

PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
Ministry of Higher Education and Scientific Research

University May 8, 1945 Guelma

Faculty of Mathematics, Computer Science and Material Sciences

Computer science department



MASTER THESIS

Branch : Computer science

Option : Computer science systems

Theme

**A containerized management for deploying machines in the
computing center of the computer science
department-University of Guelma 8 may 1945**

Supervised by:

Dr. Benamira Adel

Presented by:

Karki Ahmed Raid

Examiner: Dr. Tadjer Houda

Chairman: Dr. Aggoune Aicha

June 2024

*I dedicate this thesis to my beloved parents,
for their endless support, and encouragement.
To my siblings Omar, Abdelhakim, Amin, Kawther and
Soundouss , for their understanding and inspiration.
To my friends Chouaib, Khayrou, for their unwavering
support and camaraderie.
And to my professors and mentors and colleagues of IT
department,
for their guidance and invaluable knowledge.*

Thank you all for being a part of this journey.

Acknowledgments

Alhamdulillah, for making this achievement possible. I am deeply thankful for the abilities I have been granted. I extend my heartfelt gratitude to my supervisor, **Dr. Benamira**, for his invaluable guidance, encouragement, and patience. His kindness and insightful advice have been crucial in pushing me forward .

I am profoundly grateful to my family for their unwavering support—my *parents and brothers, aunts, uncles* who have been my strength and motivation throughout this journey.

I extend my heartfelt gratitude to all the teachers and professors of the IT department for their invaluable lessons, knowledge, and patience throughout my five years of study. Their dedication to imparting knowledge has been instrumental in shaping my understanding and skills in the field

I also wish to acknowledge the Faculty of Mathematics, Computer Science, and Material Sciences at University May 8, 1945 Guelma for providing the academic environment and resources essential for my education.

I am deeply thankful to my new friends whom I met by chance during my internship at JVGAS association, especially *Belkhier, Tarek, Hichem and Mouhamed*. Their kindness, encouragement, and unwavering support have made a significant impact on my journey.

Finally, I express my gratitude to all those who have supported and inspired me along the way, contributing to my personal and academic growth.

Abstract

This thesis project endeavors to develop an innovative platform tailored for practical sessions within the educational context. The platform aims to provide customized environments for each teaching module through the implementation of technological containers. These containers are designed to ensure robust stability and secure isolation of environments, facilitating seamless access to essential tools and libraries for both teachers and students. Additionally, the platform seeks to automate the management processes of the educational computing center, streamlining administrative tasks for system administrators.

Keywords: Linux, Docker, Kubernetes, DevOps, Containers

Résumé

Ce projet de thèse vise à développer une plateforme innovante destinée aux sessions pratiques dans le cadre éducatif. La plateforme vise à fournir des environnements spécifiques pour chaque module d'enseignement grâce à l'utilisation de conteneurs technologiques. Ces conteneurs sont conçus pour assurer une stabilité robuste et une isolation sécurisée des environnements, facilitant l'accès transparent aux outils et bibliothèques essentiels tant pour les enseignants que pour les étudiants. De plus, la plateforme vise à automatiser les processus de gestion du centre informatique éducatif, simplifiant les tâches administratives pour les administrateurs système.

Mots Clés : Linux, Docker, Kubernetes, Conteneurs

Contents

List of Figures	i
List of Tables	iii
General Introduction	1
1 Virtualization technology	2
1.1 Introduction	2
1.2 Virtualization	2
1.3 Historic of virtualization	3
1.4 Hypervisors	3
1.5 Linux and virtualization	5
1.6 Types of virtualizations	6
1.7 Conclusion	9
2 Containerization technology	10
2.1 Introduction	10
2.2 Overview of containerization technology	10
2.3 Containerization versus virtualization	11
2.4 Containerization debut	13
2.5 Key base of containerization	15
2.5.1 Control groups (cgroups)	15
2.5.2 Namespaces	17
2.5.3 Filesystem rootfs	18
2.6 Containerization engines	19
2.7 Docker	19
2.7.1 Docker architecture	20
2.7.2 Docker components	21
2.7.3 Docker Objects	24

2.8	Container orchestration	26
2.8.1	Container orchestration benefits	26
2.8.2	Orchestrations tools	27
2.8.3	What is Kubernetes	28
2.8.4	How Is Kubernetes Different from Docker Swarm	28
2.8.5	Why we use Docker and Kubernetes	29
2.8.6	How Did Kubernetes Come into Existence	30
2.8.7	Kubernetes architecture	31
2.9	Features of kubernetes	33
2.10	Conclusion	35
3	Conception	36
3.1	Introduction	36
3.2	Problem Statement	36
3.2.1	Description of the Problem:	36
3.2.2	Background and use cases:	37
3.2.3	Objectives:	39
3.3	Requirements Analysis	40
3.4	System Architecture	41
3.4.1	High-Level Architecture:	41
3.4.2	Architecture design:	43
3.5	UML Diagrams	45
3.5.1	Class diagram	45
3.5.2	Use Case Diagram	46
3.5.3	Database schema	46
3.6	Conclusion	48
4	Implementation	49
4.1	Introduction	49
4.2	Setting Up the Environment	49
4.2.1	Kubernetes cluster	50
4.2.2	Creating Virtual Machines on Microsoft Azure	51
4.3	Developing classroom companion	55
4.3.1	Technologies and Tools	55
4.3.2	Functionality	57
4.4	Conclusion	62

General conclusion	63
Bibliography	65

List of Figures

1.1	Difference between physical compute and virtual	3
1.2	Hypervisor type 01 schema	4
1.3	Hypervisor type 02 schema	5
1.4	Application virtualisation schema	6
1.5	General Architecture Of Network Virtualization	7
1.6	Desktop virtualization	7
1.7	Storage virtualization	8
2.1	Virtual machines and containers architectures	12
2.2	Cgroups architecture [1]	16
2.3	Docker architecture [2]	20
2.4	Docker daemon [3]	21
2.5	Docker desktop GNU [4]	23
2.6	Docker compose role [5]	24
2.7	Image layers [6]	25
2.8	Container process [7]	26
2.9	Kubernetes architecture [8]	31
2.10	Vertical scaling vs Horizontal scaling [9]	33
3.1	Packages conflict figure	39
3.2	High-level architecture	41
3.3	Core architecture	42
3.4	General architecture of the environment	43
3.5	Class diagram figure	45
3.6	use case diagram	46
3.7	Database schema	47
4.1	Kubernetes cluster	55
4.2	Admin upload teachers into database	57

- 4.3 Admin upload Subjects into database 57
- 4.4 Admin assign subjects to teacher 58
- 4.5 Authentication interface 59
- 4.6 Teacher dashboard 59
- 4.7 Result of searching for tools 60
- 4.8 interface for subject environment 60
- 4.9 Kubernetes cluster with environment 61

List of Tables

2.1	Comparison between Virtual Machines and Containers	13
2.2	Container technology timeline	14
2.3	Comparison of Docker Swarm and Kubernetes [10]	29
4.1	resources requirements of the kubernetes cluster	51

General Introduction

In the digital age of education, where technology is increasingly integrated into teaching and learning processes, the role of educational computing infrastructure has become paramount. Providing students with tailored environments for practical modules is fundamental for fostering effective learning experiences. However, this endeavor is not without its challenges.

Machines are shared across multiple modules, each having its own development environment, resulting in unstable machine states, and requiring frequent reconfiguration. To tackle this, our solution leverages container technology, a rapidly evolving field, offering complete isolation of software dependencies, including those of the operating system.

The proposed platform goes beyond stability; it automates the entire process of managing the educational computing center. Through containerization, each instructor is equipped with a personalized environment that automatically initiates upon login to a master machine. This not only enhances efficiency but also relieves administrators of the burdensome tasks associated with manual reconfiguration. In essence, our project aims to revolutionize the way educational computing resources are managed, providing a seamless and stable experience for both educators and students alike.

This thesis is organized as follows: The first chapter delves into virtualization technology, exploring its foundational concepts and its significance in modern computing environments, particularly in educational settings. The second chapter examines containerization technology, detailing how it builds on virtualization principles to offer isolated and efficient environments. In the third chapter, we outline the design and architectural considerations of our proposed solution, including detailed diagrams and conceptual frameworks. Finally, the fourth chapter covers the practical implementation of our project, discussing the steps taken, challenges faced, and results achieved, along with the deployment and management of the containerized environments.

Chapter 1

Virtualization technology

1.1 Introduction

The field of computer science is constantly evolving at a rapid pace, necessitating continuous adaptation to the latest technological advancements. Over recent years, the industry has seen the emergence of groundbreaking technologies like distributed computing, parallel processing, virtualization, cloud computing, and most recently, the Internet of Things (IoT). Each of these innovations has paved the way for subsequent developments, contributing to a robust foundation for further progress. For instance, virtualization has played a transformative role and laid the groundwork for cloud computing.

Since the inception of computers, maximizing resource utilization has been a common practice, whether through time sharing, multitasking, or more recent trends in virtualization.

1.2 Virtualization

In philosophy, "virtual" refers to something that lacks physical reality; however, in computer science, "virtual" denotes a simulated hardware environment. In this context, we replicate the functionalities of physical hardware and present them to an operating system. The technology employed to achieve this environment is termed virtualization technology, or simply, virtualization. The physical system that hosts the virtualization software, such as a hypervisor, is referred to as a host, while the virtual machines running atop the hypervisor are termed guests [11]

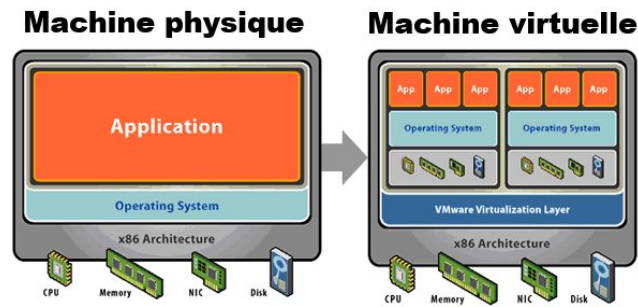


Figure 1.1: Difference between physical compute and virtual compute [12]

1.3 Historic of virtualization

The concept of virtualization dates back to the 1960s when IBM developed virtual machine (VM) technology for its mainframe computers. [13]

IBM's VM/370, introduced in the early 1970s, was one of the earliest implementations of virtualization technology. It allowed multiple instances of IBM's operating system (OS/370) to run concurrently on a single mainframe, effectively partitioning the hardware resources and providing isolated execution environments for different users. In the 1990s, as many companies faced the challenge of maintaining their single-vendor IT stacks and existing applications, they became aware of the need to make better use of their often underutilized server resources. By adopting virtualization, they could not only more efficiently partition their server infrastructure but also run their legacy applications on different types and versions of operating systems. Due to its vast network composed of many types of computers running on different operating systems, the Internet contributed to the adoption of virtualization. [14]

1.4 Hypervisors

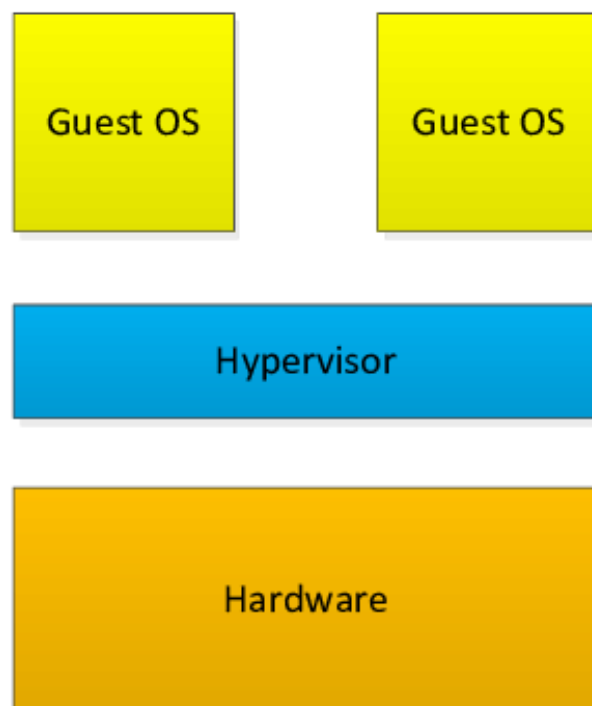
A hypervisor, also known as a virtual machine monitor (VMM), is a software layer that allows multiple virtual machines (VMs) to run on a single physical machine, known as the host system. The primary purpose of a hypervisor is to abstract and virtualize the underlying physical hardware, such as the CPU, memory, storage, and networking resources, so that they can be shared among multiple VMs. A hypervisor enables the simultaneous execution of multiple operating systems on the host hardware. These instances can be utilized by various users, highlighting the hypervisor's capability to facilitate multi-tenancy. Additionally, hypervisors ensure isolation between different guest processes, a crucial factor in supporting multi-tenancy. Furthermore, hypervisors can manage proces-

sor changes transparently, without impacting the user's operating system or applications, thus offering essential agility for cloud infrastructure [13]

Types of hypervisors

- Type 1 Hypervisor (Bare-Metal Hypervisor)

This hypervisor runs directly on the physical hardware without the need for an underlying operating system. It manages the hardware resources and provides virtualization services directly to the guest VMs. Examples include VMware ESXi, Microsoft Hyper-V Server, and Xen.



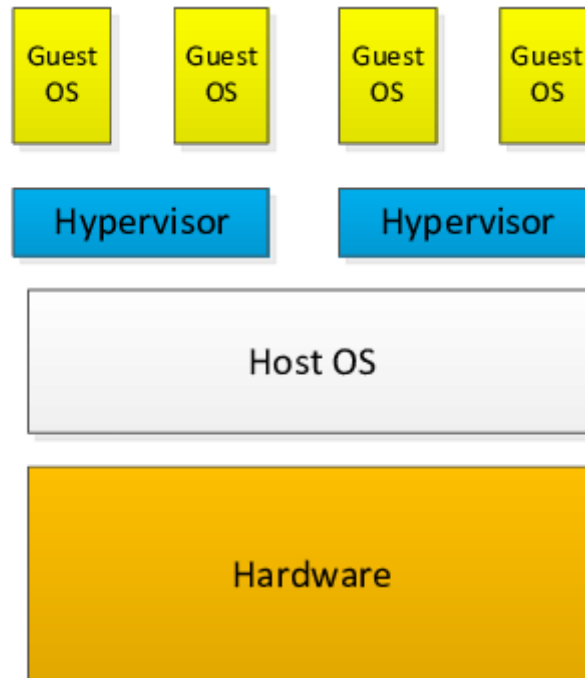
Type 1 (bare-metal)

Figure 1.2: Hypervisor type 01 schema [15]

- Type 2 Hypervisor (Hosted Hypervisor):

This hypervisor runs on top of a conventional operating system installed on the physical hardware. It relies on the underlying operating system to manage hardware

resources and provides virtualization services through software. Examples include VMware Workstation, Oracle VirtualBox, and Parallels Desktop for Mac.



Type 2 (hosted)

Figure 1.3: Hypervisor type 02 schema [15]

1.5 Linux and virtualization

Virtualization made its debut in Linux with User-Mode Linux (UML), igniting the momentum needed to position Linux as a contender in the virtualization landscape. Presently, Linux offers a diverse range of virtualization solutions that enable the transformation of a single computer into multiple virtual machines. Prominent among these solutions are KVM, Xen, and QEMU. The allure of Linux virtualization lies in its openness, flexibility, and performance capabilities. Similar to other open-source software, Linux virtualization solutions are developed collaboratively, leveraging the advantages of the open-source model. This collaborative approach fosters wider community input, ultimately leading to reduced research and development costs, enhanced efficiency, and improved performance and productivity. Moreover, the open-source model inherently fosters innovation, driving

continuous improvement within the Linux virtualization ecosystem [11]

1.6 Types of virtualizations

In simple terms, virtualization involves the abstraction of various components such as hardware, network, storage, applications, and access. This means that virtualization can be applied to any of these components. Specifically, virtualization entails concealing the underlying physical hardware to enable its sharing and utilization by multiple operating systems. This process, also known as platform virtualization, introduces an intermediary layer called a hypervisor or Virtual Machine Monitor (VMM) between the underlying hardware and the operating systems running on it. The operating system running atop the hypervisor is referred to as the guest or virtual machine (VM).

Application virtualization

This enables users to virtually access an application, where the server retains all the application's data and features, enabling remote execution via the internet. Application virtualization streamlines the deployment of applications within an environment, facilitating efficient upgrades and support processes [16]

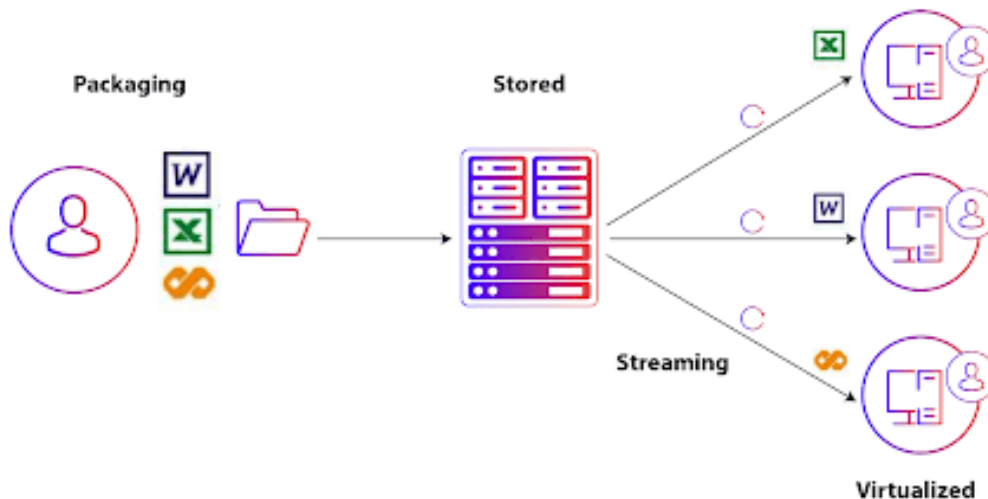


Figure 1.4: Application virtualisation schema [16]

Network virtualization

This permits the operation of numerous virtual networks, each possessing distinct control and data plans. Network virtualization offers the capability to swiftly generate and

allocate virtual networks, encompassing logical switches, routers, firewalls, load balancers, Virtual Private Networks (VPNs), and workload security, all in minimal time [17]

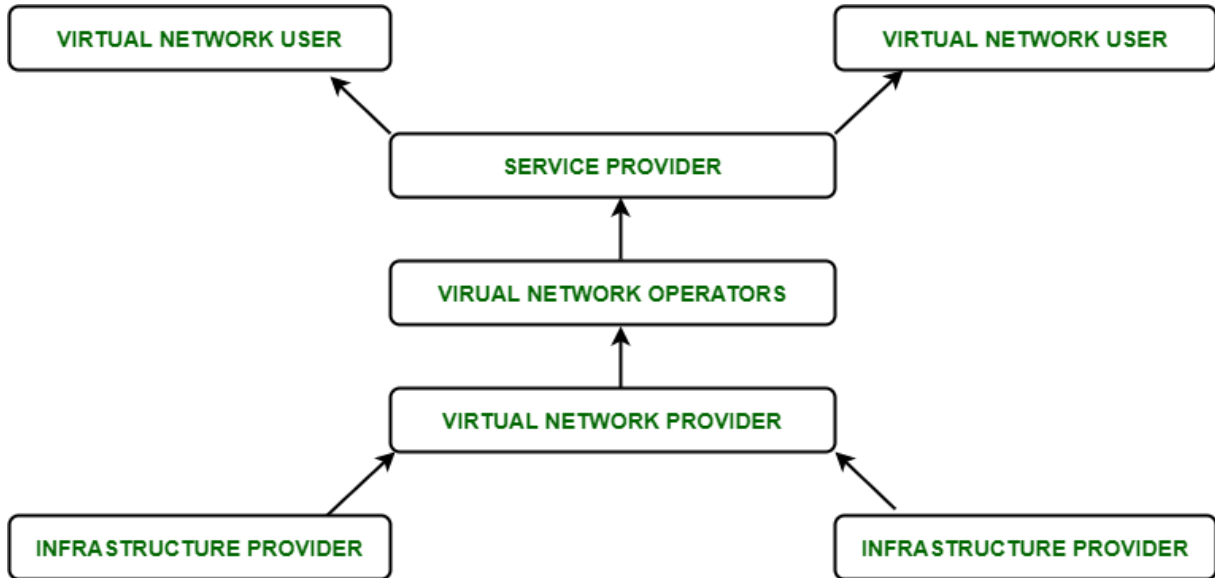


Figure 1.5: General Architecture Of Network Virtualization [17]

Desktop virtualization

This enables users to remotely access their desktops. Those desiring to utilize operating systems other than Windows Server can effectively do so through desktop virtualization. Key advantages include enhanced portability, simplified software installation, and patch management [17]

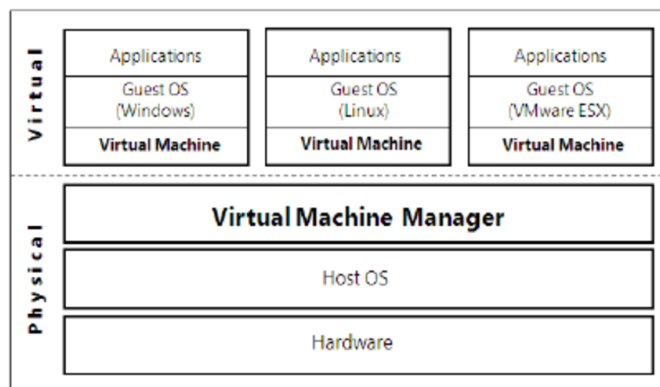


Figure 1.6: Desktop virtualization [17]

Storage virtualization

This feature offers a virtual storage solution that is secure, isolated, and reliable, spanning multiple networks. It can accomplish all this using just a single storage component. Even in the event of failures in the underlying systems, storage virtualization guarantees seamless and uninterrupted operation [17]

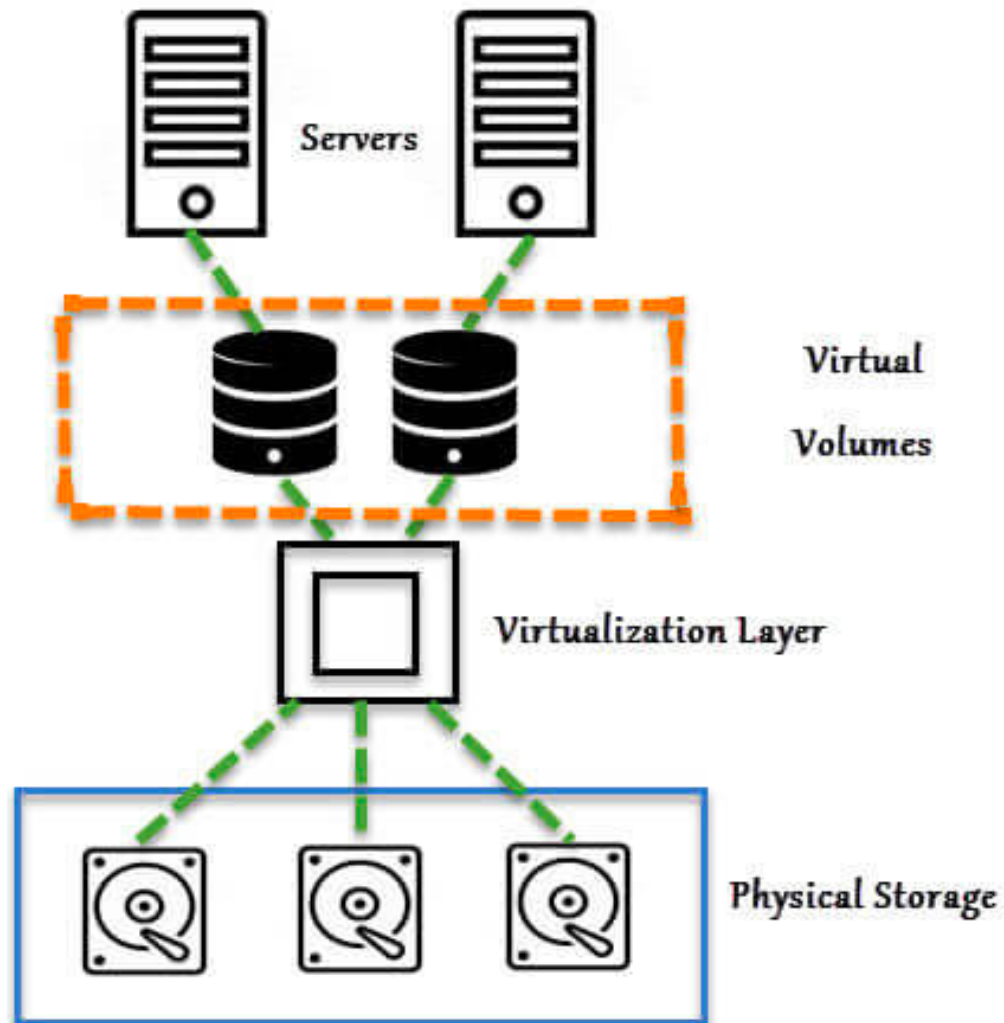


Figure 1.7: Storage virtualization

Server virtualization

This form of virtualization involves concealing server resources, where the primary server (physical server) is partitioned into multiple distinct virtual servers by altering identifiers and processors. Consequently, each system can function with its own operating system in isolation, with each sub-server recognizing the central server's identity. This approach enhances performance and diminishes operational expenses by distributing main server

resources into sub-server resources. It proves advantageous for tasks like virtual migration, energy conservation, and minimizing infrastructure expenses [17]

Data virtualization

This type of virtualization involves gathering data from diverse sources and centralizing it in a single location without delving into technical intricacies such as data collection, storage, and formatting. The data is then logically organized to create a virtual representation accessible to interested parties, stakeholders, and users via various cloud services from remote locations. Numerous major companies, including Oracle and IBM, offer such services [17]

1.7 Conclusion

In conclusion, virtualization technology has emerged as a powerful tool for optimizing resource utilization, enhancing flexibility, and improving scalability in computing environments. Through the creation of virtual instances of hardware, software, or storage resources, virtualization enables efficient allocation and management of IT resources. It facilitates the deployment of multiple operating systems and applications on a single physical machine, leading to cost savings and increased efficiency. Moreover, virtualization plays a crucial role in disaster recovery, high availability, and workload migration strategies. As organizations continue to embrace cloud computing and data center consolidation initiatives, virtualization will remain a cornerstone technology in modern IT infrastructures, driving innovation and enabling agility in the face of evolving business needs

Chapter 2

Containerization technology

2.1 Introduction

In the preceding chapter, we embarked on a comprehensive exploration of virtualization technology, dissecting its core principles, applications, and implications within computing environments. We navigated through the complexities of managing shared server resources, optimizing infrastructure partitioning, and leveraging the versatility of running diverse operating systems simultaneously on a single physical machine. As we delved into the nuances of virtualization, we gained valuable insights into its transformative potential across various domains. In this chapter, we shift our focus to a groundbreaking technology poised to revolutionize the landscape of computing infrastructure: containerization. Unlike traditional virtualization methods, containerization offers a lightweight, efficient approach to application deployment and management. By encapsulating applications and their dependencies into portable, self-contained units known as containers, containerization enables seamless deployment across diverse computing environments, from local development environments to production servers.

2.2 Overview of containerization technology

Containerization is a technology that helps package up software and its dependencies so it can run consistently across different computing environments. Imagine a container like a self-contained box that holds everything an application needs to run smoothly. This includes the application itself, along with any software libraries and settings it relies on. One of the key benefits of containerization is that it abstracts away the complex details of the underlying infrastructure. This means developers can focus on writing code without worrying about the specific setup of the server or computer where the application will run.

Containers are lightweight and portable, making them easy to move between different environments, like from a developer's laptop to a production server. They also start up quickly and use resources efficiently, which can help speed up deployment times and save on computing costs. Overall, containerization has become a popular choice for software development and deployment because it simplifies the process of building, shipping, and running applications. It's like a modern-day shipping container, revolutionizing the way software is packaged and delivered. [18]

2.3 Containerization versus virtualization

Containers are frequently likened to virtual machines (VMs) as they both contribute to notable computational efficiencies by enabling the execution of various software types (whether Linux- or Windows-based) within a unified environment. Nevertheless, container technology is demonstrating remarkable advantages beyond those offered by virtualization, swiftly garnering preference among IT professionals. Virtualization technology enables the concurrent operation of multiple operating systems and software applications, all utilizing the resources of a single physical computer. This means that an IT organization can run a mix of operating systems, like Windows and Linux, or multiple instances of the same operating system, alongside various applications on a single server. Each application, along with its associated files, libraries, and dependencies, including a complete copy of the operating system, is bundled together as a virtual machine (VM). By hosting multiple VMs on a single physical machine, substantial savings can be realized in terms of capital investment, operational expenses, and energy consumption

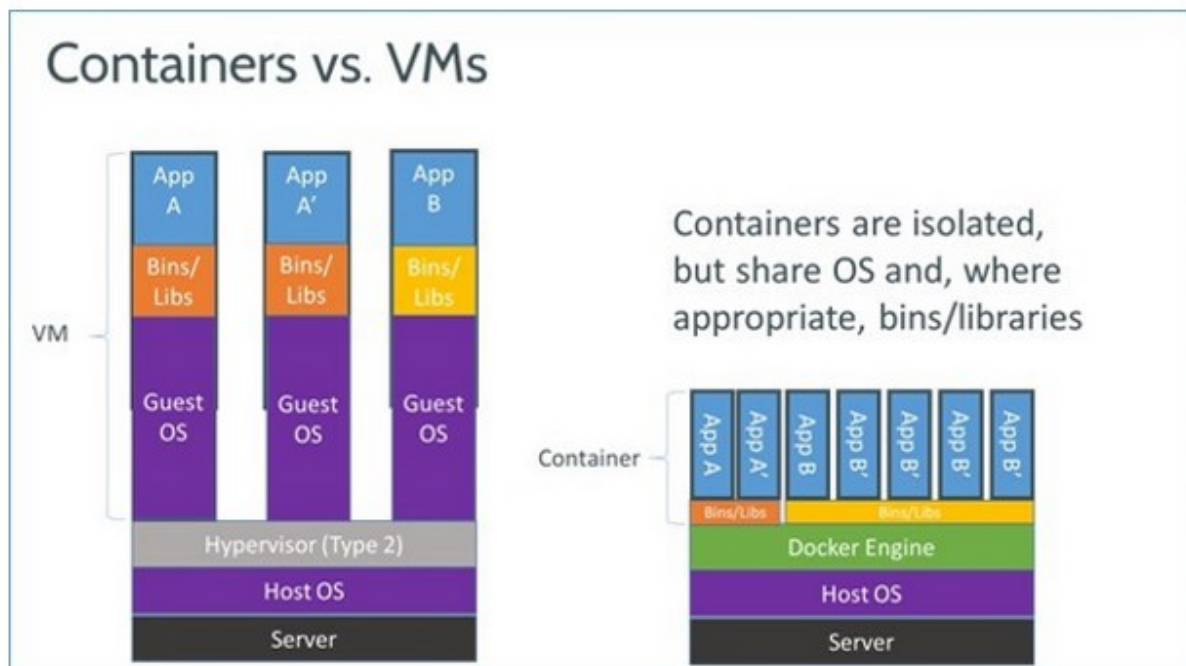


Figure 2.1: Virtual machines and containers architectures [19]

On the contrary, containerization maximizes the utilization of computing resources even further. A container consolidates all necessary components for running an application into a single executable package of software. This includes application code, configuration files, libraries, and dependencies. Unlike virtual machines (VMs), containers do not include a duplicate of the operating system (OS). Instead, the container runtime engine is installed on the host system's OS, serving as the mechanism through which all containers on the computing system access and share the same OS [20]

Parameter	Virtual Machines	Containers
Guest OS	Each VM runs on virtual hardware and Kernel is loaded into its own memory region	All the guests share same OS and Kernel. Kernel image is loaded into the physical memory
Communication	Will be through Ethernet Devices	Standard IPC mechanisms like signals, pipes, sockets, etc.
Security	Depends on the implementation of Hypervisor	Mandatory access control can be leveraged
Performance	Virtual Machines suffer from a small overhead as the machine instructions are translated from Guest to Host OS.	Containers provide near-native performance as compared to the underlying Host OS.
Isolation	Sharing libraries, files, etc. between guests and between guests hosts is not possible.	Subdirectories can be transparently mounted and can be shared.
Startup time	VMs take a few minutes to boot up	Containers can be booted up in a few seconds as compared to VMs.
Storage	VMs take much more storage as the whole OS kernel and its associated programs have to be installed and run	Containers take a lower amount of storage as the base OS is shared

Table 2.1: Comparison between Virtual Machines and Containers

As mentioned, containers are often described as "lightweight" because they share the operating system (OS) kernel of the host machine. Unlike virtual machines VMs, containers don't need to include a whole separate OS for each application. This means they use less space and can start up faster. Additionally, common bins and libraries can be shared among multiple containers, making them smaller and quicker to start than VMs. By running multiple containers on the same computing power as a single VM, servers can be used more efficiently, leading to lower costs for servers and software licenses.

2.4 Containerization debut

Virtualization emerged in response to the need to optimize existing computing resources, enabling multiple virtual machines to run on a single host, each with its isolated environment. Hypervisors, acting as mediators between the host and virtual machines, facilitate this isolation. Containerization, the next evolutionary step in virtualization, builds upon these principles, offering isolation at both the operating system and application levels. The concept of containers traces back to the late 1970s, originating from Unix operating systems with the introduction of chroot. However, modern container technology began

to take shape in the early 2000s, marking significant milestones in the evolution of containerization.

Year	Technology	First introduced in OS
1982	chroot	Unix-Like operating system
2000	jail	FreeBSD
2000	Virtuozzo containers	Linux, Windows
2001	Linux VServer	Linux, Windows
2004	Solaris container	Sun Solaris, Open Solaris
2005	Openvz	Linux
2008	LXC	Linux
2013	Docker	Linux, FreeBSD, Windows

Table 2.2: Container technology timeline

Several container technologies, as outlined in Table 2, serve distinct purposes. For instance, chroot offers filesystem isolation by altering the root directory for running processes and their offspring. Conversely, technologies like Solaris containers (zones) and LXC deliver comprehensive operating system-level virtualization. The lineage of many contemporary containers can be traced back to LXC, which debuted in 2008. LXC's emergence was made feasible by essential functionalities integrated into the Linux kernel from the 2.6.24 version onwards, as elaborated in the subsequent section. [21]

Here's a brief explanation of each

- Chroot

Chroot is a Unix system call that changes the apparent root directory for the current running process and its children. It's often used for creating isolated environments within a Unix-like operating system.

- Virtuozzo Containers

Virtuozzo Containers, formerly known as OpenVZ, is a commercial virtualization solution that provides container-based virtualization for Linux and Windows environments. It enables the creation of lightweight, isolated containers with shared kernel resources.

- Linux-VServer

Linux-VServer is an open-source operating system-level virtualization solution for Linux systems. It allows the creation of multiple isolated virtual environments, known as virtual private servers (VPS), on a single physical server

- Solaris Container

Solaris Containers, also known as Solaris Zones, are a form of operating system-level virtualization technology developed by Sun Microsystems (now Oracle). They enable the creation of multiple isolated environments, or zones, on a single Solaris host.

- OpenVZ:

OpenVZ is an open-source container-based virtualization solution for Linux systems. It provides lightweight, efficient virtualization by leveraging container technology to create isolated environments on a single physical server.

- LXC

Linux Containers (LXC) is an open-source operating system-level virtualization solution for Linux systems. It utilizes Linux kernel features such as namespaces and control groups (cgroups) to create and manage lightweight containers.

- Docker

Docker is a popular platform for building, shipping, and running containers. It simplifies the process of containerization by providing tools and services for creating, deploying, and managing containers, along with a centralized repository called Docker Hub for sharing container images

2.5 Key base of containerization

Containers leverage specific functionalities within the Linux kernel to establish an isolated environment within the host machine. This environment closely resembles a virtual machine but operates independently of a hypervisor.

- Control groups (cgroups)
- Namespaces
- Filesystem or rootfs

2.5.1 Control groups (cgroups)

Control groups (cgroups) are a Linux kernel feature exactly on the virtual file system VFS .it enables the management of system resources, such as CPU, memory, disk I/O, and

network bandwidth, among processes or groups of processes. Cgroups allow administrators to define resource constraints, prioritize resources, and isolate processes to prevent resource contention and ensure efficient resource utilization on a system. They provide a mechanism for organizing and controlling the resource usage of processes, making them a fundamental building block for containerization and resource management in modern operating systems. To understand the importance of cgroups, consider a common scenario: A process running on a system requests certain resources from the system at a particular instance, but unfortunately the resources are unavailable currently, so the system decides to defer the process until the requested resources are available. The requested resources may become available when other processes release them. This delays the process execution, which may not be acceptable in many applications. Resource unavailability such as this can occur when a malicious process consumes all or a majority of the resources available on a system and does not allow other processes to execute [21]. Containerization utilizes control groups (cgroups) in Linux to manage and limit the resource usage of containerized processes. Cgroups allow system administrators to allocate and control resources such as CPU, memory, disk I/O, and network bandwidth among different processes or groups of processes. By assigning containers to specific cgroups, containerization platforms can enforce resource constraints and priorities, ensuring that containers do not consume excessive resources and that system resources are fairly distributed among them. This helps in maintaining system stability, improving performance, and preventing resource contention in multi-tenant environments. Overall, cgroups play a vital role in enabling efficient resource management and isolation in containerized environments.

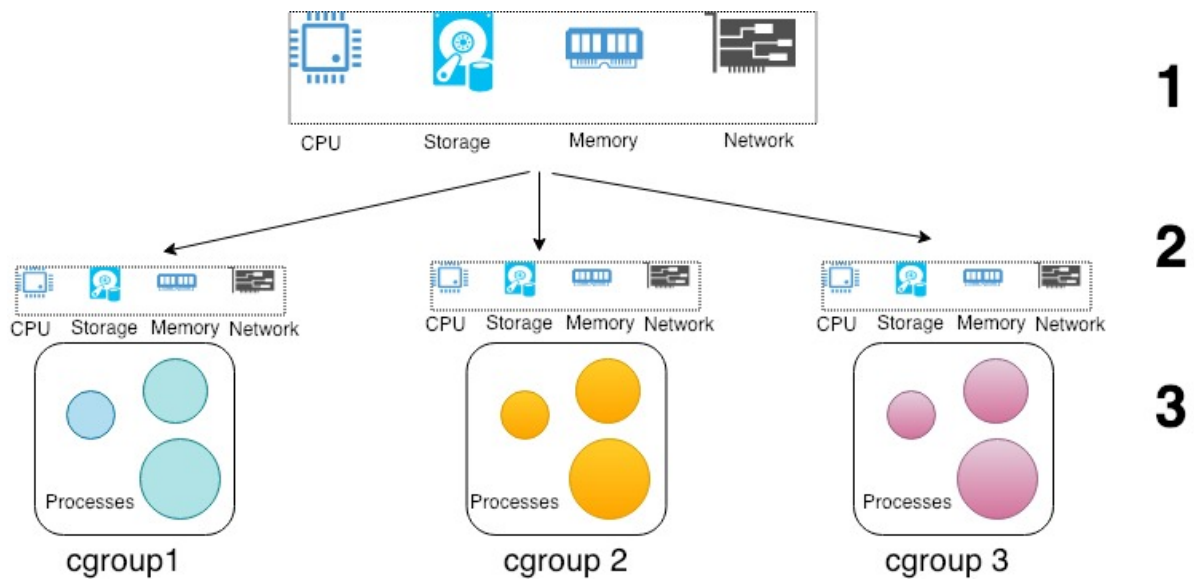


Figure 2.2: Cgroups architecture [1]

2.5.2 Namespaces

Namespaces in the context of operating systems, particularly in Linux, provide a way to isolate and virtualize system resources. They offer a mechanism for partitioning various kernel resources such as processes, network interfaces, mount points, and more, so that each partitioned environment appears as a separate instance to processes within it. Essentially, namespaces provide a form of resource abstraction, allowing processes within a namespace to have their own isolated view of certain system resources. This isolation enables the creation of lightweight and portable environments, commonly used in containerization, where applications and services can run independently without interfering with each other or the underlying host system. In simpler terms, namespaces act as a tool to create isolated environments within a single operating system, providing a level of separation and encapsulation for different sets of processes and resources. [21] containerization ensures that processes within a container have their own isolated view of these resources, making them appear as if they are running independently of other containers and the host system. For example, the PID namespace isolates process IDs, allowing processes within a container to have their own PID space without conflicting with processes outside the container. Similarly, the network namespace isolates network interfaces and routing tables, enabling containers to have their own independent network stack. Overall, namespaces play a crucial role in containerization by providing the necessary isolation to create lightweight and portable environments for running applications.

2.5.2.1 Type of namespaces

In the Linux kernel, there are several types of namespaces, each responsible for isolating specific sets of system resources. Here are some common types of namespaces

- PID Namespace

This namespace isolates the process ID number space. Processes in different PID namespaces can have the same PID, allowing for separate process trees.

- Network Namespace (net):

Network namespaces provide isolation for network resources such as network interfaces, routing tables, and firewall rules. Processes in different network namespaces have their own network stack and appear as separate instances on the network

- Mount Namespace (mnt)

Mount namespaces provide isolation for filesystem mount points. Processes in different mount namespaces have their own view of the filesystem hierarchy, allowing

for independent mount and unmount operations.

- UTS Namespace

UTS namespaces isolate system identification and hostname. Processes in different UTS namespaces can have different hostnames and domain names.

- IPC Namespace

IPC namespaces isolate inter-process communication resources such as message queues, semaphores, and shared memory segments. Processes in different IPC namespaces cannot communicate with each other using these resources.

- User Namespace

User namespaces provide isolation for user and group IDs. They allow processes to have their own set of user and group IDs, providing a level of privilege separation between processes.

- Cgroup Namespace

Cgroup namespaces isolate control groups, which are used for resource management and accounting. Processes in different cgroup namespaces have their own set of control groups, allowing for finer-grained resource control.

These namespaces are fundamental building blocks for containerization and provide the necessary isolation to create lightweight and portable environments.

2.5.3 Filesystem rootfs

the term "rootfs" refers to the initial filesystem that is mounted as the root filesystem inside a container. It serves as the starting point for all filesystem operations within the container environment. The root filesystem typically contains all the essential files, directories, and binaries required to boot and run a minimal Linux environment. This includes system binaries, libraries, configuration files, device nodes, and other necessary components. When a container is created, the root filesystem is usually populated from a base image, which serves as a template for the container's filesystem. This base image contains the initial set of files and directories needed to create a functional container environment. The rootfs is mounted as the root filesystem inside the container, isolating it from the host system's filesystem. This isolation ensures that the containerized application has its own independent filesystem environment, separate from other containers and the host system. Containerization leverages filesystem technology to create isolated environments

for applications and services. Each container typically has its own root filesystem, which is based on a base image containing the necessary files, directories, and dependencies required to run the application. Filesystem layers, such as those provided by Docker's Union File System (UFS), enable efficient storage and sharing of container images by allowing multiple layers to be stacked on top of each other. This layered approach allows for quick and lightweight container creation, as only the differences between layers need to be stored and transmitted. Additionally, containerization platforms provide mechanisms for persisting data generated by containers, such as volumes and bind mounts, ensuring that data is retained even if the container is destroyed or recreated. Overall, filesystem technology is fundamental to containerization, enabling the encapsulation, distribution, and execution of applications in isolated and portable environments.

2.6 Containerization engines

A container engine, also known as a container runtime, is a software tool responsible for managing containers on a host system. It provides the necessary runtime environment and infrastructure to create, run, and manage containers efficiently. Container engines handle tasks such as starting and stopping containers, managing their lifecycle, configuring networking, and interacting with container registries to pull container images. One of the most well-known container engines is Docker Engine, which played a significant role in popularizing containerization technology. Docker Engine includes various components such as the Docker daemon, containerd, and runc, which work together to manage containers. Other container engines include containerd, which serves as a core container runtime used by platforms like Docker and Kubernetes, as well as alternatives like cri-o, Podman, and rkt (pronounced "rocket"). These container engines offer similar functionalities but may differ in their implementation, architecture, and compatibility with different container orchestration platforms. In summary, a container engine is a critical component of the containerization ecosystem, providing the runtime environment necessary to create and manage containers effectively. It abstracts away the complexities of container management, allowing developers and system administrators to focus on deploying and running containerized applications.

2.7 Docker

Docker is a versatile platform designed to streamline the development, deployment, and management of applications. It empowers users to decouple their applications from the

underlying infrastructure, facilitating swift software delivery. Docker allows for the unified management of both applications and infrastructure, employing efficient methodologies for packaging, testing, and deploying code. By leveraging Docker’s tools and practices, organizations can minimize the time lag between code creation and production deployment, thereby enhancing the agility and efficiency of their software development processes. [2] Initially released on March 20, 2013, by Solomon Hykes, revolutionized the world of software development and deployment. Written in Go, Docker quickly gained popularity for its innovative approach to containerization, Docker continues to evolve, catering to the diverse needs of the development community. The project is open-source and maintained on GitHub under the repository github.com/moby/moby, reflecting Docker’s commitment to collaboration and transparency within the software ecosystem. Through its relentless focus on simplicity, efficiency, and scalability, Docker has become an indispensable asset for modern software development workflows [22]

2.7.1 Docker architecture

Docker operates with a setup where there’s a client side and a server side. The client, called Docker client, communicates with the server, known as Docker daemon. The daemon handles the main tasks of constructing, operating, and sharing Docker containers. Both the client and daemon can be installed on one machine, or the client can link up with a remote daemon. They talk to each other using a REST API, either through UNIX sockets or over a network

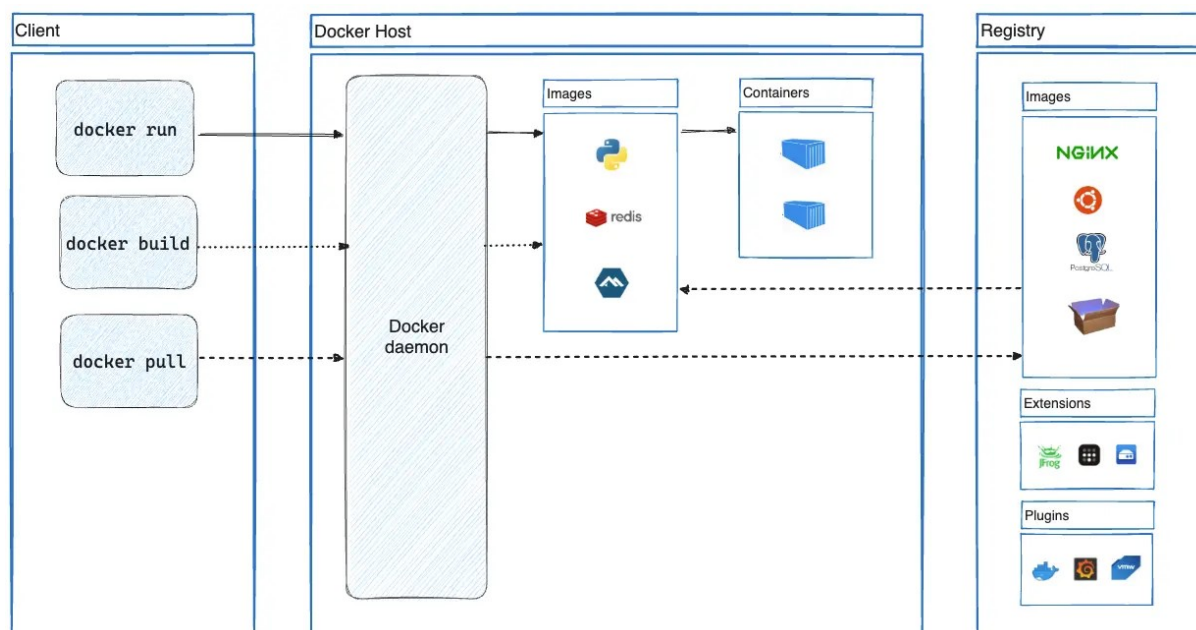


Figure 2.3: Docker architecture [2]

2.7.2 Docker components

Docker as a whole is composed of many different tools, but when most people talk about installing and using Docker it's in reference to the Docker daemon and Docker CLI.

2.7.2.1 Docker daemon

The Docker daemon, also known as dockerd, is a background process responsible for managing Docker containers on a host system. It serves as the central component of the Docker Engine, handling tasks related to container lifecycle management, image handling, networking, storage, also Additionally, a daemon has the capability to interact with other daemons for overseeing Docker services [23].

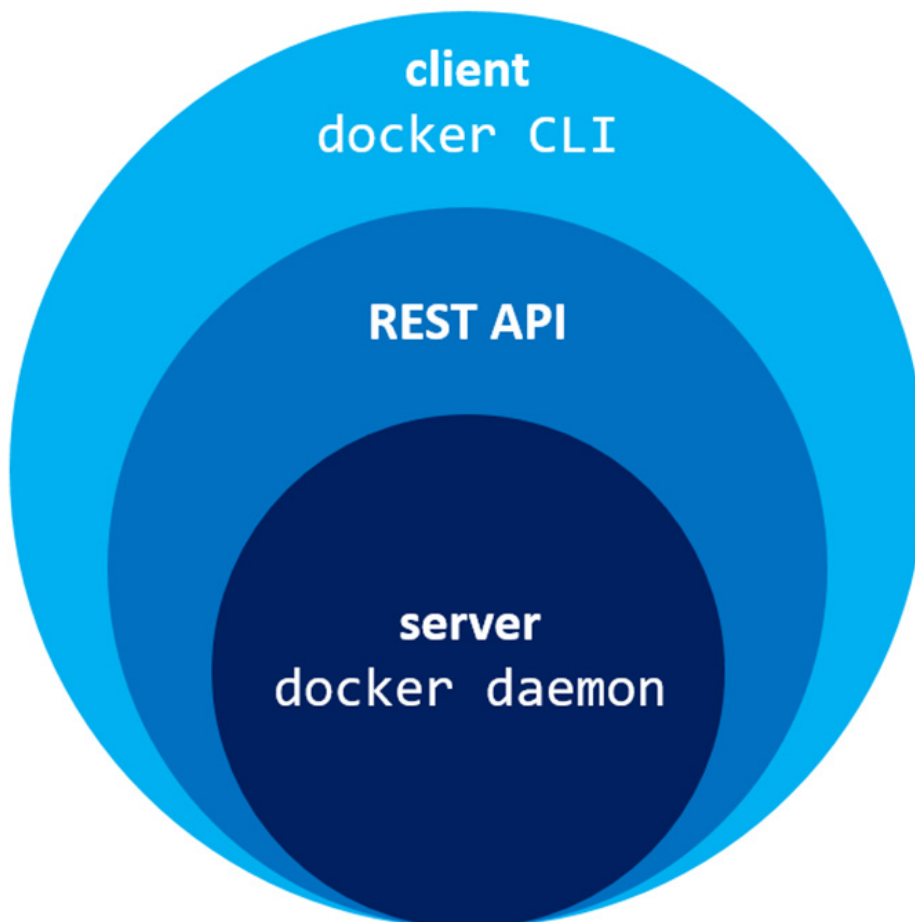


Figure 2.4: Docker daemon [3]

Key responsibilities of the Docker daemon include:

- API Server

The Docker daemon exposes a REST API that allows users and other Docker components (such as the Docker CLI) to interact with the Docker Engine programmatically

- Container Management

The Docker daemon is responsible for creating, starting, stopping, and deleting containers as per user requests.

- Image Handling

It manages Docker images, which serve as templates for creating containers. The daemon pulls, pushes, and caches images from and to Docker registries, and it builds images from Dockerfiles.

- Networking

The Docker daemon configures and manages networking for containers, including creating virtual network interfaces, assigning IP addresses, and configuring port mappings.

- Storage Management

It handles storage for containers, managing container volumes and storage drivers to ensure data persistence and efficient use of storage resources.

2.7.2.2 Docker client

The Docker client, known as "docker," is the main interface through which most Docker users interact with the system. Actions like running containers with "docker run" are communicated from the client to the dockerd, which then executes them. The docker command operates via the Docker API and has the ability to communicate with multiple daemons [2]

2.7.2.3 Docker desktop

Docker Desktop is a user-friendly software application designed for Mac, Windows, or Linux systems. It simplifies the process of creating and distributing containerized applications and microservices. Docker Desktop comprises essential components such as the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper [2]

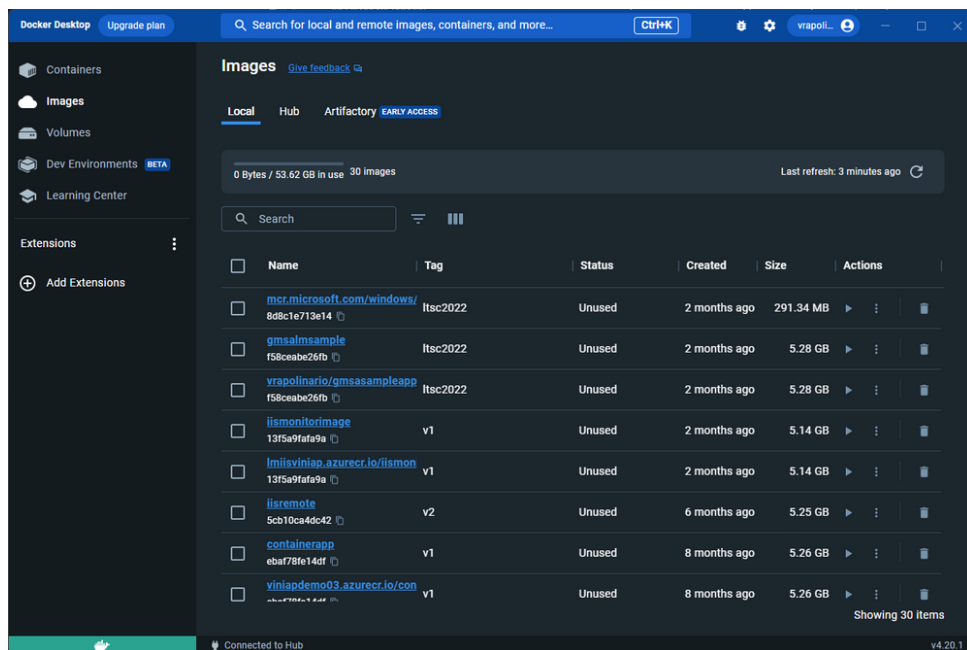


Figure 2.5: Docker desktop GNU [4]

2.7.2.4 Docker compose

is a tool for defining and running multi-container Docker applications, It uses YAML files to configure the application's services and performs the creation and start-up process of all the containers with a single command, The docker-compose CLI utility allows users to run commands on multiple containers at once; for example, building images, scaling containers, running containers that were stopped, and more, commands related to image manipulation, or user-interactive options, are not relevant in Docker Compose because they address one container. The docker-compose.yml file serves to specify an application's services and encompasses diverse configuration choices. For instance, the build option delineates configuration parameters like the Dockerfile path, while the command option permits overriding default Docker commands, among other functionalities. Docker Compose's initial public beta release (version 0.0.1) occurred on December 21, 2013. The first stable, production-ready version (1.0) was introduced on October 16, 2014.

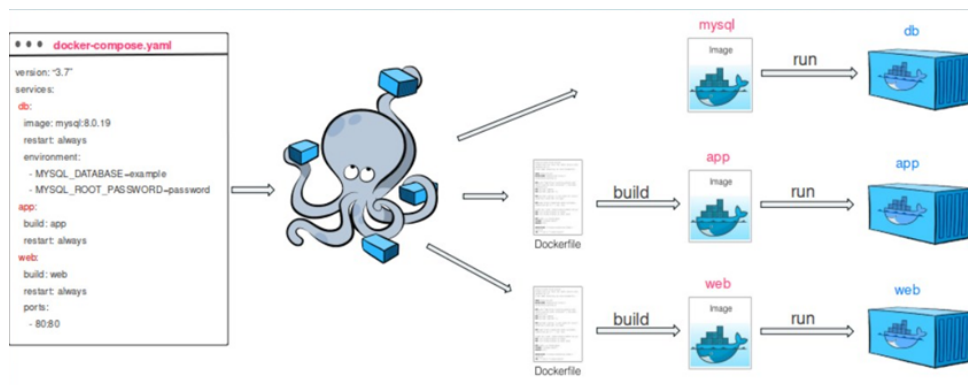


Figure 2.6: Docker compose role [5]

2.7.3 Docker Objects

Now that we have a grasp of containers, let's explore other related concepts like images, and DockerFile

2.7.3.1 Images

A Docker image, also known as a container image, is an independent executable file utilized for container creation. It encompasses all the essential libraries, dependencies, and files essential for the container's operation. Docker images are portable and shareable, enabling deployment across various locations concurrently, akin to software binary files. Images can be stored in registries to manage complex software architectures, projects, business sectors, and user group permissions. For instance, Docker Hub's public registry houses images including operating systems, programming language frameworks, databases, and code editors. Docker images are built using a Dockerfile, which is essentially a set of instructions guiding the platform on how to assemble the image using layers. Each instruction in the Dockerfile corresponds to the creation of a new layer within the image. The process typically involves the following steps:

1. Base Image Specification

The Dockerfile begins with specifying the base image to be used for the container. This base image serves as the starting point for building the final image.

2. Layer Creation

Subsequent instructions in the Dockerfile contribute to creating additional layers in the image. Each instruction represents a distinct action, such as installing dependencies, copying files into the image, or executing commands.

3. Layering Process

As each instruction is executed, Docker creates a new layer based on the changes introduced by that instruction. These layers are stacked on top of each other to form the final image.

4. Default Command Definition

The last line of the Dockerfile typically defines the default command to run when creating a container from the built image. This command specifies the primary executable or process that should be initiated within the container upon startup

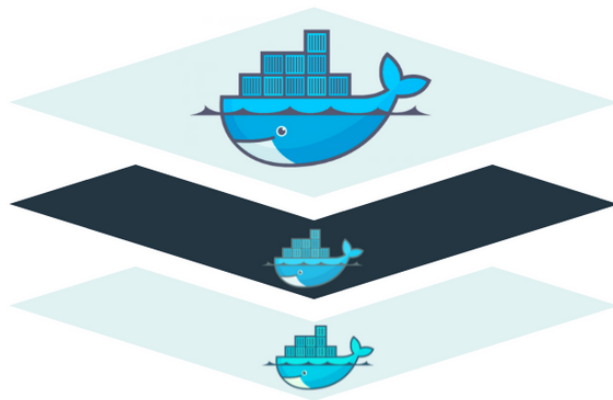


Figure 2.7: Image layers [6]

2.7.3.2 DockerFile

Docker has the capability to automatically construct images by interpreting instructions outlined in a Dockerfile. Essentially, a Dockerfile is a textual document encompassing all the commands that a user might execute via the command line to compile an image. These Dockerfile commands dictate the construction process of an image. To build an image from a Dockerfile, Docker daemon receives this file and executes its instructions to transform it into an image [24]

2.7.3.3 Containers

A container represents a runnable instance derived from an image. Using the Docker API or CLI, you have the ability to create, start, stop, move, or delete a container. Additionally, you can link a container to one or multiple networks, allocate storage to it,

or generate a new image based on its current status. By default, a container operates with a degree of isolation from other containers and the host machine. However, you have the flexibility to adjust the level of isolation for a container's network, storage, and other underlying subsystems, either from other containers or the host machine. The definition of a container encompasses its associated image and any configuration parameters supplied during its creation or launch. Upon deletion of a container, any modifications to its state not preserved in persistent storage will be lost. [2]

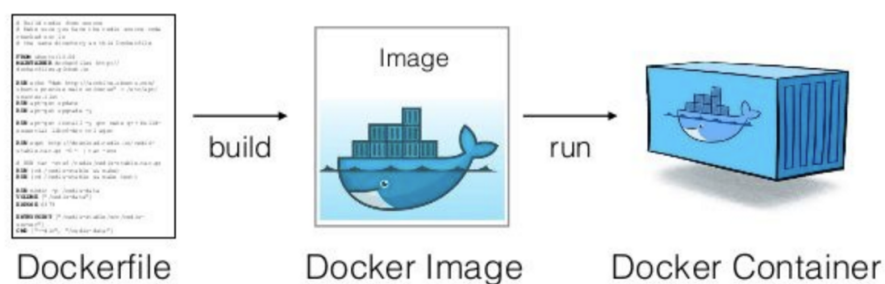


Figure 2.8: Container process [7]

2.8 Container orchestration

During the Gluecon conference on May 21, 2014, Joe Beda, co-founder of Kubernetes at Google, mentioned that "everything at Google runs in a container" and that "we start over 2 billion containers per week." . . . , with such a large number of containers, it leads us to wonder how they are managed. Container orchestration involves automating many of the operational tasks necessary for managing containerized workloads and services. This encompasses various aspects of managing the lifecycle of containers, such as provisioning, deploying, scaling (both horizontally and vertically), networking, load balancing, and additional functionalities. [25]

2.8.1 Container orchestration benefits

Container orchestration plays a crucial role in maximizing the advantages of containers, offering several benefits for containerized environments: [26]

- Streamlined operations

This stands out as the primary advantage of container orchestration and is a key driver for its adoption. Containers bring about a significant level of complexity, which can become overwhelming without proper orchestration to handle it efficiently

- Enhanced resilience

Container orchestration platforms have the capability to automatically restart or scale containers or clusters as needed, thereby improving resilience and ensuring uninterrupted operation

- Improved security

Through its automated processes, container orchestration contributes to maintaining the security of containerized applications by minimizing the risk of human errors and enhancing overall security measures

2.8.2 Orchestration tools

There are many orchestration tools available in the market to manage containerized applications at scale in production environments. among the popular options are Docker Swarm and Kubernetes (K8s), each with its own set of features and capabilities tailored to different use cases and requirements. Additionally, other tools such as Apache Mesos, Nomad, Amazon ECS, and OpenShift provide alternative solutions for container orchestration, catering to diverse infrastructure setups and operational needs, In this chapter, we'll dive into a detailed exploration of Kubernetes. But before we get into that, it's important to familiarize ourselves with some other orchestration tools.

- Docker swarm

Docker Swarm is Docker's native clustering and orchestration tool, designed to manage a cluster of Docker hosts. It enables users to deploy and manage a group of Docker hosts as a single virtual system, providing features such as service discovery, load balancing, and automatic scaling. Docker Swarm follows a simple and straightforward approach, making it easy for Docker users to orchestrate containerized applications without needing to learn complex new concepts. [27]

- Apache Mesos

Apache Mesos: Apache Mesos is a distributed systems kernel that abstracts CPU, memory, storage, and other compute resources, allowing users to run containers, virtual machines, and traditional applications on a shared pool of resources. [28]

- OpenShift (Red Hat):

OpenShift is a Kubernetes-based container platform developed by Red Hat, providing features for building, deploying, and managing containerized applications across hybrid cloud environments [29]

- Amazon ECS (Elastic Container Service):

Amazon ECS is a fully managed container orchestration service provided by Amazon Web Services (AWS), offering scalable and reliable deployment of containers on AWS infrastructure [30]

2.8.3 What is Kubernetes

Kubernetes helps you handle containerized apps smoothly and automatically. It's like a manager for containers. In tech talk, it's often called K8s pronounced Kate's. With tools like Docker, you bundle your code and all it needs into an image, then run it to make containers. But containers are temporary. If one stops working, Kubernetes makes sure another starts up right away. It also helps manage resources and organize containers into groups called clusters. [31]

2.8.4 How Is Kubernetes Different from Docker Swarm

Both Kubernetes and Docker Swarm are tools used for container orchestration, but they have different strengths. Docker Swarm is known for its simplicity and seamless integration with other Docker technologies. If you're already using Docker, Swarm might be the preferred choice due to its ease of use. However, Kubernetes offers more advanced features and capabilities for container orchestration, particularly in terms of scalability and monitoring. Kubernetes has built-in horizontal scaling, which Docker Swarm lacks. Additionally, Docker Swarm offers automatic load balancing, which Kubernetes does not. Keep in mind that Kubernetes has a steeper learning curve compared to Docker Swarm, both for developers and administrators [31]

CRITERIA	DOCKER SWARM	KUBERNETES
INSTALLATION AND SETUP	Easy to set up using the docker command	Complicated to manually set up the Kubernetes cluster
TYPES OF CONTAINERS THEY SUPPORT	Only works with Docker containers	Supports Container, Docker, CRI-O, and others
HIGH AVAILABILITY	Provides basic configuration to support high availability	Offers feature-rich support for high availability
CLI	Don't need to install other CLI	Need to install other CLIs such as kubectl
COMPLEXITY	Simple and lightweight	Complicated and offer a lot of features
LOAD BALANCING	Automatic load-balancing	Manual load-balancing
SCALABILITY	Does not support automatic scaling	Supports automatic scaling
SECURITY	Only supports TLS	Supports RBAC, SSL/TLS, and secret management

Table 2.3: Comparison of Docker Swarm and Kubernetes [10]

2.8.5 Why we use Docker and Kubernetes

Container runtimes like Docker and container orchestrators like Kubernetes are now the foundation of containerized application development. And together, microservices, containers, and Kubernetes form an indispensable part of the cloud computing ecosystem. All these technologies complement each other. So, it goes like this: using a microservices architecture, you will decompose your application into smaller services, which is needed so that the whole application is loosely coupled and the failure of one service doesn't bring down the whole application. Then, you will use a container runtime like Docker to run each of these services as a container, which is needed because containers are very lightweight and hence can be scaled up and down very quickly, responding to incoming traffic. Then, you will use a container orchestrator like Kubernetes to manage and orchestrate all application containers, which is required to provide high availability and fault tolerance to the application. And all these application containers will run on either a physical machine (most likely not) or a virtual machine, which would be running in the cloud [31]

2.8.6 How Did Kubernetes Come into Existence

Kubernetes comes from the Greek word, which means helpsman or ship pilot

Kubernetes has its roots in Borg, a private container management system used by Google for its major online services like Search, Gmail, and YouTube. The idea for Kubernetes came about in 2013 when Craig McLuckie proposed an open-source container management system to Urs Hölzle. Initially, the idea didn't gain much traction, but eventually, it evolved into Kubernetes. Originally called Seven of Nine, the project was later renamed Kubernetes. Kubernetes inherited four of its core features directly from Borg:

- Pod

A pod serves as the basic unit for scheduling tasks within a container management system. It allows for the execution of one or multiple containers, ensuring that containers within the same pod are always allocated to the same machine

- Services

In Kubernetes, a service functions to expose applications, which operate on a group of pods, as network services. This enables naming and load balancing within container management systems like Borg and Kubernetes.

- Labels

Within container management systems such as Borg and Kubernetes, labels are used to categorize and organize sets of objects. Label selectors, on the other hand, are utilized to choose objects based on specific labels.

- IP-per-pod

Both Borg and Kubernetes adhere to the IP-per-pod networking model. In this model, each pod possesses a distinct IP address assigned from its node's pod CIDR range. Additionally, all containers within the same pod share this IP address.

Between Borg and Kubernetes, Google designed and implemented another container management system called Omega. Omega was created to enhance and improve the architecture and design of Borg. So, Google designed three container management systems: Borg, Omega, and Kubernetes. Google kept Borg and Omega as internal systems while open-sourcing Kubernetes [31]

2.8.7 Kubernetes architecture

Deploying Kubernetes results in obtaining a Kubernetes cluster, which typically consists of one or more master nodes and one or more worker nodes. Each Kubernetes cluster must include at least a single worker node. In figure illustrates a standard Kubernetes cluster, emphasizing three key elements: the master node (also referred to as the control plane), the worker node, and the cloud API provider. The master node serves as the hub for all cluster management and operational decisions, essentially functioning as the brain of the Kubernetes cluster. On the other hand, the worker node is where pods are instantiated and executed. A pod represents the fundamental unit within Kubernetes where your application operates. Additionally, the master node oversees the management of the worker nodes

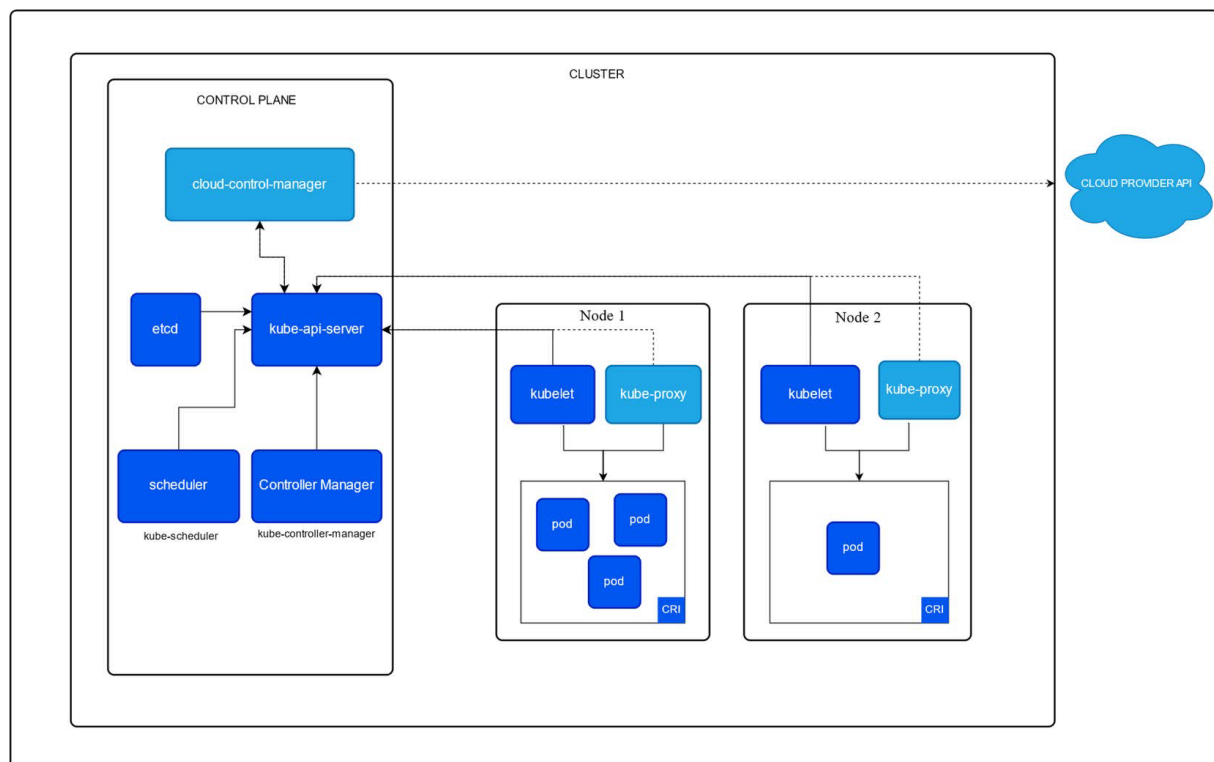


Figure 2.9: Kubernetes architecture [8]

In a production environment, it's advisable to have multiple master nodes distributed across various failure zones to ensure fault tolerance and high availability. Nonetheless, technically speaking, a Kubernetes cluster can function with just a single master node.

2.8.7.1 Master Node (Control Plane)

The master node includes several critical components that manage the cluster's operations:

- **API Server:** Serves as the frontend for Kubernetes. All administrative tasks and interactions within the cluster are handled through the API server.
- **Scheduler:** Responsible for assigning workloads (pods) to worker nodes based on resource availability and constraints.
- **Controller Manager:** Manages different controller processes that regulate the state of the cluster, such as node controller, replication controller, and endpoints controller.
- **etcd:** Distributed key-value store that stores the cluster's configuration data and current state. It ensures consistency and provides the basis for all cluster operations.

2.8.7.2 Worker Node

The worker node is where pods are instantiated and executed. It includes:

- **Kubelet:** Agent running on each worker node, responsible for communicating with the master node and managing containers (pods) on the node.
- **Container Runtime:** Software responsible for running containers, such as Docker, containerd, or CRI-O.
- **Kube Proxy:** Network proxy that reflects Kubernetes networking services on each node. It maintains network rules and performs connection forwarding.

2.8.7.3 Add-ons

Additional components and services that enhance Kubernetes functionality:

- **DNS:** Provides DNS-based service discovery for services running inside the cluster.
- **Dashboard:** Web-based Kubernetes user interface for managing and monitoring the cluster.
- **Ingress Controller:** Manages external access to services within the cluster, typically through HTTP and HTTPS routes.
- **Storage Plugins:** Allow Kubernetes to work with various storage solutions, providing persistent storage to applications.

2.9 Features of kubernetes

- **Scalability**

Kubernetes offers automatic horizontal scaling of pods based on CPU utilization. Administrators can configure the threshold for CPU usage, allowing Kubernetes to dynamically adjust the number of pods to meet demand. This automated scaling ensures efficient resource utilization and optimal performance without manual intervention. The scale has two types: [32]

- Vertical scale: This refers to scale by adding more resources such as CPU and RAM; the container will allow the use of the more powerful machine.
- Horizontal scale: This approach means adding more servers to the current one (more nodes); in that case, the container will run on different machines

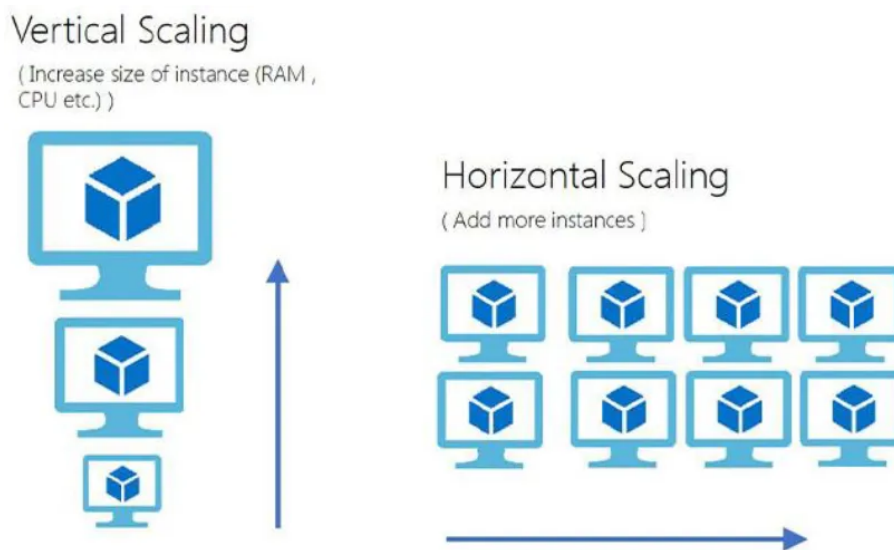


Figure 2.10: Vertical scaling vs Horizontal scaling [9]

- **Self-healing**

Kubernetes automatically replaces and reschedules containers from failed nodes, terminates unresponsive containers based on defined health checks, and prevents traffic from being routed to non-responsive containers according to configured rules/policies

- **Automated rollouts and rollbacks**

Kubernetes seamlessly rolls out and rolls back application updates and configuration changes, while constantly monitoring application health to prevent any downtime

- **Secret and configuration management** Kubernetes manages sensitive data and configuration details for an application separately from the container image
- **Portability**
a cluster can run on any main stream Linux distribution processor architectures (either virtual machines or baremetal) cloud providers (AWS,AZURE OR google platform)

2.10 Conclusion

In conclusion, containerization represents a pivotal component of our project's vision to streamline the delivery and management of educational computing environments. By leveraging container technology, we aim to provide educators and students with tailored, isolated environments for their specific teaching modules, enhancing stability and accessibility. Containers enable the encapsulation of necessary software dependencies and configurations, ensuring consistent and reproducible environments across different machines and semesters. Additionally, containerization automates the management of our educational computing center, alleviating the administrative burden on system administrators. Through this approach, we anticipate facilitating a seamless and efficient experience for both educators and students during practical sessions, ultimately enhancing the learning environment and promoting academic success.

Chapter 3

Conception

3.1 Introduction

In the previous chapters, we delved into the fundamentals of containerization, virtualization we are now transiting from theory to practice. This chapter, dedicated to the conception phase, serves as a critical bridge between understanding and implementation. We will start by defining the problem statement and setting clear objectives. From there, we will analyze the requirements, both functional and non-functional, to ensure that all aspects of the system are well-understood and documented.

3.2 Problem Statement

The educational environment has a significant connection to advancements in the IT field, such as networking, virtualization, cloud computing, artificial intelligence, and systems administration. These domains undergo continuous evolution, marked by daily implementations of fresh innovations. Thus, addressing challenges within education is imperative to overcome barriers hindering the advancement of teaching methodologies and practical applications.

3.2.1 Description of the Problem:

In IT educational infrastructure, teachers often share the same resources, tools, and technologies during sessions. However, problems arise when different versions of technologies are needed. For instance, if one teacher requires a newer version of a tool already installed, the admin might have to uninstall the previous version and install the new one. This is not a good practice. Additionally, conflicts can occur when two tools require different

versions of the same dependency, leading to package and dependency issues. we frequently encounter a lot of resources that are not being fully utilized, in many classrooms, you'll find numerous machines that are out of service. this underutilization of available resources poses a significant obstacle to maximizing the effectiveness of our educational environment, thus we frequently encounter software-related issues with machines, such as the absence of essential programs like Java or C. These problems often arise due to students modifying packages or the lack of necessary dependencies or binaries. Consequently, these machines remain out of service for extended periods, leading to underutilization of resources throughout the year. This underutilization not only represents a waste of energy and time for students attempting to use the machines but also incurs financial costs for maintaining these resources. Moreover, it poses a substantial obstacle to the smooth operation of our sessions, hindering the learning process and impeding the achievement of educational objectives. Addressing these issues is essential to optimize resource utilization, enhance efficiency, and create a more conducive learning environment.

3.2.2 Background and use cases:

we will describe typical issues that our project aims to address and provide solutions for.

Two versions of JDK on the same classroom machines

in IT department Providing machines with two versions is possible but has significant drawbacks to consider. Here are the potential negatives of this method:

- **Complexity in Configuration:**

Environment Variables: Managing environment variables such as "JAVA HOME" and the "Path variable" can become complex and error-prone. Incorrectly setting these can lead to runtime errors or build failures.

Shell Configuration: On Unix-based systems, modifying shell configuration files like ".bashrc", ".bash profile, or ".zshrc" for tools like jEnv can be confusing for less experienced users.

- **Manual Management:**

Frequent Switching: Developers often need to switch between JDK versions frequently. Manually changing environment variables or using command-line tools for this purpose can be time-consuming and interrupt the workflow.

Human Error: There's a higher chance of human error when setting paths and environment variables manually, leading to potential issues with project builds and executions.

- **Resource Intensive:**

Disk Space: Installing multiple JDK versions on the same machine consumes more disk space. Each JDK installation can be quite large, and having multiple versions can quickly use up available storage.

Memory Usage: Running multiple development environments or tools that require different JDK versions simultaneously can consume significant system memory and resources, potentially slowing down the machine.

- **Maintenance Overhead:**

Updates and Security: Keeping multiple JDK versions up to date with the latest security patches and updates can be a maintenance burden. Each version must be individually updated and managed. *Tooling Support:* Not all development tools or IDEs handle multiple JDK installations smoothly, which can lead to additional configuration and troubleshooting efforts.

You can apply this use case to any technology or tool a teacher might need for their session

Machine Learning with TensorFlow and PyTorch

let's consider a real-world example involving Python where different tools or packages require different versions of a dependency. This scenario often occurs in data science and machine learning projects.

Suppose you are working on a machine learning project that requires both TensorFlow and PyTorch, and each framework depends on different versions of a common library, such as numpy in figure 3.1 describe this kind of conflict .

- -TensorFlow: Requires numpy version 1.19.x.
- -PyTorch: Requires numpy version 1.21.x.

When you try to install both TensorFlow and PyTorch in the same Python environment, you will encounter a conflict

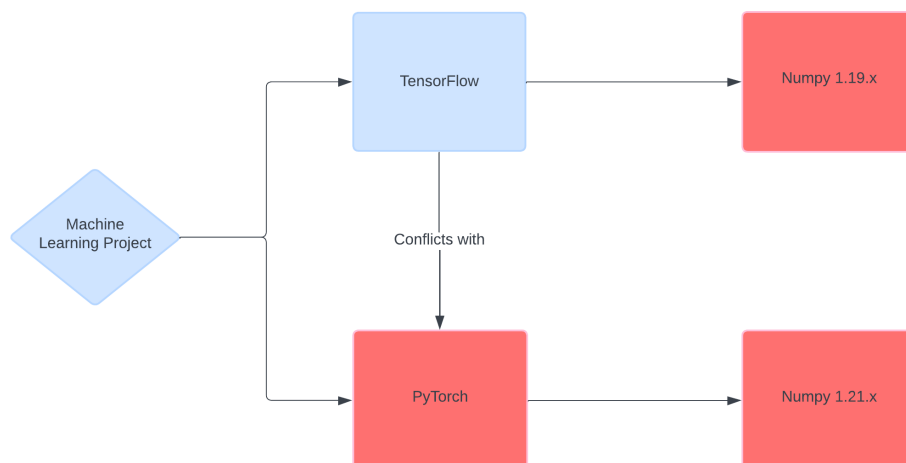


Figure 3.1: Packages conflict figure

3.2.3 Objectives:

Separation of concerns is an approach widely adopted across various IT fields. It involves breaking down complex systems into manageable components, each responsible for a specific aspect of functionality. In the context of education infrastructure, providing each teacher with an environment tailored to their needs can greatly enhance efficiency and effectiveness. Similarly, implementing this approach within our infrastructure can address many of the educational challenges we face, streamlining operations and optimizing resource utilization we can

To achieve this, we will use containerization technology. Containerization allows us to encapsulate software and its dependencies into isolated containers, ensuring that each environment is consistent and free from conflicts. This approach not only enhances portability and scalability but also simplifies the management of diverse application requirements.

We are going to build a prototype to demonstrate this concept. The prototype will provide a practical example of how containerization can be used to create isolated, tailored environments for different educational needs, thereby improving the overall effectiveness and efficiency of our infrastructure.

3.3 Requirements Analysis

Functional Requirements

- **environment management:**

Teacher have possibility to create there environment of each module with the way they like with technologies they need

- **Isolation:** Ensure that each teacher environment is isolated from others to prevent interference and conflicts. and isolated from the host where this environment is running

- **Containerization Support:**

Containerization ensure the isolation and separation of objects, so the platform should support containerization technology to encapsulate the software environment for each module by using one of container runtime

- **Automated Deployment:**

Implement automated deployment mechanisms to streamline the setup of environments

- **User Authentication:** Provide user authentication mechanisms to control access to module environments when they start sessions.
- **availability :** Develop a platform that guarantees the environment is accessible whenever it is needed.

Non-Functional Requirements:

- **Usability:** The platform should have an intuitive user interface to facilitate easy navigation and usage
- **Reliability:** Ensure reliable operation of the platform, minimizing downtime and system failures

The environment we are aiming to build it focus on three axes automatisation ,disponibility,system managment

3.4 System Architecture

3.4.1 High-Level Architecture:

The system architecture for the educational computing platform using containerization consists of several key components working together to provide an efficient environment for running educational modules. The architecture can be broken down into the following layers and components:

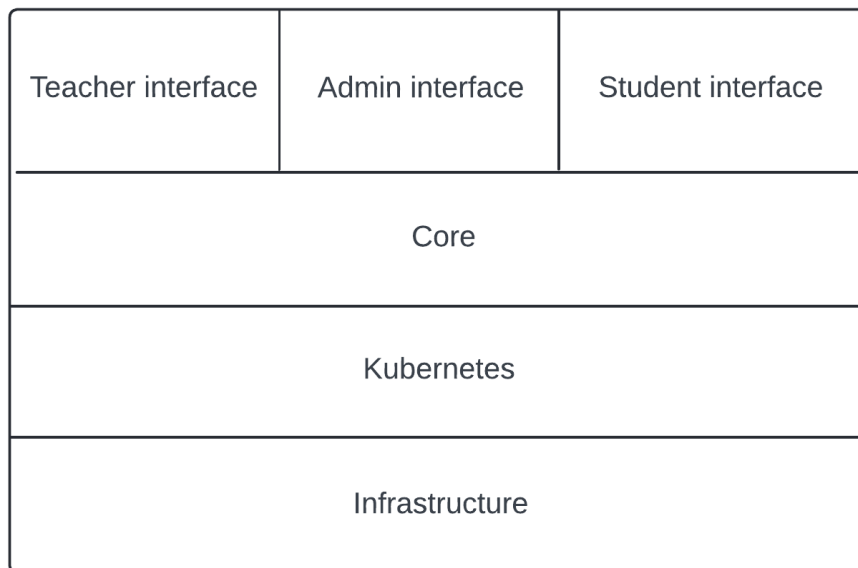


Figure 3.2: High-level architecture

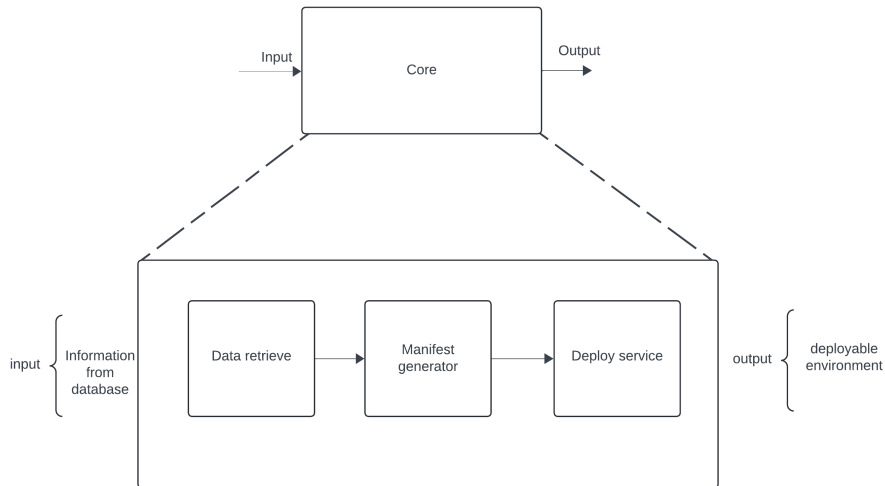
User Interface Layer:

- Web Portal:

A user-friendly web interface where teachers and administrators can interact with the platform.

- APIs:

RESTful APIs to allow integration with other systems and automation scripts

Core Layer:**Figure 3.3:** Core architecture

- Data retrieve service:

This service is responsible for retrieve data of the environments built by teachers then collect the necessary information to send it to manifest generator service

- Manifests generator service:

This service is responsible for generate manifests.yaml to deploy the environment

- Deploy service:

This service is responsible for deploy the environment in the kubernetes cluster

Kubernetes Layer:

Kubernetes cluster contain master node it receive manifest.yaml from the core then it apply those configurations in the worker nodes

- Container Orchestration:

Kubernetes: Manages the deployment, scaling, and operation of containerized applications.

- Container Runtime:

The environment where containers run, managed by Docker Engine or containerd or cri-o.

- Running hosts:

Provide a good architecture and networking for the machines that will be a platform of teacher's environments

Infrastructure Layer:

- Compute Resources:

Virtual machines or physical servers hosting the containerized environments.

- Networking:

Overlay Network: Enables communication between containers across different hosts.

3.4.2 Architecture design:

To create a flexible and efficient educational computing infrastructure, we propose the plan in the diagram below for managing and deploying multiple versions of technologies and tools. This model aims to address compatibility issues and conflicting dependencies by utilizing containerization solutions

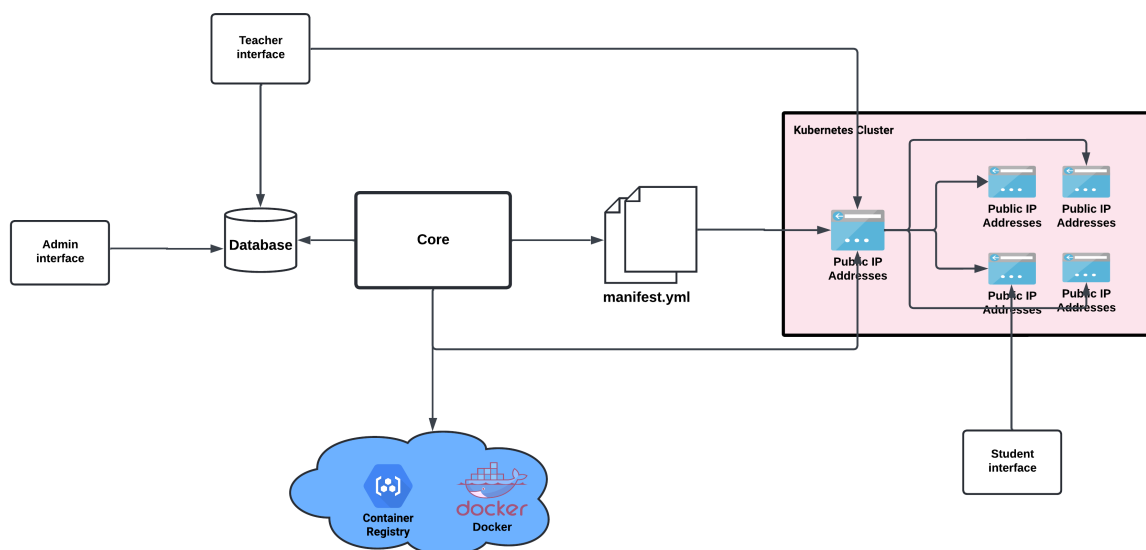


Figure 3.4: General architecture of the environment

Classroom companion

Classroom companion it's the name of the web application where the admin and teachers can manage there configurations and environments

- Admin interface

The admin is responsible for managing module information, class details, and teacher information. They assign modules to teachers, manage their account credentials and calendars, and ensure the overall stability of the environment. The admin also has access to the environment's database to perform these tasks.

- Teacher interface

In the teacher's space, they can find the modules assigned to them by the admin. The teacher is responsible for building the environment for each module by selecting the programming languages and frameworks, along with their specific versions, that they want to use. They can test the environment through the same web application from anywhere, ensuring it meets their requirements. Once satisfied with the setup, they can finalize and apply the environment for each module. This setup process is typically done once, unless the teacher needs to update the environment by changing versions or removing unnecessary tools. The system allows for easy updates and modifications to ensure the environment stays current with their teaching needs.

- Database

Our IT system relies on a central database to store all information, including data of teachers, modules, and more. This database is pivotal to the functionality of our system, serving as its intuitive core.

- Core

The core is the main part that interact to the computer stuff by sending orders and setups through manifests.yml files. It gathers data from the database, creates manifests, and sends them to the components that need them.

- manifest files

The manifest files, typically named 'manifest.yml', are crucial for defining the configuration and deployment details of each environment. These files are used by the system to automate the deployment process in the Kubernetes cluster.

Kubernetes cluster

The Kubernetes cluster is the backbone of our container orchestration system, providing the necessary infrastructure to manage and deploy containerized applications efficiently. The classes of the department it will be considered as one kubernetes cluster, each class belong to a subnetwork, each machine is a worker node in this cluster

Container image registry

The Container’s Image Registry is a crucial component of our containerized environment, serving as a centralized repository for distributing container images. This registry ensures that all the required images for various modules and environments are readily available for deployment. the kubernetes cluster will interact with those registries to get specific images based on manifest files

3.5 UML Diagrams

To provide a detailed and structured view of the system design, the following UML diagrams are included:

3.5.1 Class diagram

The Class Diagram provides a detailed view of the system’s structure by depicting the system’s classes, attributes, methods, and the relationships among the classes. This diagram is crucial for understanding the static design of the system and how different components interact at a code level. It serves as a blueprint for the system’s implementation and helps in identifying the key objects and their interactions.

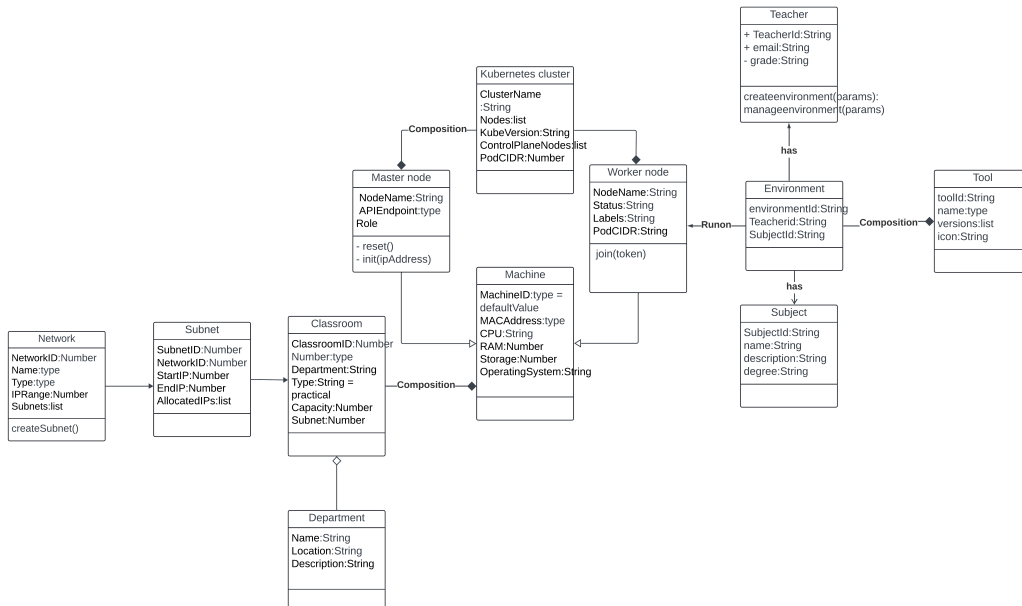


Figure 3.5: Class diagram figure

3.5.2 Use Case Diagram

The Use Case Diagram shows the interactions between users and the system, highlighting the main functionalities provided.

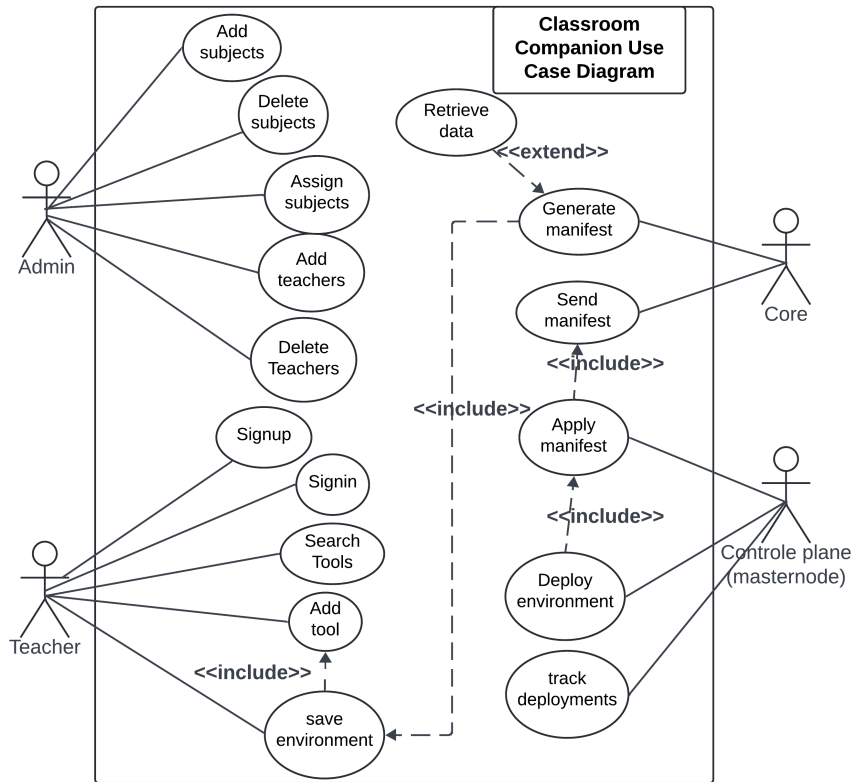


Figure 3.6: use case diagram

3.5.3 Database schema

Our database schema is designed to efficiently store and manage the data necessary for the system prototype. It ensures data integrity, supports necessary relationships between entities, and optimizes performance for the application's key operations.

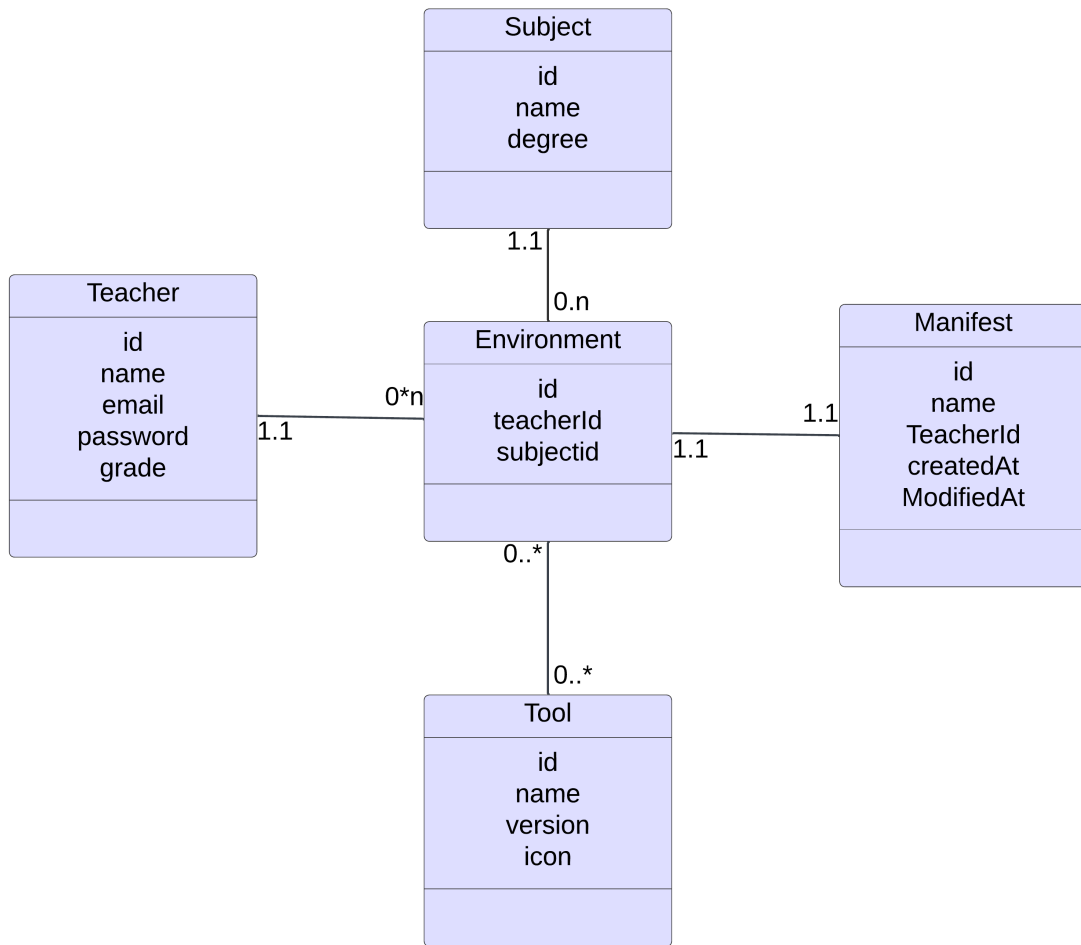


Figure 3.7: Database schema

3.6 Conclusion

In this chapter, we have transitioned from the theoretical foundations of containerization and virtualization to the practical aspects of designing our educational computing platform. We started by clearly defining the problem statement and objectives, ensuring a solid understanding of the challenges we aim to address. We analyzed the functional and non-functional requirements to provide a comprehensive overview of the system's needs.

We then outlined the system architecture, detailing the high-level components and their interactions, including the user interface, application layer, containerization layer, and infrastructure layer. This architecture serves as the blueprint for our platform, emphasizing scalability, security, and efficiency.

Additionally, we discussed the design considerations for the admin interface and the teacher's space, ensuring that both administrators and teachers have the necessary tools to manage and utilize the platform effectively. We also touched on essential aspects such as manifest files, Kubernetes clusters, and container image registries.

In the following sections, we provided detailed UML diagrams to visualize the system's structure and interactions, further clarifying the design. These diagrams, along with the subsequent sections on database design, component design, and API design, form the foundation for the implementation phase.

By thoroughly documenting the conception phase, we have established a clear and detailed plan for building our educational computing platform. This will guide us through the implementation, ensuring that we address the identified challenges and achieve our objectives effectively.

Chapter 4

Implementation

4.1 Introduction

In the previous chapter, we outlined the conceptual framework and design of our educational computing platform using containerization. With a clear understanding of the problem, requirements, and system architecture, we now transition to the implementation phase. This phase is crucial as it involves translating our design into a functional and operational system.

The primary objectives of the implementation phase are to develop the user interface, set up the backend services, and integrate containerization to ensure a scalable, efficient, and secure environment for educational modules. We will detail the setup of the development environment, the construction of the user interface and backend, the creation of Docker images, and the deployment of these components using Kubernetes. Additionally, we will cover the integration of security measures, monitoring, and logging to maintain a robust and reliable system.

This chapter will serve as a comprehensive guide through the implementation process, providing detailed steps and considerations for each aspect of the system prototype. By the end of this chapter, we aim to have a fully functional beta platform ready for deployment, capable of addressing the challenges outlined in the problem statement and meeting the needs of our users.

4.2 Setting Up the Environment

To fully understand our system, building a prototype is a great idea. This prototype will allow us to experiment with different configurations, test various components, and identify potential issues early in the development process. Setting up the environment

involves several steps, including the installation of necessary software, configuration of development tools, and preparation of the infrastructure to support containerization and orchestration.

4.2.1 Kubernetes cluster

Kubernetes offers extensive documentation for beginners to learn from scratch. It also provides tools that allow you to try Kubernetes and learn its basics. However, these tools do not encompass all of Kubernetes' features.

- kind

Kind allows you to run Kubernetes on your local computer. To use this tool, you need to have Docker or Podman installed. [33]

- Minikube

Similar to Kind, Minikube is a tool that enables you to run Kubernetes locally. Minikube can set up either a single-node or a multi-node Kubernetes cluster on your personal computer, whether it's running Windows, macOS, or Linux. This allows you to experiment with Kubernetes or use it for daily development tasks. [33].

In our case, we need to try a production environment to benefit from all the features and capabilities that Kubernetes offers. This will enable us to accurately simulate real-world scenarios, optimize resource allocation, and ensure the stability and scalability of our applications.

A production-quality Kubernetes cluster requires planning and preparation. It must be configured to be resilient. A production Kubernetes cluster environment has more requirements than a personal learning environment. A production environment may require secure access by many users, consistent availability, and the resources to adapt to changing demands.

There are many methods and tools for setting up your own production Kubernetes cluster. For example:

- kubeadm

- kops:

An automated cluster provisioning tool. For tutorials, best practices, configuration options and information on reaching out to the community, please check the kOps website for details.

- kubespray:

A composition of Ansible playbooks, inventory, provisioning tools, and domain knowledge for generic OS/Kubernetes clusters configuration management tasks. You can reach out to the community on Slack channel

4.2.1.1 Resources Requirements

We detail the necessary resources for setting up a Kubernetes cluster using kubeadm. Meeting these requirements is vital for the cluster's smooth operation and reliability. Each machine in the cluster must adhere to these prerequisites. Below is a comprehensive list of the hardware and network requirements.

Requirement	Details
Compatible Linux Host	The Kubernetes project provides generic instructions for Linux distributions based on Debian and Red Hat, and those distributions without a package manager.
Memory	2 GB or more of RAM per machine (any less will leave little room for your apps).
CPU	2 CPUs or more.
Network Connectivity	Full network connectivity between all machines in the cluster (public or private network is fine).
Unique Identifiers	Unique hostname, MAC address, and product_uuid for every node. See here for more details.
Open Ports	Certain ports must be open on your machines.

Table 4.1: resources requirements of the kubernetes cluster

Trying Kubernetes at home requires virtual machines, so I chose to test this cluster on Microsoft Azure cloud resources.

4.2.2 Creating Virtual Machines on Microsoft Azure

To set up a Kubernetes cluster, we need to create three virtual machines (VMs) on Microsoft Azure. Follow these steps to create the VMs:

- Step 1: Log In to the Azure Portal

Navigate to <https://portal.azure.com> and log in with your Azure credentials.

- Step 2: Install Azure CLI

Install the Azure CLI by following the instructions at <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli>.

- Step 3: Log In to Azure CLI

Open a terminal and log in to Azure:

```
az login
```

- Step 4: Create a Resource Group

```
az group create --name myResourceGroup --location eastus
```

- Step 5: Create the First Virtual Machine

Create the first VM (e.g., *k8s-master*):

```
az vm create \  
  --resource-group myResourceGroup \  
  --name master \  
  --image UbuntuLTS \  
  --size Standard_B2s \  
  --admin-username azureuser \  
  --generate-ssh-keys
```

- Step 6: Create the Second and Third Virtual Machines

Create the second VM (e.g., *k8s-node1*):

```
az vm create \  
  --resource-group myResourceGroup \  
  --name worker \  
  --image UbuntuLTS \  
  --size Standard_B2s \  
  --admin-username azureuser \  
  --generate-ssh-keys
```

the third VM (e.g., *k8s-node2*):

```
az vm create \  
  --resource-group myResourceGroup \  
  --name Worker2 \  
  --image UbuntuLTS \  
  --size Standard_B2s \  
  --admin-username azureuser \  
  --generate-ssh-keys
```

- Step 7: Open Necessary Ports

Open the necessary ports for Kubernetes communication on each VM:

```
az vm open-port --port 6443 --resource-group myResGroup --name master  
az vm open-port --port 2379-2380 --resource-group myResGroup --name master  
az vm open-port --port 10250 --resource-group myResGroup --name master  
az vm open-port --port 10251 --resource-group myResGroup --name master  
az vm open-port --port 10252 --resource-group myResGroup --name master  
az vm open-port --port 10250 --resource-group myResGroup --name worker  
az vm open-port --port 10250 --resource-group myResGroup --name worker2
```

By following these steps, you will have created three virtual machines on Azure, which will serve as the nodes for your Kubernetes cluster.

Configuring Vms

After creating virtual machines comes the step to configure them by following the official documentation provided by kubernetes

- Disabling swap

You MUST disable swap if the kubelet is not properly configured to use swap. For example, `sudo swapoff -a` will disable swapping temporarily. To make this change persistent across reboots, make sure swap is disabled in config files like `/etc/fstab`, `systemd.swap`, depending how it was configured on your system.

- Installing container runtime

we need to install a container runtime into each VM in the cluster so that Pods can run there. installing a container runtime require to install and configure some prerequisites you might find in the configuration file

- Installing kubernetes packages

You will install these packages on all of your machines:

1. kubeadm: the command to bootstrap the cluster.
2. kubelet: the component that runs on all of the machines in your cluster and does things like starting pods and containers.
3. kubectl: the command line util to talk to your cluster.

we will apply this configuration on all Vm's

Configuring master Vm

In the previous configuration, we outlined the settings that need to be applied to all Kubernetes nodes. In this section, we will focus on the configuration specific to the master node.

- Pull images

This step is optional and applies only if you want kubeadm init and kubeadm join to avoid downloading the default container images hosted at registry.k8s.io. By doing this, you won't need an internet connection when adding a computer to the cluster.

- Initializing your control-plane node

The control-plane node hosts essential control plane components, including etcd (the cluster's database) and the API Server (which interfaces with the kubectl command line tool). As a result of this process, an invitation token is generated, which is necessary for machines to join the cluster.

By following these steps and forums, you will achieve configurations similar to those in this GitHub repository: <https://github.com/raidkarki/kubeadm-configuration/tree/main/scripts>, and a robust cluster capable of handling our environment requirements.

```
root@master:/home/karkiraid35# kubectl get nodes
NAME          STATUS    ROLES    AGE     VERSION
master        Ready    control-plane   2d23h   v1.29.0
worker        Ready    <none>         2d23h   v1.29.0
worker2       Ready    <none>         2d23h   v1.29.0
root@master:/home/karkiraid35# |
```

Figure 4.1: Kubernetes cluster

4.3 Developing classroom companion

Classroom companion it's a web application designed to be part of our system, built with MERN stack

4.3.1 Technologies and Tools

To develop the Classroom Companion, we utilize the following technologies and tools:

- **MongoDB:** A NoSQL database that provides a flexible, scalable way to store and manage data. It is used to store information about teachers, students, classes, and resources.
- **Express.js:** A minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It is used to build the backend of the Classroom Companion.
- **React:** A JavaScript library for building user interfaces. It is used to develop the frontend of the application, providing a dynamic and responsive user experience.
- **Node.js:** A JavaScript runtime built on Chrome's V8 JavaScript engine. It allows us to run JavaScript on the server side, enabling the development of the backend services for our application.
- **Docker:** Used to containerize the application, ensuring consistency across different environments and simplifying deployment.
- **Jest:** A testing framework for JavaScript, used to write and run tests to ensure the reliability and correctness of the code.

In addition to these core technologies, we also use various libraries and tools to enhance development and ensure a smooth workflow, including:

- **Mongoose:** An Object Data Modeling (ODM) library for MongoDB and Node.js, used to manage database operations.

- **Postman:** An API development tool used for testing and debugging the backend APIs.

By leveraging these technologies and tools, we ensure that the Classroom Companion is robust, scalable, and user-friendly, providing a seamless experience for educators and students alike. you can test classroom companion by visit <http://4.233.222.154/>¹,also this is the repository of the project <https://github.com/raidkarki/PFE>

¹The URL may be inaccessible due to system maintenance or updates.

4.3.2 Functionality

4.3.2.1 Uploading data to system

The admin interface is a dashboard that allows management of teacher accounts by adding their credentials and assigning subjects to them

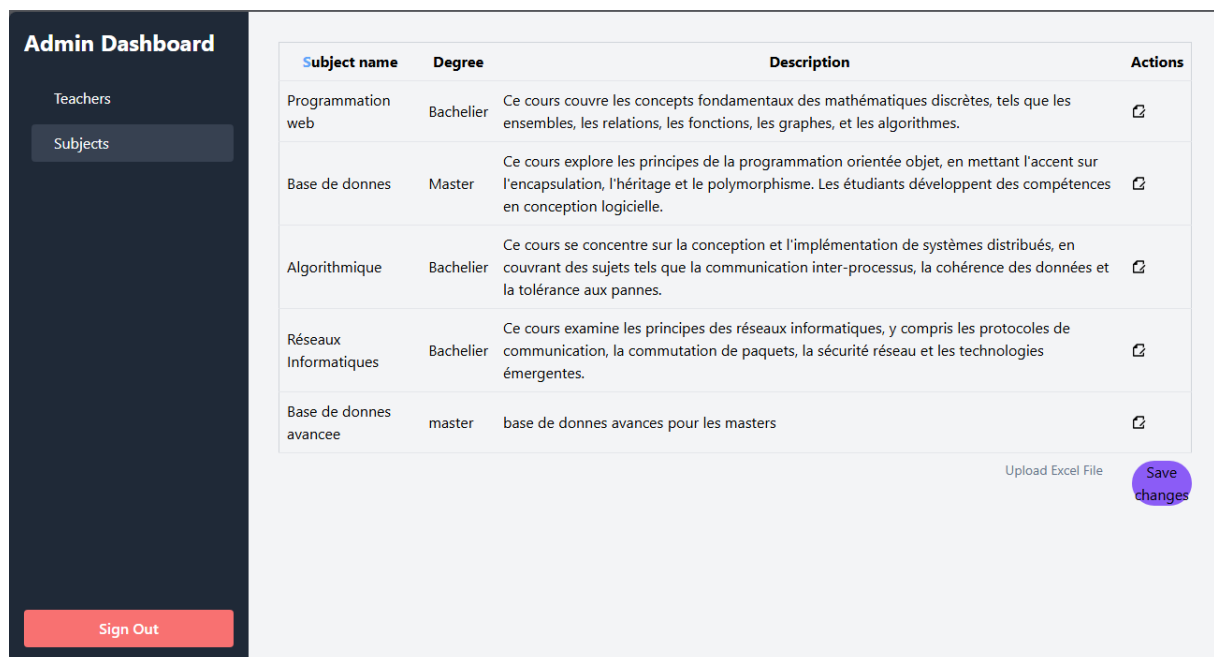


Name	Surname	Email	Connection Status
Abdelmoumene	Hiba	abdelmoumene.hiba@univ-guelma.dz	Connected
Aggoune	Aicha	aggoune.aicha@univ-guelma.dz	Connected
Aichour	Malek	aichour.malek@univ-guelma.dz	Not connected
Zemmouchi	Fares	zemmouchi.fares@univ-guelma.dz	Not connected
Zendaoui	Zakaria	zendaoui.zakaria@univ-guelma.dz	Not connected

Upload Excel File [Save](#)

[Sign Out](#)

Figure 4.2: Admin upload teachers into database



Subject name	Degree	Description	Actions
Programmation web	Bachelier	Ce cours couvre les concepts fondamentaux des mathématiques discrètes, tels que les ensembles, les relations, les fonctions, les graphes, et les algorithmes.	✎
Base de donnes	Master	Ce cours explore les principes de la programmation orientée objet, en mettant l'accent sur l'encapsulation, l'héritage et le polymorphisme. Les étudiants développent des compétences en conception logicielle.	✎
Algorithmique	Bachelier	Ce cours se concentre sur la conception et l'implémentation de systèmes distribués, en couvrant des sujets tels que la communication inter-processus, la cohérence des données et la tolérance aux pannes.	✎
Réseaux Informatiques	Bachelier	Ce cours examine les principes des réseaux informatiques, y compris les protocoles de communication, la commutation de paquets, la sécurité réseau et les technologies émergentes.	✎
Base de donnes avancee	master	base de donnes avances pour les masters	✎

Upload Excel File [Save changes](#)

[Sign Out](#)

Figure 4.3: Admin upload Subjects into database

4.3.2.2 Assign subjects to teacher

The administrator decides which teacher will teach each subject. In our situation, the administrator may also hold the position of the head of the IT department.

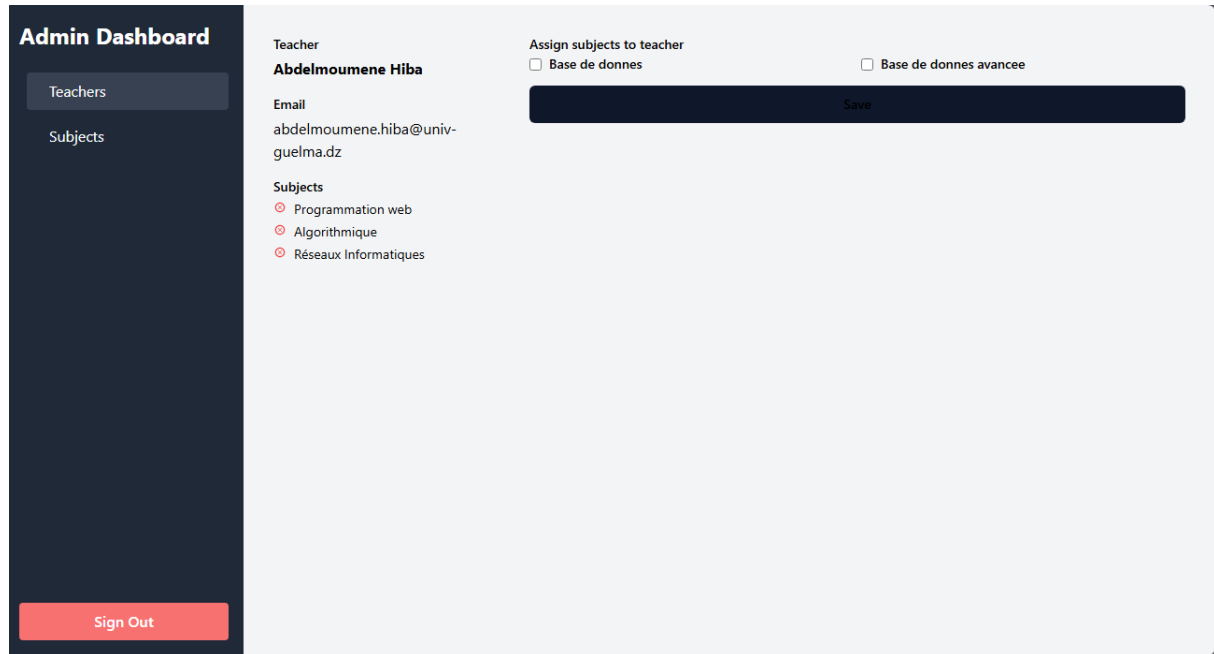
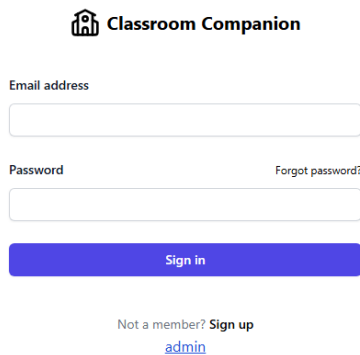


Figure 4.4: Admin assign subjects to teacher

4.3.2.3 User authentication

Authentication is a security measure must be when we need to provide functionality to a specific actor ,in our application the teacher can sign up only if the admin add his credentials in the database then he can sign in to the platform ,this action to prevent Intruders use the platform

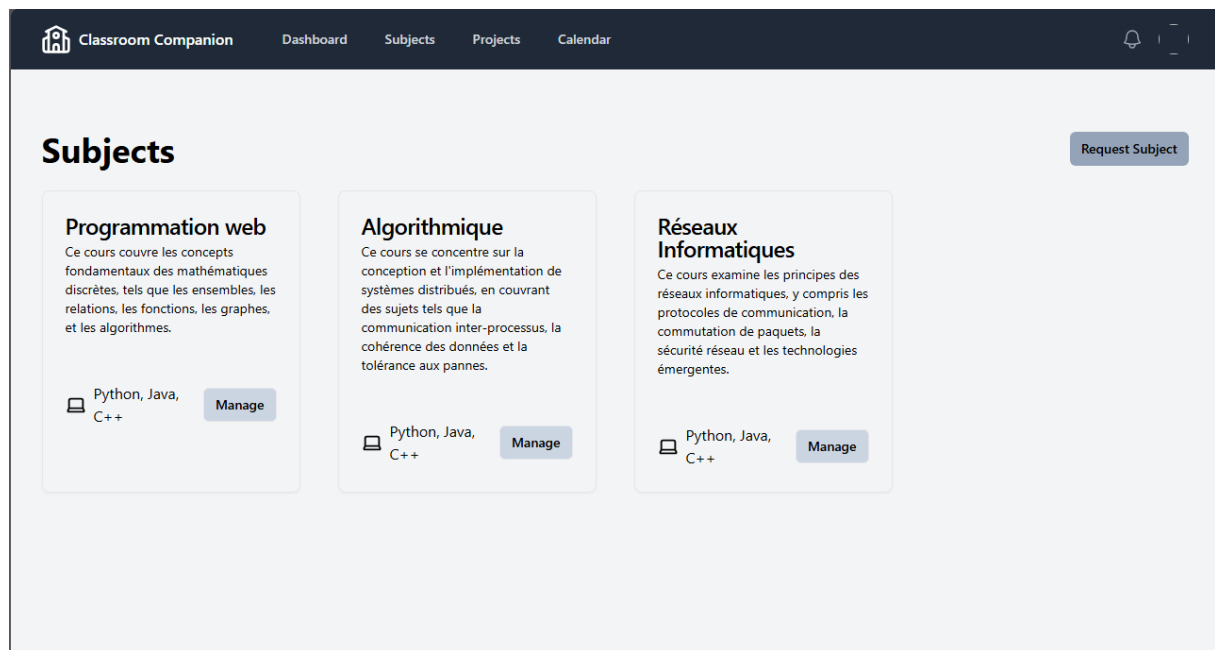


The authentication interface for Classroom Companion features a dark header with the logo and title. Below the header, there are two input fields: 'Email address' and 'Password'. A 'Forgot password?' link is positioned to the right of the password field. A prominent blue 'Sign in' button is centered below the fields. At the bottom, a link for 'Not a member? Sign up admin' is displayed.

Figure 4.5: Authentication interface

4.3.2.4 Manage environments

After the teacher is authenticated, they proceed to manage their environments. They will find the subjects assigned to them and then manage each subject by searching for tools with the specific versions they require during their session.



The teacher dashboard, titled 'Subjects', features a dark navigation bar with 'Dashboard', 'Subjects', 'Projects', and 'Calendar' options. A 'Request Subject' button is located in the top right corner. The main content area displays three subject cards, each with a description, supported languages (Python, Java, C++), and a 'Manage' button. The subjects are: 'Programmation web', 'Algorithmique', and 'Réseaux Informatiques'.

Figure 4.6: Teacher dashboard

In 4.7 teacher make a search for node js runtime

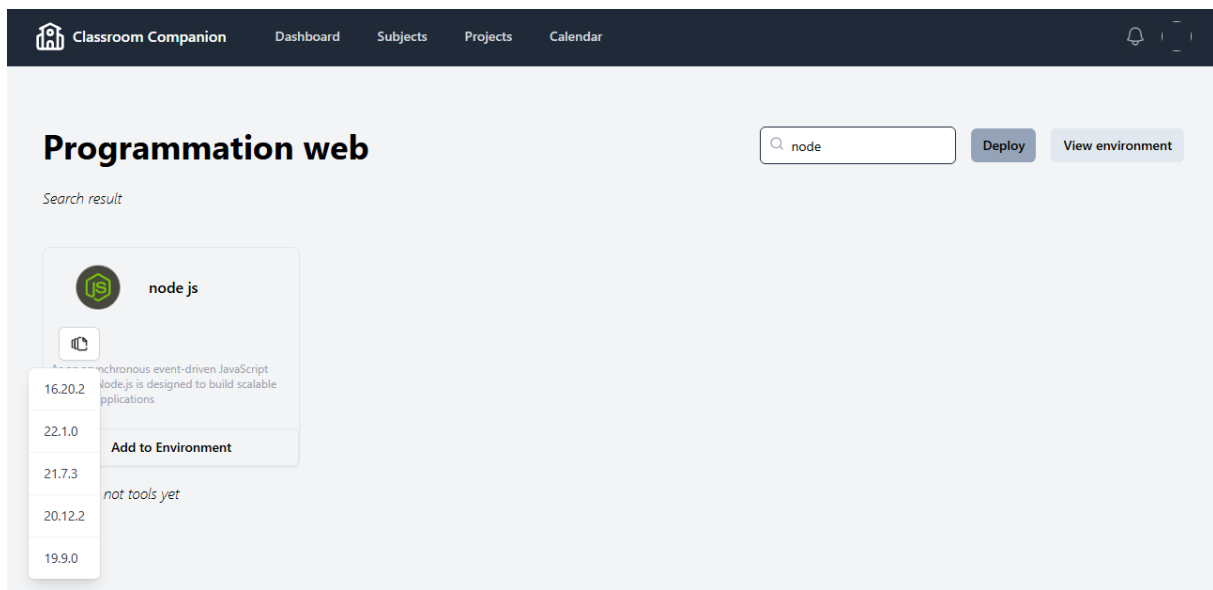


Figure 4.7: Result of searching for tools

In 4.8, the teacher selects Python 3.1 and Node 16.20 as the necessary tools for the web development subject

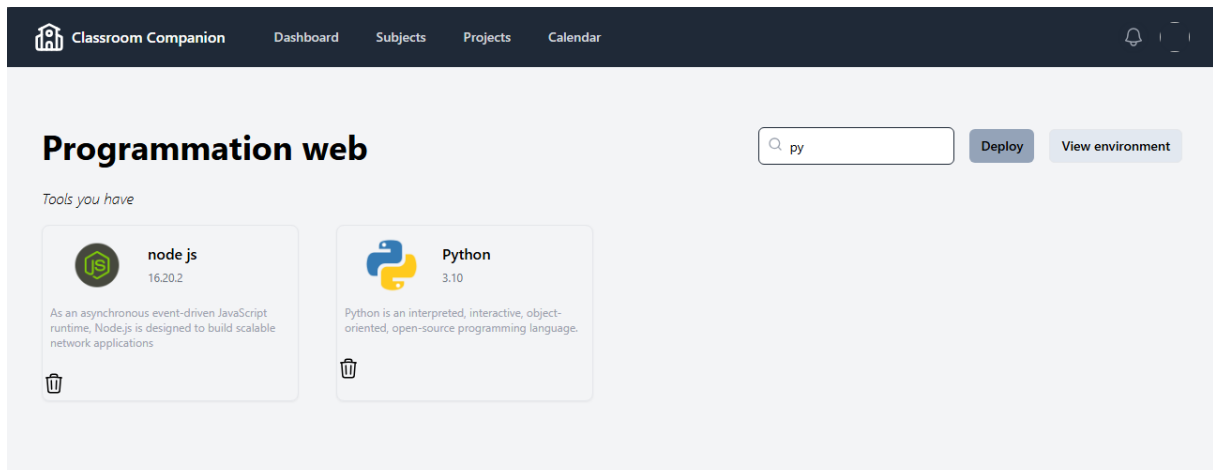


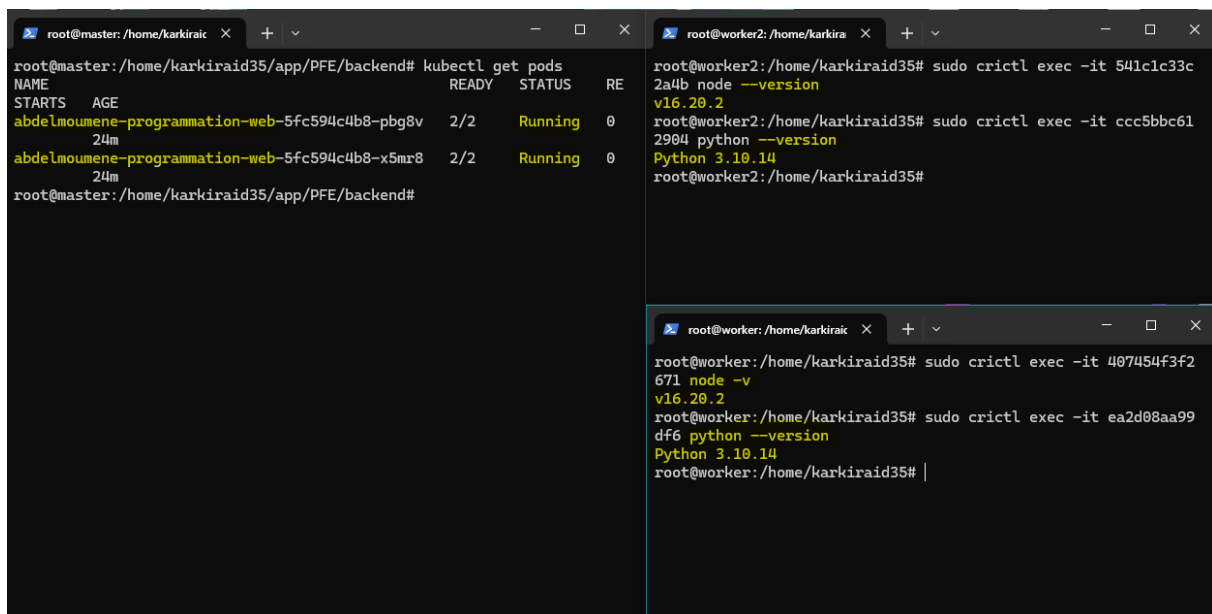
Figure 4.8: interface for subject environment

After selecting the desired tools, the next step is to deploy this environment in the Kubernetes cluster. In this prototype, we focus on demonstrating how containerization

can address the problem, so we developed only the essential components. This can be done by pressing the deploy button shown in the previous figure 4.8.

4.3.2.5 Test deployed environment

After pressing the deploy button, the core system will create and send a manifest.yaml file to the control plane of the Kubernetes cluster. The control plane will then officially deploy the environment on the worker nodes.



```
root@master:/home/karkiraic x + v - □ x
root@master:/home/karkiraid35/app/PFE/backend# kubectl get pods
NAME                                READY  STATUS  RE
STARTS  AGE
abdelmoumene-programmation-web-5fc594c4b8-pbg8v  2/2    Running  0
24m
abdelmoumene-programmation-web-5fc594c4b8-x5mr8  2/2    Running  0
24m
root@master:/home/karkiraid35/app/PFE/backend#

root@worker2:/home/karkiraid35# sudo crictl exec -it 541c1c33c2a4b node --version
v16.20.2
root@worker2:/home/karkiraid35# sudo crictl exec -it ccc5bbc612904 python --version
Python 3.10.14
root@worker2:/home/karkiraid35#

root@worker:/home/karkiraid35# sudo crictl exec -it 407454f3f2671 node -v
v16.20.2
root@worker:/home/karkiraid35# sudo crictl exec -it ea2d08aa99df6 python --version
Python 3.10.14
root@worker:/home/karkiraid35# |
```

Figure 4.9: Kubernetes cluster with environment

Now, we observe on figure 4.9 that the environment the teacher constructed has been successfully deployed across all worker nodes in the Kubernetes cluster. in workers machines we have node js 16.20 and python 3.10

4.4 Conclusion

The implementation chapter details the crucial transition from conceptualization to realization in the development of an educational computing platform leveraging containerization. Beginning with the setup of a robust Kubernetes cluster on Microsoft Azure, the chapter meticulously guides through each step—from configuring virtual machines to installing necessary software and ensuring network connectivity. This foundation is essential for creating a scalable and efficient environment capable of supporting educational modules.

Moving forward, the development of the Classroom Companion web application using the MERN stack is explored. This application serves as an integral part of the platform, facilitating the management of teacher credentials, subject assignments, and session environments. Key technologies such as MongoDB, Express.js, React, and Node.js are employed to ensure flexibility, scalability, and responsiveness in both backend and frontend development.

Throughout the chapter, emphasis is placed on integrating security measures, monitoring, and logging to uphold system reliability. The deployment process using Docker containers and Kubernetes showcases the platform’s capability to manage diverse educational environments effectively. By the conclusion of this chapter, a fully functional beta platform emerges, poised to address the outlined challenges and meet the educational needs of its users effectively.

General conclusion

The journey from conceptualization to implementation of our educational computing platform using containerization has been both challenging and rewarding. In this thesis, we embarked on a comprehensive exploration of the design, development, and deployment phases, aiming to address the critical needs of modern educational environments.

Conceptual Framework and Design: We began by outlining a robust conceptual framework that identified key challenges in educational technology, emphasizing scalability, efficiency, and security. The design phase meticulously translated these conceptual ideals into a structured system architecture, centered around Kubernetes for container orchestration.

Implementation Phase: Transitioning from design to implementation, we navigated through the setup of a Kubernetes cluster on Microsoft Azure, configuring virtual machines, installing necessary software, and ensuring network connectivity. This foundational step was crucial in creating a resilient and scalable environment capable of supporting diverse educational modules.

Classroom Companion Application: A pivotal component of our platform, the Classroom Companion application, was developed using the MERN stack. This web application empowered educators to manage sessions, deploy customized learning environments using containerized tools, and foster collaborative learning experiences seamlessly.

Integration of Security and Reliability Measures: Throughout the implementation, we integrated robust security measures, monitoring tools, and logging mechanisms to safeguard data integrity and ensure system reliability. This holistic approach aimed to build trust among users and maintain operational continuity.

Deployment and Future Directions: Leveraging Docker containers and Kubernetes, we successfully deployed our educational platform, demonstrating its capability to manage complex educational environments efficiently. Looking ahead, further enhancements could focus on optimizing resource allocation, enhancing user interfaces, and expanding support for additional educational tools and modules.

What I learned: In this thesis, I gained extensive knowledge, particularly in the areas of containers, Docker, Kubernetes. I deepened my understanding of how containerization

enhances scalability, efficiency, within software environments. Moreover, I honed my skills in setting up Kubernetes clusters, configuring virtual machines on cloud platforms like Microsoft Azure, deploying application in production environments,

The research and exploration into these topics also introduced me to DevOps practices and microservices architecture. Understanding their role in modern software development and deployment has sparked my interest in further exploring these fields.

This experience not only enriched my technical expertise but also underscored the importance of adaptive learning and continuous improvement in the rapidly evolving field of educational technology.

Future Work: As we conclude this thesis, we recognize the ongoing evolution of technology and educational practices. Future work could :

- Explore integrating CI/CD pipelines for automated deployment and management of educational workloads.
- Develop a seamless solution to integrate IDEs like VS Code into the platform for enhanced development environments.
- Implement a robust system for securely storing and accessing projects of both teachers and students remotely.
- Extend the platform's capabilities to department machines for broader institutional use.
- Enhance security measures across the system to safeguard data and user interactions effectively.

In conclusion, this thesis has laid the groundwork for a scalable, efficient, and secure educational computing platform powered by containerization. By addressing the outlined challenges and leveraging cutting-edge technologies, we've envisioned a platform capable of meeting the evolving needs of educational institutions and empowering educators and students alike.

Bibliography

- [1] Cgroups <https://fr.wikipedia.org/wiki/cgroups>.
- [2] <https://docs.docker.com/get-started/overview/>.
- [3] Nick Janetakis. Understanding how the docker daemon and docker cli work together. May 2017.
- [4] Docker desktop and windows – what’s the best option for you?
- [5] Huawei. Huawei. 09 2023.
- [6] Krish Sivanathan. packagecloud, 04 2022.
- [7] docker-image-vs-container-vs-dockerfile.
- [8] kubernetes, 10 2023. Accessed: 2024-04-14.
- [9] mehmet ozkaya. Scalability — vertical or horizontal scaling when designing architectures. *medium*, 2021.
- [10] Donald Le. betterstack, 12 2023. Accessed: 2024-04-14.
- [11] C. H. Devassy. Mastering kvm virtualisation. *Mastering KVM virtualisation*, 2016.
- [12] Laurent. Déclaration d’amour : la virtualisation d’un serveur. Mar 2014.
- [13] G. Manjunath and D. Sitaram. Moving to the cloud developing apps in the new word of cloud computing. *Moving to the cloud developing apps in the new word of cloud computing*, 2012.
- [14] Hewlett Packard Enterprise. hewlett packard.
- [15] P. K. Dang. researchgate https://www.researchgate.net/publication/273119623_embedded_virtualiza
- [16] application-virtualization <https://www.appviewx.com/education-center/application-virtualization/>.

- [17] P. Meher. <https://www.linkedin.com/pulse/virtualization-its-types-meher-pranav/>.
- [18] IBM. Ibm containerization topic.
- [19] A. Singh and Abhinav. </what-is-the-difference-between-containerization-docker-and-virtualization-vmware-virtualbox-xen>. *Quora*.
- [20] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. *IEEE International Conference on Cloud Engineering*, 2014.
- [21] S. Kumaran. Practical lxc and lxd linux containers for virtualisation and orchestrations. *Practical LXC and LXD linux containers for virtualisation and orchestrations*, 2017.
- [22] wikipedia [https://en.wikipedia.org/wiki/docker_\(software\)](https://en.wikipedia.org/wiki/docker_(software)).
- [23] Emily E. Mell. Tech target. June 2023.
- [24] the-difference-between-docker-images-and-containers.
- [25] Vmware. Accessed: 2024-04-14.
- [26] What are the benefits of container orchestration?
- [27] Docker. Accessed: 2024-04-14.
- [28] Christophe Bardy. Lemagit. Accessed: 2024-04-14.
- [29] Openshift. Accessed: 2024-04-14.
- [30] Amazon elastic container service. Accessed: 2024-04-14.
- [31] himanshu agrawal. *Kubernetes fundamentals: A Step-by-Step Development and interview guide*. apress.
- [32] ousama mostafa. *A Complete Guide to DevOps with AWS*. apres, 2023.
- [33] Learning environment <https://kubernetes.io/docs/tasks/tools/>.