



**Prifysgol Cymru**  
Y Drindod Dewi Sant  
**University of Wales**  
Trinity Saint David

**Name:** Patel Vinay Bhikhabhai

**Student ID:** 2519364

**Course Name:** MSc Cybersecurity and Digital Forensics

**Module:** Cloud Engineering Principles and Practice

**Module Code:** ACCA7027\_LON

**University:** University of Wales, Trinity Saint David

---

## Table of Contents

- 1) Abstract
- 2) Introduction
- 3) Foundations of Virtualisation and Containerisation Technologies
  - 3.1 Principles of Virtualisation
  - 3.2 Types of Virtualisation (Type 1 & Type 2 Hypervisors)
  - 3.3 Limitations of Virtual Machines
  - 3.4 Principles of Containerisation
  - 3.5 Comparison: Virtualisation vs Containerisation
  - 3.6 Importance of Containerisation in Cloud Computing
- 4) Docker and Container Technologies
  - 4.1 Overview of Docker
  - 4.2 Docker Architecture (Client, Daemon, API)
  - 4.3 Docker Components
    - 4.3.1 Docker Images
    - 4.3.2 Docker Containers
    - 4.3.3 Docker Registries (DockerHub)
  - 4.4 Docker Networking
    - 4.4.1 Bridge Network
    - 4.4.2 Host Network
    - 4.4.3 Overlay Network
  - 4.5 Docker Storage and Volumes
    - 4.5.1 Creation and Management of Volumes
    - 4.5.2 Persistence of Data in Containers
- 5) Practical Implementation
  - 5.1 Development Environment Setup

- 5.2 Application File Creation and Verification
- 5.3 Creating the Dockerfile
- 5.4 Building Docker Image
- 5.5 Running the Container
- 5.6 Tagging and Pushing to DockerHub
- 5.7 Docker Networking Implementation
  - 5.7.1 Creating Docker Network
  - 5.7.2 Connecting Containers
  - 5.7.3 Installing Network Tools
  - 5.7.4 Testing Communication
  - 5.7.5 Inspecting Network
  - 5.7.6 Verifying Containers
- 5.8 Docker-Based Web Server Implementation
  - 5.8.1 HTML Page Creation
  - 5.8.2 Project Setup
  - 5.8.3 Dockerfile Creation
  - 5.8.4 Building Image
  - 5.8.5 Running Container
  - 5.8.6 Verification Using Logs
- 5.9 Practical Conclusion
- 6) Cloud Computing Terminologies
  - 6.1 Load Balancing
    - 6.1.1 Definition and Significance
    - 6.1.2 Implementation in AWS
  - 6.2 Scalability
    - 6.2.1 Vertical vs Horizontal Scaling
    - 6.2.2 AWS Auto Scaling
  - 6.3 Application Gateways
    - 6.3.1 Role and Functionality
    - 6.3.2 AWS API Gateway / ALB
- 7) Integration and Evaluation
  - 7.1 Relationship Between Load Balancing and Scalability
  - 7.2 Why They Are Complementary
  - 7.3 Benefits of Combining Technologies
  - 7.4 Impact on Performance, Security, and Reliability
- 8) Driving Forces of Cloud Computing
  - 8.1 Cost Efficiency
  - 8.2 Flexibility and Elasticity
  - 8.3 Innovation and Managed Services
  - 8.4 Industry Adoption Trends
- 9) Serverless vs Containerisation
  - 9.1 Introduction to Serverless Computing
  - 9.2 AWS Lambda / Azure Functions
  - 9.3 Comparison with Containerisation
  - 9.4 Advantages and Limitations
  - 9.5 Use Cases
- 10) Conclusion
- 11) References

## **1. Abstract**

=> This paper provides an insight into the application of containerisation in cloud computing with particular emphasis on the Docker technology. The study commences with the examination of virtualisation concepts and how the container technology takes advantage of them in order to enhance portability and application efficiency. Advantages of container architecture are discussed and they include scalability and easy deployments.

=> The practical aspect of using container architecture in cloud computing is illustrated through the implementation of Docker containers, which include image creation, storage, network configuration, and publication of images in Docker Hub. Also, key components in cloud computing, namely load balancing, scalability, and application gateways, are described along with how they work.

=> Lastly, factors behind cloud computing adoption are identified, which include cost-effectiveness and flexibility. Additionally, the pros and cons of two approaches used in cloud computing, namely, containerisation and serverless computing based on function as service are compared.

## **2. Introduction**

=> Cloud computing has revolutionised the way applications and services are offered to consumers over the internet. Instead of using costly hardware, cloud computing allows organisations to offer their services through internet-based platforms. These services include computing resources such as storage space, networking, and processing capabilities. Cloud computing has changed the way developers create, deploy, and manage applications.

=> One of the most important technologies used in developing cloud computing is virtualisation technology. Virtualisation technology involves creating virtual machines that enable multiple computers to operate simultaneously within a physical machine. Although virtualisation technology increases the efficiency of computing resources, it is still quite resource-intensive because it requires running multiple operating systems simultaneously. Therefore, containerisation technology has evolved to offer better solutions than virtualisation.

=> The Docker platform allows users to develop container images, which can be run by the computer as separate processes. These images are built from scratch but with the necessary dependencies installed. Furthermore, Docker offers a platform for building, managing, and deploying images. Images can be stored on a repository like Docker Hub.

## **3. Foundations of Virtualisation and Containerisation Technologies**

### **3.1 Principles of Virtualisation**

=> Virtualisation is one of the key technologies in cloud computing, which makes it possible to create several virtual environments on one physical machine. Virtualisation is achieved through the separation of hardware resources, including CPU, RAM, and storage capacity, among others, which are then allocated to several VMs. Virtualisation is made possible through a software layer called a hypervisor.

=> The principle underlying virtualisation is the efficient allocation of resources. Rather than dedicating one physical machine to one application, virtualisation makes it possible to have several operating systems and applications run concurrently within one machine.

### **3.2 Types of Virtualisation (Type 1 & Type 2 Hypervisors)**

=> The main function of hypervisors is to facilitate virtualisation, which is usually done in two ways.

=> Type 1 hypervisors, which are otherwise referred to as bare-metal hypervisors, do not require a host operating system to run. They execute directly on the underlying physical hardware. As a result, type 1 hypervisors offer higher performance and efficiency compared to type 2. They include products such as VMware ESXi and Microsoft Hyper-V.

=> Type 2 hypervisors are installed on top of the host operating system to manage virtual machines. They are easier to set up than type 1 hypervisors, but their performance levels tend to be lower because of the overhead associated with host operating systems. They include products such as Oracle VirtualBox and VMware Workstation.

### 3.3 Limitations of Virtual Machines

=> Despite the great features of virtual machines, there are some disadvantages associated with virtualization that have been responsible for the development of containers.

=> The first disadvantage is high resource utilization. Every virtual machine requires a whole operating system to be installed, and thus much memory space, disk space, and processing power are consumed. The result is reduced efficiency of VMs, particularly if many VMs are involved.

=> The second disadvantage is high startup time. In order to start a new VM, one needs to boot up the whole operating system, which may take quite some time.

### 3.4 Principles of Containerisation

=> Containerisation is a sophisticated technology for deploying software applications that works at the level of the operating system and not at the level of hardware. Unlike other technologies that rely on machine virtualization, containerisation packages applications and their dependencies as isolated components that share the kernel of the host machine.

=> The primary principle behind containerisation is isolation. Containers achieve isolation using mechanisms like namespaces and control groups.

### 3.5 Comparison Table: Virtualisation vs Containerisation

| Key Characteristics | Virtualisation (VMs)                                                        | Containerisation (Containers)                                   |
|---------------------|-----------------------------------------------------------------------------|-----------------------------------------------------------------|
| Architecture        | Uses a hypervisor to create multiple virtual machines, each with its own OS | Uses container engine (e.g., Docker) sharing the host OS kernel |
| Operating System    | Each VM includes a full operating system                                    | Containers share the host OS kernel                             |
| Resource Usage      | High (requires more CPU, RAM, and storage)                                  | Low (lightweight and efficient)                                 |
| Startup Time        | Slow (minutes to boot OS)                                                   | Fast (seconds or milliseconds)                                  |
| Performance         | Lower efficiency due to overhead                                            | High efficiency with minimal overhead                           |
| Isolation           | Strong isolation between VMs                                                | Process-level isolation (slightly less than VMs)                |
| Scalability         | Slower to scale due to heavy resource requirements                          | Easily scalable and suitable for dynamic workloads              |
| Deployment Speed    | Slower deployment                                                           | Rapid deployment                                                |
| Management          | More complex (managing multiple OS instances)                               | Simpler management using container tools                        |
| Examples            | VMware, VirtualBox, Hyper-V                                                 | Docker, Kubernetes                                              |

### 3.6 Importance of Containerisation in Cloud Computing

=> Containerisation has become an integral part of contemporary cloud computing because it allows for effective scaling and portability in application deployment. Containerisation helps build cloud-native applications by creating services that can operate within a distributed environment.

=> A vital function of containerisation is facilitating the microservices architecture. It allows applications to be split into small services that can be separately built, deployed, and scaled.

## 4. Docker and Container Technologies

### 4.1 Overview of Docker

=> Docker is an open-source technology that aims to make easier to develop, deploy and manage the application through containerization. Docker allows users to create a container that includes the application and all the required dependencies, libraries and configuration. Thus, it guarantees that the same application behaves uniformly on all platforms such as development, testing and production environments.

=> Docker is considered one of the important technologies that have been introduced in the domain of cloud computing. The reason is that Docker helps users to achieve their goals in terms of quick deployment, scalability and portability. By addressing compatibility concerns among systems, Docker makes easy for organizations to use the approach of DevOps that involves CI/CD practices.

### 4.2 Architecture of Docker (Client, Daemon, API)

=> Docker employs a client-server architecture made up of various elements used to control containers.

=> The **Docker Client** is the user's interface when using Docker. Using the Docker Client, the user can issue commands like building images and running containers. All commands given using the Docker Client are processed by the Docker Daemon.

=> The **Docker Daemon** is the main software that runs in the background and helps to control various Docker objects like images, containers, network, and volumes. The Docker Daemon performs various actions based on requests from the Docker Client.

=> The **Docker REST API** serves as the middleware between the Docker Client and Docker Daemon. This helps other software to communicate with Docker and perform various tasks automatically.

### 4.3 Docker components

#### 4.3.1 Docker images

=> Images are immutable templates from which containers are generated. Everything necessary for running the application is included in such templates. This includes the software code, operating environment, libraries, and dependencies. Images are built based on a Dockerfile that contains a set of instructions for building images.

=> An important feature of Docker images is that they consist of multiple layers. Every instruction in the file creates a layer that enables more efficient use of storage resources and caching. As such, images can be stored, reused, distributed, and versioned. The ability to do so makes images vital in consistent application deployment.

#### 4.3.2 Docker Containers

=> A Docker container is a process-based isolated and portable executable environment generated from an image. Containers enable the execution of programs in an isolated environment that does not interact with other processes in the host environment. Containers are lightweight, easy to delete or generate, and very efficient.

=> Containers are temporary, i.e., they can be terminated and discarded without affecting the under-

lying image template. However, the data produced by running applications inside containers can be stored persistently using volumes. In your practical work, containers have been used to run the Flask application.

### **4.3.3 Docker Registries (DockerHub)**

=> A Docker Registry is simply a place where Docker images can be stored. The DockerHub registry is the most popular open-source registry. It allows developers to upload, distribute, and download Docker images.

=> Version control is facilitated using tags within a repository. This makes it possible to keep track of various versions of the same application. In your practical exercise, you had to push images to DockerHub before pulling them for portability and remote execution.

## **4.4 Docker Networking**

=> Docker networking is essential in allowing containers to interact with one another and with other external entities. It comes with various network drivers suited to their intended applications.

### **4.4.1 Bridge Network**

=> The bridge network is the default network driver in Docker. It allows the containers to communicate with one another through their internal IP addresses or container names. It is an ideal network driver for development purposes.

=> In your practical work, you were tasked with setting up a customized bridge network, which will allow the containers to interact with one another.

### **4.4.2 Host Network**

=> With the host network driver, the network stack is shared among both the host and the container. As such, there is no need to provide the container with its own IP address. This option is recommended for applications requiring high performance.

### **4.4.3 Overlay Network**

=> Overlay network is a technique that allows us to link containers running on different host machines together. These networks are frequently implemented in distributed systems and container management systems like Kubernetes and Docker Swarm. The overlay network facilitates scalable communication in a cloud-based environment.

## **4.5 Docker Storage and Volumes**

=> Containers are stateless by default; therefore, any information saved within them will disappear once the container has been removed. For this purpose, Docker offers various ways to manage storage, one of which involves the use of volumes.

### **4.5.1 Creation and Management of Volumes**

=> Volumes are created for storage purposes, as they allow saving data independent of the container. The following command may be used for this purpose:

**docker volume create myvolume**

=> This allows you to attach a container to a particular volume so that data remains available even after removing the associated container.

### **4.5.2 Persistence of Data in Containers**

=> Many types of applications require data persistence. For example, a database application needs

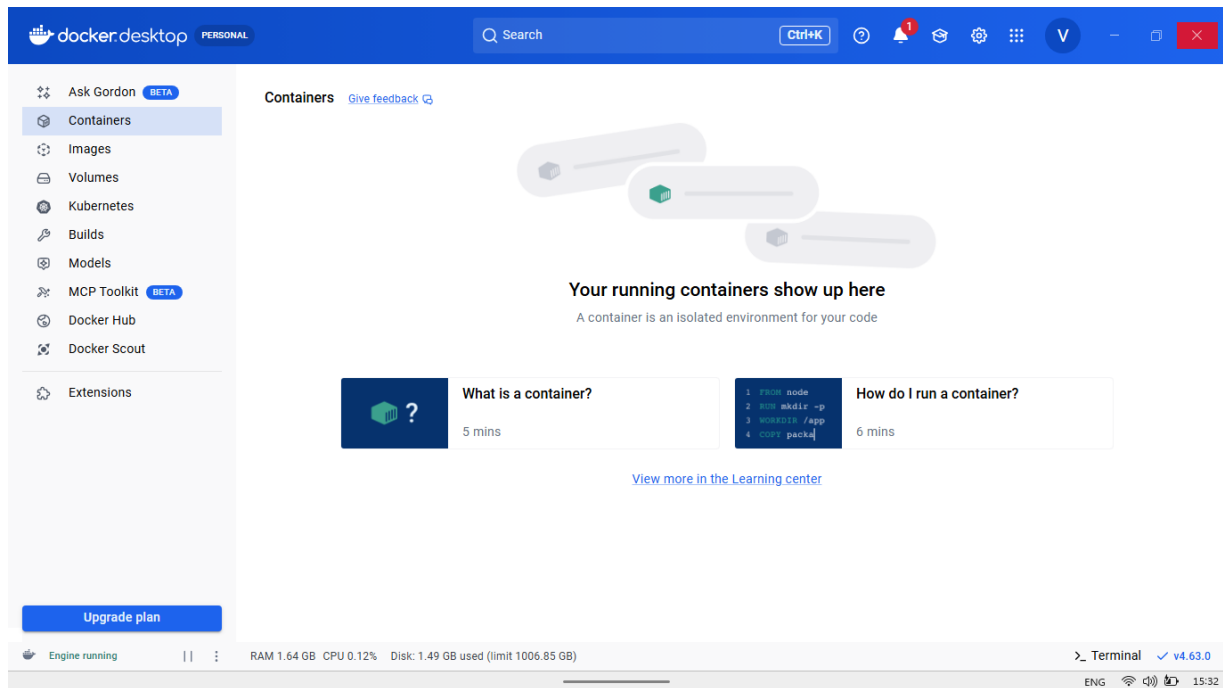
its data to stay intact even if the underlying container was terminated.

=> Your practical tasks involved managing containers' data persistence through the use of volumes. This approach follows modern best practices of cloud computing that require maintaining data integrity alongside scalability and flexibility.

## 5. Practical Implementation

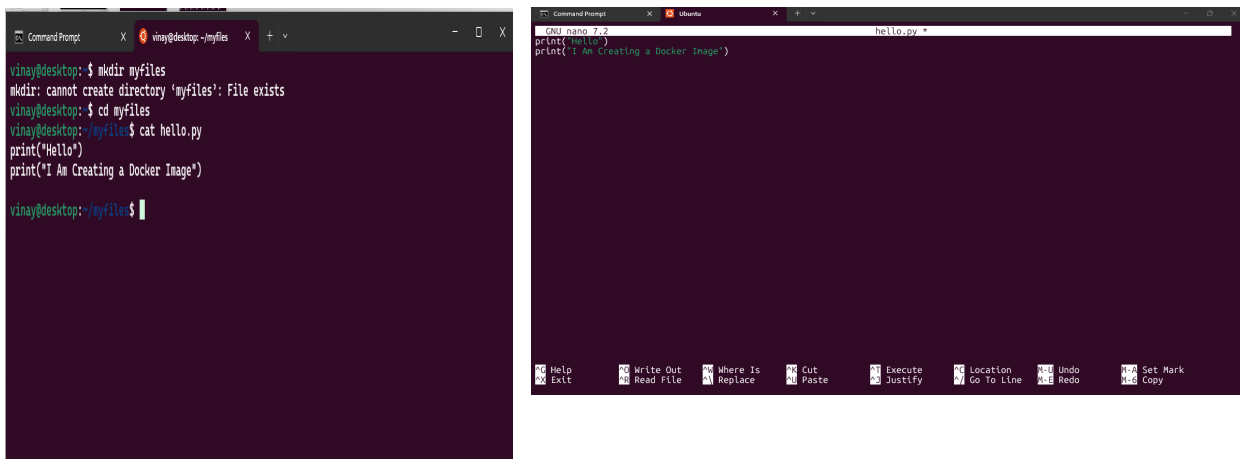
### 5.1 Development Environment Setup (Windows + Docker Desktop)

=> This confirms that the Docker engine is successfully running on the Windows system..



=> This setup offers all the necessary resources required for the development, management, and deployment of containers. Docker Desktop works seamlessly with WSL 2, thus offering the capability to use Linux containers on a Windows machine.

### 5.2 Application File Creation and Verification

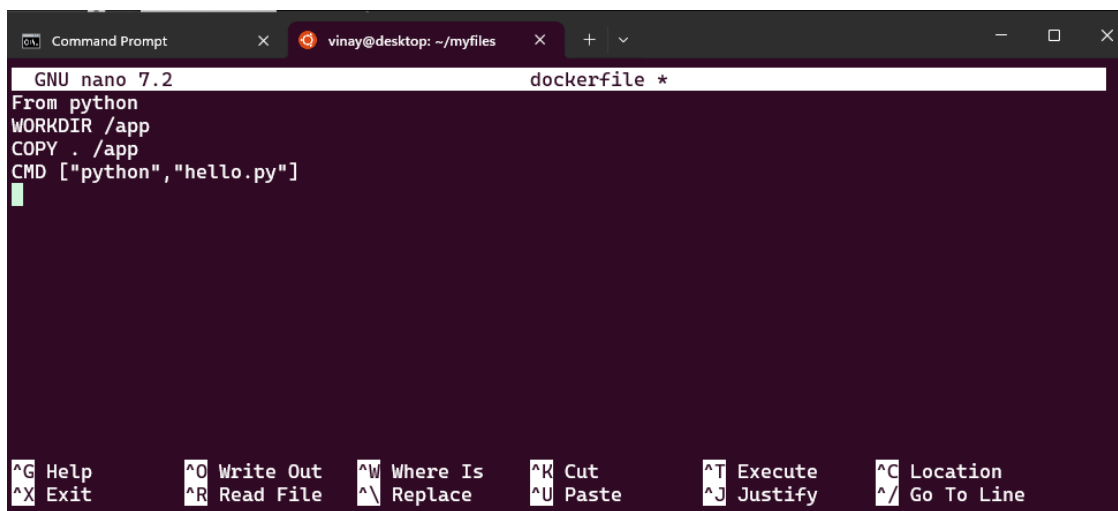


=> In this chapter, attention is paid to developing and testing of the application file that will be further containerized using Docker. A very simple Python script (hello.py) has been created for demonstration purposes.

=> First, the application file was created a Linux-based terminal environment. It contains simple Python commands used to output information to the terminal window in order to check whether the application operates properly. This is necessary in order to avoid any possible bugs when making containers and ensure that the application runs smoothly inside the container.

=> After the file was created, its correctness was checked with terminal commands. The cat hello.py command helped to view the content of the file in question and make sure that it was made without mistakes. Any mistakes made when writing the file may result in a failure while creating and running containers based on this file.

### 5.3 Creating the Dockerfile



```
GNU nano 7.2 dockerfile *
From python
WORKDIR /app
COPY . /app
CMD ["python", "hello.py"]

^G Help      ^O Write Out  ^W Where Is   ^K Cut        ^T Execute    ^C Location
^X Exit      ^R Read File  ^\ Replace    ^U Paste      ^J Justify    ^_ Go To Line
```

=>A Dockerfile is a configuration file used to build a Docker image automatically with the help of specific instructions. The configuration defines the environment inside which the application will be executed by specifying the base image, the working directory, and the execution instructions.

=> In the current solution, the Dockerfile was developed on a Linux terminal editor. Instructions included in this Dockerfile are as follows:

→ FROM python

=> The FROM instruction specifies the base image which will be used to execute the code. In the current example, the official Python base image was specified. This means that a base image which has the Python runtime available will be used to execute the Python script.

→ WORKDIR /app

=> The WORKDIR instruction defines the working directory inside the Docker container. By using this instruction, we set the working directory of the container to /app.

→ COPY . /app

=> By specifying the COPY instruction, we copy all files from the current directory into the defined working directory inside the container.

→ CMD ["python", "hello.py"]



=> Once it had been verified that the container works properly, the subsequent step included getting ready to distribute the Docker image by pushing it on DockerHub.

=> However, at first there was a problem with authentication when trying to push the Docker image. It was fixed by logging into Docker with the docker login command, which verifies the identity of a Docker user with DockerHub.

=> Then the Docker image was tagged according to the following command:

**docker tag vinay/mysamplehello vinaybhikhabhaipatel/mysamplehello**

=> This procedure should take place since tagging provides for associating images with repositories.

=> At last, it was time to push the Docker image to DockerHub:

**docker push vinaybhikhabhaipatel/mysamplehello**

=> It is possible to see from the results that all layers of the Docker image have been successfully uploaded. It should be noted that this example shows how easy and efficient it is to share containerized applications.

## 5.6 Implementation of Data Persistence using Docker Volumes

```
myvolume
vinay@vinaya:~$ docker volume ls
DRIVER      VOLUME NAME
local       myvolume
vinay@vinaya:~$ docker run -it --name testvol -v myvolume:/data ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
689b91d88a0f: Pull complete
b22f29e93d86: Download complete
Digest: sha256:84e77dee7d1bc93fb029a45a3c6cb9d8aa4831ccfcc7103d36e876938d28895b
Status: Downloaded newer image for ubuntu:latest
root@8b2524bd7836:/# echo "docker volume works">
bash: syntax error near unexpected token 'newline'
root@8b2524bd7836:/# echo "docker volume works" > /data/test.txt
root@8b2524bd7836:/# docker volume works
bash: docker: command not found
root@8b2524bd7836:/# cat /data/test.txt
docker volume works
root@8b2524bd7836:/#
```

```
vinay@vinaya:~$ docker volume create myvolume
docker: unknown command: docker volume create

Usage:  docker volume COMMAND

Run 'docker volume --help' for more information
vinay@vinaya:~$ docker volume ls
DRIVER      VOLUME NAME
local       myvolume
vinay@vinaya:~$ docker volume create myvolume
myvolume
vinay@vinaya:~$ docker volume ls
DRIVER      VOLUME NAME
local       myvolume
vinay@vinaya:~$
```

=> By default, data stored in the Docker containers is stateless, i.e., it gets erased after removing the container. This limitation can be circumvented using volumes in Docker which store the data externally from the life cycle of containers.

=> In this case, a volume was implemented with the help of the following command:

**docker volume create myvolume**

=> At first, the wrong command was provided which led to failure in creation of a new volume. The right command was used later to implement the required solution.

=> After creating the volume, it was tested with the command below:

**docker volume ls**

=> This indicated that the volume myvolume had been successfully implemented and could now be used.

=> Further, a container with attached volume was started by providing the following command:

**docker run -it --name testvol -v myvolume:/data ubuntu bash**

=> This command attaches a volume myvolume to the folder/data of the container. It means that all

data stored in that directory will be saved in the volume.

=> Finally, in order to check whether data is persisted in the volume or not, the following commands were provided:

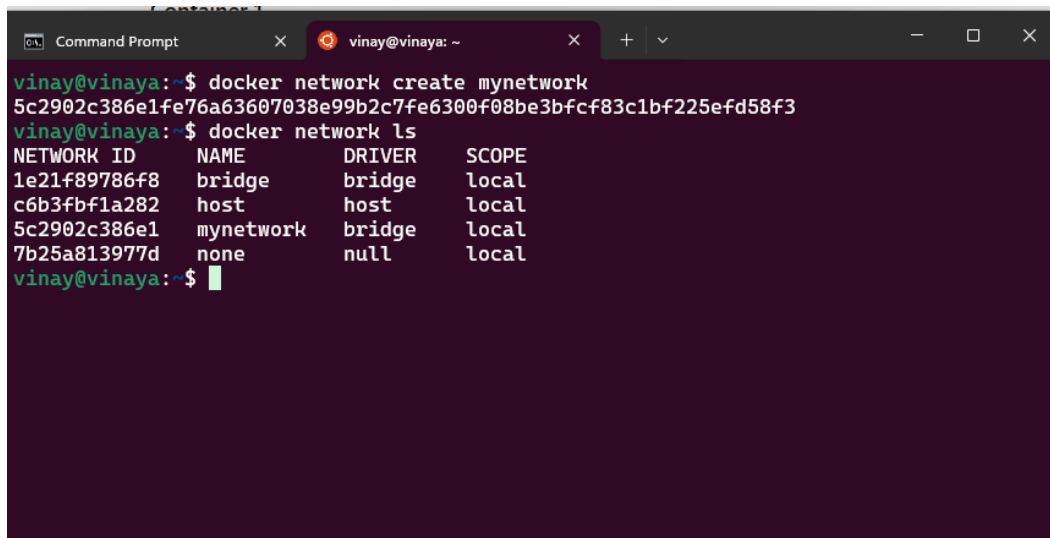
```
echo "docker volume works" > /data/test.txt
```

```
cat /data/test.txt
```

=> As evident from the above output, the data is successfully saved in the volume. This shows that data persistence is made possible through Docker volumes. The ability to achieve data persistence in Docker volumes is important since it enables the storage of data in applications like databases, logging services, and configurations.

## 5.7 Docker Networking Implementation

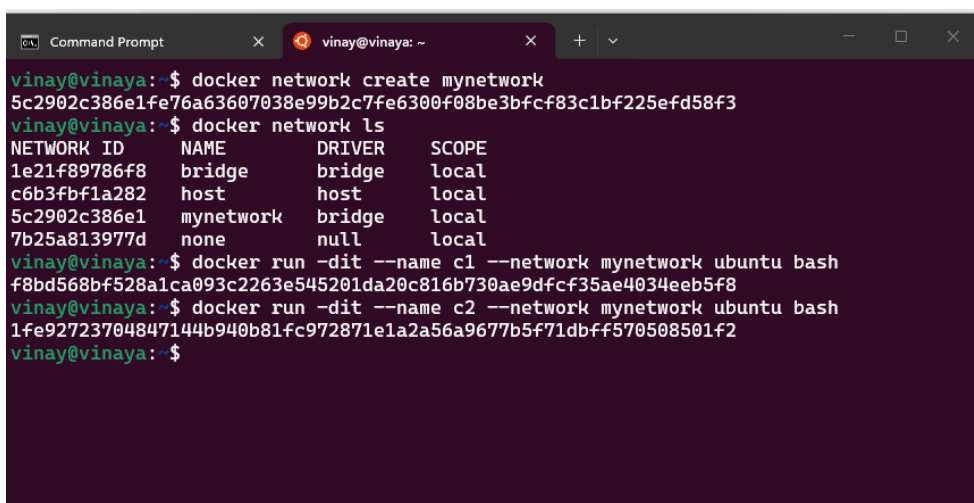
### 5.7.1 Creation and Verification of Docker Network

A terminal window titled 'vinay@vinaya: ~' showing the execution of Docker network commands. The user runs 'docker network create mynetwork', which returns a long ID. Then, the user runs 'docker network ls', which displays a table of existing networks.

```
vinay@vinaya:~$ docker network create mynetwork
5c2902c386e1fe76a63607038e99b2c7fe6300f08be3bfcf83c1bf225efd58f3
vinay@vinaya:~$ docker network ls
NETWORK ID          NAME       DRIVER  SCOPE
1e21f89786f8        bridge    bridge  local
c6b3fbf1a282        host      host    local
5c2902c386e1        mynetwork bridge   local
7b25a813977d        none     null    local
vinay@vinaya:~$
```

=> To create the Docker network known as mynetwork, the docker network create command was used. This ensures that the container communicates with other containers. Docker network ls was run to check the status of the network created, which showed the creation of mynetwork as well as other default networks such as bridge and host. It is important to create a custom network since it enables the containers to effectively communicate.

### 5.7.2 Connecting Containers to Docker Network

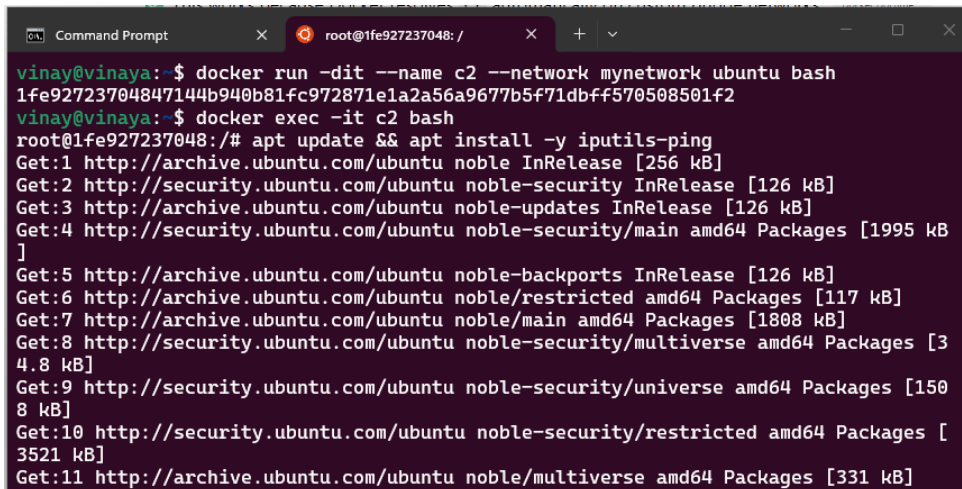
A terminal window titled 'vinay@vinaya: ~' showing the execution of Docker commands to create and connect containers to the custom network. The user runs 'docker network create mynetwork' (output shown in previous block), then 'docker network ls' (output shown in previous block). Then, the user runs 'docker run -dit --name c1 --network mynetwork ubuntu bash', which returns a long ID. Finally, the user runs 'docker run -dit --name c2 --network mynetwork ubuntu bash', which also returns a long ID.

```
vinay@vinaya:~$ docker network create mynetwork
5c2902c386e1fe76a63607038e99b2c7fe6300f08be3bfcf83c1bf225efd58f3
vinay@vinaya:~$ docker network ls
NETWORK ID          NAME       DRIVER  SCOPE
1e21f89786f8        bridge    bridge  local
c6b3fbf1a282        host      host    local
5c2902c386e1        mynetwork bridge   local
7b25a813977d        none     null    local
vinay@vinaya:~$ docker run -dit --name c1 --network mynetwork ubuntu bash
f8bd568bf528a1ca093c2263e545201da20c816b730ae9dfcf35ae4034eeb5f8
vinay@vinaya:~$ docker run -dit --name c2 --network mynetwork ubuntu bash
1fe92723704847144b940b81fc972871e1a2a56a9677b5f71dbff570508501f2
vinay@vinaya:~$
```

=> Two containers (c1 and c2) have been established and connected to my custom network, mynetwork, by use of the docker run command. This way, communication between the two containers be-

comes possible. It is better to make use of custom networks in order to enhance communication among containers.

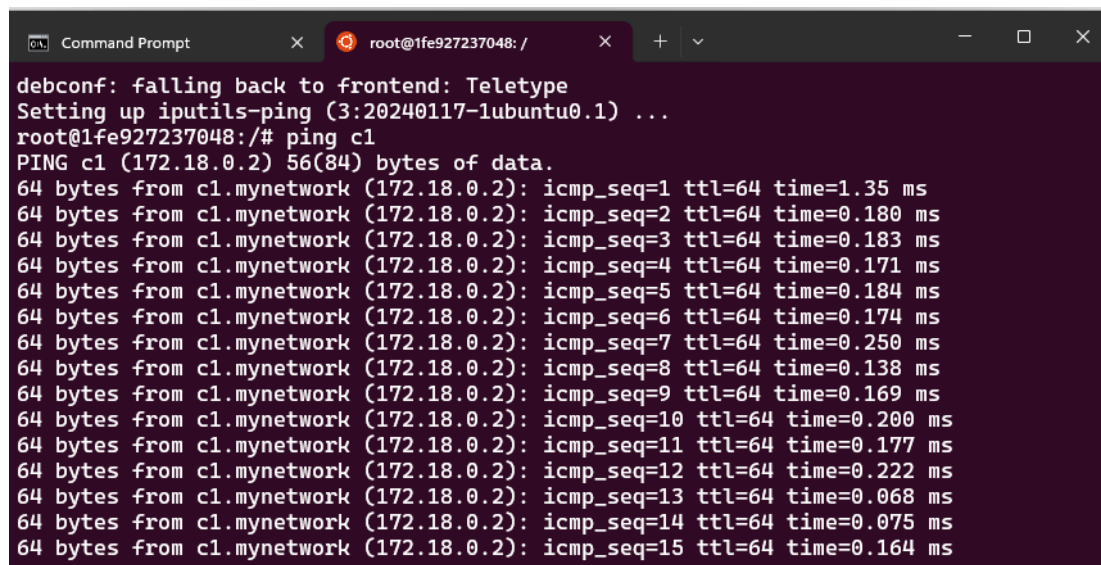
### 5.7.3 Installing Network Tools in Container



```
Command Prompt x root@1fe927237048: / x + v
vinay@vinaya:~$ docker run -dit --name c2 --network mynetwork ubuntu bash
1fe92723704847144b940b81fc972871e1a2a56a9677b5f71dbff570508501f2
vinay@vinaya:~$ docker exec -it c2 bash
root@1fe927237048:/# apt update && apt install -y iputils-ping
Get:1 http://archive.ubuntu.com/ubuntu noble InRelease [256 kB]
Get:2 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:3 http://archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:4 http://security.ubuntu.com/ubuntu noble-security/main amd64 Packages [1995 kB]
Get:5 http://archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:6 http://archive.ubuntu.com/ubuntu noble/restricted amd64 Packages [117 kB]
Get:7 http://archive.ubuntu.com/ubuntu noble/main amd64 Packages [1808 kB]
Get:8 http://security.ubuntu.com/ubuntu noble-security/multiverse amd64 Packages [34.8 kB]
Get:9 http://security.ubuntu.com/ubuntu noble-security/universe amd64 Packages [1508 kB]
Get:10 http://security.ubuntu.com/ubuntu noble-security/restricted amd64 Packages [3521 kB]
Get:11 http://archive.ubuntu.com/ubuntu noble/multiverse amd64 Packages [331 kB]
```

=> The Ip-utils-ping was installed in the container using the apt install command. This tool must be used to confirm that the containers can communicate with each other. With this software, we can confirm connectivity by pinging from one container to another.

### 5.7.4 Testing Container Communication



```
Command Prompt x root@1fe927237048: / x + v
debconf: falling back to frontend: Teletype
Setting up iputils-ping (3:20240117-1ubuntu0.1) ...
root@1fe927237048:/# ping c1
PING c1 (172.18.0.2) 56(84) bytes of data.
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=1 ttl=64 time=1.35 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=2 ttl=64 time=0.180 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=3 ttl=64 time=0.183 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=4 ttl=64 time=0.171 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=5 ttl=64 time=0.184 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=6 ttl=64 time=0.174 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=7 ttl=64 time=0.250 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=8 ttl=64 time=0.138 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=9 ttl=64 time=0.169 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=10 ttl=64 time=0.200 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=11 ttl=64 time=0.177 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=12 ttl=64 time=0.222 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=13 ttl=64 time=0.068 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=14 ttl=64 time=0.075 ms
64 bytes from c1.mynetwork (172.18.0.2): icmp_seq=15 ttl=64 time=0.164 ms
```

=> The ping command was executed to check whether communication is possible between two containers on the Docker network. The ping command worked between the two containers, which indicates that both containers have established connectivity. This confirms that Docker networking is properly functioning.

### 5.7.5 Inspecting Docker Network Configuration

=> To get a more comprehensive description of the network, we can use the docker network inspect mynetwork command to view the network description. It will show us the description of our network configuration and the attached containers to the network.

```
vinay@vinaya:~$ docker network inspect mynetwork
[
  {
    "Name": "mynetwork",
    "Id": "5c2902c386e1fe76a63607038e99b2c7fe630f08be3bfcf83c1bf225efd58f3",
    "Created": "2026-04-13T15:40:59.333303655Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv4": true,
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Options": {
      "com.docker.network.enable_ipv4": "true",
      "com.docker.network.enable_ipv6": "false"
    },
    "Labels": {},
    "Containers": {
      "1fe92723704847144b940b81fc972871e1a2a56a9677b5f71dbff570508501f2": {

```

## 5.7.6 Verifying Running Containers

```
    },
    "Status": {
      "IPAM": {
        "Subnets": {
          "172.18.0.0/16": {
            "IPsInUse": 5,
            "DynamicIPsAvailable": 65531
          }
        }
      }
    }
  }
}
vinay@vinaya:~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
1fe927237048   ubuntu   "bash"    9 minutes ago    Up 9 minutes           c2
f8bd568bf528   ubuntu   "bash"    10 minutes ago   Up 10 minutes          c1
vinay@vinaya:~$
```

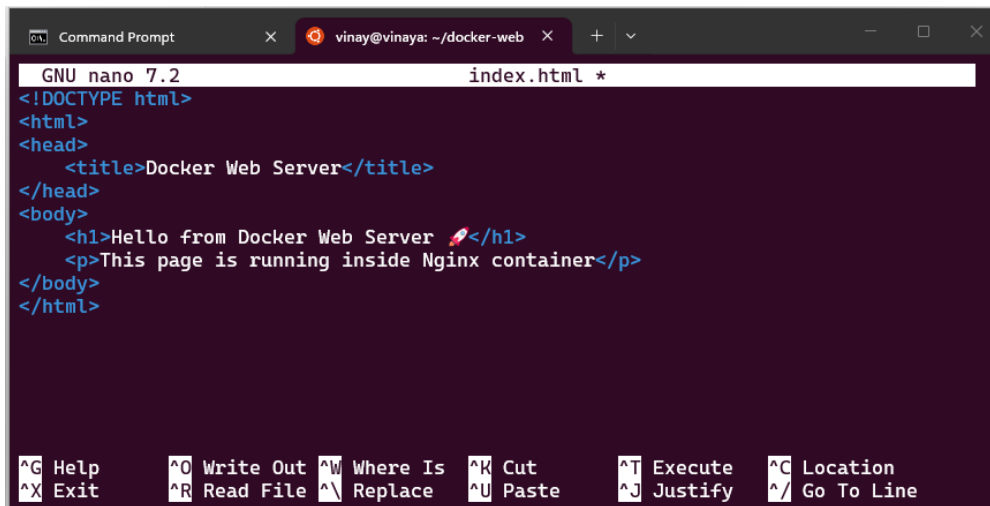
=> The command `docker ps` was executed to check whether the containers (c1 and c2) are functioning properly or not. This ensures that both the containers are active and connected to the network.

## 5.8 Implementation of Docker-Based Web Server

### 5.8.1 HTML Page Creation for Web Server

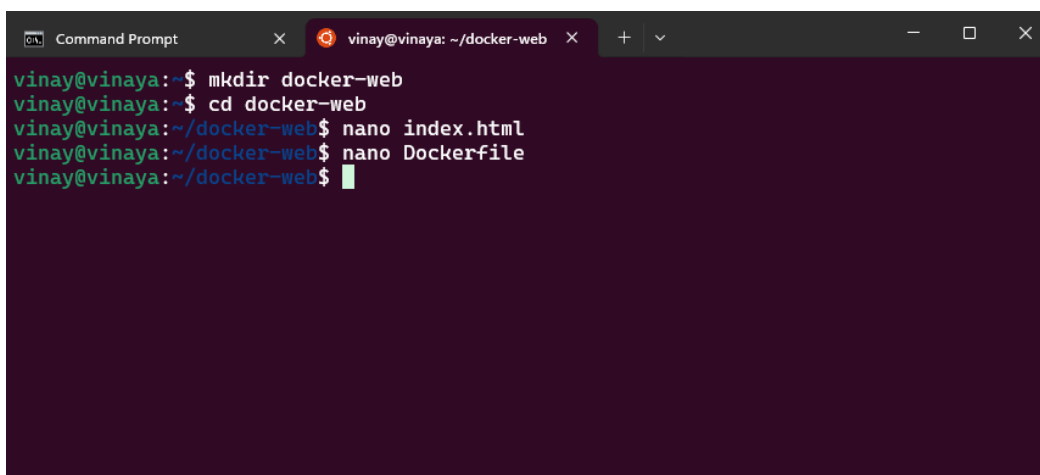
=> An `index.html` document was generated using HTML to present information on the web server. The document includes simple texts which will be displayed when the Nginx container is running.

=> The document serves as the homepage of the web server.



```
GNU nano 7.2 index.html *
<!DOCTYPE html>
<html>
<head>
  <title>Docker Web Server</title>
</head>
<body>
  <h1>Hello from Docker Web Server 🚀</h1>
  <p>This page is running inside Nginx container</p>
</body>
</html>
```

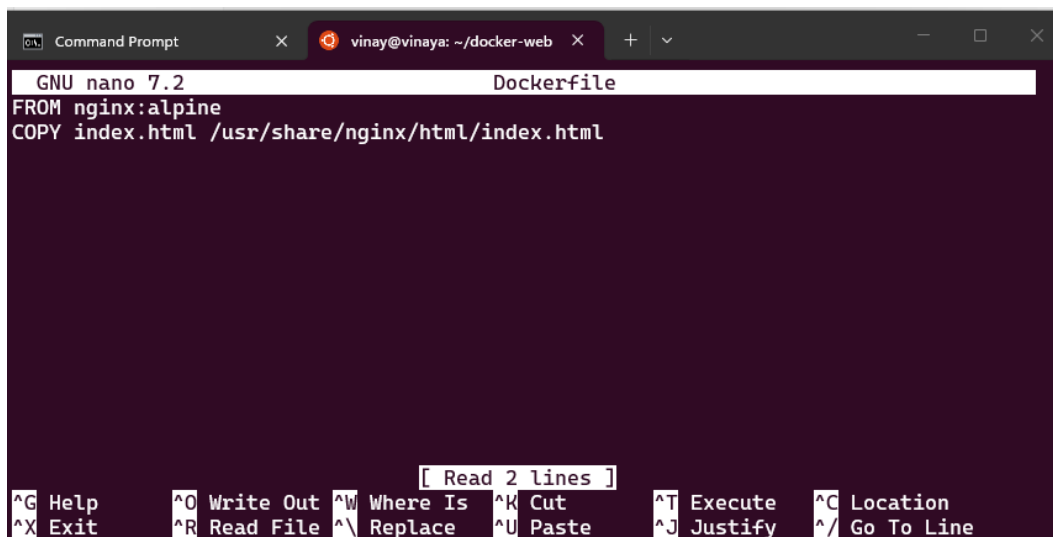
## 5.8.2 Creating Project Directory and Files



```
vinay@vinaya:~$ mkdir docker-web
vinay@vinaya:~$ cd docker-web
vinay@vinaya:~/docker-web$ nano index.html
vinay@vinaya:~/docker-web$ nano Dockerfile
vinay@vinaya:~/docker-web$
```

=> The creation of a new directory called docker-web was done to store all files related to the web application. In this directory, the HTML file and the Dockerfile were both created using the nano editor. This ensures all project files are stored in a single place for creating the Docker image.

## 5.8.3 Creating Dockerfile for Nginx Web Server



```
GNU nano 7.2 Dockerfile
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
```

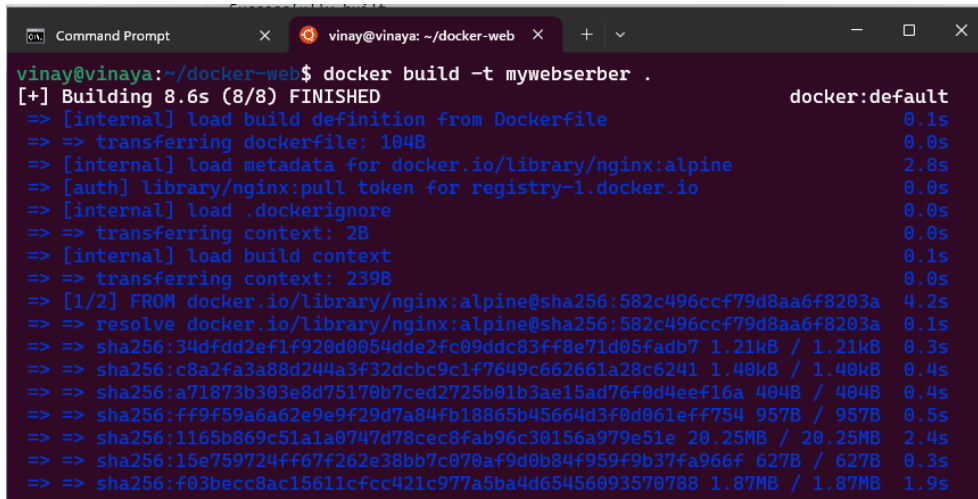
=> The following Dockerfile is used for creating a container that will act as a web server based on the Nginx image.

FROM nginx:alpine

COPY index.html /usr/share/nginx/html

=> With this configuration, the container will be able to host the webpage inside its containerized environment.

#### 5.8.4 Building Docker Image for Web Server

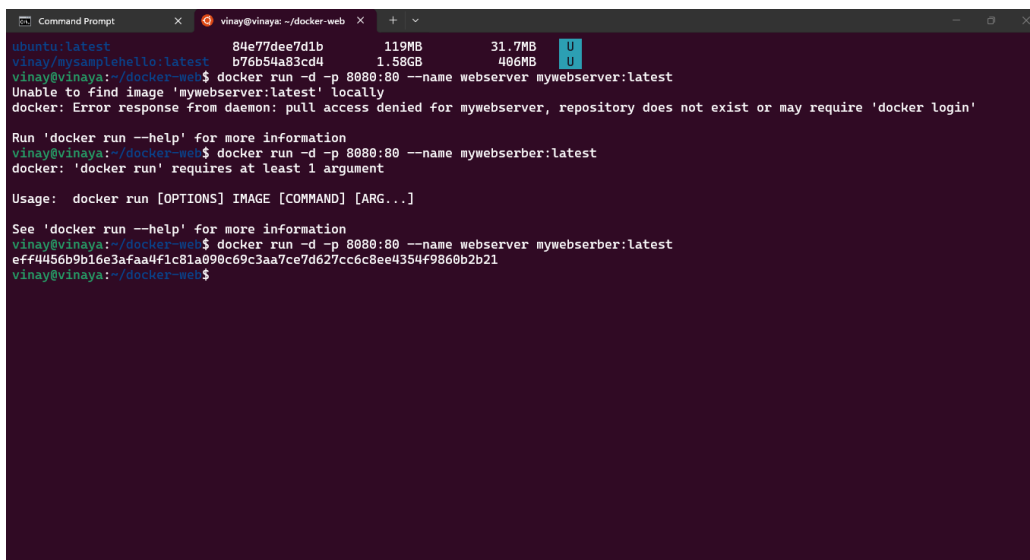


```
vinay@vinaya: ~/docker-web$ docker build -t mywebserver .
[+] Building 8.6s (8/8) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile              0.1s
=> => transferring dockerfile: 104B                             0.0s
=> [internal] load metadata for docker.io/library/nginx:alpine  2.8s
=> [auth] library/nginx:pull token for registry-1.docker.io     0.0s
=> [internal] load .dockerignore                                0.0s
=> => transferring context: 2B                                    0.0s
=> [internal] load build context                                0.1s
=> => transferring context: 239B                                  0.0s
=> [1/2] FROM docker.io/library/nginx:alpine@sha256:582c496ccf79d8aa6f8203a 4.2s
=> => resolve docker.io/library/nginx:alpine@sha256:582c496ccf79d8aa6f8203a 0.1s
=> => sha256:34dfdd2ef1f920d0054dde2fc09ddc83ff8e71d05fadb7 1.21kB / 1.21kB 0.3s
=> => sha256:c8a2fa3a88d244a3f32dcbc9c1f7649c662661a28c6241 1.40kB / 1.40kB 0.4s
=> => sha256:a71873b303e8d75170b7ced2725b01b3ae15ad76f0d4eef16a 404B / 404B 0.4s
=> => sha256:ff9f59a6a62e9e9f29d7a84fb18865b45664d3f0d061eff754 957B / 957B 0.5s
=> => sha256:1165b869c51a1a0747d78cec8fab96c30156a979e51e 20.25MB / 20.25MB 2.4s
=> => sha256:15e759724ff67f262e38bb7c070af9d0b84f959f9b37fa966f 627B / 627B 0.3s
=> => sha256:f03becc8ac15611cfcc421c977a5ba4d65456093570788 1.87MB / 1.87MB 1.9s
```

=> This Docker image was created by running the command docker build. The docker build command reads Dockerfile, downloads the base image for Nginx, and builds a Docker image with the HTML file added to it.

=> Successful creation of this Docker image indicates that the web server is ready to be used.

#### 5.8.5 Running the Web Server Container



```
vinay@vinaya: ~/docker-web$ docker run -d -p 8080:80 --name webserver mywebserver:latest
Unable to find image 'mywebserver:latest' locally
docker: Error response from daemon: pull access denied for mywebserver, repository does not exist or may require 'docker login'

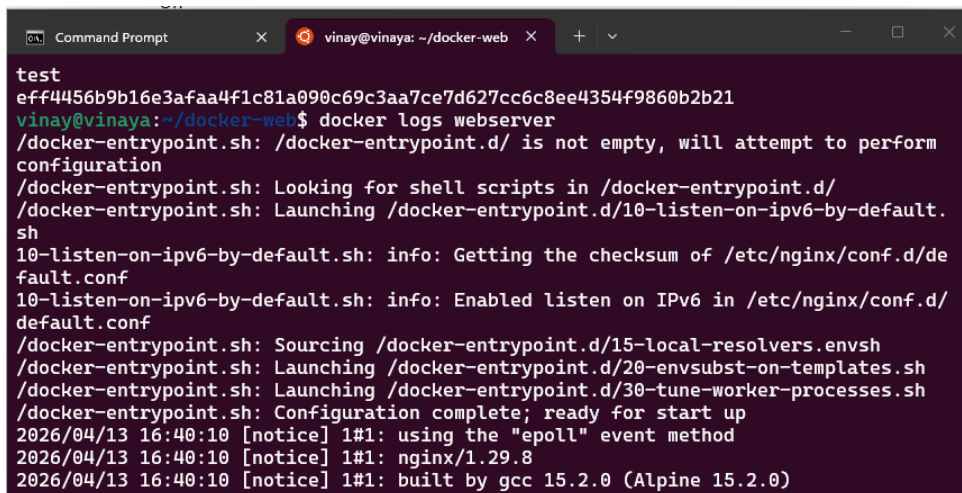
Run 'docker run --help' for more information
vinay@vinaya: ~/docker-web$ docker run -d -p 8080:80 --name mywebserver:latest
docker: 'docker run' requires at least 1 argument

Usage: docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

See 'docker run --help' for more information
vinay@vinaya: ~/docker-web$ docker run -d -p 8080:80 --name webserver mywebserver:latest
eff4456b9b16e3afaa4f1c81a090c69c3aa7ce7d627cc6c8ee4354f9860b2b21
vinay@vinaya: ~/docker-web$
```

=> The container for Docker was started by making use of the docker run command with port mapping (8080:80). There were initial issues with command execution, which were later resolved, allowing for the successful execution of the command. It indicates that the Nginx web server is running.

## 5.8.6 Verifying Web Server Execution Using Logs

A screenshot of a terminal window with a dark background. The window title bar shows 'Command Prompt' and 'vinay@vinaya: ~/docker-web'. The terminal output shows the command 'docker logs webserver' and its output, which includes messages from the Docker entrypoint script and Nginx logs. The output confirms that the Nginx server started successfully without any errors.

```
test
eff4456b9b16e3afaa4f1c81a090c69c3aa7ce7d627cc6c8ee4354f9860b2b21
vinay@vinaya:~/docker-web$ docker logs webserver
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.
sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/de
fault.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/
default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2026/04/13 16:40:10 [notice] 1#1: using the "epoll" event method
2026/04/13 16:40:10 [notice] 1#1: nginx/1.29.8
2026/04/13 16:40:10 [notice] 1#1: built by gcc 15.2.0 (Alpine 15.2.0)
```

=> To determine whether the Nginx server runs successfully within the container, I used the following command:

### **docker logs**

=> The output confirms that the Nginx server started successfully without any errors.

=> Therefore, the web server is working well.

## 5.9 Practical Conclusion

=> This practical implementation proved successful by illustrating how the containerization can be achieved through the use of Docker. From the results, one can conclude that containerized applications are lightweight and flexible and therefore ideal in cloud computing. This means that their use is highly recommended in cloud computing, especially in terms of efficiency and flexibility.

## 6. Cloud Computing Terminologies

=> Cloud computing makes use of flexible resources available on the Internet. Load balancing, scalability, and application gateways are some of the key terminologies associated with cloud computing that ensure effective functioning of the system.

### 6.1 Load Balancing:

#### 6.1.1 Definition and Significance:

=> Load balancing refers to the allocation of network traffic among various servers in order to prevent the overburdening of any one server. This helps improve system efficiency and availability.

Load balancing is significant since:

- It prevents server overburdening.
- It enhances the performance of applications.
- It guarantees high availability.
- It offers fault tolerance.

=> Load balancing is highly common in cloud computing environments where there are many users and requests to be processed.

#### 6.2.2 Implementation of Load Balancing in AWS

=> Load balancing services such as the Elastic Load Balancer (ELB) are used in AWS to automatically distribute traffic among EC2 instances.

=> Types of load balancers in AWS include:

- Application Load Balancer (ALB): This performs load balancing on application layer protocols such as HTTP/HTTPS.
- Network Load Balancer (NLB): Performs load balancing on transport layer protocols such as TCP.
- Classic Load Balancer: An old generation.

## **6.2 Scalability:**

### **6.2.1 Vertical or Horizontal Scaling:**

=> This means that scalability involves the capacity of a system to deal with rising workload by increasing resources.

=> There are two methods:

#### **A).Vertical Scaling**

- Involves adding resources (CPU, memory) to one computer only
- Easy to implement but limited
- An example is upgrading your server

#### **B).Horizontal Scaling**

- Involves adding more computers/instances
- Flexible and reliable method
- Used extensively in cloud computing

=>Horizontal scaling is favored in cloud-based architectures because of the flexibility it provides in handling large systems.

### **6.2.2 AWS Auto Scaling**

=> The AWS Auto Scaling service helps to increase and decrease the number of EC2 instances according to demand.

=> Some of its capabilities include:

- Increasing the number of instances during heavy traffic
- Decreasing the number of instances during less traffic
- Decreasing cost and increasing efficiency

## **6.3 Application Gateways:**

### **6.3.1 Role and Functionality:**

=> The application gateway acts as a bridge between the user and the back-end services. It is responsible for managing requests made to it and routing the request to the correct back-end service.

=> Functions include:

- Routing of requests
- Security (authenticating, filtering)
- Load balancing
- Traffic management

### **6.3.2 AWS API Gateway / ALB**

=> In AWS, the functions of an application gateway are fulfilled by:

→API Gateway

- Used for managing APIs
- Responsible for routing requests, authentication, and monitoring
- Frequently employed in serverless architectures

→Application Load Balancer (ALB)

- Routes HTTP/HTTPS traffic
- Offers support for path-based and host-based routing
- Ideal for web applications and microservices

## **7. Integration and Evaluation**

=> The cloud computing model depends on various elements working together in order to offer efficient and reliable services. Scalability, load balancing, and application gateway are very similar and enhance the performance of the whole system.

### **7.1 Relationship Between Load Balancing and Scalability**

=> Load Balancing and Scalability are interlinked in the context of cloud computing. While load balancing refers to the distribution of traffic to various servers, scalability involves making sure that the system can cope up with more loads by either increasing or reducing resources accordingly.

=> In case there is an increase in traffic, there will be a corresponding increase in the number of servers through horizontal scalability, which is then balanced by load balancing techniques.

### **7.2 Why They Are Complementary**

=> Load balancing and scalability go hand-in-hand since they both contribute toward ensuring system stability and performance.

- Load balancing handles traffic management
- Scalability handles resource management

=> In the absence of load balancing, new server additions might go unused. In the absence of scalability, load balancing will fail to deal with the increasing load. Thus, both are necessary for creating a resilient cloud environment.

### **7.3 Benefits of Combining Load Balancing, Scaling, and Gateways**

=> These three technologies together offer many benefits such as:

- Increased performance due to traffic management
- Higher availability because of system protection from failure
- Efficient resource usage
- Handling large amounts of data efficiently
- Increased flexibility in the cloud

=> The application gateway offers additional functionality by properly managing requests and APIs.

### **7.4 Effects on Performance, Security, and Reliability**

=> The incorporation of load balancing, scalability, and gateways plays a crucial role in cloud systems:

A).Performance:

=>Traffic management and scalability guarantee speedy responses and an efficient user interface experience.

B).Security:

=> Security measures such as authentication and filtering are offered through gateways.

C).Reliability:

=> The system's reliability can be enhanced by distributing tasks among multiple servers and reducing the risk of having a point of failure.

## **8. Driving Forces of Cloud Computing**

=> There are a number of reasons for the popularity of cloud computing, which has been adopted because of its advantages over traditional IT architecture. Some of the drivers for cloud computing include cost-effectiveness, flexibility, innovation, and industry adoption.

### **8.1 Cost Efficiency**

=> Cost savings form one of the most compelling arguments for switching to cloud computing technology. A conventional system requires a large amount of money for installation, setup, and maintenance costs. However, with cloud computing, the process works differently, since people pay according to usage.

This reduces:

- Initial capital investment
- Maintenance costs
- Operational expenses

### **8.2 Flexibility and Elasticity**

=>The cloud provides users with high levels of flexibility and elasticity, enabling them to increase or decrease resource usage according to their needs.

- Flexibility means that users can access cloud services from anywhere.
- Elasticity means that resource usage can automatically be adjusted.

=> For instance, when there is a lot of traffic, more servers can be utilized, but when there is low traffic, fewer resources are used.

### **8.3 Innovation and Managed Services**

=> Cloud platforms offer access to a vast array of innovative tools and managed services that facilitate rapid innovation.

Some examples are:

- AI services
- Machine learning
- Database management
- Serverless computing

=> Managed services minimize the burden of managing systems manually, giving developers ample time to develop and enhance applications.

### **8.4 Industry Adoption Trends**

=> Cloud computing has gained widespread adoption in many industries including the health sector, banking institutions, educational institutions, and eCommerce.

=>The reason why companies are adopting the cloud is because of:

- Flexibility and scalability
- Economic benefits
- Better collaboration
- Rapid deployment of applications

=> The trend indicates continued adoption of cloud technology as firms strive to be more digital and competitive in today's market environment.

## **9. Serverless vs Containerisation**

=> The development of cloud technologies brought up several ways of deploying applications: one of them was a serverless computing and another – containerization. Both of them have advantages in terms of scalability and flexibility, but there are still some differences between approaches.

### **9.1 Introduction to Serverless Computing (FaaS)**

=> The approach that is known as Function-as-a-Service (or FaaS) represents the ability of a developer to execute code regardless of having to manage infrastructure. All of the mentioned processes are done by the cloud provider automatically in case of serverless computing.

=> Moreover, the developer needs only to write the code, while the application will be executed upon receiving certain signals or events from the external sources.

### **9.2 AWS Lambda / Azure Functions**

=> The platforms that allow using serverless technologies include:

#### **A).AWS Lambda**

=> AWS Lambda enables developers to execute their code when receiving certain events, for instance, sending an HTTP request. The service automatically scales depending on demand, and the billing is done based on execution time.

#### **B).Azure Functions**

=> It also allows running code when specific events take place. Besides, the platform provides good integration with other products of the Microsoft company.

### **8.3 Comparison with Containerisation**

=> There are some differences between serverless and containerisation:

#### **A).Infrastructure Management:**

=> Serverless needs no server management but the latter needs setup and orchestration.

#### **B).Scalability:**

=> Serverless offers auto-scaling capabilities but containers need to be scaled manually or by tools such as Kubernetes.

#### **C).Execution Model:**

=> Serverless works with events while containers offer a service mode of execution.

#### **D).Control:**

=>More control can be exerted in containers than in serverless.

=>Both serverless and containerisation have their importance based on application needs.

### **9.4 Advantages and Limitations**

- Advantages of Serverless Computing:
- No Infrastructure Management
- Auto-Scaling
- Efficient Cost: Pay per Use
- Faster Deployment

## 9.5 Use Cases Table: Serverless vs Containerisation

| Category           | Serverless (FaaS)                            | Containerisation                                      |
|--------------------|----------------------------------------------|-------------------------------------------------------|
| Application Type   | Event-driven applications                    | Web applications                                      |
| Architecture Style | APIs and lightweight microservices           | Microservices architecture                            |
| Processing Tasks   | Data processing and real-time event handling | Long-running and complex applications                 |
| Deployment Model   | Trigger-based execution                      | Continuous deployment pipelines                       |
| Scalability        | Automatic scaling based on events            | Scalable using orchestration tools (e.g., Kubernetes) |
| Execution Time     | Short-lived functions                        | Long-running services                                 |
| Management Effort  | Minimal (fully managed by cloud provider)    | Requires management and configuration                 |

## 10. Conclusion

=> In conclusion, the current report is based on a detailed analysis of containerisation and its use in the context of cloud computing. Initially, the paper presented the basic principles underlying virtualisation and containerisation and discussed the benefits of the latter approach compared to using conventional virtual machines.

=> In addition, Docker technology as a way to implement containerisation was discussed. This part included not only theoretical but also practical aspects of containerisation by creating images, containers, setting up networking and volume management, as well as deploying containers in Docker-Hub. All these steps proved the viability of solutions involving containerisation.

=> Moreover, the following paper examined such fundamental components of cloud computing as load balancing, scalability, and application gateways. Their significance and interdependence for building effective solutions was evaluated.

## 11. IEEE References

- [1] Docker Inc., "Docker Documentation," Docker, 2024. [Online]. Available: <https://docs.docker.com>. [Accessed: 22 April 2026].
- [2] Amazon Web Services, "AWS Documentation," AWS, 2024. [Online]. Available: <https://docs.aws.amazon.com>. [Accessed: 22 April 2026].
- [3] Microsoft, "Azure Functions Documentation," Microsoft Learn, 2024. [Online]. Available: <https://learn.microsoft.com>. [Accessed: 22 April 2026].
- [4] Kubernetes, "Kubernetes Documentation," Kubernetes, 2024. [Online]. Available: <https://kubernetes.io/docs>. [Accessed: 22 April 2026].
- [5] VMware Inc., "Virtualization Overview," VMware, 2023. [Online]. Available: <https://www.vmware.com>. [Accessed: 22 April 2026].
- [6] Red Hat, "What is Containerization?," Red Hat, 2023. [Online]. Available: <https://www.red-hat.com>. [Accessed: 22 April 2026].
- [7] IBM Cloud, "Cloud Computing Overview," IBM, 2023. [Online]. Available: <https://www.ibm.com/cloud>.

[tps://www.ibm.com/cloud](https://www.ibm.com/cloud). [Accessed: 22 April 2026].

[8] Google Cloud, “Cloud Architecture Framework,” Google, 2023. [Online]. Available: <https://cloud.google.com>. [Accessed: 22 April 2026].

[9] NGINX Inc., “NGINX Documentation,” NGINX, 2024. [Online]. Available: <https://nginx.org/en/docs/>. [Accessed: 22 April 2026].

[10] Cisco Systems, “Load Balancing Overview,” Cisco, 2023. [Online]. Available: <https://www.cisco.com>. [Accessed: 22 April 2026].

## ⇒ **Other References**

[11] Oracle Corporation, “What is Cloud Computing?,” Oracle Cloud, 2023. [Online]. Available: <https://www.oracle.com/cloud/what-is-cloud-computing/>. [Accessed: 22 April 2026].

[12] DigitalOcean, “How Containers Work: An Introduction,” DigitalOcean, 2023. [Online]. Available: <https://www.digitalocean.com/resources>. [Accessed: 22 April 2026].

[13] Cloud Native Computing Foundation (CNCF), “Cloud Native Definition,” CNCF, 2023. [Online]. Available: <https://www.cncf.io>. [Accessed: 22 April 2026].

[14] Microsoft Azure, “What is Virtualization?,” Microsoft, 2023. [Online]. Available: <https://azure.microsoft.com>. [Accessed: 22 April 2026].

[15] Google Cloud, “Introduction to Containers and Kubernetes,” Google, 2023. [Online]. Available: <https://cloud.google.com>. [Accessed: 22 April 2026].

---

---

## ⇒ **Video Presentation Link:**

[https://youtu.be/umM4EI8ixp4?si=k-7fALmi\\_79VGLiM](https://youtu.be/umM4EI8ixp4?si=k-7fALmi_79VGLiM)