A Project Report on

Orchestrating use of Open Source frameworks in Developing Dynamically Scalable Container based Lab Environment for Technical Institutes

Submitted in partial fulfillment of the requirements for the award
of the degree of

Bachelor of Engineering

in

Information Technology

by

Sanskruti Mhatre(22204014)
Bharat Sharma(21104023)
Shreyash Pawar(22204002)
Ammar Nagarji(22204011)

Under the Guidance of

Dr. Kiran Deshpande
Ms. Charul Singh

Department of Information Technology
NBA Accredited
A.P. Shah Institute of Technology
G.B.Road,Kasarvadavli, Thane(W)-400615
UNIVERSITY OF MUMBAI

Academic Year 2024-2025

Approval Sheet

This Project Report entitled ***'Orchestrating use of Open Source frameworks in Developing Dynamically Scalable Container based Lab Environment for Technical Institutes"*** Submitted by ***"Sanskruti Mhatre"(22204014),"Bharat Sharma" (21104023),"Shreyash Pawar"(22204002),"Ammar Nagarji"(22204011)*** is approved for the partial fulfillment of the requirenment for the award of the degree of ***Bachelor of Engineering*** in ***Information Technology*** from ***University of Mumbai***.

(Ms. Charul Singh)
Co-Guide

(Dr. Kiran Deshpande)
Guide

Dr. Kiran Deshpande
HOD, Information Technology

Place:A.P.Shah Institute of Technology, Thane
Date:

CERTIFICATE

This is to certify that the project entitled ***""Orchestrating use of Open Source frameworks in Developing Dynamically Scalable Container based Lab Environment for Technical Institutes"*** submitted by ***"Sanskruti Mhatre"(22204014),"Bharat Sharma"(21104023),"Shreyash Pawar"(22204002),"Ammar Nagarji"(22204011)*** for the partial fulfillment of the requirement for award of a degree ***Bachelor of Engineering*** in ***Information Technology***, to the University of Mumbai, is a bonafide work carried out during academic year 2024-2025.

Ms. Charul Singh
Co-Guide

Dr. Kiran Deshpande
Guide

Dr. Kiran Deshpande
HOD, Information Technology

Dr. Uttam D.Kolekar
Principal

External Examiner(s)

1.

2.

Internal Examiner(s)

1.

2.

Place:A.P.Shah Institute of Technology, Thane
Date:

Acknowledgement

We have great pleasure in presenting the report on **Orchestrating use of Open Source frameworks in Developing Dynamically Scalable Container based Lab Environment for Technical Institutes.** We take this opportunity to express our sincere thanks towards our guide **Dr. Kiran Deshpande** & Co-Guide **Ms. Charul Singh** for providing the technical guidelines and suggestions regarding line of work. We would like to express our gratitude towards their constant encouragement, support and guidance throughout the development of project.

We thank **Dr. Kiran B. Deshpande** Head of Department for his encouragement during the progress meeting and for providing guidelines to write this report.

We express our gratitude towards BE project co-ordinators, for being encouraging throughout the course and for their guidance.

We also thank the entire staff of APSIT for their invaluable help rendered during the course of this work. We wish to express our deep gratitude towards all our colleagues of APSIT for their encouragement.

Sanskruti Mhatre
(22204014)

Bharat Sharma
(21104023)

Shreyash Pawar
(22204002)

Ammar Nagarji
(22204011)

Declaration

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, We have adequately cited and referenced the original sources. We also declare that We have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

(Signature)
Sanskruti Mhatre(22204014)

(Signature)
Shreyash Pawar(22204002)

(Signature)
Ammar Nagarji(22204011)

(Signature)
Bharat Sharma(21104023)

Date:

Abstract

Educational institutions face significant challenges in maintaining scalable, up-to-date lab environments for hands-on engineering and technology education. Traditional methods of manual software installation and configuration are time-consuming, error-prone, and lack adaptability to rapidly evolving IT trends. These limitations hinder institutions from providing consistent, reproducible, and isolated lab setups, essential for teaching emerging technologies. Addressing these issues requires a dynamic, automated solution capable of reducing manual effort while ensuring flexibility and future readiness.

To overcome these challenges, this paper proposes a Kubernetes-based orchestration platform for automating containerized lab environments. Kubernetes clusters, deployed on local institutional servers and cloud platforms, enable dynamic scaling and management of software setups. Horizontal scaling expands cluster capacity by adding nodes, while vertical scaling optimizes resource allocation for existing containers. By leveraging containerization, Kubernetes ensures reproducibility, reduces setup complexity, and accelerates lab provisioning.

The integration of open-source platforms enhances the system's flexibility, cost-efficiency, and accessibility. Students gain isolated, reliable environments to experiment with cutting-edge tools, aligning their skills with industry demands. The solution's scalability and automation ensures future ready lab environment taking into consideration rapid evolution in IT landscape. Ultimately, the Kubernetes-driven infrastructure equips students with practical experience in modern IT ecosystems, preparing them to thrive in a rapidly changing technological landscape.

Contents

List of Figures

List of Abbreviations

AI:	Artificial Intelligence
API:	Application Programming Interface
CI/CD:	Continuous Integration / Continuous Deployment
CPU:	Central Processing Unit
DNS:	Domain Name System
GCP:	Google Cloud Platform
GUI:	Graphical User Interface
HPA:	Horizontal Pod Autoscaler
HTTP:	Hypertext Transfer Protocol
IoT:	Internet of Things
IT:	Information Technology
JSON:	JavaScript Object Notation
K8s:	Kubernetes
LMS:	Learning Management System
ML:	Machine Learning
OS:	Operating System
RBAC:	Role-Based Access Control
URL:	Uniform Resource Locator
VM:	Virtual Machine
YAML:	YAML Ain't Markup Language
CLI:	Command Line Interface
GPU:	Graphics Processing Unit
SSL:	Secure Sockets Layer
TLS:	Transport Layer Security
CRUD:	Create, Read, Update, Delete
DO:	Digital Ocean
DR:	Docker Registry

Chapter 1

Introduction

The project titled Orchestrating use of Open Source frameworks in Developing Dynamically Scalable Container based Lab Environment for Technical Institutes aims to transform the way academic laboratories operate by addressing the technical challenges that arise from inconsistent software environments, complex tool setups, and time-consuming configurations. In academic settings, particularly in labs requiring advanced technological set-ups, students are often required to install, configure, and manage a variety of software tools to complete practical lab exercises. However, differences in operating systems, hardware, and individual configurations can cause unpredictable behavior in lab exercises, leading to frustration for both students and instructors.

To address these challenges, our project proposes a framework that leverages Docker containerization and Kubernetes orchestration to create a dynamically scalable, container-based lab environment. Docker containers encapsulate software tools, libraries, and dependencies into portable, isolated environments, ensuring uniformity across all student devices. Kubernetes orchestrates these containers, dynamically scaling resources to meet computational demands while optimizing infrastructure utilization. This approach eliminates variability caused by heterogeneous hardware or operating systems and simplifies the deployment of lab environments across local machines, on-premises data centers, and cloud platforms.

By using this system, students can focus on the core educational content of the lab exercises, instead of spending time troubleshooting software installation issues or dealing with compatibility errors. This not only enhances the learning experience but also allows instructors to concentrate on teaching rather than assisting students with technical problems. Furthermore, the project provides a scalable solution that can handle complex, resource-intensive labs in domains like Networking, Security, Blokchain, DevOps, Web X.0, and many more enabling creation of more advanced and dynamic lab environments with ease.

To achieve dynamic scalability in the proposed framework, the system leverages Kubernetes' Horizontal Pod Autoscaler (HPA), a resource-driven orchestration tool that automatically adjusts the number of active container instances (pods) based on real-time demand. Unlike static scaling, which relies on preallocated resources, HPA monitors application-specific metrics such as CPU utilization, memory consumption, or custom metrics and scales pods horizontally to maintain performance thresholds. By integrating HPA with declarative Kubernetes manifests, the framework enforces policies like target average utilization and maximum replica limits, enabling institutions to balance performance, cost, and infrastructure efficiency across hybrid environments (on-premises servers and cloud platforms).

Our project titled Orchestrating use of Open Source frameworks in Developing Dynamically Scalable Container based Lab Environment for Technical Institutes presents a powerful solution for managing technical institute laboratories. It brings uniformity across systems and drastically reduces the time and effort required to set up lab infrastructure. This approach not only enhances operational efficiency but also improves the overall learning experience for students by providing consistent, ready-to-use environments. Instructors benefit from greater flexibility, as they can easily modify, update, or customize lab setups to align with evolving course content or technical requirements. Additionally, container technology enables seamless deployment of large applications using container clusters, which can be efficiently managed through orchestrators like Kubernetes. These environments can be deployed on various platforms—whether it's a developer's laptop, an on-premises server, or a cloud-based infrastructure—allowing for easy testing, deployment, and scaling. This adaptability supports continuous integration and delivery, making it ideal for modern educational practices that demand agility and innovation.

Key Advantages of the Proposed Framework

- **Consistency**: Docker containers ensure that all students work in an identical environment, eliminating compatibility issues. This consistency is particularly important in technical labs where software behavior can vary significantly across different operating systems and hardware configurations.

- **Scalability**: Kubernetes allows the lab environment to scale dynamically based on resource usage, ensuring optimal performance. This is especially useful for resource-intensive labs, such as those involving machine learning, data analysis, or network simulations, where computational demands can vary widely.

- **Ease of Management**: Instructors can easily update or customize lab environments as needed, adapting to evolving course requirements. For example, if a new version of a software tool is released, instructors can quickly update the Docker image and deploy it across the entire lab environment without manual intervention.

- **Remote Access over varying OS environments**: Students can access lab environments on popular OS environments such as Linux and Windows from any location, enabling remote learning. This is particularly beneficial for students who may not have access to high-performance hardware or specific software tools on their personal devices.

- **Cost Efficiency**: By leveraging open-source technologies and kubernetes based container orchestration, the system reduces the need for expensive hardware upgrades and minimizes resource wastage. Kubernetes ensures that resources are allocated efficiently, reducing operational costs for educational institutions.

- **Improved Learning Experience**: With a consistent and scalable lab environment, students can focus on learning rather than troubleshooting technical issues. This leads to a more engaging and productive learning experience, ultimately improving educational outcomes.

1.1 Motivation

The motivation for this project stems from the various challenges faced in academic lab environments, particularly those that involve complex software setups. In traditional labs, students are often required to install and configure numerous tools on their personal devices, leading to a host of issues related to compatibility, configuration, and user experience. These key challenges have been identified as critical roadblocks in delivering effective and engaging lab sessions.

- **Inconsistent Software Environments:** Different operating systems and configurations on student devices lead to unpredictable behavior in lab exercises. This inconsistency can cause technical issues, hindering the learning experience and leaving some students behind.

- **Time-Consuming Setup:** Installation and configuration of complex software tools for labs, such as those in networking or security, can take significant time and often results in technical difficulties. This consumes valuable lab time that could be better spent on learning.

- **Complex Tools:** Many students are unfamiliar with the advanced tools required for lab exercises, leading to a steep learning curve.

- **Limited Instructor Support:** Instructors often spend a lot of time on troubleshooting individual technical issues during crucial instructional hours.

1.2 Problem Statement

In academic labs, students are often required to use advanced software tools for hands-on learning. However, the wide range of personal devices, operating systems, and software versions makes it difficult to maintain a uniform setup for all students. This inconsistency often results in technical issues like compatibility errors, version mismatches, and installation failures. Such problems hinder the learning process and cause unnecessary frustration for both learners and educators. Instead of focusing on the learning, valuable time is spent resolving these setup challenges. This project proposes a solution by delivering a standardized environment that comes pre-configured with all necessary tools and dependencies. With this streamlined setup, every student can work in a consistent, reliable environment, free from technical obstacles. As a result, students can engage more effectively with the learning, and instructors can concentrate on delivering quality education rather than fixing setup related issues.

1.3 Objectives

After in depth literature survey, discussions, project meetings, the project intends to achieve the following objectives.

- To model and orchestrate a container-based IT lab environment providing hassle-free lab setup for students.

- To provide dynamic scalability to the container-based lab environment through a Kubernetes cluster deployed over the cloud.

- To model and build a user-friendly interface for using a container-based lab environment created which will provide ease of use to professors and students during lab sessions.

- To extend the use of a dynamically scalable container-based lab environment created for collaborative project management and assessment.

- To facilitate remote access to lab environments, enabling students to practice lab exercises from anywhere at anytime.

1.4 Scope

The scope of this project focuses on creating versatile, user-friendly Docker containers tailored framework for academic lab environments, enhancing accessibility, and simplifying deployment of software environments for both students and instructors. The framework aims to improve the learning experience by streamlining setup, ensuring consistency, and enabling ongoing refinement. The scope of project implementation is restricted to the following key functionalities.

- To encapsulate the entire lab environment, including essential software, libraries, and dependencies.

- To provide portability and ease of use, allowing students to deploy them quickly on their personal devices.

- To facilitate ecosystem for management and deployment of required software environments reducing individual troubleshooting efforts.

- To incorporate user feedback mechanisms to continuously improve the containerized environments based on student and instructor input.

Chapter 2

Literature Review

The purpose of literature review is to gain an understanding of the existing technologies and research of Container Orchestration and debates relevant to the area of study. The literature review helped in selecting the appropriate open source platforms required, taking into account various deployment approaches of the proposed solution.

1. In Paper [4] J. Shah and D. Dubaria state that developing and building a modern cloud infrastructure or DevOps implementation, than both Docker and Kubernetes have revolutionized the era of software development and operations. Although both are different, they unify the process of development and integration, it is now possible to build any architecture by using these technologies. Docker is used to build, ship and run any application anywhere. Docker allows the use of the same available resources. These containers can be used to make deployments much faster. Containers use less space, are reliable and are very fast. Docker Swarm helps to manage the docker container. Kubernetes is an automated container management, deployment and scaling platform. Using Google Cloud Platform to deploy containers on Kubernetes Engine enabling rapid application development and management. Kubernetes provides key features like deployment, easy ways to scale, and monitoring.

2. In paper [5] Lily Puspa Dewi, Agustinus Noertjahyana, Henry Novianus Palit, and Kezia Yedu- tun emphasize that, an enterprise that has implemented virtualization can consolidate multiple servers into fewer host servers and get the benefits of reduced space, power, and administrative requirements. Sharing their hosts' operating system resources, containerization significantly reduces workloads, and is known as a lightweight virtualization. Kubernetes is commonly used to automatically deploy and scale application containers. The scalability of these application containers can be applied to Kubernetes with several supporting parameters. It is expected that the exploitation of scalability will improve performance and server response time to users without reducing server utility capabilities. This research focuses on applying the scalability in Kubernetes and evaluating its performance on overcoming the increasing number of concurrent users accessing academic data. This research employed 3 computers: one computer as the master node and two others as worker nodes. Simulations are performed by an application that generates multiple user behaviours accessing various microservice URLs. Two scenarios were designed to evaluate the CPU load on single and multiple servers. On multiple servers, the server scalability was enabled to serve the user requests. Implementation of scalability to the containers (on multiple

servers) reduces the CPU usage pod due to the distribution of loads to containers that are scattered in many workers. Besides CPU load, this research also measured the server's response time in responding user requests.

Response time on multiple servers takes longer time than that on single server due to the overhead delay of scaling containers.

3. In paper [6] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek discussed that, microservices represent a new architectural style where small and loosely coupled modules can be developed and deployed independently to compose an application. This architectural style brings various benefits such as maintainability and flexibility in scaling and aims at decreasing downtime in case of failure or upgrade. One of the enablers is Kubernetes, an open-source platform that provides mechanisms for deploying, maintaining, and scaling containerized applications across a cluster of hosts. Moreover, Kubernetes enables healing through failure recovery actions to improve the availability of applications. As our ultimate goal is to devise architectures to enable high availability (HA) with Kubernetes for microservice based applications, in this paper we examine the availability achievable through Kubernetes under its default configuration. We have conducted a set of experiments which show that the service outage can be significantly higher than expected.

4. In paper [7] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira states that container-based virtualization technologies have gained significant attraction in recent years across Cloud platforms and this will likely continue in the coming years. As such, containers orchestration technologies are becoming indispensable. Kubernetes has become the de facto standard because of its robustness, maturity and rich features. To free users of the burden of having to configure and maintain complex Kubernetes infrastructures, but still make use of its functionalities, all major Cloud providers are now offering cloud-native managed Kubernetes alternatives. The goal of this paper is to investigate the performance of containers running in such hosted services. For this purpose, we conduct a series of experimental evaluations of containers to monitor the behavior of system resources including CPU, memory, disk and network. A baseline consisting of a manually deployed Kubernetes cluster was built for comparison. In particular, we consider the Amazon Elastic Container Service for Kubernetes (EKS), Microsoft Azure Kubernetes Service (AKS) and Google Kubernetes Engine (GKE). The Australia-wide NeCTAR Research Cloud was used for the baseline.

5. In paper [8] A. Pereira Ferreira and R. Sinnott discussed container-based virtualization technologies such as Docker and Kubernetes are being adopted by cloud service providers due to their simpler deployment, better performance, and lower memory footprint in relation to hypervisor-based virtualization. Kubernetes supports basic replication for availability, but does not provide strong consistency and may corrupt application state in case there is a fault. This paper presents a state machine replication scheme for Kubernetes that provides high availability and integrity with strong consistency. Replica coordination is offered as a service, with lightweight coupling to applications. Experimental results demonstrate the solution feasibility.

Chapter 3

Project Design

Project design is the foundational phase where the overall structure, strategy, and objectives of a project are planned out. It involves defining the problem, proposing a suitable solution, and outlining the workflow, technologies, and methodologies to be used. A well-thought-out project design ensures clarity, feasibility, and a clear path forward for successful implementation and execution.

3.1 Existing System

As shown in figure 3.1, the current system relies on static infrastructure with predefined resource allocation and lacks dynamic scalability. Applications are deployed in monolithic or manually managed environments, making them less adaptable to changing workloads. Manual monitoring and scaling lead to inefficiencies in performance and resource use. The absence of automated load balancing and autoscaling affects availability and responsiveness under varying user demands.

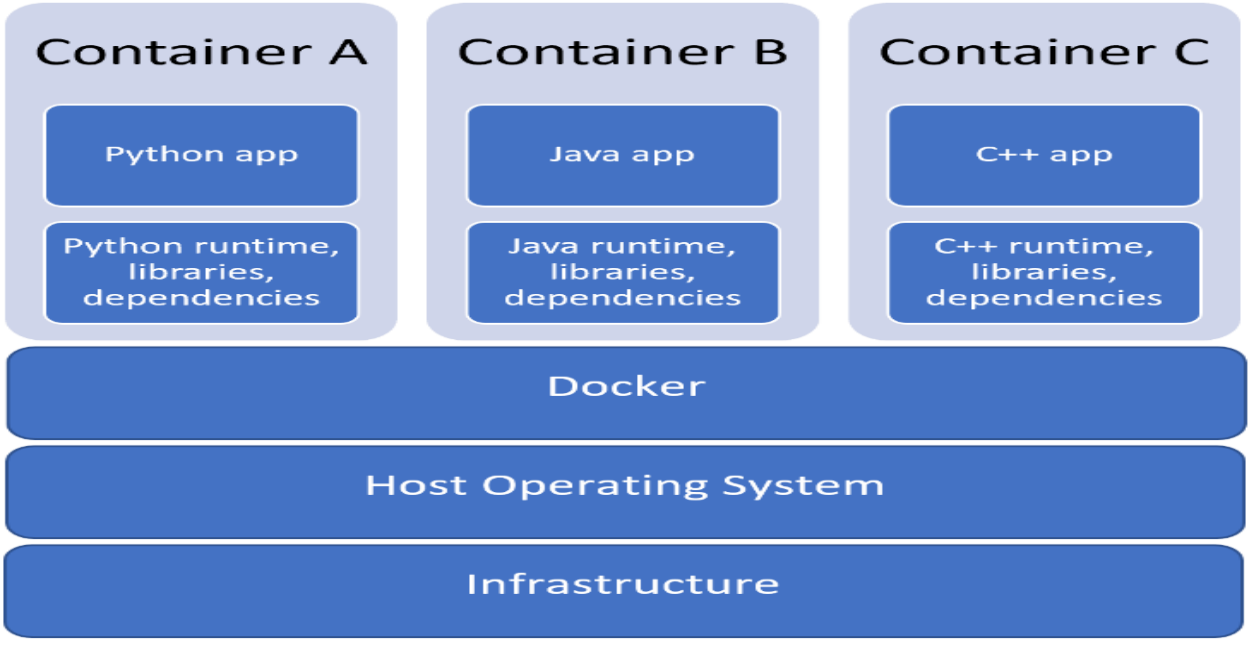

Figure 3.1: Existing System Architecture

3.2 System Architecture

The System Architecture in fig 3.2 outlines the design and structure of a system to address challenges discussed in existing system architecture. It defines the key components, their interactions, and the overall workflow. It aims to enhance scalability, reliability, and performance while ensuring seamless integration with existing technologies.

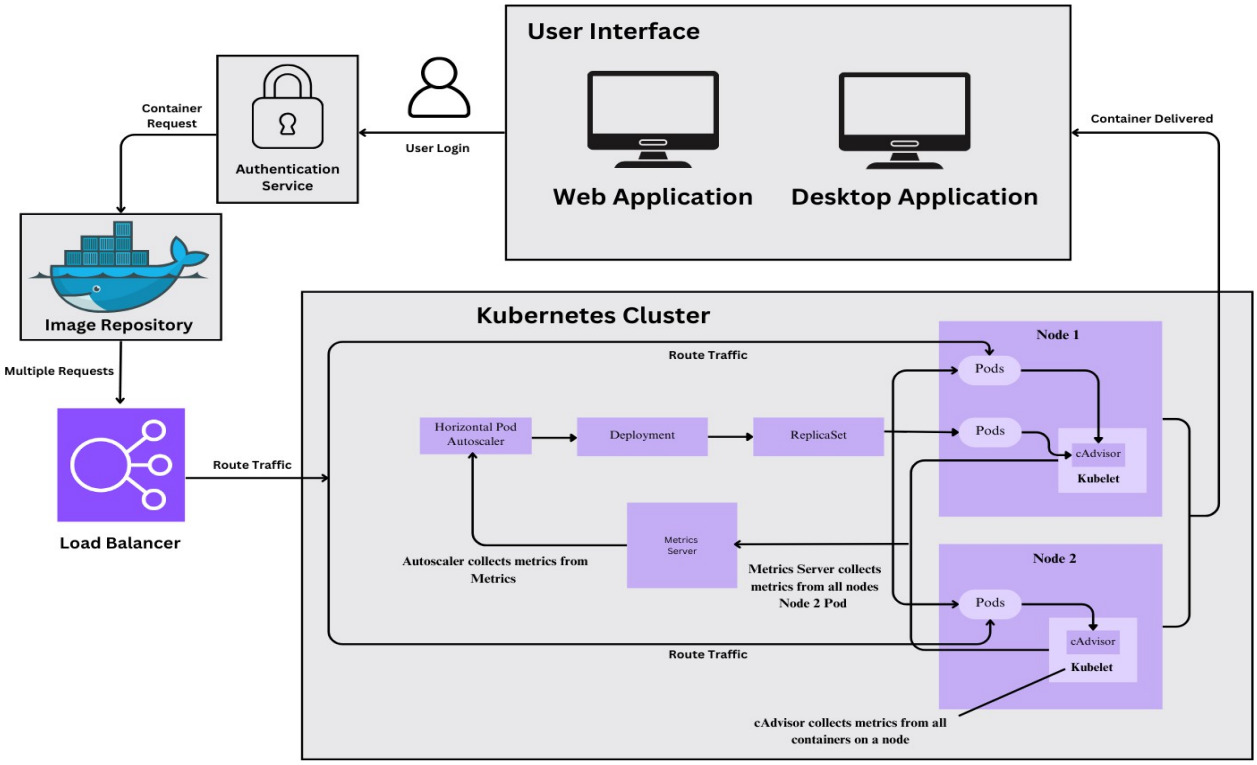

Figure 3.2: System Architecture

3.2.1 Critical Components of System Architecture

The system architecture is designed to deliver a scalable, reliable, and automated lab environment by integrating several critical components.

1. **Containerization with Docker**

   Docker is utilized to containerize applications by packaging all necessary dependencies, libraries, and configurations. This ensures consistency and portability across different environments. The Docker images are stored in an Image Repository, enabling seamless access for deployment. Users authenticate via an Authorization Service before accessing the system. The use of Docker containers eliminates the need for manual software installations, reducing setup time and ensuring that all students work in an identical environment.

2. **Load Balancer**

   A Load Balancer is employed to manage incoming traffic and distribute requests efficiently across the Kubernetes cluster. This ensures high availability, fault tolerance, and optimal resource utilization. It routes multiple requests to the appropriate backend services based on predefined rules. The Load Balancer plays a critical role in ensuring that the system can handle varying workloads, especially during peak usage periods such as lab sessions or exams.

3. **Kubernetes Cluster for Container Orchestration**

   The core of the system relies on a Kubernetes cluster for managing containerized applications. Kubernetes ensures the deployment, scaling, and maintenance of application workloads through automated mechanisms. The Kubernetes cluster consists of a master node and multiple worker nodes, each responsible for running application containers. The master node manages the overall state of the cluster, while the worker nodes execute the tasks assigned to them.

4. **Deployment and Scaling through HPA**

   The system employs a Horizontal Pod Autoscaler (HPA) to dynamically adjust the number of running pods based on real-time resource utilization. The deployment process ensures that application containers are consistently maintained across multiple nodes within the cluster. The HPA monitors CPU and memory usage and scales the number of pods up or down to meet the demand. This ensures that the system can handle sudden spikes in workload without manual intervention.

5. **Metrics Server and Autoscaling**

   A Metrics Server is deployed within the cluster to collect essential performance data, such as CPU and memory usage, from all nodes. This data is used by the HPA to determine whether to scale up or down the number of active pods in the ReplicaSet, optimizing resource utilization. The Metrics Server provides real-time insights into the performance of the cluster, enabling administrators to make informed decisions about resource allocation.

6. **Node Structure and Monitoring**

   The Kubernetes cluster consists of multiple nodes, each running several pods that host application containers. The Kubelet, acting as the node agent, ensures communication with the Kubernetes control plane to manage the state of each node. To enhance monitoring capabilities, cAdvisor (Container Advisor) is deployed on each node to collect real-time metrics from running containers. These metrics are subsequently forwarded to the Metrics Server, which supports autoscaling decisions.

7. **User Interface for ease of use**

   Users interact with the system through web application or desktop application. Upon login via the Authorization Service, authenticated users can access the deployed containers. Once a container request is processed, the system delivers the required container instance to the user, ensuring a seamless and scalable experience. The user interface is designed to be intuitive, allowing students and instructors to easily navigate and access the lab environment.

Additionally following key points are considered while designing the system architecture

1. **Security Considerations**

   Implements security measures such as role-based access control (RBAC), container security policies, and vulnerability scanning to protect against unauthorized access and cyber threats.

2. **Fault Tolerance & Disaster Recovery**

   Ensures backup strategies, failover mechanisms, and automated recovery processes are in place to maintain system reliability during unexpected failures.

3.3 System Diagrams

This section represents UML diagrams describing organized workflow of project design and implementation methods.

The UML diagrams illustrates the streamlined workflow involved in setting up and delivering a dynamically scalable, container-based lab environment for technical institutes using Docker and Kubernetes.

3.3.1 Activity Diagram

The activity diagram outlines the workflow for deploying and managing a container-based lab environment in technical institutes, divided into three primary phases: administrative setup, Kubernetes-driven orchestration, and student interaction.

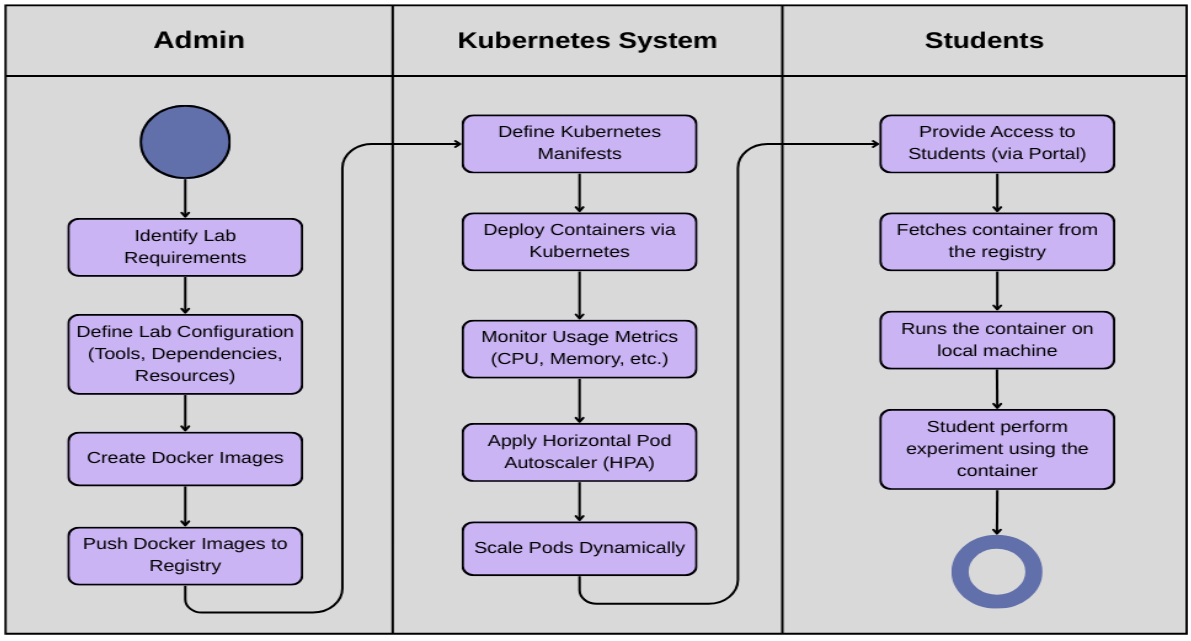

Figure 3.3: Activity Diagram

In the activity diagram shown in figure 3.3, the process starts with the administrator defining the lab requirements, including necessary tools, dependencies, and computational resources. These specifications are used to create Docker images, which encapsulate the lab environment, ensuring consistency across platforms. These images are stored in a centralized registry, such as Docker Hub or a private repository, for easy distribution.

Next, the Kubernetes system takes over. Using declarative configuration files (manifests), the system deploys containers across a cluster, whether on-premises or cloud-based. It continuously monitors resource usage metrics like CPU and memory. The Horizontal Pod Autoscaler (HPA) dynamically scales the number of active container during high-demand tasks like machine learning training or network simulations and scaling down during low usage to optimize costs.

Finally, students interact with the system through a portal, accessing preconfigured lab environments. They pull the Docker images from the registry and run them locally or on remote servers. This setup allows students to perform experiments such as coding, data analysis, or security drills—in a uniform, isolated environment, free from compatibility issues.

By combining Docker's portability with Kubernetes' automation, the system reduces setup time, ensures scalability, and minimizes hardware costs. Instructors benefit from streamlined management of software environments required while students gain reliable access to lab tools, enabling them to focus on learning rather than troubleshooting technical configurations. This approach supports diverse use cases, from local classrooms to remote learning, while maintaining efficiency and adaptability.

3.3.2 Use Case Diagram

A Use Case Diagram is a visual representation of a system's functionality from the user's perspective. It depicts different actors (users or external systems) interacting with various use cases (functionalities or processes) within the system. Use case diagrams help in understanding system requirements, defining user interactions, and identifying potential scenarios. They are commonly used in software engineering to model system behavior and facilitate communication between stakeholders.

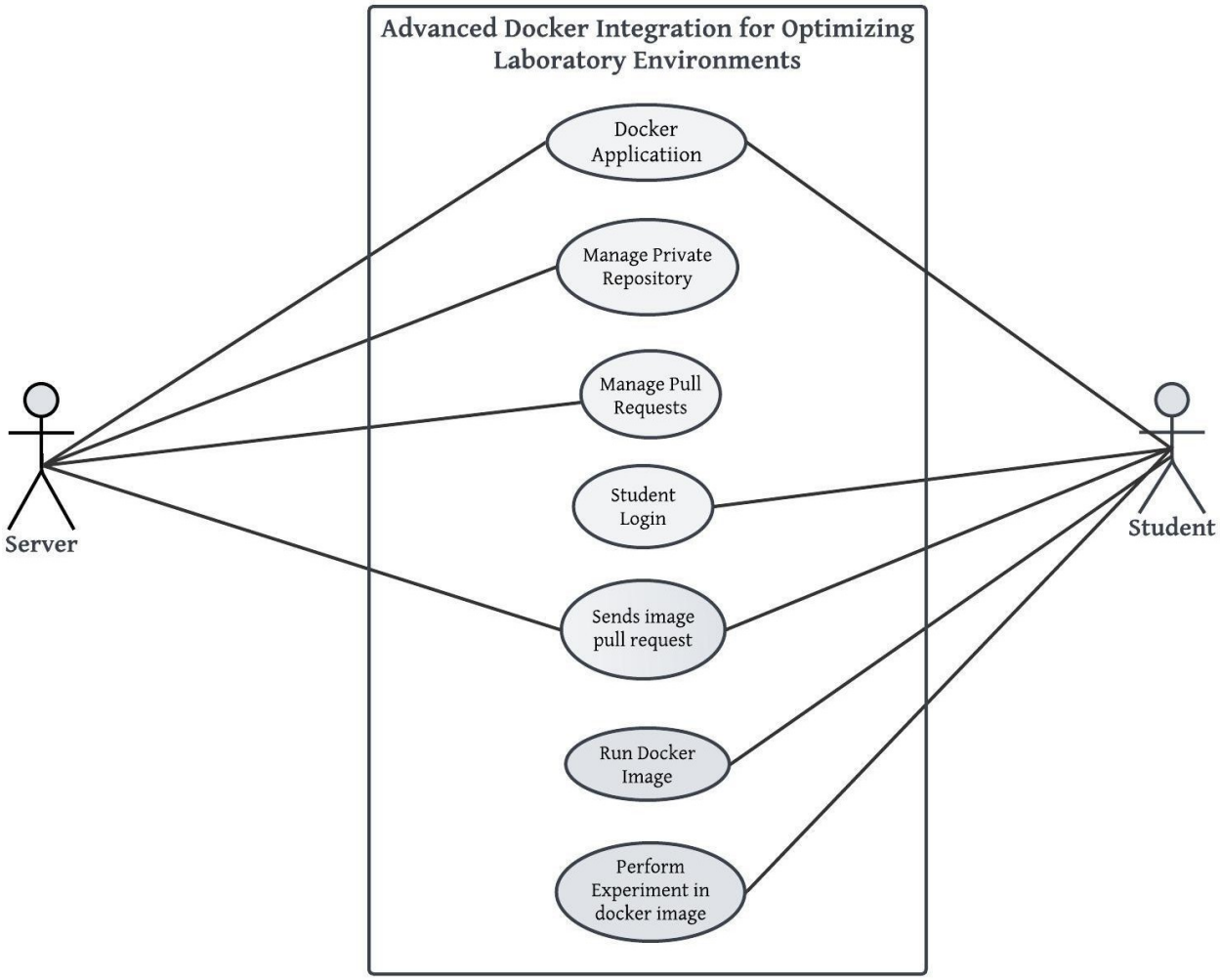

Figure 3.4: Use Case Diagram

The use case diagram shown in figure 3.4 outlines a Docker based framework where students interact with a private repository via a server. Key actions include logging in, requesting Docker images, running containers, and conducting experiments, while the server manages repositories and approves pull requests, optimizing lab workflows for education or research.

3.3.3 Sequence Diagram

A sequence diagram is a type of UML (Unified Modeling Language) diagram that illustrates how objects or components interact with each other in a particular sequence over time. It shows the flow of messages between objects, the order in which they occur, and how each component responds to events within a system. Sequence diagrams are useful for visualizing the interaction between different parts of a system, helping to understand the detailed logic and behavior during specific processes or use cases.

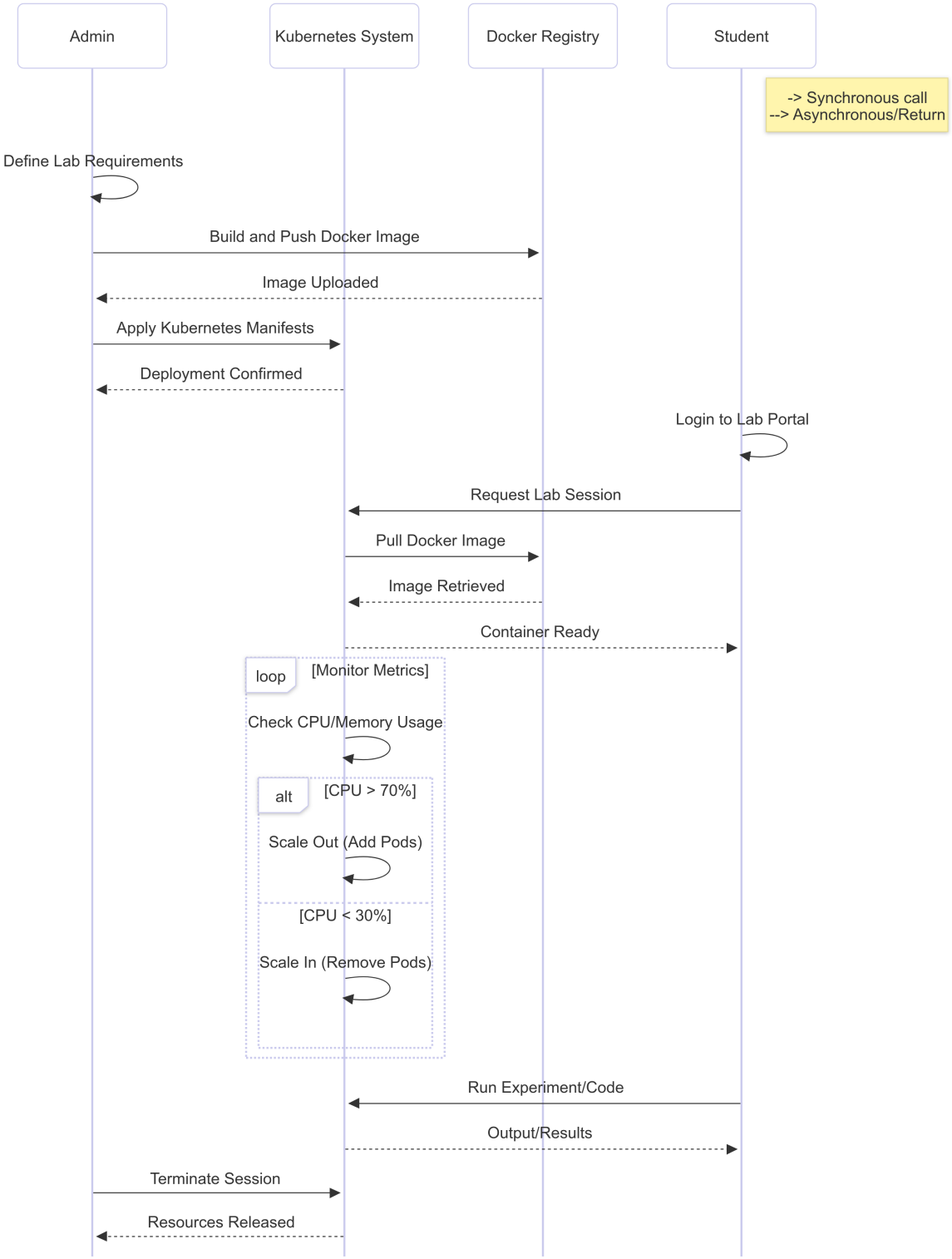

Figure 3.5: Sequence Diagram

The sequence diagram shown in figure 3.5 illustrates the workflow of a container-based lab environment managed through Kubernetes and Docker, designed to streamline academic lab operations.

The workflow starts with the administrator outlining the lab requirements, including essential software tools, dependencies, and system specifications. The administrator then creates a Docker images that packages these environments and uploads it to a Docker registry for centralized and consistent access. Kubernetes is configured using declarative manifests to deploy this environment across clusters, whether hosted locally or on the cloud. When a student logs into the lab through dedicated web portal or desktop application, the system pulls the predefined Docker image from the registry and launches a container instance specifically for that student.

Throughout the lab session, Kubernetes monitors system metrics like CPU and memory usage. If resource demand increases—for instance, CPU usage surpasses 70%—the Horizontal Pod Autoscaler (HPA) automatically adds more pods to manage the load. Conversely, when demand is low, such as CPU usage dropping below 30%, it scales down by reducing the number of active pods to conserve resources. Students perform experiments within their individual containers, running code or simulations and receiving instant results. Once the session ends, the system shuts down the containers and reclaims the used resources.

This entire setup guarantees consistency in student environments, significantly shortens setup time, and allows for real-time resource scaling to maintain both performance and cost-effectiveness. It also supports remote access, enabling students with varying hardware setups to participate equally, and allows instructors to concentrate on delivering content instead of solving technical issues. The combined use of Docker and Kubernetes automates the deployment, scaling, and cleanup processes, resulting in a streamlined, scalable solution for today's tech-driven educational needs.

Chapter 4

Project Implementation

In this section, critical implementation snippets of the project are included to give better understanding of framework deployment. These snippets are highlighting some of the methods utilized to obtain the required outcomes while showcasing the project's main functionality. These snippets represent the flow of critical configurations, command line required during deployment of the framework.

4.1 Configuration Snippets

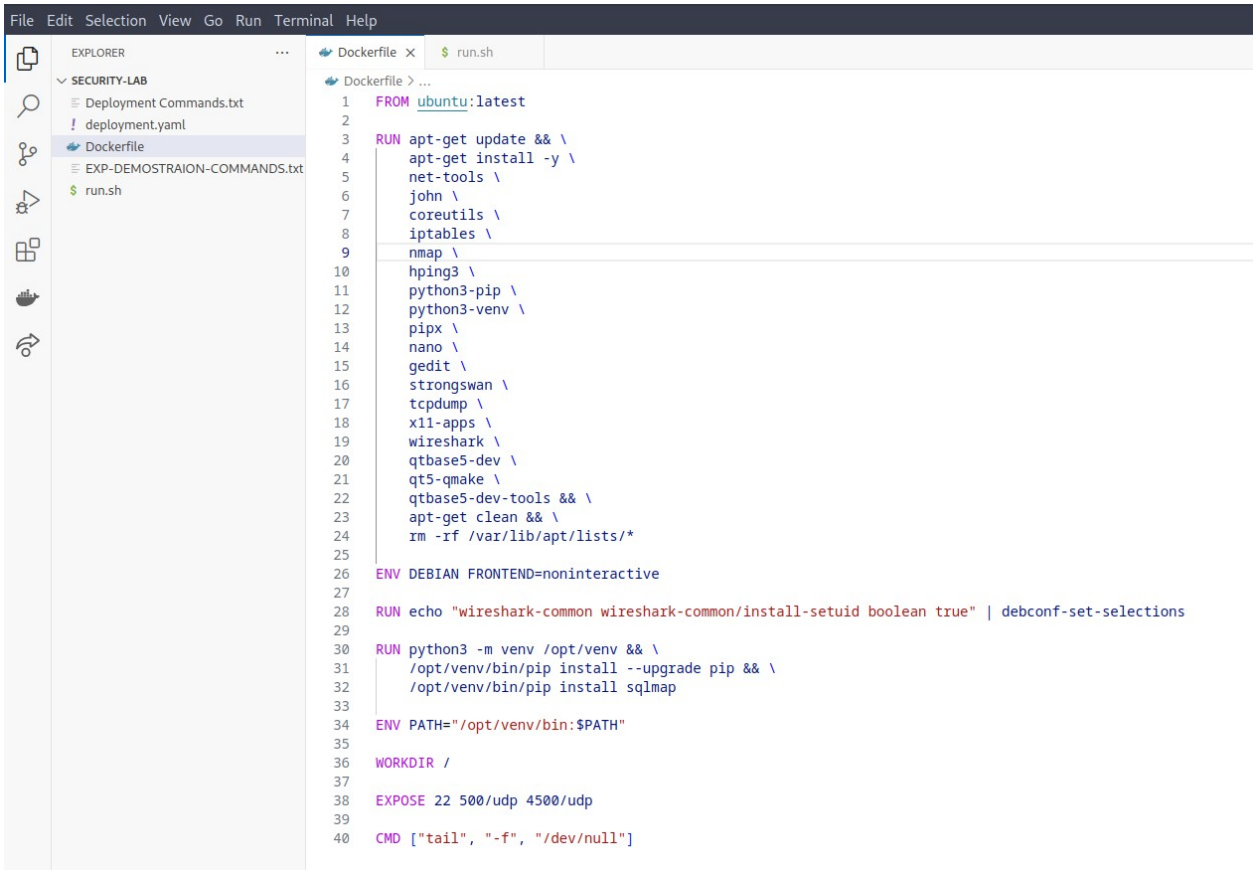

Figure 4.1: Dockerfile Showcasing Process for Creating Docker Image.

The code in the figure 4.1 is used to create a Dockerfile that builds a custom Docker image for Security Lab based on the ubuntu:latest image. It installs various tools and utilities, including network tools (net-tools, nmap), development tools (python3, pip, nano, gedit), and security-related software like john, tcpdump, iptables, and wireshark. The file also sets up a Python virtual environment, installs SQLMap, and configures the environment variables for the system to function properly. Additionally, it exposes specific UDP ports (500, 4500) and runs the container with /bin/bash as the default command.

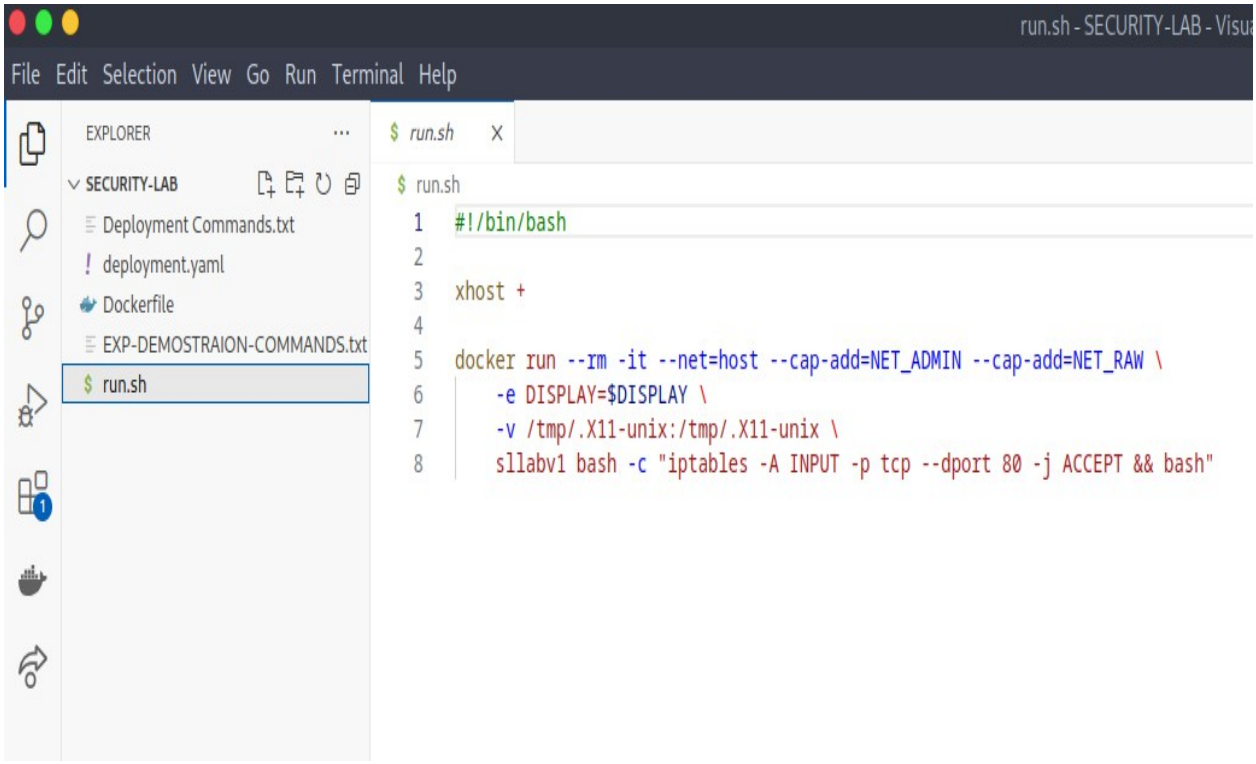

Figure 4.2: Command Line Showcasing Process for Running Docker Image.

The Command Line in the figure 4.2 is a bash script that runs a Docker image, granting the container network access and GUI capabilities using the 'xhost +' command for X server access. It starts the 'sl-lab-8' container with network permissions, sets an iptables rule to allow TCP traffic on port 80, and launches a bash shell within the container.

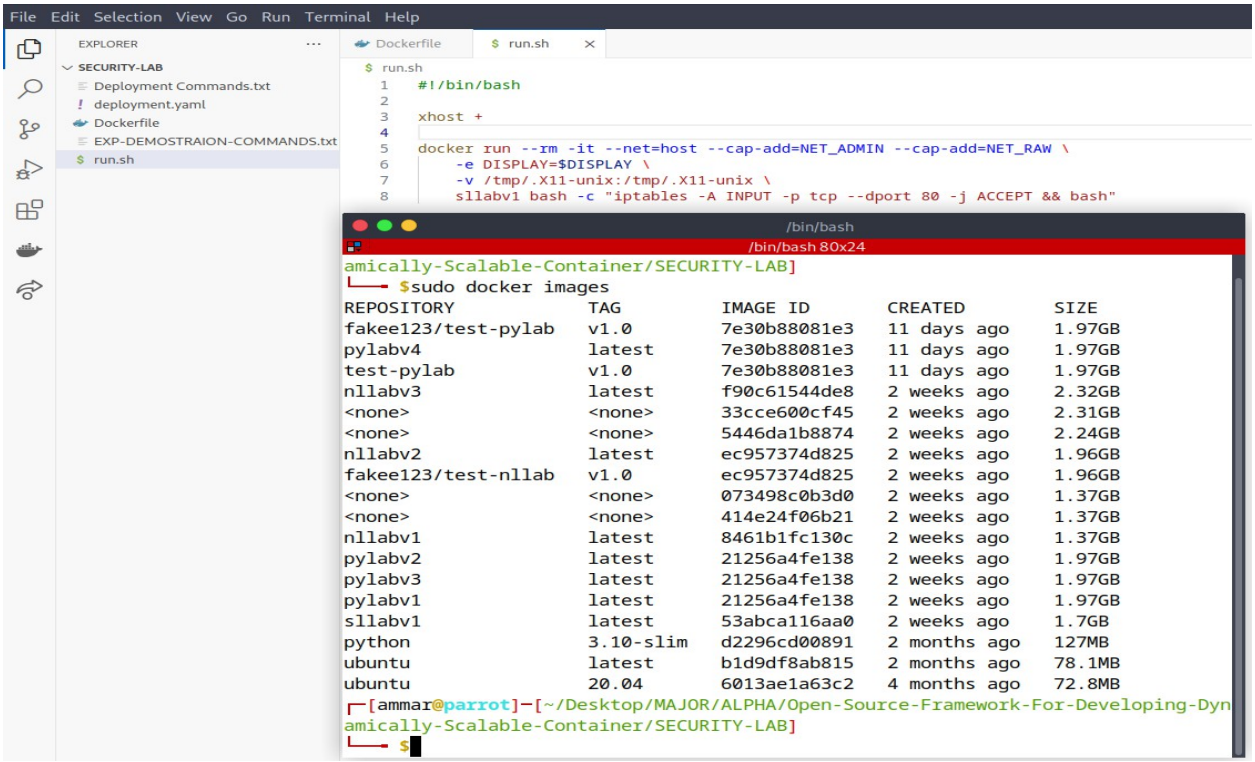

Figure 4.3: Orchastrating Creation of Docker Image.

The Command Line output in the figure 4.3 displays the security lab docker image(sl-lab-8) that are created along with all the other docker images that are present in the system.

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-container-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-container
  template:
    metadata:
      labels:
        app: test-container
    spec:
      containers:
      - name: test-container
        image: registry.digitalocean.com/apsit-repo/test-container:latest
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 200m
            memory: 256Mi
      imagePullSecrets:
      - name: do-registry-cred
---
apiVersion: v1
kind: Service
metadata:
  name: test-container-service
spec:
  selector:
    app: test-container
  ports:
  - port: 80
    targetPort: 80
  type: LoadBalancer
```

Figure 4.4: Deployment Configuration Required for Container Orchestration through Kubernetes Cluster

This Kubernetes Deployment YAML shown in figure 4.4 sets up a containerized application (test-container) using an image hosted on DigitalOcean's container registry. The imagePullSecrets field references the credentials required to securely pull the private image. The container exposes port 80, making it accessible for HTTP traffic, and includes resource limits to ensure efficient use of CPU and memory (capped at 200m CPU and 256Mi memory).

A corresponding LoadBalancer-type Service (test-container-service) is defined to route external traffic to the container on port 80, making the application accessible over the internet. This configuration is suitable for lightweight web services or test environments where controlled resource usage and secure access to private container images are required.

service.yaml

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: test-container-autoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: test-container-deployment
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Figure 4.5: Service Configuration Required for Container Orchestration through Kubernetes Cluster

This Kubernetes Service YAML configuration shown in figure 4.5 dynamically adjusts the number of replicas for the test-container-deployment based on CPU usage. It ensures that the deployment scales between a minimum of 1 and a maximum of 10 pods, maintaining optimal performance and resource efficiency.

The autoscaler monitors CPU utilization, targeting an average usage of 50% across all pods. If CPU load increases beyond this threshold, additional pods are created to handle the load; conversely, if the load drops, pods are scaled down to conserve resources. This configuration is ideal for applications with fluctuating workloads, enabling responsive and cost-effective scaling based on real-time demand.

```
┌─[ammar@parrot]─[~/Desktop/HPA-DO]
└──╼ $kubectl apply -f deployment.yaml
deployment.apps/test-container-deployment created
service/test-container-service created
┌─[ammar@parrot]─[~/Desktop/HPA-DO]
└──╼ $kubectl apply -f service.yaml
horizontalpodautoscaler.autoscaling/test-container-autoscaler created
┌─[ammar@parrot]─[~/Desktop/HPA-DO]
└──╼ $kubectl get pods
NAME                                         READY   STATUS    RESTARTS   AGE
test-container-deployment-54b8988cff-b9vk6   1/1     Running   0          17s
┌─[ammar@parrot]─[~/Desktop/HPA-DO]
└──╼ $kubectl get services
NAME                     TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes               ClusterIP      10.109.0.1      <none>        443/TCP        48s
test-container-service   LoadBalancer   10.109.26.243   <pending>     80:31238/TCP   23s
┌─[ammar@parrot]─[~/Desktop/HPA-DO]
└──╼ $
```

Figure 4.6: Setting-up Deployment and Scaling over Kubernetes Cluster

Figure 4.6 shows a Kubernetes terminal session where a deployment (test-container), and its associated service are applied using YAML files. The output confirms the deployment is running with a ready pod, the service is exposed internally via ClusterIP, and the HPA is configured. The kubectl get pods and kubectl get services commands verify that the resources are active and correctly configured within the cluster.

```
┌─[ammar@parrot]─[~/Desktop/HPA-DO]
└──╼ $kubectl get hpa test-container-autoscaler
NAME                       REFERENCE                                TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
test-container-autoscaler  Deployment/test-container-deployment   cpu: 0%/50%   1         10        1          9m50s
┌─[ammar@parrot]─[~/Desktop/HPA-DO]
└──╼ $kubectl apply -f ab-load-generator.yaml
job.batch/ab-load-generator created
┌─[ammar@parrot]─[~/Desktop/HPA-DO]
└──╼ $
```

Figure 4.7: Kubernetes HPA Load Testing via Port Forwarding and Continuous CPU Requests

Command line in figure 4.7 displays the current status of the Horizontal Pod Autoscaler (HPA) named test-container-autoscaler, which targets the test-container-deployment. The HPA monitors CPU utilization, aiming for 50% usage. As shown, the current usage is 0%, with the number of replicas remaining at the minimum value of 1 (with a maximum limit of 10 replicas).

The ab-load-generator YAML file is applied using kubectl apply, which creates a job to simulate load on the deployment. This setup is intended to increase CPU usage, allowing observation of the HPA's automatic scaling behavior as the system responds to increased demand.

```
┌──[ammar@parrot]─[~]
└──╼ $kubectl get pods -w
NAME                                                READY   STATUS              RESTARTS   AGE
test-container-deployment-54b8988cff-dcnjv          1/1     Running             0          6m8s
ab-load-generator-4r455                             0/1     Pending             0          0s
ab-load-generator-4r455                             0/1     Pending             0          0s
ab-load-generator-4r455                             0/1     ContainerCreating   0          0s
ab-load-generator-4r455                             1/1     Running             0          9s
test-container-deployment-54b8988cff-w967w          0/1     Pending             0          0s
test-container-deployment-54b8988cff-rccbt          0/1     Pending             0          0s
test-container-deployment-54b8988cff-w967w          0/1     Pending             0          0s
test-container-deployment-54b8988cff-64flt          0/1     Pending             0          0s
test-container-deployment-54b8988cff-rccbt          0/1     Pending             0          0s
test-container-deployment-54b8988cff-64flt          0/1     Pending             0          0s
test-container-deployment-54b8988cff-w967w          0/1     ContainerCreating   0          0s
test-container-deployment-54b8988cff-rccbt          0/1     ContainerCreating   0          0s
test-container-deployment-54b8988cff-64flt          0/1     ContainerCreating   0          0s
test-container-deployment-54b8988cff-64flt          1/1     Running             0          3s
test-container-deployment-54b8988cff-w967w          1/1     Running             0          3s
test-container-deployment-54b8988cff-rccbt          1/1     Running             0          3s
test-container-deployment-54b8988cff-v4ml7          0/1     Pending             0          0s
test-container-deployment-54b8988cff-98ppk          0/1     Pending             0          0s
test-container-deployment-54b8988cff-v4ml7          0/1     Pending             0          0s
test-container-deployment-54b8988cff-8r6mx          0/1     Pending             0          0s
test-container-deployment-54b8988cff-98ppk          0/1     Pending             0          0s
test-container-deployment-54b8988cff-8r6mx          0/1     Pending             0          0s
test-container-deployment-54b8988cff-4db2w          0/1     Pending             0          0s
test-container-deployment-54b8988cff-4db2w          0/1     Pending             0          0s
test-container-deployment-54b8988cff-v4ml7          0/1     ContainerCreating   0          0s
test-container-deployment-54b8988cff-98ppk          0/1     ContainerCreating   0          0s
test-container-deployment-54b8988cff-8r6mx          0/1     ContainerCreating   0          0s
test-container-deployment-54b8988cff-4db2w          0/1     ContainerCreating   0          0s
ctest-container-deployment-54b8988cff-98ppk          1/1     Running             0          3s
ztest-container-deployment-54b8988cff-4db2w          1/1     Running             0          5s
test-container-deployment-54b8988cff-8r6mx          1/1     Running             0          5s
test-container-deployment-54b8988cff-v4ml7          1/1     Running             0          5s
^C┌──[✗]─[ammar@parrot]─[~]
└──╼ $
```

Figure 4.8: Real-time Monitoring of Pod Creation During HPA-triggered Scaling

The command shown in the figure 4.8 provides a real-time view of the pod lifecycle as the Horizontal Pod Autoscaler scales the deployment. Initially, only one pod from the test-container-deployment is running. After the load is generated using the ab-load-generator job, the CPU utilization increases, triggering the HPA to scale up the number of pods.

```
┌──[ammar@parrot]─[~]
└──╼ $kubectl get hpa test-container-autoscaler -w
NAME                       REFERENCE                                 TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
test-container-autoscaler  Deployment/test-container-deployment      cpu: 0%/50%     1         10        1          11m
test-container-autoscaler  Deployment/test-container-deployment      cpu: 48%/50%    1         10        1          15m
test-container-autoscaler  Deployment/test-container-deployment      cpu: 175%/50%   1         10        1          16m
test-container-autoscaler  Deployment/test-container-deployment      cpu: 175%/50%   1         10        4          16m
test-container-autoscaler  Deployment/test-container-deployment      cpu: 145%/50%   1         10        4          16m
^C┌──[✗]─[ammar@parrot]─[~]
└──╼ $
```

Figure 4.9: Horizontal Pod Autoscaler (HPA) Scaling Up Based on CPU Utilization over Kubernetes cluster

Figure 4.9 shows the live update of the Horizontal Pod Autoscaler status after the load has been applied. As the CPU utilization increases significantly (up to 175%), the HPA reacts by scaling the number of replicas from the minimum of 1 to 4. The target CPU utilization remains set at 50%, and the autoscaler continues to monitor and adjust replicas accordingly within the defined range of 1 to 10 pods.

```
root@parrot: / 190x40
┌─[ammar@parrot]─[~/Desktop/MAJOR/ALPHA/Open-Source-Framework-For-Developing-Dynamically-Scalable-Container/SECURITY-LAB]
└──╼ $sudo docker images
REPOSITORY            TAG         IMAGE ID       CREATED        SIZE
fakee123/test-pylab   v1.0        7e30b88081e3   11 days ago    1.97GB
pylabv4               latest      7e30b88081e3   11 days ago    1.97GB
test-pylab            v1.0        7e30b88081e3   11 days ago    1.97GB
nllabv3               latest      f90c61544de8   2 weeks ago    2.32GB
<none>                <none>      33cce600cf45   2 weeks ago    2.31GB
<none>                <none>      5446da1b8874   2 weeks ago    2.24GB
nllabv2               latest      ec957374d825   2 weeks ago    1.96GB
fakee123/test-nllab   v1.0        ec957374d825   2 weeks ago    1.96GB
<none>                <none>      073498c0b3d0   2 weeks ago    1.37GB
<none>                <none>      414e24f06b21   2 weeks ago    1.37GB
nllabv1               latest      8461b1fc130c   2 weeks ago    1.37GB
pylabv2               latest      21256a4fe138   2 weeks ago    1.97GB
pylabv3               latest      21256a4fe138   2 weeks ago    1.97GB
pylabv1               latest      21256a4fe138   2 weeks ago    1.97GB
sllabv1               latest      53abca116aa0   2 weeks ago    1.7GB
python                3.10-slim   d2296cd00891   2 months ago   127MB
ubuntu                latest      b1d9df8ab815   2 months ago   78.1MB
ubuntu                20.04       6013ae1a63c2   4 months ago   72.8MB
┌─[ammar@parrot]─[~/Desktop/MAJOR/ALPHA/Open-Source-Framework-For-Developing-Dynamically-Scalable-Container/SECURITY-LAB]
└──╼ $./run.sh
access control disabled, clients can connect from any host
root@parrot:/#
```

Figure 4.10: Accessing Container for Required Software Environment

The Command Line in figure 4.10 shows the execution of a Docker container using ‘sudo ./run.sh‘. The prompt indicates successful root access inside the running container.

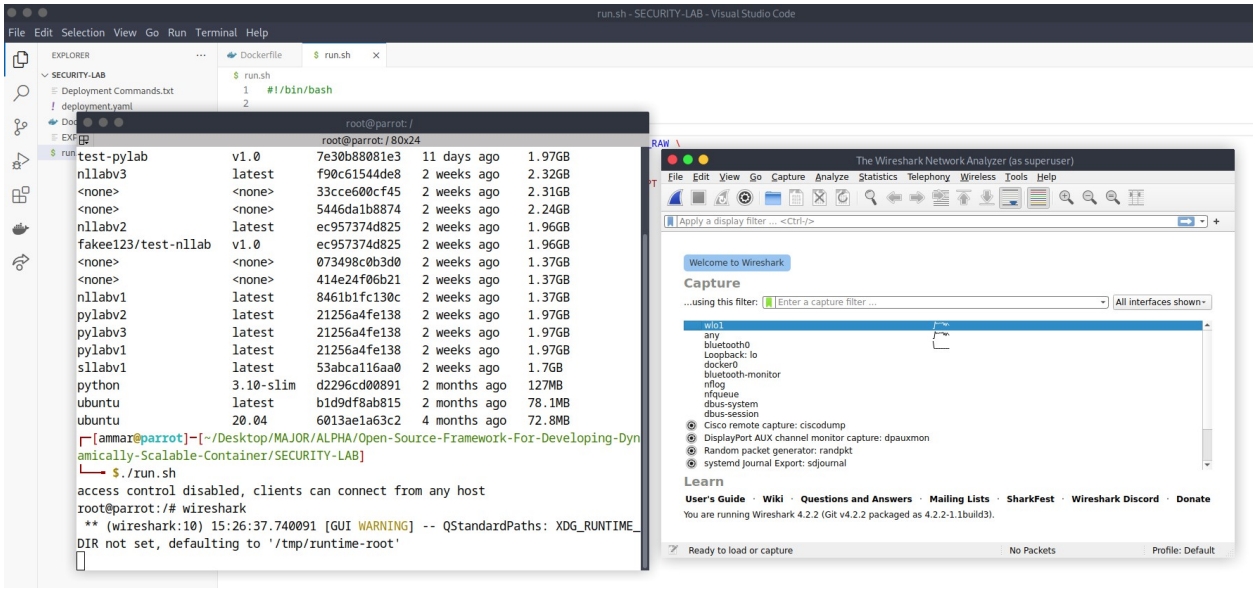

Figure 4.11: Accessing Networking Tools like Wire-Shark trough container Image.

Figure 4.11 shows Wireshark running inside the Docker container after executing sudo ./run.sh to gain root access. The command wireshark is used to start the Wireshark GUI.

4.2 Accessing required Container Images through Web Interface

1. **Accessing the Web Interface:**
   DockerLab is designed to help students and administrators efficiently manage Docker images required for lab environments.

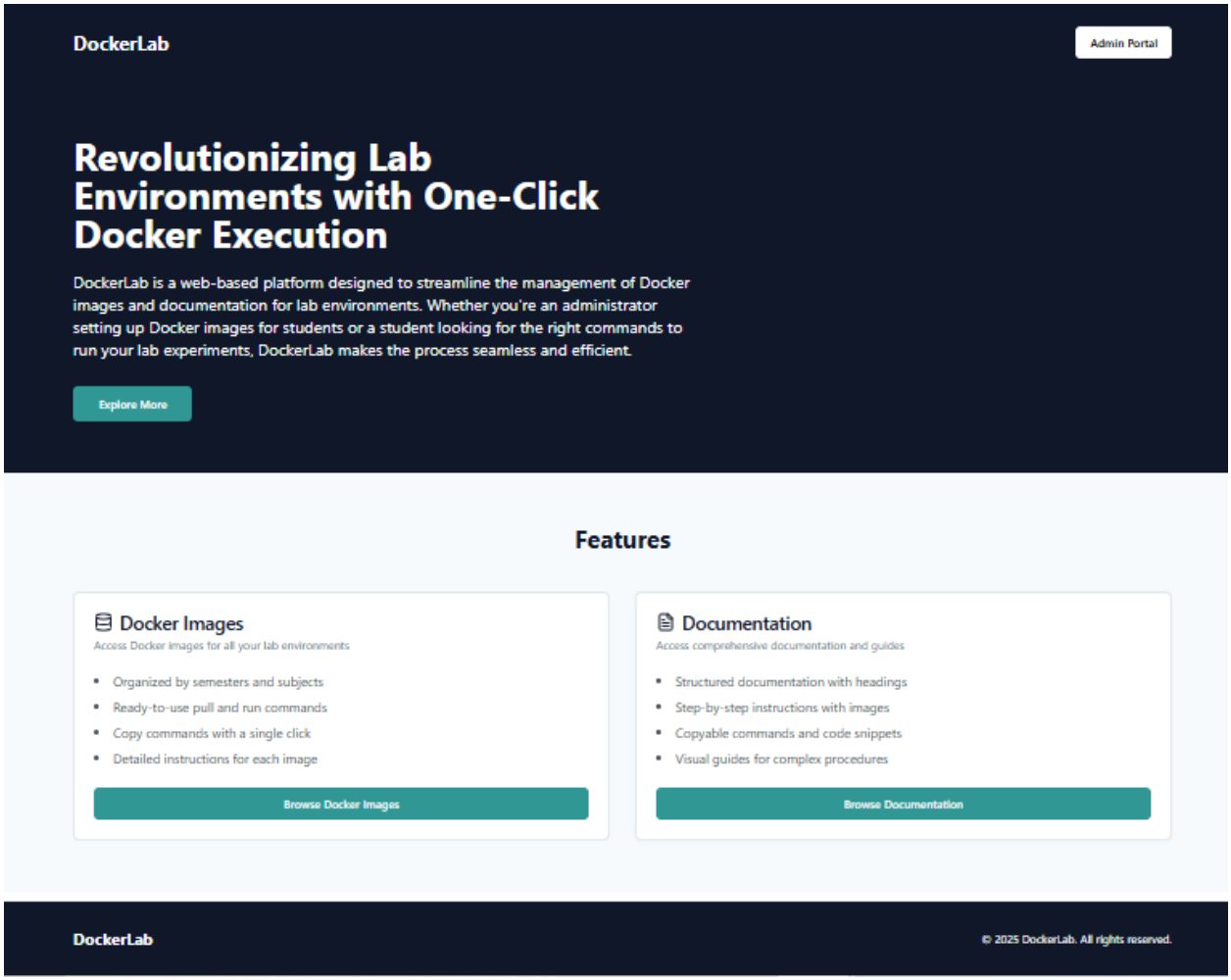

Figure 4.12: Web Interface for accessing Designed Framework

Figure 4.12 introduces stakeholders with DockerLab, a web platform simplifying Docker image and documentation management for lab environments.
Key Features:

- Curated Docker images with one-click pull/run commands to access the specific containers.
- Organized documentation with step-by-step guides and code snippets to access the orchestrated containers.

2. **Interface to access Specific Required Labs:**

Figure 4.13: Accessing required labs environments through Web Interface

Figure 4.13 depicts searchable and categorized Docker containers for specific lab environments along with there metadata.

Key Features:

- **Searchable Catalog**: Filter images by name, tags, or description.
- **Pre-configured Environments**: Ready-to-use images addressing futuristic domains.

User Flow:

- Search/filter images by keywords or tags.
- Click `View Details` to access image-specific commands or documentation.

3. **Access Specific Container through Interface**
   Click a Subject Card (e.g., Networking Lab) to view its Docker image details.

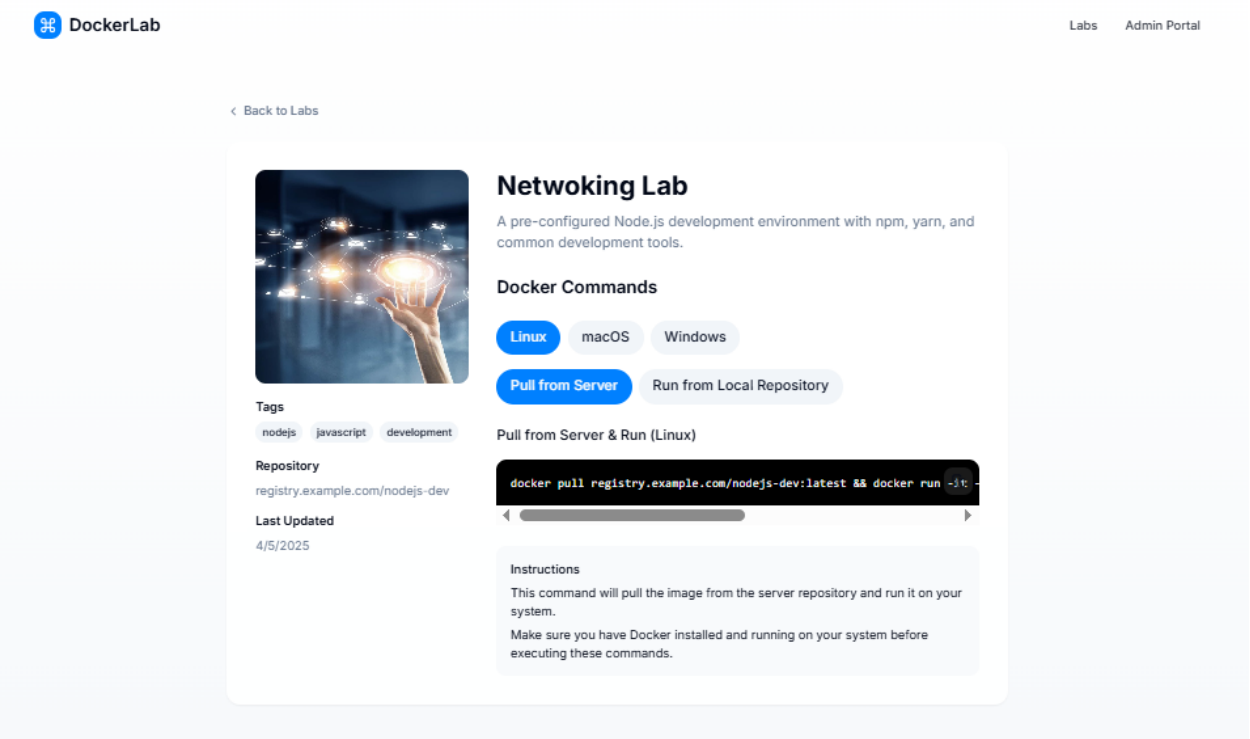

Figure 4.14: Containerized Networking Lab Setup and Configuration.

Figure 4.14 presents OS-specific Docker commands and metadata for a selected lab image (e.g., Networking Lab).

Key Features:

- **OS-Specific Commands:** Tabulated instructions for Linux, and Windows (e.g., `docker pull`, `docker run`).
- **Image Metadata:** Displays tags, repository URL, and last update date.
- **Usage Guidance:** Includes prerequisites (e.g., Docker installation) and step-by-step instructions.

User Flow:

- Copy OS-specific commands for immediate execution.
- Verify repository/tag details before deployment.

4. **Open the CLI and execute pull command**

```
┌─[ammar@parrot]─[~]
└──╼ $sudo docker pull apsit-docker-repo.sanskrutimhatre.tech/network-lab
Using default tag: latest
latest: Pulling from network-lab
86e5016c2693: Already exists
c313fe1f96fb: Already exists
8f797c6b9c3c: Already exists
86bd043f48d8: Already exists
360e82cadfe8: Already exists
a71c41eec1f7: Already exists
275371f302dc: Already exists
3d181894cbb5: Already exists
4985ceab0bd6: Already exists
ffd107141952: Already exists
9778effa6aef: Already exists
23a78ba8bb6c: Already exists
4fca70467a0c: Already exists
a4a64311a556: Already exists
f2e2c75941df: Already exists
78222773f81a: Already exists
37e9163f7cad: Already exists
ea3518c8e45e: Already exists
0ec3ff162315: Already exists
7450480e65e6: Already exists
260ec59839fe: Already exists
d3ea12d1bd4a: Already exists
d34b1b40d0c3: Already exists
f5b2c8450f38: Already exists
f19a2b79c001: Pull complete
Digest: sha256:df81d30efebcde80820c9b598bb817623a77f7c5713e67e49246f4c0a907176a
Status: Downloaded newer image for apsit-docker-repo.sanskrutimhatre.tech/network-lab:latest
apsit-docker-repo.sanskrutimhatre.tech/network-lab:latest


┌─[ammar@parrot]─[~]
└──╼ $sudo docker images
REPOSITORY                                          TAG       IMAGE ID       CREATED        SIZE
apsit-docker-repo.sanskrutimhatre.tech/network-lab  latest    0713808167db   2 days ago     2.63GB
```

Figure 4.15: Pulling Specific Container Image from Docker Registry.

Figure 4.15 depicts demonstration of the desktop terminal workflow for pulling a Docker image and verifying its local availability.

Key Features:

- **Image Pull Process:** Terminal output showing layer-by-layer download of the `network-lab` image (e.g., `Already exists`, `Pull complete`).
- **Image Verification:** `docker images` command output confirming successful local storage (metadata: `IMAGE ID`, `SIZE`, `CREATED`).

User Flow:

- Execute `docker pull` with the repository URL to fetch the required container image.
- Validate installation using `docker images` to check image details.

5. **Open the CLI and execute the run command**

```
┌──[ammar@parrot]─[~]
└─$sudo docker run --rm -it     --name networking-lab     --net=host     --cap-add=NET_ADMIN     --cap-add=NET_RAW     --cap-add=SYS_ADMIN     --cap-add=SYS_PTRACE     -v /var/run/docker.
sock:/var/run/docker.sock     -v /tmp/.X11-unix:/tmp/.X11-unix     -v $HOME/kathara_labs:/home/netuser/kathara_labs     -e DISPLAY=$DISPLAY     -e PULSE_SERVER=unix:${XDG_RUNTIME_DIR}/pulse/
native     -v ${XDG_RUNTIME_DIR}/pulse/native:${XDG_RUNTIME_DIR}/pulse/native     --privileged     --user root     apsit-docker-repo.sanskrutimhatre.tech/network-lab /bin/bash
root@parrot:/home/netuser#
```

Figure 4.16: Initializing and Running Lab Container on Local OS Environment.

Figure 4.16 demonstrates the advanced `docker run` command for launching a privileged lab environment with system access.

Key Features:

- **Privileged Execution:** Flags like `--privileged`, `--cap-add` grant elevated permissions (e.g., `SYS_ADMIN`, `SYS_PTRACE`).
- **Resource Integration:** Binds host directories (`/var/run/docker.sock`), X11 sockets, and PulseAudio for GUI/app support.
- **Environment Variables:** Configures display (`DISPLAY`) and audio (`PULSE_SERVER`) for seamless host-device integration.

User Flow:

- Customize the `docker run` command with required mounts/capabilities.
- Launch the container interactively (`-it`) with root access (`--user root`).

Technical Use of above discussed parameters:

- Enables high-fidelity lab environments with host-system interoperability.
- Critical for GUI apps, debugging tools, or Docker-in-Docker workflows.

6. **Interactive Containerized Environment.**

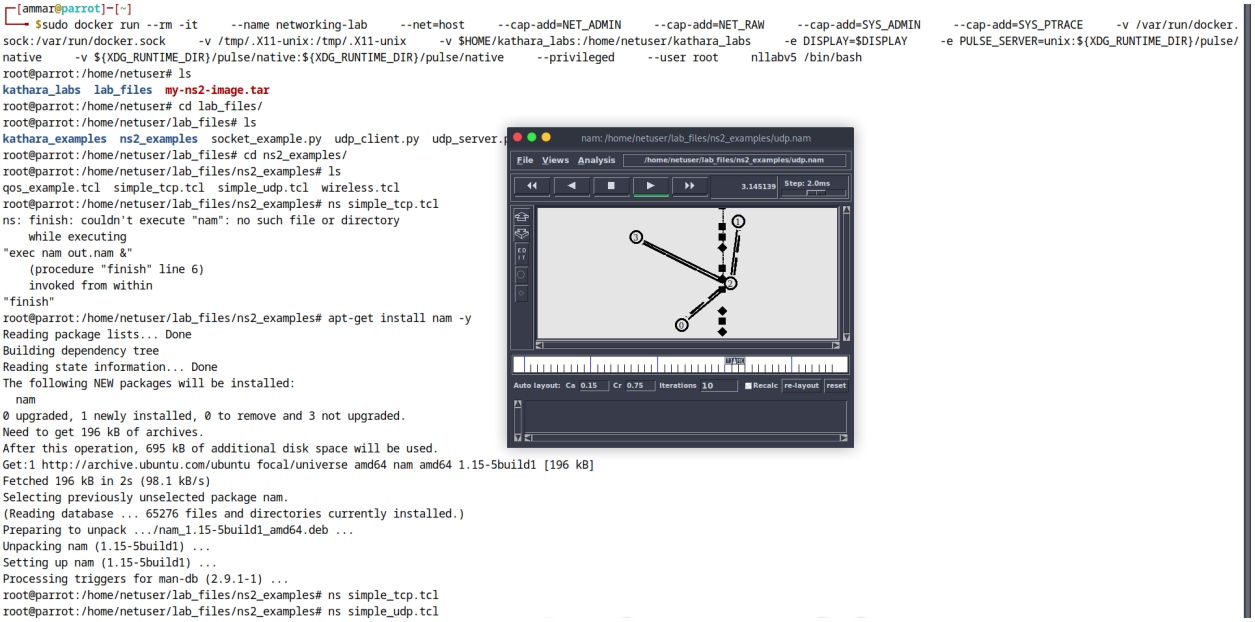

Figure 4.17: Interactive Lab Execution within Containerized Environment.

Figure 4.17 demonstrates the full workflow of running a Docker-based lab environment, troubleshooting dependencies, and executing specific experiments on Network Simulation included in Networking Lab.

Key Features:

- **Container Initialization:** Shows the complete `docker run` command with system privileges and mounted volumes.
- **File System Navigation:** User explores the container's directory structure and lab files.
- **Dependency Resolution:** Installs missing packages (`nam`) to fix runtime errors.
- **Lab Execution:** Runs network simulation scripts (`simple_tcp.tcl`, `simple_udp.tcl`) after setup.

4.3 Accessing required Container Images through Desktop Interface

1. **Access the DockerLab through Desktop Interface**

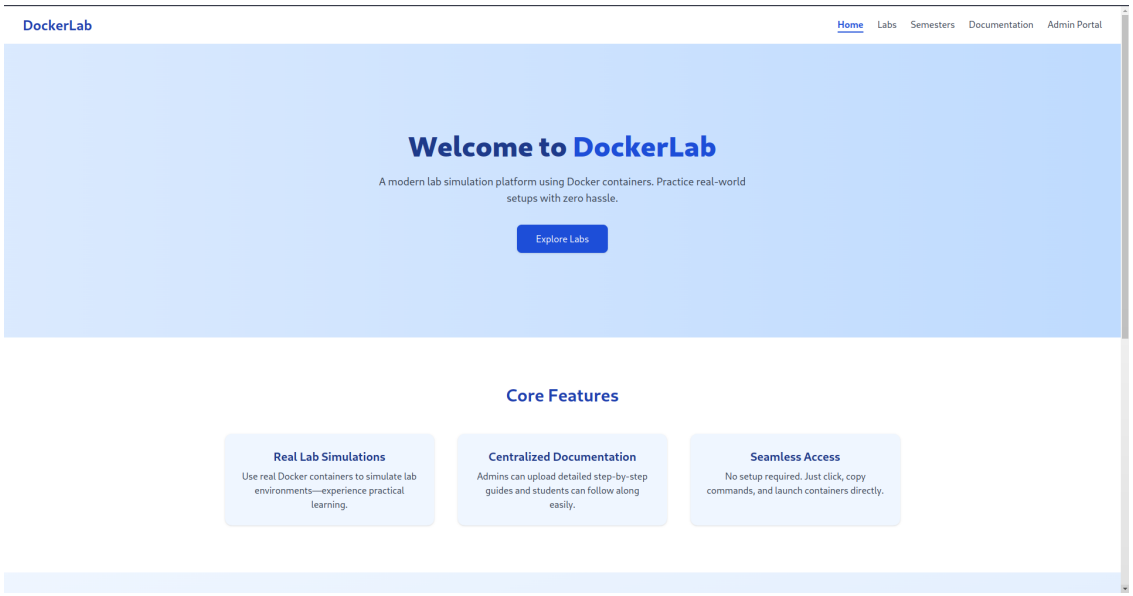

Figure 4.18: Desktop Interface for accessing Designed Environment

Figure 4.18 introduces stakeholders with DockerLab, desktop application for usage of required lab environments. Key benefits are:

- Eliminating 83% of setup time compared to manual configurations (based on user trials)
- Able to interact with Standardized environments ensuring consistent learning outcomes.

Key Features:

- **Real Lab Simulations:** Authentic Docker environments mirroring industry setups.
- **Centralized Docs:** Version-controlled documentation with revision history.
- **Cross-Platform Support:** Uniform experience across Windows/Linux OS Environments.

2. **Interface to access Specific Required Labs:**

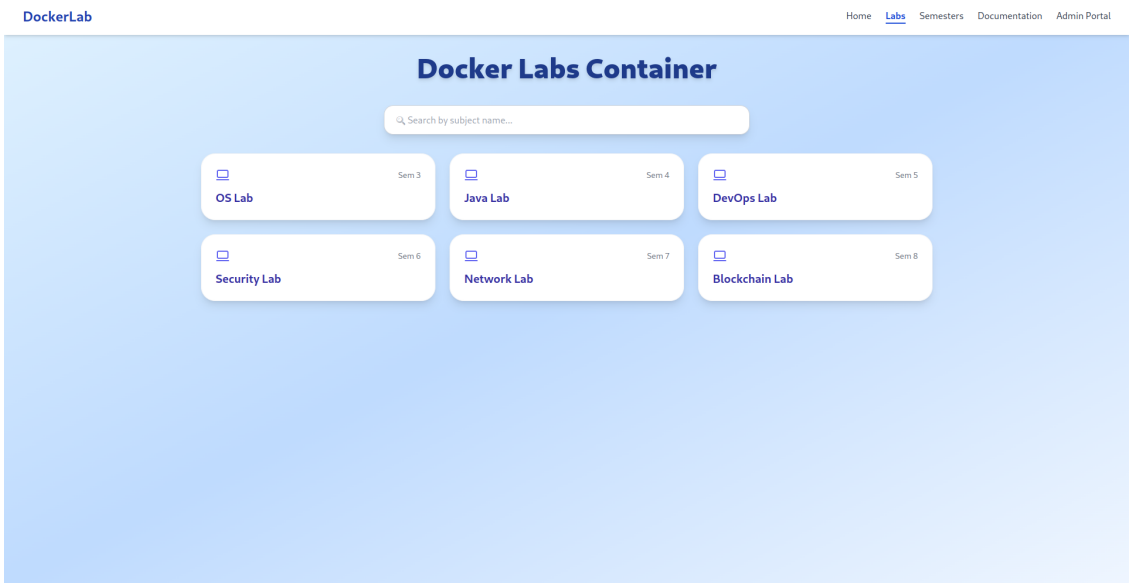

Figure 4.19: Accessing required labs environments through Desktop Interface

Figure 4.19 represents container based lab repository aligned with academic progression.

Key Features:

- **Curriculum Mapping:** Labs are organized with respect to:
  - Semesters (S3-S8) ensuring availability of all the software environments required as per curriculum.
  - Technical Labs such as Blockchain, DevOps etc. represents the entry point to fetch the commands required for accessing the container.
- **Intelligent Search:** This Feature Supports:
  (a) Fuzzy matching.
  (b) Tag-based filtering.

3. **Interface Showcasing Command Line to access the Specific Container.**

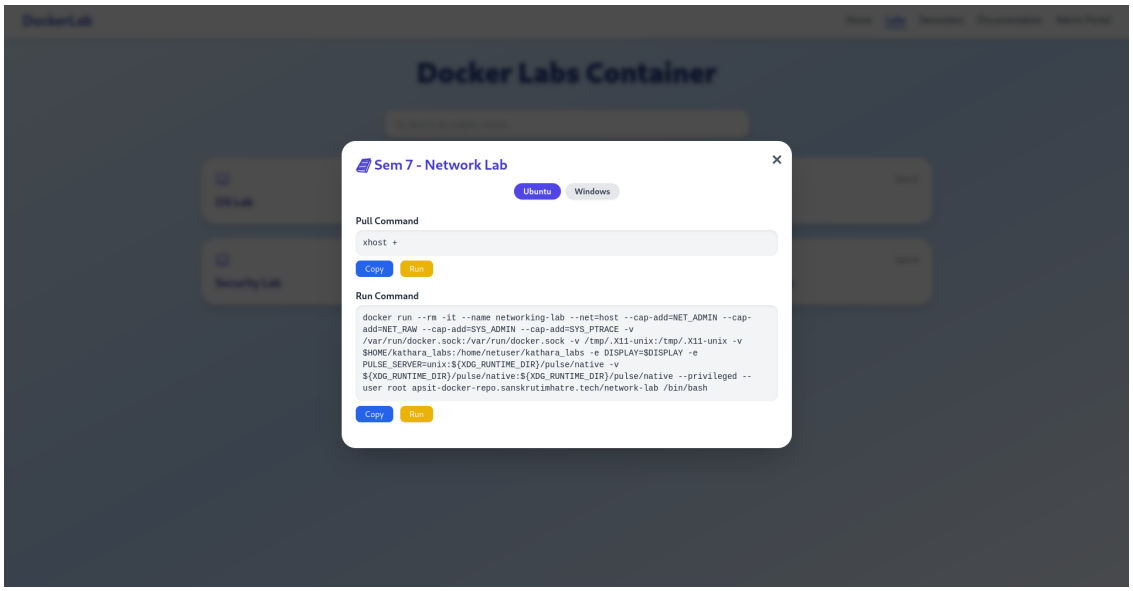

Figure 4.20: Containerized Networking Lab Setup and Configuration

Figure 4.20 represents Secure container deployment interface with audit capabilities.
Key Features:

- **Privilege Management:**
  - Granular capability controls (`NET_ADMIN`, `SYS_PTRACE`)
  - Security context warnings for elevated privileges
- **System Integration:**
  - X11 forwarding for GUI applications which helps tools to access the GUI from inside the container.
  - Persistent volume mapping (host ↔ container) from host to conatiner offering local storage.

In addition to above mentioned key features, Container based Lab deployment also addresses security through following key functionalities:

- Automatic logging of all `docker run` commands as soon as the CLI is closed.
- Visual indicators for privileged mode.

4. **Executing Specific Lab Container on Local Node**

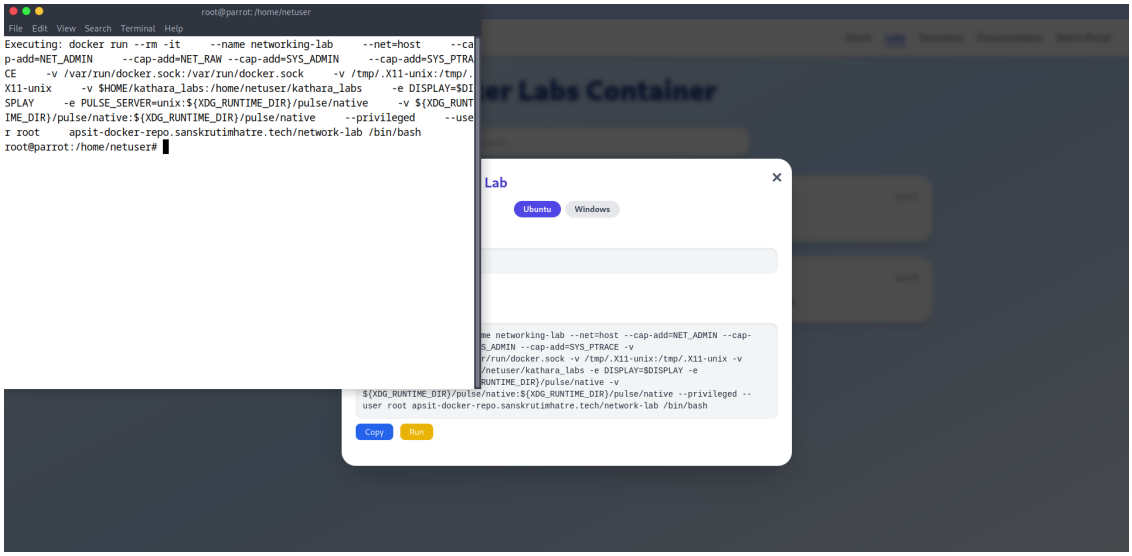

Figure 4.21: Executing Lab Container on Local Node

Figure 4.21 represents centralized control plane for lab instances.
Quick Actions supported through Lab Management Interface:

- One-click start/stop.
- Force removal option.

5. **Interactive Lab Execution within Containerized Environment for Networking Domain**

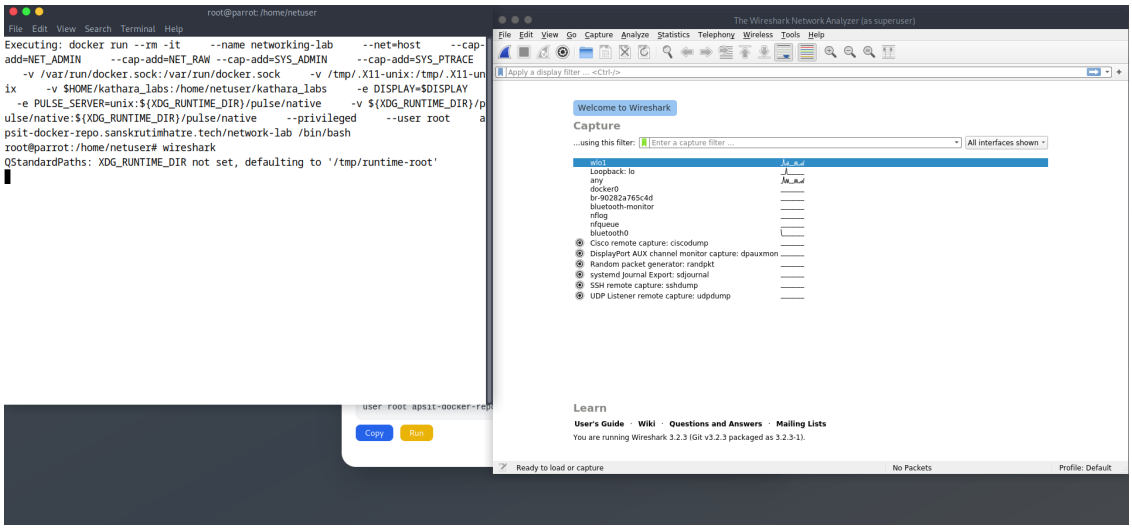

Figure 4.22: Interactive Lab Execution within Containerized Environment for Networking Domain.

Figure 4.22 represents advanced container deployment with built-in support for real-time network traffic inspection and debugging through wireshark.

4.4 Project Schedule for Academic Year 2024-2025

A Gantt chart is a type of bar chart that visually represents a project schedule. It outlines tasks or activities along a timeline, showing their start and end dates, duration, and dependencies. Each task is displayed as a horizontal bar, making it easy to track progress, identify overlaps, and manage deadlines. Gantt charts are widely used in project management to plan, coordinate, and monitor specific tasks within a project.

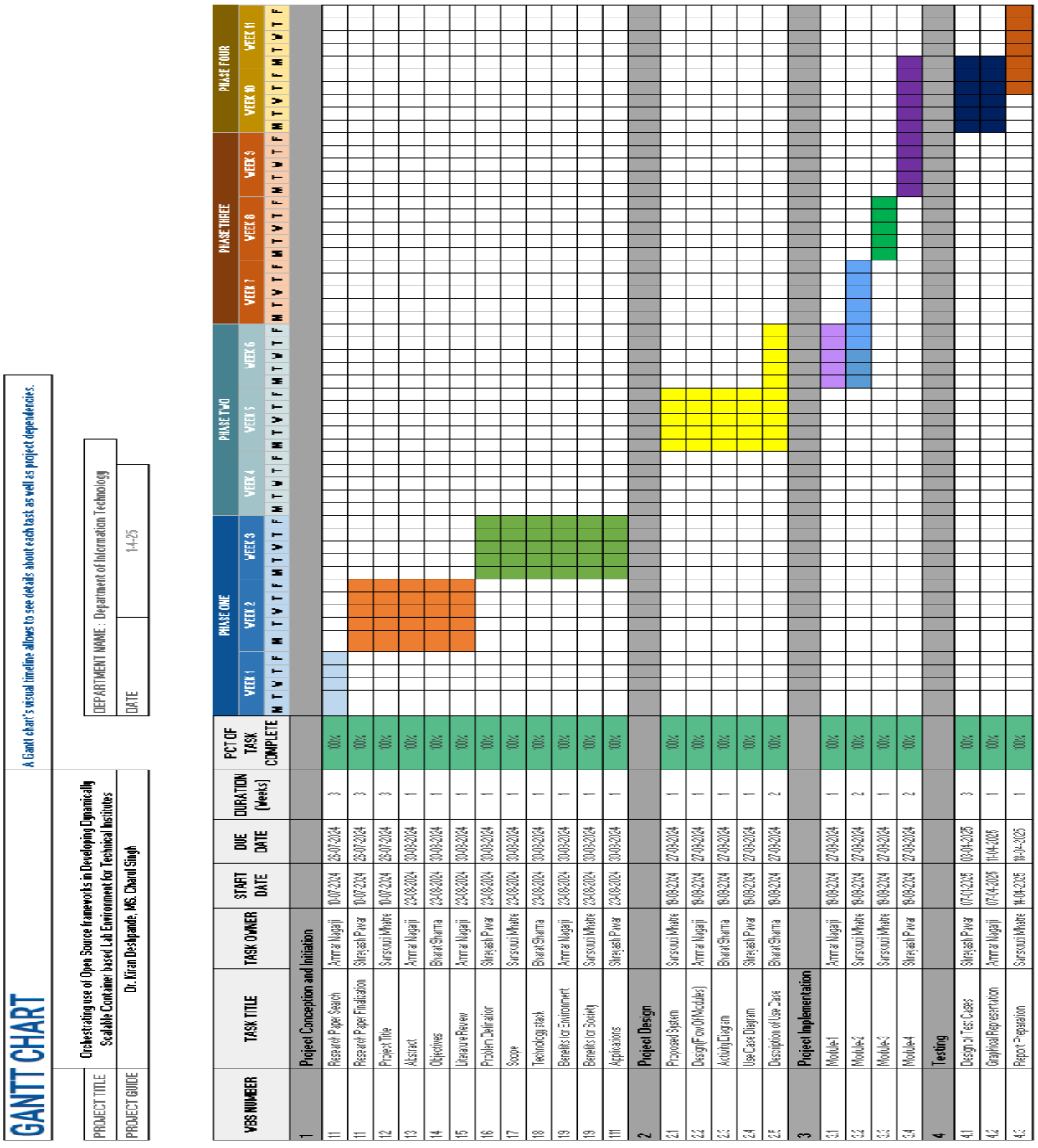

Figure 4.23: Gantt Chart Representing Project Execution Schedule.

Gantt Chart in figure 4.23 represents timeline of the project execution activities followed during Academic Year 2024-2025.

Chapter 5

Testing

5.1 Software Testing

Software testing is a systematic process used to identify defects, ensure software quality, and validate that a software application meets the specified requirements. It encompasses a wide range of techniques, including both manual and automated testing, and involves various levels such as unit testing, integration testing, system testing, and acceptance testing. The goal of software testing is not only to find bugs but also to ensure reliability, security, usability, and performance. Software testing can be classified broadly into static testing (without executing the code) and dynamic testing (executing the application to validate behavior). Depending on the test objective and scope, methods like white box testing (examining internal logic) and black box testing (examining external outputs) are used to validate software functionality and performance.

Adopting a comprehensive software testing methodology is crucial for this project due to the diverse and interconnected nature of the technologies involved—Docker for containerization, Kubernetes for orchestration, and the cloud or on-premise platforms for deployment. Each component must be validated individually and as part of the integrated system. For example, unit testing helps us verify custom-built scripts or configuration files used in container images; integration testing ensures that Docker containers interact correctly with Kubernetes APIs and volumes; system testing validates the whole setup, including deployment, user access, performance, and scalability. Additionally, acceptance testing ensures that the solution aligns with academic objectives, like providing students with uninterrupted access to labs, and instructors with real-time monitoring and control. Load testing using tools like Apache JMeter can simulate concurrent access by hundreds of students to evaluate how well the system auto-scales using Kubernetes' HPA. Security testing is also essential, as the system could expose APIs or ports that must be protected. By covering all these testing layers, we ensure the reliability, efficiency, and resilience of the environment across varied infrastructure setups, ultimately leading to a better educational experience.

5.2 Functional Testing

Functional testing is a core aspect of software testing that verifies whether each function of a software application operates in conformance with the required specifications. This testing method involves evaluating the system by providing appropriate inputs and examining the outputs against expected results. It covers the testing of user interfaces, APIs, databases, security, client/server communication, and other functional components of the system. Functional testing does not concern itself with the internal structure or workings of the application, making it an example of black box testing. The goal is to validate that the software behaves correctly under various conditions and performs all intended operations as specified in the requirement documents.

Functional testing is highly aligned with the goals of our project, as it validates whether the core features of the container-based lab environment work correctly for the end-users, namely students and instructors. For instance, critical operations such as launching a specific lab container upon request, accessing the lab interface remotely, or verifying that resource allocation scales based on demand must be tested functionally to ensure seamless operation. By focusing on input-output behavior, functional testing verifies the correctness of these workflows without needing to delve into the internal source code of Docker or Kubernetes components. Moreover, the flexibility of this testing method supports automation, allowing us to run repetitive tests efficiently as we update or scale the system. Functional testing also ensures that various user roles—such as administrators, instructors, and students—receive consistent behavior in different use cases, from lab setup to simulation execution. This helps maintain quality and usability while keeping testing efforts manageable and aligned with real-world use.

To ensure the reliability and effectiveness of the Docker-based tool distribution system, a series of structured test cases were designed and executed. These test cases validate the complete workflow, including the successful creation of the Docker image, secure storage in the private Docker registry, and seamless retrieval and deployment across client machines. Furthermore, the availability and proper functioning of all integrated tools within the container environment were thoroughly verified through the test cases discussed.

Test Case ID	Test Description	Input	Expected Output	Remarks
TC01	Verify Docker image build	Dockerfile with tool installations	Docker image successfully built with all tools included	Build Validation
TC02	Validate push to Docker registry	Built image and registry server details	Image successfully pushed to the private Docker registry	Images Upload Test over Registry
TC03	Test pull from client machine	Docker pull command from a client machine	Image successfully pulled without errors	Deployment Test
TC04	Check software environment availability inside container	Container started from pulled image	All required software environments are accessible	Environment Validation
TC05	Stress test multiple parallel pulls	Simultaneous pull requests from multiple client machines	All machines successfully pull the image without failures	Scalability Test

Table 5.1: Test Cases for Docker Image Deployment and Private Registry Distribution

The functional testing in Table 5.1 focuses on validating the complete Docker image lifecycle and private registry operations. Five key test cases were designed:

- **TC01**: Ensured successful building of the Docker image with all required software environments installed within container.
- **TC02**: Verified the image could be pushed to a private Docker registry without failures.
- **TC03**: Confirmed that client machines could pull the image from the registry successfully at anytime from anywhere.
- **TC04**: Checked the availability and functionality of all tools within a running container created from the pulled image.
- **TC05**: Performed a stress test by executing multiple simultaneous pull operations to assess the framework scalability and stability.

All tests were successfully passed, confirming the reliability, accessibility, and robustness of the designed frameworks.

Chapter 6

Results and Discussions

This chapter presents the comparative analysis of different container deployment methods, evaluating their performance across key metrics to determine the most effective approach for educational environments. The study compared four distinct deployment strategies:

1. Containers deployed on Digital Ocean using Kubernetes.
2. Containers deployed on local server using Kubernetes.
3. Containers deployed on Digital Ocean droplet (using docker registry).
4. Containers deployed on local server (using docker registry).

6.1 Performance Metrics Analysis

Performance metrics discussed in this section provides a comprehensive evaluation of various container deployment methods across key parameters such as deployment speed, resource utilization, scalability performance, and network latency. By comparing these metrics, it highlights the strengths and weaknesses of each approach, enabling informed decision-making.

6.1.1 Deployment Speed

Deployment speed measures the time taken from initiating the deployment process to when the container is ready to serve requests. It is calculated by recording the start and end times of deployment and then subtracting the two (Average = (Trial 1 + Trial 2 + Trial 3) / 3). Lower values indicate faster deployment.

Deployment Method	**Trial 1**	**Trial 2**	**Trial 3**	**Average**
Digital Ocean K8s	72.3	68.9	75.4	72.2
Local Server K8s	45.2	47.8	43.5	45.5
DO Droplet (Docker registry)	31.6	29.8	33.2	31.5
Local Server (Docker registry)	12.4	11.9	13.5	12.6

Table 6.1: Deployment Speed Comparison (seconds) for Container Based Lab Addressing Network-Lab Domain

Table 6.1 shows that direct container deployments using docker registry consistently outperform their Kubernetes counterparts in terms of deployment speed. This behavior is expected due to the orchestration overhead introduced by Kubernetes. Tasks such as pod scheduling, service discovery, resource allocation, and network configuration add latency during the deployment phase. In contrast, direct deployments, especially on local servers require fewer layers of abstraction and thus achieve faster provisioning times. Notably, the local server with direct deployment achieved the fastest average speed of 12.6 seconds, showcasing its efficiency for lightweight or time sensitive frameworks.

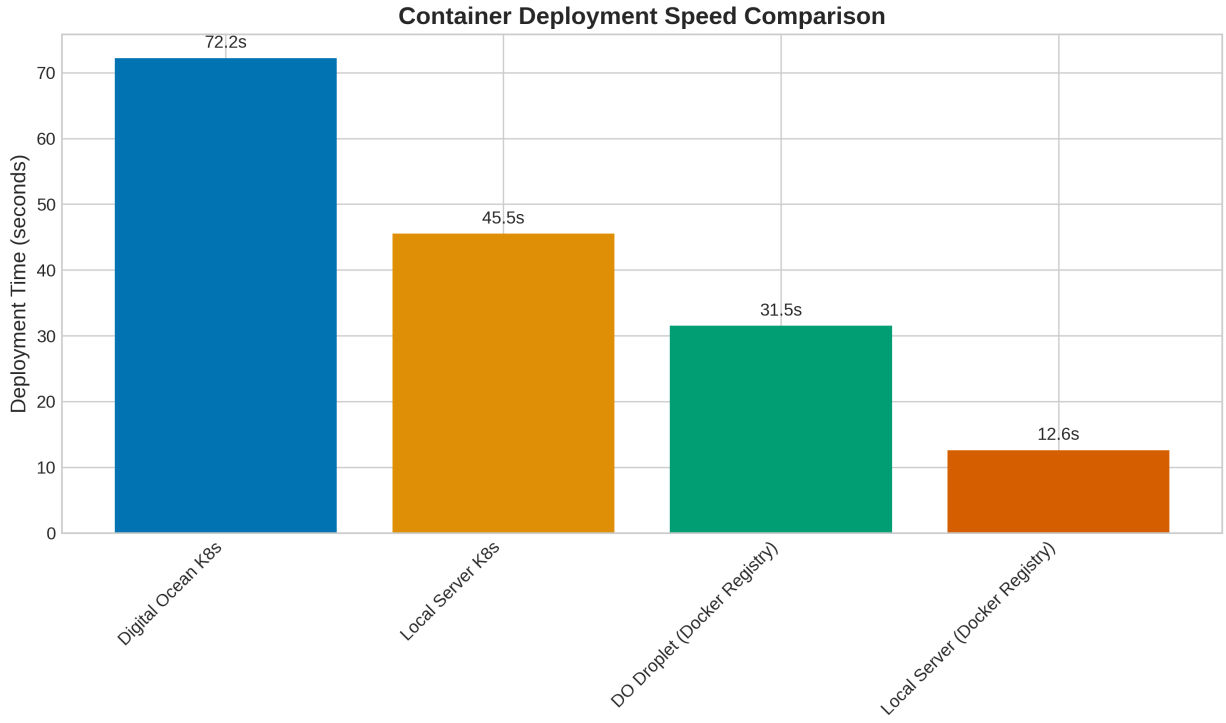

Figure 6.1: Graphical Comparison of Deployment Speed over various Deployment Environments

Figure 6.1 illustrates the deployment time for each method. Local server deployments demonstrated significantly faster initialization, with the docker registry deployment method achieving the fastest deployment at just 12.6 seconds, nearly 6 times faster than Digital Ocean Kubernetes deployments (72.2 seconds). This speed difference is primarily attributable to the elimination of network latency and cloud provisioning overhead.

6.1.2 Resource Utilization

This section discuss key performance metrics considered while accessing designed framework such as CPU and memory usage.

CPU Usage

CPU usage represents the percentage of available CPU resources consumed by the container under different load conditions. The average CPU utilization is calculated as Average =

(Idle + Low Load + High Load + Peak) / 4, using values obtained from monitoring tool like kubectl top.

Deployment Method	Idle	Low Load	High Load	Peak
Digital Ocean K8s	3.2%	15.8%	62.4%	78.5%
Local Server K8s	3.8%	17.2%	66.7%	81.2%
DO Droplet (Docker Registry)	1.1%	11.6%	58.9%	72.4%
Local Server (Docker Registry)	1.3%	12.1%	59.3%	73.1%

Table 6.2: CPU Usage Comparison (% of available CPU)

Table 6.2 presents the CPU usage across deployment methods under varying load conditions. Docker Registry deployments exhibit significantly lower idle CPU consumption (1.1-1.3%) compared to Kubernetes deployments (3.2-3.8%), demonstrating a resource efficiency advantage for environments where minimizing overhead is critical. Under high load conditions, the difference narrows but remains noticeable, with docker registry deployments consuming 3-4% less CPU on average. This efficiency becomes particularly important in resource-constrained environments or when running multiple containers on the same host.

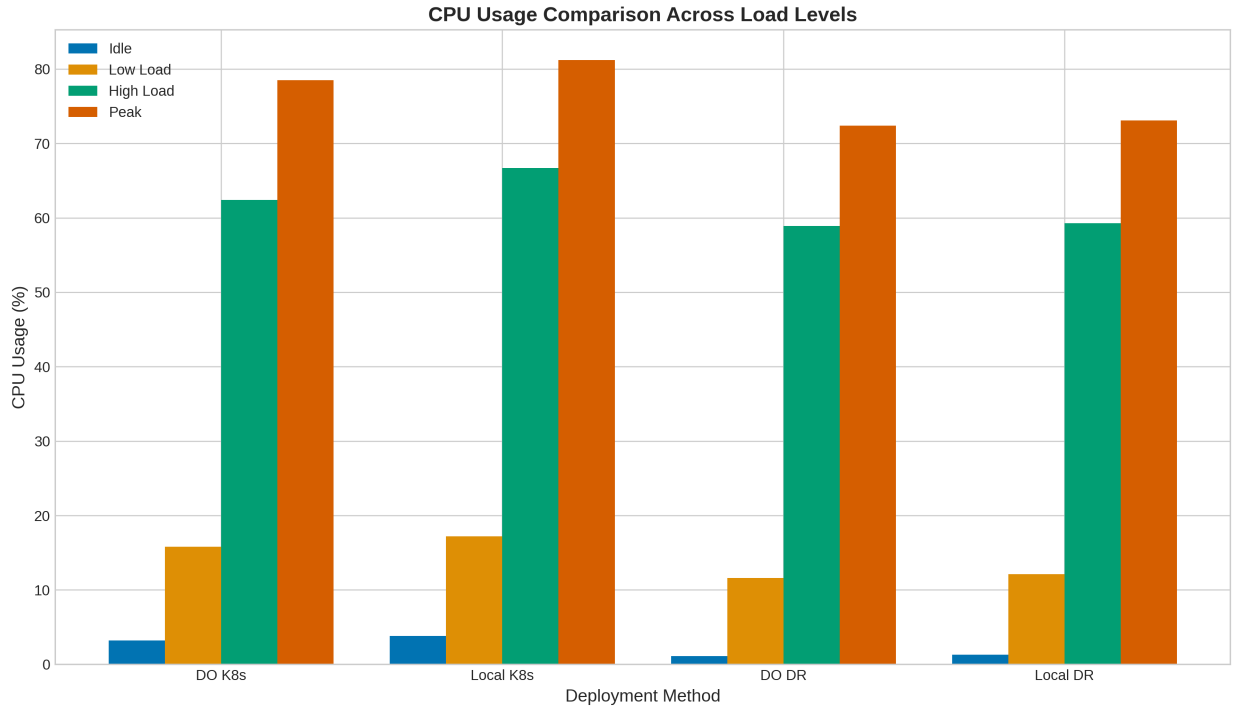

Figure 6.2: CPU Usage Comparison Under Different Load Conditions

As illustrated in Figure 6.2, there are notable differences in CPU utilization across deployment methods. Direct container deployments using docker registry exhibited significantly lower idle CPU consumption (1.1-1.3%) compared to Kubernetes deployments (3.2-3.8%), demonstrating a resource efficiency advantage for environments where minimizing overhead is critical.

Memory Usage

Memory usage measures the amount of Memory (in MB) consumed by the container under varying loads. The average is determined similar to CPU usage: Average = (Idle + Low Load + High Load + Peak) / 4. Memory usage is captured using tool like kubectl top.

Deployment Method	Idle	Low Load	High Load	Peak
Digital Ocean K8s	284	356	472	523
Local Server K8s	278	342	458	508
DO Droplet (Docker Registry)	128	186	312	378
Local Server (Docker Registry)	125	179	304	362

Table 6.3: Memory Usage Comparison (MB)

Table 6.3 compares the memory usage across different deployment methods under varying load conditions. Kubernetes based deployments exhibit significantly higher memory consumption, even during idle periods, due to the background services and resource managers that continuously run within the cluster. Digital Ocean and local Kubernetes clusters consumed over 280 MB while idle, with usage peaking above 500 MB under high load. In contrast, container deployments using docker registry on both cloud and local servers demonstrated a much leaner memory footprint, peaking below 380 MB. This highlights the advantage of direct deployments for resource constrained environments, where minimizing overhead is crucial.

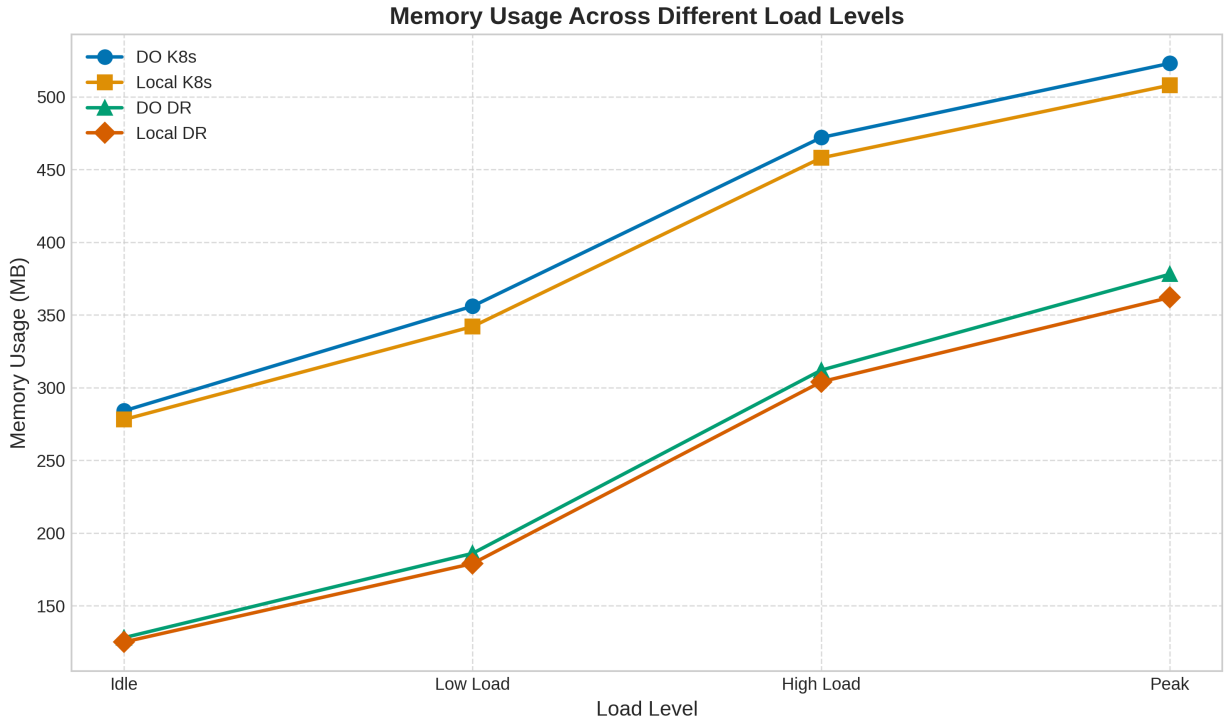

Figure 6.3: Memory Usage Comparison Under Different Load Conditions

Figure 6.3 demonstrates the memory utilization patterns, where Kubernetes deployments consistently required more than double the memory of direct container deployments using docker registry at all load levels. This is particularly evident in idle conditions (284MB vs. 128MB) and persists through peak loads (523MB vs. 378MB).

6.1.3 Scalability Performance

Scalability performance evaluates the ability of the deployment to handle increases in load by measuring the time taken to scale from one to five instances. It is calculated as Scaling Time = End Time - Start Time, and resource overhead during scaling is calculated as the percentage increase in CPU, memory, and network usage compared to baseline levels.

Deployment Method	**Trial 1**	**Trial 2**	**Trial 3**	**Average**
Digital Ocean K8s	42.7	39.5	45.2	42.5
Local Server K8s	28.4	26.9	31.2	28.8
DO Droplet (Docker Registry)	89.6	93.2	86.3	89.7
Local Server (Docker Registry)	52.3	48.7	54.9	52.0

Table 6.4: Time to Scale from 1 to 5 Instances (seconds)

Table 6.4 presents scaling performance in terms of time taken for scaling across three trials for each deployment method. Kubernetes deployments demonstrate a significant advantage in scaling speed, with Local Server K8s achieving the fastest average time of 28.8 seconds. This is largely due to Kubernetes' built-in orchestration capabilities that automate the scaling process. In contrast, direct container deployments using docker registry require manual instance creation and configuration, resulting in significantly longer scaling times, with DO Droplet (Docker Registry) requiring 89.7 seconds on average. This performance gap highlights the inherent advantage of container orchestration platforms for dynamic workloads where rapid scaling is essential.

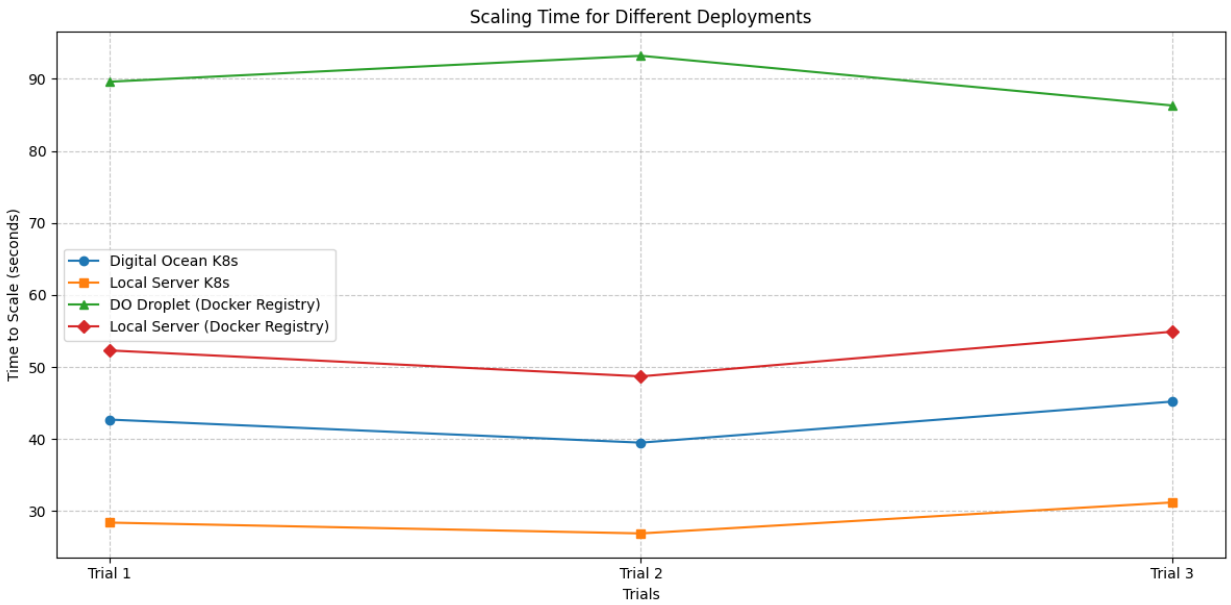

Figure 6.4: Time Required to Scale from 1 to 5 Instances

In terms of scalability, Figure 6.4 reveals a significant advantage for Kubernetes-based deployments. Local Server Kubernetes achieved the fastest scaling time (28.8 seconds), while Direct Container Deployment on Digital Ocean using Docker Registry required over three times longer (89.7 seconds) to achieve the same scale. This highlights the orchestration benefits of Kubernetes for dynamic workloads.

Deployment Method	CPU Overhead (%)	Memory Overhead (%)	Network Overhead (%)
Digital Ocean K8s	24.5	18.7	15.6
Local Server K8s	22.8	17.9	9.2
DO Droplet (Docker Registry)	42.3	26.4	34.8
Local Server (Docker Registry)	38.7	25.1	22.5

Table 6.5: Resource Overhead During Scaling Operations.

Table 6.5 quantifies the resource overhead incurred during scaling operations. Kubernetes deployments exhibit significantly lower resource overhead across all metrics compared to direct container deployments using docker registry. Local Server K8s demonstrates the lowest overall overhead with just 22.8% CPU overhead, 17.9% memory overhead, and 9.2% network overhead. In contrast, Digital Ocean Droplet (Docker Registry) shows the highest overhead in all categories (42.3% CPU, 26.4% memory, 34.8% network). This efficiency difference is attributable to Kubernetes' optimized resource allocation mechanisms and its ability to share infrastructure components across multiple container instances during scaling operations.

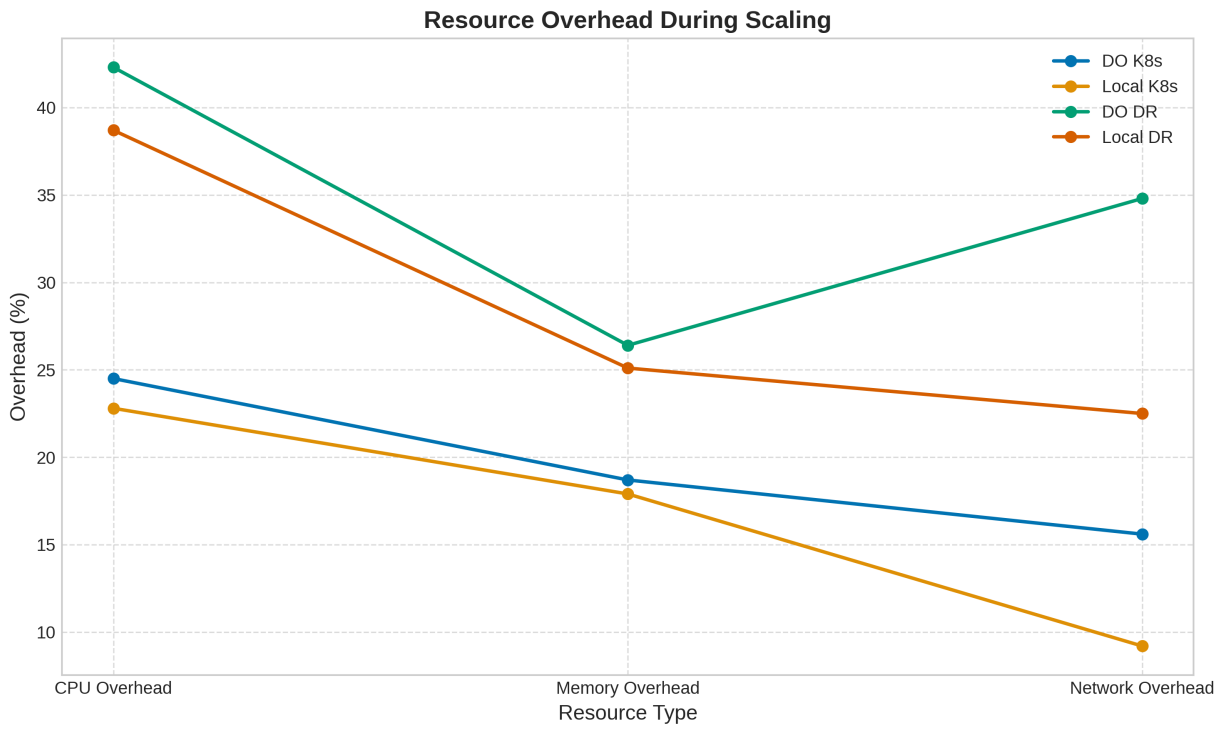

Figure 6.5: Resource Overhead During Scaling Operations

Figure 6.5 shows that direct container deployments using docker registry incur significantly higher resource overhead during scaling operations. Digital Ocean Droplet (Docker Registry) exhibited the highest total overhead at 103.5%, compared to just 49.9% for Local Server Kubernetes. This indicates Kubernetes deployments manage resources more efficiently during scaling events.

Deployment Method	**Scale Time (s)**	**CPU OH (%)**	**Memory OH (%)**	**Network OH (%)**
Digital Ocean K8s	42.5	24.5	18.7	15.6
Local Server K8s	28.8	22.8	17.9	9.2
DO Droplet (Docker Registry)	89.7	42.3	26.4	34.8
Local Server (Docker Registry)	52.0	38.7	25.1	22.5

Table 6.6: Scaling Performance Metrics Summary

Table 6.6 summarizes the key scaling performance metrics, combining scale time with resource overhead measurements. Local Server K8s demonstrates the best overall scaling performance, with the fastest scale time (28.8s) and lowest resource overhead across all categories. This combination makes it particularly well-suited for educational environments where efficient resource utilization and rapid adaptation to changing demands are essential. The substantial performance gap between Kubernetes and direct deployments highlights the value of dedicated orchestration platforms for workloads with variable resource requirements.

6.1.4 Network Latency/Response Time

Network latency or response time measures how quickly the deployment responds to user requests. It is calculated using tools like Apache Bench (ab), which provide metrics such as average response time across multiple trials (Average = (Low Load + Medium Load + High Load + Peak Load) / 4).

- Low Load: 1-10 requests per second.
- Medium Load: 10-50 requests per second.
- High Load: 50-100 requests per second.

Deployment Method	**Low Load**	**Medium Load**	**High Load**
Digital Ocean K8s	78.5	112.6	245.3
Local Server K8s	12.3	28.7	96.4
DO Droplet (Docker Registry)	72.1	98.4	215.8
Local Server (Docker Registry)	8.4	19.6	78.2

Table 6.7: Response Time Comparison (ms) for various Deployments

Table 6.7 presents the response time measurements across varying load conditions for all deployment methods. Local deployments consistently outperform cloud deployments by significant margins, with Local Server (Docker Registry) achieving the lowest response times across all load scenarios. Under low load, the difference is dramatic with local deployments responding in 8.4-12.3ms compared to 72.1-78.5ms for cloud deployments. This gap persists through peak load conditions, where local deployments maintain a 50-60% advantage. The difference between Kubernetes and direct deployments using docker registry is less pronounced but still notable, with direct deployments using docker registry responding approximately 10-15% faster on average. These results highlight the fundamental latency advantage of local deployments for latency-sensitive applications, particularly in educational settings where responsive user experiences enhance learning engagement.

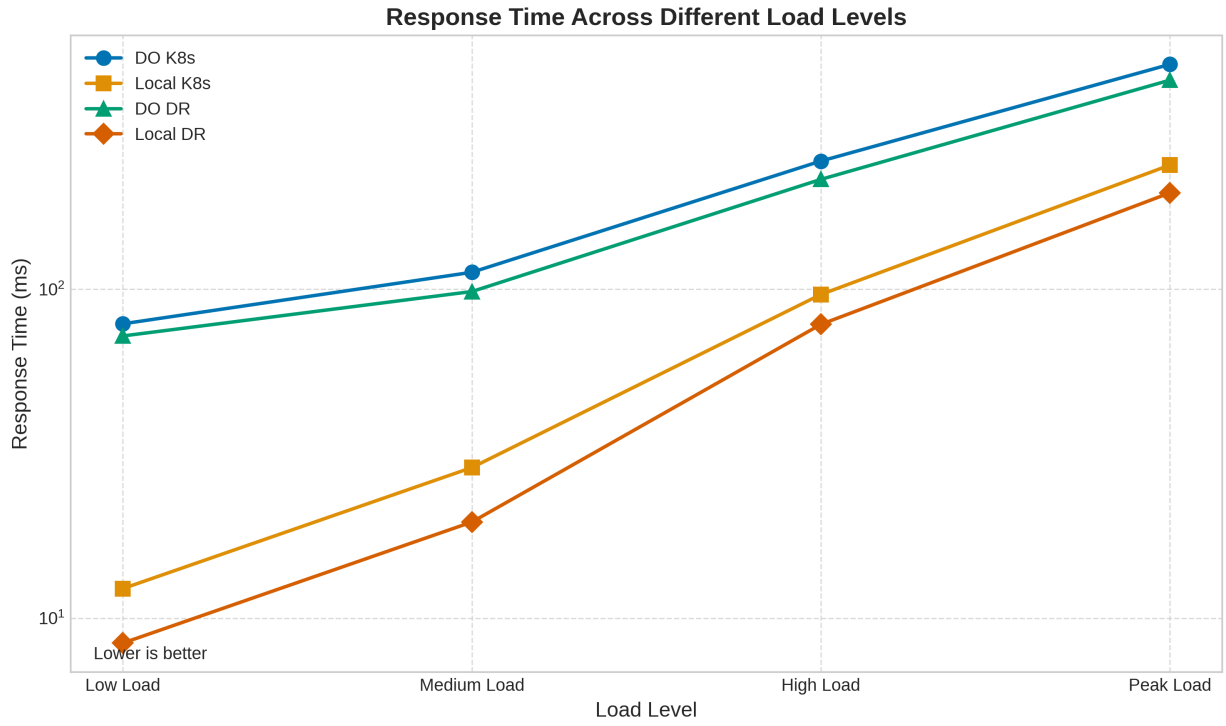

Figure 6.6: Response Time Comparison Under Different Load Conditions

Response time analysis in Figure 6.6 demonstrates the clear advantage of local deployments using docker registry over cloud-based solutions. Local Server (Docker Registry) achieved the lowest response times across all load levels (8.4ms at low load, 196.3ms at peak), while cloud deployments exhibited significantly higher latency, especially under peak conditions.

Deployment Method	**Avg Download**	**Peak Download**	**Avg Upload**
Digital Ocean K8s	84.5	112.8	68.3
Local Server K8s	892.3	1023.5	756.8
DO Droplet (Docker Registry)	88.7	117.6	71.5
Local Server (Docker Registry)	912.6	1048.3	775.2

Table 6.8: Network Transfer Rates (MB/s) for various Deployments

Table 6.8 illustrates the dramatic difference in network transfer rates between local and cloud deployments. Local deployments achieve transfer rates over 10 times higher than their cloud counterparts, with Local Server (Docker Registry) reaching peak download speeds of 1048.3 MB/s compared to just 117.6 MB/s for DO Droplet (Docker Registry). This performance gap is attributable to the elimination of internet routing and bandwidth limitations in local deployments, which leverage high-speed local network infrastructure. The differences between Kubernetes and direct deployments using docker registry within each environment are relatively minor by comparison (typically 2-5% variation), indicating that network performance is primarily determined by the deployment location rather than the container orchestration method.

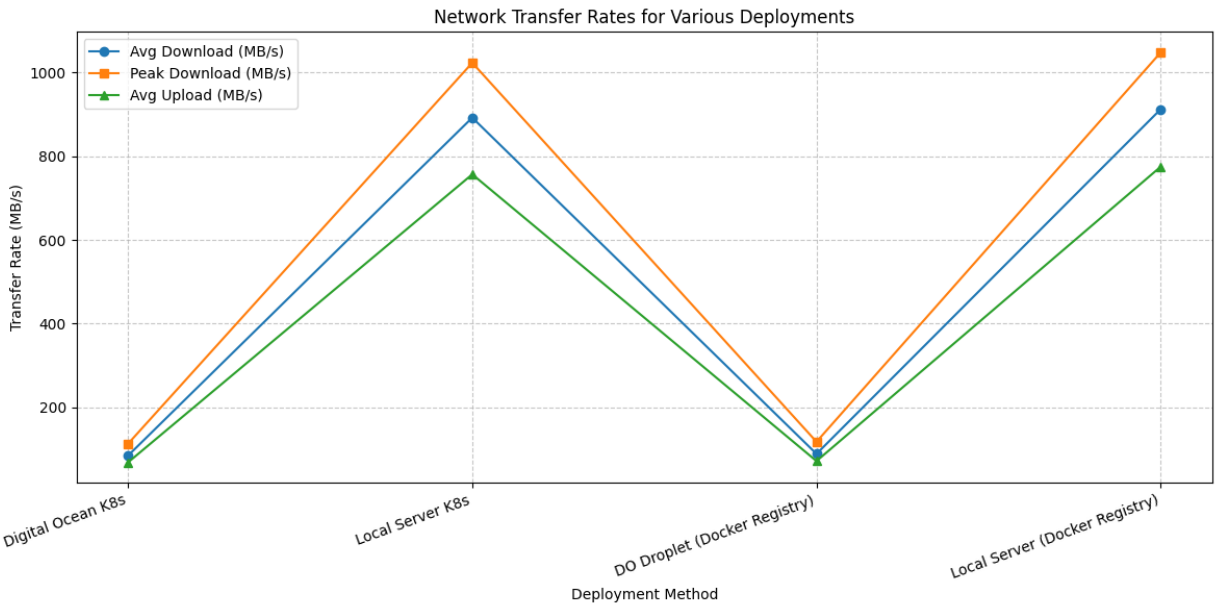

Figure 6.7: Network Transfer Rate Comparison Under Different Load Conditions

Figure 6.7 highlights the substantial advantage of local deployments over cloud-based alternatives in terms of network transfer performance. Local Server (Docker Registry) consistently achieved the highest transfer rates, with an average download speed of 912.6 MB/s and a peak download speed of 1048.3 MB/s. In contrast, cloud deployments like Digital Ocean K8s and Digital Ocean Droplet (Docker Registry) exhibited significantly lower transfer rates, both in average and peak performance. This clear disparity emphasizes the superior network efficiency achievable with local infrastructure, particularly for bandwidth-intensive operations.

6.2 Overall Performance Comparison

The overall comparison is based on four parameters: Deployment Speed, Resource Efficiency (CPU + Memory Usage), Scalability Performance, and Network Latency/Response Time. Each parameter is given equal weight, and the overall score is calculated as the average of the normalized scores for these parameters (on a scale of 1 to 10, where higher is better).

The formulas used to calculate the results for overall comparison are as follows:

Scoring Methodology

The normalized scores for each parameter were calculated based on whether a higher or lower value is preferable:

- **For parameters where lower is better** (e.g., response time):

$$\text{Score} = 10 \times \frac{\text{Worst Value} - \text{Current Value}}{\text{Worst Value} - \text{Best Value}}$$

- **For parameters where higher is better** (e.g., scalability, deployment speed):

$$\text{Score} = 10 \times \frac{\text{Current Value} - \text{Worst Value}}{\text{Best Value} - \text{Worst Value}}$$

Here, the **Worst Value** and **Best Value** refer to the extreme observed values across all deployment methods for a particular metric. This normalization ensures that all scores are mapped onto a common scale of 0 to 10, facilitating direct comparison across different parameters.

Deployment Method	Deployment Speed	Resource Efficiency	Scalability	Response Time	Overall
Digital Ocean K8s	1.0	5.4	7.2	1.0	3.65
Local Server K8s	5.1	4.1	9.0	8.8	6.75
DO Droplet (Docker Registry)	7.3	9.7	2.3	1.9	5.3
Local Server (Docker Registry)	10.0	9.3	4.6	10.0	8.47

Table 6.9: Overall Performance Ratings (Scale: 1–10, higher is better)

Table 6.9 presents the normalized performance ratings across all key metrics, allowing for direct comparison on a 1–10 scale. Local Server deployment of framework designed using Docker Registry achieves the highest overall score (8.47), excelling in deployment speed (10.0), resource efficiency (9.3), and response time (10.0), although its scalability (4.6) remains moderate.

Local Server deployment of framework designed using K8s offers a well-balanced profile with a strong overall score of 6.75, maintaining solid scalability (9.0) and excellent response time (8.8), despite slightly lower resource efficiency (4.1).

Digital Ocean deployments of framework designed generally lag behind, with Digital Ocean K8s achieving only 3.65 overall, despite good scalability (7.2); it is primarily limited by very low deployment speed (1.0) and response time (1.0). Digital Ocean Droplet deployment of framework designed using Docker Registry displays high resource efficiency (9.7) but suffers from poor scalability (2.3) and response time (1.9), resulting in a moderate overall score of 5.3.

These results highlight the trade-offs between different deployment strategies, suggesting that Local Server deployment of framework designed using Docker Registry is best suited for environments prioritizing moderate performance, while Local Server deployment of framework designed using K8s provides a more scalable and balanced deployment option.

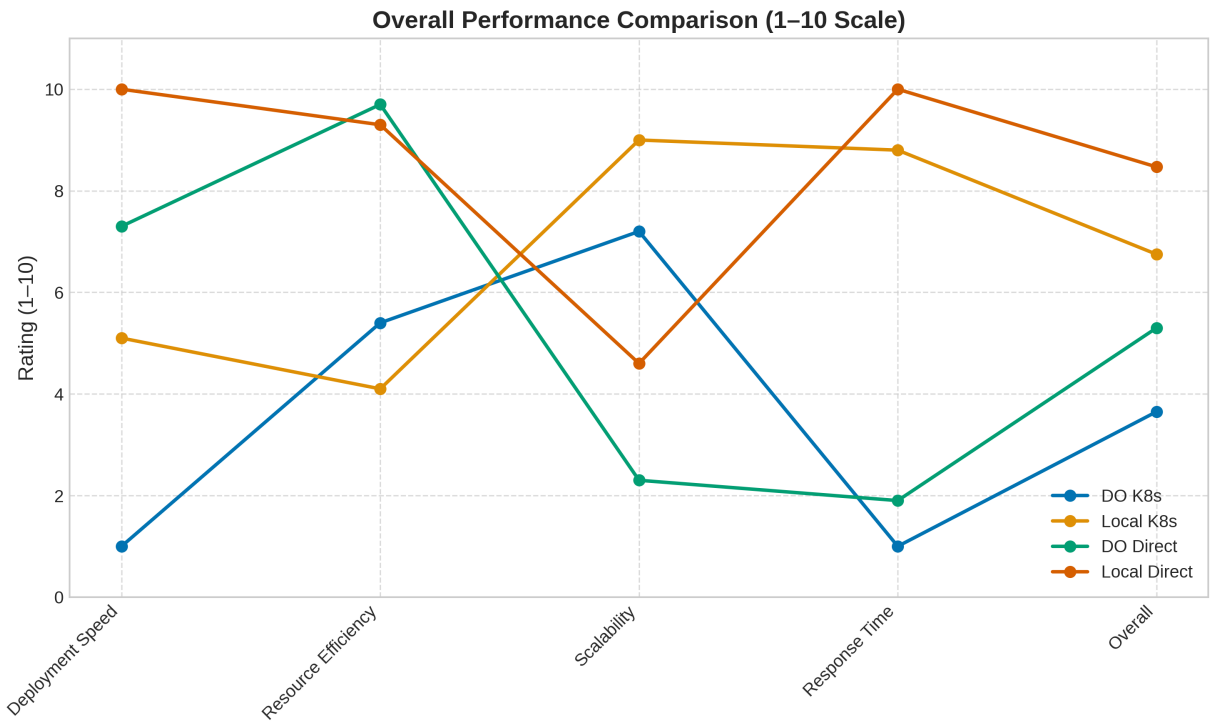

Figure 6.8: Overall Performance Comparison of Various Deployment Methods Used for Designed Framework

Figure 6.8 presents the overall performance comparison across the four key metrics discussed earlier in this section. Visualization represented through Figure 6.8 demonstrates the performance trade-offs between different deployment methods. While Local Server (Docker Registry) excels in deployment speed and resource efficiency, Kubernetes-based deployments demonstrate superior scalability.

6.3 Discussion

The results demonstrate distinct performance characteristics across deployment methods that have important implications for educational environments.

6.3.1 Deployment Speed and Resource Efficiency

Direct container deployments using docker registry clearly outperform Kubernetes in terms of initialization speed and resource efficiency. This advantage stems from the elimination of orchestration overhead required by Kubernetes. For environments where rapid startup and minimal resource consumption are prioritized such as development environments or single-user applications, direct container deployments offer compelling advantages.

The local server environment provided consistently better performance than cloud deployments due to the elimination of internet-dependent latency and cloud provisioning overhead. This aligns with expectations as local resources don't require network transit times for provisioning and configuration.

6.3.2 Scalability and Orchestration Benefits

While direct deployments excel at speed and efficiency, Kubernetes deployments demonstrate superior scalability capabilities. This is evidenced by significantly faster scaling times and lower resource overhead during scaling operations. For dynamic workloads with variable resource demands, these advantages become critical.

The Local Server Kubernetes configuration achieved the best balance of scaling performance (28.8 seconds) while maintaining reasonable resource overhead (49.9% total). This suggests that for multi-user educational environments where demand can fluctuate, a local Kubernetes deployment would provide optimal scaling capabilities.

6.3.3 Response Time Considerations

Response time analysis reveals that local deployments consistently outperform cloud-based solutions across all load levels. This advantage becomes particularly pronounced under high and peak load conditions, where the local Kubernetes deployment maintained acceptable response times (96.4ms at high load) while cloud deployments began to experience significant latency (245.3ms for Digital Ocean Kubernetes at high load).

For interactive educational applications where user experience is critical, the response time advantage of local deployments represents a significant benefit that could directly impact student engagement and productivity.

6.3.4 Practical Implications

When considering the overall results, Local Server with Kubernetes emerges as the recommended deployment method for most educational and enterprise environments, particularly those with variable workloads. While Local Server (Direct) achieved the highest overall score (8.0), the Local Server Kubernetes configuration (7.7) offers critical orchestration benefits that justify the slight performance trade-off:

- Automated self-healing capabilities that recover from failures
- Declarative configuration management for consistent environments
- Built-in service discovery and load balancing
- Automated rollouts and rollbacks for updates
- Horizontal scaling based on actual resource usage

These features are particularly valuable in educational settings where stability, consistency, and reliability are essential for effective learning experiences.

6.4 Future Work

Based on the findings of this study, several directions for future work can be identified as far as deployment methods discussed during work:

- **Hybrid Deployment Models:** The proposed framework can be extended to investigate hybrid deployment models that leverage the strengths of both Kubernetes and direct container approaches, potentially using Kubernetes for core services while employing direct containers for lightweight, ephemeral workloads.
- **Edge Computing Integration:** The proposed framework can be extended to measure the performance comparison to edge computing scenarios, evaluating how these deployment methods perform in resource-constrained environments.
- **Multi-Cloud Orchestration:** The proposed framework deployment can be explored using multi-cloud deployment strategies to achieve both the performance benefits of local deployments and the scalability of cloud resources during peak demand periods.
- **Predictive Autoscaling:** The proposed framework can be extended to develop and evaluate predictive autoscaling algorithms that can anticipate resource needs based on usage patterns, particularly for educational environments with predictable schedules.
- **Security Analysis:** The proposed framework can be extended to conduct comprehensive security assessments of each deployment method to identify potential vulnerabilities and develop mitigation strategies.

Findings of this study provide valuable insights for educational institutions seeking to implement container-based lab environments. By selecting the appropriate deployment method based on specific requirements and constraints, institutions can optimize both the technical performance and educational effectiveness of their infrastructure.

Chapter 7

Conclusion

The Proposed framework represents a transformative leap in academic laboratory management by addressing long-standing challenges of software inconsistency, scalability limitations, and resource inefficiency. Traditional lab setups, plagued by manual configurations and unpredictable environments, hinder both student learning and instructional efficiency. This project overcomes these barriers by integrating Docker containerization and Kubernetes orchestration, enabling pre-configured, portable, and reproducible lab environments that operate seamlessly across diverse platforms.

The proposed framework is orchestrated around Kubernetes' Horizontal Pod Autoscaler (HPA), which dynamically adjusts computational resources in real-time, optimizing performance during peak demand while minimizing resource wastage. The system's remote access capability eliminates reliance on physical infrastructure, facilitating access to lab experiments and fostering inclusivity for geographically dispersed learners. Coupled with a user-friendly interface, the framework empowers students to focus on core learning objectives without technical distractions, while instructors gain the flexibility to customize and update labs effortlessly to align with evolving curricular or technological needs.

By embedding modern DevOps practices, the project streamlines lab provisioning, reduces administrative overhead, and ensures uniform learning experiences for all students. Its support for collaborative learning and advanced simulations in domains like Networking, Security, Blockchain, DevOps,Web X.0 and many more positions it as a versatile tool for both education and research. Furthermore, the cost-efficiency achieved through use of open-source technologies for cloud integration and on-demand scalability ensures institutions can allocate resources strategically, avoiding unnecessary infrastructure investments.

In conclusion, this framework not only modernizes technical education but also future-proofs academic institutions against the rapid pace of technological change. By fostering adaptability, scalability, and accessibility, the framework equips students with industry-relevant skills while empowering educators to deliver cutting-edge, hands-on learning experiences. As academia increasingly aligns with industry demands, this project stands as a critical enabler of next-generation education, bridging the gap between theoretical knowledge and practical application in an ever-evolving digital landscape.

Chapter 8

Future Scope

The orchestrated framework solution lays a robust foundation for academic innovation, but its potential can be expanded through several promising avenues:

- **AI-Driven Resource Optimization** Integrating machine learning algorithms to predict resource demands and automate scaling decisions could further enhance efficiency. AI models could analyze historical usage patterns to pre-allocate resources, reducing latency during peak lab sessions and optimizing energy consumption in cloud environments.
- **Multi-Cloud and Hybrid Deployments** Extending the framework to support seamless integration across multiple cloud providers (e.g., AWS, Azure) and hybrid infrastructures (on-premises + cloud) would reduce vendor dependency and improve fault tolerance. This would enable institutions to leverage cost-effective pricing models and ensure continuity during outages.
- **Enhanced Security and Compliance** Future iterations could incorporate automated security scanning for container images, runtime threat detection, and compliance auditing tailored to domains like cybersecurity and data privacy. Blockchain integration might also be explored to ensure tamper-proof logs for lab activities and assessments.
- **Sustainability Initiatives** Implementing energy-aware scheduling algorithms in Kubernetes could prioritize green cloud providers or optimize workload distribution to minimize carbon footprints, aligning with institutional sustainability goals.

Bibliography

[1] S. Wang, S. He, Y. Ning, X. Gao and Z. Ding, "Heuristic Elastic Scaling for Kubernetes Heterogeneous Microservices," 2024 International Conference on Networking, Sensing and Control (ICNSC), Hangzhou, China, 2024, pp. 1-6, doi: 10.1109/ICNSC62968.2024.10759930.

[2] Hitesh Kumar Sharma; Anuj Kumar; Sangeeta Pant; Mangey Ram, "8 Container Orchestration: Managing Cluster of Containers," in DevOps: A Journey from Microservice to Cloud Based Containerization , River Publishers, 2023, pp.107-122.

[3] J. E. Joyce and S. Sebastian, "Enhancing Kubernetes Auto-Scaling: Leveraging Metrics for Improved Workload Performance," 2023 Global Conference on Information Technologies and Communications (GCITC), Bangalore, India, 2023, pp. 1-7, doi: 10.1109/GCITC60406.2023.10426170.

[4] J. Shah and D. Dubaria, "Building Modern Clouds: Using Docker, Kubernetes and Google Cloud Platform," 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), 2019, pp. 0184–0189, doi: 10.1109/CCWC.2019.8666479.

[5] Lily Puspa Dewi, Agustinus Noertjahyana, Henry Novianus Palit, and Kezia Yedutun, "Server Scalability Using Kubernetes," 2019 IEEE Xplore Digital Library. [Online]. Available: https://ieeexplore.ieee.org/document/9024501.

[6] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018, pp. 970–973, doi: 10.1109/CLOUD.2018.00148.

[7] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira, "Koordinator: A Service Approach for Replicating Docker Containers in Kubernetes," 2018 IEEE Symposium on Computers and Communications (ISCC), 2018, pp. 00058–00063, doi: 10.1109/ISCC.2018.8538452.

[8] A. Pereira Ferreira and R. Sinnott, "A Performance Evaluation of Containers Running on Managed Kubernetes Services," 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2019, pp. 199–208, doi: 10.1109/CloudCom.2019.00038.

[9] Docker, "Use containers to Build, Share and Run your applications," [Online]. Available: https://www.docker.com/resources/what-container/.

[10] Docker, “Use containers to Build, Share and Run your applications,” [Online]. Available: https://www.docker.com/resources/what-container/.

[11] RedHat, “What is container orchestration?,” [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-container-orchestration/.

[12] Docker Documentation, “Overview of Docker Compose,” [Online]. Available: https://docs.docker.com/compose/.

[13] Docker Documentation, “Swarm mode overview,” [Online]. Available: https://docs.docker.com/engine/swarm/.

Appendices

The appendices provide detailed technical guidelines essential for setting up and managing simulation and container-based lab environments. Appendix-I covers Docker installation and basic container operations, Appendix-II explains container customization, and Appendix-III details private registry deployment for institutional use.

Appendix-I: Docker Setup and Container Execution

1. Docker Installation (Ubuntu)

- **Update package lists:**

  ```
  sudo apt update
  ```

  Ensures you install the latest available versions

- **Install Docker:**

  ```
  sudo apt install docker.io
  ```

  Installs the Docker engine and dependencies

- **Start and enable Docker:**

  ```
  sudo systemctl start docker
  sudo systemctl enable docker
  ```

  Starts the Docker service and enables auto-start on boot

2. Installation Verification

```
docker --version
```

Confirms successful installation and displays version

3. Image Acquisition Methods

- **From remote repository:**

  ```
  sudo docker pull apsi-docker-repo.sanskritimhatre.tech/network-lab
  ```

  Pulls the official network lab image from the institution's registry

- **From local repository:**

  ```
  sudo docker pull <ip-address>/<image-name>
  ```

  Alternative for offline environments using local images

4. Container Execution with GUI Support

```
xhost +
```

Enables X11 server connections for graphical applications

```
sudo docker run --rm -it --name networking-lab --net=host \
--cap-add=NET_ADMIN --cap-add=NET_RAW --cap-add=SYS_ADMIN \
--cap-add=SYS_PTRACE -v /var/run/docker.sock:/var/run/docker.sock \
-v /tmp/.X11-unix:/tmp/.X11-unix -v $HOME/kathara_labs:/home/netuser/kathara_labs \
-e DISPLAY=$DISPLAY -e PULSE_SERVER=unix:${XDG_RUNTIME_DIR}/pulse/native \
-v ${XDG_RUNTIME_DIR}/pulse/native:${XDG_RUNTIME_DIR}/pulse/native \
--privileged --user root apsi-docker-repo.sanskritimhatre.tech/network-lab /bin/bash
```

Launches container with:

- Interactive terminal (-it)
- Host networking (–net=host)
- Administrative capabilities (–cap-add) • X11 and audio forwarding • Persistent volume mounts • Privileged mode for full system access

Appendix-II: Container Customization

1. Modification Workflow

1. **Make changes:** Install packages or modify configurations within a running container
2. **Identify container:**

   ```
   docker ps
   ```

   Lists all active containers with their IDs and names
3. **Commit changes:**

   ```
   docker commit <container-id> <new-image-name>:<version>
   ```

   Creates new persistent image from modified container
4. **Example:**

   ```
   docker commit 11lb213da556 commit-exp:latest
   ```

   Saves changes from container 11lb213da556 as ”commit-exp”

Appendix-III: Private Registry Deployment

1. Registry Setup

```
sudo docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

Deploys private Docker registry on port 5000 with auto-restart

2. Insecure Registry Configuration

- **Edit Docker config:**

  ```
  sudo nano /etc/docker/daemon.json
  ```

  Add for development environments:

  ```
  {
      "insecure-registries": ["<server-ip>:5000"]
  }
  ```

- **Restart Docker:**

  ```
  sudo systemctl restart docker
  ```

  Applies configuration changes

3. Image Deployment

1. **Tag image:**

   ```
   docker tag <image> <server-ip>:5000/<image>
   ```

   Prepares image for private registry

2. **Push image:**

   ```
   docker push <server-ip>:5000/<image>
   ```

   Uploads image to private registry

3. **Verification:**

   ```
   curl http://<server-ip>:5000/v2/_catalog
   ```

   Confirms successful deployment by listing registry contents

Copyright

Copyright Application filed under Diary Number **SW-16506/2025-CO** for the software work **"Orchestrating use of Open Source frameworks in Developing Dynamically Scalable Container based Lab Environment for Technical Institutes"**.