UNIVERSITY OF THESSALY

DEVELOPMENT OF TECHNIQUES FOR THE OPTIMIZATION OF THE EXECUTION OF MACHINE LEARNING APPLICATIONS IN CLOUD AND EDGE ENVIRONMENTS

Diploma Thesis

Nikolaos P. Angelopoulos

Supervisor

Associate Professor **Korakis Athanasios**

A Thesis submitted in fulfillment of the requirements for the degree of Diploma Thesis in the

Network Implementation Testbed Laboratory (NITLab)

Department of Electrical and Computer Engineering

Volos, 2021

Πανεπιστήμιο Θεσσαλίας

ΑΝΑΠΤΥΞΗ ΤΕΧΝΙΚΩΝ ΓΙΑ ΤΗΝ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ ΤΗΣ ΕΚΤΕΛΕΣΗΣ ΕΦΑΡΜΟΓΩΝ ΜΗΧΑΝΙΚΗΣ ΕΚΜΑΘΗΣΗΣ ΣΕ CLOUD και EDGE ΠΕΡΙΒΑΛΛΟΝΤΑ

Διπλωματική Εργασία

Νικόλαος Π. Αγγελόπουλος

Επιβλέπων

Αναπληρωτής Καθηγητής **Κοράκης Αθανάσιος**

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 2 Ιουλίου 2021

……	……	……
Αναπληρωτής Καθηγητής	Αναπληρωτής Καθηγητής	Αναπληρωτής Καθηγητής
Κοράκης Αθανάσιος	Αργυρίου Αντώνιος	Μπαργιώτας Δημήτριος

Βόλος, 2021

This Thesis is dedicated to me and my family.

Acknowledgments

First and foremost, I would like to express my immeasurable appreciation and deepest gratitude to my supervisor Professor Korakis Athanasios for giving me the chance to work on this thesis that was a perfect match for my interests. His guidance and patience were crucial for the completion of this Thesis. I would also wish to express my gratitude to my advisor Ilias Syrigos for his constant guidance, his valuable comments, and key suggestions throughout all stages of this work.

Also, I would like to thank my family for their unconditional love, support, understanding and unwavering belief in me throughout all those years. Finally, I am more than thankful to my friends for their constant support, who are by my side and help me in any problem that arises.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The 11 points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The Declarant

Nikolaos Angelopoulos
2 July 2021

Abstract

The evolution of Machine Learning and the widespread dominance of its applications in our everyday lives has led to the need for more sophisticated algorithms and mathematical models for achieving optimal performance. However, boosting performance comes with higher demands in terms of computational power, while at the same time request processing of real-time applications becomes slower, as complex Machine Learning models are deployed further away from data sources. To overcome these undesired effects, along with the development of applications' infrastructure, research has focused on the effective deployment of Machine Learning applications spreading from Cloud to the Edge and IoT devices. This leads in better performance, with the least possible energy consumption, and therefore lower cost.

This diploma thesis focuses on the implementation of an algorithm that aims to distribute the components of a Machine Learning application across the computing continuum, to adapt effectively to the application's needs. This way, energy intensive processes are deployed in a computer cluster featuring the necessary computing resources. At the same time, processes requiring low latency, but exhibit low cpu and memory utilization are deployed at the Edge, closer to the end user. This achieves better resource management, optimal performance, and overall better user experience.

Περίληψη

Στη σημερινή εποχή οι εφαρμογές Μηχανικής Μάθησης αυξάνονται ολοένα και περισσότερο. Αυτό οδηγεί στην ανάπτυξη αλγορίθμων και μαθηματικών μοντέλων για την καλύτερη απόδοση αυτών των εφαρμογών. Καλύτερη απόδοση όμως, σημαίνει μεγαλύτερες απαιτήσεις ως προς την επεξεργαστική ισχύ που χρειάζονται, αλλά και στον χρόνο αναμονής της επεξεργασίας αιτημάτων. Η δημιουργία αυτών των αναγκών, σε συνδυασμό με την ανάπτυξη διαφόρων τεχνολογιών για την υποδομή των εφαρμογών, οδήγησε στην αναζήτηση για τεχνικές που κατανέμουν τις λειτουργίες αυτών των εφαρμογών ανάλογα με τις απαιτήσεις τους. Αυτό έχει ως αποτέλεσμα, την καλύτερη απόδοση, με την λιγότερη δυνατή κατανάλωση ενέργειας, αρα και μικρότερο κόστος.

Σκοπός αυτής της διπλωματικής είναι η υλοποίηση ενός αλγορίθμου με στόχο την κατανομή των κομματιών μιας εφαρμογής Μηχανικής Μάθησης στην κατάλληλη συστοιχία υπολογιστών ανάλογα με τις απαιτήσεις της. Με αυτόν τον τρόπο, οι ενεργειακά απαιτητικές διεργασίες θα ανατίθενται σε συστοιχίες υπολογιστών που διαθέτουν τους αναγκαίους υπολογιστικούς πόρους. Παράλληλα οι διεργασίες που απαιτούν μικρό χρόνο αναμονής, και λιγότερους πόρους θα ανατίθενται σε συσκευές που βρίσκονται πιο κοντά στον τελικό χρήστη. Με αυτόν τον τρόπο επιτυγχάνεται καλύτερη διαχείριση των πόρων και βελτιώνεται η εμπειρία του χρήστη της εφαρμογής.

Contents

Acknowledgments ..... **v**

Abstract ..... **viii**

Περίληψη ..... **ix**

List of Figures ..... **xiii**

List of Charts ..... **xiv**

List of Tables ..... **xiv**

CHAPTER 1: INTRODUCTION **1**

1.1: Background ***1***

1.2: Motivation ***3***

1.3: Content Overview ***5***

CHAPTER 2: TECHNICHAL BACKGROUND & TOOLS **6**

2.1: Introduction ***6***

2.2: OS-Level Virtualization ***6***

2.2.1: Overview 6

2.2.1: Building blocks 7

2.3: Docker ***8***

2.3.1: Overview 8

2.3.2: Container 8

2.3.3: Images 9

2.3.4: Registries 11

2.4: Kubernetes ***11***

2.4.1: Overview 11

2.4.2: Architecture 12

2.4.3: Concepts 14

2.4.4: Storage 15

2.4.4.1: Volume 15

2.4.4.2: Persistent Volume 16

2.4.4.3: Storage Class 17

2.5: Machine Learning 17

2.5.1: Overview 17

2.5.2: Steps of machine learning application 18

2.6: Kubeflow 20

2.6.1: Overview 20

2.6.2: Architecture 21

2.6.3: Notebook Servers 22

2.6.4: Kubeflow Pipelines 25

2.6.5: Software Define Kit 27

2.7: Metrics & Display 28

2.7.1: Prometheus 28

2.7.2: Grafana 29

CHAPTER 3: IMPLEMENTATION 30

3.1: Introduction 30

3.2: Stage 1: Setup Infrastructure 30

3.2.1: NITOS Testbed 30

3.2.2: Tools & Versions 31

3.2.2.1: Kubernetes 31

3.2.2.2: Kubeflow 34

3.2.2.3: Prometheus & Grafana 36

3.3: Stage 2: Machine Learning Application 38

3.3.1: Application Description 38

3.3.2: Deploy Process 40

3.4: Stage 3: Scheduling Algorithm 42

3.4.1: Introduction 42

3.4.2: Kubernetes Scheduling Algorithm 43

3.4.3: Kubernetes-Scheduler-Extension 44

3.4.4: Latency Awareness Scheduler 46

CHAPTER 4: EXPERIMENTS & RESULTS 49

4.1: Introduction 49

4.2: Infrastructure *49*

4.3: Experiments *52*

4.3.1: Algorithm Verification 52

4.3.2: Measurements 54

CHAPTER 5: CONCLUSION & FUTURE WORK 59

5.1: Conclusion *59*

5.2: Future Work *59*

Bibliography 61

List of Figures

Figure 1: Container Architecture....9
Figure 2: Virtual Machine Architecture....9
Figure 3: Docker Image Layers....10
Figure 4: Kubernetes Architecture....12
Figure 5: Kubeflow Central Dashboard....22
Figure 6: Notebook Server UI....23
Figure 7: Jupyter Notebook Configuration Page....23
Figure 8: Deployed Notebook Server....24
Figure 9: Pipeline UI....25
Figure 10: Machine Learning Pipeline in the form of a graph....26
Figure 11: Graph after the end of the experiment....27
Figure 12: Prometheus UI....29
Figure 13: Grafana Dashboard....29
Figure 14: NITOS Architecture....31
Figure 15: Machine Learning Pipeline definition - Python file....40
Figure 16: Import example of Pipeline component....41
Figure 17: Declaration example of Pipeline component....41
Figure 18: Machine Learning Application in the form of a graph....42
Figure 19: Extender policy file in Golang....45
Figure 20: Scheduler Algorithm Pseudocode in Golang....48
Figure 21: Policy configuration file....50
Figure 22: LAS scheduler Pod (Container: Extender & Container: Default-Scheduler)
....51

List of Charts

Chart 1: Training process (Default Scheduler) - Execution times in Seconds 55
Chart 2: Prediction process (Default Scheduler) - Latency values in seconds 56
Chart 3: Training process (LAS) - Execution times in seconds.....57
Chart 4: Training process (LAS) - Execution times in seconds.....58

List of Tables

Table 1: Access URLs for Kubeflow, Prometheus and Grafana.....38
Table 2: Hardware specifications 49
Table 3: Node Latency Values for each Location 52
Table 4: Configuration changes at Machine Learning Pipeline-Location:NITOS-CLOUD.....53
Table 5: Configuration changes at Machine Learning Pipeline-Location:NITLAB-EDGE 53
Table 6: Configuration changes at Machine Learning Pipeline-Location:HOME-EDGE 53
Table 7: Training Parameters.....55

CHAPTER 1: INTRODUCTION

1.1: Background

The evolution of **Machine Learning** in the last few years, is rapid. From the old days, researchers were interested in having machines learn from data. They wanted to make computer systems that can mimic human behavior. That is what we call nowadays Artificial Intelligence. Out of the quest for Artificial Intelligence, a new subfield grew rapidly, Machine Learning. This field enables computer systems to learn from past data (historical, numbers, images etc.) or experiences without being explicitly programmed. It uses algorithms and network models with the purpose to increase the performance of computer systems and give accurate outputs. All these algorithms and models are often used by different people (developers, researchers etc.) to create applications that are going to be used by end users. We can give some examples of such applications. First, we have facial recognition that allows social platforms to help users tag and share photos between friends. Recommendation systems, with the help of machine learning, suggest movies or series to users, based on their preference from past movies and series selections. Finally, self-driving cars have dominated the field. Cars, powered by machine learning, navigate without human intervention. Already today, some people have bought self-driving cars and use them in their everyday life.

Another field that has seen great growth in recent years is **Cloud Computing** [1]. Cloud or fog computing was the first network infrastructure that offered on-demand services through the internet. The most important ones are data storage and computational power, but the list does not stop there. Cloud computing brought revolution to the way we handle data, and the way businesses provide applications and services to their customers. Cloud Customers do not own the physical infrastructure, but they rent the usage from third party providers. There are a lot of applications of cloud computing in today's world. Some of these applications are using the cloud for storage, like Gmail or Dropbox, others for networking like healthcare

services, and others just for virtual machines that are hosted in the cloud. The popularity of cloud computing grows day by day due to numerous benefits and we are going to see a lot more people integrating their applications to the cloud as the time passes.

As we mentioned before, cloud computing services are increasing rapidly as we discover new ways to use them. However, cloud computing users as they go deeper, and use more of these services, will face limitations such as higher latency, network congestion and lower bandwidth that will prevent technology from fulfilling business requirements. Because of that, and because of the fast evolution of IoT technology, and end devices, a new data center infrastructure rose. This new type of computing was named **Edge Computing** [2]. The idea behind edge computing is distributed computation and data storage across the entire network, instead of centralizing it, into cloud. This aims to mitigate the latency and bottlenecks of an application and provides better user experience. Today, the use cases of edge computing are too many. One important example of edge computing use case is the Smart Grid. Sensors and IoT devices are connected with an edge infrastructure and provide better energy consumption. Another important example is content delivery networks. You may have seen them as CDNs which is their abbreviation. CDNs are caching content (e.g., music, video stream, web pages) at the edge, which provides lower latency and flexibility. These are only a few of the examples of edge computing that we meet every day, but it helps us understand how important it is.

Separately, each of these topics make a profound impact in the world. However, many times, they are combining their strengths to create some form of a computing continuum on which disruptive machine learning applications can be built.

1.2: Motivation

Machine learning applications, as the years go by, become more and more demanding in terms of the processing power they consume. Many of these applications are processing large amounts of data and performing heavy processes to achieve the desired output. Also, as if the processing power requirements were not enough, some of them require low latency, to provide better user experience. So, these two requirements raise the question: Where should we deploy our machine learning application? To understand how challenging the selection of an infrastructure, for such applications, is, we will use an example of real-time application.

One application that combines heavy computations and the need of low latency is an example of augmented reality application [3]. This real-time application explores the points of interest (POIs) that a tourist is currently visiting. This application involves heavy image processing that extracts features from captured images and a trained network-model that matches features from an extensive object catalog. This is a perfect example for our case because it is computation-intensive and latency-sensitive. An application like that is demanding a lot of resources, something that a home computer cannot provide. Also, to provide a quality experience to the end user, some functionality needs to be close to the user. For the purpose of this Thesis, we are going to analyze only the choices of cloud and edge infrastructure.

One idea is to deploy this application to the cloud. There, there is the illusion of infinite resources, thanks to Horizontal Scaling. So, an operation like image processing will be done relatively fast. However, while the cloud can provide vast computational resources, accessing those resources may involve multiple hops through the network. This will lead to an increase of latency in the processing of client requests. In an application like that, this is a problem, because as we said we need an infrastructure for a latency-sensitive application, where fast responses will be required. So, the cloud is not the optimal solution. Another idea is to deploy the whole application to the edge.

In edge computing [4], resources are scarce through the network and must be managed very efficiently. Especially for mobile devices, where the battery is limited to a certain amount. Here we have the advantage of fast response, as the application is closer to the client. However, in this scenario we face another problem. The computational resources are not enough for heavy processes. That will slow down the image processing of such an application, and might not even work, if the resources are not enough.

The ideal environment for such an application, as we might think, will be the cloud, for heavy image processing and the edge for the latency-sensitive part of the application. We could break this into two components and deploy each component above the right infrastructure. With this logic, the processing part will be fast and optimal, as the response part, so the user can enjoy an improved quality of experience.

Like the example above, every machine learning application can be broken down into 7 processing steps, from inception to practical application. By name these steps are: **Gathering Data**, **Preparing Data**, **Choosing a Model**, **Training**, **Evaluation, Hyper-parameter Tuning**, **Prediction.** Each of these steps is demanding different amounts of resources, as well as latency levels.

That was the motivation behind this thesis. To utilize tools and find techniques that distribute machine learning application's components based on the resource and latency demands of each component, to achieve better user experience. So, we created a **scheduling algorithm** that decides for us, where each part of the machine learning application will be scheduled either at a cloud environment or at an edge environment.

1.3: Content Overview

This Thesis is organized into five chapters, each one of those includes smaller sections and possibly subsections.

- **Chapter 1**: makes an introduction to the machine learning world and how it is connected to the cloud and edge environments. Also, reference is made to the motivation of this thesis that led to this specific implementation.
- **Chapter 2**: reference is made to the technical background and tools that the reader should be aware of, to completely understand the work behind the research.
- **Chapter 3:** is an important section of the document. First, it describes the infrastructure that was used for the research steps. Also, it analyzes the use of various tools, as well as how they interact between them, to reach the desired implementation. Finally, it describes in detail the methodology which was used to create our algorithm.
- **Chapter 4**: results and metrics are presented from the validation of the algorithm and the execution of a machine learning application, under the supervision of this specific algorithm.
- **Chapter 5**: presents the conclusions that emerged throughout the research, as well as what emerged from experiments. Also, are mentioned some future additions that can be made to the algorithm, in order the algorithm to become more global to the scheduling of machine learning application.

CHAPTER 2: TECHNICHAL BACKGROUND & TOOLS

2.1: Introduction

This chapter refers to the tools and the technical background that the reader of this thesis must have to understand it. It describes and analyzes some technologies that have been used to achieve the solution with the implementation of the algorithm.

2.2: OS-Level Virtualization

2.2.1: Overview

Operating system[1] level virtualization is a technology paradigm in which the kernel allows multiple isolated user-space instances to co-exist. These instances, also known as Containers, look like real computers from the point of view of programs and processes that run inside them. But this is an illusion as we will see in more detail below. Each Container shares the host's OS. This means that it uses the OS's normal system call interface and does not need to be subjected to emulation or be run in an intermediate virtual machine. This makes Containers very lightweight, since they require less overhead to be launched, in comparison to full virtualization technologies.

Containers [5] offer a logical packaging mechanism in which applications can be abstracted from the environment in which they run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of the target environment.

[1] Operating System: OS

2.2.1: Building blocks

As mentioned earlier, Containers are essentially just a way of partitioning up a system into a few sandboxed execution environments with their own resource limits, while all these environments share a single operating system. That is why we talked about an illusion. But how this new illusion-virtualization is done. The base of this new technology is three fundamental kernel features: namespaces, cgroups and union filesystem.

- **Namespace** [6]: The Linux Namespaces are a kernel mechanism that, at a high level, limits the visibility that a group of processes has on the rest of the system. This mechanism does not restrict access to resources like CPU or disks. it achieves isolation by exposing a specific subset of them to processes that run inside the namespace. For example, you can limit visibility to certain process trees, network interfaces, user IDs or filesystem mount.
- **Cgroups** [6]**:** Cgroups, which stands for control groups, is another Linux Kernel feature that limits and measures the total resources (CPU, memory, disk I/O, network, etc.) used by a group of processes running on a system. With cgroups, administrators can set limits to a set of processes as to how many resources they can consume.
- **Union filesystem** [7]**:** Unification filesystem is a service of Linux that allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. Contents of directories which have the same path within the merged branches will be seen together in a single merged directory, within the new, virtual filesystem.

2.3: Docker

2.3.1: Overview

Docker [8] is a set of platforms as a service (PaaS)[2] products that use OS-level virtualization to deliver software in packages. Docker uses the building blocks mentioned above to create an interface on top, to make it easier to manipulate and parameterize the lifetime of Containers. Docker can be installed in any operating system. We have already talked a little about these containers but let us give a more in-depth description of them and the difference between them and virtual machines.

2.3.2: Container

A container is a standard unit of software that packages up code and all its dependencies so the application can run quickly and reliably from one computing environment to another. Users can use this, without the fear of what environment exists underneath. This gives flexibility and portability to their application. This is very important, because it lets developers focus more on the development side of the project and not how to set up the application to different environments.

Emphasis should be given to the difference between Container and Virtual machine. Virtual Computers need an operating system to work, which makes them very slow to start and large in capacity. Also, they contain packages that many applications do not use, which add more burden to the system. Finally, they create, many times, problems in their portability from system to system and it is difficult to expand. The different architectures of the two components are shown in the pictures[3] below:

[2] PaaS: Provide cloud services to certain software.
[3] https://www.docker.com/resources/what-container

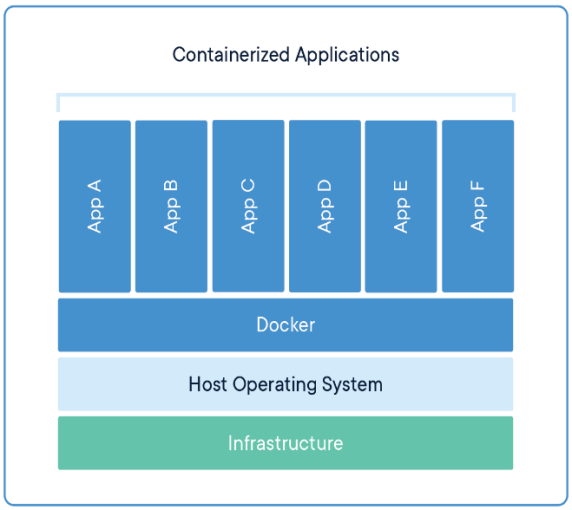

Figure 1: Container Architecture

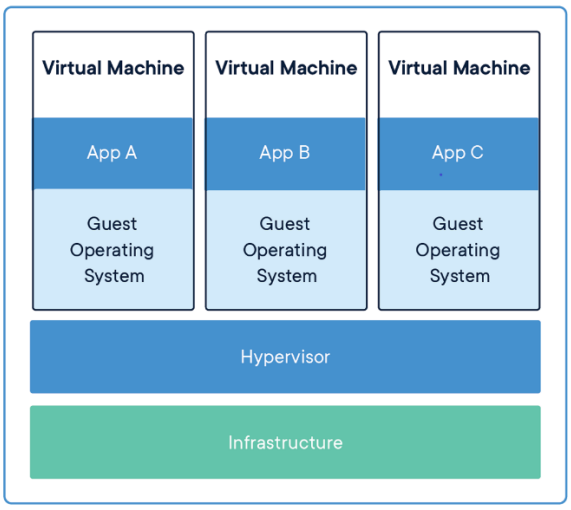

Figure 2: Virtual Machine Architecture

2.3.3: Images

Docker containers are based on Docker images. Docker Image is just a series of instructions that a docker container must follow. It is a binary that includes all of the requirements for running a single Docker container, as well as metadata describing its needs and capabilities. It has information on both the structure of the filesystem that will be used, as well as which processes will be started inside the Container. The Image is an immutable file which essentially is a snapshot of the Container. You can think of it as a packaging technology. Docker containers only have access to resources defined in the image unless you give the container additional access when creating it.

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Let us see an example to understand more about this layering. Consider the following Dockerfile:

```
# syntax=docker/dockerfile:1
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

This Dockerfile contains four commands. Each of these commands creates a layer. From top to bottom it has the FROM statement. This statement starts out by creating a layer from the ubuntu:18.04 Image. This is a prebuilt image that exists in a public registry called DockerHub and we pull it from there. After that it has the COPY command which adds some files from the Docker client's current directory. The RUN command builds your application using the make command. Finally, it has the last layer that specifies what command to run within the container. Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. These layers are read-only as shown in the image below. When a user creates a new container, he/she adds a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The image[4] below shows a container based on the Ubuntu 15.04 image.

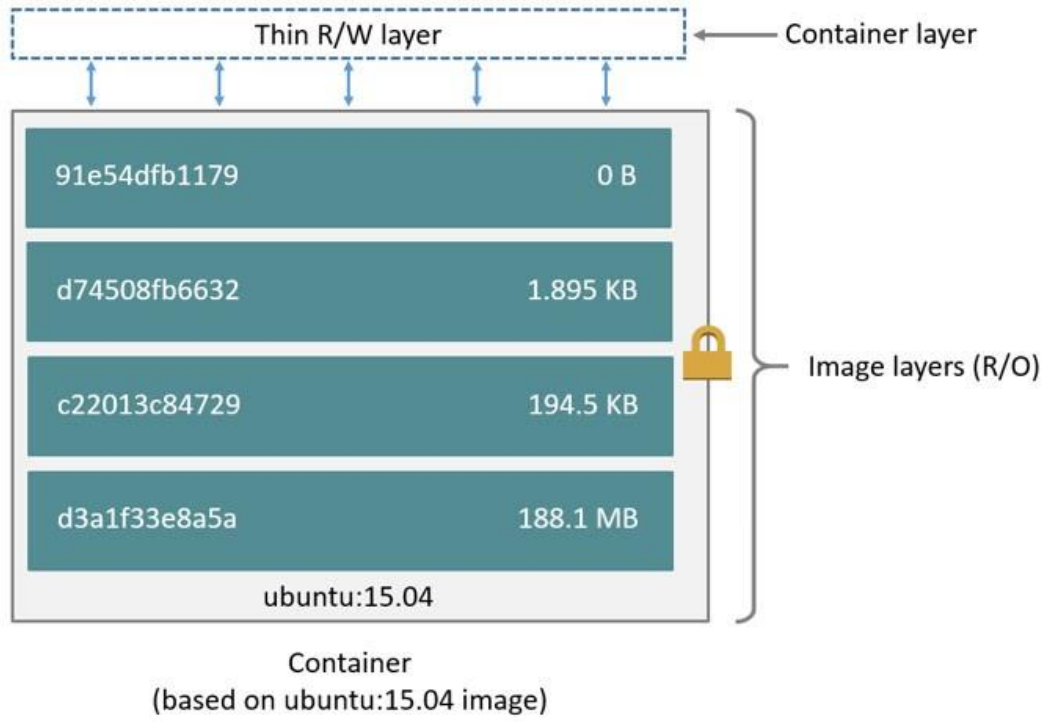

Figure 3: Docker Image Layers

[4] https://docs.docker.com/storage/storagedriver/

After the analysis of docker images, the next thing that comes to our minds is, where those images can be stored for fast access and reusability. Docker registries come to the rescue. The next section describes how they accomplish that.

2.3.4: Registries

Docker is using a distributed system for storing Docker images. This storage is called Docker registry and contains named Docker Images. Each one of these images might have multiple different versions, identified by their tags. A Docker registry is organized into Docker repositories , where a repository holds all the versions of a specific image. Users can pull images from there. The registry allows Docker users to pull images locally, as well as push new images to the registry. In this thesis the DockerHub was used , for storing the needed images and pulling from there. DockerHub is a just cloud-based public registry.

2.4: Kubernetes

2.4.1: Overview

Kubernetes [9], or else K8s, is an open-source project that helps at the organization of containers running inside nodes that belong to the same cluster. It was originally created by Google, with version 1.0 launched in 2015 and is now maintained by the Cloud Native Computing Foundation (CNCF) [5]. It has a large, rapidly growing ecosystem. This tool has many capabilities that every user can learn and use very easily. Each user can define which service wants to execute, at how many nodes, with how much resource power and other parameters. In general, K8s orchestrates computing, networking, and storage infrastructure on behalf of users' containerized workloads.

[5] Cloud Native Computing Foundation (CNCF)

2.4.2: Architecture

Kubernetes follows a client-server architecture. It consists of a master node and a set of worker machines, called nodes that run containerized applications. It is possible to have a multi-master setup, but by default there is a single master node which is the "brain" that controls the cluster.

Master Nodes provide the cluster's control plane. They make decisions about the cluster, and they detect and respond to cluster events. All these decisions are made with the help of some components called Control plane components. These can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine.

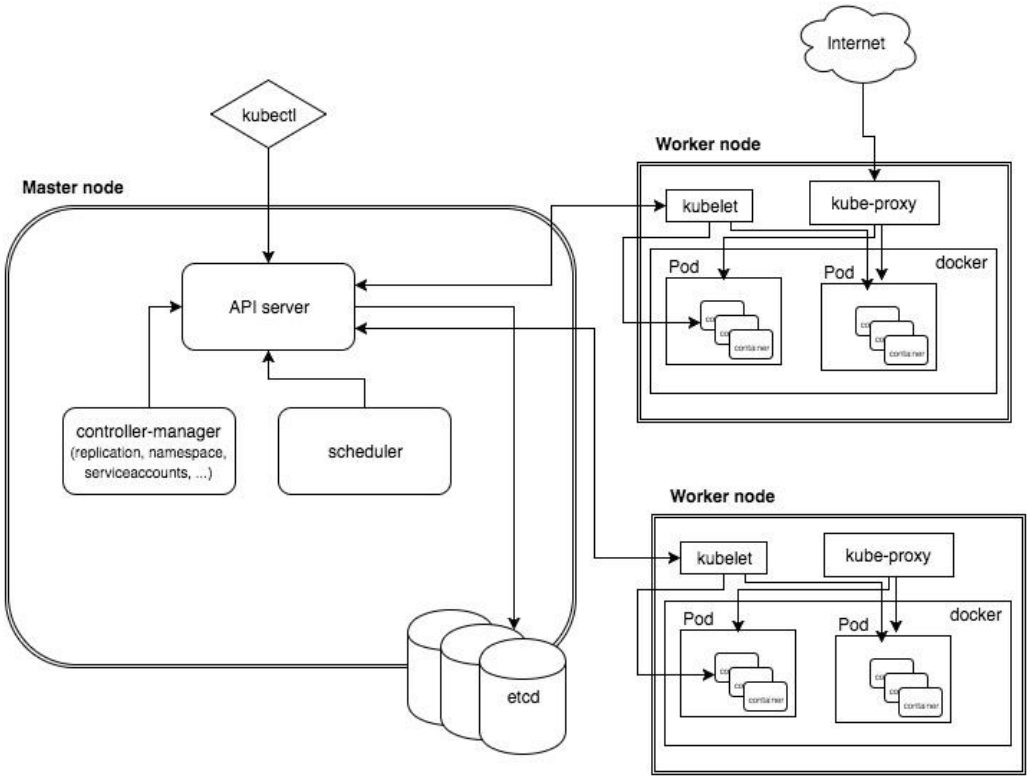

Figure 4: Kubernetes Architecture[6]

[6] Kubernetes Architecture Image

Master Node Components

Below are the main components found on the master node:

- **kube-apiserver:** As the name suggests is the component that exposes the Kubernetes API. This is the front-end for the Kubernetes control-plane. Kube-apiserver is designed to scale horizontally that is, it scales by deploying more instances. You can have many instances of kube-apiserver to balance the traffic between those instances.
- **Etcd:** Etcd is a simple, distributed key-value store, used as Kubernetes backing store for cluster data (such as number of pods, their desired state, namespace, etc.). Periodically it is important to back up those data, in case of disaster scenarios, such as losing all the control plane nodes. So, we can recover the Kubernetes cluster.
- **Kube-scheduler:** Kube-scheduler watches for newly created pods with no assigned node and selects the best fit node for them, based on resource utilization and other parameters like hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines. The kube-scheduler uses an algorithm for this decision. More about this algorithm at Chapter 3: Implementation.
- **Kube-controller-manager:** Kube-controller-manager is a control plane component that runs and manages controller processes. Controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.
- **Cloud-controller-manager:** This control-plane component does not appear in the image below, but it is very important, especially for the production environments where cloud providers are present. Cloud-controller-manager embeds cloud-specific control-logic. The cloud controller manager lets you link

your cluster into your cloud provider's API and separates out the components that interact with that cloud platform from components that only interact with your cluster.

Worker Node Components

Below are the main components found on every node:

- **Kubelet:** Kubelet is an agent that runs on each node in the cluster. It makes sure that containers are running in a pod. Kubelet is responsible only for containers that were created from Kubernetes. Also, it is the component that starts the pod after the selection of the feasible node from the kube-scheduler.
- **Kube-proxy:** Kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes network service. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.
- **Container Runtime:** The container runtime is the software that is responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd[7], CRI-O[8], and any implementation of the Kubernetes CRI (Container Runtime Interface).

2.4.3: Concepts

Kubernetes constantly monitors itself and tries to achieve the desired state of the application. This desired state is presented in a YAML file with different types of abstractions. So, it is important to understand these abstractions that are used to represent the state of the system-application such as pods, services, deployments, and namespaces.

[7] https://containerd.io/docs/
[8] https://cri-o.io/#what-is-cri-o

- **Pods:** Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.
- **Services:** Services is just an abstract way to expose an application running on a set of Pods as a network service. The question here is “Why don't we use the pods itself?”. Kubernetes Pods are created and destroyed to match the desired state of your cluster. So, they cannot have permanent IP. That is why we use services that give as a permanent IP, that we can use to access our functionality in our application. We have four types of Services in Kubernetes:
    - **ClusterIP (default):** Exposes a service that is accessible only from inside the Cluster.
    - **NodePort:** Exposes a service with each Node's IP with a static Port. This type of service is accessible from outside the cluster.
    - **LoadBalancer:** It uses the cloud’s provider load balancer to expose the service. This type of service is accessible externally.
    - **ExternalName:** It maps the Service to the contents of a predetermined externalName field by returning a value for the CNAME record.
- **Deployments:** Deployment is a way to describe the desired state of a pod or a replica set. Deployment Controller changes the state of the environment by deleting or creating replicas, until it achieves the desired state.
- **Namespace:** Namespace is just a virtual environment backed by the same physical cluster.

2.4.4: Storage

2.4.4.1: Volume

On-disk, files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. First if a container crashes the user will

lose all the data inside. Kubelet will restart the container with a clean state. Second it creates some problems when we want to share files between the containers that run in the same Pod. Kubernetes face these problems with the help of volumes.

Kubernetes offers different types of volumes. A pod can use any number of volumes simultaneously. It can use both persistent and non-persistent volumes. Non-persistent volumes are ephemeral which means that Kubernetes is going to delete them in case a pod ceases to exist. However, Kubernetes does not destroy persistent volumes. To use a volume, specify the volumes to provide for the Pod in .spec.volumes and declare where to mount those volumes into containers in .spec.containers[*].volumeMounts.

2.4.4.2: Persistent Volume

Kubernetes wanted to abstract the details of how storage is provided from how it is consumed. That is why K8s introduced two new API resources: Persistent Volumes and Persistent Volume Claims. Persistent Volume (PV) is just another piece of resource in the cluster, like a node is a cluster resource. You can provision this type of storage as an administrator or dynamically with the help of storage classes. The difference between PVs and Volumes is that PVs have a lifecycle independent of any individual Pod that uses the PV. So even if the pod crashes, the linked PV will remain untouched.

Persistent Volume Claims (PVC) is a request for storage by a user. It is like a Pod. Pods consume node resources and PVCs consume PV resources. Also, a pod can request a specific level of resources (CPU and Memory) like a Claim can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany). Each type of storage can accept specific access modes.

- **ReadWriteOnce**: The volume can be mounted as read-write by a single node.
- **ReadOnlyMany**: The volume can be mounted read-only by many nodes.

- **ReadWriteMany:** The volume can be mounted as read-write by many nodes.

2.4.4.3: Storage Class

Storage class is the dynamic way to provision a Persistent Volume. A cluster administrator can define as many storage classes as needed inside the cluster. This resource is provided as an Object from the storage.k8s.io API group. Each one of these storage classes has a provisioner (AWSElasticBlockStore, NFS[9], Local etc.) that decides what volume plugin is going to be used for the provisioning of a persistent volume. Also, we can provide some parameters for the specific provisioner. Finally, we can specify a reclaim policy which is going to decide what will happen to the PV after it has been released from its claim (PVC).

2.5: Machine Learning

2.5.1: Overview

Machine learning is one of the most important fields in modern computer science. It is a vast field with many applications in our everyday life (medicine, e-commerce, banking etc.) which grows everyday more and more. Although machine learning is part of computer science, it differs from traditional computational approaches where algorithms are built to calculate things or solve problems. In machine learning, algorithms are used to train the computer, based on some data inputs, and produce models that will be used for decision making processes. But how does it work under the hood? Each Machine Learning application can be broken down in a sequence of steps [10] that describe the application from inception to practical application.

[9] NFS: Network File System

2.5.2: Steps of machine learning application

Step 1: Data Collection

This is by far the most important step for developing the machine learning model. A user needs to gather relevant data that will help to create the most appropriate model for his/her purpose. Mistakes such as choosing the incorrect features may lead to an ineffective model. That is why it is crucial that the necessary considerations are made when gathering data as the errors made in this stage would only increase as we progress to later stages.

Step 2: Preparing Data

Once the user has gathered the data, he/she needs to prepare them. The user needs to make sure that his/her data are not biased, and they are random. This is because the users do not want the order to affect the model's decision. Also, the user needs to make sure that the data are not skewed over a specific feature .This skewness may give correct results for a particular feature but not for the rest of them. Finally, he/she needs to break the data into two parts. The training data that are going to be used for the model training, and the test data for evaluation purposes. Someone can understand that well-prepared data will improve the model's efficiency and accuracy at the prediction step.

Step 3: Choosing a Model

This step is also important for the model, because here it will be decided what logic the model will use to train itself. There are different and various models, developed by data scientists and researchers, that have been created for various purposes. Some of them are well suited for image data, others for numerical data, others for sequences (text

etc.) and other text-based data. Someone needs to make sure that the right choice is made.

Step 4: Training

The core of a machine learning application is the training step. Here the step uses the training data from the Data preparation step, to train the model to differentiate between the features. The process of training is iterative. A user trains the model repeatedly with different input from the data, until the model reaches the desired level of accuracy. So, anyone can understand that this is a long process with a lot of experimentation.

Step 5: Evaluation

Once the training is complete, it is time to see if the model is any good, with Evaluation. In this step the user uses the test data that has never been used for training. This metric will show how the model might perform in real world situations, where data are not known to the model. If the results of the evaluation step are not satisfactory, he/she needs to revisit the prior steps, and find the root that causes this underperformance of the model.

Step 6: Hyperparameter Tuning

At the Hyperparameter Tuning step someone can change different parameters to improve the model accuracy. Two important parameters are the number of training steps and learning rate. The first one is how many times we run through the training dataset during training. This may lead to higher accuracy. The second one is important for the size of each step at each iteration. How much a user shifts his/her step, based on the information from the previous training step. There are a lot more parameters that we can use for this step but those two are the most important ones. All these parameters play an important role in how accurate the model can be.

Step 7: Prediction

The final step of a machine learning application is the prediction. This is the step where the user uses the model for real world and practical applications. The model should be ready to answer questions, without human interference. This is also the challenge for machine learning applications, to be able to outperform or at least match human judgment in different scenarios.

These seven steps describe a complete real world machine learning process. This is a highly collaborative process that demands efficient communication between engineers that are working on each step. Below we will describe a tool that can break these steps in different components and how it connects these components for an efficient workflow.

2.6: Kubeflow

2.6.1: Overview

Kubeflow [11] is a free and open-source project designed to orchestrate complicated Machine Learning workflows running on Kubernetes. Kubeflow was first released in 2017, built by developers from Google, Cisco, IBM, Red Hat and more that continue to contribute to Kubeflow project. Since then, it has expanded into a multi-architecture, multi-cloud framework for running huge machine learning pipelines. But what exactly is Kubeflow?

Kubeflow is just a platform for data scientists who want to build and experiment with machine learning pipelines. Also, Kubeflow can help operational teams and machine learning engineers to deploy their machine learning workflow to different environments for development, testing and production-level serving. We can think of it as a machine

learning toolkit for Kubernetes that simplifies the deployment process of machine learning applications on K8s.

2.6.2: Architecture

Kubeflow aims to provide a cohesive experience regarding creation, management, and deployment of machine learning applications on Kubernetes. To achieve this, it provides a wide set of machine learning tools that are deployed on top of Kubernetes. We know that machine learning workflow consists of several stages, that is why we have different Kubeflow tools for each of these stages. Some of these tools are Jupyter Notebooks, TensorBoard, kfServing, Pipelines and more.

Reference must be made to the Central Dashboard that Kubeflow Deployment provides us. This Kubeflow UI provides quick access to Kubeflow components that are deployed in our cluster. Except for the shortcuts, this component displays a list of recent pipelines, notebook servers, and metrics, that gives us an overview of our jobs, and our cluster.

Below we briefly describe the UI components of the central dashboard - Kubeflow UI.

- **Home**: Kubeflow Dashboard for navigation between components.
- **Pipelines**: Kubeflow Pipelines dashboard (Upload, Deletion, management of pipelines).
- **Notebook Servers**: Dashboard for creation and delete of Kubeflow Servers.
- **Katib**: Tool for hyperparameter Tuning.
- **Artifact Store**: Tool for tracking artifact metadata.
- **Manage Contributors:** Share user access across namespaces.
- **GitHub:** Open-source project repository

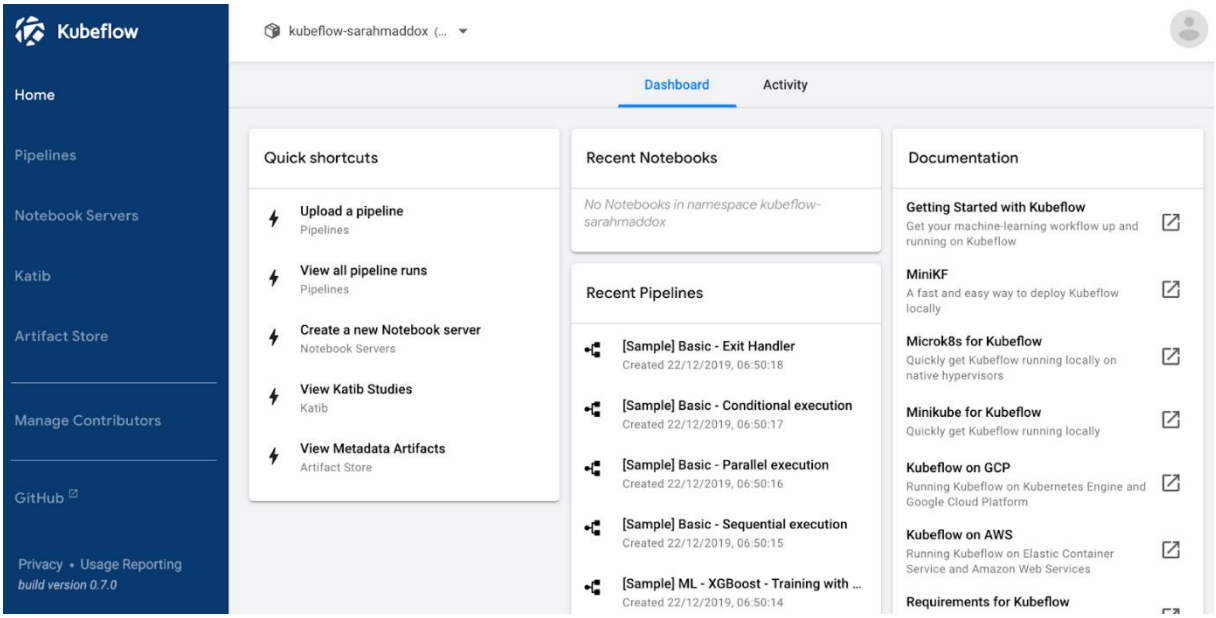

Figure 5: Kubeflow Central Dashboard

2.6.3: Notebook Servers

One of the most important components in Kubeflow is Notebook Servers. Enterprise environments have benefited a lot from the integration of Jupyter notebooks in the Kubeflow. First, it is much easier to deploy a Jupyter notebook directly into our cluster, rather than locally on our workstations. It is much faster and simpler. Each admin can provide a standard notebook image for each developer in their organization. After that with some credentials each user will have access to different notebooks. Overall, Kubeflow - hosted notebooks are better integrated with other components while providing extensibility for notebook images.

Below we set up an example Notebook within our cluster.

When we first login, we see the home page of the central dashboard. In the left-hand panel we select the **Notebook Servers** choice to access the Jupyter Notebook services that are deployed in the Kubeflow.

Before we select the **NEW SERVER** choice, we must select the namespace that corresponds to our Kubeflow Profile. For this example, we created a profile called: kubeflow-nickangelopoulos as we can see from the image below.

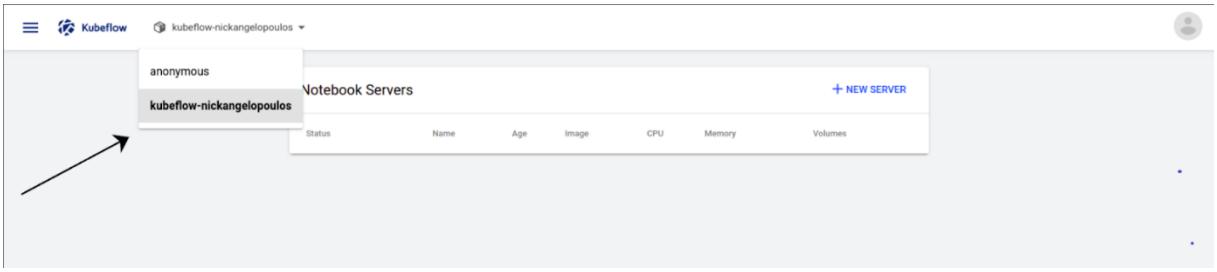

Figure 6: Notebook Server UI

After that select, the NEW SERVER button on the Notebook Server page to access the configuration page where we can specify the details for our new Notebook Server.

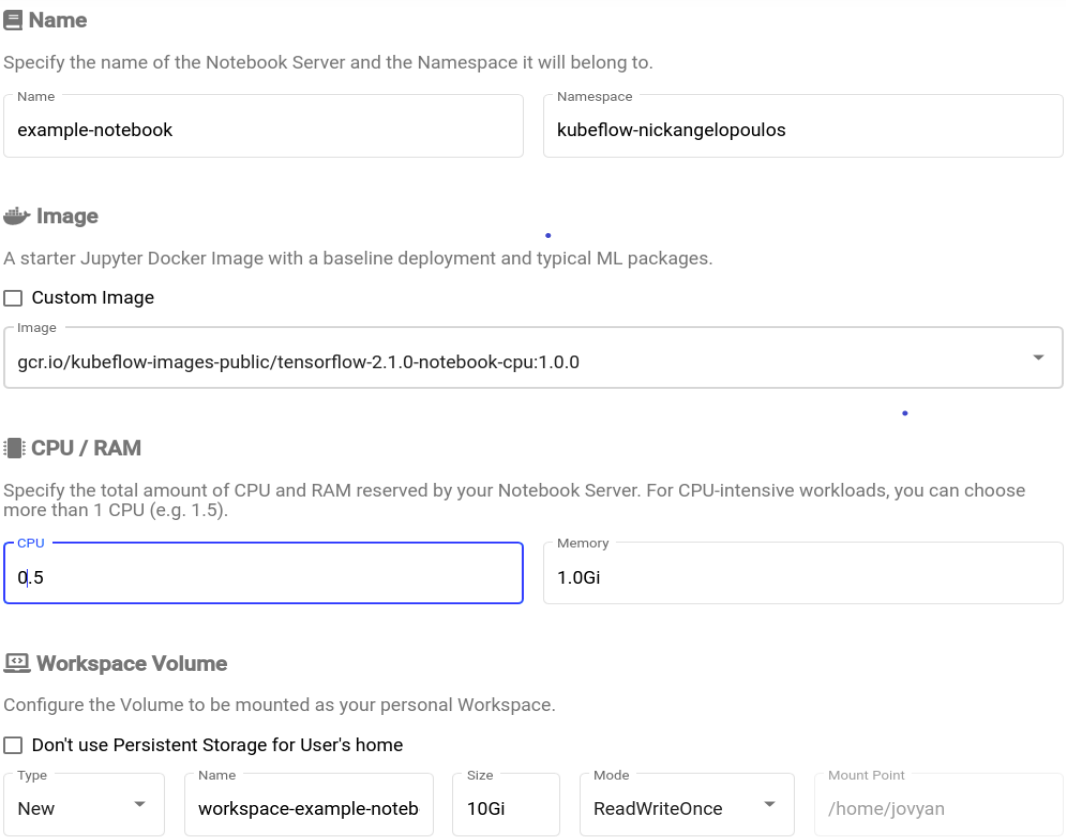

Figure 7: Jupyter Notebook Configuration Page

Here we can specify some details for our Notebook Server:

- **Name**: Enter the name of our choice.
- **Namespace**: Kubeflow automatically updates the value in the namespace field to be the same as the namespace that we selected in a previous step.
- **Image**: We specify the docker image of our choice, for the baseline deployment of our notebook server. We have two choices:
    - **Custom Image**: We must specify a custom image in the form of registry/image: tag.
    - **Standard Image**: Kubeflow provides us with a list of available images, that include typical machine learning packages that we can use within our Jupyter notebooks. For this example, we selected one from the standard Images.
    - **Resources**: One of the most important details we can specify are the resources of the Jupyter Notebook. How much **CPU** or **MEMORY** this Notebook can utilize**?**
    - **Workspace**: Finally, we can specify a workspace volume to hold our work inside the Jupyter Notebook. This type of workspace is persistent, which ensures that you can retain data even if you destroy the notebook.

These are the necessary details we must specify in this configuration page. This page provides us also, with some optional choices, that we are not going to analyze here. After that we press the Launch button and wait until the Notebook is deployed like the image below.

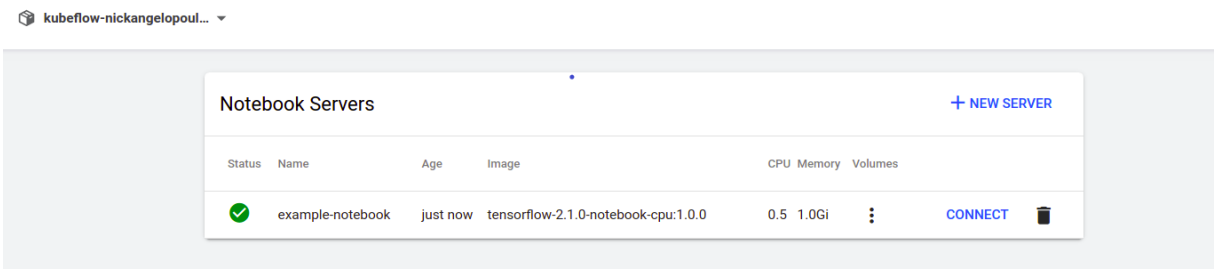

Figure 8: Deployed Notebook Server

2.6.4: Kubeflow Pipelines

In general, in programming, a pipeline is a set of data processing components, connected in a series. Each component has some outputs that are the inputs of another element. In Kubeflow we have the Kubeflow Pipeline which is a description of a Machine Learning workflow. This component is the most important component in Kubeflow that helps Machine Learning engineers and operations systems to describe their Machine Learning workflows as a simple graph. From a technical perspective when we run a machine learning pipeline, the system launches one or more Kubernetes Pods, corresponding to the steps of our workflow.

Below we represent a simple example of a pipeline. The example pipeline is one of the Kubeflow's examples called **[Tutorial] Data passing in a python component.** First, we click the name of the sample on the Pipeline UI.

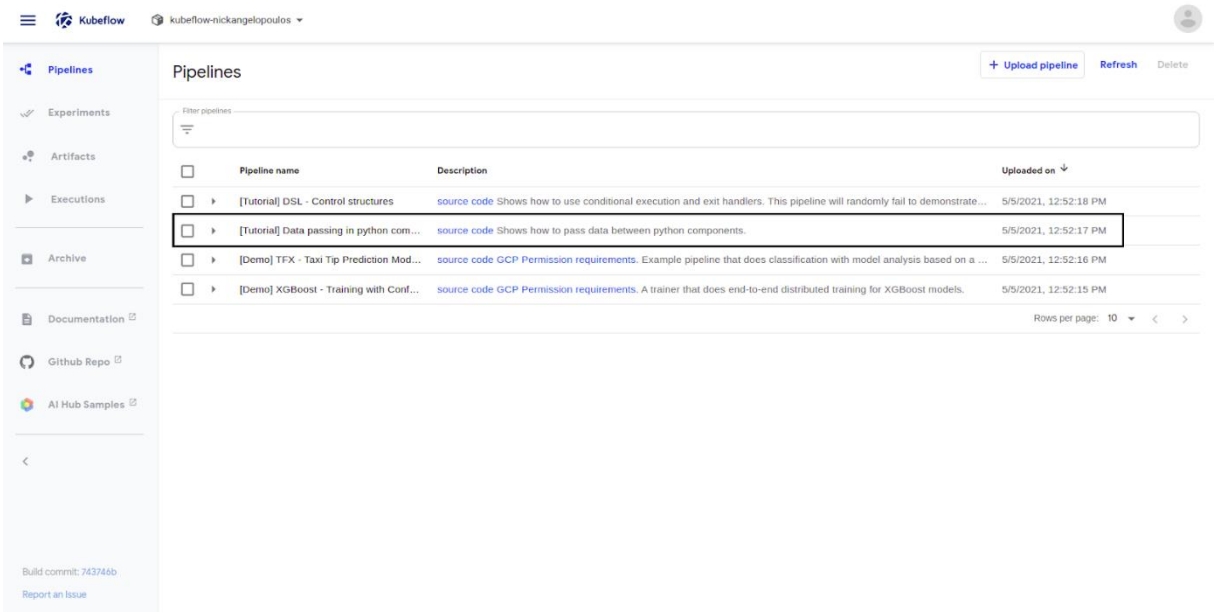

Figure 9: Pipeline UI

After that we see the pipeline in the form of a graph, where each component connects with other components with arrows. There we see that we have some choices for our pipeline.

- **Create run:** Create a sample run for our pipeline.
- **Upload Version**: Upload a new version of a pipeline, where we could have made some modifications for this specific pipeline.
- **Create experiment:** An experiment where we can run a lot of run samples.
- **Delete:** Remove this pipeline from our workstation.

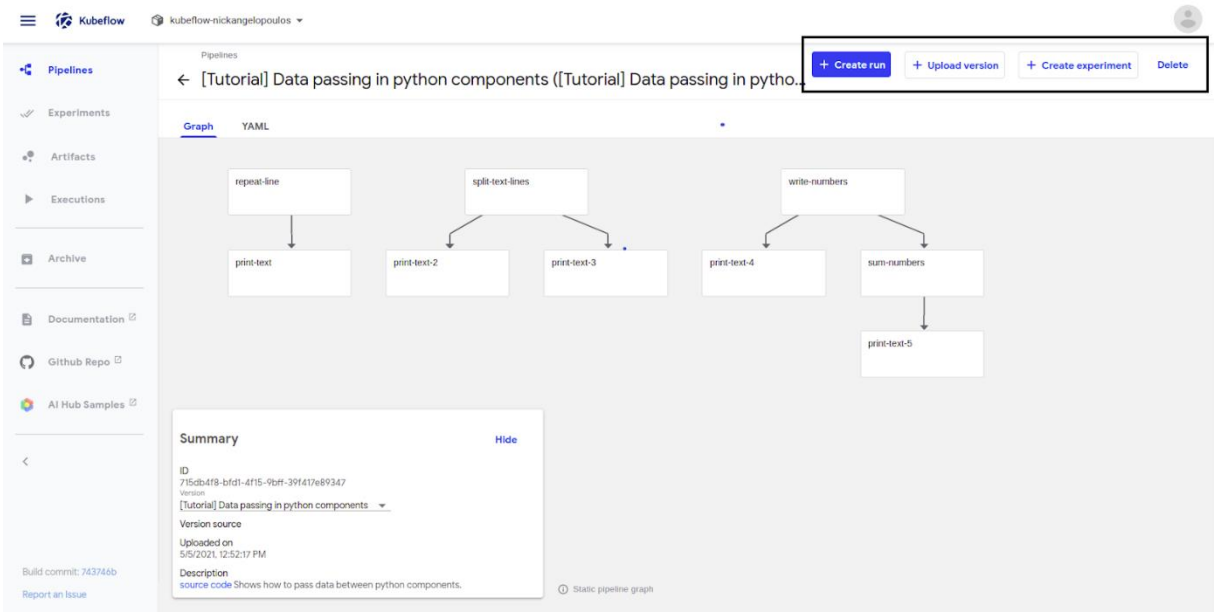

Figure 10: Machine Learning Pipeline in the form of a graph

After the end of the experiment, we click the name of the run from the experiment's dashboard. This leads us into a graph dashboard, where we can explore the graph and other aspects of our run, by clicking on the components.

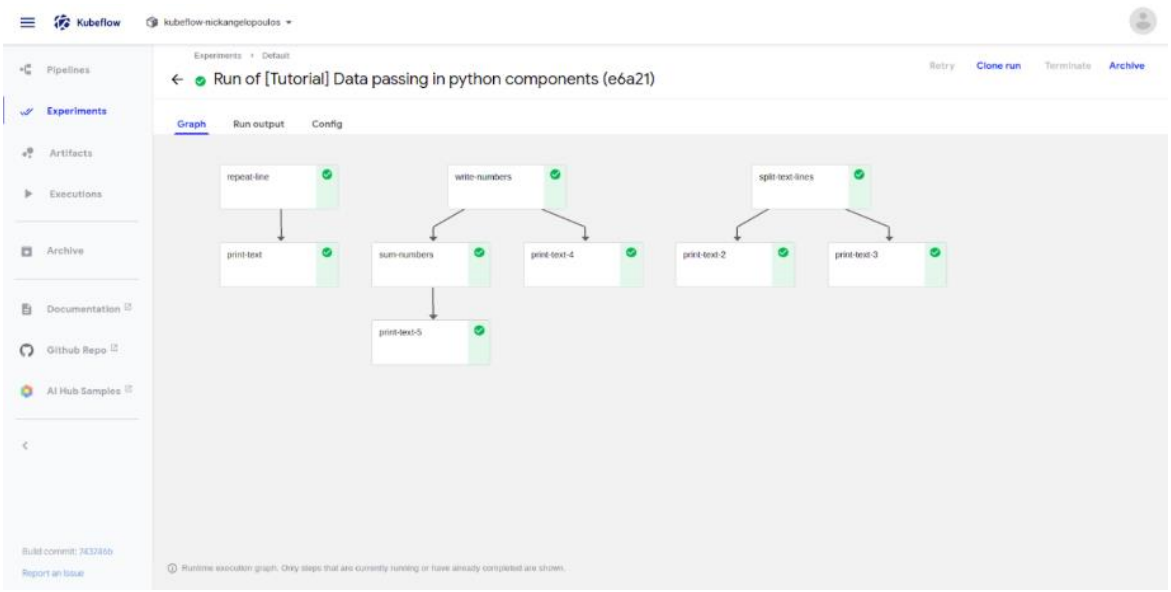

Figure 11: Graph after the end of the experiment

After we have seen this example, we can understand that Kubeflow pipelines want to create an environment where the machine learning development and management will be simpler and easier. To achieve this, they provide this end-to-end orchestration system which is easy to understand and use with the help of UIs.

2.6.5: Software Define Kit

If we want to define SDK, we could say that it is a collection of software development tools in one installable package. Kubeflow provides us with an SDK that consists of python packages that we can use to specify and run our Machine Learning workflows. These tools help us to convert our python machine learning application into a YAML file that the pipeline interface will understand. Also, some of these packages help us use already existing YAML, pipeline components from other developers. Below we will name and give a definition for some of these packages.

Packages

- **kfp.compiler:** Consists of classes and methods that will be used to compile pipeline DSL into a workflow YAML.

- **kfp.components**: Consists of classes and methods that will be used to interact with pipeline components.
- **kfp.dsl:** contains the domain-specific language (DSL) that you can use to define and interact with pipelines and components.
- **kfp.client:** Contains Python libraries that every user can use to interact with the Kubeflow Pipelines API.

2.7: Metrics & Display

Monitoring your jobs that run inside a cluster is very important. A user can understand which node or specific job consumes more resources, and after that the user can improve the performance or optimize the deployment, based on the information. There are many tools on the internet that can scrape data from a running cluster. The cluster at this Thesis uses Prometheus for scraping the metrics and Grafana to display them.

2.7.1: Prometheus

Prometheus [12] is the de facto standard metrics solution in the cloud . It is an open-source project that provides a monitoring and alerting toolkit based on a time-series data model. Since its inception in 2012, many companies and organizations have added Prometheus in their stack. Also, like Kubernetes, that we analyzed before, Prometheus joined Cloud Native Computing Foundation in 2016 which is a vendor home for fast-growing open-source projects.

Prometheus collects data from different services and stores them inside his server with a specific format. Metric name and a millisecond-precision time stamp. This storage system allows Prometheus to query metrics fast and efficiently. Unlike other monitoring tools which communicate with an agent deployed on the service's host machine, Prometheus uses exporters to receive the metrics. Also, it provides an

interface that a user can use to query metrics from the server. This interface is presented with an image below:

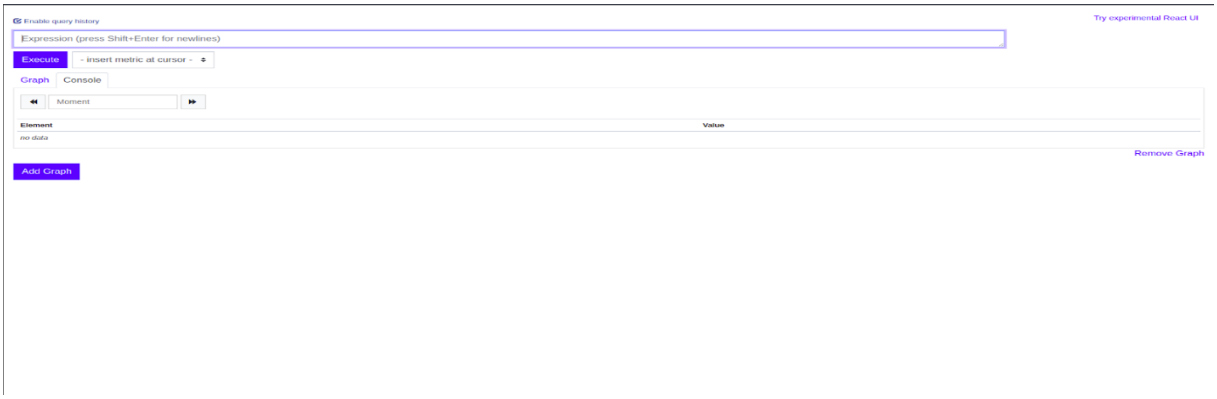

Figure 12: Prometheus UI

2.7.2: Grafana

Grafana [13] is a powerful tool for displaying time-series data. It is the platform that you need when you want to visualize and analyze the metrics. Grafana query the metrics from the Prometheus server and display them to the dashboard. Also, it comes with the ability to upload existing dashboards with unique ids based on your needs. Below we can see an example of the dashboard that this system uses.

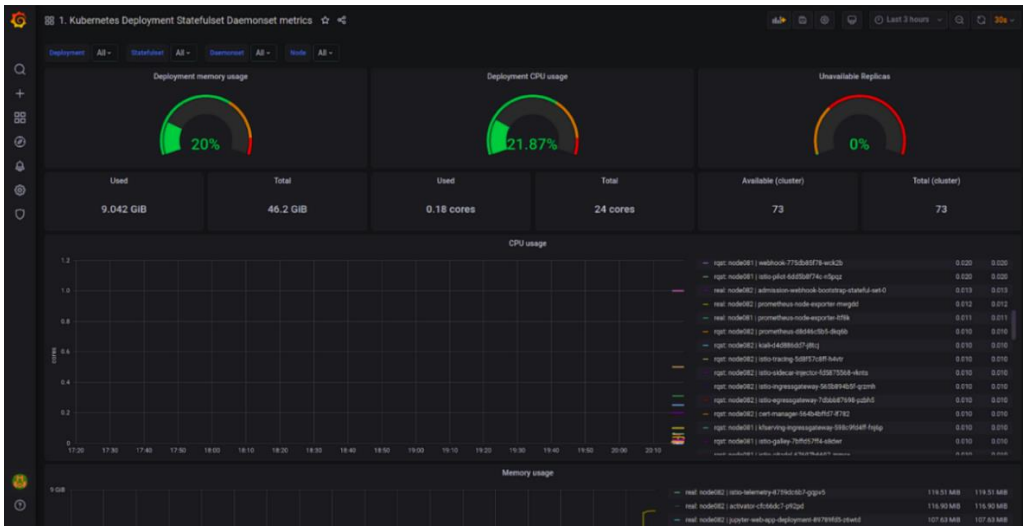

Figure 13: Grafana Dashboard

CHAPTER 3: IMPLEMENTATION

3.1: Introduction

In this chapter, we are going to cover all the steps that we followed, to achieve the implementation of this algorithm. More specifically, we divided our implementation in three stages. In the first stage, we talk about the support infrastructure, on which all the tools were installed and analyze each of these tools. In the second stage, we analyze the machine learning application that was used for the whole research. Finally, we are describing the process, and the development of the algorithm, from Kubernetes Scheduler to our implementation.

3.2: Stage 1: Setup Infrastructure

3.2.1: NITOS Testbed

For our implementation we used 3 nodes, from the **NITOS** testbed [14]. The NITOS Testbed consists of 2 wireless testbeds, one indoor and one outdoor, for experimentation with heterogeneous technologies. It was conceived and developed from NITLab (Network implementation Testbed Laboratory). The indoor testbed consists of high-processing-powered and cutting-edge nodes and is located at NITLab building.The outdoor testbed is located at the exterior of University of Thessaly (**UTH**) campus building. It features WiMAX, Wi-Fi, and LTE support. Both testbeds are a powerful tool that enables the experimentation and the implementation of different algorithms and protocols, in a large-scale environment.

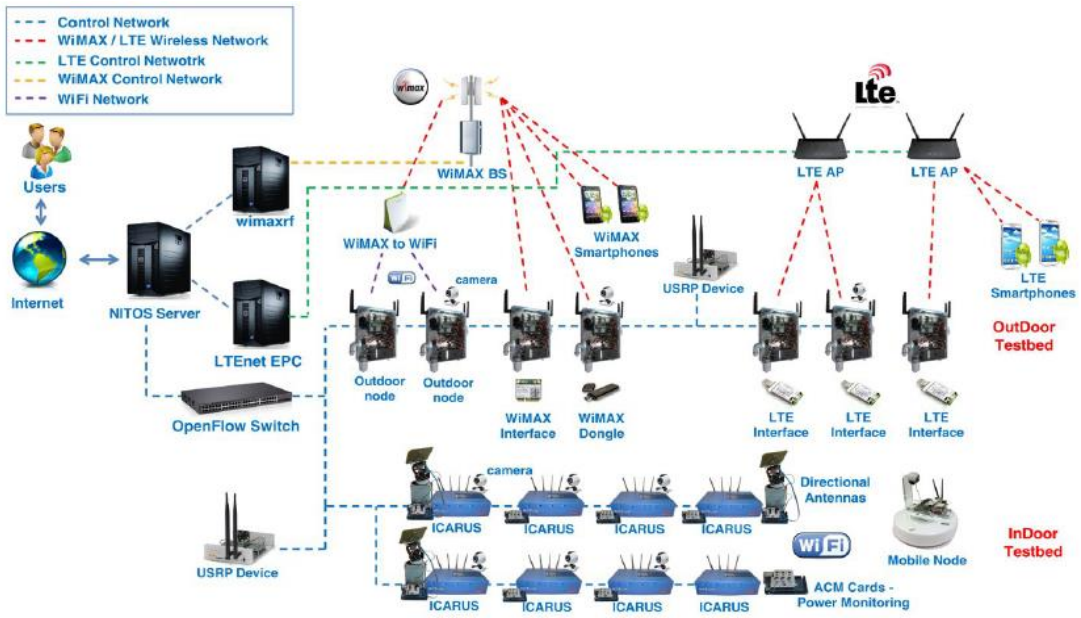

Figure 14: NITOS Architecture

3.2.2: Tools & Versions

3.2.2.1: Kubernetes

The cloud consists of many computers, who must be connected with some kind of networking, in order to utilize their resources like a group and not as single entities. Kubernetes is the best tool for this job. It can create a group of computers, called clusters, so that users can deploy their applications in the form of containers. So, on top of our three nodes infrastructure, we installed **Kubernetes**. Specifically, we installed the version 1.14.10 because this version was compatible with the **Kubeflow** version 1.0 that we installed above the Kubernetes cluster.

After the installation process we had to find a way to create a viable Kubernetes cluster. For this purpose, we used the **Kubeadm** [15] tool. This is a great tool, for an automatic way of setting up a cluster for testing and experimentation purposes. Below we display the commands that we should run to start our Kubernetes cluster.

```
$ kubeadm init --apiserver-advertise-address=10.64.X.X --pod-network=192.168.0.0/16
```

With the above command we initialize the control plane node, which is the master node of our cluster. The first argument specifies the network interface that will advertise the API server of the control plane. This argument is optional, which means that if we do not define it, it is going to be the default gateway of the node. The second argument, which is also optional, specifies the range of IPs for the pods that is going to be created.

Now it is time for kubeadm to run some pre-flight tests, to ensure that the computer is ready to host a Kubernetes cluster. These tests will expose warnings and exits on errors. After these tests, the command will automatically create the necessary cluster resources that a minimum viable Kubernetes cluster should have, to run smoothly. These components are: **kube-scheduler**, **kube-controller-manager**, **kube-apiserver**, **etcd** for the control plane and **kubelet**, **kube-proxy** and **container runtime** for all nodes. Below we display the results from our terminal.

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:
  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 10.64.93.90:6443 --token 6uaumt.przrqldf307zh9q0 \  --discovery-token-ca-cert-
hash sha256:ce044c74233831bf9e779edf798e373aa88217e7df90849fd73ed6c0a0b87f5d
```

We can easily understand that the control-plane has been initialized successfully with the necessary components. Next, we must create the **./kube** directory and copy the configuration file from the master node into our machine. This will enable us to send

requests to our cluster as regular users through the **API Server**. After the initialization of the control plane, it is time to add some workers inside our cluster. Kubeadm provides us with the join. This simple command lets us add a new worker node, inside our cluster. The user is now ready to check the status of the cluster with the command below. This command will list the nodes in our running cluster and provide us with some information about them, like the duration that the node is active, the status, the role of the node and the version of **Kubernetes**.

```
$ kubectl get nodes
NAME      STATUS    ROLES    AGE     VERSION
node081   NotReady  master   18m     v1.14.10
node082   NotReady  <none>   2m42s   v1.14.10
node083   NotReady  <none>   114s    v1.14.10
```

From the above list of nodes , we observe the **Not Ready** status, at the status column. This is because we have not configured our nodes with a network policy, and they will not be able to communicate with each other. So, for network configuration, we choose the **Calico Network**. This is a container network solution for Kubernetes clusters. To deploy this network, we must create the calico.yaml , which we took from the official calico website[10].

```
$ kubectl create -f https://docs.projectcalico.org/v3.14/manifests/calico.yaml
```

After the deployment of the **Calico Network**, we can check the status of Kubernetes cluster, with the same *kubectl get nodes* command and see that all the nodes are ready now to host some pods, except of course, the master node. Finally, we can check the running pods inside our cluster with the command below. This command informs us about the health of our pod, the namespace that is deployed, how many times it restarted before starting to run, and the age of the pod.

[10] https://www.projectcalico.org/

```
$ kubectl get pods -A
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	calico-kube-controllers-865795499c-6gqcb	1/1	Running	0	18m
kube-system	calico-node-7fzvg	1/1	Running	0	18m
kube-system	calico-node-sltwl	1/1	Running	0	18m
kube-system	calico-node-vthgd	1/1	Running	0	18m
kube-system	coredns-6dcc67dcbc-jkqgn	1/1	Running	0	46m
kube-system	coredns-6dcc67dcbc-pwjnk	1/1	Running	0	46m
kube-system	etcd-node081	1/1	Running	0	45m
kube-system	kube-apiserver-node081	1/1	Running	0	45m
kube-system	kube-controller-manager-node081	1/1	Running	0	45m
kube-system	kube-proxy-7rxc5	1/1	Running	0	30m
kube-system	kube-proxy-fvvxn	1/1	Running	0	46m
kube-system	kube-proxy-qkkkl	1/1	Running	0	29m
kube-system	kube-scheduler-node081	1/1	Running	0	45m

3.2.2.2: Kubeflow

As we already described, in the Chapter 2: Technical Background, every machine learning application can be broken down in a sequence of steps. Each step has different resource requirements like individual operations. So, our idea was to find a tool that would allow us to define each ML step as a separate process. The perfect tool for this job is **Kubeflow**. This platform with the pipeline feature will give us the opportunity to define **components** that will encapsulate each **ML step**. In general, each one of these components is a kubernetes container inside our cluster. But more on that in the next Stages. Now we must deploy Kubeflow on top of the Kubernetes cluster that we initialized in the previous section.

Before starting with the deployment process, it is important to say that the Kubeflow deployment requires dynamic persistent volume provisioning. We need a StorageClass for that purpose. So first we create a Storage class deployment. By default, the provisioner will be installed at the namespace *local-path-storage.*

```
$ kubectl apply -f
https://raw.githubusercontent.com/rancher/local-path-provisioner/master/deploy/local-path-storage.yaml[11]
```

After the installation we can check if the storage class is deployed successfully:

```
$ kubectl get sc
NAME                  PROVISIONER           AGE
local-path (default)  rancher.io/local-path  4s
```

Storage class needs to be default, to be able to provision resources dynamically. Now it is time to follow the installation process for the Kubeflow deployment. First, we should download the binary **kfctl.** This binary will be used to create the deployment. After that we create the environment variables, that is going to make the installation process much easier:

```
$ export KF_NAME= Kubeflow
$ export BASE_DIR= ~/Kubeflow/
$ export KF_DIR= ${BASE_DIR}/${KF_NAME}
```

We can check the resources deployed in the *Kubeflow* namespace. These are only a few of the pods from the Kubeflow deployment.

```
$ kubectl -n kubeflow get all
NAME                                          READY   STATUS    RESTARTS   AGE
admission-webhook-bootstrap-stateful-set-0    1/1     Running   0          15m
admission-webhook-deployment-64cb96ddbf-v9k88 1/1     Running   0          15m
application-controller-stateful-set-0         1/1     Running   0          16m
argo-ui-778676df64-jbk5d                      1/1     Running   0          15m
…….
tf-job-operator-7d7c8fb8bb-gkqhd              1/1     Running   0          15m
workflow-controller-945c84565-9f5kq           1/1     Running   0          15m
```

[11] https://github.com/rancher

Like any application that we want to access outside of our cluster, Kubeflow needs to have a NodePort Service that exports its application. As we already mentioned, NodePort is a Type of Kubernetes Service that enables an application to be accessible outside of the cluster. This is provided to us from the istio-ingress gateway service at port 31380.

3.2.2.3: Prometheus & Grafana

The final step, in the setup process, is to deploy monitoring tools for research purposes. As we already mentioned, there are many tools for this job. However, Prometheus and Grafana are the best ones, because they integrate very well with Kubernetes. Together, they provide a set of tools and graphs that makes fetching and display of metrics very simple. Grafana query the data that the Prometheus server exports from each node and depicts them on different types of custom graphs.

There are many ways to install applications on top of Kubernetes clusters, but **Helm** is the easiest one. Helm is a Kubernetes package manager. It is the Kubernetes equivalent of *apt* command in Ubuntu machines. Helm gives us the opportunity to install packaged applications in the form of **charts**. Each of these charts are simple Kubernetes YAML files, combined in a single package that can be advertised in your Kubernetes cluster. This makes the installation process of containerized applications much easier and simpler.

After the Helm installation we are ready to start with the Prometheus-Grafana installation process. The first step is to add the chart's repositories. Each repository contains the charts for each tool and is maintained by open-source contributors. The second step is to fetch the values of the charts locally to our computer, to make some configuration changes. We must export the Prometheus and Grafana services outside of our cluster. For this purpose, we again use a NodePort Service that is provided by Kubernetes. We are going to export those two services at two different ports (31323 -

31322) . Finally, we need to deploy the values.yaml to Kubernetes cluster and wait until both tools are up and running.

Prometheus-Steps

- helm repo add prometheus-community https://prometheus/community.github.io/helm-charts
- helm inspect values prometheus-community/prometheus > /tmp/prometheus.values
    - vim /tmp/prometheus.values
    - search: Service
    - Service(NodePort):
        - nodePort: 32323 (external port)
- helm install prometheus-community/prometheus --name prometheus --values=/tmp/prometheus.values

Grafana-Steps

- helm repo add grafana https://grafana.github.io/helm-charts
- helm inspect values grafana/grafana > /tmp/grafana.values
    - vim /tmp/prometheus.values
    - search: Service
    - Service(NodePort):
        - nodePort: 32322 (external port)
- helm install grafana/grafana --name grafana --values=/tmp/grafana.values

Now we can access Kubeflow, prometheus and grafana dashboards from the URLs below.

Table 1: Access URLs for Kubeflow, Prometheus and Grafana

TOOLS	URLs
Kubeflow Dashboard	http://cluster_node_ip:31380
Prometheus Dashboard	http://cluster_node_ip:32323
Grafana Dashboard	http://cluster_node_ip:32322

3.3: Stage 2: Machine Learning Application

3.3.1: Application Description

After the deployment of the tools in the Kubernetes Cluster, we had to create or find a machine learning application which we would use both at the research and at the experimental stage. This application was selected according to some criteria that we needed for the research:

- The application should be able to break in the steps that constitute a machine learning application(Data Fetching, Training, Evaluation etc.), with the help of Kubeflow, to deploy each step as a separate process.
- Also, we should be able to change the computational power that the training step needs, by modifying only some parameters. This will help us take measurements and evaluate the algorithm in the experimental stage.

For our scenario we used one of the Kubeflow pipelines examples. This example was already converted in a Kubeflow pipeline, which helped us a lot. The only thing that we

had to do, was to modify the code and the steps according to our needs. The purpose of the example was to train an initial model with XGBOOST algorithm, evaluate it and calculate the metrics. If the model error was too high, then more training was performed until the model was good. For this sample, the Taxi Trips dataset from Chicago[12], was used and the predictions were made for the taxi drivers tips. So, the next step was to modify this pipeline to match our needs. After the changes, the Machine Learning application included the following steps:

- **Data Fetching**: This step is responsible for fetching the Train and Test data from the Taxi drivers dataset. The first one will be used for the training process, and the second one for the evaluation. Here it is important to emphasize that the data fetching was broken into two different processes with the help of Kubeflow. One for fetching Training Data, and the other one for fetching Test Data.
- **Training:** This step uses the same algorithm as the example, the XGBOOST algorithm. XGBoost stands for "Extreme Gradient Boosting". It is a decision-tree based, machine learning algorithm. This algorithm is very popular with structured/tabular data. From technical perspective it was used the xgboost library[13], which is an optimized distributed algorithm, designed to be flexible and portable.
- **True Values Preparation:** The Preparation step is also broken down into two different processes like the Data Fetching Step. This process cleans the test data by removing the not needed header. Next, it extracts the true values column from the data set , which will be used at the evaluation step.
- **Prediction:** This step is responsible for taking the model that the training step generated and the test data and producing predictions. These predicted values are going to be used in the next step.

[12] Chicago Taxi Trips Dataset
[13] https://xgboost.ai/

- **Evaluation:** Last but not least, it evaluates the prediction with the help of true values that it extracted from a previous step and creates an accuracy percentage.

According to the above, this application satisfies one of the two criteria of the research requirements, that we listed above. For the second need, the XGBoost provides two parameters that a user can modify, to change the computational demands of the application. These parameters are the **learning rate** and **number of iterations.**

- **Number of iterations**: Number of training steps. More steps mean, heavier training process.
- **Learning Rate**: It is a configurable hyperparameter, used in the training process. The learning rate controls how fast the model is adapted to the problem. It is like the magnitude of each training step in a training process. In general, someone needs the lowest possible learning rate. However, this means that it needs more steps, to reach the optimal training (heavier training process).

3.3.2: Deploy Process

After the explanation of the used machine learning application, it is time to deploy it, as separate processes with the help of Kubeflow. As already mentioned, Kubeflow, provides a Software define kit that includes a set of packages, which a user can use to convert the machine learning application in a Pipeline. From these packages we used the **kfp.dsl.pipeline** for defining the pipeline inside a python file:

```
@kfp.dsl.pipeline(
    name='DEMO-MACHINE LEARNING APP',
    description='My machine learning pipeline'
)
def train_until_good_pipeline():
    #machine-learning component-1
    #...
    #machine-learning component-n
```

Figure 15: Machine Learning Pipeline definition - Python file

For each component of the pipeline, we used the library **components** from the **kfp** package. This lets us import all the YAML files for each component of the machine learning app. Here it is important to emphasize that all the YAML-pipeline components were taken from the Kubeflow GitHub repository[14] which is open-sourced. An example of such an addition is the following:

```
chicago_taxi_dataset_op = components.load_component_from_file(
    "./yaml_files/Dataset_Chicago_Taxi_Trips/component.yaml")
```

Figure 16: Import example of Pipeline component

Also, after the addition, we need to define and use this component inside the pipeline. Below we provide a code example of adding the above component as a pipeline step.

```
training_data = chicago_taxi_dataset_op(
    where='trip_start_timestamp >= "2019-01-01" AND trip_start_timestamp < "2019-05-01"',
    select='tips,trip_seconds,trip_miles,fare,tolls,extras,trip_total',
    limit=15000,
).output
```

Figure 17: Declaration example of Pipeline component

All the above variables (where,select,limit) are predefined variables from the imported YAML file. The first one specifies what part of the dataset we want. The second one declares the features that we want and the last one the maximum amount of data.

In the same way, we imported and declared the other components of the machine learning application. The Fetching of test data, the training step, values preparation step ,the prediction step, and the evaluation step. After defining the pipeline in Python, we must compile the pipeline to an intermediate representation before we can submit it to the Kubeflow. This representation is a workflow specification in the form of a YAML

[14] https://github.com/kubeflow/kubeflow

file compressed into a **.tar.gz** file. For this process we need to install the **Kubeflow Pipelines SDK** and run the below command at the terminal:

```
$ dsl-compile --py=demo_machine_learning.py --output=demo.tar.gz
```

Next, we upload the demo.tar.gz file to the Kubeflow Dashboard and generate the pipeline. The process is the same as we have shown at the 2.6.4: Kubeflow Pipelines. Now we can look at the graph of our application in the Pipeline UI.

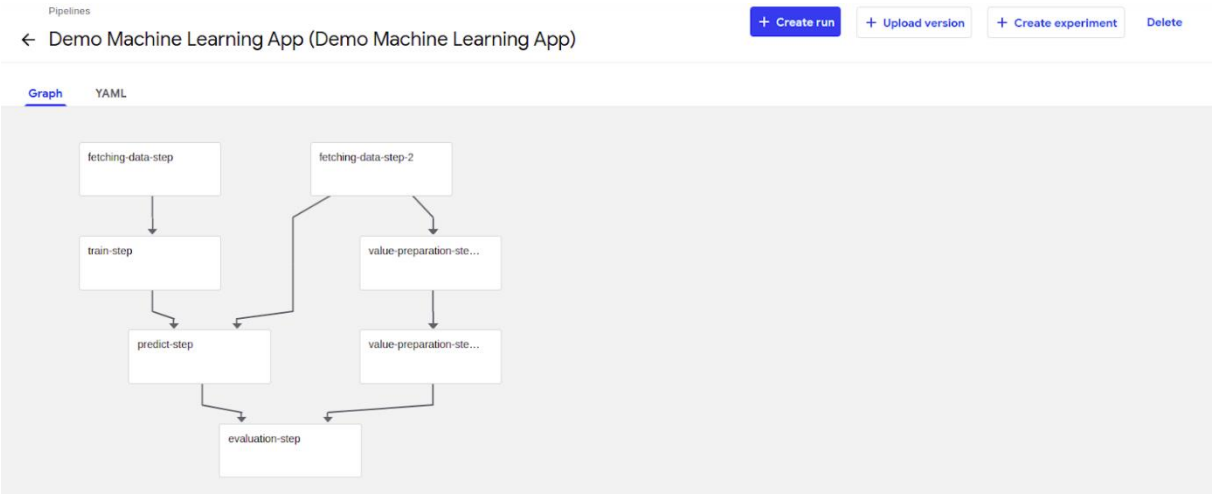

Figure 18: Machine Learning Application in the form of a graph

3.4: Stage 3: Scheduling Algorithm

3.4.1: Introduction

Kubernetes is an ideal tool for running cloud applications. But how does it decide where each application will be executed ? kubernetes implements its own scheduling module. This module is called **Kubernetes-Scheduler** and it is responsible for assigning pods(applications) to suitable nodes. This chapter analyzes the kubernetes algorithm behind kube-scheduler and how it was extended , with the addition of the algorithm.

3.4.2: Kubernetes Scheduling Algorithm

Kube-scheduler is running as an independent component usually inside the master node. For every newly created pod or other unscheduled pods , kube-scheduler selects an optimal node for them to run. However, every pod and every container, inside each pod, has different resource requirements. So, the kube-scheduler must do some filtering and selection to find the optimal nodes.

Kube-scheduler continuously monitors a waiting queue that contains all the pods that need allocation. After that it follows a 2-step operation to select a node for each pod:

1. **Filtering:** The first step is called the filtering. In this step, the kube-scheduler verifies which nodes can run this pod and discards the other ones. For this purpose, it uses some properties called predicates. An example of a predicate is the **PodFitsResources**. This filter checks whether a candidate Node has enough available resources to meet a Pod's specific resource request. After this step, the node list contains any suitable Nodes, often, there will be more than one. There is always a possibility that a pod deployment might not be scheduled. In that case, kube-scheduler triggers an event that explains the reason for failed scheduling and the pod remains unscheduled.

2. **Scoring**: if the list of suitable nodes contains more than one node, then the kube-scheduler forwards at the node priority calculation (Scoring). At this step, kube-scheduler takes the list of nodes from the filter step, and scores them based on some properties called priorities. An example of a priority is the **NodeAffinityPriority**. The node is scored according to node-affinity rules .For example, a node with a specific label is scored higher than others. The node with the highest scoring is chosen to run the specific pod. With this scoring step the algorithm selects the most suitable node for our pod.

Finally, kubernetes scheduler assigns the pod to the highest-ranking node. If there is more than one node with equal scores, then it selects one node at random. After that **Kubelet** is responsible for starting the pod inside the node.

Another great feature of Kubernetes, that helps Kube-scheduler make better decisions in terms of scheduling, is resource **limits** and **requests**. For example, machine learning engineers can specify resource requests and limits on the machine learning pod configuration file The resource limit is the maximum number of resources(CPU & Memory)that can be allocated for the containers inside the pod. The resource request is the minimum number of resources that a node should have to host this specific pod. This feature is a double-edged sword, because if a developer sets up incorrect resource rules, he/she may allocate resources that the pod does not need. It is a great feature, but it should be used carefully.

3.4.3: Kubernetes-Scheduler-Extension

Although kubernetes-scheduler provides a range of features for scheduling pods inside the cluster, the metrics applied in the decision-making process are limited. The kube-scheduling services are using only CPU and RAM usage rates to decide. Kube-scheduler does not concern about other parameters like Latency of the application. However, it is crucial for latency-sensitive machine learning applications to be deployed on nodes that can provide low response times in predictions. If this does not happen, then the specific application can become unstable, and predictions may arrive too late. So, in this thesis we propose an **extension** of **Kubernetes scheduler**, that makes scheduling decisions, not only based on CPU-RAM, but according to **Latency sensitivity** too. We named it **LAS (Latency - Awareness - Scheduler)**. There are three proposed ways of extending Kubernetes Scheduler [16]:

1. Adding new scheduler policies(predicates/priorities) to the Kube-scheduler and recompiling it.

2. Implementing a different scheduling process, that runs instead of, or alongside of Kube-scheduler.
3. Implementing a “scheduler-extender” , that the kubernetes scheduler calls as a final step before making the decision.

The third approach is used when the scheduler needs to make decisions based on resources that are not managed directly from Kubernetes Scheduler. Our scheduler is based on the third approach, because it needs to make decisions based on network parameters like latency. But how exactly does it work ? When the Kube scheduler is trying to schedule a pod, the Kubernetes - extender allows an external process to filter and/or prioritize the nodes. Below we display the structure of the configuration file, to communicate with the extender through code. This extender config contains parameters that will specify the behavior of the scheduling.

```
type ExtenderConfig struct {
    URLPrefix string `json:"urlPrefix"`
    FilterVerb  string `json:"filterVerb,omitempty"`
    PrioritizeVerb string `json:"prioritizeVerb,omitempty"`
    BindVerb string `json:"bindVerb,omitempty"`
    Weight int `json:"weight,omitempty"`
    EnableHttps bool `json:"enableHttps,omitempty"`
    TLSConfig *client.TLSClientConfig `json:"tlsConfig,omitempty"`
    HTTPTimeout time.Duration `json:"httpTimeout,omitempty"`
}
```

Figure 19: Extender policy file in Golang

From top to bottom, we have the *URLPrefix*. This is the most important parameter, which specifies the endpoint at which the extender will be available. After that we have the three verbs. The *FilterVerb*, the *PrioritizeVerb* and the *BindVerb.* Each of these verbs, specifies which function the extender will call, to do the filtering, or the prioritizing or the binding. If any of these verbs are unspecified or empty, then it is assumed that

the extender chose not to provide that extension. The arguments that are passed on the FilterVerb on the extender are the set of nodes filtered through the Kubernetes Scheduler predicates and the given pod while the arguments passed on the PrioritizeVerb also contain the priorities of each node. Following the verbs, we have the *weight* parameter. This integer is a numeric multiplier for the node scores that the prioritize call creates. Next, we have two security parameters. The EnableHttps, which specifies whether https will be used for the communication with the extender and the TLSConfig which specifies the security configuration at the Transport Layer. Finally, we have the *HTTPTimeout*. As the name implies, it is the timeout duration for a call to the extender.

This was a brief description of the kubernetes scheduler extension. Next, we are going to describe the logic and the usage of this extender to achieve the latency awareness in the scheduling part.

3.4.4: Latency Awareness Scheduler

The Latency Aware Scheduler has been implemented by extending the Default scheduler of Kubernetes through priority endpoint. This algorithm uses the help of Kubernetes Labels and Latency values to make a scheduling decision. Labels are just key/value pairs that are attached to objects, like pods, and help identify object attributes that are important to users. So, to identify the attributes that are important for our scheduling algorithm, we declare two types of labels for the pod configuration file , and one type of label for the node, for each location. The logic for this implementation was inspired by the paper Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing applications [17].

Pod

The first type of label is named ***typeOfComponent***. As the name implies, the algorithm uses this label, to learn what type of machine learning operation we are trying to

schedule. The second type of Label is called ***OurLocation***, and indicates the location we are in. The LAS will use this Label to deploy that process as close as possible to this location, with the help of Latency values.

Node

As we already mentioned we want to know the latency values of each node, for each location. So, at each individual Node we add a Label for each location we want to include into our algorithm with the respective Latency value . For example, let us say we have Location A and Location B. First, we calculate the Latency from the Node to each location. Let us say the latency for Location A is 0.8 ms and for Location B is 0.6 ms. We add a label ***{Location : LatencyValue}*** for each location. So, for our example, we have two Labels:

- {LocationA:0.8ms}
- {LocationB:0.6ms}

Algorithm

As we already mentioned, each Machine Learning application has the following steps: **Gathering Data**, **Preparing Data**, **Choosing a Model**, **Training**, **Evaluation, Hyper-parameter Tuning**, **Prediction.** So, the algorithm first fetches the typeOfComponent from the pod information. If it is not the prediction step, then we let the Default Scheduler decide the optimal node for scheduling, based on its predicates and priorities. If it is the Prediction Step, then we iterate through each node and find the minimum Latency based on the Location that we have configured in the pod configuration file at the OurLocation Label and return the node with the minimum Latency. Below we display the pseudocode of the algorithm in Golang.

```
//Algorithm(Golang-pseudocode)

selectNode(Pod, listofNodes) (selectedNode, error){

    typeOfComponent, err = getTypeOfcomponent(Pod)
    if err != nil {
        return error
    }
```

```
    if typeOfComponent == "Training" { // Scheduled By  Default Kubernetes Scheduler
            return nil,error.Error()
    }else if typeOfComponent == "Predicting" {


        OurLocation,err = getOurLocation(Pod)
        if err != nil {
            return nil,error // Scheduled By  Default Kubernetes Scheduler
        }

        latencyMap := make(map[float64]Node)
        minLatency := math.Inf(+1)

        for node := range listofNodes {
            currentLatency := GetLatency(node,OurLocation)

            if currentLatency < min {
                minLatency = currentLatency
            }
            latency[currentLatency] = node
        }


        priorityList = make([]schedulerapi.HostPriority, 1)

        priorityList[0] = schedulerapi.HostPriority{
            Host:  latencyMap[min].name,
            Score: 1,
        }

        return &priorityList, nil
    }else {
        return nil,error // Scheduled By  Default Kubernetes Scheduler
    }
}
```

Figure 20: Scheduler Algorithm Pseudocode in Golang

CHAPTER 4: EXPERIMENTS & RESULTS

4.1: Introduction

In this chapter, some experiments are presented to observe the behavior of the algorithm in a real machine learning application. In addition, it is found, through diagrams and experiments, that scheduling with our algorithm , leads to shorter times, in terms of latency and training time. Finally, it is important to emphasize that the experiments were performed under different conditions, regarding the computational requirements of the application at the training step and the size of each request at the prediction step.

4.2: Infrastructure

As for the infrastructure on which the experiments were performed, it consists of 5 different machines. Three of the five computers are NITOS nodes, which we talked about in the 3.2.1: NITOS Testbed section and the other two computers are virtual computers set up on physical machines. The above computers consist of the following Hardware features:

Table 2: Hardware specifications

Computers	CPU	RAM	HDD
3 NITOS Nodes	8 cores	16GB	120GB
2 VMs	2 cores	4GB	40GB

One of the 3 NITOS Nodes, operates as **Master Node**, and the other 4 as **Worker Nodes** for the Cluster. These Nodes, use as operating system the **Linux Ubuntu 18.04.04 LTS**. The necessary tools were installed in them, to create a network

between them so they can communicate. These tools are the **Kubernetes**, **Docker,** and **Calico Network**. Also, the **Kubeflow** was deployed on top of the cluster to break the machine learning Application into separate processes. Finally, the proposed scheduler (LAS) was deployed as a default scheduler, in order to perform the experiments. The pod configuration file for the LAS is shown at the Figure 22 below. As can be seen, the pod is composed of two containers: the extender and the scheduler. The extender is responsible for performing our proposed scheduling operation and the scheduler is the Default Scheduler of Kubernetes. As we already mentioned the extender needs the policy configuration file, in order to operate smoothly. In Figure 21 below, the policy file for the LAS is shown. Finally, it is important to emphasize that the same machine learning application was used for all the experiments. This machine learning app was described in detail in chapter 3.3: Stage 2: Machine Learning Application.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-scheduler-policy
  namespace: kube-system
data:
  policy.cfg: |
    {
      "kind" : "Policy",
      "apiVersion" : "v1",
      "priorities" : [
        {"name" : "LeastRequestedPriority", "weight" : 1},
        {"name" : "BalancedResourceAllocation", "weight" : 1},
        {"name" : "ServiceSpreadingPriority", "weight" : 1},
        {"name" : "EqualPriority", "weight" : 1}
      ],
      "extenders" : [{
        "urlPrefix": "http://localhost:8800/scheduler",
        "prioritizeVerb": "priorities/nikos_priority",
        "enableHttps": false,
        "weight": 10
      }],
    }
```

Figure 21: Policy configuration file

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
    - name: my-scheduler-extender-ctr
      image: nickange/test-extender:7.0
    - command:
        - kube-scheduler
        - --bind-address=127.0.0.1
        - --v=5
        - --kubeconfig=/etc/kubernetes/scheduler.conf
        - --leader-elect=true
        - --policy-configmap=my-scheduler-policy
      image: nickange/test-scheduler:6.0
      imagePullPolicy: IfNotPresent
      livenessProbe:
        failureThreshold: 8
        httpGet:
          host: 127.0.0.1
          path: /healthz
          port: 10251
          scheme: HTTP
        initialDelaySeconds: 15
        timeoutSeconds: 15
      name: kube-scheduler
      resources:
        requests:
          cpu: 100m
      volumeMounts:
        - mountPath: /etc/kubernetes/scheduler.conf
          name: kubeconfig
          readOnly: true
  hostNetwork: true
  priorityClassName: system-cluster-critical
  volumes:
    - hostPath:
        path: /etc/kubernetes/scheduler.conf
        type: FileOrCreate
      name: kubeconfig
```

Figure 22: LAS scheduler Pod (Container: Extender & Container: Default-Scheduler)

4.3: Experiments

4.3.1: Algorithm Verification

The first experiment was responsible for the verification of the algorithm. We wanted to create an environment, in which our algorithm could take scheduling decisions based on the latency values and the resource specifications of each node. For this purpose, we located the nodes at three different locations (**NITOS-CLOUD**, **NITLAB-EDGE**, **HOME-EDGE**). The nitos-cloud is the cloud infrastructure of nitos, the nitlab-edge is located at NITLab building, and the last one is at my home. The three heavy resourced nodes at the cloud location and the other two at the edge's locations, respectively.

As we already mentioned, the scheduler takes scheduling decisions based on the latency labels that each node has. So, the next step was to assign latency values at each node . For this purpose , the **ping** tool was used for finding an average RTT time. These values were assigned to each node as a label so that delay values can be considered in the scheduling process. Times are displayed in milli-seconds.

Table 3: Node Latency Values for each Location

NODE	NITOS-CLOUD	NITLAB-EDGE	HOME-EDGE
Nitos-Cloud-Node	0.289	2.669	89.667
NITLab-Edge-Node	41.966	42.551	9.8
Home-Edge-Node	2.669	0.306	42.551

Once the labels have been placed at the corresponding nodes of the cluster, it is time to test the algorithm for different cases. According to the algorithm we can specify at which location we want to deploy the application, with the help of OurLocation label, inside the pod configuration file. However, the machine learning application is deployed as Kubeflow pipeline Component. So, we cannot access the pod Configuration file

directly. That is why the kfp package was used to specify the labels for each process of the machine learning application. The number of cases that we tested the algorithm is equal with the number of locations. Below we display how we must configure the predict and the training step of our machine learning application, for each location.

Table 4: Configuration changes at Machine Learning Pipeline-Location:NITOS-CLOUD

Training	• training_proc.add_pod_label("typeOfComponent","Training")
Predict	• predict_proc.add_pod_label("typeOfComponent","Predict") • predict_proc.add_pod_label("OurLocation","NITOS-CLOUD")

Table 5: Configuration changes at Machine Learning Pipeline-Location:NITLAB-EDGE

Training	• training_proc.add_pod_label("typeOfComponent","Training")
Predict	• predict_proc.add_pod_label("typeOfComponent","Predict") • predict_proc.add_pod_label("OurLocation", "NITLAB-EDGE")

Table 6: Configuration changes at Machine Learning Pipeline-Location:HOME-EDGE

Training	• training_proc.add_pod_label("typeOfComponent","Training")
Predict	• predict_proc.add_pod_label("typeOfComponent","Predict") • predict_proc.add_pod_label("OurLocation","HOME-EDGE")

The LAS algorithm has successfully scheduled both steps at the correct nodes, as we can see from the kubectl get command. The heavy Training process at a cloud node, which has more available resources(CPU & RAM), and the Predict Step at an edge node, which is much closer to the location that the client requested at each case. The kubectl get command returns a lot of information, but we display only the important ones. It is important to say that inside the cluster, the names of each step of the machine learning application have the format ***demo-machine-learning-app-***

($***uniqueId***), because this is the name of the pipeline to which they belong. The uniqueId parameter represents each step of the pipeline. However, in each command below we refer to each step with the real name, for easy understanding.

```
$ kubectl -n kubeflow get pods
NAMESPACE     NAME           NODE
kubeflow      Predict        nitos-cloud-node
kubeflow      Training Step  nitos-cloud-node
```

```
$ kubectl -n kubeflow get pods
NAMESPACE     NAME           NODE
kubeflow      Predict        nitlab-edge-node
kubeflow      Training Step  nitos-cloud-node
```

```
$ kubectl -n kubeflow get pods
NAMESPACE     NAME           NODE
kubeflow      Predict        home-edge-node
kubeflow      Training Step  nitos-cloud-node
```

4.3.2: Measurements

Once we have verified that our algorithm is working properly and distributes the individual processes according to the requirements in latency and resources, it is time to take measurements. These measurements will provide information about the behavior of the machine learning Application, regarding the execution time of the training process and the response time of the prediction process. To achieve this , we divided the measurements into two experiments. At the first one , we deployed the Machine Learning application as an individual process, with all its steps inside a container. Experiment was done first at an **edge node** and then at a **cloud node** of our environment. After that we calculated the completion time of the training process for light training and heavy training at each node, respectively. Also, we calculated

the latency response of the predict step for different sizes of requests. At the second experiment, we calculated the same things but now having broken the application into separate processes and using the LAS algorithm.

To achieve different conditions on the training process we modified the parameters learning rate and num_iterations. For this experiment the below values were used:

Table 7: Training Parameters

Training	Learning Rate	Number of Iterations	RMSE
Light Training	0.01	10000	0.07478
Heavy Training	0.001	100000	0.02578

As we can see from the RMSE column above, the Heavy Training achieves a smaller **Root Mean Square Error** compared with the Light Training Process. Smaller RMSE means smaller Prediction errors , which is better for the trained model.

Below we display the execution time in seconds, for each case and the response times after the deployment of the predict step :

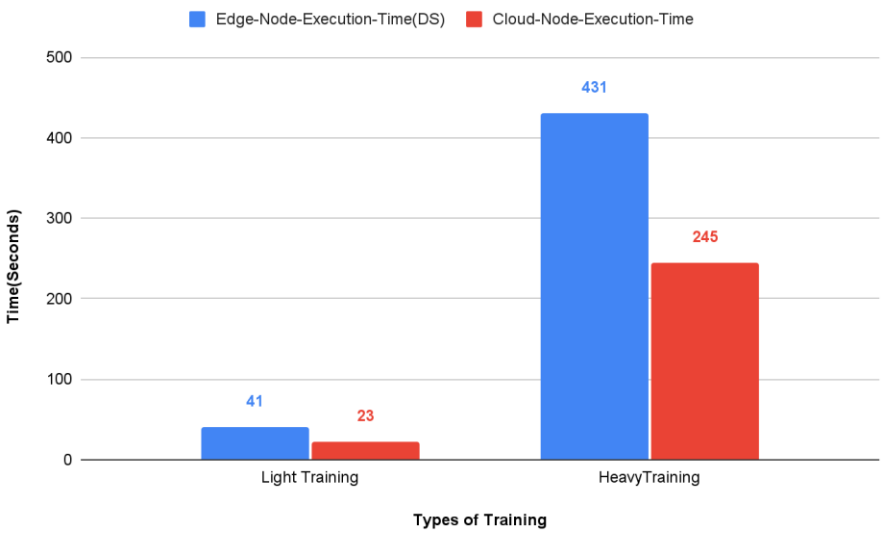

Chart 1: Training process (Default Scheduler) - Execution times in Seconds

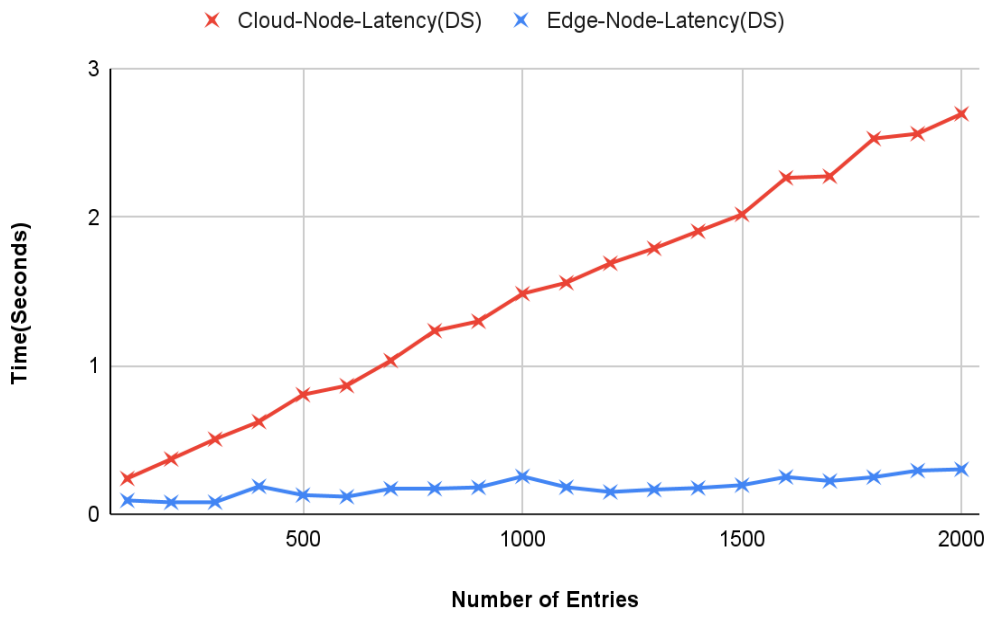

Chart 2: Prediction process (Default Scheduler) - Latency values in seconds

From the charts above, we observe that in a cloud node where the resources are much more, we achieve shorter execution times, than the edge node where the resources are much less. The same thing is observed even at the light training case which requires less computational resources. From the chart 1, we can see that the execution time is almost half at the cloud node. However, the exact opposite happens in the case of response time. There the edge node dominates, because it is much closer to the end user who makes the various prediction requests. From the chart above, as we increase the number of requested predictions, the response time of the edge node is staying almost constant and close to half a second, while at the cloud node is increasing and at 2.000 requested predictions it reaches almost 3 seconds.

From the above, it is understood that we cannot achieve at the same time short execution and response time in a machine learning application, if we deploy it as a single entity. That is why we broke the application into its steps with the help of Kubeflow and after that we deployed it in the cluster with the help of our algorithm (LAS).

At the second experiment, the machine learning application is deployed with the supervision of our algorithm. This time , the heavy training process will be scheduled at a cloud node, to achieve fast execution time, while the predict process will be scheduled at the edge node based on the latency values as we saw at the Verification Experiment. After that , the same measurements took place and showed us that both time parameters are shorter for better utilization of resources and for better user experience. Below we display another two charts for the execution and the response time of the heavy training process. Here it is important to emphasize that for chart 4 we took measurements with even higher values of prediction requests, to show that even for these values the time remains short and increases slightly for the edge node. Also, we display the response time for the cloud , to realize the big difference.

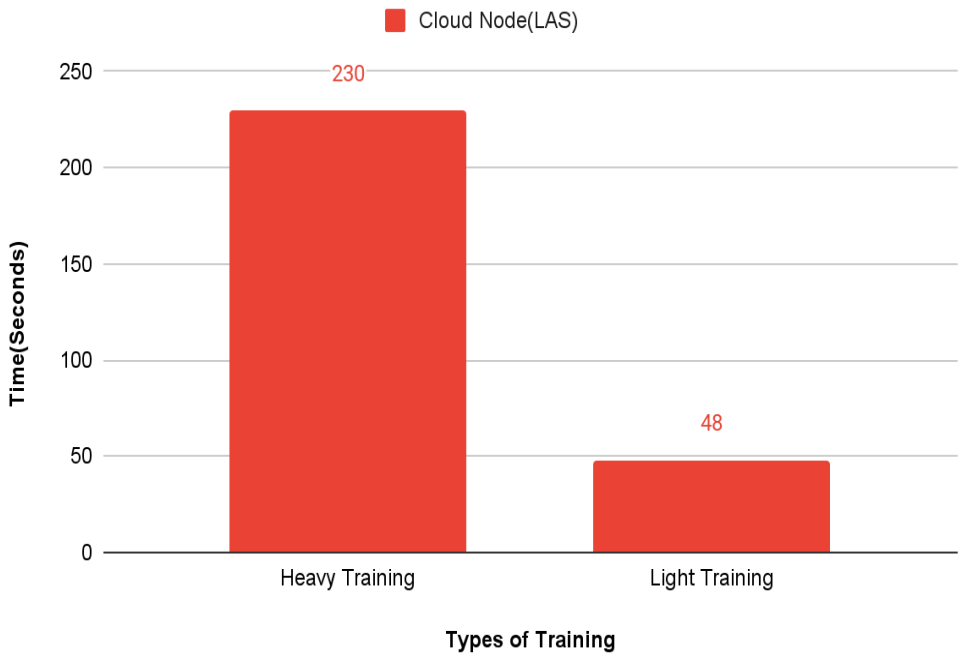

Chart 3: Training process (LAS) - Execution times in seconds

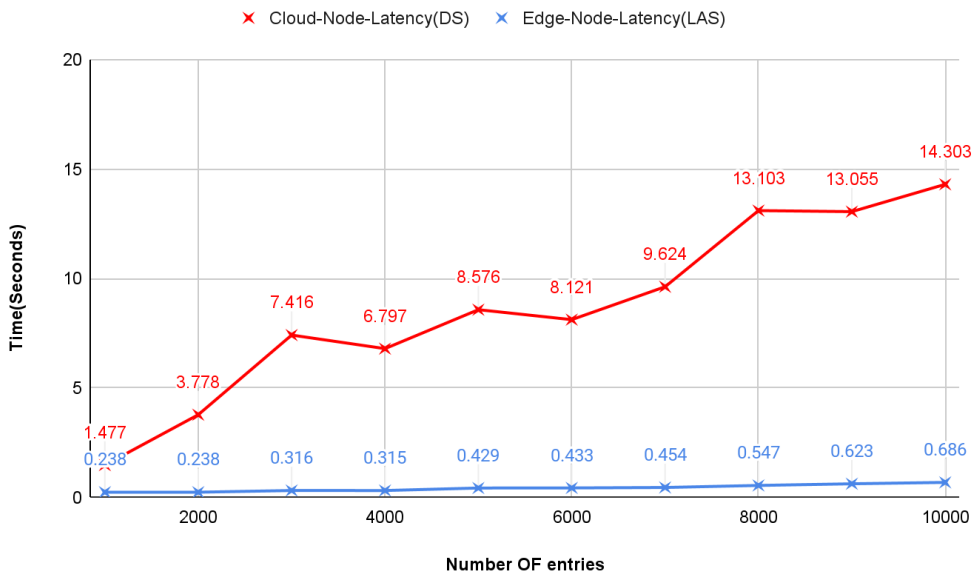

Chart 4: Training process (LAS) - Execution times in seconds

CHAPTER 5: CONCLUSION & FUTURE WORK

In this final Chapter, we present the conclusion of our work. Following that, we conclude by mentioning a few possible steps that should be done in the future, to convert this algorithm into a more global scheduling choice.

5.1: Conclusion

All in all, the primary goal of this thesis was to implement an algorithm that would deploy the various steps of a machine learning application in the appropriate locations, depending on their needs (computational, latency). This would enable machine learning developers to deploy their applications in a more performant-optimal way. The goal was achieved as we can see from the experiments above. The algorithm was implemented and tested, and the measurements showed how important it is to deploy a latency sensitive application to the edge.

5.2: Future Work

In this Thesis a big step was taken in terms of creation and evaluation of the scheduling algorithm. However, there are still some steps that need to be taken in the implementation and the evaluation stage before it becomes a global way of scheduling machine learning applications. Below we describe these future steps.

First, we need to make a more extensive evaluation of the algorithm. The number of nodes and locations must be increased to evaluate the behavior of the algorithm in a more “competitive” environment where more computers will compete for the scheduled process . Also in future experiments, the number of applications should be increased to study the scheduling decision of the algorithm in a more realistic environment where

there will not be only a single machine learning application that is trying to be scheduled.

From implementation perspective some changes will be done in the latency assignment part of the nodes. So far, we place the latency values on each node for each location, with the help of the ping tool, hardcoded. This latency assignment logic cannot be followed in a real environment where the locations and nodes will be countless. A future goal is to assign these values with predictive techniques using machine learning that is going to predict the latency for each location.

Bibliography

[1] Microsoft Azure, "What is cloud computing?," [Online].
Available: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/.
[Accessed 13 May 2021].

[2] Red Hat, "What is edge computing?," [Online].
Available: https://www.redhat.com/en/topics/edge-computing/what-is-edge-computing.
[Accessed 14 May 2021].

[3] L. Baresi, D. F. Mendonça and . M. Garriga, "Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture.," in *6th European Conference on Service-Oriented and Cloud Computing (ESOCC)*, Oslo, Norway, September 2017.

[4] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, "Edge Computing: Vision and Challenges," in *IEEE Internet of Things Journal*, 2016.

[5] docker, "What is a Container?," 2013. [Online].
Available: https://www.docker.com/resources/what-container.
[Accessed 6 May 2021].

[6] D. M. "Silicon UK," 18 February 2016. [Online].
Available: https://www.silicon.co.uk/software/open-source/linux-kernel-cgroups-namespaces-containers-186240.
[Accessed 16 April 2021].

[7] Wikipedia, "UnionFS," 10 January 2021. [Online].
Available: https://en.wikipedia.org/wiki/UnionFS.
[Accessed 18 April 2021].

[8] Docker, "Docker," 2013. [Online].
Available: https://www.docker.com/.
[Accessed 16 February 2021].

[9] Kubernetes, "Production-Grade Container Orchestration," 2021. [Online].
Available: https://kubernetes.io/.
[Accessed 15 March 2021].

[10] J. C. Martinez, "7 Steps of Machine Learning," 2 January 2020. [Online].
Available: https://livecodestream.dev/post/7-steps-of-machine-learning/.
[Accessed 14 April 2021].

[11] Kubeflow, "Kubeflow," 2018-2020. [Online].
Available: https://v1-0-branch.kubeflow.org/.
[Accessed 30 January 2021].

[12] Prometheus, "From metrics to insight," 2014. [Online].
Available: https://prometheus.io/.
[Accessed February 5 2021].

[13] Grafana, "Your observability wherever you need it," [Online].
Available: https://grafana.com/.
[Accessed 29 March 2021].

[14] NITLAB, "Network Implementation Testbed Laboratory," 2015. [Online].
Available: https://nitlab.inf.uth.gr/NITlab/.
[Accessed 15 November 2020].

[15] Kubernetes, "Overview of kubeadm.," 2018. [Online]. Available: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/. [Accessed 29 March 2021].

[16] Github - Kubernetes, "Scheduler Extender," 2019. [Online]. Available:https://github.com/kubernetes/community/blob/master/contributors/design-proposals/scheduling/scheduler_extender.md. [Accessed 10 May 2021].

[17] J. S. T. W. B. V. and F. D. T. , "Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing applications," in *IEEE Conference on Network Softwarization (NetSoft)*, Ghent, Belgium, 2019.