

Kubernetes Performance Analysis on Different Architectures

by

MD Badsha Faysal

18201136

MD Sheikh Amin

18341004

Bushra Tabassum

22341076

Tamim Raiyan Khan

18341003

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
September 2022

© 2022. BRAC University
All rights reserved.

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:



MD Badsha Faysal
18201136



MD Sheikh Amin
18341004



Bushra Tabassum
223410767



Tamim Raiyan Khan
18341003

Approval

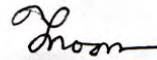
The thesis/project titled “Kubernetes Performance Analysis on Different Architectures” submitted by

1. MD Badsha Faysal (18201136)
2. MD Sheikh Amin (18341004)
3. Bushra Tabassum (22341076)
4. Tamim Raiyan Khan (18341003)

Of Summer, 2022 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on September 20, 2022.

Examining Committee:

Supervisor:
(Member)



Ms. Jannatun Noor Mukta
Senior Lecturer
Department of Computer Science and Engineering
BRAC University

Program Coordinator:
(Member)

Dr. Md. Golam Rabiul Alam
Associate Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi
Associate Professor and Chairperson
Department of Computer Science and Engineering
Brac University

Abstract

The web as we know it is continuously evolving and changing its shape rapidly. And every day the rate of new technologies being introduced is also increasing. Kubernetes is an excellent tool for cloud computing. Kubernetes is still in its early days. Developers are changing their deployment strategy to use Kubernetes. Kubernetes is a powerful tool for horizontal scaling. Cloud providers like GCP, AWS, Azure, and Oracle are offering Kubernetes services. They offer both x86 and ARM platforms. But x86 covers most of the market and very few people are currently working on ARM. We want to find out a more efficient and cost-effective implementation of Kubernetes. ARM offers cheap cloud products at a cheaper rate. ARM is energy efficient and ARM is a newer CPU design than traditional x86. We want to compare the Kubernetes performance on x86 vs ARM to study whether moving to ARM is a better option or not.

Keywords: CPU Architecture, Cloud, Kubernetes, Docker, GCP, AWS, Azure, x86, ARM, Oracle

Acknowledgement

Firstly, all praise to the Great Allah for whom our thesis has been completed properly.

Secondly, to our Supervisor Ms. Jannatun Noor Mukta mam, for her support in our work. She guided us throughout our journey. And finally to our parents for their unconditional support.

Table of Contents

Declaration	i
Approval	ii
Abstract	iii
Acknowledgment	iv
Table of Contents	v
List of Figures	viii
List of Tables	ix
Nomenclature	x
1 Introduction	1
1.1 Background and Motivation	1
1.2 Research Problem	2
1.3 Research Objectives	3
1.4 Our Contributions	3
1.5 Overview of the Thesis	4
2 Background	5
2.1 CPU Architecture	5
2.2 x86 vs ARM	6
2.3 ARM Advantages	6
2.4 Bare Metal	7
2.5 Virtualization	8
2.6 Containerization	9
2.7 Docker	10
2.7.1 Docker Client and Server	10
2.7.2 Docker Images	10
2.7.3 Docker Registries	11
2.7.4 Docker Containers	11
2.7.5 Scalability	11
2.7.6 Speed	12
2.7.7 Portability	12
2.7.8 Density	12
2.8 LXC	12

2.9	Kubernetes	12
2.9.1	Control Plane Components	13
2.9.2	Kube-API-Server	14
2.9.3	etcd	14
2.9.4	Kube-Scheduler	14
2.9.5	Kube-Controller Manager	14
2.9.6	Cloud-Controller Manager	15
2.9.7	Node Components	15
2.9.8	Kube-proxy	15
2.9.9	Container Runtime	15
2.9.10	Services	15
3	Related Study	16
3.1	Kubernetes Performance Major Cloud Providers	16
3.2	Kubernetes Auto Scalability	16
3.3	Different Types of Kubernetes	16
3.4	Current Situation of CPU Platforms	17
3.5	ARM Support on Major Cloud Providers	17
4	Methodology	18
4.1	Testing Plan	18
4.2	Deploying Clusters	19
4.2.1	x86 Intel Cluster	19
4.2.2	x86 AMD Cluster	19
4.2.3	ARM Ampere Cluster	19
4.2.4	ARM Graviton Cluster	19
4.3	HTTP Tools	20
4.3.1	Creating REST API	21
4.3.2	CPU Information Collection API	22
4.3.3	Containerization	23
4.3.4	Uploading to Image Registry	24
4.4	Sysbench	24
5	Experimental Evaluation	25
5.1	Experimental Settings	25
5.1.1	Connecting with the cluster	25
5.1.2	Deploying pods and services on the clusters	25
5.1.3	Setting up Benchmarking Tool	27
5.1.4	Benchmarking Parameters	27
5.1.5	Collecting CPU information	27
5.2	Experimental Results	27
5.2.1	Express Results	28
5.2.2	Flask Results	30
5.2.3	Gin Results	31
5.2.4	Actix Results	32
5.2.5	I/O Results	33
5.3	Experimental Findings	34
5.3.1	Theoretical Comparative Study	37

6	Discussion	38
6.1	Overview	38
6.2	Limitations	38
7	Conclusion and Future Work	39
7.1	Conclusion	39
7.2	Future Plans	39
	References	42

List of Figures

1.1	Cloud Computing [24]	1
2.1	Bare Metal Server [29]	8
2.2	KVM, XEN [31]	9
2.3	VM vs Container [17]	9
2.4	Docker [25]	10
2.5	Docker Image [30]	11
2.6	Docker Container [23]	11
2.7	Kubernetes [27]	13
2.8	Kubernetes Pods [28]	14
4.1	Testing Model	18
4.2	Fibonacci Pseudo code	21
4.3	Express API	22
4.4	CPU info collection API	23
4.5	Dockerfile	24
5.1	Express API Deploy	26
5.2	Express Throughput	29
5.3	Express Success Rate	29
5.4	Flask Throughput	30
5.5	Flask Success Rate	30
5.6	Gin Throughput	31
5.7	Gin Success Rate	32
5.8	Actix Throughput	32
5.9	Actix Success Rate	33
5.10	I/O Performance	34
5.11	Cost Difference	35
5.12	Express Price to Performance Ratio	35
5.13	Flask Price to Performance Ratio	36
5.14	Gin Price to Performance Ratio	36
5.15	Actix Price to Performance Ratio	37

List of Tables

4.1	Platforms	20
5.1	Testing Client	27
5.2	API Benchmark Results	28
5.3	I/O Results	28
5.4	Comparison	37

Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

API Application Programming Interface

AWS Amazon Web Services

CISC Complex Instruction Set Computer

CPU Central Processing Unit

CRI Container Runtime Interface

CRUD Create, Read, Update and Delete

EKS Elastic Kubernetes Services

GCP Google Cloud Platform

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

LXC Linux Container

RAM Random Access Memory

RISC Reduced Instruction Set Computer

VM Virtual Machine

Chapter 1

Introduction

1.1 Background and Motivation

For the last few years, the progress and innovation in the world of cloud computing have been astonishing. More and more services and technologies are being developed every day by both large corporations and the open-source community. Many services are now available on top of the cloud, such as Software as a Service (SaaS), Infrastructure as a Service (IaaS), Function as a Service (FaaS). Most of the cloud providers are providing automated services that reduce work from the developer's end. Kubernetes is now available as a PaaS.

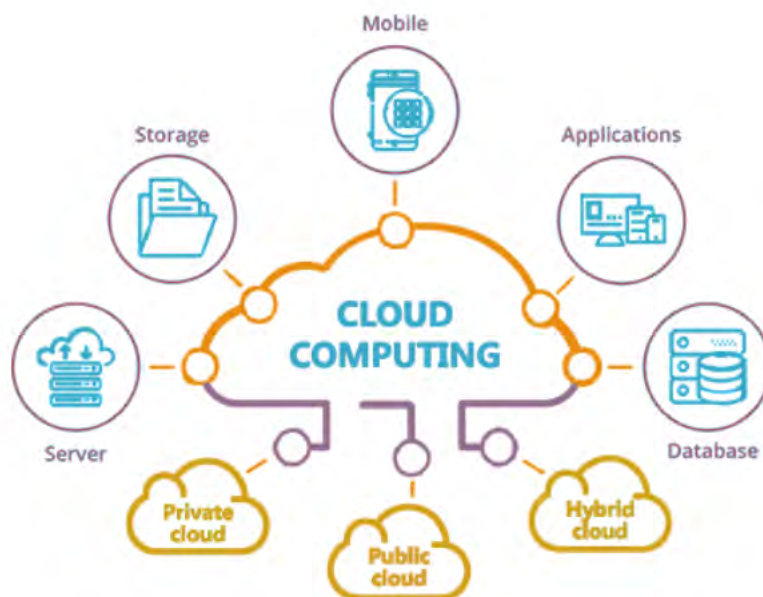


Figure 1.1: Cloud Computing [24]

Containerization is the latest trend in development. Everyone is converting their applications into containers. Docker is the most famous implementation of containerization. There are other containerization technologies such as LXC that are not as famous as Docker [2]. Though it adds extra steps to create an image of the application and some applications require a complex Dockerfile, it saves time if the deployment needs to be repeated on multiple devices. A complex setup of deployment that can be scripted by using docker-compose. Still, multiple containers need

to be load balanced manually and implementing on multiple instances take time. Kubernetes removes this complication, and automates and load balances pods [2]. Most docker images can be used in Kubernetes. In Kubernetes, containers are called pods, and each of the computing instances is called nodes. The node which controls other nodes is called master. In a Kubernetes system, at least one master node and one worker node are needed. But in a small-scale scenario, one master with three worker configurations is expected. In a highly available system, multiple master nodes are load balanced and provide failover. Intel introduced the x86 architecture with the launch of 8086. It was initially launched with 16-bit support in 1978. Then in 1985, it added support for 32-bit and support for 64-bit was added in 2003. It is now the most used architecture in the world. Almost all current personal and server computers are based on x86 platforms. The 64-bit architecture was introduced by AMD. Though x86 is currently the most used, it is currently changing. The use of ARM is rising day by day. ARM is another architecture. It supports 32-bit and 64-bit. ARM was introduced in 1999. ARM's most significant advantage is there are many manufacturers in ARM development. ARM is power efficient compared to x86. ARM processors are easy to modify according to need. All mobile telecommunication devices are based on ARM. Kubernetes on ARM looks promising.

1.2 Research Problem

Kubernetes is a complex tool setup. It has an upfront complexity. In the long run, it helps with deployment and scaling. There are multiple ways to set Kubernetes. Not to mention the source code is open source. So, many developers are modifying the source codes to meet their demands better. The availability of the system is also dependent on the deployment.

Kubernetes supports both x86 and ARM architecture. But it does not mean everything will work on ARM. By default, everything is made to run on x86 these days. Most of the runtime supports ARM, but there are a few exceptions. Few programs still do not have official support for ARM. So, developers have to use some alternative or compile source codes on their own.

Setting up Kubernetes is a complex task. The Kubernetes cluster is controlled by the master node. Kubectl is a CLI tool that lets the user control a cluster. Kubectl needs an auth key named Kubeconfig that provides access to a cluster.

Many cloud providers provide Kubernetes clusters to the user. It requires low maintenance. It is easy to scale. But it is not the cheapest option. But its availability is better or more reliable than on-prem configuration.

Kubernetes is evolving continuously. But most of the time, one production or large-scale implementation currently is on x86. There are a few cloud providers that provide Kubernetes on ARM architecture. The advantage of using ARM is, it is energy efficient. ARM processors do not have a lot of cores compared to x86 machines. But a large core count is not necessary for a distributed system.

The industry is moving towards ARM processors. Our computers, phones, cars, and IoT devices are leaning towards ARM processors. The question this research is trying to answer is:

What is the feasibility of using an ARM Kubernetes cluster instead of a

traditional x86 platform?

This research will try to discover if an ARM or hybrid system is more efficient and cost-effective than a traditional x86 system by running different benchmark tests on each system and trying to determine how it performs in each scenario.

1.3 Research Objectives

This research will compare different Kubernetes solutions in different scenarios. We need to benchmark each of the systems and try to keep the playing ground fair as much as possible. The objectives of our proposed research are:

1. To understand how containerization works
2. To understand how to make a Kubernetes cluster
3. To understand how Kubernetes work
4. To understand how an application works in different architectures
5. To develop Kubernetes clusters or provision clusters from major cloud providers
6. To develop a benchmark system, which will cover different scenarios based on workload (CPU, Memory, I/O)
7. To run benchmarks on those systems.
8. To evaluate the results.

1.4 Our Contributions

In this thesis, we tried to show a price per performance result of the Kubernetes in different architectures so that it can help developers in real time work. However, our sole intention is to show the developer community the result of Kubernetes analysis in different architectures so that they have an understanding about price to performance scale. Which will eventually help them to make the right decision of choosing Kubernetes services on a certain architecture that can save time and money on a large scale. Our Contributions are:

- Comparing both x86 vs ARM platforms.
- Finding out the cost-effective platform.
- Using Custom API to benchmark Kubernetes systems.
- Measured CPU, Memory, and I/O.
- Tested on major cloud providers' services.

1.5 Overview of the Thesis

Our thesis title is Kubernetes performance analysis on different architectures. Here we tried our best to come up with real world results which we have gathered from testing a couple of Kubernetes services offered by different organizations based on different architectures. The overview of our research is given below:

1. Chapter 1 : Here we have talked about Introduction, our research problem ,research objectives, our contributions etc.
2. Chapter 2: In this section we have mainly talked about Background, CPU architecture, x86 vs ARM, ARM advantages, Bare metal, Virtualization etc.
3. Chapter 3: In this portion we discussed about Related study mainly. Where we have covered Kubernetes Auto Scalability, Current situation of CPU Platforms, Different types of Kubernetes etc topics.
4. Chapter 4: Here we have talked thoroughly about methodology.We covered Testing Plan, Deploying Clusters, HTTP Tools, Creating REST API, CPU information collection API etc topic. Which is the core of our methodology.
5. Chapter 5: In this chapter we have covered Experimental Evaluation.Here we have talked about Experimental Settings, Connecting with cluster, Deploying pods and services on the clusters, Setting up Benchmark tools, Benchmarking Parameters etc. Which is a step by step process of our Experimental Evaluation.
6. Chapter 6: In this portion we have talked about Discussion part. Where Overview, Limitations are the core part of our discussion.
7. Chapter 7: In this chapter we have concluded our research in conclusion and also shared some of our future plan regarding this work.

Chapter 2

Background

Kubernetes is a powerful tool for a developer. Though it has an upfront complexity, for someone who has overcome that hurdle it is an extremely helpful tool. Our primary goal is to find a cost-effective solution based on CPU architecture.

Kubernetes is a complex deployment. It is the result of the evolution of many underlying systems. Firstly, people started to deploy VMs to make the best use of their resources. Then it improved into containers. Once people started to deploy large-scale applications the need for a tool that manages containers arose. Kubernetes is the result of that demand.

We are going to cover each underlying technology of Kubernetes. Starting from bare metal servers. Then we will discuss virtualization. Containerization technology comes after virtualization. Then we will cover Kubernetes.

2.1 CPU Architecture

The central processing unit, or CPU, is a component that powers computers (CPU). Most likely, CPUs are produced by businesses such as Intel, AMD, Fujitsu, Zhaoxin, and Qualcomm, and their technical specifications include things like Quad Core, 3.2 GHz, and 6Mb of cache. The CPU has components called registers that can store data. They function somewhat similarly to RAM, however, the memory cells are made entirely of logic gates instead of capacitor-based memory cells. While registers function far more quickly than RAM, they cannot store as much data. The ALU is the core of the CPU. It is capable of adding binary numbers. Numerous more arithmetic operations, like subtraction and incrementation, are also among its capabilities. The ALU may also carry out logical operations, such as determining whether or not two binary numbers are equal. The control unit interprets each command to determine its meaning before directing the other components' actions. Therefore, the control unit will signal what the ALU and memory are supposed to perform when it receives an instruction, which is simply a binary number. The instruction could tell the computer to add two numbers together or to save a particular number in RAM. Both the instructions and the data necessary for the computer to carry out those instructions are stored in the RAM. The stored-program computer is built on the concept of storing both data and instructions in the same memory. Two registers are required for reading from or writing to RAM: one register is used to store the RAM address that is being read from or written to, and the other register

is used to store the actual data. Buses are a collective term for the wire bundles that connect all of these parts. As a result, there are buses for transporting data, addresses, and instructions. A computer will typically have some input and output devices that can take in outside data and subsequently output the computation's findings. This might be a keyboard and monitor, or it might be something more complex like a data link.

2.2 x86 vs ARM

The CPU family based on the Intel 8086 and 8088 microprocessors is referred to as X86. Backward compatibility for instruction set architectures is ensured by these microprocessors. x86 initially had an 8-bit instruction set but was expanded to 16- and 32-bit sets. A family of central processing units (CPUs) known as ARM processors is based on the RISC (reduced instruction set computer) architecture. Advanced RISC Machine is the ARM acronym. Compared to more well-known server architectures like x86, ARM architectures provide a distinct approach to how the hardware for a system is created. The two main CPU processors, X86 and ARM, each have their advantages and disadvantages. They can be contrasted based on a few important factors, including the instruction sets they use, power usage, software, and application. The newest of its sort, RISC, divides tasks into short instructions that are processed in a single clock cycle, allowing for the speedier processing of millions of these instructions per second. Despite having to process numerous instructions, it operates at a faster overall speed thanks to its potent processors and pipelining. As opposed to X86 processors, which use the CISC architecture (Complex Instruction Set Computing). Multiple clock cycles are used to process complex instructions in a single step. It prioritizes process efficiency by handling numerous instructions in a single step while making use of the memory that is available. It accomplishes numerous jobs with higher throughput and performance by utilizing more registers. To handle multiple instructions, ARM demands extra memory. Even when GPUs and other peripherals are used, it uses 5W of electricity. X86 processors use more registers and place a greater emphasis on performance and high throughput. As a result, there is greater energy use and heat production in this area. devices with ARM chips Process is powered by Android operating systems, which were created specifically for ARM. Operating systems designed for X86 processors, such as Unix, Linux, and Windows, power desktops, laptops, and servers. Processor selection is based on application requirements and predicted performance levels. Compared to X86, ARM is favored in high-end, contemporary, and digital application devices. X86 is preferred by low-end, traditional back-end applications where reliable performance is needed.

2.3 ARM Advantages

A series of CPUs known as the Advanced RISC Machine (ARM) Processor is used extensively in electronic devices like smartphones, wearables, tablets, and multimedia players. This CPU used extremely little power and just needed a small number of instructions. The complexity of the circuits has decreased. Because there are fewer circuits, it is ideal for small-sized devices (It is more relevant now due to

demand for more compact devices). In terms of CPU efficiency, ARM is very effective. ARM processors are easier to develop and frequently considerably smaller than other CPUs because of their RISC architecture, which is less complex. Because of this, processors may now be used in smaller devices. The increased consumer need for more handheld and portable gadgets will profit from this according to study [26].

1. **Affordable:** ARM processors are incredibly inexpensive. It is produced for a substantially lower cost as compared to other processors. They are therefore suitable for producing inexpensive mobile phones and other electronics. Making ARM CPUs is often inexpensive and does not call for sophisticated machinery according to study [26].
2. **Work Faster:** ARM conducts 1 operation at a time. This speeds up its operation. It features a speedier response time and decreased latency according to study [15].
3. **Low Power Consumption:** Less power is used by ARM Processors. Initially, they were intended to operate at lower power. Even so, their architecture uses fewer transistors. They also have additional characteristics that make this possible.
4. **Multiprocessing feature:** The design of ARM processors allows for their usage in multiprocessing systems, which process data by using many processors according to study [15]. The first AMP processor, known as ARMv6K, had the hardware ability to support 4 CPUs.
5. **Load Store Architecture:** Data is stored in different registers by the processor using a load-store architecture (to reduce memory interactions).

2.4 Bare Metal

Physical hosting equipment dedicated to a single client is a "bare metal server". A bare metal server, which is often installed on-site or in a third-party data center (which can be rented or used for colocation), processes more data than any other hosting solution because the user has exclusive access to all computing capabilities. It has exclusive access to CPU, RAM, Disk space, and Bandwidth. Bare metal deployment is a fully dedicated computing resource but there are more reasons why companies choose this for example high levels of processing power, High data privacy due to the lack of other tenants, Consistent input/output operations per second (IOPS), Predictable billing (typically monthly), Complete control over the server's hardware and the software stack, Consistently high performance. The advantages of bare metal are difficult to beat if your program is sensitive to performance and you want to keep data at a single-tenant device. But when it comes to running multiple applications on a single device problem arises, multiple applications may require a different environment set up which is impossible to run in the same environment setup. To solve that problem companies need different devices for each application which is a fully waste of physical resources. Because one application might not need 100 percent of a device but still other applications will not be able to run on the same device as a result physical resources are highly underutilized. Bare metals are

hard to partition for different applications and difficult to scale. Not to mention buying a new machine for every single application is expensive to buy and maintain.



Figure 2.1: Bare Metal Server [29]

2.5 Virtualization

Virtualization took off in the 1990s, solving the issues with bare metal deployments. VMs, or virtual machines, were becoming very popular. Virtualization's primary goal is to manage workload by improving the scalability, effectiveness, and cost-effectiveness of traditional computing. Operating system virtualization, hardware-level virtualization, and server virtualization are just a few examples of the many areas where virtualization can be used. In cloud computing, customers are virtually allotted space and memory on the servers, which calls for a host (platform) on which to execute a hypervisor (software that communicates with the hardware) according to study [3]. A hypervisor is the essential component of hardware that powers virtual machines (VMs). The available physical resources can be divided into several virtual ones, known as Virtual Machines, using a hypervisor, which is software that emulates a specific piece of computer hardware or the complete computer (VMs). The VMs that are created and maintained by the hypervisor is referred to as Guest Systems, while the machine that runs the hypervisor is referred to as the Host System. Hypervisors can be installed either on top of the OS or top of the hardware. However, occasionally it might be difficult to distinguish between the two, like in the case of Kernel-based Virtual Machine (KVM), the virtualization module included within the Linux kernel. Decoupling allows I/O devices to be time- and space-multiplexed, which permits the implementation of multiple logical devices by a lower number of physical devices.

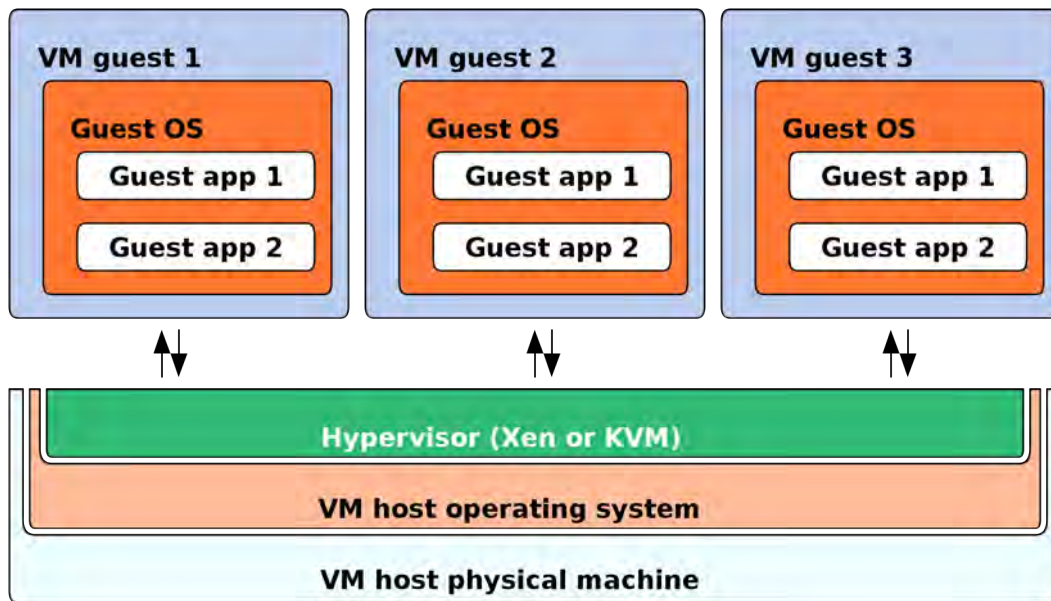


Figure 2.2: KVM, XEN [31]

2.6 Containerization

Containerization is a smart to deploy application. The theory of containerization is not new but in recent years the usage of containerization has skyrocketed. It is kind of like a Virtual Machine but without the extreme overhead. In containerization the process shares the Kernel. So there is no need to virtualize hardware and software. It has revolutionized the way we create, deploy and test applications.

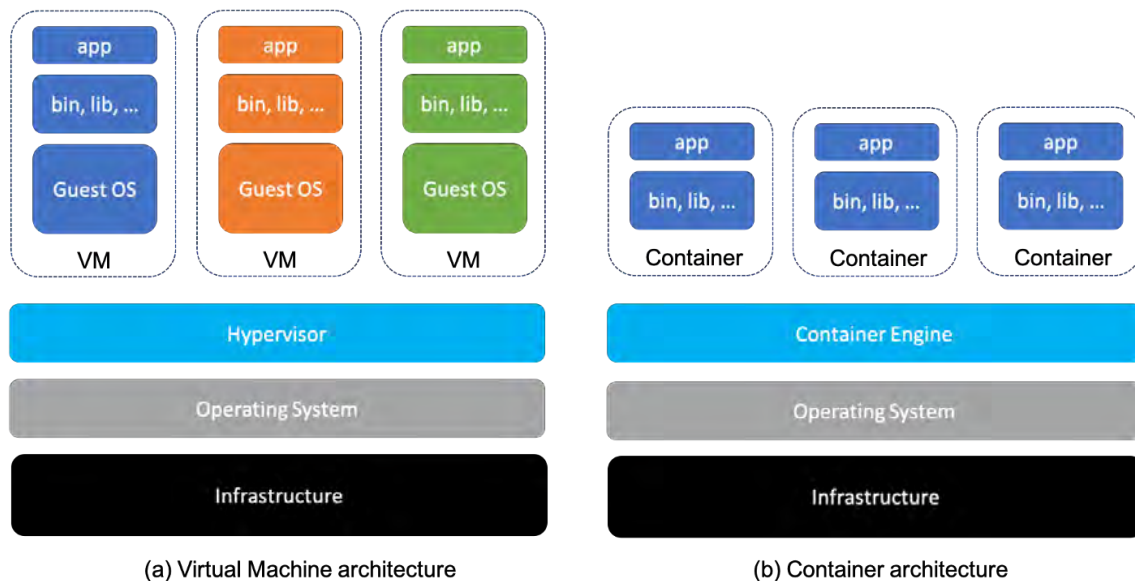


Figure 2.3: VM vs Container [17]

2.7 Docker

Out of all the containerization technology, nobody is as popular as Docker. It is impossible to think about containerization without thinking about Docker. Docker is open-source, maintained by Google and written in Golang. Which is also maintained by Google. Each runtime is called containers. According to study [5], Docker is extremely efficient in deployment. Docker helps developers to build and deploy applications faster. According to study [6] Docker helps developers to create cloud-native applications. Docker helps developers to create isolated environments faster to run their code. According to study [7], it helps to test applications in an isolated environment first then deploy on the production server.

2.7.1 Docker Client and Server

The machine that runs docker is a docker server. Through docker socket a client can access the resources of that server. In most scenarios the client and the server will be on the same machine. Through the RESTful API a client can send commands to a remote server, covered by study [4].

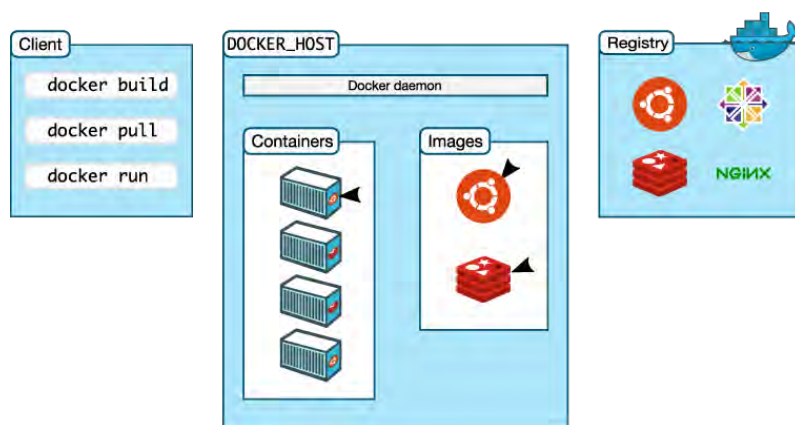


Figure 2.4: Docker [25]

2.7.2 Docker Images

Docker images are like templates. Container images are not a new concept. The actual code is not images. Users can make their Image then create a container on that image. Images are based on some sort of Linux Os. Most of them are based on Debian or Alpine. Alpine is extremely light. They do not have all the functionality of the OS. Images are layered. Images are built using a file named Dockerfile. After creating the image the image needs to be tagged to index the image for future use.

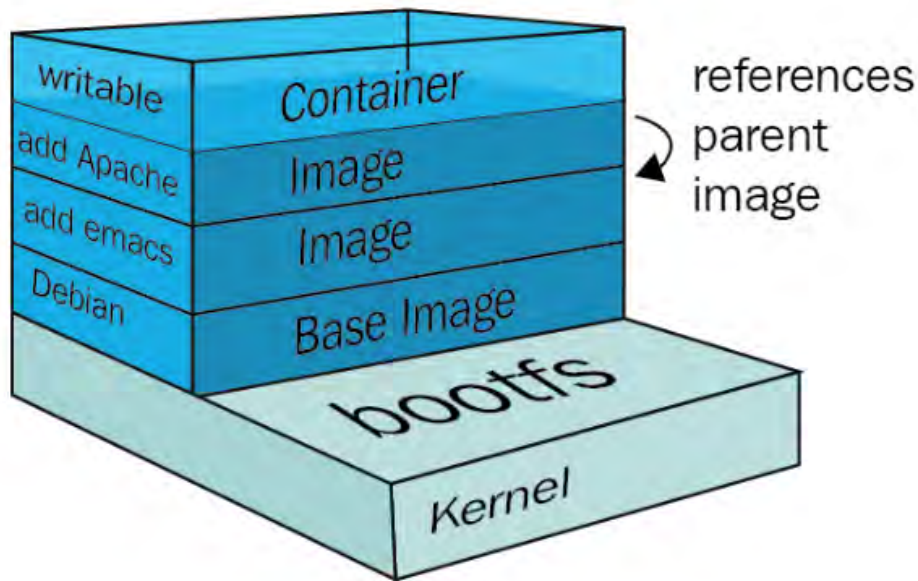


Figure 2.5: Docker Image [30]

2.7.3 Docker Registries

Docker Registries are storage for Docker images. To use Docker images on different devices users first need to create an image then tagged then upload the image to a repository. Docker Hub is the default registry. Images can be public and private.

2.7.4 Docker Containers

Docker containers are the actual runtime. Docker containers run on Docker images. To run an application such as a web server, database etc a container is needed. Docker containers are created using Docker run. To run multiple containers or to run containers with specific configuration we use docker-compose.



Figure 2.6: Docker Container [23]

2.7.5 Scalability

Docker helps with horizontal scaling. To Increase the performance of single threaded applications performance developers can create containers of the same application and use a load balancer to scale the application. Database clusters can be created using multiple Docker containers.

2.7.6 Speed

According to the study [8], Docker helps with fast deployment. As a developer can create Docker images and push it to a registry, they can pull this image to use in multiple servers located in multiple places faster. Instead of pulling the code and setting up all the variables they will only need to run the docker-compose to use their latest image.

2.7.7 Portability

Images can be easily moved from device to device using the repository. It helps with portability. Just pulling the image is all a developer needs to do to get their codes.

2.7.8 Density

By using Docker, we get more performance. Hypervisor uses a lot of resources. Docker does not need that. So, it gives us more performance on the same device. On the same device we can get more utilization than a hypervisor.

2.8 LXC

LXC is Linux Containers. They are not as light-weight as docker. They are more like a VM and have their own networking. Docker generally use bridge networking and creates multiple virtual networks for routing. LXC acts as a individual device but lighter than VM.

2.9 Kubernetes

We are going to discuss about Kubernetes architecture now. The architecture of Kubernetes has many components. First comes a cluster. It is a collection of hosts and network resources according to study [10]. A Kubernetes cluster can be a single node cluster. These types of clusters are used for testing and learning. But a very basic cluster should be 1 master and 1 worker node cluster. For production level usage 1 master 3 worker is recommended. A Kubelet is primarily a Kubernetes process that allows the cluster to talk to one another, communicate with one another, and execute actions on those nodes, such as launching application processes. Each worker node has containers for different applications deployed on it; therefore, a user may have a variety of docker containers operating on worker nodes depending on how the workload is split. The actual work is done primarily in the worker nodes. The user application will run on the worker node. Now we must learn about master nodes. The master nodes run numerous Kubernetes programs that are required to properly run and govern the cluster. An API server, which is also in a container, is one of these processes. The entry point to the Kubernetes cluster is an API server. So, this is how multiple Kubernetes clients communicate with each other, such as a UI if the user is using the Kubernetes dashboard, or an API if the user is using scripts and automation technologies, as well as a command-line tool. In addition, another process known as a controller manager is operating in master nodes. This essentially maintains track of what's going on in the cluster, such as

whether something needs to be fixed, whether a container has died and needs to be restarted, and so on. Another component is the scheduler, which is in charge of scheduling containers across nodes based on workloads and server resources available on each node. It's an intelligent procedure that determines which worker nodes the next container should be scheduled on based on the worker nodes' available resources and the container's load requirements. Etcd key-value storage is another component of the entire cluster. Which encapsulates the current state of the Kubernetes cluster at any given time. As a result, it contains all of the data configuration as well as all of the status data for each node and container within the node. Finally, the virtual network is a crucial component. It allows those nodes, worker nodes, and master nodes to communicate with one another. In basic terms, a virtual network transforms all of the nodes in a cluster into a powerful machine with the sum of all of the nodes' resources. Let's take a deeper look at Kubernetes components.

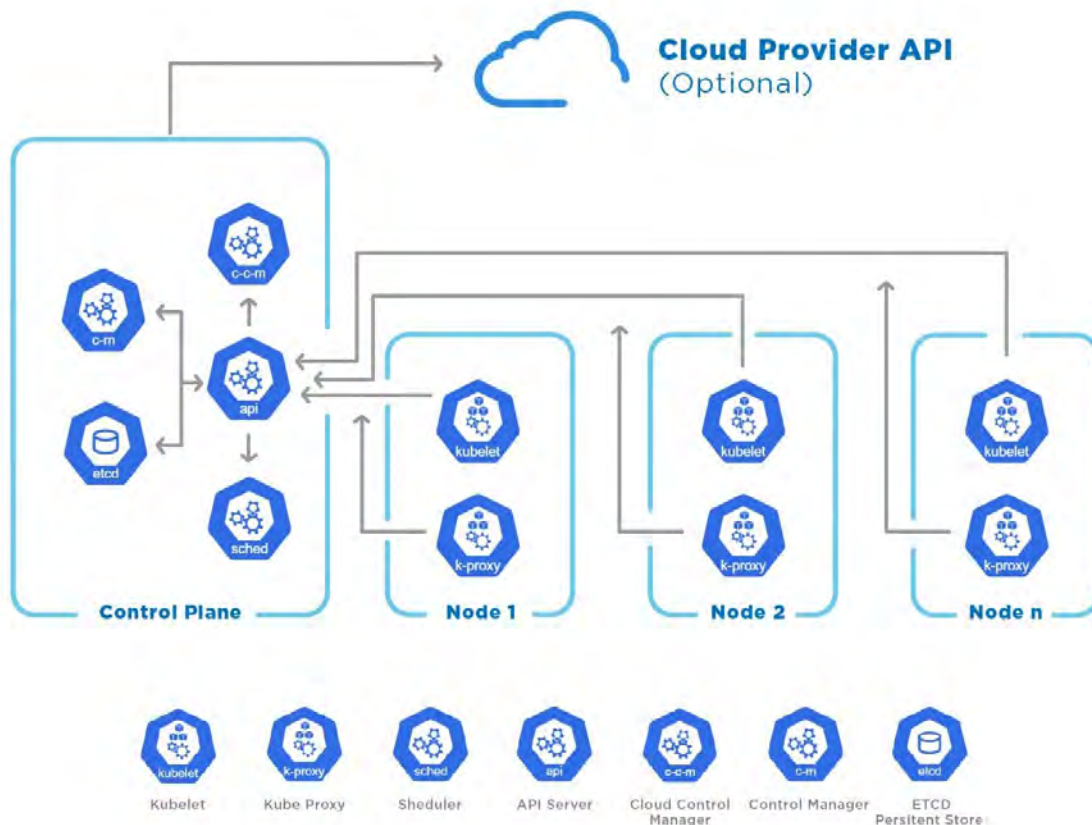


Figure 2.7: Kubernetes [27]

2.9.1 Control Plane Components

Control plane components are responsible for managing Kubernetes resources and making sure everything is working properly. It tracks all the events on a cluster. If a pod fails to run it will take care of that.

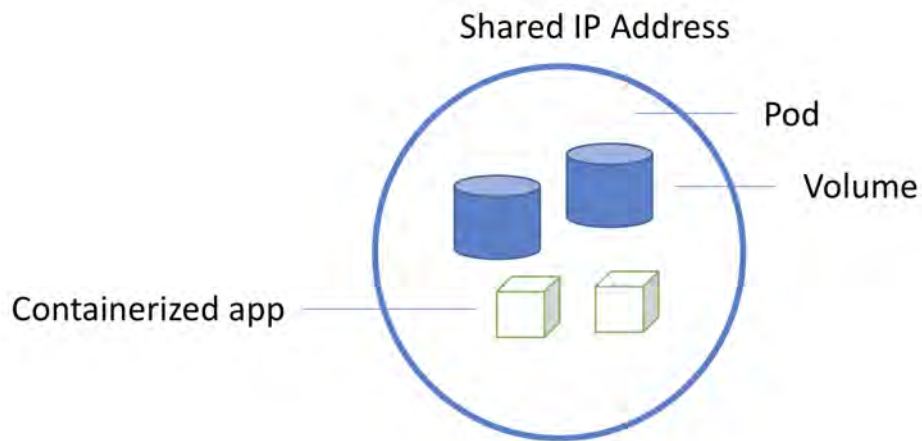


Figure 2.8: Kubernetes Pods [28]

Any machine in the cluster can be used to run control plane components. To keep things simple, set-up scripts usually do not launch user containers on the machine where the control plane components are started.

2.9.2 Kube-API-Server

KubeAPI-server manages the connection between nodes and controllers. When new nodes are added to the cluster the API server communicates with the nodes and scales the cluster. It sends commands to the nodes and tracks their behavior. It also divides I/O loads.

2.9.3 etcd

Etcd is a the database of the cluster. It contains all the configuration and authentication keys.

2.9.4 Kube-Scheduler

Kube-Scheduler maintains the daily scheduled tasks. A lot of tasks are needed to be done routinely to make sure things are running properly.

2.9.5 Kube-Controller Manager

These are the Kube-Controller Manager.

Node Controllers: Node controllers are in charge of identifying outages and taking appropriate action.

Job Controller: Keeps an eye out for Job objects that stand for sporadic duties and subsequently constructs Pods to carry out those activities.

Endpoints Controller: Adds objects to the Endpoints object by joining Services and Pods.

Service Account & Token Controllers: Generate API access tokens and default accounts for new namespaces.

2.9.6 Cloud-Controller Manager

In most scenarios Kubernetes will run on the cloud. So the cloud provider adds their application in the cluster to manage the clusters and provide a client on demand basis. For different cloud providers these implementations are different.

Node Controller: After a node stops responding, to check the cloud provider to see whether it has been destroyed in the cloud.

Route Controller: Controls internal routing.

Service Controller: Manages Kubernetes services.

2.9.7 Node Components

Every node has some services to run alongside Kubernetes applications. They communicate with the master node or nodes to receive tasks and instructions. They track pods' resource utilization and send them to the master. If something goes wrong, it reports to the master node.

2.9.8 Kube-proxy

Kubernetes relies on complex networking. It communicates in between for many services. It is not easy to access an running application on Kubernetes on a default port. Firewall protocols and network rules are set up by this Kube-Proxy.

2.9.9 Container Runtime

Kubernetes supports container runtime other than Docker. Some of them are containerd, CRI-O etc. Users can choose which container runtime they want to use for their application.

2.9.10 Services

To access the application running on Kubernetes, services are needed like Node Port, Load Balancer, Cluster IP, Ingress. To use a FQDN on an application, Ingress is used.

Chapter 3

Related Study

3.1 Kubernetes Performance Major Cloud Providers

According to study [2], Kubernetes is a better deployment system than most container orchestration systems. According to study [12], the author compared a few managed Kubernetes systems and benchmarked their CPU, Memory, and I/O. All of them are synthetic benchmarks and though they give us an Idea about the performance, Real world applications are much more complicated. Most of the time, the loads are HTTP-based and HTTP loads are not linear. In our testing, we are primarily focused on determining the CPU's performance based on the HTTP performance.

3.2 Kubernetes Auto Scalability

Kubernetes's biggest strength is scalability and availability. The small unit of application running instances is known as pods. Pods will automatically scale or restart based on requirements. It is also easy to increase or decrease the number of instances based on demand according to study [11]. Kubernetes also balances load automatically according to study [14]. One can create a cluster with a single node for testing but no one will use that in production. So, there will be multiple nodes in a cluster. And the load will be equally distributed among the nodes. It confirms the fair allocation of resources.

3.3 Different Types of Kubernetes

There are also different types of Kubernetes. Each of them manages the clusters, pods, and services in their way. They have their pros and cons according to study [21]. Though Creating an ARM cluster is a relatively new concept in cloud computing. In the Homelab community, many people are creating clusters on Raspberry Pi according to study [19]. According to study [22], A raspberry PI cluster was created based on K3s. K3s is a lighter version of Kubernetes. Google's vanilla Kubernetes is known as K8s. The price to performance of a raspberry pi cluster is great. Though it has some limitations. It lacks availability. It is hard for an individual to create a highly available cluster. The individual will also have to bear the upfront cost of owning the hardware. Here, a cloud solution makes more sense.

3.4 Current Situation of CPU Platforms

Currently, only Intel and AMD make x86 processors. Almost all Desktop and Laptops run on x86 CPU. All mobile devices are ARM based according to study [18]. Intel has some ARM CPUs but their performance is very poor. Other than Apple, nobody has still created any desktop ARM CPU that performs well. Apple's M1 CPU is revolutionary [13]. Raspberry Pi is a great example of ARM's success. It is cheap and power efficient. So it is perfect for small-scale servers or IoT applications according to study [1]. It acts as a gateway for a lot of developers as it is cheap and easily accessible.

3.5 ARM Support on Major Cloud Providers

For server or enterprise applications Intel has Xeon-based CPUs according to study [9]. They have a high Core count and a lot of PCI-E lanes for additional hardware. AMD has EPYC CPUs for the enterprise. They are available up to 64 cores and up to 128 PCI-E Gen-4 lanes according to study [20]. These products have served well till now in high-performance computing. ARM's enterprise lineup of products is Neoverse. Neoverse N1 has up to 128 cores and they are cheap and power efficient. AWS's Graviton CPU is based on Neoverse N1[16]. Our testing system will determine the best price-to-performance architecture for general Kubernetes applications.

Chapter 4

Methodology

In this paper, we created a stress testing model described in 4.1. Stress testing cloud systems are pretty complex. Our goal was to find out the performance of each architecture on the same load and then compare the price to performance to determine the better platform based on cost. The following aspects were given importance:

1. What is the complexity of deploying pods in clusters based on different architectures?
2. What kind of benchmark tool will be best for our experiment?
3. How will these clusters perform under stress tests?
4. Which is the better option when the cost is considered?

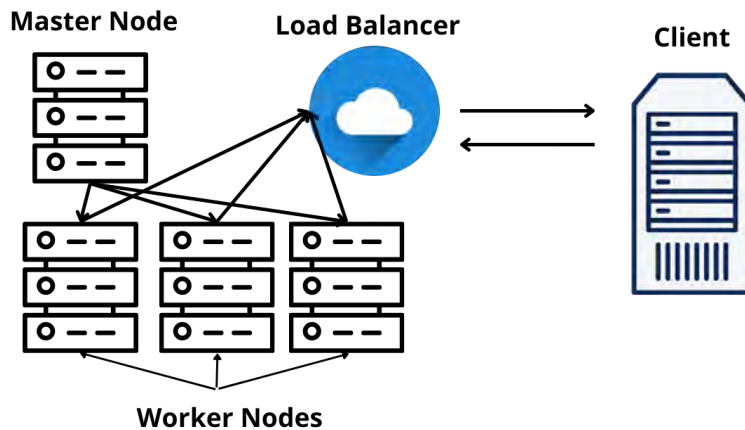


Figure 4.1: Testing Model

4.1 Testing Plan

Kubernetes is primarily used in microservices or web applications. Microservices are mostly based on HTTP protocol. REST API is a good example of microservice.

GET, POST, DELETE, PATCH, and PUT protocol are used to fetch data from server, send data to server, delete data from server and update data of the server. We created clusters on different Cloud providers which are running on different CPU architectures. We created the same application on different popular frameworks and then ran benchmark on them. These tests covers a lot of scenario. We would also ran a sythetic benchmark called Sysbench to test our nodes' I/O. It gave us read/s and write/s of the system.

4.2 Deploying Clusters

We tested on 4 different platforms. We tested on Intel, AMD, Ampere, and Graviton. Here Intel and AMD are based on x86 architecture. Ampere and Graviton are running on ARM architecture. We made Kubernetes clusters on these platforms. We created these clusters on the public cloud. For our testing, we created clusters on GCP, AWS, and Oracle Cloud. We tried to keep the testing field consistent as much as possible. All the clusters had 3 Nodes each consisting of 2 Cores and 8 GB of RAM. Each cluster had its own Virtual Cloud Network. All the clusters are deployed in different fault domains/ zones. Which increases availability even more at the cost of negligible latency. Clusters were created on the web GUI of Google Cloud Platform and Oracle Cloud Platform. To create clusters on AWS we had to use the AWS-CLI tool to access our cloud resources and we used EKSCTL to create the Kubernetes cluster from a YAML configuration file.

4.2.1 x86 Intel Cluster

An Intel x86 cluster was created on the Google Cloud Platform. The Nodes were running on Intel® Xeon® Gold 6314U processors. These Nodes were based on GCP N2 type instances. Kubernetes version 1.22.11-gke.400 was used to create the cluster. The region of the data center was "asia-southeast1-a".

4.2.2 x86 AMD Cluster

An AMD x86 cluster was created on the Google Cloud Platform. The Nodes were based on GCP T2D instances. The Kubernetes version and datacenter region are the same as the Intel x86 cluster. The Nodes were running on an AMD EPYC 7B13 processor.

4.2.3 ARM Ampere Cluster

An Ampere cluster was created on Oracle Cloud. The cluster was based on Kubernetes version 1.24.1. The cluster nodes are based on Oracle. It was hosted on Oracle Cloud "ap-singapore-1" zone. The nodes were based on VM.Standard.A1.Flex.

4.2.4 ARM Graviton Cluster

A Graviton cluster was created on AWS. Kubernetes version of the cluster was 1.22. The nodes were based on AWS m6g.large instances. The cluster was located on "ap-southeast-1" AWS zone. These nodes were running on Neoverse N1 CPU.

Cloud Provider	Intance Type	Number of Nodes	CPU	Cores	RAM	Kubernetes Version	Datacenter Location	CPU Architecture
GCP	N2	3	Intel® Xeon® Gold 6314U	2	8	1.22.11-gke.400	asia-southeast1-a	x86
GCP	T2D	3	AMD Epyc 7B13	2	8	1.22.11-gke.400	asia-southeast1-a	x86
Oracle Cloud	VM.Standard.A1.Flex	3	Ampere A1	2	8	1.24.1	ap-singapore-1	ARM
AWS	m6g.large	3	Graviton	2	8	1.22	ap-southeast-1	ARM

Table 4.1: Platforms

4.3 HTTP Tools

We created HTTP applications to benchmark the clusters. The primary goal of the HTTP application will be to stress the CPU of the nodes and to check the performance of the HTTP . As we want to keep any other aspect of the system isolated from the test. We made an application that calculates the Fibonacci series. The last position of the series is passed through the application by GET request and the result and sent in a JSON body. We only focused on the CPU of the system. So we avoid using any external API, external database, block or object storage. These things will introduce many variables. We can not deploy all of our testing in the same datacenter as different providers provide different systems running on different CPU architectures. The complexity of our application is $O(n)$. A single execution of the application does not put much stress on the system. But for our experiment we are going to create synthetic load to find the performance of the system in the worst case scenario.

```

procedure fibonacci : fib_num

  IF fib_num less than 1
    DISPLAY 0

  IF fib_num equals to 1
    DISPLAY 1

  IF fib_num equals to 2
    DISPLAY 1, 1

  IF fib_num greater than 2
    Pre = 1,
    Post = 1,

    DISPLAY Pre, Post
    FOR 0 to fib_num-2
      Fib = Pre + Post
      DISPLAY Fib
      Pre = Post
      Post = Fib
    END FOR
  END IF

end procedure

```

Figure 4.2: Fibonacci Pseudo code

4.3.1 Creating REST API

We are using the REST API as it is widely used in the industry and it will give us the best scenario of real-life implementation of Kubernetes. The last index of the Fibonacci series will be delivered to the application through a GET request and it will be in the URL. The application will calculate the value and send the value through an HTTP response with a status code 200 and the value in the body as a JSON. We have created the same application on 4 major frameworks. They are Express, Flask, Gin, and Actix. They are running Node JS, Python, Golang, and Rust. Currently, they are one of the most used cloud-native languages. Node Js and Python are two mature languages. Though Golang and Rust are comparatively new in the programming world, they are gaining massive popularity overnight for their performance and memory management.


```
const express = require('express');
const app = express();
app.use(express.json());
app.use('/:value', (req, res) => {
  let num = req.params.value
  let num1 = 0;
  let num2 = 1;
  let sum;
  for (let i = 0; i < num; i++) {
    sum = num1 + num2;
    num1 = num2;
    num2 = sum;
  }
  console.log(num2)
  res.status(200).json({
    status: "ok",
    data: num2
  });
});
app.listen(5000);
```

Figure 4.3: Express API

4.3.2 CPU Information Collection API

From the dashboard of every major cloud provider we can see the CPU's family or generation but they do not show which specific CPU our system is running on. To get the specific CPU model we created an Express Application that sends the System's CPU information such as model, clock speed and number of cores in a JSON body.

```

const os = require('os');
const express = require('express');
const app = express();
app.use(express.json());
app.get('/', (req, res) => {
  try {
    res.status(200).json({
      cpu: os.cpus()[0].model,
      thread: os.cpus().length
    });
  } catch (e) {
    res.status(404).json({
      status: 'failed'
    });
  }
});
app.listen(5000);

```

Figure 4.4: CPU info collection API

4.3.3 Containerization

To run our application on a Kubernetes cluster we had to convert our application into a docker image. Here we faced our first challenge for running our application on different architectures. Though the code for running on different architectures was the same, the runtime created for x86 will not work on ARM as they have different instructions. Such as Node JS for x86 and Node Js for ARM are not the same binary file. When we are creating an HTTP docker image we use some sort of Docker image as a base and build on top of it. So for creating x86 systems we used Docker images intended for x86 systems and for ARM systems we used Docker images intended for ARM systems. To simplify our building we created a Dockerfile to create the image. We created the x86 Image on an x86 system by running the Dockerfile on it. To create the ARM image we ran the Dockerfile on a Raspberry Pi 4. The images were tagged differently to make them easy to identify.

```
FROM node:latest
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 5000
ENTRYPOINT ["node","index.js"]
```

Figure 4.5: Dockerfile

4.3.4 Uploading to Image Registry

We have created clusters on different datacenters. To access the images on different datacenters we needed to upload our images to an image registry. We uploaded our images to Docker Hub which is a free image registry for public images. First, we had to create an account on docker hub and then login through the cli to upload the image with a custom tag.

4.4 Sysbench

We used Sysbench to measure the storage performance of the nodes. We used an existing Sysbench application container. 1 single pods was deployed to measure the best possible storage performance.

Chapter 5

Experimental Evaluation

Now we are going to run our experiment and analyze the results to conclude. From our results, we will be able to understand the performance differences between the two architectures.

5.1 Experimental Settings

We had to connect with our cluster and then create pods and services to access our application. Then we called our API with a client simulator that will create a massive number of requests which will stress test the system.

5.1.1 Connecting with the cluster

To access the resources of a Kubernetes cluster we need to use a tool called Kubectl. We use Kubectl version 1.24.0. With the help of Kubectl, we can see a cluster's nodes, pods, namespaces, and services. We can also inspect logs and create interactive shells inside the pods. To connect with a cluster we need a configuration file named KubeConfig. To access the GCP clusters we used the Google Cloud CLI tool to access the KubeConfig file. For the AWS cluster, we used the AWS-CLI tool and to access the OCP cluster we used the OCI tool.

5.1.2 Deploying pods and services on the clusters

After connecting with the cluster we used a YAML file to deploy our application on the Kubernetes cluster. For our experiment we deployed 15 pods, so 5 pods on each of our nodes. Kubernetes uses internal networking to communicate within the cluster. To access the HTTP application we used an external load balancer. The load balancer distributed the load among the nodes. All of this was deployed in the default namespace. The services were also deployed in the same YAML file. We deployed the x86 images on x86 clusters and ARM images on the ARM clusters. The application that collects CPU information was also deployed with a YAML file and routed with an external load balancer.

```
apiVersion: v1
kind: Service
metadata:
  name: node-thesis
spec:
  type: LoadBalancer
  selector:
    app: node-thesis
  ports:
    - port: 31111
      targetPort: 5000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-thesis
spec:
  replicas: 15
  selector:
    matchLabels:
      app: node-thesis
  template:
    metadata:
      labels:
        app: node-thesis
    spec:
      containers:
        - name: k8s-nginx
          image: faysalkhan8p/thesis-node
          ports:
            - containerPort: 5000
```

Figure 5.1: Express API Deploy

5.1.3 Setting up Benchmarking Tool

We used an open-source tool named Vegeta to stress test our cluster. Vegeta was set up on a Linux machine running Linux Mint 21. It was running on Intel 5820K. Intel 5820K has 6 cores and 12 threads. It had 32 GB of RAM and 256 GB of SSD. It also had 1 GBPS uplink to the cloud data centers.

OS	CPU	Core	Threads	RAM	CPU Architecture	Linux Kernel
Linux Mint 21	Intel 5820K	6	12	32 GB	x86	5.15

Table 5.1: Testing Client

5.1.4 Benchmarking Parameters

We stress-tested each cluster with Vegeta. Vegeta is based on Golang. We ran the test for 30 seconds for each cluster running each application. For Express, Gin, and Actix applications we sent 8000 requests per second for 30 seconds. For Flask we sent 500 requests for 30 seconds. We noticed if we were sending more than 500 requests per second the results were degrading exponentially. So for Flask, we stuck with 500 requests per second.

To measure I/O, we ran a Sysbench container. A single pod was deployed in the cluster. The parameter of the test was set to default. Prime Number Limit was set to 2000 and a single thread was used to run this benchmark

5.1.5 Collecting CPU information

We ran the CPU information collecting API to get the CPU model, clock speed, and the number of cores.

5.2 Experimental Results

After running the benchmark we have gathered data on the benchmark. We noted down throughput, and success rate, and calculated the price to performance.

Platform		Express JS	Flask	Gin	Actix
x86 Intel	Throughput (requests/second)	2128.84	309.2	7981.29	5647.71
	Cost per Month (\$)	296.1			
	Price to Performance	7.189598109	1.04424181	26.95471125	19.07365755
	Success Rate (%)	77.25	92.63	100	99.92
x86 AMD	Throughput (requests/second)	1092.93	305.36	7982.81	6855.55
	Cost per Month (\$)	314.47			
	Price to Performance	3.475466658	0.971030623	25.38496518	21.80033072
	Success Rate (%)	48.95	91.6	100	99.92
ARM Ampere	Throughput (requests/second)	7958.88	361.18	7989.75	5797.38
	Cost per Month (\$)	120.4			
	Price to Performance	66.10365449	2.999833887	66.36004983	48.15099668
	Success Rate (%)	100	99.97	100	99.92
ARM Graviton	Throughput (requests/second)	7956.84	243.25	7980.15	6226.32
	Cost per Month (\$)	243.42			
	Price to Performance	32.68770027	0.9993016186	32.78346069	25.57850629
	Success Rate (%)	99.96	72.24	99.93	99.82

Table 5.2: API Benchmark Results

Platform	CPU Architecture	Read/s	Write/s
GCP Intel	x86	2137.06	1424.71
GCP AMD	x86	2086.93	1391
Oracle Cloud	ARM	937.51	624.94
AWS	ARM	1807.59	1205.06

Table 5.3: I/O Results

5.2.1 Express Results

The throughput of express API is highest on Ampere. In the second place, there is Graviton. Intel and AMD and third and fourth respectively. The success rate is also the highest on Ampere. Graviton is close second and Intel and AMD are third and fourth again. If we consider price to performance Ampere is ahead by a long shot. Graviton’s price-to-performance is half of Ampere and compared to ARM platforms, x86 platforms’ price to performance is really low.

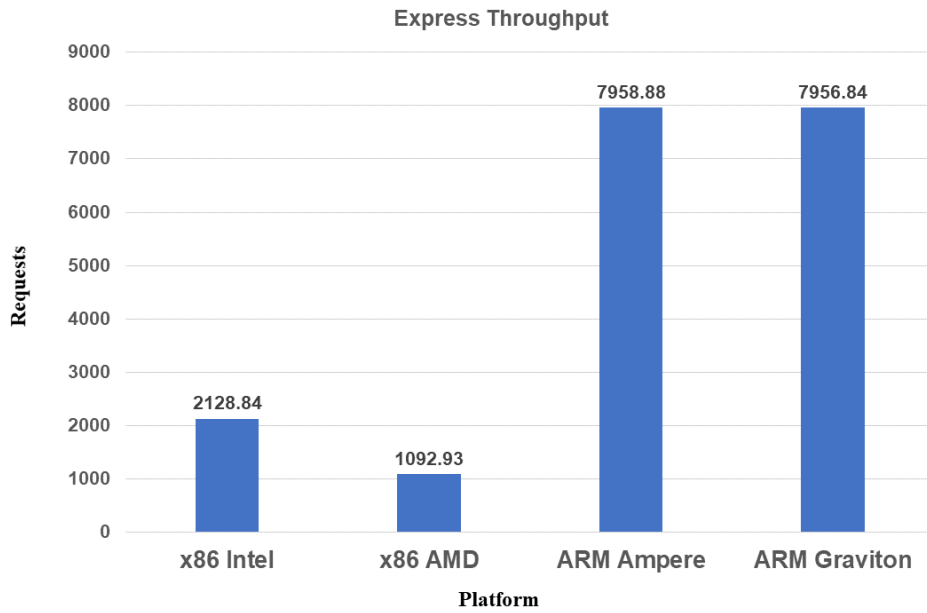


Figure 5.2: Express Throughput

From figure 5.2 we can see that the express API is highest on ARM Ampere then in ARM Graviton. x86 AMD takes the lowest amount of requests which is 1092.93.

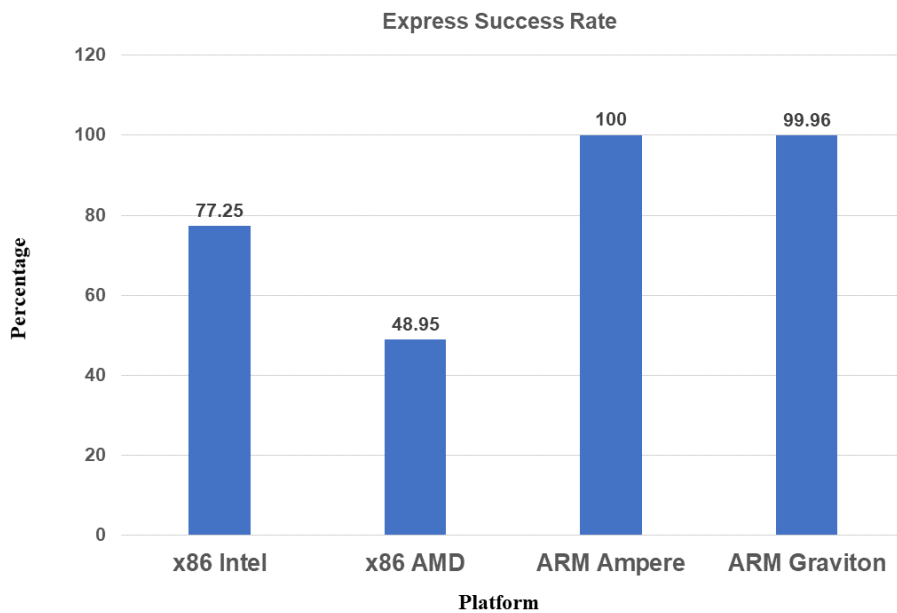


Figure 5.3: Express Success Rate

From figure 5.3 we can see that the highest rate of success in ARM Ampere which is 100 percent and ARM Graviton in the second position having 99.96 percent of rate. x86 AMD provides the lowest rate of success which is 48.95 percent.

5.2.2 Flask Results

Flask's throughput is highest on Ampere. Followed by Intel, AMD, and Graviton. The success rate is also highest on Ampere and Intel, AMD and Graviton are second, third, and fourth respectively. For Ampere price to performance is also very high compared to other platforms. In the second place, there is Intel and it is less than half of Ampere. AMD is close third and Graviton's price to performance is the poorest here.

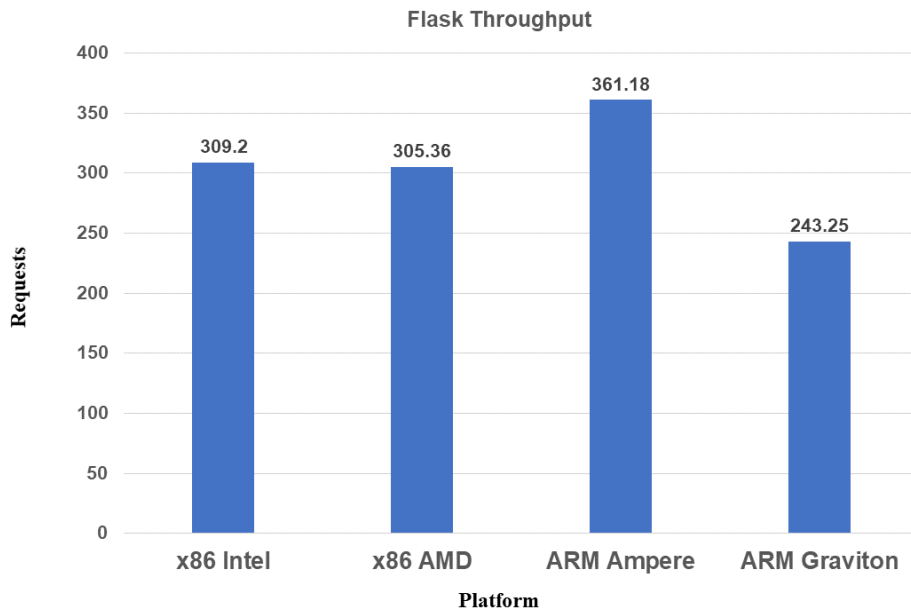


Figure 5.4: Flask Throughput

In figure 5.4 Flask throughput is also highest on ARM Ampere and lowest on ARM Graviton. Intel and AMD are second, third respectively.

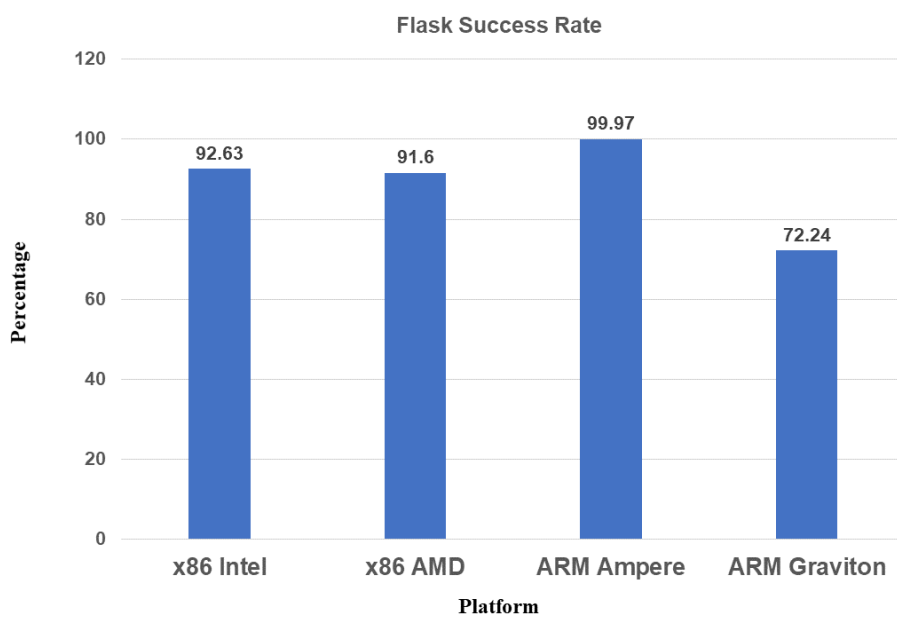


Figure 5.5: Flask Success Rate

In figure 5.5 Flask success rate is highest on Ampere and second, third on Intel and AMD respectively. Graviton shows the lowest performance rate that is 72.24 percent.

5.2.3 Gin Results

Throughput is almost the same for all platforms. For ARM, Gin's price to performance is equivalent to Express. But the price to performance increased for x86 platforms. The success rate for all platforms is close to 100 percent.

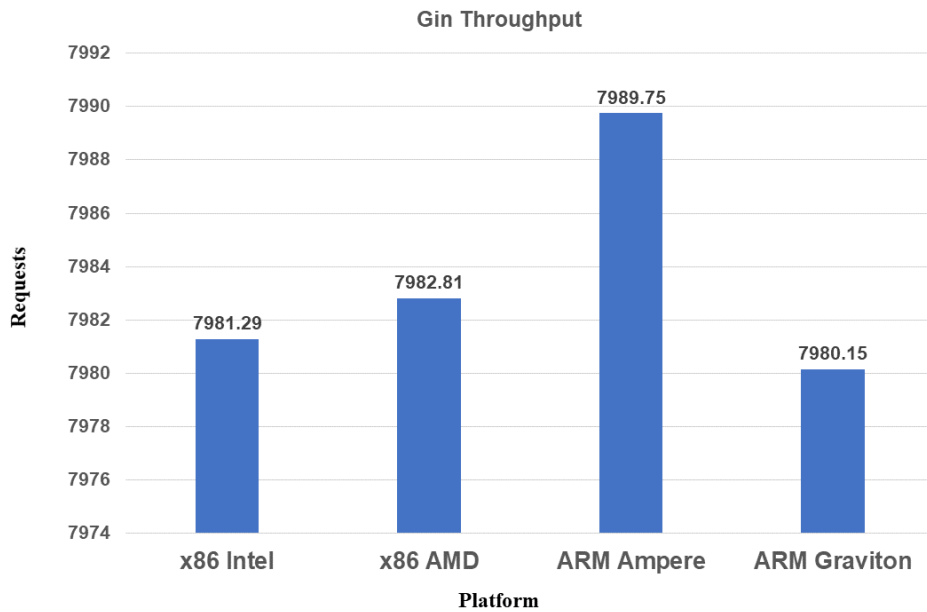


Figure 5.6: Gin Throughput

In figure 5.6 we can see that the Ampere takes the highest request that is 7989.75 and AMD ,Intel ,Graviton are second ,third and fourth respectively.

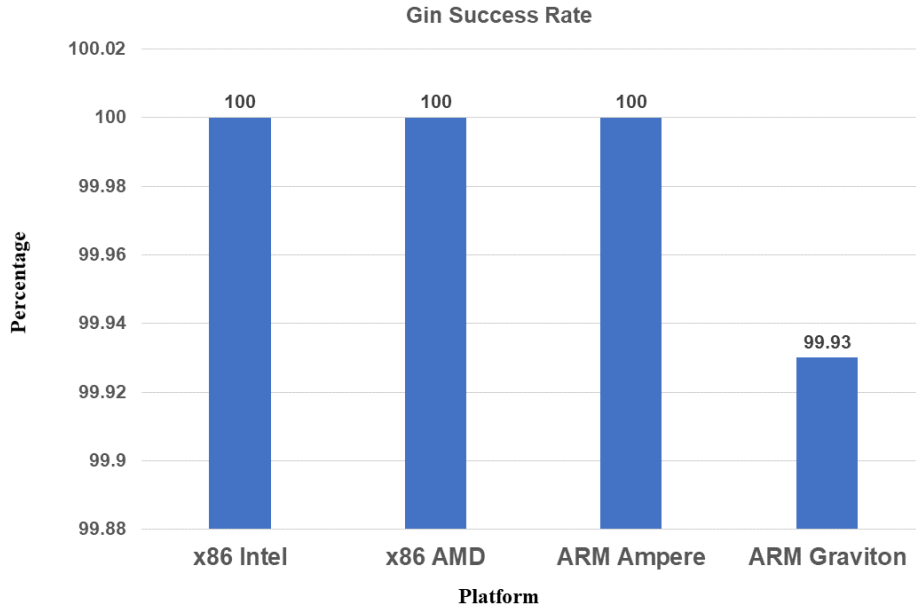


Figure 5.7: Gin Success Rate

In the figure 5.7 we got 100 percent rate in Intel, AMD and Ampere platforms. Only the Graviton gave 99.93 percent of the rate.

5.2.4 Actix Results

For Actix AMD's throughput was the highest. Graviton, Ampere, and Intel are second, third, and fourth respectively. The success rate is close to 100 percent for all platforms. In price to performance, Ampere is the winner. Followed by Graviton, AMD, and Intel.

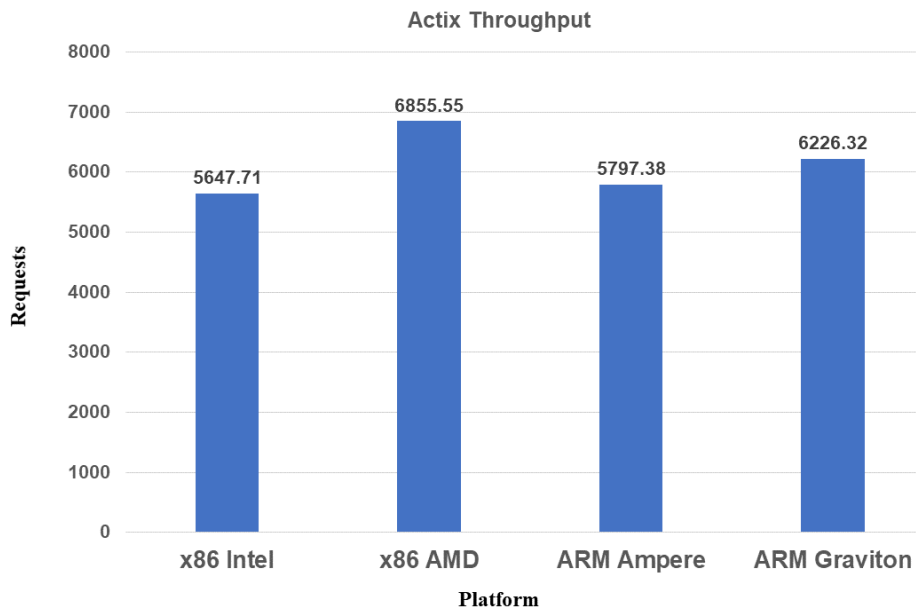


Figure 5.8: Actix Throughput

From figure 5.8, we can figure the highest rate of request on AMD for Actix throughput. Graviton having the second position and Ampere, Intel are third and fourth respectively.

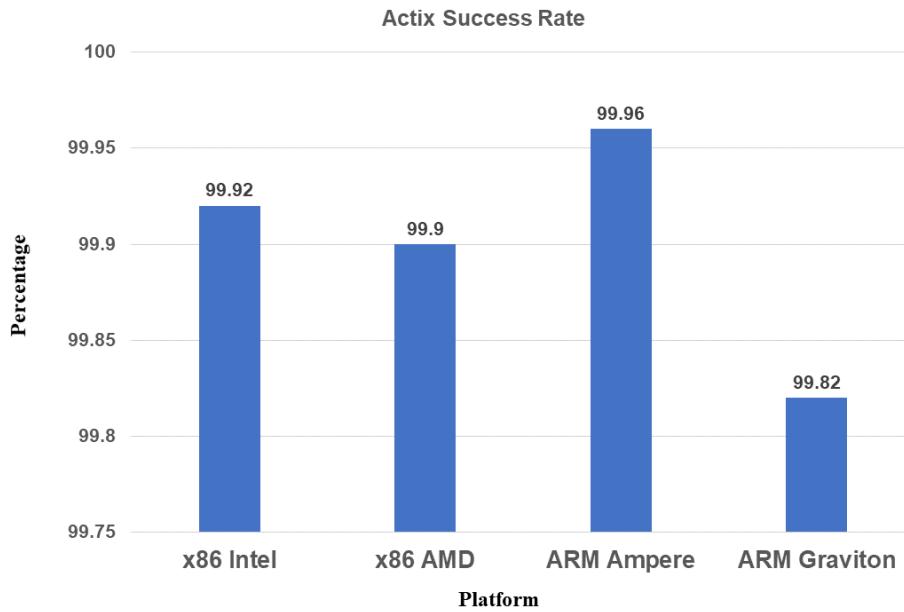


Figure 5.9: Actix Success Rate

From figure 5.9, here in Actix success rate the Ampere got 99.96 percent which is the highest one. Later Intel, AMD and Graviton comes second, third and fourth respectively.

5.2.5 I/O Results

In I/O GCP platform gave us better performance. Here Intel and AMD have the Lead. Graviton's performance is not far behind. Amerpe's performance is the lowest here.

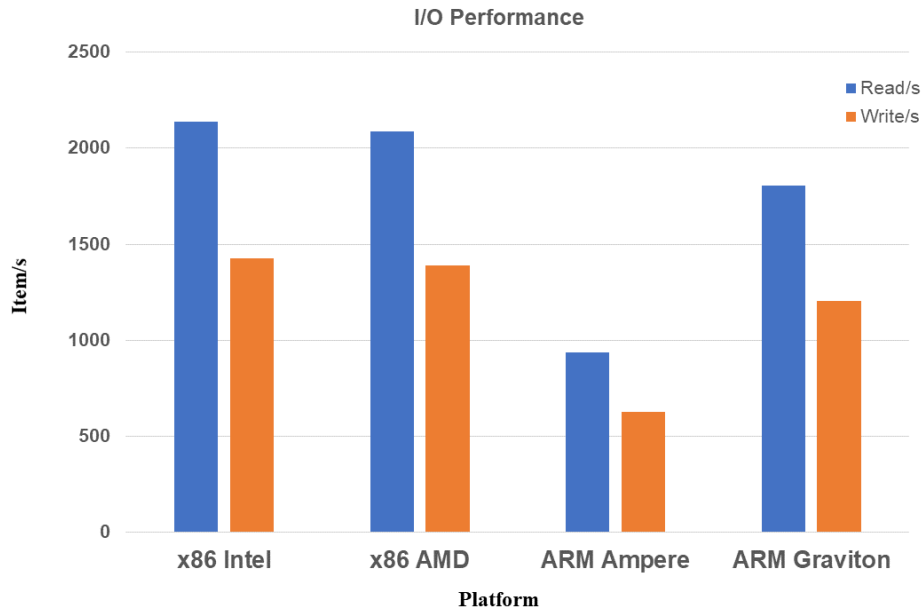


Figure 5.10: I/O Performance

In this chart, we find a different result where Ampere got the lowest value of performance where the read is less than 1000 items per second and write is above 500 items per second. Intel got the highest value of reading approximately above 2000 and writing is below 1500 items per second. Moreover, AMD and Graviton are second, and third respectively according to performance.

5.3 Experimental Findings

In most scenarios ARM had the upper hand. Express's performance on X86 was very poor compared to ARM. Graviton's and Ampere's performance was pretty much in the same ballpark. But Due to Ampere's low cost in price to performance ratio, Ampere always had a great advantage.

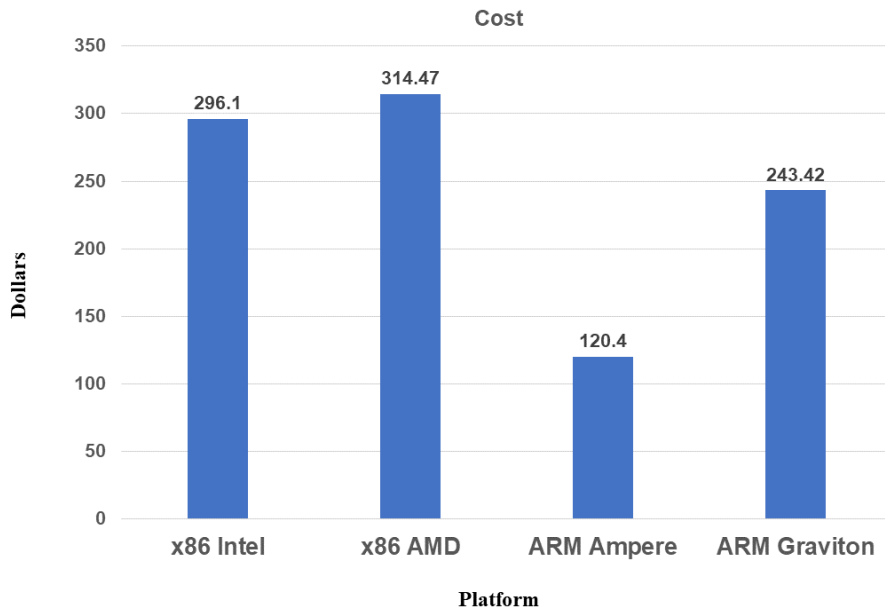


Figure 5.11: Cost Difference

In this figure 5.11, we can see that the most efficient platform is the Ampere, it cost only 120.4 dollars. On the other hand AMD platform has the highest cost around 314.47 dollars.

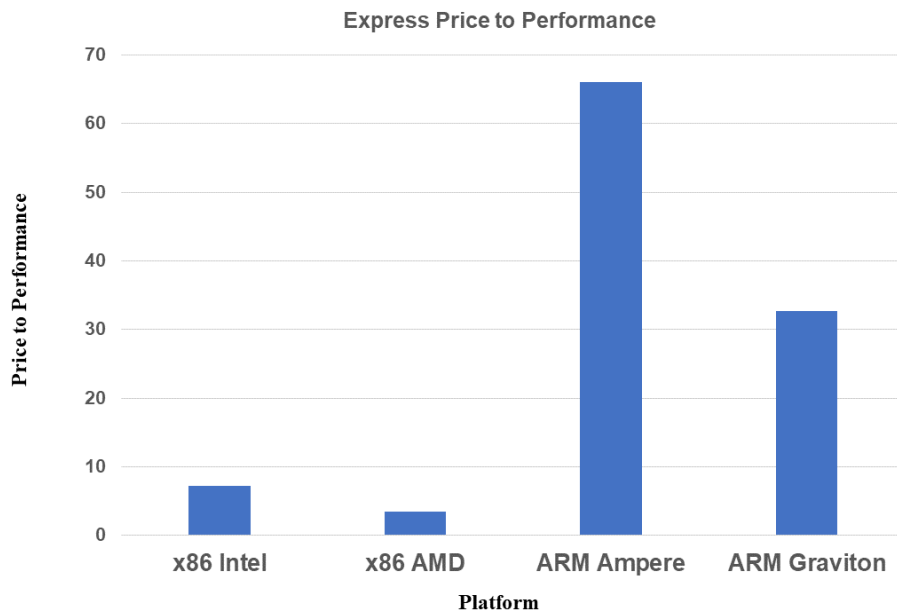


Figure 5.12: Express Price to Performance Ratio

From figure 5.12, In case of price to performance Ampere has the highest value approximately 70 while Graviton has the second highest value around 32. On the other hand AMD has the lowest value that is less than 5.

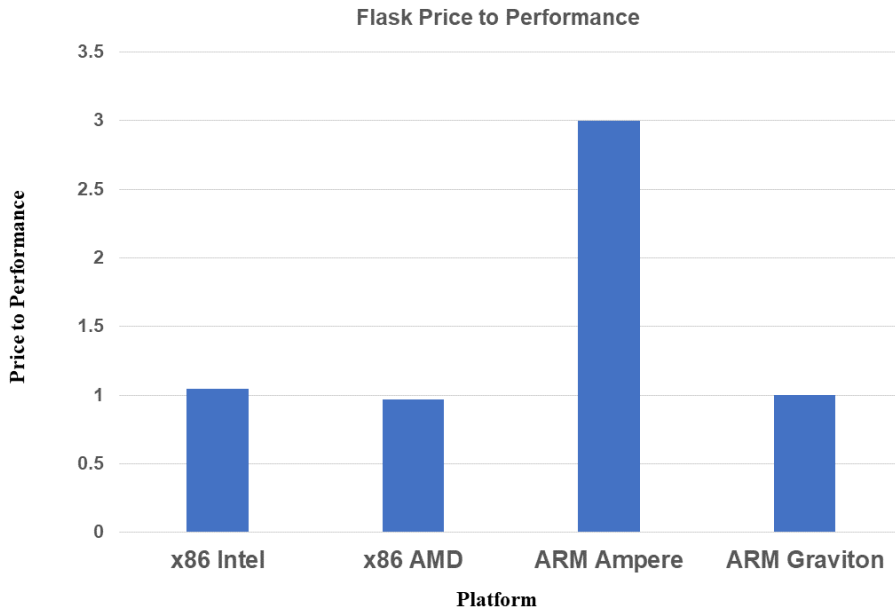


Figure 5.13: Flask Price to Performance Ratio

From figure 5.13, In the flask price to performance chart Ampere got the peak value of performance that is 3 while Intel is slightly higher than 1 and Graviton got the value exactly 1. On the other hand AMD has the lowest performance that is below 1.

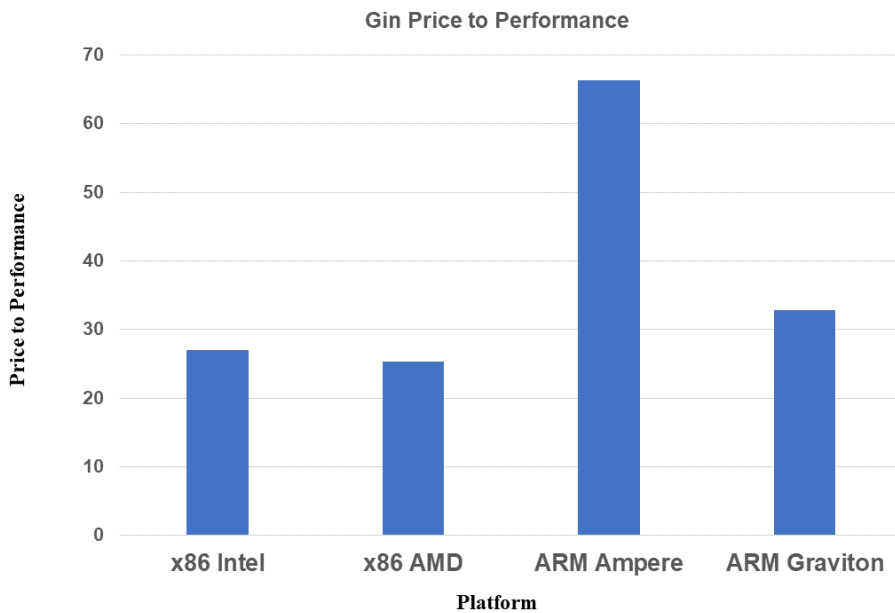


Figure 5.14: Gin Price to Performance Ratio

From figure 5.14, In this figure Ampere has the highest performance approximately 65 and Graviton has the second highest performance which is below 35 while the other two Intel, AMD got the value below 30.

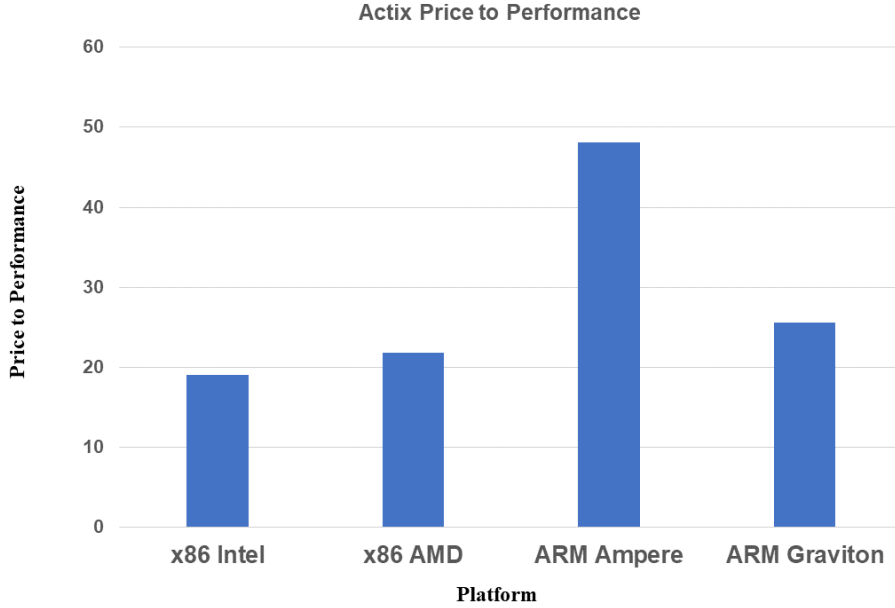


Figure 5.15: Actix Price to Performance Ratio

From figure 5.15, In the above figure we will find Ampere having the highest performance value which is near to 50. Graviton and AMD got the value that is above 20 respectively while Intel has got the lowest performance value which is below 20.

5.3.1 Theoretical Comparative Study

Name	Benchmark		Cloud		Highly Available Cluster	Platform	
	Synthetic	Custom Built	Public	Private		x86	ARM
Our Methodology	Yes	Yes	Yes		Yes	Yes	Yes
A Performance Evaluation of Containers running on Managed Kubernetes Services	Yes		Yes		Yes	Yes	
Self-Hosted Kubernetes: Deploying Docker Containers Locally with MINIKUBE				Yes		Yes	
A Comparison of Kubernetes and Kubernetes-Compatible Platforms	Yes		Yes		Yes	Yes	
Design and Deployment of Kubernetes Cluster on Raspberry Pi OS				Yes			Yes

Table 5.4: Comparison

From the above table we can see that we have used Synthetic and custom build benchmark tools. And our cloud service was public and highly cluster was available and we have tested on x86 and ARM architectures. But others did not use these benchmark tools both, their cloud service was private and more importantly they did not test it in x86 and ARM both architectures. Which gives our methodology of testing more accuracy than others. Finally, we can say that our result that we presentedd on this paper is much more realistic than others and it will help the developer community in real time.

Chapter 6

Discussion

6.1 Overview

Our thesis title is "Kubernetes performance analysis on different architectures." Here we tried our best to come up with real-world results which we have gathered from testing a couple of Kubernetes services offered by different organizations based on different architectures. Here we have tested the x86 platform both on Intel and AMD, ARM platform in Ampere, and Graviton also. Where we have tested them using Express JS, Gin, Flask, Actix, etc. First, we had to write API in the mentioned framework. We had to connect with our cluster and then create pods and services to access our application. Then we called our API with a client simulator that will create a massive number of requests which will stress test the system. Then based on the result we have reached our final verdict.

6.2 Limitations

There are a few limitations in our paper

1. We could not check every single major CPU.
2. We could not check every single cloud service provider.
3. Different cloud providers use different virtualization technology, different storage speeds, etc. So the same CPU can show different performance.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

From the experiment we can see that ARM's performance is greater than most x86 in most scenarios. It's because ARM is comparatively new architecture and has better instructions. But in the industry the use of x86 is widespread. So, a lot of businesses use x86. The mass transition of ARM is still on the way. When we start considering cost, ARM becomes a clear winner. ARM's cost is low to begin with and Cloud providers like Oracle are providing their Ampere platform at a very cheap rate compared to the competition. They are providing this service at this rate to promote their product. Even the most complex ARM chips are orders of magnitude (or two) simpler than the most basic contemporary X86 CPUs, making ARM significantly less expensive. They are widely produced by many different businesses under license, in contrast to X86, where two companies essentially control the market, which is possibly relevant. Most ARM processors can be used without spending a fortune. ARM processors are easier to develop and frequently considerably smaller than other CPUs because of their RISC architecture, which is less complex. For a large corporation who need highly available, highly scalable and high performing Kubernetes to run its infrastructure, ARM is a great choice for them as it is cheap and energy efficient.

7.2 Future Plans

Furthermore, plenty of room is available to extend this study. In the future, we can include more diverse benchmarking to narrow down each architecture's strength and weakness. To test the performance, we chose the Google Cloud Platform, although there are many other platforms as well, including Amazon Web Services (AWS), Alibaba, Microsoft Azure, and IBM Bluemix. In order to advance the research, we can also evaluate how well these various platforms performs.

References

- [1] J. D. Brock, R. F. Bruce, and M. E. Cameron, “Changing the world with a raspberry pi,” *Journal of Computing Sciences in Colleges*, vol. 29, no. 2, pp. 151–153, 2013.
- [2] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE cloud computing*, vol. 1, no. 3, pp. 81–84, 2014.
- [3] L. Malhotra, D. Agarwal, A. Jaiswal, *et al.*, “Virtualization in cloud computing,” *J. Inform. Tech. Softw. Eng*, vol. 4, no. 2, pp. 1–3, 2014.
- [4] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.
- [5] C. Boettiger, “An introduction to docker for reproducible research,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.
- [6] T. Bui, “Analysis of docker security,” *arXiv preprint arXiv:1501.02967*, 2015.
- [7] B. Russell, “Passive benchmarking with docker lxc,” *KVM & OpenStack*, 2015.
- [8] T. Vase, “Integrating docker to a continuous delivery pipeline: A pragmatic approach,” 2016.
- [9] P. Czarnul, “Benchmarking performance of a hybrid intel xeon/xeon phi system for parallel computation of similarity measures between large vectors,” *International Journal of Parallel Programming*, vol. 45, no. 5, pp. 1091–1107, 2017.
- [10] G. Sayfan, *Mastering kubernetes*. Packt Publishing Ltd, 2017.
- [11] L. P. Dewi, A. Noertjahyana, H. N. Palit, and K. Yedutun, “Server scalability using kubernetes,” in *2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON)*, IEEE, 2019, pp. 1–4.
- [12] A. Pereira Ferreira and R. Sinnott, “A performance evaluation of containers running on managed kubernetes services,” in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2019, pp. 199–208. DOI: 10.1109/CloudCom.2019.00038.
- [13] N. Hajdarbegović, *Apple m1 overview and compatibility*, Dec. 2020. [Online]. Available: <https://www.toptal.com/apple/apple-m1-processor-compatibility-overview>.
- [14] Z. He, “Novel container cloud elastic scaling strategy based on kubernetes,” Jun. 2020, pp. 1400–1404. DOI: 10.1109/ITOEC49072.2020.9141552.

- [15] S. Jain, *Advantages and disadvantages of arm processor*, Jul. 2020. [Online]. Available: <https://www.geeksforgeeks.org/advantages-and-disadvantages-of-arm-processor/>.
- [16] A. Pellegrini, N. Stephens, M. Bruce, *et al.*, “The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc,” *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.
- [17] S. Afreen, *Docker vs. vm (virtual machine): Key differences you need to know: Simplilearn*, Dec. 2021. [Online]. Available: <https://www.simplilearn.com/tutorials/docker-tutorial/docker-vs-virtual-machine>.
- [18] K. Gupta and T. Sharma, “Changing trends in computer architecture: A comprehensive analysis of arm and x86 processors,” 2021.
- [19] Stan, *5 reasons to build a raspberry pi kubernetes cluster for your homelab*, Feb. 2021. [Online]. Available: <https://turingpi.com/5-reasons-to-build-a-raspberry-pi-kubernetes-cluster-for-your-homelab/>.
- [20] L. Szustak, R. Wyrzykowski, L. Kuczynski, and T. Olas, “Architectural adaptation and performance-energy optimization for cfd application on amd epyc rome,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, pp. 2852–2866, 2021.
- [21] S. Telenyk, O. Sopov, E. Zharikov, and G. Nowakowski, “A comparison of kubernetes and kubernetes-compatible platforms,” in *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, IEEE, vol. 1, 2021, pp. 313–317.
- [22] M. H. Todorov, “Design and deployment of kubernetes cluster on raspberry pi os,” in *2021 29th National Conference with International Participation (TELECOM)*, IEEE, 2021, pp. 104–107.
- [23] E. K.-0.-0. 11:24 and E. Kisller, *A beginner’s guide to understanding and building docker images*, Jun. 2022. [Online]. Available: <https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/>.
- [24] C. Baird, *A primer on cloud computing*, Mar. 2022. [Online]. Available: https://medium.com/@colinbaird_51123/a-primer-on-cloud-computing-9a34e90303c8.
- [25] *Docker overview*, Sep. 2022. [Online]. Available: <https://docs.docker.com/get-started/overview/>.
- [26] J. Hopkins, *What is an arm processor? comparison to x86 and its advantages and disadvantages*, Mar. 2022. [Online]. Available: <https://www.totalphase.com/blog/2022/03/what-is-arm-processor-comparison-x86-and-advantages-disadvantages>.
- [27] R. Shivalkar, *The most popular kubernetes alternatives and competitors*, Apr. 2022. [Online]. Available: <https://www.clickittech.com/devops/kubernetes-alternatives/>.
- [28] *Pods*. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/pods/>.

- [29] D. Technologies, *Poweredge server solutions*. [Online]. Available: <https://www.dell.com/en-us/dt/servers/index.htm>.
- [30] *Understanding docker images and layers*. [Online]. Available: <https://subscription.packtpub.com/book/application-development/9781788992329/1/ch01vl1sec05/understanding-docker-images-and-layers>.
- [31] *Virtualization: Basic concept*. [Online]. Available: <https://documentation.suse.com/smart/linux/html/concept-virtualization/index.html>.