Design and Implementation of a Secure Virtualized IT Platform with Containerized Applications

Course: IT Platforms

Submitted by:
Group 10
Mehmet Emin Cicek #96769995
Puthyka Thearith #25996283
Jane Booysen #53898092
Zein Al-Hashimi #41925384
Roxana Ramazanova #72750731

1. Introduction

This project demonstrates the design, implementation, and verification of a complete IT platform that integrates network security, virtualization, and container-based application deployment. The objective was to create a realistic enterprise environment where multiple infrastructure layers work together to support secure and scalable services.

The platform follows a layered architecture model, with each layer building upon the capabilities of the previous one. At the foundation, a secure network infrastructure provides controlled connectivity, traffic isolation, and access control through firewall configuration. The second layer introduces virtualization technology, which enables the hosting of multiple operating systems on a single physical machine, thereby simulating a multi-server environment. The uppermost layer implements modern application deployment using containerization, demonstrating how applications can be packaged, distributed, and executed in isolated, reproducible environments.

This layered approach reflects modern IT infrastructure design, where security, resource virtualization, and application deployment are treated as distinct but interconnected concerns. Each component was configured according to industry practices, tested to verify correct operation, and integrated with other platform components to ensure end-to-end functionality. The implementation demonstrates how traditional network security mechanisms, operating system virtualization, and modern containerization technologies can be combined to create a functional IT platform suitable for hosting enterprise applications.

2. Network Design and Firewall Configuration

2.1 Network Architecture and Security Objectives

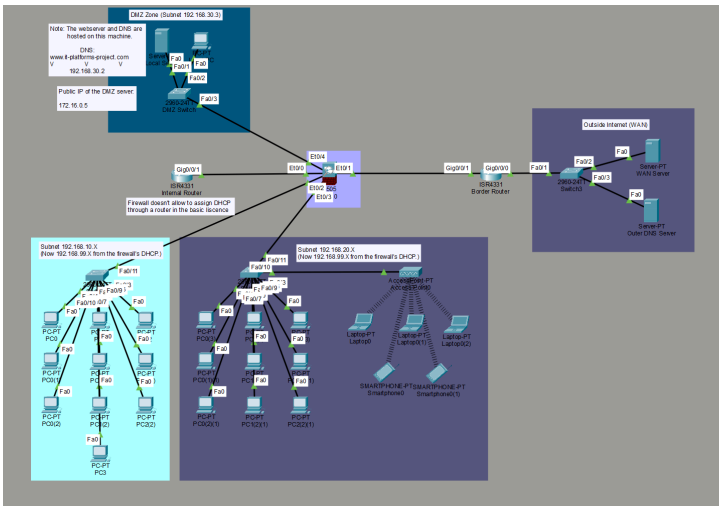


Figure 1: *Overall network topology with ASA firewall, DMZ, Inside, and Outside networks*

This part's objective was to configure a secure network using a Cisco ASA 5505 Firewall. The objective was to establish a protected internal LAN with automatic IP assignment (DHCP), secure Internet access via NAT, and a Demilitarized Zone (DMZ) to host a web server which would be accessible publicly

2.2 IP Addressing and Network Segmentation

A structured IP addressing scheme was implemented to clearly delineate security boundaries. Three distinct IP ranges were allocated: one for the internal network where client devices reside, one for the DMZ hosting public-facing services, and one for the outside interface representing external connectivity. This segmentation ensures that traffic between zones must traverse the firewall, where security policies can be enforced.

The ASA firewall was configured with three VLAN interfaces, each assigned to a different security zone. The Inside interface serves as the default gateway for internal clients, the Outside interface connects to the simulated internet, and the DMZ interface provides network access to the public web server. Infrastructure components such as the firewall itself and the DMZ server were assigned static IP addresses to ensure predictable routing behavior, while internal clients receive dynamic addresses through DHCP.

Device	Interface	IP Address	Subnet Mask	Description
ASA Firewall	Vlan 1 (Inside)	192.168.99.1	255.255.255.0	LAN Gateway
	Vlan 2 (Outside)	172.16.0.1	255.255.255.0	ISP Connection
	Vlan 3 (DMZ)	192.168.30.1	255.255.255.0	DMZ Gateway
Inside PCs	NIC	DHCP Assigned	255.255.255.0	Internal Client
Web Server	NIC	192.168.30.2	255.255.255.0	Hosted Server
Border Router	Gi0/1	172.16.0.2	255.0.0.0	Simulated Outside Internet

Figure 2: *IP addressing scheme for Inside, DMZ, and Outside zones used for routing and firewall policy enforcement.*

2.3 Design Constraints and Architectural Decisions

The first design was separate VLANs for the cyan (10.x) and purple (20.x) networks. However, the Cisco ASA 5505 Base License in the simulation environment cannot act as a DHCP server for remote subnets (networks not directly connected to its interface). Solution: To fulfill the requirement of the Firewall acting as the DHCP server, the Cyan and Purple networks were consolidated into a single "Inside" security zone (192.168.99.0/24). This allows the ASA to directly manage DHCP requests from all client PCs.

The DMZ implementation also required specific configuration considerations due to licensing restrictions. The ASA 5505 Base License only allows to use a third VLAN if it’s restricted. To enable the DMZ (Vlan 3), the interface was configured with the command no forward interface vlan 1. Implication: This restricts traffic flow between the Inside network and the DMZ. The internal employees cannot access the DMZ server directly (because of this licensing block), the server is fully accessible to the Outside (Internet), which is the primary use case of a public web server.

2.4 Firewall Service Configuration

2.4.1 DHCP Implementation

The firewall was configured to provide DHCP services to the internal network, automatically assigning IP addresses, subnet masks, default gateway information, and DNS server addresses to client devices. The DHCP address pool was configured within the internal subnet range, with appropriate exclusions for statically assigned infrastructure addresses. This configuration eliminates manual IP address management for internal clients while ensuring consistent network parameter distribution.

```
ciscoasa#show run dhcpd
dhcpd auto_config outside
!
dhcpd address 192.168.99.10-192.168.99.41 inside
dhcpd dns 8.8.8.8 interface inside
dhcpd enable inside
!
!
```

Figure 3: *ASA DHCP configuration showing the address pool and DNS settings for the Inside network.*

2.4.2 Network Address Translation

Dynamic NAT was configured to enable internet access for internal clients while concealing their private IP addresses from the external network. When internal devices initiate outbound connections, the firewall translates their source addresses to the Outside interface IP address. This provides both connectivity and security, as external hosts cannot directly address internal systems.

For the DMZ web server, static NAT was implemented to make the service accessible from the simulated internet. A one-to-one mapping was established between the server's private DMZ address and a public IP address on the Outside interface. This allows external clients to connect to the web server using a publicly routable address while the actual server resides in the protected DMZ subnet.

```
ciscoasa#conf t
ciscoasa(config)#object network internal_supernet
ciscoasa(config-network-object)# subnet 192.168.0.0 255.255.0.0
ciscoasa(config-network-object)# nat (inside,outside) dynamic interface
ciscoasa(config-network-object)#
```

Figure 4: *Dynamic NAT rule translating Inside client traffic to the ASA Outside interface address for outbound internet access.*

```
ciscoasa(config)#object network dmz_server
ciscoasa(config-network-object)# host 192.168.30.2
ciscoasa(config-network-object)# nat (dmz,outside) static 172.16.0.5
ciscoasa(config-network-object)#exit
```

Figure 5: *Static NAT mapping of the DMZ web server to a public IP address to enable inbound access from the Outside network.*

2.4.3 Access Control Configuration

Access Control Lists (ACLs) were applied to the Outside interface to regulate traffic destined for the DMZ. The ACLs permit only specific protocols—HTTP for web service access and ICMP for connectivity testing—to reach the DMZ server. All other inbound traffic from the outside network is implicitly denied. This restrictive approach minimizes the attack surface while allowing essential services to function.

```
ciscoasa#conf t
ciscoasa(config)#access-list OUTSIDE_IN extended permit tcp any host 192.168.30.2 eq 80
WARNING: <OUTSIDE_IN> found duplicate element
ciscoasa(config)#access-list OUTSIDE_IN extended permit icmp any host 192.168.30.2
WARNING: <OUTSIDE_IN> found duplicate element
ciscoasa(config)#access-group OUTSIDE_IN in interface outside
```

Figure 6: *Static NAT configuration for the DMZ web server combined with access control list rules permitting HTTP and ICMP traffic from the Outside network.*

2.4.4 ICMP Inspection Policy

The ASA firewall blocks ICMP traffic by default as a security measure. To enable network troubleshooting and connectivity verification, ICMP inspection was configured through a policy map applied to security zones. This allows ICMP echo requests and replies to traverse the firewall under controlled conditions, supporting operational testing without compromising security posture.

```
ciscoasa(config)#policy-map global_policy
ciscoasa(config-pmap)# class inspection_default
ciscoasa(config-pmap-c)#  inspect icmp
```

Figure 7: ICMP inspection is enabled in the ASA global policy to allow controlled ping troubleshooting across security zones.

2.5 Network Verification

The network configuration was validated through systematic testing. Internal clients successfully obtained IP addresses via DHCP and were able to reach external network addresses, confirming correct NAT operation. The DMZ web server was accessible from the outside network using its mapped public IP address, verifying both static NAT and ACL configuration. ICMP connectivity tests between all network zones confirmed proper routing and firewall inspection behavior. These tests collectively demonstrate that the network infrastructure meets its security and connectivity objectives.

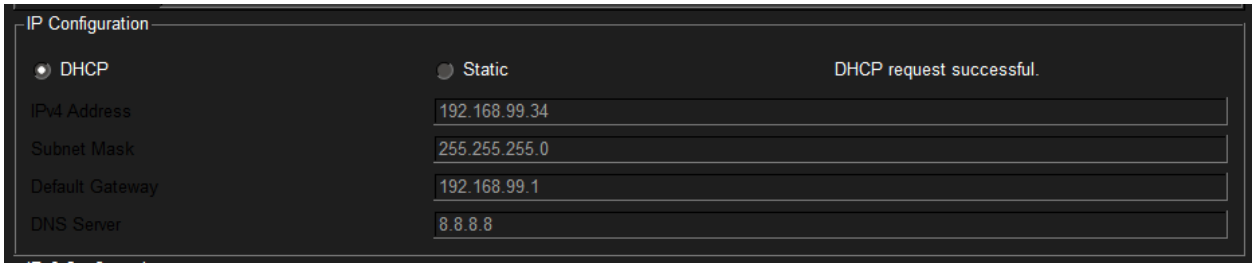


Figure 8: Internal client successfully receives network parameters via DHCP, validating the Inside network DHCP operation.

```
C:\>ping 172.16.0.2

Pinging 172.16.0.2 with 32 bytes of data:

Reply from 172.16.0.2: bytes=32 time<1ms TTL=254
Reply from 172.16.0.2: bytes=32 time<1ms TTL=254
Reply from 172.16.0.2: bytes=32 time=1ms TTL=254
Reply from 172.16.0.2: bytes=32 time=34ms TTL=254

Ping statistics for 172.16.0.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 34ms, Average = 8ms
```

```
Router#ping 172.16.0.5

Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 172.16.0.5, timeout is 2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 0/2/5 ms
```

Figure 9-10: NAT and connectivity verification showing successful ICMP reachability to the DMZ server via its public IP mapping (172.16.0.5).

3. Virtualization Infrastructure

3.1 Virtualization Platform Selection and Installation

Oracle VirtualBox was selected as the virtualization platform due to its multi-platform compatibility, and suitability for development and testing environments.

Oracle VirtualBox version 7.2.4. was installed on a Windows computer, which was used during the project to be able to virtualize the two operating systems.

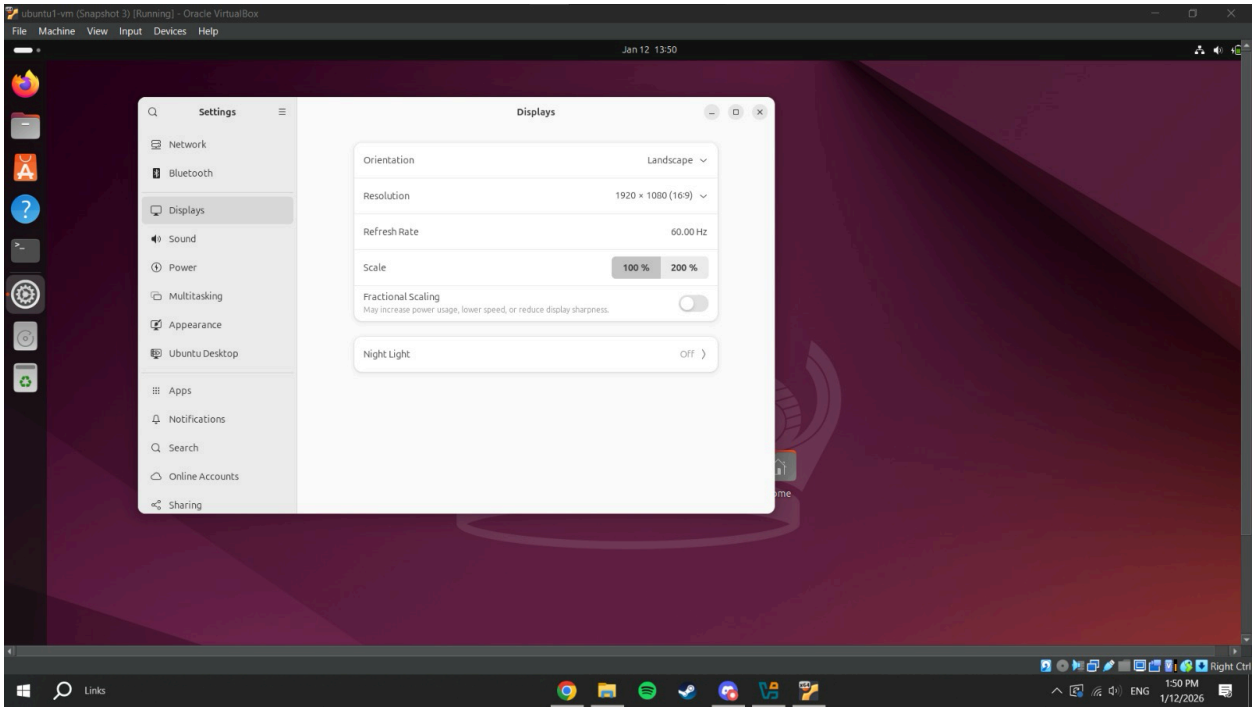


Figure 11: Changing the view of the OS into full-screen mode

3.2 Virtual Machine Provisioning

Two virtual machines were provisioned to represent different server environments within the platform. An Ubuntu 24.04.3 virtual machine was created to serve as the primary application deployment platform where most of the future work will be run on. Debian 13.2.0 virtual machine

was also installed and configured. Together, they will be used to run cross operating system network connectivity.

Each virtual machine was allocated dedicated CPU cores, memory, and storage resources appropriate for its intended workload. The Ubuntu system received additional resources to support Docker container execution and overall heavier workload to run the operating system, while the Debian system was configured with baseline resources sufficient for operating system operation and package management tasks.

Provisioned VM specifications for Ubuntu and Debian, including CPU, memory, storage, and network mode settings:

Debian (13.2.0)	Ubuntu (24.04.3)
Name: debian-vm	Name: ubuntu-vm
Operating System: Debian (64-bit)	Operating System: Ubuntu (64-bit)
CPU: 2 cores	CPU: 4 cores
Base Memory: 2048 MB	Base Memory: 9916 MB
Storage: 20 GB	Storage: 25 GB
Network Mode: NAT Network	Network Mode: NAT Network

3.3 Virtual Network Configuration

Initially, both virtual machines were configured using VirtualBox's default NAT networking mode. This resulted in both systems receiving identical IP addresses within their isolated networks, preventing inter-VM communication. To enable connectivity between virtual machines while maintaining internet access, a custom NAT Network was created within VirtualBox.

The NAT Network provides a shared virtual network segment with its own DHCP server, allowing each virtual machine to obtain a unique IP address within the same subnet. This configuration enables direct communication between virtual machines, while the NAT Network gateway provides internet connectivity to all attached systems. Both virtual machines were reconfigured to use this NAT Network as their primary network interface: ubuntu-vm: 10.0.2.3 and debian-vm: 10.0.2.15

3.4 Connectivity Verification

Network connectivity between virtual machines was verified using ICMP echo requests. Bidirectional ping tests were executed from each virtual machine to the other, confirming successful communication over the virtual network. The successful exchange of ICMP packets demonstrates that both systems have unique IP addresses, correct routing configuration, and functional network interfaces within the virtualized environment.

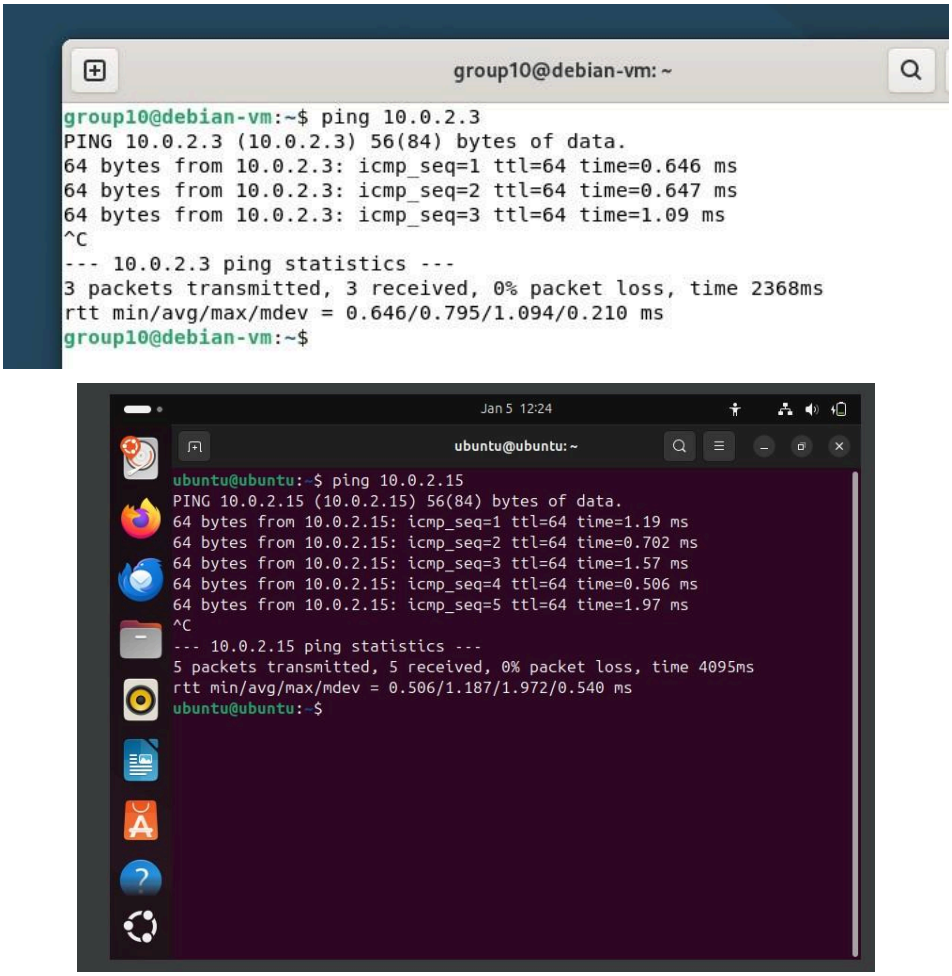


Figure 12-13: *Bidirectional ICMP connectivity tests confirming successful communication between Debian and Ubuntu virtual machines over the shared NAT Network.*

3.5 Operating System Configuration

Operating system configuration was completed on both virtual machines to ensure administrative access and baseline system readiness.

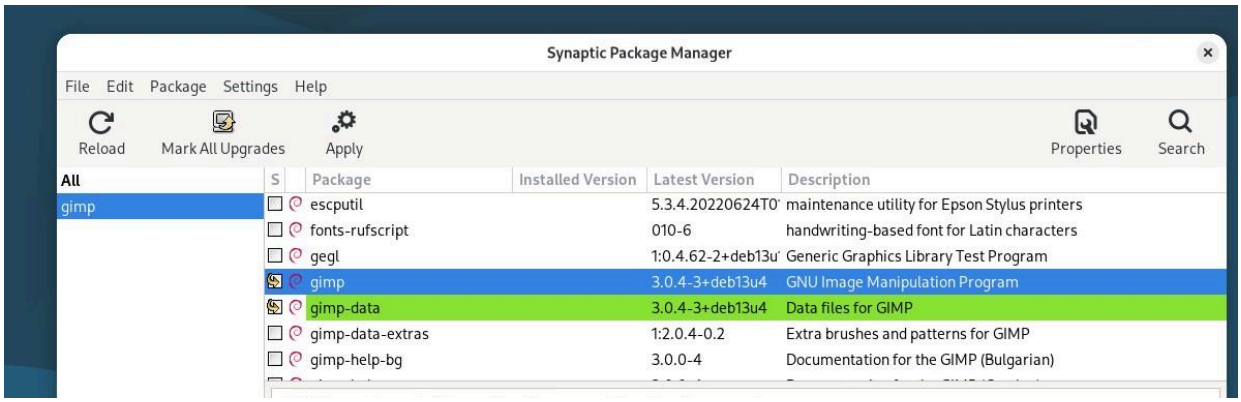
On the Ubuntu system, an administrative user account was verified to have sudo privileges, ensuring permission to perform system-level configuration tasks. Python installation was validated to confirm that the environment was prepared for subsequent application development and container-based Python application deployment. The following commands were used to verify user privileges and Python availability:

```
sudo usermod -aG sudo adminuser
groups adminuser
sudo apt install python3
```

On the Debian system, a user account was configured with administrative privileges to enable system management tasks. The Synaptic package manager was installed to demonstrate graphical package management and software installation capabilities. This confirmed proper integration with Debian package repositories and full operating system functionality:

```
sudo usermod -aG sudo adminuser
groups adminuser
sudo apt install synaptic
```

The virtualization infrastructure provides a stable, interconnected environment for subsequent platform components. The Ubuntu virtual machine serves as the foundation for Docker installation and containerized application deployment, while both systems demonstrate the platform's capability to support multiple Linux distributions within a unified virtual network.




```
group10@debian-vm:~$ sudo apt update
Hit:1 http://deb.debian.org/debian trixie InRelease
Hit:2 http://deb.debian.org/debian trixie-updates InRelease
Hit:3 http://security.debian.org/debian-security trixie-security InRelease
All packages are up to date.
group10@debian-vm:~$ sudo apt install synaptic -y
synaptic is already the newest version (0.91.7).
Summary:
  Upgrading: 0, Installing: 0, Removing: 0, Not Upgrading: 0
```

Figure 14-15: *Synaptic Package Manager showing the installation of GIMP as the required graphical photo editing application on the Debian system.*

4. Docker Engine Implementation

4.1 Docker Installation

Docker was installed on Ubuntu by following the official Docker Desktop documentation. In the document, there are prerequisites, the installation instructions and post installation instructions. The prerequisites include meeting the general system requirements, having an x86-64 system running Ubuntu 22.04, 24.04, or the latest non-LTS version, and ensuring that gnome-terminal is installed if the system is not using the GNOME desktop environment. Since the system is already running GNOME, this step can be skipped.

The first step involved setting up Docker's package repository by running the following command provided in the terminal:

```
# Add Docker's official GPG key:
sudo apt update
sudo apt install ca-certificates curl
```

```
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
sudo tee /etc/apt/sources.list.d/docker.sources <<EOF
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: $(. /etc/os-release && echo
"${UBUNTU_CODENAME:-$VERSION_CODENAME}")
Components: stable
Signed-By: /etc/apt/keyrings/docker.asc
EOF

sudo apt update
```

Figure 16: *Initial Docker installation steps on Ubuntu.*

The following step involves downloading and installing a Debian (DEB) package provided. In the terminal, a change directory command was used to move into the “Downloads” folder. Further on, an installation command that is provided was used.

```
cd Downloads/
sudo apt-get install ./docker-desktop-amd64.deb
```

Figure 17: *Final Docker installation steps on Ubuntu.*

4.2 Docker Installation Verification

Docker was successfully installed on Ubuntu using Docker’s official APT repository. The installation was verified by checking the Docker version using the “docker - -version” command. The output is shown below.

```
adminuser@adminuser-VirtualBox:-$ docker --version
Docker version 28.2.2, build 28.2.2-0ubuntu1-24.04.1
```

Figure 18: *Docker version output confirming successful installation of the Docker Engine on Ubuntu.*

4.3 “hello world” & “Ubuntu” Container Execution

After installing Docker, a Docker account was created. The next part of the task is to pull the “hello world” image from Docker Hub and run it. This was done by firstly using the “docker pull” command to download a Docker image from Docker Hub to the local machine. Next, the “docker run” command was used to create and start a container from a specified Docker image (in this case it is called “hello-world”).The output is shown below.

```
adminuser@adminuser-VirtualBox:-$ docker pull hello-world
adminuser@adminuser-VirtualBox:-$ docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm64v8)
 3. The Docker daemon created a new container from that image which runs
the executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent
it to your terminal.
To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/
For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Figure 19: *Execution of the official hello-world Docker image*

Moving on to the next part of the task, the goal is to pull an ubuntu image from the docker hub and run it. This was done by firstly using the “docker pull” command to download a Docker image from Docker Hub to the local machine. Next, the “docker run” command was used to create and start a container from a specified Docker image (in this case, it is called “ubuntu”). A few more commands, like “-it” serving the purpose of making the terminal interactive and “--name,” were added to assign a custom name (team10itplatform) to the container instead of using a randomly generated name. “Docker ps” is lastly used to check if there are active containers that were checked to verify that the image was running correctly. The output is shown below.

```
adminuser@adminuser-VirtualBox:-$ docker pull ubuntu
adminuser@adminuser-VirtualBox:-$ docker run -it --name team10itplatform
ubuntu
root@a696dcd1a86a:/#
adminuser@adminuser-VirtualBox:-$ docker ps
CONTAINER ID   IMAGE          COMMAND      CREATED         STATUS     PORTS      NAMES
ca9321ab6195    ubuntu:latest  "/bin/bash"  9 seconds ago  Up 8 seconds          itteam10platform
```

Figure 20: *Interactive execution of an Ubuntu container and verification of the running container using the docker ps command.*

The last part of the task is, in the ubuntu container and find the bin folder and create a new ls command. This was done by using command touch /bin/task3newcommand was executed to create a new empty file named task3newcommand in the /bin directory, This was followed by ls bin to list the contents of the bin directory and verify that the new file was successfully created.

```
adminuser@adminuser-VirtualBox:-$ touch /bin/task3newcommand
adminuser@adminuser-VirtualBox:-$ ls bin
```

Figure 21: *File system interaction inside the Ubuntu container showing creation of a new file within the /bin directory.*

5. Custom Application Containerization

5.1 Application Design and Project Structure

A custom Python application was developed to demonstrate the containerization workflow for purpose-built software. A dedicated project directory named task-docker-app was created to contain all application-related files. This directory-based organization ensures clear separation between different project components and serves directly as the Docker build context during image creation.

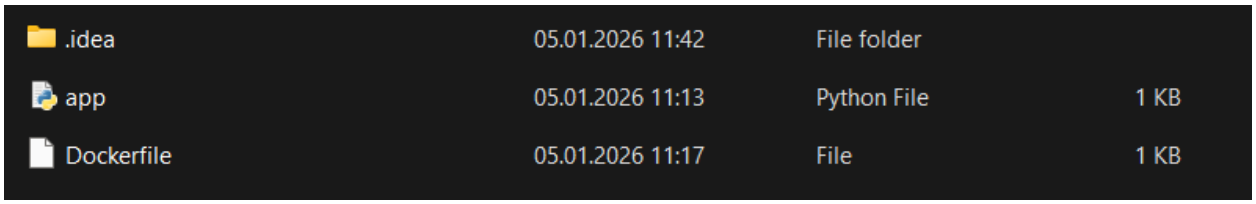


Figure 22: *Project directory structure for the custom Dockerized Python application, including the application file and Dockerfile.*

The application itself consists of a simple Python program that outputs text to the terminal when executed. The application logic was intentionally kept minimal, as the primary objective was to demonstrate containerization principles—application packaging, isolation, and reproducible execution—rather than complex application functionality. The visible terminal output provides clear evidence that the application executes within the container environment.

5.2 Dockerfile Configuration

A Dockerfile was created within the task-docker-app directory to define the container image build process. The Dockerfile specifies a lightweight Python base image from Docker Hub, establishes a working directory within the container, copies application files from the build context into the container image, and defines the command that executes automatically when containers are created from the image.

This declarative configuration approach enables automated, reproducible image builds. The Dockerfile captures all dependencies and execution requirements, ensuring that the application runs identically regardless of the host environment.

5.3 Image Build and Container Execution

The Docker image was built using the docker build command with the task-docker-app directory as the build context. During the build process, Docker processed the Dockerfile instructions

sequentially, pulled the specified base image from Docker Hub, and created a new image layer containing the application files and execution configuration. Successful build completion confirmed that the Dockerfile syntax was correct and all required resources were available.

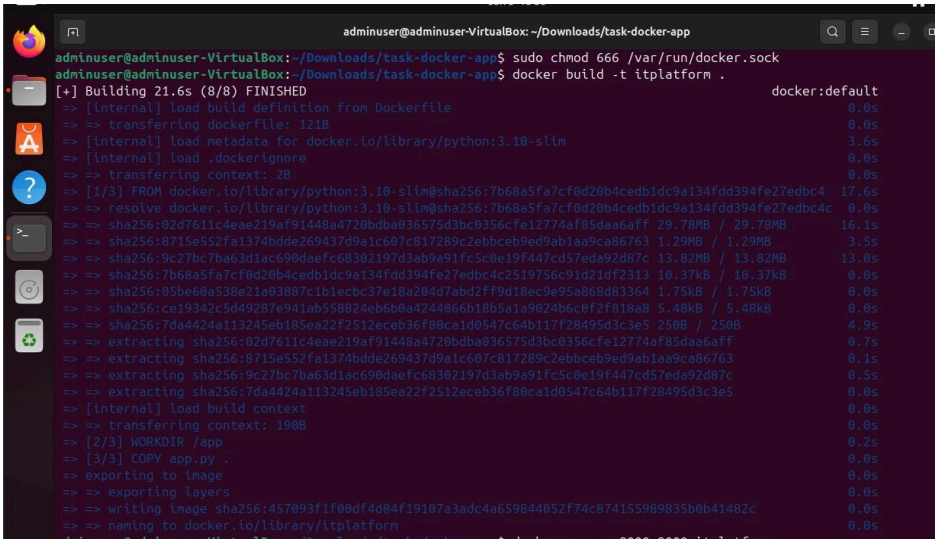


Figure 23: *Terminal output showing a successful Docker image build process for the custom Python application.*

A container was then instantiated from the built image using the docker run command. Upon container startup, the Python application executed automatically as defined in the Dockerfile, and its output appeared in the terminal. This confirmed that the application runs within the containerized environment rather than on the host system, demonstrating successful application isolation and containerization.

Figure 24: *Execution of the custom Docker container displaying the Python application output inside the container environment.*

6. Web Application Deployment

6.1 Flask Application Architecture

A Python Flask web application was developed to demonstrate containerized web service deployment. Flask is a lightweight web framework well-suited for demonstrating fundamental web application concepts, including HTTP request handling, server-side template rendering, and dynamic content generation.

The application follows standard Flask project conventions, with HTML templates stored in a dedicated templates directory. This structure allows Flask's template engine to locate and render templates during request processing.

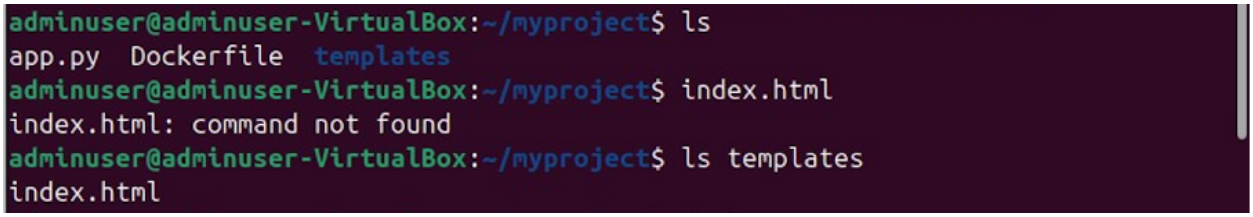


Figure 25: *Project directory structure following Flask conventions, showing the application file, Dockerfile, and templates directory containing index.html.*

The application defines a single HTTP route that renders an HTML template and passes dynamic data to it during the rendering process.

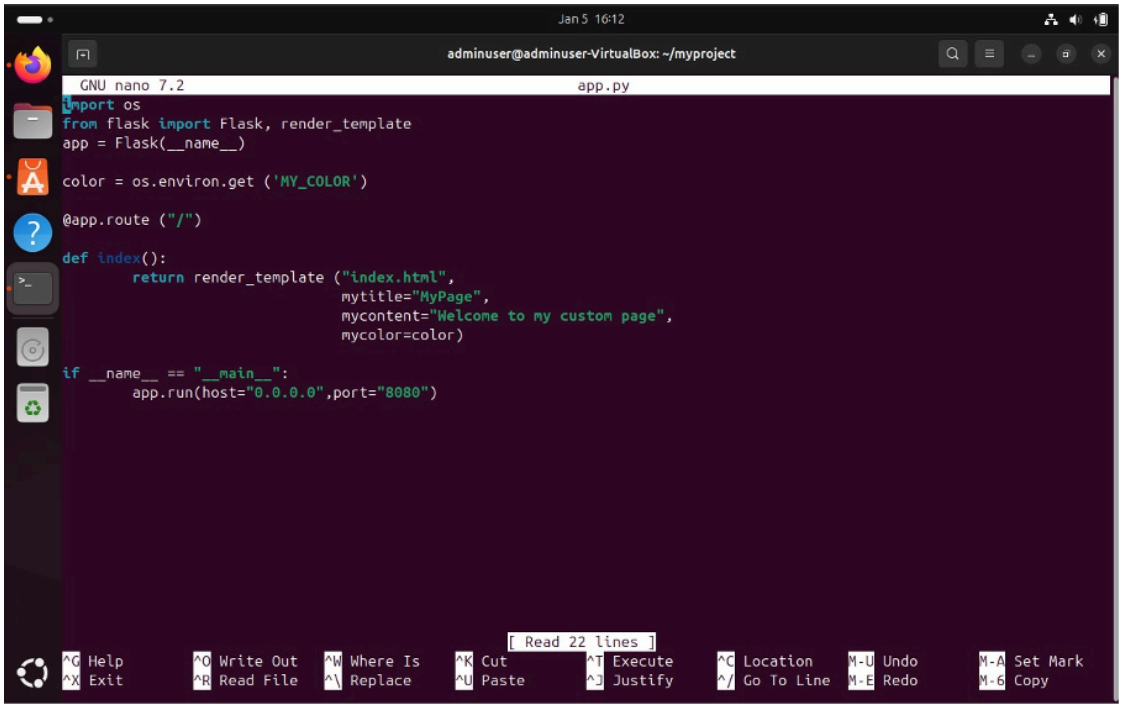


Figure 26: *Flask application backend code defining a single route and rendering an HTML template with dynamic variables.*

6.2 Dynamic Configuration Through Environment Variables

The Flask application retrieves an environment variable to control the background color of the rendered webpage. This design demonstrates runtime configuration capabilities without modifying application source code. Environment variables provide a standard mechanism for configuring containerized applications, allowing the same container image to be executed with different configurations in different environments.

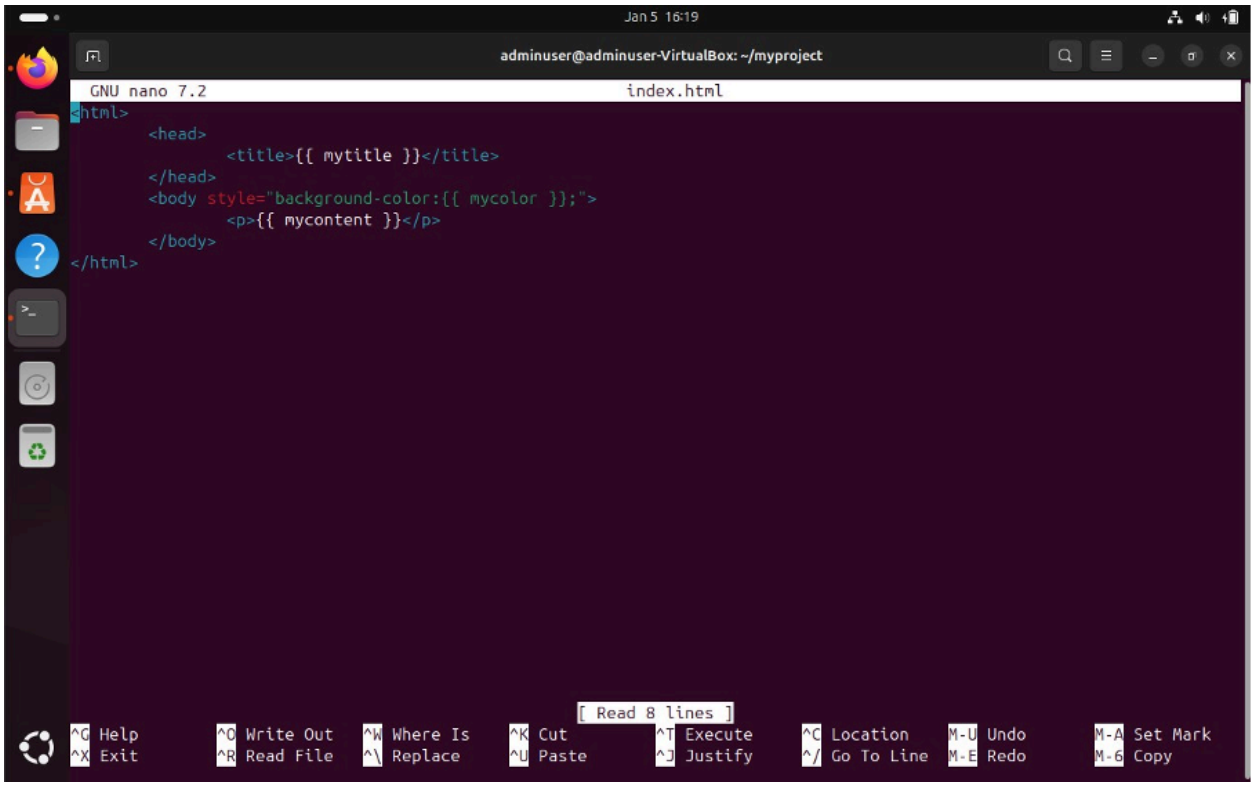


Figure 27: *HTML template (index.html) using Flask placeholders for dynamic content rendering and background color configuration.*

The use of environment variables for configuration follows the Twelve-Factor App methodology, which advocates for strict separation between code and configuration. This approach enables operational flexibility and aligns with cloud-native application design principles.

6.3 Web Application Containerization

A Dockerfile was created for the Flask application, defining the containerization process for the web service. The Dockerfile uses a Python base image, copies all application files, including the templates directory, into the container, installs the Flask framework using pip, exposes the network port on which the Flask application listens, and defines the command to start the Flask development server automatically when the container runs.

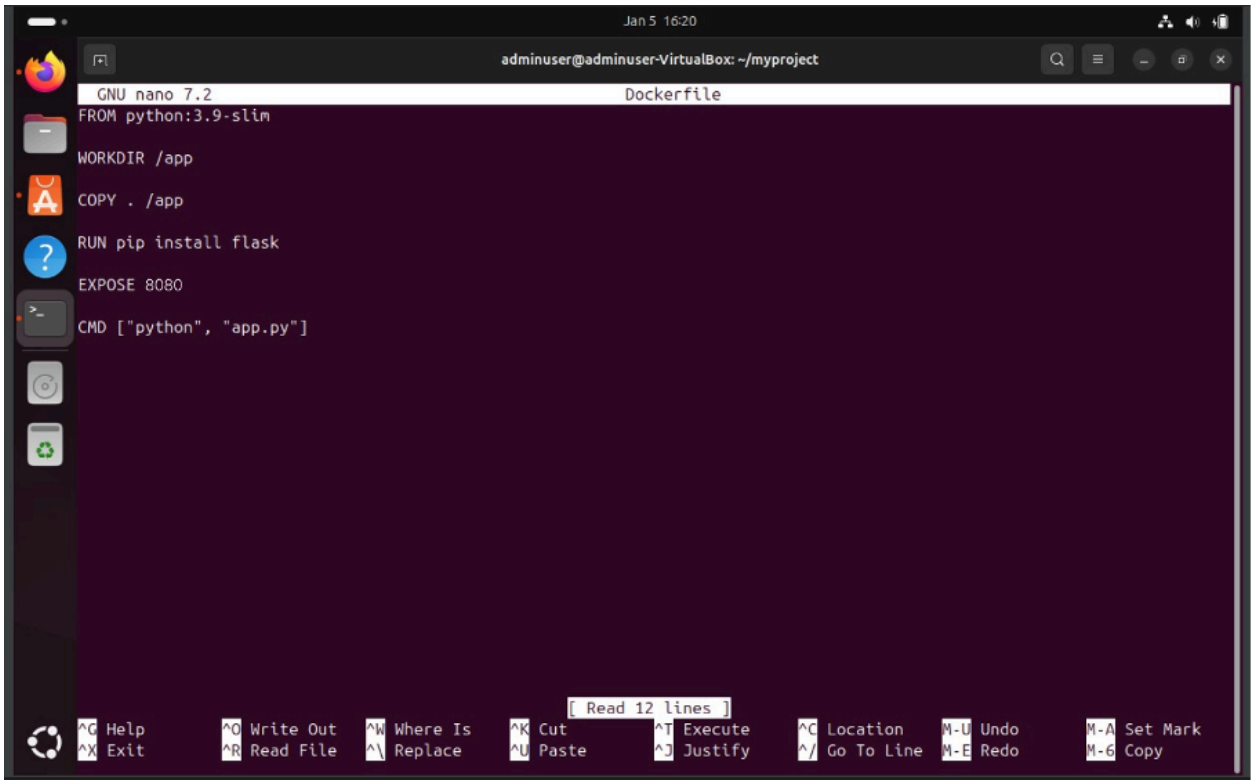


Figure 28: *Dockerfile defining the containerized Flask application environment, including base image, dependency installation, port exposure, and execution command.*

The image build process followed the same workflow as the custom Python application, with Docker processing the Dockerfile instructions and creating a container image that encapsulates both the application code and its runtime dependencies.

```
adminuser@adminuser-VirtualBox:~/myproject$ docker build -t myflaskapp .
```

Figure 29: *Docker image build process for the Flask web application using a custom Dockerfile.*

6.4 Service Exposure and Runtime Configuration

The Flask container was executed with port mapping to make the web service accessible from the host system.

```
adminuser@adminuser-VirtualBox:~/myproject$ docker run -p 8080:8080 -e MY_COLOR=
white myflaskapp
```

Figure 30: Execution of the Flask application container with port mapping and environment variable configuration.

The Docker port mapping mechanism forwards traffic from a host port to the container's internal port, allowing the browser running on the host to connect to the web service running inside the container.

An environment variable was passed to the container at runtime using the -e flag in the docker run command. This variable controls the webpage background color, demonstrating how containerized applications can be configured dynamically without rebuilding the container image. The Flask application reads this environment variable during request processing and includes its value in the rendered HTML output.

6.5 Web Service Verification

The Flask application was verified by accessing it through a web browser using the host port specified in the port mapping configuration.

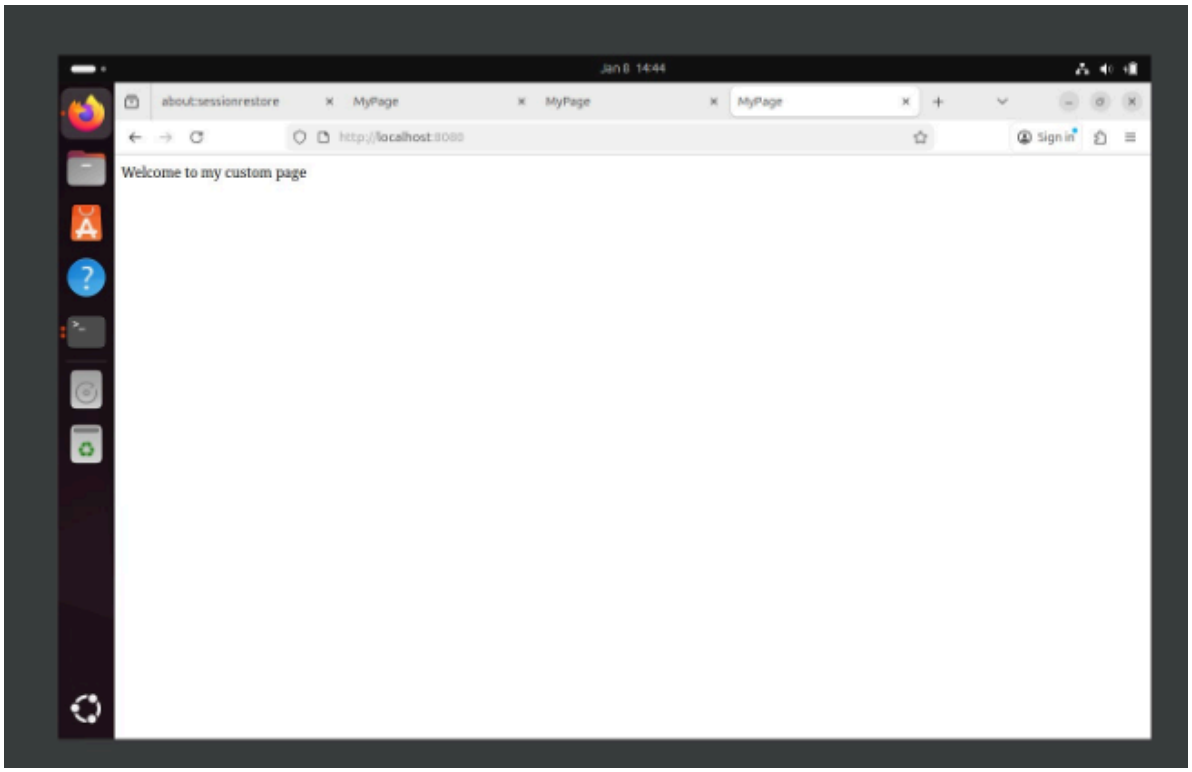


Figure 31: *Web browser output showing the running Flask application accessed through Docker port mapping and configured via environment variables.*

The webpage loaded successfully, displaying the expected content, including the page title and application output. The background color matched the value provided through the environment variable, confirming that the application correctly processes runtime configuration parameters.

This verification demonstrates several key containerization concepts: network connectivity between the host and containerized services, proper port mapping configuration, successful environment variable injection, and correct application behavior within the isolated container environment. The successful browser-based access confirms end-to-end functionality of the containerized web service.

7. Testing and Validation

The IT platform was validated through systematic testing at each architectural layer. Network layer testing confirmed that the firewall correctly implements security policies, DHCP services provide automatic IP configuration to internal clients, NAT translations enable both outbound internet access and inbound DMZ connectivity, and ACLs restrict traffic appropriately between security zones. ICMP connectivity tests verified routing behavior across all network segments.

Virtualization layer testing demonstrated that multiple virtual machines can operate concurrently on shared hardware, virtual network configuration enables inter-VM communication, and each virtual machine maintains an isolated operating environment. Connectivity tests between virtual machines confirmed network functionality within the virtualized infrastructure.

Container platform testing validated Docker installation and daemon operation, container image retrieval from Docker Hub, container instantiation and execution, file system isolation between containers and the host, and the correct behavior of both batch and service-oriented containerized applications. The custom Python application demonstrated basic containerization principles, while the Flask web service verified network service deployment, port mapping, environment variable injection, and browser-based accessibility.

Collectively, these tests confirm that all platform components function correctly both individually and as an integrated system. Each layer successfully builds upon the capabilities of the layer below it, demonstrating the hierarchical nature of modern IT infrastructure.

8. Conclusion

This project successfully implemented a complete IT platform integrating network security, virtualization, and container-based application deployment. The layered architecture approach demonstrates how different infrastructure technologies work together to create a functional environment for hosting enterprise applications.

The network security layer established a protected environment with appropriate traffic controls, zone isolation, and controlled internet connectivity. The virtualization layer provided resource consolidation and the ability to run multiple operating systems on shared hardware. The containerization layer demonstrated modern application deployment practices, including application packaging, isolation, reproducible execution, and runtime configuration.

Each component was configured according to industry practices and verified to ensure correct operation. The integration of these components demonstrates fundamental principles of modern IT infrastructure design: defense in depth for security, resource abstraction through virtualization, and application portability through containerization. The platform provides a foundation that could support more complex applications and services while maintaining security, isolation, and operational flexibility.

The successful implementation and verification of all components confirms that the project objectives were achieved and that the platform meets the requirements of a functional IT infrastructure suitable for hosting secure, scalable applications in a simulated enterprise environment.