ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΧΕΔΙΑΣΜΟΣ ΚΑΙ ΑΝΑΠΤΥΞΗ ΜΗΧΑΝΙΣΜΩΝ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ ΚΑΙ ΠΡΟΒΛΕΨΗΣ ΚΑΤΑΝΑΛΩΣΗΣ ΠΟΡΩΝ ΜΕΣΩ ΑΛΓΟΡΙΘΜΩΝ ΜΗΧΑΝΙΚΗΣ ΜΑΘΗΣΗΣ ΕΙΚΟΝΙΚΩΝ ΣΥΝΑΡΤΗΣΕΩΝ ΔΙΚΤΥΩΝ ΚΑΙ ΔΥΝΑΜΙΚΗ ΠΡΟΣΑΡΜΟΓΗ ΤΟΥΣ

Διπλωματική Εργασία

Βασίλειος Ζαλοκώστας-Δίπλας

Επιβλέπων: Αθανάσιος Κοράκης

Βόλος 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

DESIGN AND IMPLEMENTATION OF MECHANISMS FOR RESOURCE MONITORING, RESOURCE PREDICTION THROUGH MACHINE LEARNING AND DYNAMIC RE-CONFIGURATION FOR VIRTUAL NETWORK FUNCTIONS

Diploma Thesis

Vasileios Zalokostas-Diplas

Supervisor: Athanasios Korakis

Volos 2021

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπων	**Αθανάσιος Κοράκης** Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Δημήτριος Μπαργιώτας** Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας
Μέλος	**Αντώνιος Αργυρίου** Αναπληρωτής Καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Ημερομηνία έγκρισης: 22-09-2021

ΕΥΧΑΡΙΣΤΙΕΣ ή ΣΧΟΛΙΑ

Θα ήθελα να ευχαριστήσω τον κ. Αθανάσιο Κοράκη ο οποίος με επέλεξε για να αναλάβω αυτό το θέμα της διπλωματικής εργασίας. Επίσης, θα ήθελα να ευχαριστήσω τους Νικόλαος Μακρής και Αλέξανδρος Βαλαντάσης που με τις υποδείξεις και τις ιδέες τους βοήθησαν στην ολοκλήρωση της εργασίας αυτής.

Επιπρόσθετα, ευχαριστώ την οικογενειά μου που με βοήθησε σε όλα αυτά τα χρόνια των σπουδών μου τόσο συναισθηματικά όσο και οικονομικά. Τέλος, ευχαριστώ την Άννα Δούβλη που με βοήθησε να ολοκληρώσω την συγγραφή της διπλωματικής μου εργασίας αλλά και για την συμπαράσταση και την στήριξη που μου έδειξε στα φοιτητικά μου αυτά χρόνια.

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

Ο Δηλών

Βασίλειος Ζαλοκώστας-Δίπλας
Ημερομηνία
09/09/21

ΠΕΡΙΛΗΨΗ

Στην σημερινή εποχή, οι προγραμματιστές καλούνται να δημιουργήσουν εφαρμογές οι οποίες τρέχουν σε διάφορα λειτουργικά συστήματα, συμπεριλαμβανομένων τοπικών διακομιστών, εικονοποιημένα ιδιωτικά και δημόσια cloud. Με το Kubernetes, το οποίο είναι μια πλατφόρμα ανοιχτού κώδικα για ανάπτυξη, κλιμάκωση και διαχείρηση εφαρμογών σε container, καθίσταται δυνατός και αρκετά ευέλικτος ο χειρισμός του έργου αυτού σε ενα σύμπλεγμα υπολογιστών διαχειρίζοντας παράλληλα το φόρτο εργασίας για να διασφαλιστεί οτι όλα λειτουργούν έτσι όπως θα ήθελε ο χρήστης. Επιτρέπει υψηλή διαθεσιμότητα και επεκτασιμότητα μέσω διάφορων μηχανισμών αυτόματης κλιμάκωσης τα οποία είναι Horizontal Pod Autoscaler(HPA), το Vertical Pod Autoscaler(VPA) και το Cluster Autoscaler(CA). Μεταξύ αυτών, ο πρώτος παρέχει απρόσκοπτη υπηρεσία με δυναμική κλιμάκωση προς τα πάνω και κάτω του αριθμού των μονάδων πόρων, που ονομάζονται Pods, χωρίς να χρειάζεται επανεκκίνηση ολόκληρου του συστήματος.

Σκοπός της εργασίας αυτής είναι η υλοποίηση ενός συστήματος με στόχο την μεταβολή αυτή του αριθμού των Pods σε πραγματικό χρόνο σύμφωνα με την χρήση αλγορίθμων μηχανικής μάθησης που θα προβλέπουν σε βάθος χρόνου το traffic σε ενάν εξυπηρετητή, χωρίς την βοήθεια του Horizontal Pod Autoscaler αλλά ουσιαστικά αντικαθιστώντας τον και αξιολογώντας τις αποδόσεις του συστήματος αυτού. Με αυτόν τον τρόπο, μπορεί να γίνει αποδοτικότερη χρήση πόρων και εξοικονόμηση ενέργειας καθώς ο εξυπηρετητής είναι ενήμερος για το μελλοντικό traffic ώστε να έχει την χρονική δυνατότητα μέχρι να καταφθάσει αυτό, να λάβει τα απαραίτητα μετρα.

Το σύστημα δοκιμάστηκε σε πραγματικό χρόνο στο NITOS Testbed Laboratory αποδεικνύοντας ότι μπορεί να ωφελήσει η συγχώνευση της μηχανικής μάθησης με μηχανισμούς κλιμάκωσης στο περιβάλλον των Kubernetes όπως ο HPA καθώς σε αρκετές περιπτώσεις υπάρχουν επαναλαμβανόμενα μοτίβα στο traffic που μπορούν να προβλεφθούν με μεγάλη ακρίβεια. Οι πόροι δεσμεύονται ή αποδεσμεύονται ανάλογα με το μελλοντικό traffic έτσι ώστε να είναι προετοιμασμένοι να ανταποκριθούν γρηγορότερα και αποδικότερα ενώ ταυτόχρονα εξοικονομείται ενέργεια καθώς δεν υπάρχουν πόροι που καλούνται να εξυπηρετήσουν εργασίες παραπάνω από αυτές που μπορούν ή πόροι που δεν χρησιμοποιούνται.

ABSTRACT

These days, developers are called on to write applications that run across multiple operating environments, including local servers, virtualized private and public clouds. Kubernetes, which is an open-source platform for developing, scaling and managing applications in a container, makes it possible and flexible enough to handle this task on a complex of computers while managing the workload to ensure everything works as the user would like to. It allows high availability and scalability through various automatic scaling mechanisms which are Horizontal Pod Autoscaler, Vertical Pod Autoscaler and Cluster Autoscaler. Among these, the first one provides seamless service by dynamically scaling up and down the number of resource units, called Pods, without having to restart the entire system.

This diploma thesis focuses on the implementation of a system that aims to change the number of Pods according to machine learning algorithms that will predict the upcoming traffic on a server, without the help of Horizontal Pod Autoscaler but essentially replacing it and evaluating the performance of this system. In this way, more efficient use of resources and energy savings can be made as the server is aware of future traffic so that it has the time to take the necessary measures until it arrives.

The system was tested in real time at the NITOS Testbed Laboratory proving that combining machine learning with scaling mechanisms in the Kubernetes environment such as HPA can be beneficial, as in many cases there are repetitive traffic patterns that can be predicted with good accuracy. Resources are being allocated or released depending on future traffic, so that they are prepared to respond faster and more efficiently while at the same time energy is being saved as there no resources called to serve tasks beyond what they can or resources that are not used.

Table of Contents

List of Figures

List of Charts

Chapter 1: Introduction

1.1 Background

The fact that technology has a great impact on our life and business is evident. Not so long ago, companies had to establish and maintain their own server environment so that they could host and run applications on their premises. Today, we have cloud computing that's revolutionizing everything. Cloud computing [1] is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user and relies on sharing of resources to achieve coherence and economies of scale. It offers a wide variety of benefits both in businesses with the efficiency, the cost-effectiveness, and the scalability that it provides, and in the average person with storing and accessing multimedia content via internet or even running software programs without installing them on their local PC. Cloud computing is gaining more and more popularity making it one of the most flamboyant technological innovation of the 21st century.

Another field of science that has seen a significant growth lately is Machine Learning [2]. Machine Learning is the study of computer algorithms that improve automatically through experience and using data. It is seen as a part of artificial intelligence and its algorithms build a model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to do so. Machine Learning algorithms are used in a wide variety of applications, such as in medicine, email filtering, speech recognition and computer vision and generally where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks by identifying the common patterns and making predictions.

Each one of these two topics have a positively influence to the world today. However, this is the beginning, and it will take some time to successfully merge these 2 technologies in order to become fully operational in important areas such as healthcare, business and banking. Machine Learning makes it easy to manipulate data in cloud. With a series of artificial intelligence research on cloud computing, cloud computing is becoming more and more intelligent. Less energy is being consumed in addition to the performance that increases at a rapid pace while the resources are being scaled up and down dynamically.

1.2 Motivation

Nowadays, cloud-computing is becoming more and more popular. Kubernetes which is used to manage container workloads in scalable infrastructure, has become one of the most famous cloud orchestrators. It is an open-source platform that enables customers to respond to user requests quickly and deploy software updates faster and with greater resilience.

Imagine a scenario where an application that had been deployed has more traffic than it is anticipated and as a result the compute resources overwork to get the job done. This is a problem that can be solved by scaling the infrastructure. For example, if the application has more traffic during the day and less during night, it doesn’t make sense to keep the same number of compute resources allocated during the off-peak hours. By using autoscaling, you can easily and dynamically provision more compute power when it is necessary. So, in many cases, like the one described above, where there are common patterns in the traffic that is being send, merging scaling mechanisms with machine learning algorithms that can identify these trends would be ideal for making the appropriate scale ups and downs.

So, that was the motivation of this diploma thesis. To **create a mechanism** utilizing the tools and finding the methodology to **a more efficient scaling** using **machine learning time series algorithms** in order to forecast the future traffic to a server, in a Kubernetes cluster. That way, the number of pods would be adapted to the future values making the scaling a lot more efficient. As a result energy is saved as there are no resources that are idle or overwork.

1.3 Thesis Structure

This thesis is divided in 5 chapters which are being descripted below:

- **Chapter 1**: An introduction to cloud computing and machine learning making a brief explanation on what these technologies represent and how beneficial their merging is. Also, a reference to the motivation and the problem statement of this thesis is being made.
- **Chapter 2**: The experimental tools and the technical background that needed for developing and testing this research in the Kubernetes Cluster using Machine Learning algorithms.
- **Chapter 3**: The way that the infrastructure and the tools are being used for implementing this diploma thesis for the desired result. Furthermore, the methodology of creating the system architecture, the selection of the Machine Learning algorithms and the scale policy for the pods are being analyzed.
- **Chapter 4**: Results and charts are being presented evaluating the efficiency of each Machine Learning algorithm comparing to each other and to the Horizontal Pod Autoscaler.
- **Chapter 5**: Conclusion based on the results from Chapter 4 and the work that can be done for optimizing this diploma thesis even more.

Chapter 2: Infrastructure and Experimental Tools

2.1 Chapter Introduction

This chapter focuses on the infrastructure and the experimental tools that had being used in order to successfully complete this diploma thesis. It analyzes in detail every technology that helped solving this problem as much in Kubernetes environment as in Machine Learning.

2.2 NITOS Testbed

This diploma thesis was developed and implemented in 2 nodes of the NITOS Testbed Laboratory [3]. NITOS is an integrated facility with heterogeneous testbeds that focuses on supporting experimentation-based research in the area of wired and wireless networks. It is remotely accessible and open to the research community 24/7. It is comprised of three different deployments which are listed below:

1. **The Outdoor Testbed:** which consists of powerful nodes that feature multiple wireless interfaces and allow for experimentation with heterogeneous wireless technologies. It is deployed at the exterior of a University of Thessaly's campus building and consists of 50 nodes.
2. **The Indoor RF Isolated Testbed:** which contains 50 Icarus nodes that feature multiple wireless interfaces (Wi-Fi, WiMAX, LTE) and is deployed in an isolated environment of a University of Thessaly's campus building. It is also equipped with directional antennas prototypes.
3. **The Office Testbed:** which comprises of 10 powerful second-generation Icarus nodes. The nodes encapsulate heterogeneous technologies, such as WIFI, WiMAX, LTE etc. and allow the experimenter to design and execute real life scenarios under a deterministic office environment.

Generally, NITOS testbed offers a variety of tools which the user can use in order to develop, design and test his own experiments in a large-scale environment and its architecture is depicted in the image[1] below.

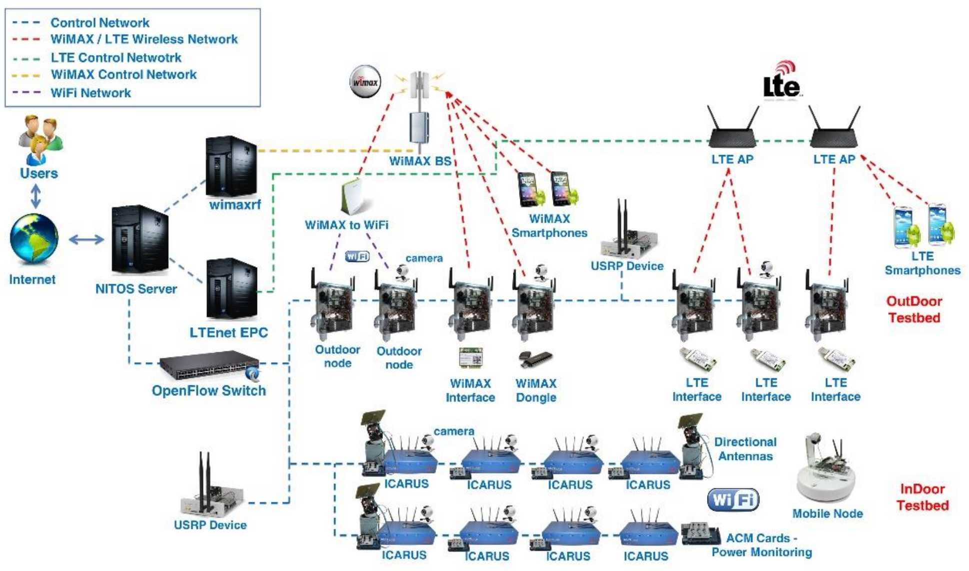


Figure 1. NITOS Testbed Architecture

2.3 Docker

2.3.1 Overview

Docker [4] is an open platform for developing, shipping and running applications. It gives developers the opportunity to better manage and deploy applications by packaging them in a sandbox (called containers) to run on the host operating system i.e., Linux by taking advantage of the OS[2]-level virtualization (an operating system paradigm in which the kernel allows the existence of multiple isolated user space instances). Docker's portability and lightweight operation also make it easy to dynamically manage workloads, scaling up or tearing down applications in real time. With the help of container's technology, the time gap between writing code and running it on production is being minimized as the deployment or the update of the app is done in a matter of minutes.

[1] https://nitlab.inf.uth.gr/NITlab/nitos
[2] OS: Operating System

2.3.2 Containers

So, what is a container [5]? A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. It uses process isolation and virtualization capabilities built into the Linux kernel such as control groups (Cgroups) for allocating resources among processes, and namespaces for restricting a processes access or visibility into other resources or areas of the system. This gives the developers the ability to create environments that are isolated from the rest of the applications and can be run anywhere.

2.3.3 Containers vs Virtual Machines

An important part that must be clarified is the difference between containers and virtual machines. At first, virtual machines are an abstraction of physical hardware turning one server into many in addition to the containers that are an abstraction of the application layer that packages code and dependencies together. So, each architecture has significant differences from the other as shown in the image[3] below. Also, the size of virtual machines images is measured in gigabytes whilst containers images in megabytes, just because the first ones include a full copy of an operating system. Finally, containers offer more portability and quicker spinning up applications in contrast with the VMs that can be slow to boot.

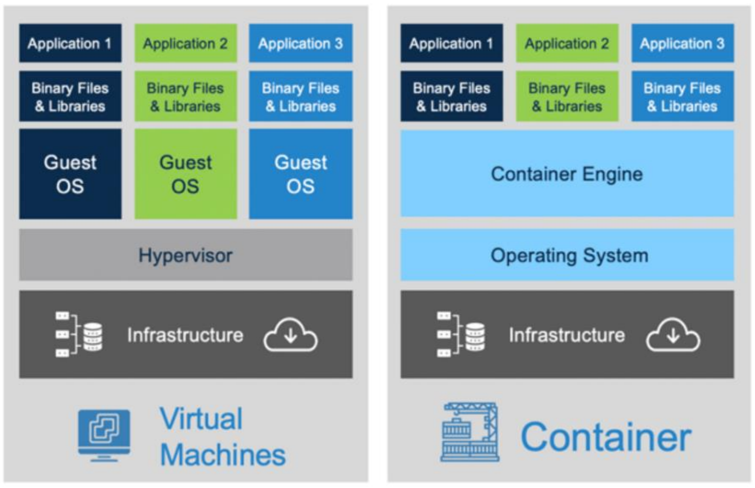


Figure 2. Virtual Machines vs Containers

[3] https://akfpartners.com/growth-blog/vms-vs-containers

2.3.4 Docker Architecture

Docker uses a client-server architecture. Users interact with Docker through the Docker client. When any commands are being executed, this client send them to Docker daemon which carries them out. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface as depicted in the picture[4] below.

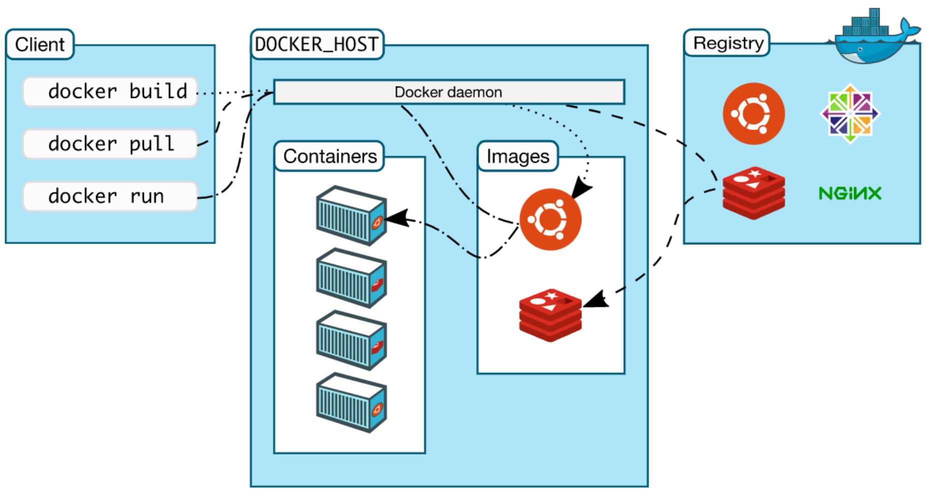


Figure 3. Docker Architecture

A few more important objects that need to be explained in order to understand the entire architecture and how Docker operates, are Docker images and Docker registry.
A Docker image is a file used to execute code in a container and acts as a set of instructions to build one, just like a template. These images make up the starting point when using Docker as they contain application code, libraries and tools that are needed to make an application run and are comparable to a snapshot in virtual machine environments. As a result, Docker images can be described as a record of a Docker container at a specific point in time.
These images can all be stored in a storage and content delivery system, available in different tagged versions, named Docker registry. Users can interact with a registry by using docker push and pull commands in order to pull images locally or push image to the registry. In this diploma thesis, DockerHub was used for storing and pulling the necessary

[4] https://docs.docker.com/get-started/overview/

docker images which is basically a hosted repository service provided by Docker for finding and sharing container images.

2.4 Kubernetes

2.4.1 Overview

Kubernetes [6] is part of the Cloud Native Computing Foundation which supports the development of shared networking standards in cloud data management software. It is an open-source system for automating deployment, scaling and management of containerized applications and groups containers that make up an application into logical units for easy management and discovery. Kubernetes provides the developers a framework where they can run distributed systems resiliently by taking care of scaling and failover for an application, providing deployment patterns and many more. Docker being the most popular container virtualization standard is widely used by Kubernetes. Simply, Docker helps to “create” containers and Kubernetes allows the developers to “manage” them at runtime.

2.4.2 Kubernetes Architecture

An environment running Kubernetes consists of the following basic components [7]: a control plane (Kubernetes Master), a storage system for keeping the cluster state consistent and cluster nodes (Kubelets, also called worker nodes).

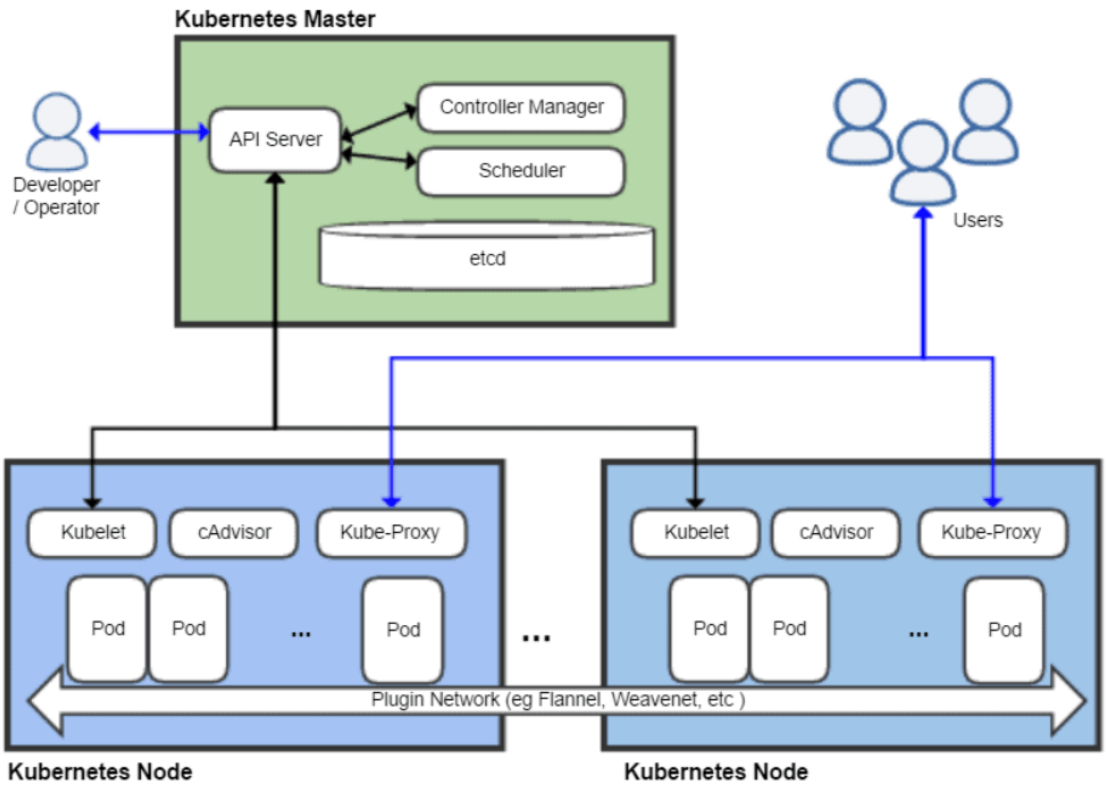


Figure 4. Kubernetes Architecture

The Kubernetes architecture can be divided into 2 basic parts: the control panel (Kubernetes Master) and the nodes (Kubernetes Nodes) as shown in the picture[5] above.

2.4.2.1 Control Panel

The control panel is the "heart" of a Kubernetes cluster. It maintains a data record of the configuration and state of all cluster's objects and is in constant contact with the compute machines in order to ensure that the cluster runs as configured. Below are presented the main components[6] of the control panel:

1. **Kubernetes API Server:** it is the front end of the Kubernetes control panel and supports updates, scaling and other kinds of lifecycle orchestration procedures by providing APIs for various types of applications. Also,

[5] https://www.jobacle.nl/?p=2688
[6] https://kubernetes.io/docs/concepts/overview/components/

Kubernetes API Server is the only component that communicates with the etcd in order to ensure that data is stored in it and is in agreement with the service details of the pods.

2. **Kubernetes Scheduler[7]:** stores the resource usage data for each compute node, check the health state of a cluster, determines whether new containers should be deployed and where they should be placed. Firstly, it checks the pod's resource demands and then selects an appropriate compute node in order to schedule the task taking resource limitations, data locality, quality of service requirements and other factors into account.
3. **Kubernetes Controller Manager:** sometimes called Cloud Controller Manager, is simply a daemon which runs the Kubernetes cluster using several controller functions. It is responsible for managing controller processes with dependencies on the underlying cloud provider such as pods, services, tokens, service accounts, nodes etc.
4. **ETCD:** is a distributed and fault-tolerant, key-value store database that stores Kubernetes cluster data like configuration files and information about the state of the cluster. It is only accessible from the API server for security reasons and enables notifications to the cluster about configuration changes.

2.4.2.2 Kubernetes Nodes

A Kubernetes cluster must have at least one compute node depending on the need of capacity. These nodes connect applications, compute and storage resources and their building blocks are being descripted below:

1. **Container Runtime Engine:** each compute node runs and manages container lifecycles using a Container Runtime Engine such as Docker.
2. **Kubelet:** is the main service of the node that communicates with the control plane to ensure that pods and their containers are healthy and running in the desired state. When the control plane requires a specific

[7] https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

action to happen in a node, the kubelet receives the new pod configuration and specifications through the API server and executes the appropriate action.

3. **Kube-Proxy**: is a proxy service that runs on each worker node in order to deal with the individual host subnetting and expose services to the external world. It forwards traffic itself or relies on the packet filtering layer of the operating system to handle network communications both outside and inside the cluster.

2.4.3 Kubernetes Objects

In this section, the most important Kubernetes Objects will be discussed which are:

1. **Pods:** represent a single instance of an application and are the simplest unit within the Kubernetes object model. It is a group of one or more containers with shared storage and network resources and a specification on how to run the containers. Two or more pods can communicate with each other using a pod network. A pod network is a medium of communication between pods and nodes. In this thesis a Flannel[8] pod network is being deployed in the Kubernetes cluster.
2. **Services:** are the Kubernetes way of configuring a proxy to forward traffic to a set of pods. Services can expose a single or multiple pods to external or internal consumers and associate specific criteria with pods to enable their discovery. Pods and services are associated through key-value pairs called selectors. With the help of selectors, services define which pods uses which service. These dynamic assignments make releasing new versions or adding pods to a service easy. There are four different service types, each with a different behavior:
    - **ClusterIP**: exposes the service on an internal IP making the service reachable only from within the cluster.

[8] https://platform9.com/docs/kubernetes/networking-integration-with-flannel

- **NodePort:** exposes the service on each node's IP at a specific port. In that way, developers have the freedom to configure their own environments. For example, NodePort was being used in this thesis for exporting Prometheus and Grafana from a node in NITOS Testbed Laboratory to a local machine in order to visualize the results.
- **LoadBalancer:** unlike ClusterIP, exposes the service externally using a cloud provider's load balancer.
- **ExternalName:** will just map a CNAME record in DNS. No proxying of any kind is established. This is commonly used to create a service within Kubernetes to represent an external datastore like a database for example, that runs externally to Kubernetes.

3. **Deployments:** is a resource object in Kubernetes that provides declarative updates to applications. A deployment allows the developers to describe an application's lifecycle such as which image to use for the app, the number of pods there should be and the way which they should be updated.
4. **Volume:** is just a directory that is accessible to a pod, which may hold data. The reason that Kubernetes Volumes are being used is that they solve two problems. First, the loss of files when a container crashes and second, the way that containers that exists in the same pod, are sharing files. The contents of the volume, how it comes to be and the medium that backs it, are determined by the volume type. Volumes can be separated into two categories:
    - **PersistentVolumes:** or (PVs), are specific to a cluster, provisioned by an administrator and tie into an existing storage resource. PVs have a lifecycle independent of any individual Pod that uses them. The PersistentVolumeClaims (PVCs) could also exist in this category. A PersistentVolumeClaim makes a storage consumption request within a namespace. It is similar to a pod. PVCs

consume PV resources just like pods consume node resources.

- **EphemeralVolumes**: unlike PersistentVolumes, ephemeral ones follow the pod's lifetime and get created and deleted along with the pod. Caching services are a descriptive example of EphemeralVolumes.

Sometimes, cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than size and access modes, without exposing users to the details of how those volumes are implemented. For these needs, there is the StorageClass resource. A StorageClass provides a way to describe the "classes" of storage they offer. This thesis uses a StorageClass in order to provision the PersistentVolumes.

5. **Namespaces:** are virtual clusters that exist inside a physical one. They are intended to provide virtually separated work environments for multiple users, teams and prevent each one from hindering each other by limiting what Kubernetes objects they can access.

2.4.4 Kubernetes Autoscaling

As mentioned above, Kubernetes has a feature called autoscaling. Autoscaling allows the cluster to dynamically adjust to demand without the intervention from the individuals in charge of operating the cluster. It is an important concept in cloud automation overall. Without autoscaling, the developers must manually provision resources every time conditions change, and it is less likely to be operating with optimal resource utilization and cloud spending. There are 3 autoscaler types:

- **Cluster Autoscaler (CA):** is a Kubernetes tool that increases or decreases the size of a cluster by simply adding or removing nodes, based on the presence of pending pods and node utilization metrics, as shown in the image[9] below:

[9] https://medium.com/kubecost/understanding-kubernetes-cluster-autoscaling-675099a1db92

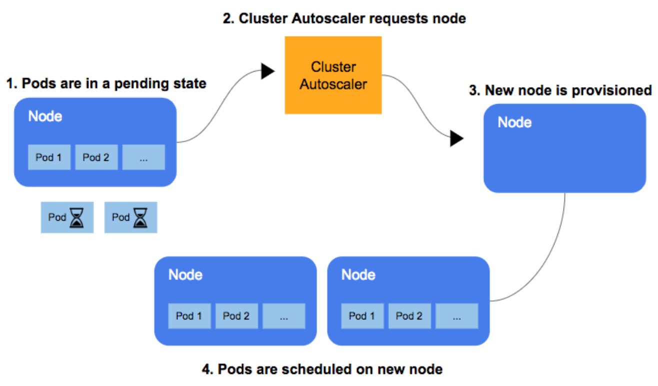

2. Cluster Autoscaler requests node
Cluster Autoscaler
1. Pods are in a pending state
Node
Pod 1
Pod 2
...
Pod
Pod
3. New node is provisioned
Node
Node
Node
4. Pods are scheduled on new node

Figure 5. Cluster Autoscaling

- **Horizontal Pod Autoscaler (HPA):** automatically scales the number of Pods (Figure 6) in a replication controller, deployment, replica set or stateful set based on observed CPU utilization or custom metrics (like this diploma thesis implements).
- **Vertical Pod Autoscaler (VPA):** dynamically modifying the attributed resources like CPU and RAM of each node in the cluster by adjusting the resource requests and limits based on the current application requirements as shown in the image[10] below:

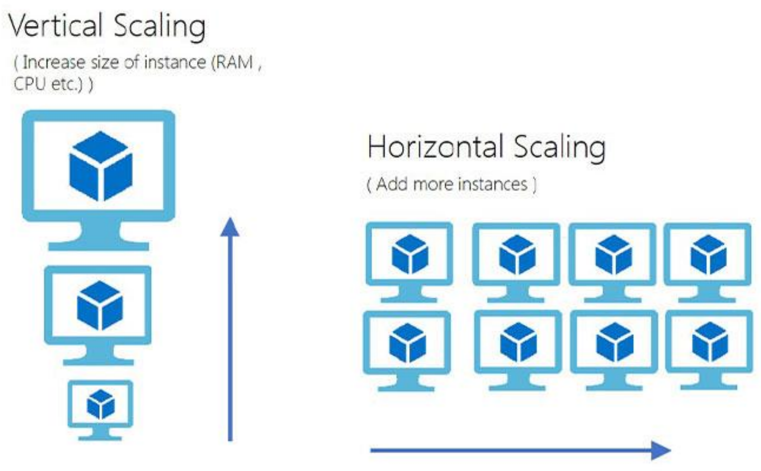


Figure 6. Vertical vs Horizontal Scaling

[10] https://www.webairy.com/wp-content/uploads/2019/07/hvsv.jpg

In this thesis, we are focusing on the HPA by using Machine Learning to forecast the future traffic to a server and comparing the results using this solution comparatively to the HPA.

2.4.4.1 Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler [8] is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The controller adjusts the number of replicas based on the observed metrics to the target specified by the developers. The controller fetches metrics from the Resource Metrics API (for per-pod resource metrics) or the custom metrics API (for all other metrics). An image[11] is being shown below describing how the HPA works, using the Resource Metrics API.

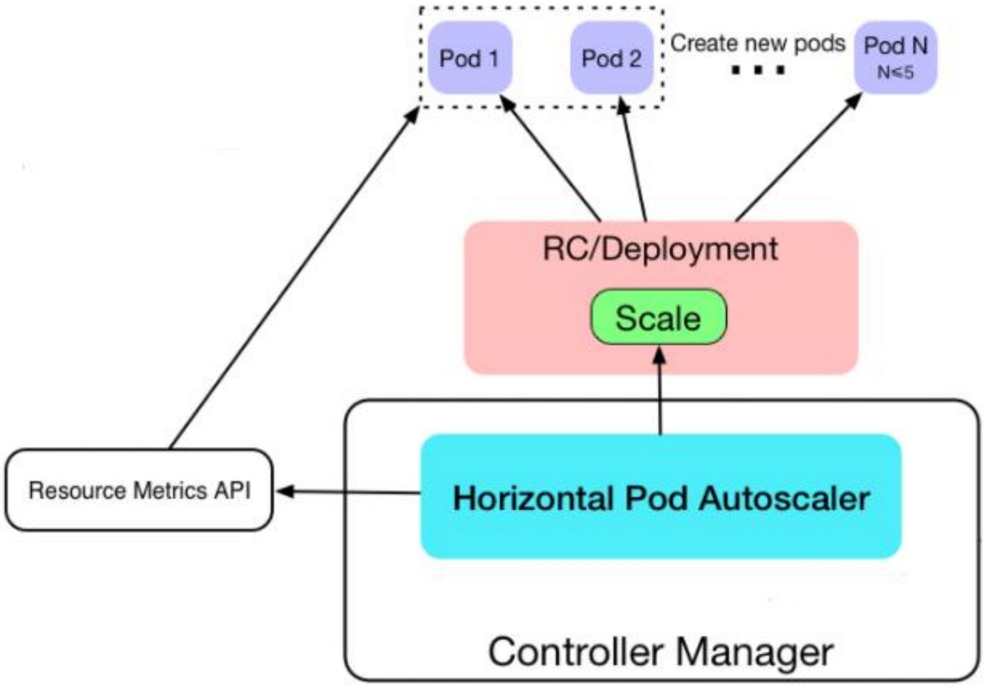


Figure 7. Horizontal Pod Autoscaler Architecture

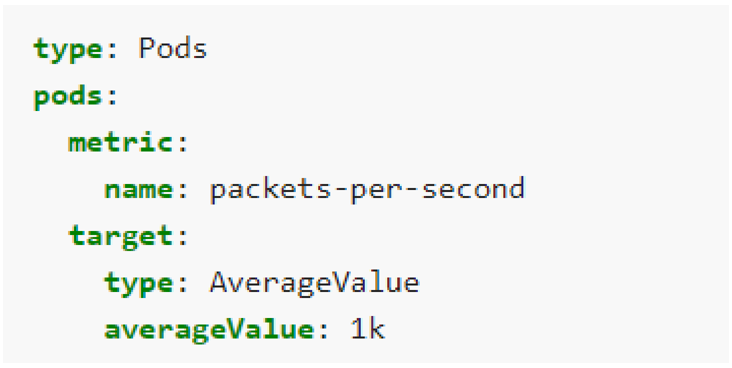
For resource metrics like CPU, the controller fetches the metrics from the resource metrics API for each pod targeted by the HPA. Then, if a target utilization value is set, the controller calculates the utilization values as a percentage of the equivalent resource

[11] https://dasydong.github.io/blog/2019/07/06/k8s-hpa/

request on the containers for each pod. If a target raw value is set, the raw metric values are used directly. Then, the controller takes the mean of the utilization or the raw value across all targeted pods and produces a ratio used to scale the number of desired replicas. Replicas are simply identical pods and is a term that it will be used a lot describing the HPA.

For custom metrics, the controller functions similarly to per-pod resource metrics, except that it works with raw values and not utilization ones. More details on how custom metrics can be obtained from a Kubernetes system are provided below.

There are two types of custom metrics: pod metrics and object metrics. The first of these describes pods and are averaged together across pods and compared with a target value to determine the replica count. They work much like resource metrics, except that they only support a target type of AverageValue. These metrics are specified using a metric block in the .yaml file that describes the HPA. Pod metrics are specified using a metric block[12] like this:



```yaml
type: Pods
pods:
  metric:
    name: packets-per-second
  target:
    type: AverageValue
    averageValue: 1k
```

Figure 8. Pod Metrics packets-per-second example

As far as the object metrics, they describe a different object in the same namespace, instead of describing pods. Object metrics support target types of both Value for direct comparison with the target metric and AverageValue for comparison between the target value and the value that returned from the API divided by the number of pods. The following example[13] is a visual representation of the extra configuration of a .yaml file where object metrics, specifically requests-per-second, are specified.

[12] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/
[13] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/

```yaml
type: Object
object:
  metric:
    name: requests-per-second
  describedObject:
    apiVersion: networking.k8s.io/v1beta1
    kind: Ingress
    name: main-route
  target:
    type: Value
    value: 2k
```

Figure 9. Pod Metrics requests-per-second example

2.4.4.2 Algorithm Details

The Horizontal Pod Autoscaler controller operates on the ratio between desired metric value and current metric value as shown below:

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

Figure 10. HPA's scale policy

The currentMetricValue is computed taking the average of the given metric across all pods in HPA’s scale target if a targetAverageValue or targetAverageUtilization is specified or taking the direct value if these fields are not specified. Before checking the tolerance and deciding on the final values, pod readiness and missing metrics are being taken into consideration. For example, pods with a deletion timestamp set and pods with missing metric are being discarded.
Then, the ratio between this currentMetricValue and the desiredMetricValue multiplied by the number of current replicas is being calculated rounding this result upwards. Finally, this is the desired number of replicas for the given case.

2.4.5 Metrics

Monitoring Kubernetes is also important in the context of resource usage, utilization and cost control. Kubernetes clusters must be actively managed to ensure pods utilize underlying node resources efficiently. The same is true for resources allocated to individual container or namespaces. As far as the tools that helped fetching and visualizing the metrics that this thesis used are: Prometheus, Prometheus Adapter and Grafana.

2.4.5.1 Prometheus

Prometheus [9] is a standalone open-source systems monitoring and alerting toolkit. It is a leading monitoring solution that has seen its community grow to large numbers. It collects and stores its metrics as time series data i.e., metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels. Prometheus sends an HTTP request, a so-called scrape, based on the configuration defined in the deployment file. The response to this scape request is stored and parsed in local storage along with the metrics for the scrape itself. The storage is a custom database on the Prometheus server and can handle a massive influx of data. The data needs to be appropriately exposed and formatted so that Prometheus can collect it. Prometheus can access data directly from the app's client libraries or by using exporters (a piece of software that accept HTTP requests). So, every application on the Kubernetes cluster is available for metrics fetching. In addition, it is possible to configure the alerting rules in the Prometheus server so that it sends alerts to the AlertManager that will create notifications to different notification systems like Email, OpsGenie etc. One last component that must be mentioned is the Service Discovery which informs Prometheus where to look for data. The whole architecture is shown in the image[14] below:

[14] https://samirbehara.com/2019/05/30/cloud-native-monitoring-with-prometheus/

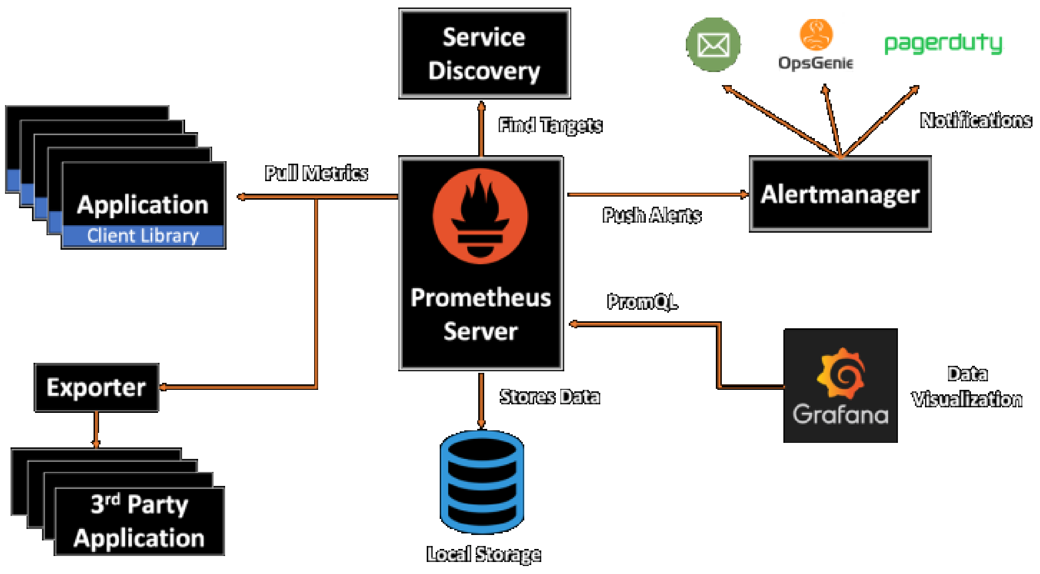


Figure 11. Prometheus Architecture

Prometheus was used for this thesis in order to get the necessary metrics such as replicas number and help visualizing them to Grafana. Prometheus Adapter was also deployed in the Kubernetes Cluster, and is being analyzed in the following section.

Underneath, the user interface of Prometheus is being shown where the developers can compose their queries and fetch the metrics they want:

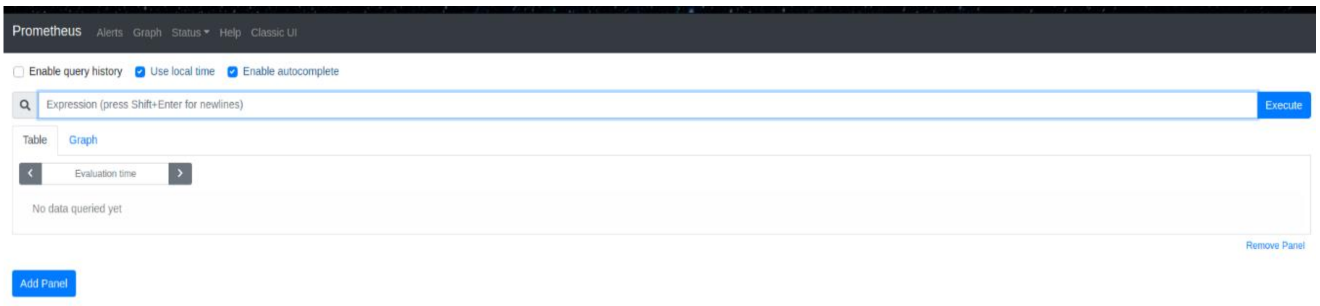


Figure 12. Prometheus User Interface

2.4.5.2 Prometheus Adapter

Prometheus Adapter [10] is a component of Prometheus and is used to leverage the metrics collected by Prometheus and use them to make scaling decisions. These metrics are exposed by an API service and can be used by the Horizontal Pod Autoscaler object. Simply, Prometheus Adapter pulls custom metrics from Prometheus, and it is running as a deployment exposed using a service in the Kubernetes cluster.

2.4.5.3 Grafana

Grafana [11] is a very common tool across Kubernetes that monitors the infrastructure and provides log analytics, predominantly to improve their operational efficiency. It comes up with a wide variety of dashboards that make tracking users and events easy just because it automates the collection, management and viewing of data and uses Prometheus as a data provider by fetching the appropriate metrics. This is how the Grafana Dashboards look [15]:

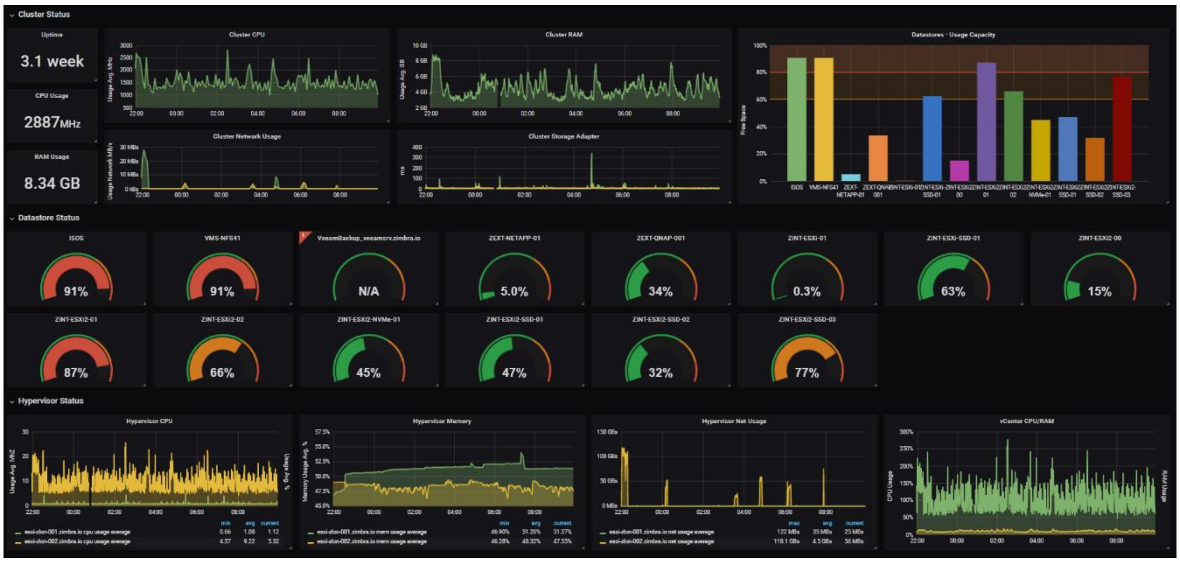


Figure 13. Grafana Dashboard User Interface

[15] https://techblog.commercetools.com/adding-consistency-and-automation-to-grafana-e99eb374fe40

2.5 Machine Learning

2.5.1 Overview

Machine Learning [12] is the study of computer algorithms that improve automatically through experience and by the use of data. It is seen as part of artificial intelligence. Machine Learning algorithms build a model based on sample data, known as "training data". As models are exposed to new data, they can independently adapt and make predictions or decisions without being explicitly programmed to do so. These algorithms learn from previous computations and identify common patterns in "training data" in order to produce reliable results.

2.5.2 Steps

To make things clearer about how Machine Learning algorithms operates, the seven steps [13] of ML[16] are being analyzed below.

1. **Data Gathering:** is the process of collecting and measuring information from different sources. It is an important part of Machine Learning because the developer must choose the appropriate data that will train the model in order to produce an effective one. The quantity and quality of them dictate how accurate the Machine Learning model will be.
2. **Data Preparation:** is the procedure of combining, structuring and organizing data so that it can be used by the algorithms. It involves cleaning the data by removing duplicates, correcting errors and dealing with missing values, in some cases randomizing the data which erases the effects of a particular order, visualizing them to help detect relationship between variables and finally, splitting the data into train and evaluation sets.
3. **Model Selection:** is made up of what Machine Learning model the developer should use. There is a wide variety of algorithms that had been developed for different purposes and each one of them is using a logic of its own. There are algorithms that produce predictions for image recognition, numerical data, text-based data and other sequences. This thesis uses time-series Machine

[16] Machine Learning

Learning algorithms that forecasts the future of a specific metric involving a time component.

4. **Model Training:** is used to train the model in order to make an as much as possible accurate prediction. It is an iterative process and is determining good values for all the weights and the bias from the "training data". Model Training is identifying common patterns amongst the data provided so that the features that best predict the outcomes to be determined.
5. **Model Evaluation:** is used to test the model accuracy against unseen data. These data are meant to be representative of model performance in the real world in order to get an estimation very close to reality. With the help of some metrics or combination of metrics the objective performance of a model is being measured.
6. **Parameter Tuning:** is all about tuning the Machine Learning model to improve its performance. By changing hyperparameters like training steps, learning rate, initialization values and many more, the developer can see differences in the predictions that the model produces choosing the best combination of these parameters.
7. **Predictions Making:** is the procedure of using data which have been withheld from the model in order to test the model by making predictions.

2.5.3 TensorFlow

TensorFlow [14] is a Python library for fast numerical computing created and released by Google. It is a foundation that can be used to create complex Machine Learning models directly or using wrapper libraries that simplify the process built on top of TensorFlow. It got its name from its core framework: **Tensor**. In Tensorflow, all the computation involve tensors. A tensor is a vector or matrix of n-dimensions that represents all types of data. All values in a tensor hold identical data type with a known shape. The shape of the data is the dimensionality of the matrix or array.

A tensor can be originated from the input data or the result of a computation. In Tensorflow, all the operations are conducted inside a **graph**. The graph is a set of computation that takes place successively. Each operation is called an op node and are connected to each other. Underneath, an example of an acyclic graph is presented:

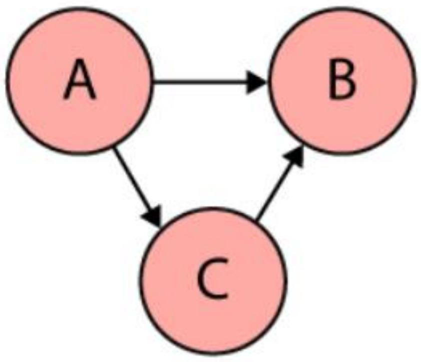


Figure 14. Tensorflow acyclic graph

Graphs come in many shapes and sizes and are used to solve many real-life problems such as representing circuit networks. They gather and describe all the series computation done during the training.

All in all, TensorFlow was being used in this diploma thesis to implement and use the LSTM Machine Learning algorithm with the help of one of its library called keras[17] as descripted in the section 3.2.7.

[17] https://keras.io/

Chapter 3: Development and Implementation

3.1 Chapter Introduction

In this chapter, the implementation is being discussed. How the tools that were described in Chapter 2, were used in order to achieve the desired outcome. It analyzes step by step the methodology that had been followed. It is separated into 4 sections: the Environment Set-up analyzing the how this environment initialized, the Machine Learning Algorithms covering which time-series Machine Learning algorithms used and information about them, the System Architecture explaining how this environment is designed and finally, the Scale Policy covering the algorithm behind the scale ups and downs.

3.2 Environment Set-Up

3.2.1 Kubernetes

As mentioned before, for the implementation of the whole experiment NITOS Testbed was being used. Two of these nodes were used, one operating as a Kubernetes master and the other as a Kubernetes worker. So, in order to utilize Kubernetes, a cluster must be deployed. A Kubernetes cluster is simply, a set of node machines that run containerized applications. After installing Docker and adding Kubernetes signing key and repository on both nodes, it was time to initialize the cluster. This was done with the help of Kubeadm, which is a tool for creating a viable Kubernetes cluster, with the following command:

```
$ sudo kubeadm init –pod-network-cidr=10.244.0.0/16
```

The above command is being executed on one node only, making it the Kubernetes Master node and the flag specify the range of IP addresses for the pod network, automatically allocating CIDRs (blocks of unique IP addresses) for every node.

The output of this command is being displayed below, pointing out that some extra things still need to be done.

```
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 192.168.100.6:6443 –token 06tl4c.oqn35jzecidg0r0m –discovery-token-ca-
cert-hash sha256:c40f5fa0aba6ba311efcb0e8cb637ae0eb8ce27b7a03d47be6d966142f2204c
```

After successfully initializing the Kubernetes Master node, a folder must be created to $Home/.kube in order to copy the configuration files to the local machine. In that way, the communication with the API Server becomes possible using kubectl commands. Then, a pod network must be deployed which is the medium between the network of the nodes. In this diploma thesis, Flannel was being used with the following command.

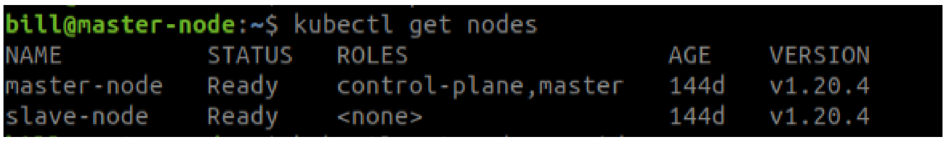
```
$ sudo kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-
flannel.yml[18]
```

Now, everything is ready for the other node, Kubernetes Worker to join the Master. This is done with the command below.

[18] https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

```
$ sudo kubeadm join 192.168.100.6:6443 –token 06t14c.oqn35jzecidg0r0m –discovery-token-ca-cert-hash
sha256:c40f5fa0aba6ba311efcdb0e8cb637ae0eb8ce27b7a03d47be6d966142f2204c
```

Checking the status of the cluster with kubectl just like below, shows that our cluster is healthy and running and consists of two nodes.

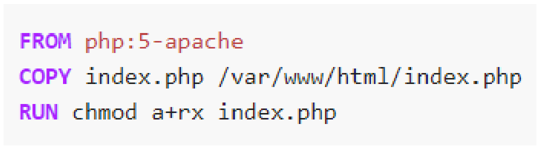


```
bill@master-node:~$ kubectl get nodes
NAME          STATUS   ROLES                  AGE    VERSION
master-node   Ready    control-plane,master   144d   v1.20.4
slave-node    Ready    <none>                 144d   v1.20.4
```

Figure 15. Kubernetes Cluster Checking

3.2.2 Php-Apache Deployment and Ubuntu Pod

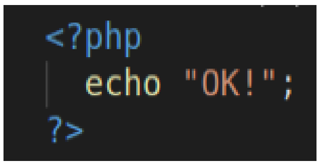
Having the Kubernetes Cluster ready with the Master and Worker up and running, more things had to be done. In order to put the whole experiment to the trial and to demonstrate the results a deployment called php-apache was created. This deployment runs a custom Docker image with the following content:



```
FROM php:5-apache
COPY index.php /var/www/html/index.php
RUN chmod a+rx index.php
```

Figure 16. Custom Docker Image

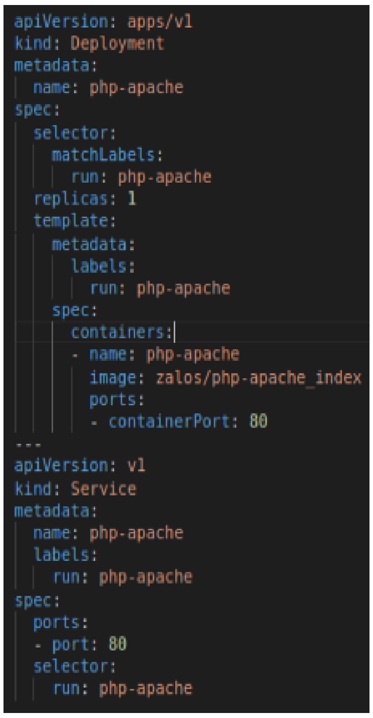
It defines an index.php page (server) which simply responds to every request with an “OK!” string, as shown below:



```
<?php
  echo "OK!";
?>
```

Figure 17. Index.php code

The .yaml file that starts a deployment running this image and exposing it as a service making it visible for the whole Cluster, is being represented below:



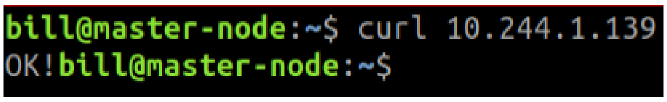
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 1
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: zalos/php-apache_index
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  labels:
    run: php-apache
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
```

Figure 18. Php-apache.yaml code

Running the following command, will create the deployment with the above configuration.

```
$ kubectl apply -f php-apache.yaml
```

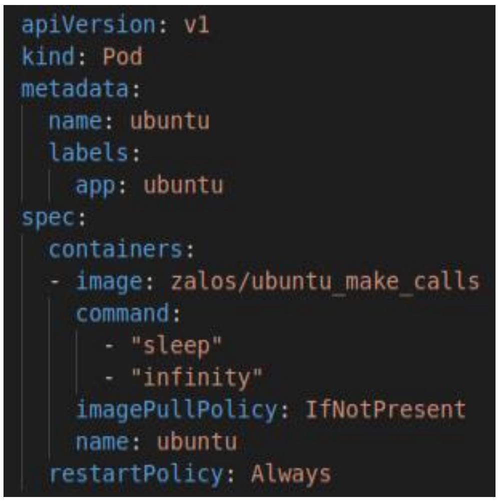
So, running a simple "curl <<php-apache-deployment IP>>" command will just display the "OK!" string as shown beneath.



```
bill@master-node:~$ curl 10.244.1.139
OK!bill@master-node:~$
```

Figure 19. Curl test

But there's a twist right there. This <<php-apache IP>>, which was retrieved from the "kubectl get pods" command, is the IP of the first replica. This is wrong because when generating traffic, it must be distributed to all the replicas not just the first one.
For this reason, a Pod called Ubuntu was created. This idea behind is that when an experiment is being held that uses a lot of "curl" commands to simulate traffic to the php-apache server in the deployment that descripted above, the traffic must be scattered equally across all replicas. This can be done by sending requests from another Pod to the php-apache server in the same Kubernetes Cluster. Below, is being displayed its configuration.



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu
  labels:
    app: ubuntu
spec:
  containers:
  - image: zalos/ubuntu_make_calls
    command:
      - "sleep"
      - "infinity"
    imagePullPolicy: IfNotPresent
    name: ubuntu
  restartPolicy: Always
```

Figure 20. Ubuntu.yaml code

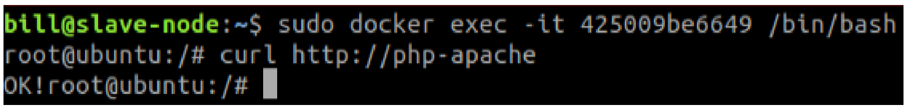
Running the following command, will create this Pod that is being discussed.

```
$ kubectl apply -f ubuntu.yaml
```

So, in every experiment that took place in this diploma thesis, the traffic was being sent to the php-apache server from the "Ubuntu" Pod. In order to get the response from the server an interaction with this running container must be done, with the help of this command:

```
$ sudo docker exec -it <<container-id>> /bin/bash
```

This command enters a bash shell session with the container specified. In this way, running the following command inside this container instead of "curl <<php-apache-deployment IP>>" from the Kubernetes Master node will bring the desired result as displayed underneath:

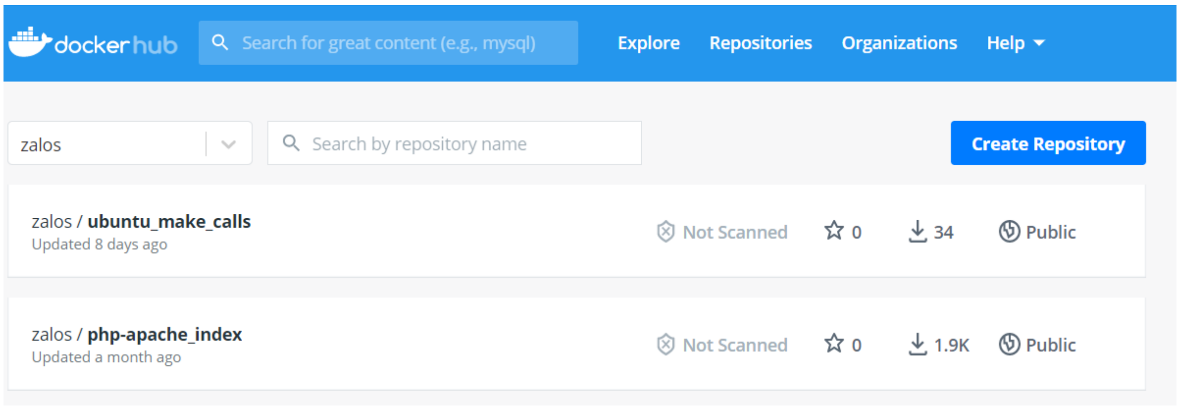


```
bill@slave-node:~$ sudo docker exec -it 425009be6649 /bin/bash
root@ubuntu:/# curl http://php-apache
OK!root@ubuntu:/#
```

Figure 21. Curl test inside pod

3.2.3 DockerHub

It can be observed from the "image" header in the configuration files of the "Ubuntu" Pod and the "Php-apache" Deployment (Figure 20, Figure 18) that the first one uses an image called "zalos/ubuntu_make_calls" and the second one an image with "zalos/php-apache_index" name. These are custom Docker images that are being pulled every time the cluster initializes, from an online repository that allows to share container images, called DockerHub [15].
After creating an account, these two images were uploaded online, making it possible to "pull" them every time the Kubernetes Cluster starts, as it can be seen below:



zalos / **ubuntu_make_calls**
Updated 8 days ago
Not Scanned ☆ 0 34 Public

zalos / **php-apache_index**
Updated a month ago
Not Scanned ☆ 0 1.9K Public

Figure 22. DockerHub User Interface and Images used

3.2.4 Prometheus and Grafana

Monitoring is a crucial feature in the context of resource usage, utilization and cost control. Kubernetes clusters have to be actively managed to ensure pods utilize underlying node resources efficiently. It allows developers to understand application and user behavior quickly, identify bugs and scale an application due to usage as needed. There is a wide variety of monitoring tools with Prometheus and Grafana being the most popular. They can be used as complementary services that, together, provide a robust time-series database with excellent data visualization.

Before the installation of these two charts, a StorageClass deployment needs to be created. Kubernetes doesn't provide data persistence out-of-the-box. That's something the developers have to explicitly configure. Because dynamic persistent volume provisioning is needed, a namespace called local-path-provisioner is being created with a provisioner installed, by executing the following command:

```
$ kubectl apply -f https://github.com/rancher/local-path-
provisioner/blob/master/deploy/local-path-storage.yaml
```

At first, Prometheus needs to be installed with the help of Helm[19]. Helm is the package manager for Kubernetes, and it allows describing the application structure through convenient helm-charts and managing it with simple commands.
Four steps need to be made to install and check Prometheus using Helm:

1. Add the Prometheus repository with the following command.

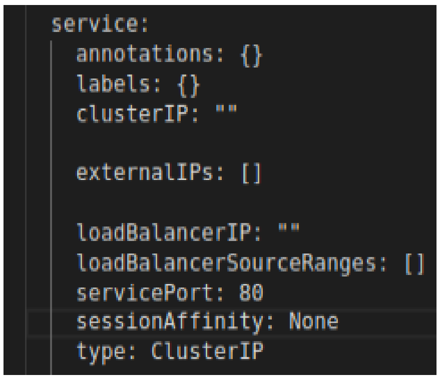
   ```
   $ helm repo add prometheus-community https://prometheus-
   community.github.io/helm-charts
   ```

2. Get the .values file from the repository and save it locally just like below:

   ```
   $ helm inspect values prometheus-community/prometheus >
   values/prometheus.values
   ```

[19] https://helm.sh/

3. Make the appropriate changes to this file and install. In server/service heading replace type:ClusterIP to type:NodePort and add nodePort:32322 as being depicted below:

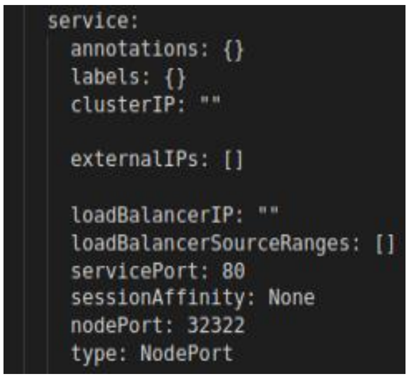


```
service:
  annotations: {}
  labels: {}
  clusterIP: ""

  externalIPs: []

  loadBalancerIP: ""
  loadBalancerSourceRanges: []
  servicePort: 80
  sessionAffinity: None
  type: ClusterIP
```

Figure 23. Prometheus Configuration before



```
service:
  annotations: {}
  labels: {}
  clusterIP: ""

  externalIPs: []

  loadBalancerIP: ""
  loadBalancerSourceRanges: []
  servicePort: 80
  sessionAffinity: None
  nodePort: 32322
  type: NodePort
```

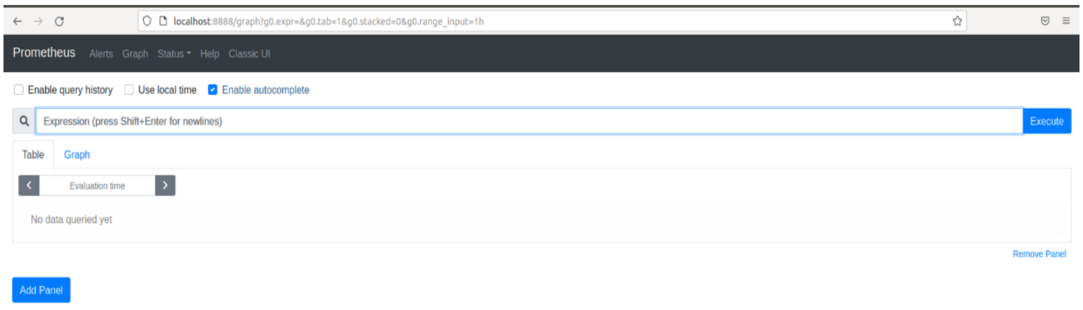
Figure 24. Prometheus Configuration after

This is done because Prometheus needs to be exported outside of the cluster so that, the user interface that it provides can be seen from a machine that isn't part of the Kubernetes cluster.

4. Run this ssh command matching the local IP (8888) that the Prometheus user interface will be displayed, the node IP of the NITOS Testbed (10.0.1.89) and the NodePort that was descripted above (32322) along with the username and the server name (vzalokost@nitlab3.inf.uth.gr).

```
$ ssh -L 8888:10.0.1.89:32322 vzalokost@nitlab3.inf.uth.gr
```

So, typing localhost:8888 in the local machine the user interface of Prometheus will appear indicating that everything is up and working.



localhost:8888/graph?g0.expr=&g0.tab=1&g0.stacked=0&g0.range_input=1h

Figure 25. Prometheus localhost checking

A similar procedure has been followed for the installation of Grafana. Its installation was done using Helm and these four steps where being described epigrammatically:

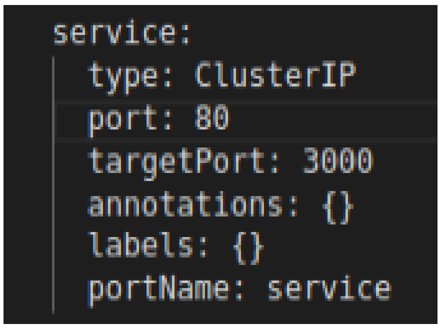
1. Add the Grafana repository just like below:

```
$ helm repo add grafana https://grafana.github.io/helm-charts
```

2. Fetch the .values file from the repository and save it locally with the following command:

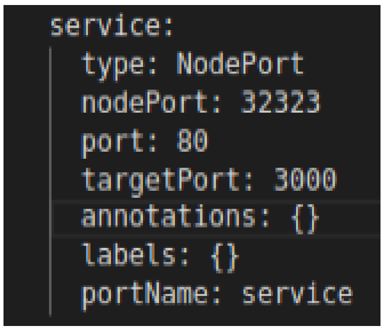
```
$ helm inspect values grafana/grafana > values/grafana.values
```

3. Make the appropriate changes to this file and install. In service heading replace type:ClusterIP to type:NodePort and add nodePort:32323 this time, as being depicted below for exporting the Grafana outside the cluster:



```
service:
  type: ClusterIP
  port: 80
  targetPort: 3000
  annotations: {}
  labels: {}
  portName: service
```

Figure 26. Grafana Configuration before



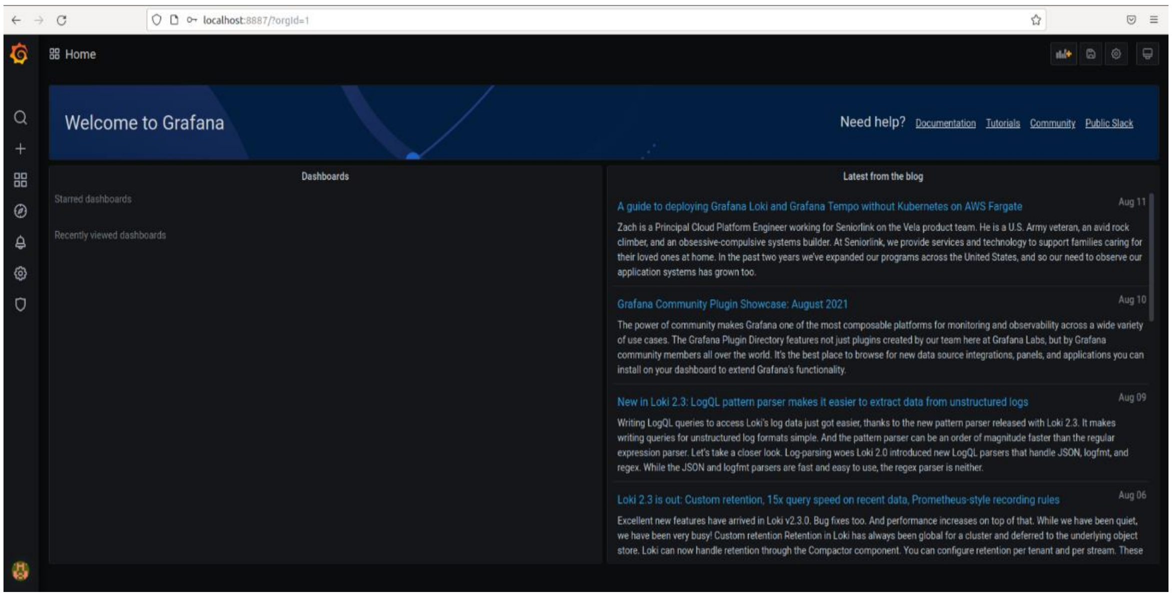
```
service:
  type: NodePort
  nodePort: 32323
  port: 80
  targetPort: 3000
  annotations: {}
  labels: {}
  portName: service
```

Figure 27. Grafana Configuration after

4. Run this ssh command matching the local IP (8887) that the Grafana user interface will be displayed, the node IP of the NITOS Testbed (10.0.1.89) and the NodePort that was descripted above (32323) along with the username and the server name (vzalokost@nitlab3.inf.uth.gr).

```
$ ssh -L 8887:10.0.1.89:32323 vzalokost@nitlab3.inf.uth.gr
```

So, typing localhost:8887 in the local machine the user interface of Grafana will appear.



A guide to deploying Grafana Loki and Grafana Tempo without Kubernetes on AWS Fargate — Aug 11

Zach is a Principal Cloud Platform Engineer working for Seniorlink on the Vela product team. He is a U.S. Army veteran, an avid rock climber, and an obsessive-compulsive systems builder. At Seniorlink, we provide services and technology to support families caring for their loved ones at home. In the past two years we've expanded our programs across the United States, and so our need to observe our application systems has grown too.

Grafana Community Plugin Showcase: August 2021 — Aug 10

The power of community makes Grafana one of the most composable platforms for monitoring and observability across a wide variety of use cases. The Grafana Plugin Directory features not just plugins created by our team here at Grafana Labs, but by Grafana community members all over the world. It's the best place to browse for new data source integrations, panels, and applications you can install on your dashboard to extend Grafana's functionality.

New in Loki 2.3: LogQL pattern parser makes it easier to extract data from unstructured logs — Aug 09

Writing LogQL queries to access Loki's log data just got easier, thanks to the new pattern parser released with Loki 2.3. It makes writing queries for unstructured log formats simple. And the pattern parser can be an order of magnitude faster than the regular expression parser. Let's take a closer look. Log-parsing woes Loki 2.0 introduced new LogQL parsers that handle JSON, logfmt, and regex. While the JSON and logfmt parsers are fast and easy to use, the regex parser is neither.

Loki 2.3 is out: Custom retention, 15x query speed on recent data, Prometheus-style recording rules — Aug 06

Excellent new features have arrived in Loki v2.3.0. Bug fixes too. And performance increases on top of that. While we have been quiet, we have been very busy! Custom retention Retention in Loki has always been global for a cluster and deferred to the underlying object store. Loki can now handle retention through the Compactor component. You can configure retention per tenant and per stream. These

Figure 28. Grafana localhost checking

3.2.5 Metrics Query

Prometheus provides a function query language called PromQL (Prometheus Query Language) that lets developers to select and aggregate time series data in real time. The result of an expression can be shown as a graph or viewed as tabular data in Prometheus's expression browser.

So, the question that raises here is which query to choose and why?

Just like the Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration paper [16] a lot of queries were tested. Beyond CPU or Memory metrics, a very good solution is to scale based on some query that describes the traffic that the php-apache deployment receives. As the traffic requests are becoming more and more often, some additional scale ups must be done so that the resources won't overload from enormous amount of requests that they must carry out. Accordingly, when in lower traffic request rate, scale downs must occur in order to not have resources allocated without serving any job.

Thus, a metric that is ideal for this case is the following:

```
avg(rate(container_network_receive_packets_total{pod=~"php-apache.+"}[10m]))
```

Separating this query to three parts, will help to better understand it:

1. **container_network_receive_packets_total** = is a counter that describes the cumulative count of packets that a container receives. Here it should be noted that this query returns as many results as the number of replicas of this container.
2. **container_network_receive_packets_total{pod=~"php-apache.+"}** = is a label that filter the output of the container_network_receive_packets_total query and returns the cumulative packet count that a container receives whose pod has the "php-apache" string inside its name. Basically, it returns the cumulative number of received packets for each replica of the php-apache container.
3. **rate(container_network_receive_packets_total{pod=~"php-apache.+"}[10m])** = calculates the per-second rate of packets received in the php-apache server as measured over the last 10 minutes for each replica separately.
4. **avg(rate(container_network_receive_packets_total{pod=~"php-apache.+"}[10m]))** = returns the average rate of cumulative packets received across all replicas for 10 minutes.

3.2.6 Prometheus Adapter

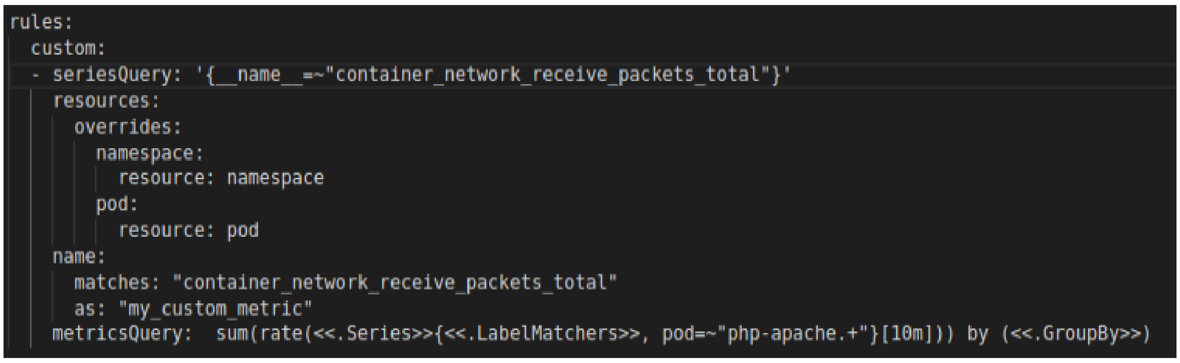
Prometheus is the standard toll for monitoring deployed workloads and the Kubernetes cluster itself. It has a custom component called Prometheus Adapter, which works like a query in the Prometheus database by gathering selected metrics and is mainly used to make scaling decisions. These metrics are being taken advantage both by the Horizontal Pod Autoscaler object and the developers for fetching and using them as they want.

In order to install the Prometheus Adapter, helm was also used following the previous procedure:

1. The Prometheus repository has already been added.
2. Get the .values file from the repository and save it locally just like below:

```
$ helm inspect values prometheus-community/Prometheus-adapter >
values/new_prometheus_adapter.values
```

3. Make the appropriate changes to this file and install. A custom rule has been defined fetching the rate of cumulative received packets of the php-apache server for each replica and matching this metrics with “my_custom_metric“ string as being depicted below:



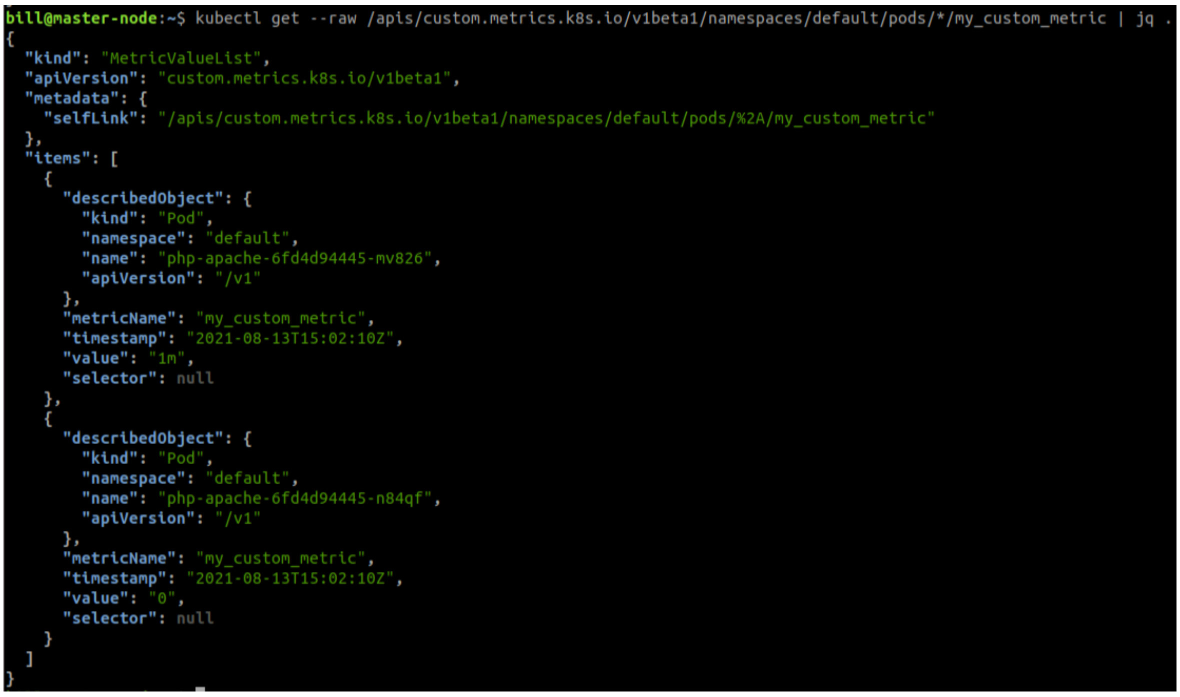
```
rules:
  custom:
  - seriesQuery: '{__name__=~"container_network_receive_packets_total"}'
    resources:
      overrides:
        namespace:
          resource: namespace
        pod:
          resource: pod
    name:
      matches: "container_network_receive_packets_total"
      as: "my_custom_metric"
    metricsQuery:  sum(rate(<<.Series>>{<<.LabelMatchers>>, pod=~"php-apache.+"}[10m])) by (<<.GroupBy>>)
```

Figure 29. Prometheus Adapter Configuration

4. In order to verify that the data is being exposed properly, the following command is used:

```
$ kubectl get --raw
/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/pods/*/my_custom_
metric |  jq .
```

This command returns as many instances as the number of replicas. Here is an example of 2 replicas.



```
bill@master-node:~$ kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1/namespaces/default/pods/*/my_custom_metric | jq .
{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/pods/%2A/my_custom_metric"
  },
  "items": [
    {
      "describedObject": {
        "kind": "Pod",
        "namespace": "default",
        "name": "php-apache-6fd4d94445-mv826",
        "apiVersion": "/v1"
      },
      "metricName": "my_custom_metric",
      "timestamp": "2021-08-13T15:02:10Z",
      "value": "1m",
      "selector": null
    },
    {
      "describedObject": {
        "kind": "Pod",
        "namespace": "default",
        "name": "php-apache-6fd4d94445-n84qf",
        "apiVersion": "/v1"
      },
      "metricName": "my_custom_metric",
      "timestamp": "2021-08-13T15:02:10Z",
      "value": "0",
      "selector": null
    }
  ]
}
```

Figure 30. Prometheus Adapter Checking

3.2.7 TensorFlow

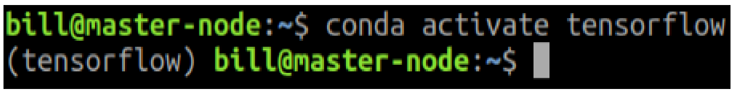
TensorFlow, being a Python library for fast numerical computing that makes easy for the developers to build and deploy Machine Learning models, was necessary for this diploma thesis. With the help of TensorFlow, the time series ML models were being used such as LSTM. Python3.7 and Miniconda were being used for the installation.

Conda is an open-source package management system and environment management system that quickly installs, runs and updates packages and their dependencies. Miniconda is simply a minimal installer for conda. After successfully installing Miniconda from this link: https://docs.conda.io/en/latest/miniconda.html, the following command was used:

```
$ conda create --name tensorflow python3.7 && conda install -c anaconda tensorflow
```

It creates a Python environment called "tensorflow" and installs tensorflow within.

Now, the only thing that left is to enter this environment and install the necessary packages for the Machine Learning models. Underneath is shown the Python environment that each ML algorithm ran.

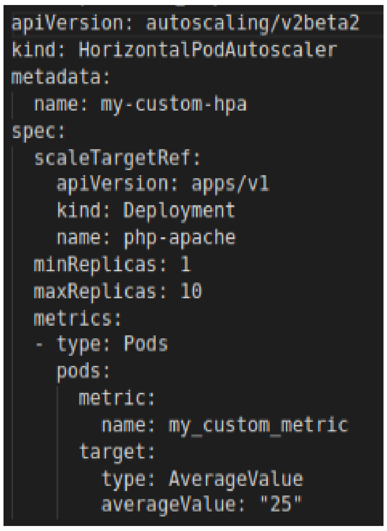


```
bill@master-node:~$ conda activate tensorflow
(tensorflow) bill@master-node:~$ 
```

Figure 31. Python Environment

3.2.8 Horizontal Pod Autoscaler

Horizontal Pod Autoscaler automatically scales the number of Pods based on a specified metric. In this diploma thesis, because the scaling was based on custom metrics (container_network_receive_packets_total), the HPA must fetch this metric from the Prometheus Adapter. As described above, the Prometheus Adapter associates the metrics query provided, with a variable called "my_custom_metric" which then, HPA uses for input metric in order to decide whether to scale or not. The configuration of the Horizontal Pod Autoscaler can be seen below:



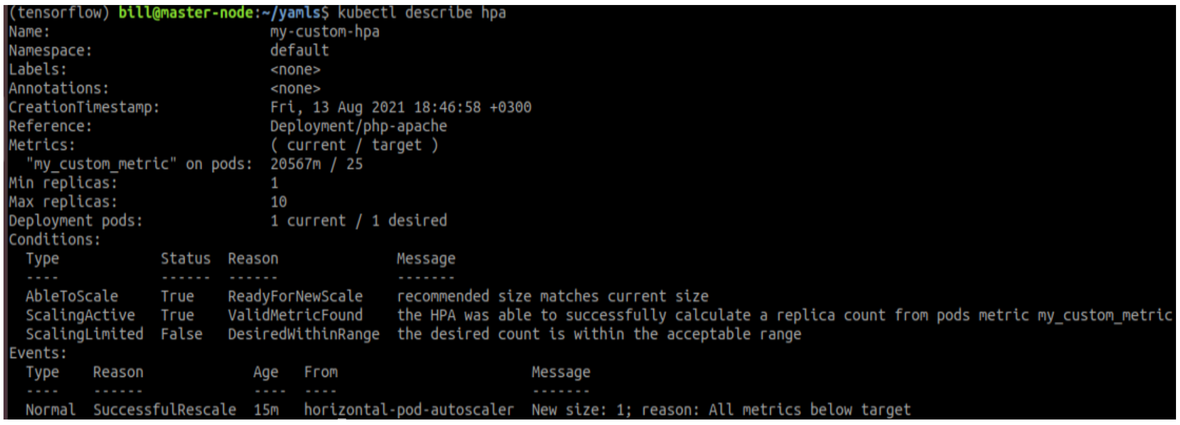
```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: my-custom-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Pods
    pods:
      metric:
        name: my_custom_metric
      target:
        type: AverageValue
        averageValue: "25"
```

Figure 32. Horizontal Pod Autoscaler Configuration

```
$ kubectl apply -f hpa.yaml
```

A Horizontal Pod Autoscaler object was created, with autoscaling/v2beta2 version, with minimum number of replicas equal to 1 and maximum to 10 being named as "my_custom_hpa". The metric fetched is the one described earlier, and the scaling is done based on an average value of 25 of the php-apache server.

So, typing "kubectl describe hpa" command, will ensure that the Horizontal Pod Autoscaler is running properly:



```
(tensorflow) bill@master-node:~/yamls$ kubectl describe hpa
Name:                          my-custom-hpa
Namespace:                     default
Labels:                        <none>
Annotations:                   <none>
CreationTimestamp:             Fri, 13 Aug 2021 18:46:58 +0300
Reference:                     Deployment/php-apache
Metrics:                       ( current / target )
  "my_custom_metric" on pods:  20567m / 25
Min replicas:                  1
Max replicas:                  10
Deployment pods:               1 current / 1 desired
Conditions:
  Type            Status  Reason              Message
  ----            ------  ------              -------
  AbleToScale     True    ReadyForNewScale    recommended size matches current size
  ScalingActive   True    ValidMetricFound    the HPA was able to successfully calculate a replica count from pods metric my_custom_metric
  ScalingLimited  False   DesiredWithinRange  the desired count is within the acceptable range
Events:
  Type    Reason             Age   From                       Message
  ----    ------             ----  ----                       -------
  Normal  SuccessfulRescale  15m   horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

Figure 33. Horizontal Pod Autoscaler Checking

3.3 Machine Learning

3.3.1 Overview

In order to learn from the past trends, identify patterns and make decisions about the future this thesis used machine learning algorithms that involve a time component called time series machine learning algorithms. Time series forecasting can be categorized into the following series:

1. **Classical / Statistical Models** = models that have mainly strong base in statistics like Moving Averages and Exponential Smoothing.
2. **Machine Learning** = models with reduction methods such as Random Forests.
3. **Deep Learning** = complex neural networks with a time component like RNN.

A model from it's of these categories was used. Arima for Classical models, XGBoost for Machine Learning models and LSTM for Deep Learning models. Each algorithm was utilized in order to make a 5-step out-of-sample prediction on the average rate of cumulative packets received from the php-apache server so that there's time for the appropriate scaling of the deployment before the predicted traffic arrives.

3.3.2 Methodology

Before explaining separately every algorithm that had been used, the general idea of how the 5-step out-of-sample predictions must be mentioned. At first, every model takes a dataset for input. This dataset is a .csv file that represents the average rate of the packets sent to the server and is used for training. Then, the Machine Learning model is being fitted on the training data so that it can be used for forecasting. The forecast that is being produced is just an observation that is not part of the input data and that's why it is called out-of-sample. Now, in order to make a 5-step out-of-sample forecast, every prediction that is being made, is used as input for the next one and the whole process repeats 5 times. To make things clearer, an example of a 3-step prediction is being presented in the following picture [20].

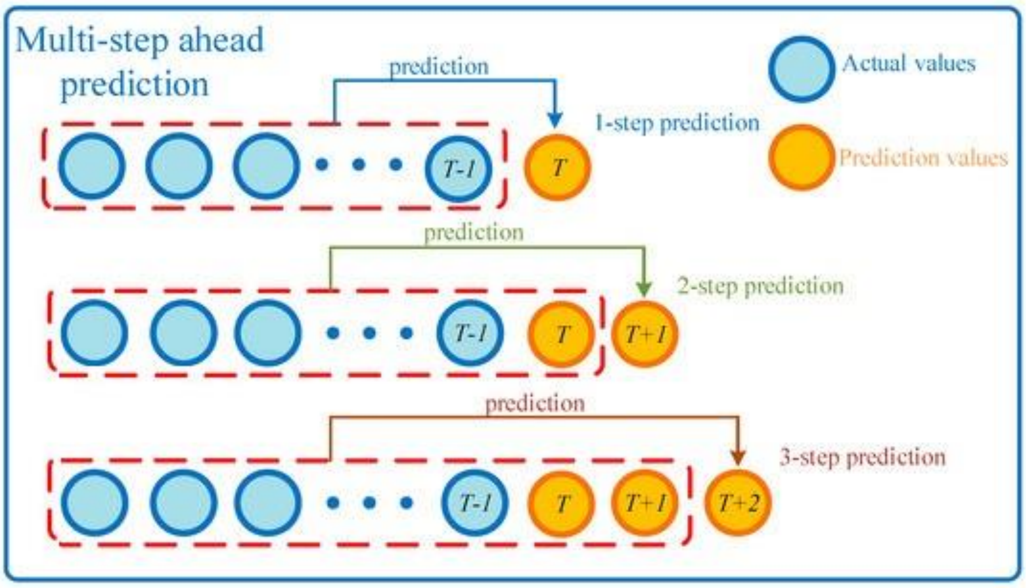


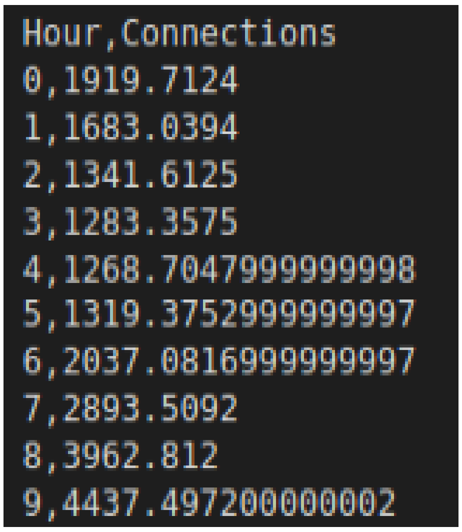
Figure 34. Multi-step prediction basic idea

[20] https://www.mdpi.com/1996-1073/13/16/4121/htm (Figure 3)

Here every blue dots represent the actual values from the input dataset and the yellow are the predicted values. So, in the first step, a yellow dot has been produced with the help of Machine Learning algorithms, that represents a future value at time T. Then, this prediction plus the input dataset are used as input by appending it to the dataset in order to be fitted by the model and a new forecast to be brought out at time T+1. The same thing is done for the T+2 dot. Furthermore, when the 3-step prediction is being made, the 3 predicted values are then being replaced with actual values making every dot blue and this process repeats all over again. Thus, the prediction for the time T+1 is based on the whole dataset plus T, the forecast on T+2 is based on the dataset plus T and T+1, and T+3 on the dataset plus on T,T+1,T+2.

3.3.3 Datasets

As far as the datasets [17] [18] that had been used, they provide information about the telecommunication activity (SMS, data, calls) over the city of Milan. Two datasets were used and got renamed to high.csv and medium.csv representing the number of connections to a base station as being shown below:



```
Hour,Connections
0,1919.7124
1,1683.0394
2,1341.6125
3,1283.3575
4,1268.7047999999998
5,1319.3752999999997
6,2037.0816999999997
7,2893.5092
8,3962.812
9,4437.497200000002
```

Figure 35. Dataset Format

After rounding up the Connections column, underneath there’s a plot that shown both datasets.

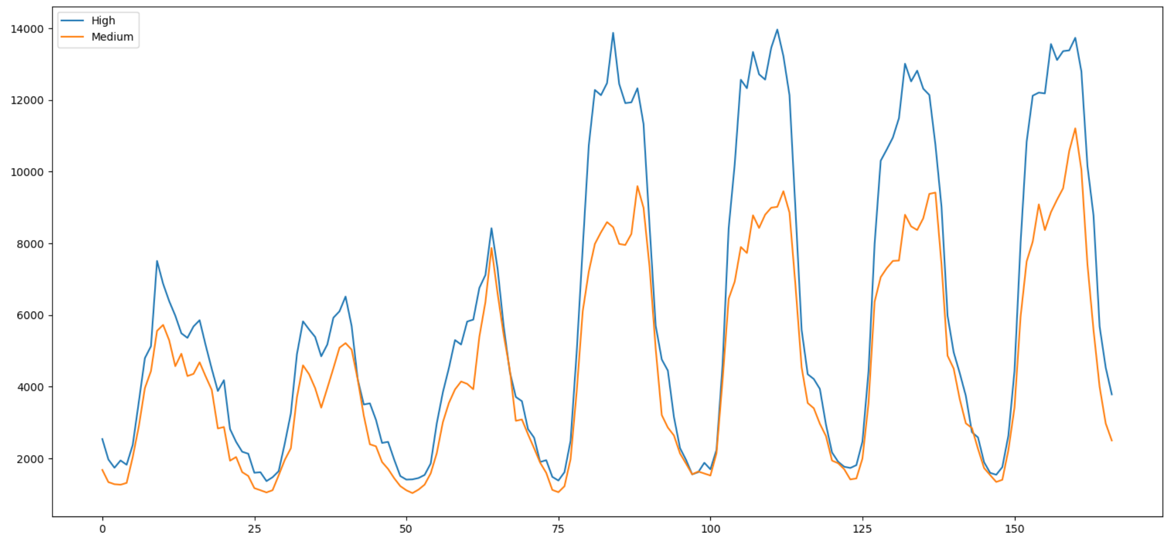


Chart 1. High and Medium Datasets

The high.csv, as its name suggests has higher number of connections than the medium.csv. However, none of these datasets were used for training. As 4.2 section explains in detail, these datasets generated traffic to the php-apache server with a simple curl command for every connection. For example, the first hour as it can be seen in Figure 31 has 1919.7 connections. So, 1918 HTTP requests were sent with each one displaying the “OK!” string, as descripted in 3.2.2.

The dataset that each Machine Learning algorithm used for training is the one each was retrieved from Grafana with the following query:

```
avg(rate(container_network_receive_packets_total{pod=~"php-apache.+"}[10m]))
```

It returns the average rate of cumulative packets received across all replicas for 10 minutes. Beneath is displayed the training dataset across 10 replicas of 1412 points based on the high.csv dataset to generate traffic:

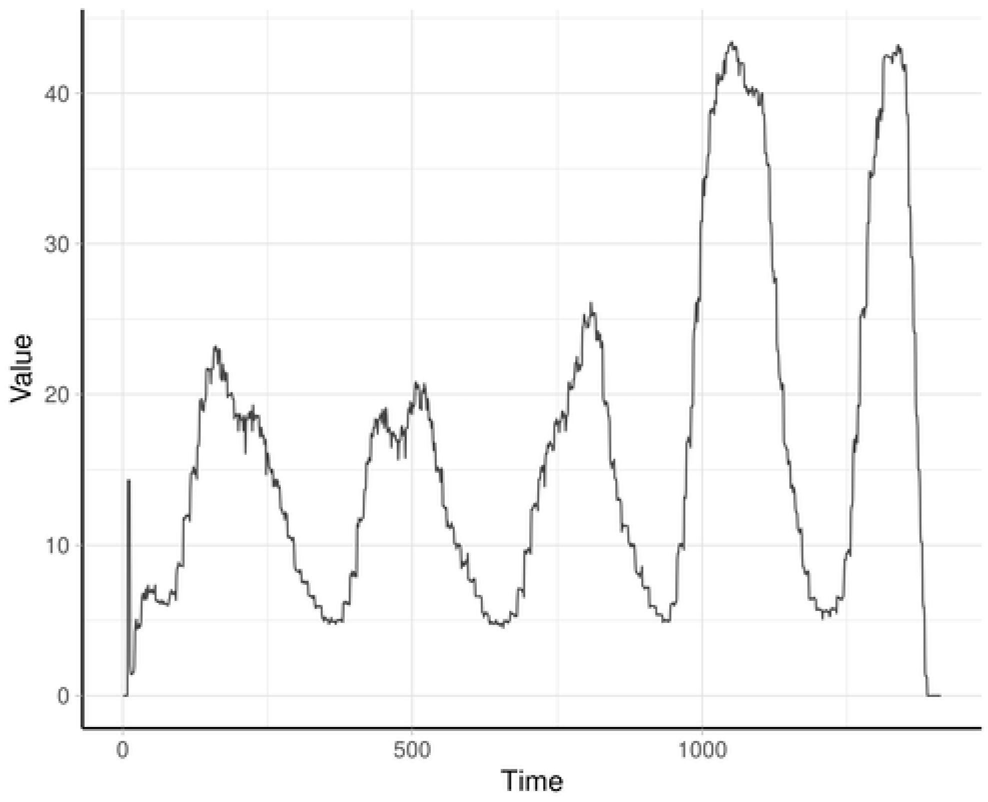


Chart 2. Training Dataset

The y-axis values are averaged across 10 replicas and not less because the y-values would be smaller in another case. This happens because the avg() function takes a measurement and then divides it by the number of the replicas. So, more replicas mean smaller values in this specific metric. This way it is easier for the Machine Learning algorithms to predict the upcoming traffic as there are no major changes in the value of this metric.

3.3.4 Algorithms

3.3.4.1 Arima

As mentioned before, a representative algorithm of Classical Machine Learning time series models, is Arima [19]. Arima is an acronym for “AutoRegressive Integrated Moving Average” and a type of model known as Box-Jenkins method. This acronym is descriptive capturing the key aspects of the model itself. Briefly, they are:

- **AR:** Autoregression. A model that uses the dependent relationship between an observation and some number of lagged observations.
- **I:** Integrated. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to

make the time series stationary. A stationary time series is one whose properties do not depend on the time at which the series is observed. So, time series with trends or with seasonality are not stationary.

- **MA:** Moving Average. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Each of these components are specified in the model as a parameter. A standard notation is used of ARIMA(p,q,d) where the parameters are substituted with integer values. These can be defined as:

- **p:** The number of lagged observations included in the model; also known as the lag order.
- **q:** The number of times that the raw observations are differenced; also known as the degree of differencing.
- **d:** The size of the moving average window; also called known as moving average.

A linear regression model is constructed including the specified number and type of terms and the data is prepared by a degree of differencing in order to make it stationary so that trend and seasonality structures to be removed because they affect the regression model negatively.

In order to use the Arima algorithm, the statsmodels[21] library must be installed. It is done with the following command using conda:

```
$ conda install statsmodels
```

After successfully importing Arima, the next step was to check the data for stationarity[22]. Stationarity implies that taking consecutive samples of data with the same size should have identical covariances regardless of the starting point. That’s the reason it is easier to be analyzed and implemented by many algorithms. Below, there are two examples[23] of

[21] https://www.statsmodels.org/stable/index.html
[22] https://otexts.com/fpp2/stationarity.html
[23] https://towardsdatascience.com/stationarity-in-time-series-analysis-90c94f27322 (Figure 1)

non-stationary and stationary times series data with the first one with a trend headed downwards and the second with no trend at all.

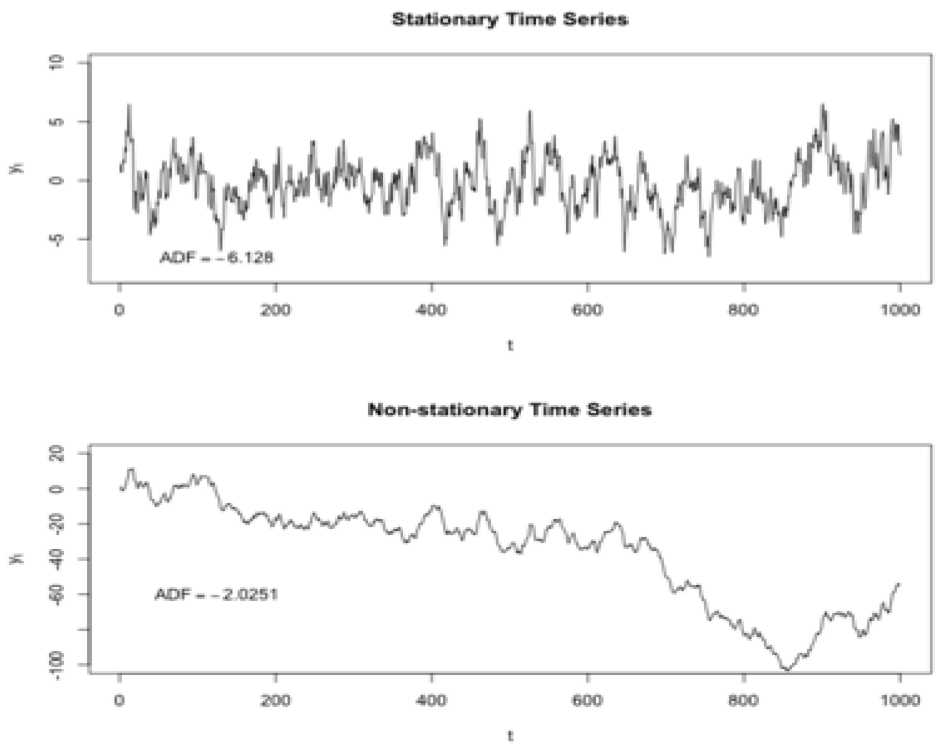
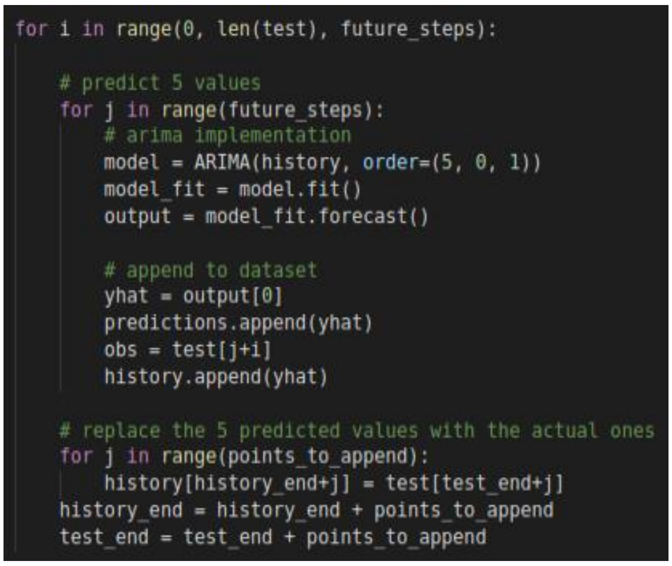


Figure 36. Stationary vs Non-Stationary examples

So, it can be easily decided that the input data are stationary data checking the second chart. The next step is to tune the hyperparameters of the model (p,q,d) that best fit the dataset provided. This is a simple process that just test every possible combination and check the Mean Average Error[24] (MAE) of the predictions. MAE informs about how close the regression line is to a set of points. It does this by taking the distances from the points to the regression line and squaring them. The predictions are 5-step, and the dataset was split into test and train. This approach is very close to the experiment that this diploma thesis focuses in order to find the optimal Arima regression for this case. Inspecting the results, an Arima model with hyperparameters: p=5, q=0, d=1 was selected.

[24] https://en.wikipedia.org/wiki/Mean_absolute_error

After tuning the model, everything is ready for the predictions. How the predictions are being made is descripted above in 3.3.2 section. Here is a code snippet of the implemented Arima algorithm:



```
for i in range(0, len(test), future_steps):

    # predict 5 values
    for j in range(future_steps):
        # arima implementation
        model = ARIMA(history, order=(5, 0, 1))
        model_fit = model.fit()
        output = model_fit.forecast()

        # append to dataset
        yhat = output[0]
        predictions.append(yhat)
        obs = test[j+i]
        history.append(yhat)

    # replace the 5 predicted values with the actual ones
    for j in range(points_to_append):
        history[history_end+j] = test[test_end+j]
    history_end = history_end + points_to_append
    test_end = test_end + points_to_append
```

Figure 37. Arima code snippet

At first, Arima is implemented and fitted to the input data (history variable). Then, the method .forecast is called to make a single prediction. This prediction is appended to the dataset so that it will be used as input. This process repeats 5 times and then after these 5 forecasts are appended to the predictions array, they get replaced with the actual values. Also, in order to evaluate the model's accuracy and to have an early point of view on how it would perform in the live scenario that this thesis' custom system will run, the above scenario was used. The dataset was split into train and test, with test containing the last 350 points out of the 1412 of the input dataset. The model was trained 1062 points and made 5-step predictions of the last 350 points. This is the chart comparing this forecast to the actual values:

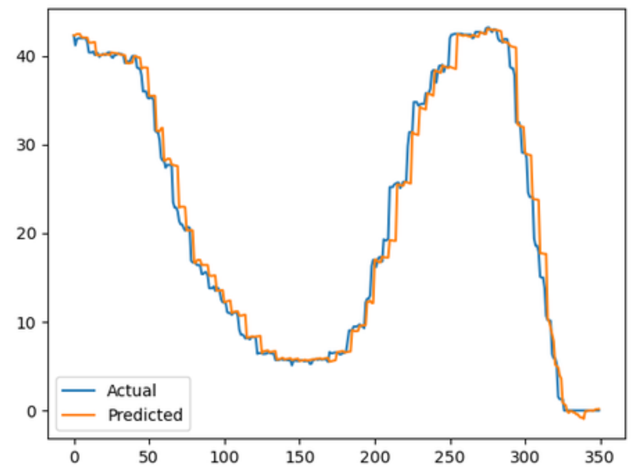


Chart 3. Arima predictions

As it can be observed, Arima's prediction are very good and accurate as the Mean Average Error of this regression is just 1.036.

3.3.4.2 XGBoost

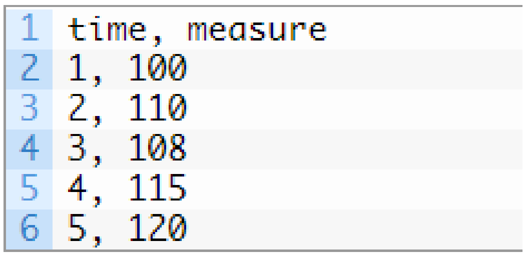
XGBoost [20] is an open-source software library and stands for eXtreme Gradient Boosting. It is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting model. This model is an additive one, where trees are grown in sequential manner and converts weak learners into strong learners by adding weights to the weak learners and reduce weights of the strong ones. So each tree learns from the previous tree grown. Also, XGBoost uses a more regularized model formalization to control over-fitting which gives better performance and is engineered to exploit every bit of memory and hardware resources for tree boosting algorithms. XGBoost is very popular nowadays because it stands out for its computational speed and model performance. It is also installed with the conda command like below:

```
$  conda install xgboost
```

Unlike Arima, the input data must be prepared accordingly for XGBoost. The data must be restructured to look like a supervised learning problem[25]. Supervised learning is an

[25] https://machinelearningmastery.com/time-series-forecasting-supervised-learning/

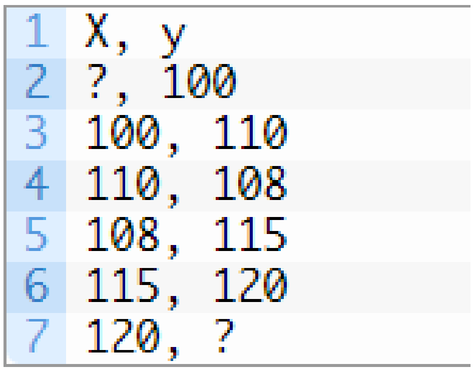
approach to Machine Learning where the machine learns from labeled data. By feeding the learner with examples together with the true labels for those examples, the machine learns a mapping from input to output. So, samples that have not seen before by the learner are fed to the model and a prediction is made based on the mapping learned. Given a sequence of numbers for a time series dataset, the previous time steps can be used as input variables and the next time step as output variables. Suppose a time series as follows:



```
time, measure
1, 100
2, 110
3, 108
4, 115
5, 120
```

Figure 38. Time series dataset format

The data must be reorganized as descripted above so that they look like this:

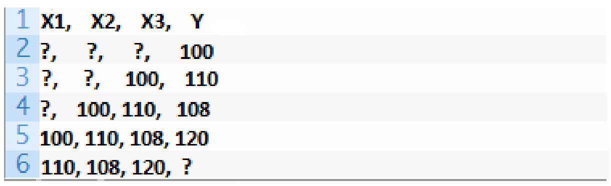


```
X, y
?, 100
100, 110
110, 108
108, 115
115, 120
120, ?
```

Figure 39. Time series supervised learning format

X value is taken as input and y as output. Also, it can be observed that the time column is dropped and some rows of data are unusable for training a model, such as the first and the last. This representation is called a sliding window, as the window of inputs and expected outputs is shifted forward through time to create new samples for a supervised learning model.

A simple extension can be made here. In this specific case, the algorithm takes one point as input and produces one point as output which is not the best solution where there's a multistep forecast, like in this thesis. So, in order to make more accurate predictions a good idea is to use more points as input. But there is a problem here. More points would definitely help increase the model's accuracy but it doesn't mean the more points to take as input the better. By testing the algorithm with such points ranging from 1 to 90 to use as input and taking into consideration the Mean Average Error, 50 points was found to be the optimal value for the points to take as input. In order to understand the input format of the data an example of 3 is shown below.

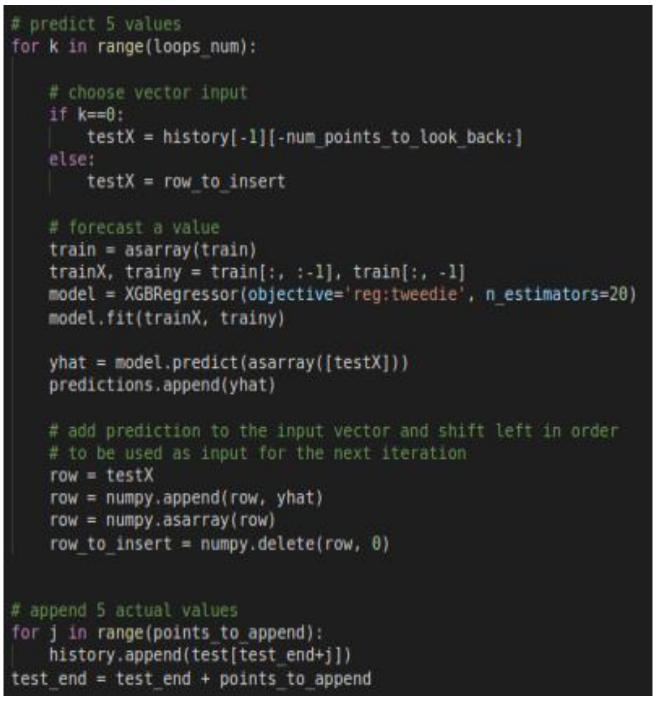


```
X1,   X2,   X3,   Y
?,    ?,    ?,    100
?,    ?,    100,  110
?,    100, 110,   108
100, 110, 108, 120
110, 108, 120,  ?
```

Figure 40. 3 points supervised learning example

The only thing that changes between this example and the input format of XGBoost is that in XGBoost format the X variables are expanded until X50 and not X3. As a result, the first and the last 50 values of the dataset are being discarded.

Then, two hyperparameters values had to be chosen: objective and n_estimators. Objective specifies the learning task and the corresponding learning objective to be used. A wide variety of objectives were tested such as count:poisson, reg:gamma, reg:squarederror with reg:tweedie being the best one for this case, due to the lowest Mean Average Error. As far as the n_estimators variable, it represents the number of gradient boosted trees. Changing this variable helped choose a value of 20 for the XGBoostRegressor function. Below, is the code that tests the XGBoost algorithm:



```
# predict 5 values
for k in range(loops_num):

    # choose vector input
    if k==0:
        testX = history[-1][-num_points_to_look_back:]
    else:
        testX = row_to_insert

    # forecast a value
    train = asarray(train)
    trainX, trainy = train[:, :-1], train[:, -1]
    model = XGBRegressor(objective='reg:tweedie', n_estimators=20)
    model.fit(trainX, trainy)

    yhat = model.predict(asarray([testX]))
    predictions.append(yhat)

    # add prediction to the input vector and shift left in order
    # to be used as input for the next iteration
    row = testX
    row = numpy.append(row, yhat)
    row = numpy.asarray(row)
    row_to_insert = numpy.delete(row, 0)


# append 5 actual values
for j in range(points_to_append):
    history.append(test[test_end+j])
test_end = test_end + points_to_append
```

Figure 41. XGBoost code snippet

After selecting the appropriate input vector for the algorithm, the model is fitted to the data and a prediction is made. Then, this prediction is added to the input vector which is shifted left in order to be used as input for the second prediction. A slightly different approach was used here. Instead of appending the predictions to the dataset and then replacing them accordingly, this XGBoost implementation only appended the actual values at the end of its 5-step forecast but using the previous predictions as inputs. So, after 5 iterations, the actual values were appended to the dataset every time.
The same scenario as in the Arima was used in order to test its accuracy. The dataset was split into train and test, with test containing the last 350 points. The following image shows the XGBoost performance with MAE= 2.72.

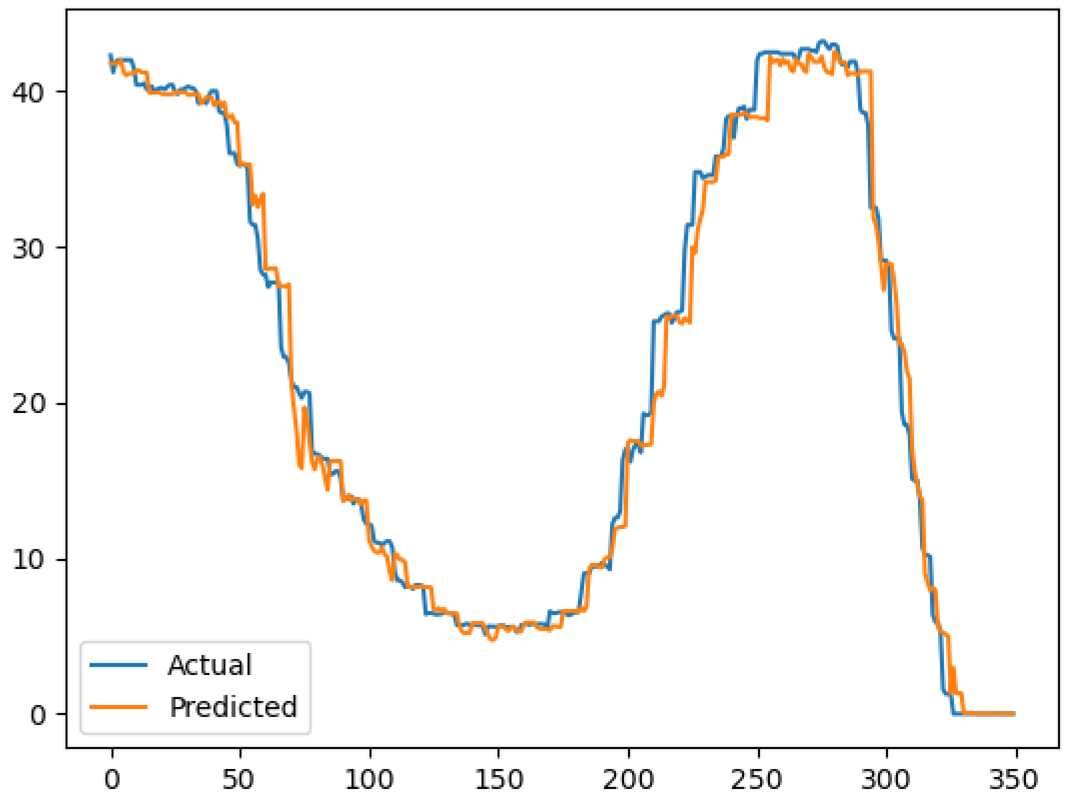


Chart 4. XGBoost predictions

3.3.4.3 LSTM

The last Machine Learning time series algorithm that this diploma thesis use is LSTM [21]. The LSTM or Long Short-Term Memory network is a type of recurrent neural network used in deep learning. Recurrent networks have an internal state that can represent context information and keep track of the past inputs for an amount of time that is not fixed but rather depends on its weights and on the input data. LSTMs are explicitly designed to be able to connect previous information to the present task in such a way that the long-term dependency problem is being solved. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn. Instead of neurons, LSTMs have memory blocks that are connected through layers. A block has components that make it smarter than a classical neuron and a memory for recent sequences. A block contains gates that manage the state and the output and operates upon an input sequence. The different types of gates are being shown in the following image[26]:

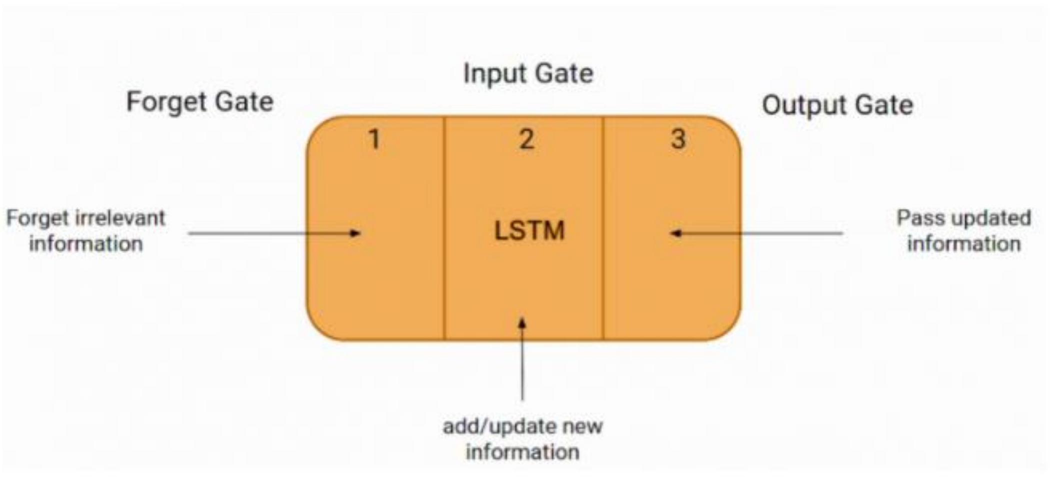


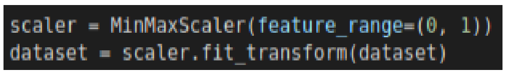
Figure 42. LSTM block architecture

There are three types of gates within a unit:

1. **Forget Gate:** conditionally decides what information to throw away from the block.
2. **Input Gate:** conditionally decides which values from the input to update the memory state.
3. **Output Gate:** conditionally decides what to output based on input and the memory of the block.

So, in order to use the LSTM network, tensorflow need to be installed in order to use its library keras. After installing tensorflow (section 3.2.7), the next step is to prepare the data. It is common in neural networks to scale the input data before training, using MinMaxScaler. MinMaxScaler is scaling the independent variables so that they lie in the range of 0 and 1. This is important because few variable values might be in thousands and few might be in very small ranges. So, to handle such cases scaling was applied to the training dataset with the additional code displayed underneath:

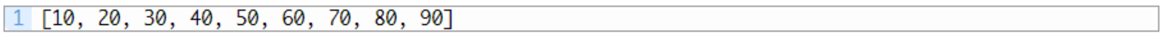
[26] https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/



```
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

Figure 43. MinMaxScaler code snippet

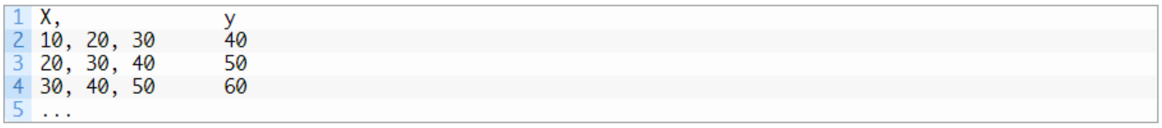
Also, the LSTM model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the LSTM can learn.
For example, for a given univariate sequence:



```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Figure 44. Univariate sequence example

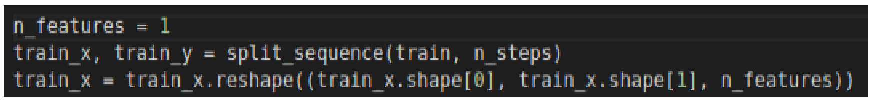
The sequence can be divided into multiple input/output patterns called samples, where in this given example, three time-steps are used as input and one time step is used as output as seen below:



```
X,              y
10, 20, 30      40
20, 30, 40      50
30, 40, 50      60
...
```

Figure 45. 3-step modification example

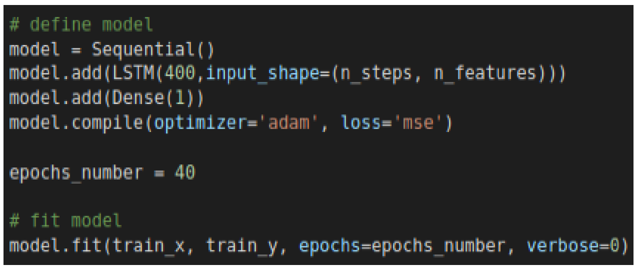
Just like XGBoost, this approach can be modified to take more points as input in order to produce a single prediction. 40 input points was found to be the optimal number of input points for this particular case. A combination of input points has been tested, checking the MAE and the time consumed to train and fit the model. So, the input has a format like this: [samples, timesteps]. But LSTM network expects a 3-dimensional input which has to be formatted to: [samples, timesteps, features]. Since the preferred output of this algorithm is one (1 prediction at a time), the input must have this format: [samples, timesteps, 1] which is being executed with the code below:



```
n_features = 1
train_x, train_y = split_sequence(train, n_steps)
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
```

Figure 46. LSTM input format

In this diploma thesis, Vanilla LSTM is used. A Vanilla LSTM model has a single hidden layer of LSTM units, and an output layer used to make a prediction and it is defined as below:



```
# define model
model = Sequential()
model.add(LSTM(400,input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

epochs_number = 40

# fit model
model.fit(train_x, train_y, epochs=epochs_number, verbose=0)
```

Figure 47. LSTM model

The hyperparameters of this network are:

1. **LSTM Units**: refers to the number of units of LSTM. A higher number of units indicates a more powerful network with raising training time and very possible to overfit the data. A lower number of units leads to a bad implementation of the algorithm with low accuracy score. 400 LSTM units was found to be the optimal number.
2. **Number of epochs:** is the number of times that the learning algorithm will work though the entire training dataset. Usually, training a neural network takes more than a few epochs with this LSTM network using 40.
3. **Loss function:** is the function that simply calculates the error. It reduces all the various good and bad aspects of a complex system down to a single number which allows candidate solutions to be ranked and compared. This thesis uses the “MSE” loss function which calculates the loss based on the difference between the model’s predictions and the ground truth, squaring it and averaging it across the whole dataset.
4. **Optimizer:** is an algorithm or a method used to change the attributes of the neural network such as weights and learning rate to reduce the

losses. Optimizers helps to get results faster. The “adam” optimizer [22] was used because its effectiveness.

In order to check if the LSTM network overfits, learning curves was used[27]. A learning curve is a plot of model learning performance over experience or time. Learning curves are a widely used as a diagnostic tool in Machine Learning for algorithms that learn from a training dataset incrementally. The model can be evaluated on the training dataset and on a holdout validation dataset after each update during training and plots of the measured performance are created to show learning curves. Reviewing these curves helps detect if the algorithm overfits, underfits or has a good fit over the validation data. A good fit is the goal of the learning algorithm and exists between an overfit and underfit model. It is identified by a training and validation loss and specifically, if both of them decrease to a point of stability and the validation loss has a small gap with the training loss. Underneath, the plot of these losses is demonstrated indicating that with the specific hyperparameters, the model has a good fit over the data with a training dataset consisting of the first 300 points and a validation dataset containing 300 to 550 points.

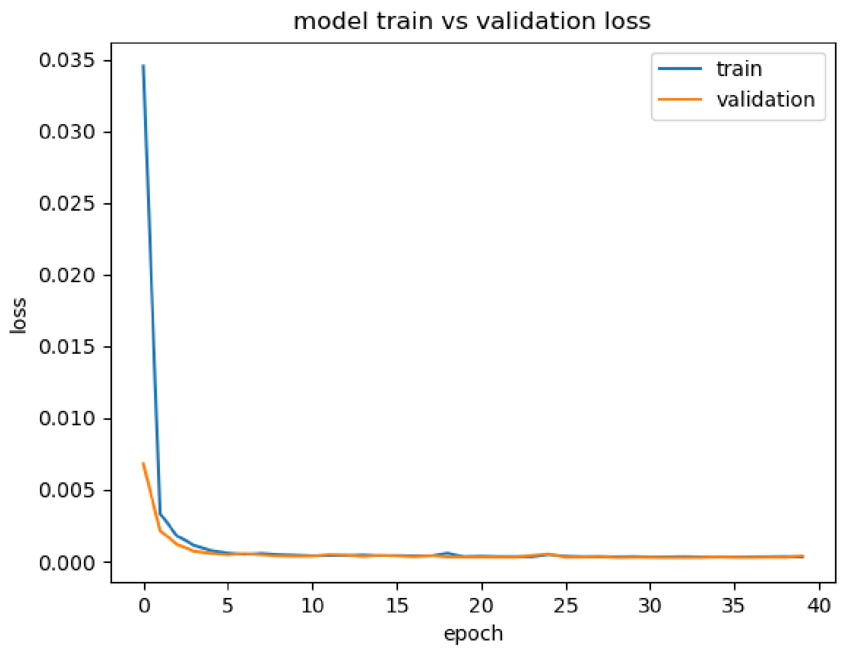


Chart 5. LSTM learning curves

[27] https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/

In order to make forecasts and validate and visualize the results as done with Arima and XGBoost , the methodology explained in 3.3.2 section was used. The input dataset was split into train (first 1062 points) and test (last 350 points) using 5-step forecasts. The following chart shows the LSTM network performance over the test dataset with MAE=3.22.

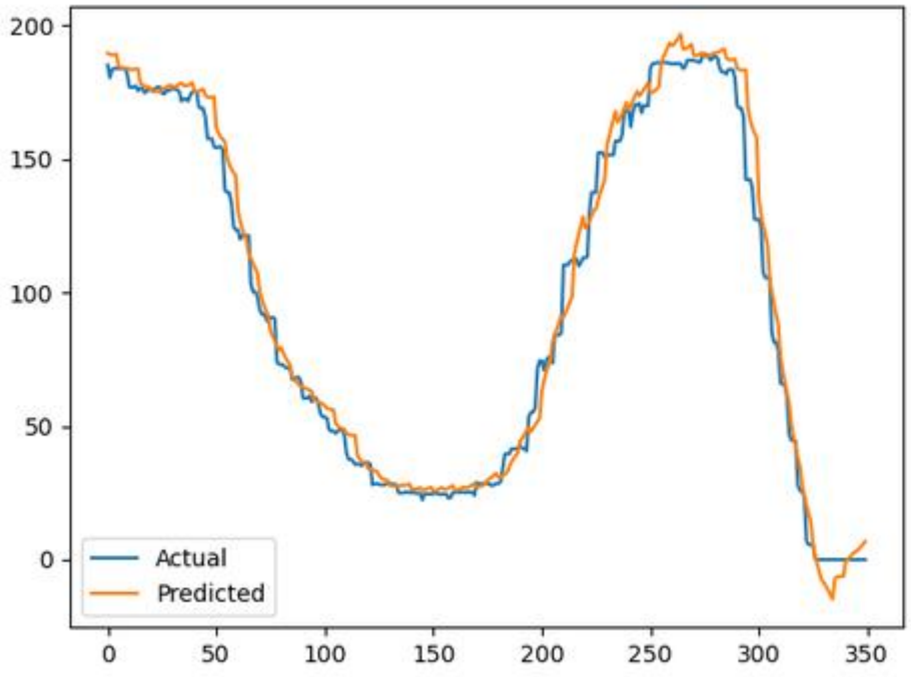


Figure 48. LSTM predictions

3.4 Scale Policy

The scale policy that this diploma thesis used, is based on the Horizontal Pod Autoscaler scaling algorithm and is shown below:

```
desiredReplicas = ceil[currentReplicas * ( futureMetricValue / desiredMetricValue )]
```

Figure 49. Custom scale policy

The only difference between the HPA’s scale policy and this thesis policy is that the currentMetricValue variable has been replaced with the futureMetricValue variable. The futureMetricValue is the prediction that has been produced from the Machine Learning part and has to do with the upcoming traffic. More specifically, it is the average rate of cumulative packets received across all replicas for the 10 following minutes. So, having this value forecasted, the deployment can scale before the traffic arrives making better use of resources.

3.5 System Architecture

Last but not least, the architecture of the environment must be analyzed. It is a Python script that runs at the same time as the traffic that the php-apache server receives predicting the next values, scaling, fetching the actual values and appending them to dataset. Here an image is being displayed explaining the logic behind.

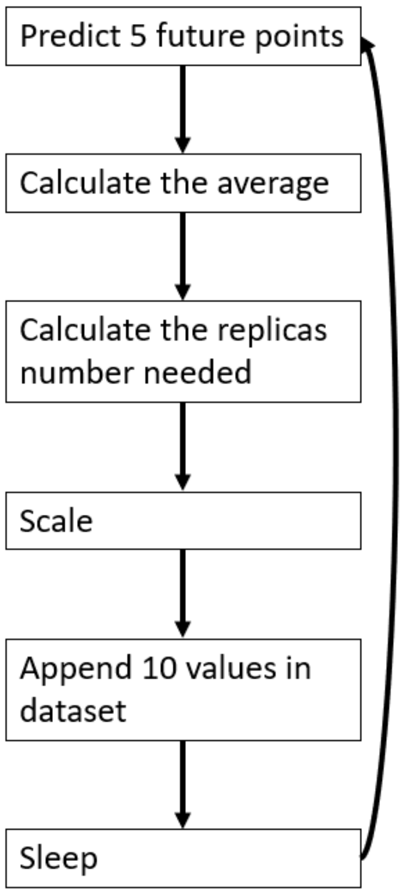

Predict 5 future points
Calculate the average
Calculate the replicas number needed
Scale
Append 10 values in dataset
Sleep

Figure 50. Thesis' system architecture

The architecture of the system that was developed for this diploma thesis can be divided into 7 sections:

1. **Predict 5 future points**: this is the part that a 5-step forecast is being made with the help of the Machine Learning time series algorithms. The value that is being predicted is the average rate of cumulative packets received across all replicas. In order to make comparisons for each algorithm, 3 python scripts were created one for each algorithm, with each algorithm used separately in each experiment. So, this is the only part that differs in these 3 python scripts.
2. **Calculate the average**: the average of these 5 values is being calculated in order to be used as input for the formula that calculates the replicas number. It was preferred to calculate the average of the next 5 values compared to simply predicting the next one because with the second approach there would be a lot of ping-pong effects. A better use of resources is being made when the deployment is not scaled in traffic spikes as it is better to strain for a few seconds to carry out the workload than to scale the whole deployment. That’s why the average of the next 5 values is used.
3. **Calculate the replicas needed:** the futureMetricValue is being replaced from this formula (Figure 10) with the average value of the 5 predictions in order to get the replicas needed to handle the upcoming traffic in the best way possible.
4. **Scale (manually)**: after getting the replicas number needed for the upcoming minutes, the deployment must be scaled, if the predicted number of replicas is different from the current number of replicas.
5. **Append 10 values in dataset**: this part consists of sampling 10 points of this metric (average rate of cumulative packets received across all replicas) every 16 seconds, with the help of Prometheus Adapter. 16 seconds where chosen in order to see changes in its value as Prometheus has a refresh rate of 15 seconds. So, in 160 seconds 10 samples are taken 16 seconds apart the one from another.

6. **Sleep**: this is the final step and is just a suspension of the execution of the program for a given number of seconds. This number is simply the subtraction of the time spent from step one to step 6 of 3 minutes.

This is the main structure of the mechanism that this diploma thesis implements, that scales in real time based on the predicted upcoming traffic without the help of Horizontal Pod Autoscaler.

Chapter 4: Experiments and Comparisons

4.1 Chapter Introduction

This chapter consists of some experiments that took place in order to test the mechanism's ,that this diploma thesis implements, functionality. Also, it is found that this mechanism scale more efficiently as the same number of replicas is being deployed earlier compared to the Horizontal Pod Autoscaler making better use of resources. The experiments that were performed are being analyzed and the HPA along with the implementationσ of this mechanism for each Machine Learning algorithm are being compared.

4.2 Experiments Description

In order to test the mechanism that this thesis implement, some experiments need to be done. The scenario explained above, is being executed 4 times, 3 for each Machine Learning time series algorithm that had been used (Arima, XGBoost, LSTM) and one for the Horizontal Pod Autoscaler. This mechanism runs at the same time as the generation of traffic, just like the Horizontal Pod Autoscaler.
Two Kubernetes Objects were being used in order to generate traffic to the php-apache server: the Php-Apache Deployment and the Ubuntu Pod, that had been analyzed in 3.2.2 section. This deployment defines the php-apache server and this Pod is used in order to distribute the traffic equally to all replicas of the server. So, the requests that the apache server receives, are all being sent from the Ubuntu Pod. After transitioning to the

container that this Pod utilize, a Python script called make_calls.py is being used that simply make requests (“curl” command) to the server every 3 minutes. The number of requests is being retrieved from the high.csv for one case and the medium.csv from the other case (3.3.3 section).

As far as the Machine Learning, a few more things need to be specified. The training of the models was done with the dataset that was collected from the traffic generated by the high.csv dataset and represent the custom metric according to which the scale that was done and is displayed in Chart 2. As mentioned above, this metric was averaged over 10 replicas so that there would not be some major ups and downs that will make it for the Machine Learning algorithms hard to train and predict.

Since the traffic generated every 3 minutes, so did the whole procedure that is being presented on the 3.5 section had to last for each iteration. In these 3 minutes, the first 20 seconds are for training the model, making 5-step prediction and scaling appropriately. The rest 160 seconds are for sampling every 16 seconds for 10 times.

So, 2 terminals were needed, one for the generation of traffic (Ubuntu Pod) and one for running the mechanism that this thesis utilize. These Python scripts run simultaneously, with the first one making requests to the php-apache server and the second one producing predictions, scaling, sampling and both of them operating every 3 minutes.

4.3 Experimental Results

Having the environment ready and the experiment rolling, the performance charts are yet to be presented. In order to demonstrate and evaluate the mechanism that this diploma thesis is all about, some diagrams were made that show how well each Machine Learning algorithm performed in predicting the future and therefore, scaling the deployment. These diagrams describe the change in the replicas number as a function of time, based on the traffic received on the php-apache server.

4.3.1 Same dataset for train and test

In this section, high.csv was used for both training and testing. As mentioned in 3.3.3 section, the algorithms are trained on the traffic that this dataset produced which in this case, is also used for testing. Underneath each Machine Learning algorithm's performance is being presented based on the change of the replicas number over time.

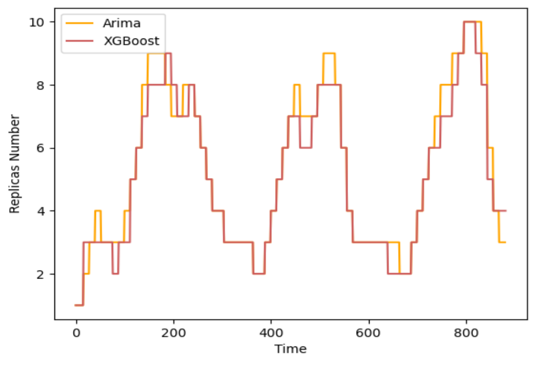


Chart 6. Arima vs XGBoost high.csv dataset

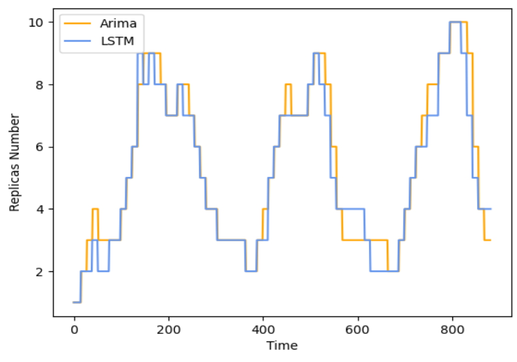


Chart 7. Arima vs LSTM high.csv dataset

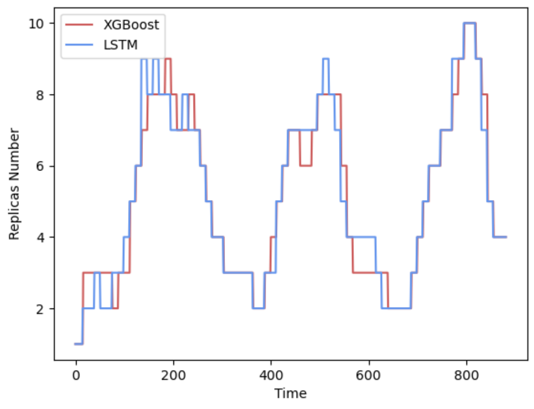


Chart 8. XGBoost vs LSTM high.csv dataset

In these diagrams, the Arima regression is with the orange line, the XGBoost with the red and the LSTM with the blue. This experiment lasted 220,5 minutes as there are 882 points in the x-axes with each one representing a 15 seconds period. While there are differences in replicas number for each algorithm, it can be seen that more or less, the overall results are similar. Furthermore, differentiating the predictions of each algorithm from the others, some of these differences are due to the factor of luck that governs communication with the php-apache server, as there are thousands of requests that arriving in the server within a short time.

4.3.2 Different dataset for train and test

In this section, high.csv was used for training and medium.csv for testing. As mentioned in 3.3.3 section, the algorithms are trained on the traffic that high.csv produced and tested on the traffic that medium.csv made. Below each Machine Learning algorithm's performance is being presented based on the change on the replicas number.

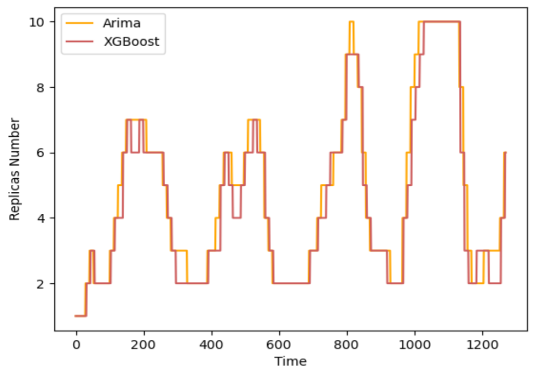


Chart 9. Arima vs XGBoost different dataset

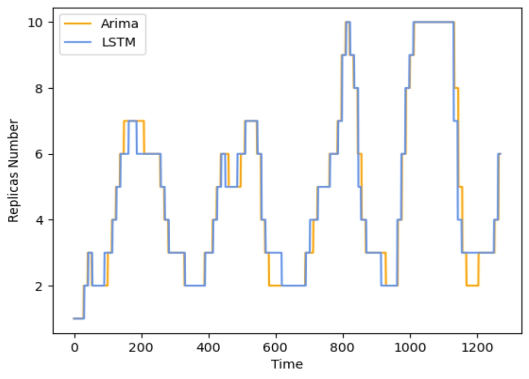


Chart 10. Arima vs LSTM different dataset

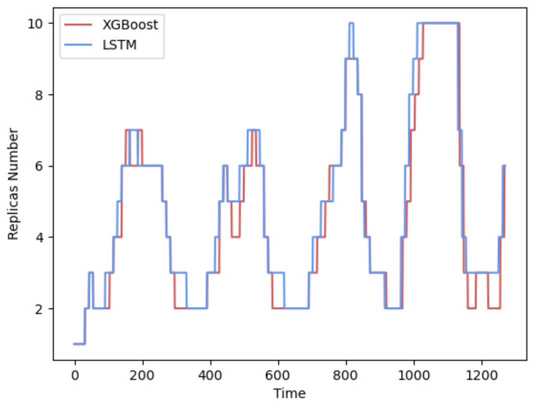


Chart 11. XGBoost vs LSTM different dataset

The Arima regression is with the orange line, the XGBoost with the red and the LSTM with the blue. This experiment lasted for 317,5 minutes as there are 1270 x-axes points. Comparing the duration of this experiment with the one described in 4.2.2 section, this experiment lasted almost 100 minutes longer. The values of medium.csv are way smaller than in high.csv, making it less time consuming to fit the Machine Learning model in order to make predictions. Also, the changes in the replicas number by every algorithm are again not far from each other.

4.4 Comparisons with Autoscaler

Having every Machine Learning algorithm's performance diagram, it is time to compare each of them with the Horizontal Pod Autoscaler in order to test whether the system that this diploma thesis implements is indeed making better use of resources or not.

4.4.1 Same dataset for train and test

In this section, high.csv was used for both training and testing. As mentioned in 3.3.3 section, the algorithms are trained on the traffic that this dataset produced which is in this case used for testing too. Below, there are displayed 3 diagrams that compare the change of replicas number between the HPA and each Machine Learning algorithm using the same dataset:

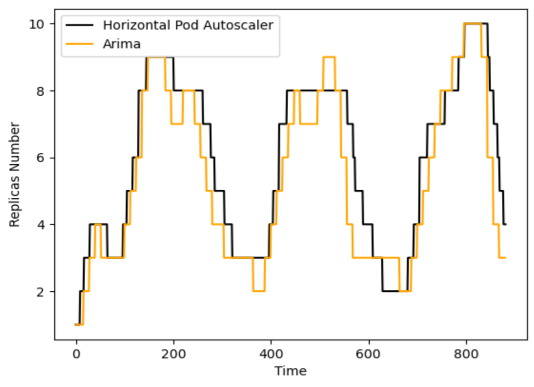


Chart 12. HPA vs Arima same dataset

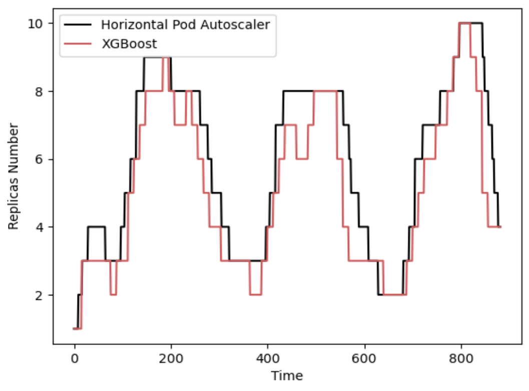


Chart 13. HPA vs XGBoost same dataset

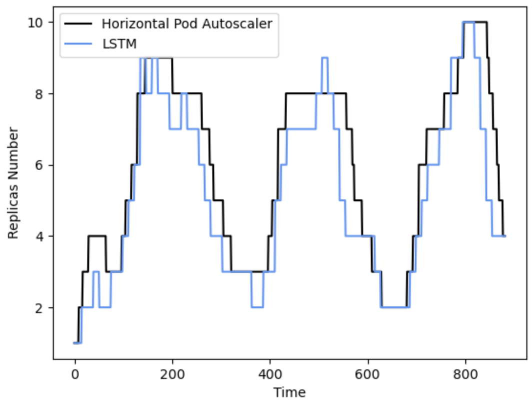


Chart 14. HPA vs LSTM same dataset

According to these graphs, it can be observed that in the case of same dataset for training and testing, this mechanism gets better results than the Horizontal Pod Autoscaler. A better use of resources is achieved as the scaling is done earlier using this mechanism. More specifically, the same number of replicas is being scaled several seconds on many occasions before the HPA would do.

4.4.2 Different dataset for train and test

In this section, high.csv was used for training and medium.csv for testing. As mentioned in 3.3.3 section, the algorithms are trained on the traffic that high.csv produced and tested on the traffic that medium.csv brought out. Underneath, can be found 3 diagrams that make comparisons between each Machine Learning algorithm used and the Horizontal Pod Autoscaler using the scenario descripted above:

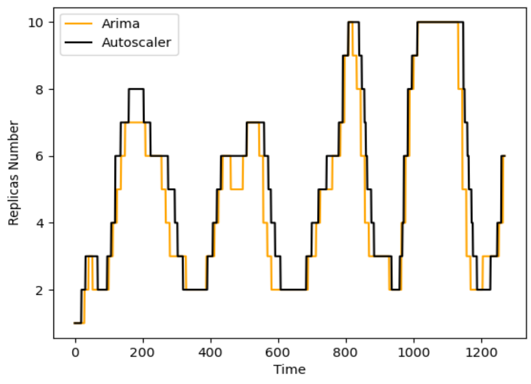


Chart 15. HPA vs Arima different dataset

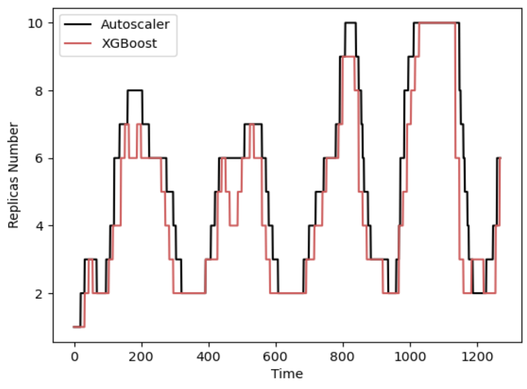


Chart 16. HPA vs XGBoost different dataset

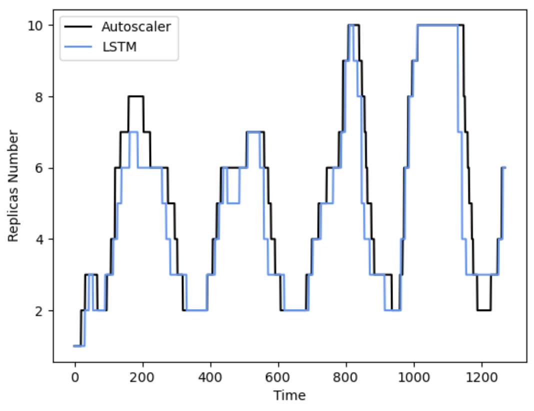


Chart 17. HPA vs LSTM different dataset

And in this scenario, the mechanism that this diploma thesis implements is performing better than the Horizontal Pod Autoscaler. The scales are done earlier than the HPA which shows that better use of resources is being achieved. It predicts the upcoming traffic in order to scale in time and when the traffic arrives everything is scaled as they should do, so that there are no resources that overwork or being allocated and not serving the traffic.

Chapter 5: Conclusion and Future Work

5.1 Chapter Introduction

In this chapter, the conclusion and the future work are being presented. The main idea of this diploma thesis that had been discussed so far is being summarized and some suggestions considering the future work are being made.

5.2 Conclusion

All in all, the main idea of this thesis is to implement a mechanism that would scale a deployment based on the upcoming traffic on the server with the help of Machine Learning algorithms. This results in better resource utilization as there are no resources that are allocated and not used or overtrying to carry out the tasks needed. Energy is also saved as without the autoscaling mechanisms like the one that this diploma thesis implements, all of the resources must be allocated in order to be able to serve the upcoming tasks which would be very energy-inefficient. The goal was achieved based on the experiments that took place above resulting that the replicas number can be well-predicted, showing that in dynamic scaling and in mechanisms such as the Horizontal Pod Autoscaler there is enough room for improvement.

5.3 Future Work

Although the goal of this diploma thesis is considered to be accomplished, there are some improvements that could be made in order to achieve better results. At first, as mentioned above (section 3.5), for every iteration that lasts 3 minutes, up to 20 seconds were consumed by the Machine Learning algorithms for training and fitting the model. But as the training dataset increases (10 points are appended in each iteration), the time it takes for the algorithms to train also increases. If this proccess exceeds 20 seconds, the corresponding Python script crashes and the mechanism that this diploma thesis implements stops working. That's why this mechanism cannot run endlessly. Therefore, a good solution to this problem that will make this custom mechanism to run without any time limit is to train and fit the model in a finite dataset. This dataset can represent a definable number of points that for every 10 new points, the 10 oldest points have to get discarded in order to keep a steady dataset length.

Also, a new experiment can be created that will further test and evaluate the importance of having as best scaling as possible, with the help of Machine Learning as this diploma thesis implements. Instead of having a php-apache server that simply responds with a string and inspecting the replicas number, an application could be also written that will have a better performance when the appropriate scaling is done. A nice example of an application like that would be a video-streaming one that would generate traffic that with poor scaling decisions the video will be laggy and with great scaling decisions would be smooth.

Finally, different Machine Learning time series forecasting algorithms can be tested and implemented, beyond Arima, XGBoost and LSTM. With the appropriate tuning, predictions with greater accuracy can be made resulting in better scaling decisions.

Bibliography

[1] Wikipedia. (2021). *Cloud Computing.*

Retrieved from: https://en.wikipedia.org/wiki/Cloud_computing

[2] Wikipedia. (2021). *Machine Learning*.

Retrieved from: https://en.wikipedia.org/wiki/Machine_learning

[3] NITLAB. (2015). *Network Implementation Laboratory*.

Retrieved from: https://nitlab.inf.uth.gr/NITlab/nitos

[4] Docker. (2013). *Docker*.

Retrieved from: https://www.docker.com/

[5] Docker. (2013). *Containers.*

Retrieved from: https://www.docker.com/resources/what-container

[6] Kubernetes. (2014). *Production-Grade Container Orchestration*.

Retrieved from: https://kubernetes.io/

[7] Avi Networks. (2021). *Kubernetes Architecture.*

Retrieved from: https://avinetworks.com/glossary/kubernetes-architecture/

[8] Kubernetes. (2021). *Horizontal Pod Autoscaler.*

Retrieved from: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[9] Prometheus. (2016). *From metrics to insight.*

Retrieved from: https://prometheus.io/

[10] Kubernetes SIGS. (2021). *prometheus-adapter.*

Retrieved from: https://github.com/kubernetes-sigs/prometheus-adapter

[11] Grafana Labs. (2013). *Dashboard anything. Observe everything*.

Retrieved from: https://grafana.com/grafana/

[12] Wikipedia. (2021). *Machine learning.*

Retrieved from: https://en.wikipedia.org/wiki/Machine_learning

[13] Matthew Mayo, KDNuggets. (2021). *The 7 steps of Machine learning.*

Retrieved from: https://www.kdnuggets.com/2018/05/general-approaches-machine-learning-process.html

[14] Tensorflow. (2015). *An end-to-end open source machine learning platform.*

Retrieved from: https://www.tensorflow.org/

[15] Docker. (2011). *Build and Ship any Application Anywhere.*

Retrieved from: https://hub.docker.com/

[16] Nguyen, T.-T.; Yeom, Y.-J.; Kim, T.; Park, D.-H.; Kim, S. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. *Sensors* **2020**, *20*, 4621. https://doi.org/10.3390/s20164621.

[17] Barlacchi, G., De Nadai, M., Larcher, R. *et al.* A multi-source dataset of urban life in the city of Milan and the Province of Trentino. *Sci Data* **2,** 150055 (2015). https://doi.org/10.1038/sdata.2015.55.

[18] Harvard Dataverse. (2021). *Telecommunications – SMS, Call, Internet – MI.*

Retrieved from:

https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/EGZHFV

[19] Adam Hayes, Investopedia. (2021). *Autoregressive Integrated Moving Average (ARIMA).*

Retrieved from:

https://www.investopedia.com/terms/a/autoregressive-integrated-moving-average-arima.asp

[20] Mohit Sharma, Medium. (2021). *Gentle Introduction to XGBoost Library.*

Retrieved from: https://medium.com/@imoisharma18/gentle-introduction-of-xgboost-library-2b1ac2669680

[21] Wikipedia. (2021). *Long Short-Term Memory.*

Retrieved from: https://en.wikipedia.org/wiki/Long_short-term_memory

[22] D. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,”International Conference on Learning Representations, 12 2014