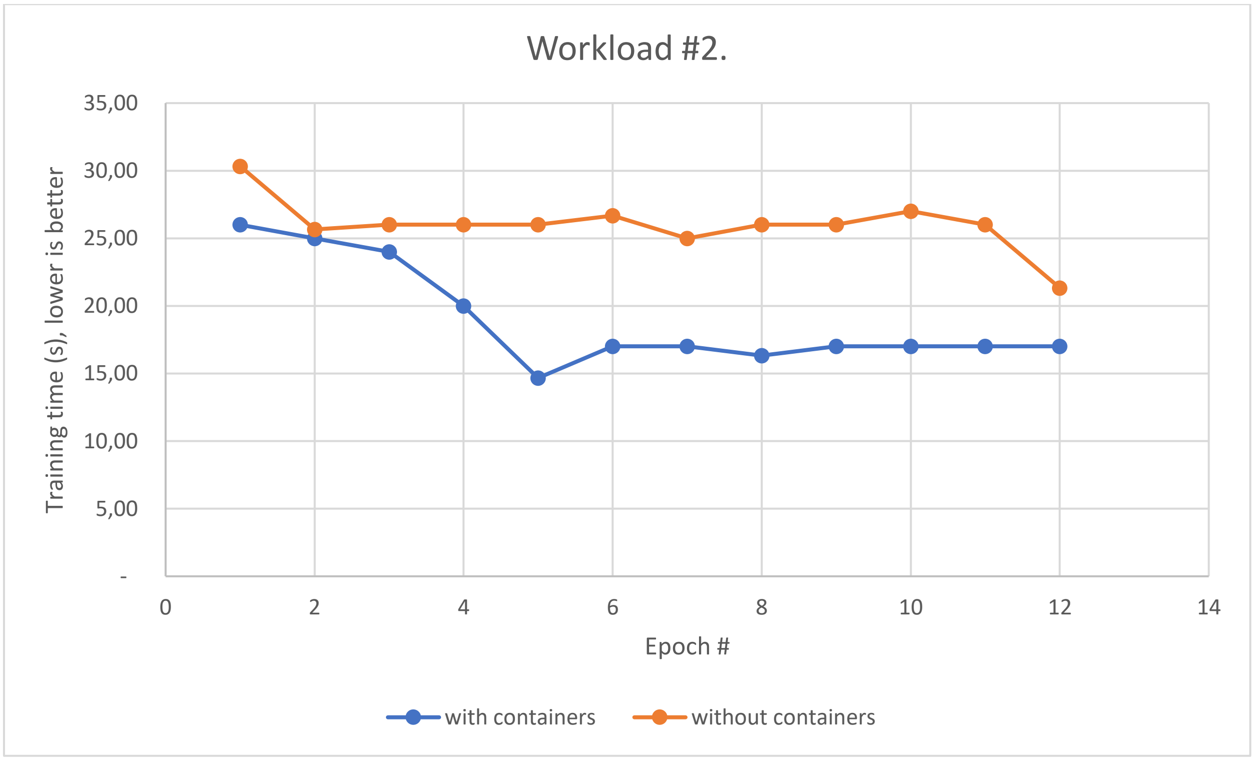


Figure 30 - Workload #2 training time.

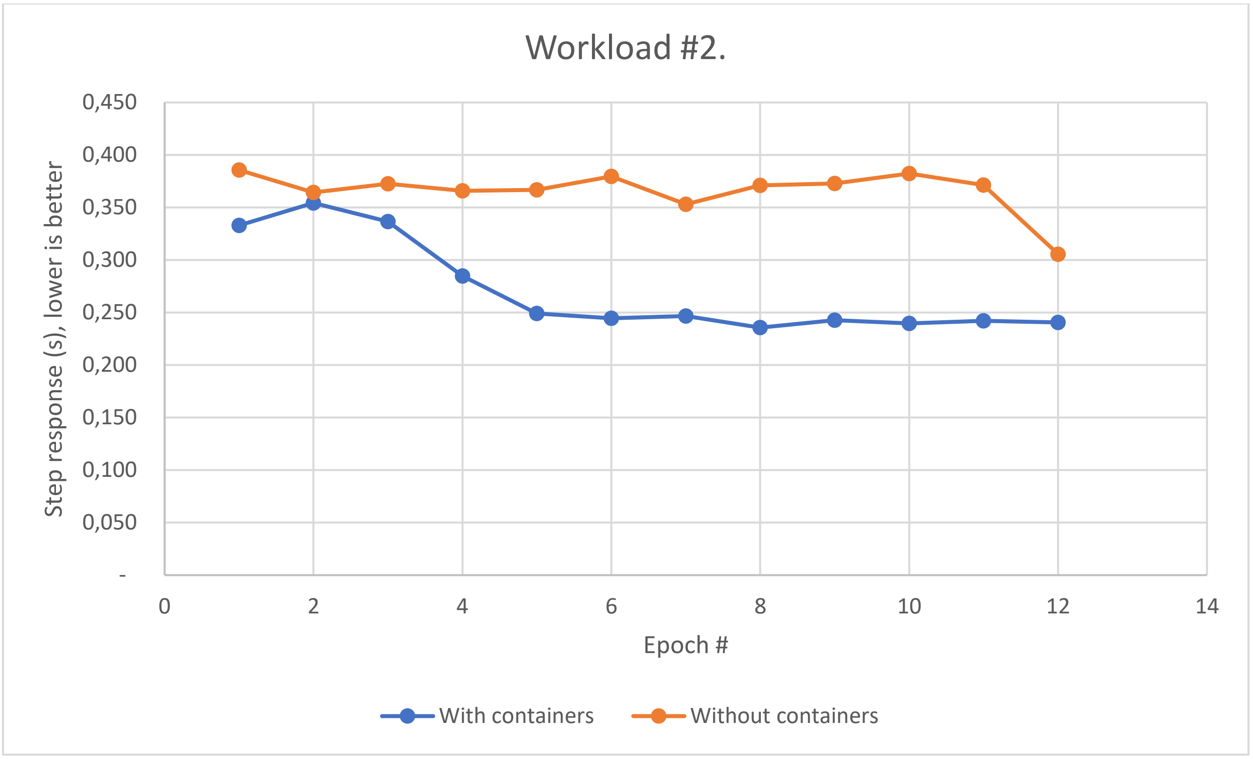


Figure 31 - Workload #2, step response.

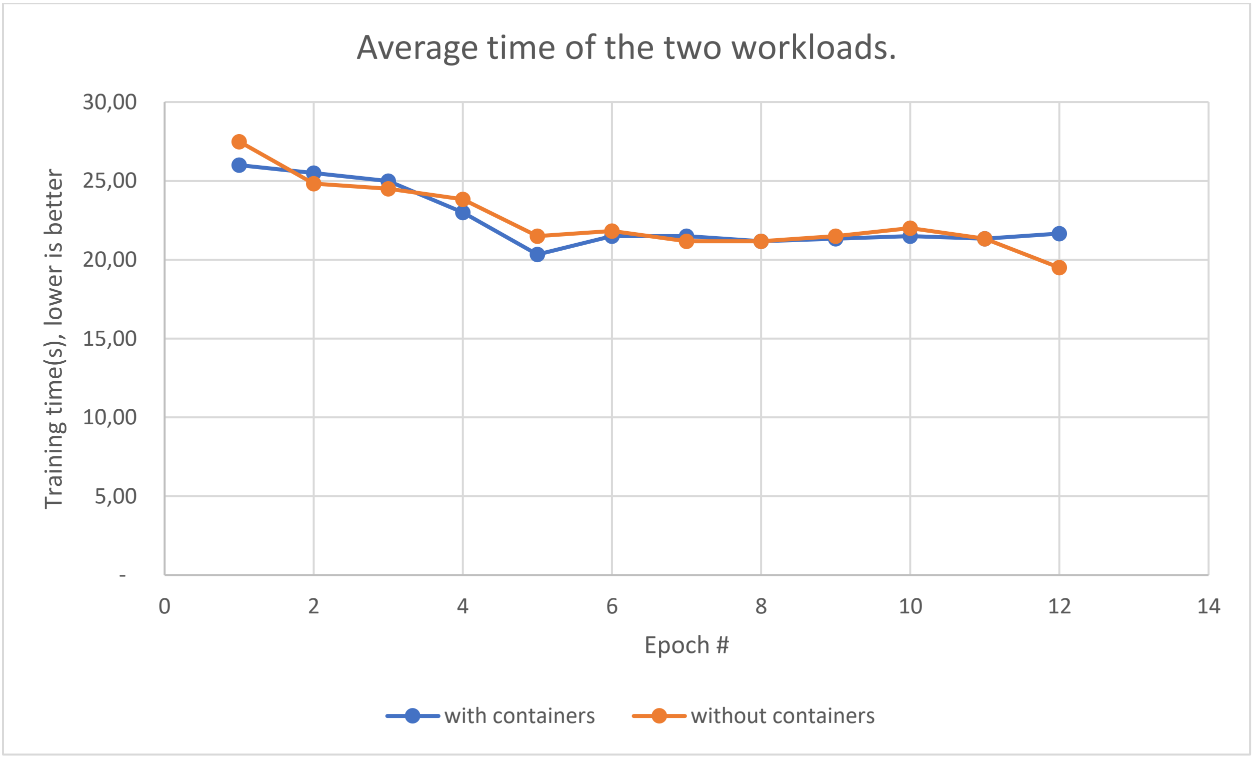


Figure 32 - Average training time comparison.

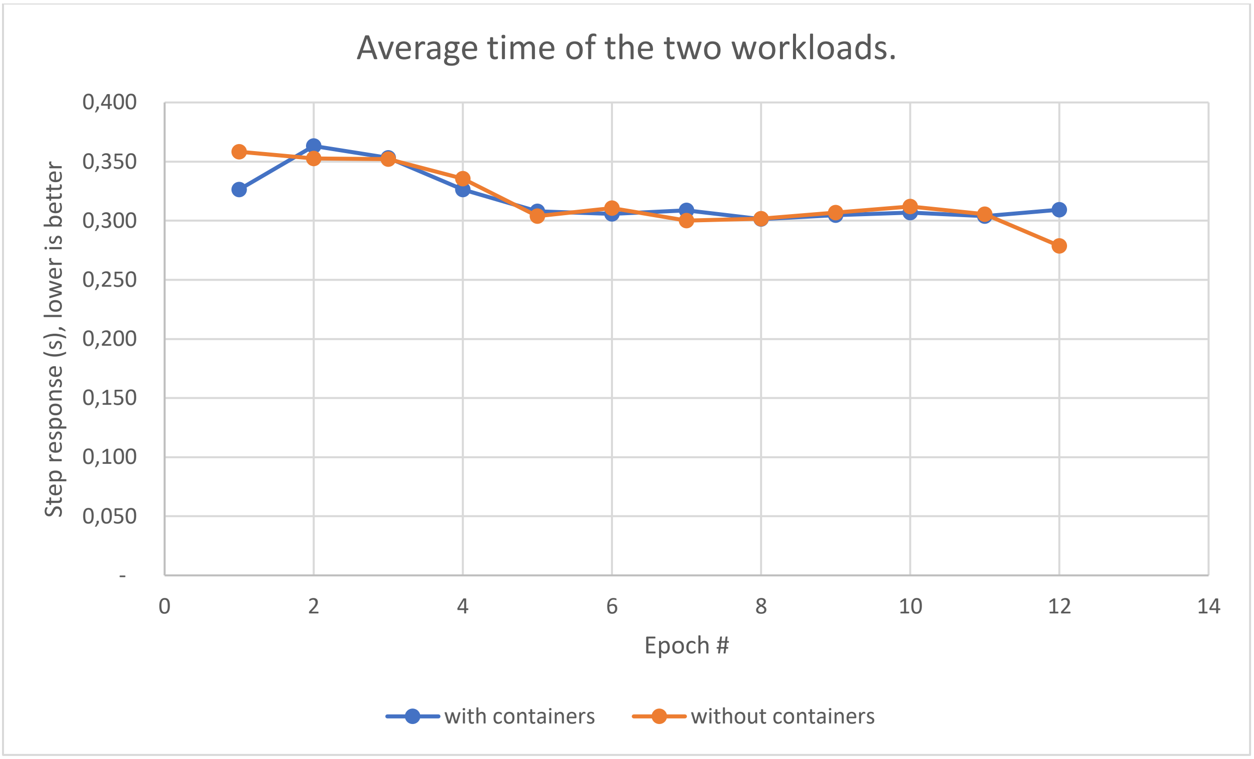


Figure 33 - Average step response comparison.

The calculated average training time for the cluster is **5 minutes and 18 seconds**, compared to 3 minutes and 5 seconds when training one workload. As for the bare metal, the average time is **4 minutes** and **10 seconds**, compared to 2 minutes and 30 seconds when training one model. The introduced delay of 20% remains the same on the simultaneous training. Therefore, it is evident that even when scaling the workload, the delay remains the same in both scenarios. However, what is easily noticed is the fact that when scaling the workload, the training time does not double. The training should last approximately 6 minutes and 10 seconds for the cluster and 5 minutes for the bare metal scenario if it doubled. Instead, it lasts about 1 minute less for the swarm and 50 seconds less for the bare metal. This time difference further demonstrates that the testbed performs well under load, compared to the bare metal, while still maintaining all the features orchestration has.

Finally, Figure 32 and Figure 33 demonstrate the times of the two workloads for the two scenarios. As deduced, for each epoch or step, both the bare metal and the orchestrated cluster perform roughly the same, within a 5% margin. The significant difference in the total training time in the multiple workloads scenario is because loading, saving, and exchanging variables takes more time in the orchestrated cluster. This difference is not apparent neither in Tables 5 and 6 nor in Figures and 33, but rather is seen at runtime.

Chapter Four

4.1 Conclusions.

In the digital age, virtualization is being adapted rapidly to address problems otherwise solved inefficiently. Employing virtualization can be done either via VMs, which are monolithic OS stacks running on one or more host devices. An alternative way of virtualizing resources is via containers, which are smaller and more lightweight than VMs. The broader architecture that utilizes containers is called Microservice architecture. In a microservice model, the problem is broken down into smaller loosely connected services. Each service solves a part of the problem efficiently with a light footprint on the underlying OS. One advantage of a microservice architecture over a monolithic approach is that the system becomes hardware-independent via abstracting the underlying hardware.

In the Machine Learning (ML) domain, distributed learning has been an appealing solution to pooling resources from different machines into training a specific model. As expected, to utilize distributed learning, each machine must be configured in the same way, most easily by cloning an existing working machine. However, this process takes time and effort while also being hardware-specific, making the scaling of the existing system challenging. To this end, several solutions have been developed to address this problem. A widely adopted approach is the use of VMs. However, VMs introduce a significant amount of overhead, reducing as such the overall performance. On the other hand, containers are being incorporated in many end products, thus abstracting the underlying hardware and enabling the engineers to tailor their workloads without any difficulties.

In this thesis, a novel way of orchestrating a system able to process machine learning workflows was presented. A cluster with Docker Swarm was created and initialized. In general, orchestrating the cluster takes place in the following steps. Firstly, the cluster structure was defined declaratively using the *docker-compose.yml* file. Furthermore, the defined architecture was deployed to the swarm cluster. Next, resolving networking was addressed, since containers are ephemeral and cannot be bound to a static IP address. Appropriate python scripts and bash wrapper scripts were developed to automate this process. Finally, bash scripts on both the host machine and all the containers were written to automate other processes such as creating a cluster, giving the correct IP addresses to each worker, and starting the workload. Finally, a cAdvisor container, along with docker stats were primarily used for debugging purposes as a way of monitoring the performance of the cluster. Other orchestration actions that could be performed on-demand would be to scale up or down the cluster if more resources were available.

Finally, a performance evaluation and comparison between the cluster and a bare-metal implementation was presented. In this direction, the workload was modified to run on the bare-metal system, while each system was then given two scenarios to process. The first scenario constitutes the training of the same model on the bare-metal system and comparing the metrics to the created cluster. In the second scenario, two models were trained simultaneously and the respective metrics were compared. According to the results, the cluster introduced a slight delay to the training.

The delay introduced is the overhead of container orchestration, which reduced by at least 20% the man-hours required to set up the testbed. In addition, from this point onwards, every time a scale up, scale down, or maintenance needs to take place, the time required will be reduced by 10-20% compared to bare-metal. It should also be noted that, in the bare-metal scenario everything must be performed manually on each system. This manual reconfiguration means that many repetitive man-hours must be allocated and requires exact knowledge of how each machine is set up, as well as the physical details of each host for the system to be reconfigured. These downsides of bare-metal systems were reduced to almost zero by implementing resource abstraction with containers, and therefore orchestration of the cluster. Finally, it should be noted that workloads that run distributed exacerbate the downsides of monolithic systems, and as such, monolithic approaches should usually be avoided [15].

Systems such as these can further enhance the efficiency of emerging technologies, such as MEC and, more broadly, IoT environments [42]–[44], [45]–[49]. An example of such use could be deploying an ML model onto a MEC network based on the users' needs. In a MEC network where hosts used video streaming mainly, an ML model would be deployed and trained to optimize the video quality and buffering times.

4.2 Future work.

The work performed in this dissertation can be further extended in the field of container orchestration alongside ML workflows. In this respect, five research directions are discussed as follows:

- Cluster security, taking into consideration all best practices as far as security of the cluster is concerned. This direction should be expanded should the system go into production.
- Expanding the supported workloads by incorporating additional workloads resembling real-life scenarios.
- Benchmarking of the system in a much larger environment to verify its performance. Additionally, alternative workloads can be considered to further ensure its performance[50].
- Optimization of the testbed by switching to GPUs or TPUs and appropriate routing to reduce training times and latencies as much as possible.
- Integration of autonomous capabilities that enable the testbed to make decisions and change its ideal state, as described in the *docker-compose.yml*. Such decisions could be the result of an RL model acting as the orchestrator.

References

[1] Y. Li, W. Li, and C. Jiang, “A Survey of Virtual Machine System: Current Technology and Future Trends,” *Electronic Commerce and Security, International Symposium*, vol. 0, pp. 332–336, Jul. 2010, doi: 10.1109/ISECS.2010.80.

[2] J. E. Smith and R. Nair, “The architecture of virtual machines,” *Computer*, vol. 38, no. 5, 2005, doi: 10.1109/MC.2005.173.

[3] M. Rosenblum and T. Garfinkel, “Virtual machine monitors: current technology and future trends,” *Computer*, vol. 38, no. 5, 2005, doi: 10.1109/MC.2005.176.

[4] “What is virtualization?” https://www.redhat.com/en/topics/virtualization/what-is-virtualization (accessed Jun. 15, 2021).

[5] “TIME-SHARING SYSTEMS: VIRTUAL MACHINE CONCEPT VS. CONVENTIONAL APPROACH.” https://web.mit.edu/smadnick/www/papers/J004.pdf (accessed Jun. 15, 2021).

[6] “IBM Archives: System/360 Dates and characteristics,” Jan. 23, 2003. https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_FS360.html (accessed Jun. 15, 2021).

[7] “IBM System/370 Principles of Operation.” http://www.bitsavers.org/pdf/ibm/370/princOps/GA22-7000-0_370_Principles_Of_Operation_Jun70.pdf (accessed Jun. 15, 2021).

[8] “Virtualization - Statistics & Facts,” *Statista*. https://www.statista.com/topics/6795/virtualization/ (accessed Jun. 15, 2021).

[9] “Overhead Memory on Virtual Machines.” https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-B42C72C1-F8D5-40DC-93D1-FB31849B1114.html (accessed Jun. 15, 2021).

[10] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[11] P. E. N, F. J. P. Mulerickal, B. Paul, and Y. Sastri, “Evaluation of Docker containers based on hardware utilization,” 2015. doi: 10.1109/ICCC.2015.7432984.

[12] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and Linux containers,” 2015. doi: 10.1109/ISPASS.2015.7095802.

[13] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O. Kwon, and B. Kim, “Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud,” 2014. doi: 10.14257/ASTL.2014.66.25.

[14] M. Chiang and T. Zhang, “Fog and IoT: An Overview of Research Opportunities,” *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, Dec. 2016, doi: 10.1109/JIOT.2016.2584538.

[15] S. Yi, C. Li, and Q. Li, “A Survey of Fog Computing: Concepts, Applications and Issues,” in *Proceedings of the 2015 Workshop on Mobile Big Data*, Hangzhou China, Jun. 2015, pp. 37–42. doi: 10.1145/2757384.2757397.

[16] A. Yousefpour *et al.*, “All one needs to know about fog computing and related edge computing paradigms: A complete survey,” *Journal of Systems Architecture*, vol. 98, pp. 289–330, Sep. 2019, doi: 10.1016/j.sysarc.2019.02.009.

[17] S. Sezer *et al.*, “Are we ready for SDN? Implementation challenges for software-defined networks,” *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, Jul. 2013, doi: 10.1109/MCOM.2013.6553676.

[18] N. Feamster, J. Rexford, and E. Zegura, “The road to SDN: an intellectual history of programmable networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014, doi: 10.1145/2602204.2602219.

[19] K. Kirkpatrick, “Software-defined networking,” *Commun. ACM*, vol. 56, no. 9, pp. 16–19, Sep. 2013, doi: 10.1145/2500468.2500473.

[20] N. M. M. K. Chowdhury and R. Boutaba, “A survey of network virtualization,” *Computer Networks*, vol. 54, no. 5, 2010, doi: 10.1016/j.comnet.2009.10.017.

[21] H. Kim and N. Feamster, “Improving network management with software defined networking,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, Feb. 2013, doi: 10.1109/MCOM.2013.6461195.

[22] J. G. Jimenez and A. G. Cervero, “Overview and Challenges of Overlay Networks: A Survey,” *International Journal of Computer Science & Engineering Survey*, vol. 2, no. 1, 2011, doi: 10.5121/ijcses.2011.2102.

[23] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-Defined Networking: A Comprehensive Survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015, doi: 10.1109/JPROC.2014.2371999.

[24] K. Cabaj, J. Wytrębowicz, S. Kuklinski, P. Radziszewski, and K. Dinh, *SDN Architecture Impact on Network Security*. 2014. doi: 10.15439/2014F473.

[25] D. Ongaro and J. Ousterhout, “In Search of an Understandable Consensus Algorithm,” p. 18.

[26] R. van Renesse, Y. Minsky, and M. Hayden, “A gossip-style failure detection service,” in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Berlin, Heidelberg, Nov. 2009, pp. 55–70.

[27] M. I. Jordan and T. M. Mitchell, “Machine learning: Trends, perspectives, and prospects,” *Science*, vol. 349, no. 6245, pp. 255–260, Jul. 2015, doi: 10.1126/science.aaa8415.

[28] D. Demirović, E. Skejić, and A. Šerifović–Trbalić, “Performance of Some Image Processing Algorithms in Tensorflow,” in *2018 25th International Conference on Systems, Signals and Image Processing (IWSSIP)*, Jun. 2018, pp. 1–4. doi: 10.1109/IWSSIP.2018.8439714.

[29] A. Jain, A. A. Awan, Q. Anthony, H. Subramoni, and D. K. D. Panda, “Performance Characterization of DNN Training using TensorFlow and PyTorch on Modern Clusters,” in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2019, pp. 1–11. doi: 10.1109/CLUSTER.2019.8891042.

[30] C. Jia *et al.*, “Improving the Performance of Distributed TensorFlow with RDMA,” *Int J Parallel Prog*, vol. 46, no. 4, pp. 674–685, Aug. 2018, doi: 10.1007/s10766-017-0520-3.

[31] P. Louridas and C. Ebert, “Machine Learning,” *IEEE Software*, vol. 33, no. 5, pp. 110–115, Sep. 2016, doi: 10.1109/MS.2016.114.

[32] Y. J. Mo, J. Kim, J.-K. Kim, A. Mohaisen, and W. Lee, “Performance of deep learning computation with TensorFlow software library in GPU-capable multi-core computing platforms,” in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, Jul. 2017, pp. 240–242. doi: 10.1109/ICUFN.2017.7993784.

[33] J. Dean, D. Patterson, and C. Young, “A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution,” *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar. 2018, doi: 10.1109/MM.2018.112130030.

[34] J. Lawrence, J. Malmsten, A. Rybka, D. A. Sabol, and K. Triplin, “Comparing TensorFlow Deep Learning Performance Using CPUs, GPUs, Local PCs and Cloud,” p. 8.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.

[36] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, doi: 10.1109/5.726791.

[37] C. Wick, C. Reul, and F. Puppe, “Calamari − A High-Performance Tensorflow-based Deep Learning Package for Optical Character Recognition,” p. 12.

[38] A. Jain, A. A. Awan, H. Subramoni, and D. K. Panda, “Scaling TensorFlow, PyTorch, and MXNet using MVAPICH2 for High-Performance Deep Learning on Frontera,” in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, Nov. 2019, pp. 76–83. doi: 10.1109/DLS49591.2019.00015.

[39] D. Cavdar *et al.*, “Densifying Assumed-sparse Tensors: Improving Memory Efficiency and MPI Collective Performance during Tensor Accumulation for Parallelized Training of Neural

Machine Translation Models," *arXiv:1905.04035 [cs]*, May 2019, Accessed: Jul. 06, 2021. [Online]. Available: http://arxiv.org/abs/1905.04035

[40] A. A. Awan, J. Bedorf, C.-H. Chu, H. Subramoni, and D. K. Panda, "Scalable Distributed DNN Training using TensorFlow and CUDA-Aware MPI: Characterization, Designs, and Performance Evaluation," *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 498–507, May 2019, doi: 10.1109/CCGRID.2019.00064.

[41] P. Mendki, "Docker container based analytics at IoT edge Video analytics usecase," in *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*, Feb. 2018, pp. 1–4. doi: 10.1109/IoT-SIU.2018.8519852.

[42] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "Mobile Edge Computing: Survey and Research Outlook," p. 31.

[43] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust, "Mobile-Edge Computing Architecture: The role of MEC in the Internet of Things," *IEEE Consumer Electronics Magazine*, vol. 5, no. 4, pp. 84–91, Oct. 2016, doi: 10.1109/MCE.2016.2590118.

[44] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A Survey on Mobile Edge Computing: The Communication Perspective," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2322–2358, Fourthquarter 2017, doi: 10.1109/COMST.2017.2745201.

[45] T. Dillon, C. Wu, and E. Chang, "Cloud Computing: Issues and Challenges," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, Apr. 2010, pp. 27–33. doi: 10.1109/AINA.2010.187.

[46] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing — The business perspective," *Decision Support Systems*, vol. 51, no. 1, pp. 176–189, Apr. 2011, doi: 10.1016/j.dss.2010.12.006.

[47] L. Wang *et al.*, "Cloud Computing: a Perspective Study," *New Gener. Comput.*, vol. 28, no. 2, pp. 137–146, Apr. 2010, doi: 10.1007/s00354-008-0081-5.

[48] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *J Internet Serv Appl*, vol. 1, no. 1, pp. 7–18, May 2010, doi: 10.1007/s13174-010-0007-6.

[49] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, Fourthquarter 2015, doi: 10.1109/COMST.2015.2444095.

[50] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv:1802.05799 [cs, stat]*, Feb. 2018, Accessed: Jul. 06, 2021. [Online]. Available: http://arxiv.org/abs/1802.05799