



Cloud Engineering Principles and Practice

ACCA7027 Assessment Report

Student Name:
Habibullah Shaikh

Student ID:
2503799

Programme:
MSc Cybersecurity and Digital Forensics

Submission Date:
Thursday, 18th December 2025

Abstract

With an emphasis on virtualization and containerization technologies, this study offers a thorough analysis of cloud engineering principles. The paper examines the advantages and difficulties of virtualization in contemporary cloud systems, including full virtualization, paravirtualization, and hardware-assisted virtualization. Compared to conventional virtual machines, Docker-based solutions offer lightweight, effective, and portable application deployment, as demonstrated by a thorough examination of containerization concepts. This report details the entire process of creating Docker images, controlling container volumes and networking, and publishing to Docker Hub through real-world application. In order to build reliable, scalable cloud infrastructures, the research also looks at important cloud computing principles including load balancing, auto-scaling, and application gateways. Lastly, the study examines use cases, financial ramifications, and architectural trade-offs between serverless computing and containerization. The practical component shows how these technologies are used in the real world by demonstrating a full web server deployment using Docker. This article demonstrates how contemporary cloud technologies enable scalable, effective, and dependable application deployment by fusing theoretical knowledge with practical application.

Table of Contents

1. Introduction
2. Virtualization Principles
 - 2.1 Overview of Virtualization
 - 2.2 Types of Virtualization
 - 2.3 Virtualization of Shared Hardware
 - 2.4 Benefits and Challenges
3. Containerization Principles and Benefits
 - 3.1 Introduction to Containerization
 - 3.2 Containers vs Virtual Machines
 - 3.3 Docker Architecture
 - 3.4 Building Docker Images - Dockerfile Construction
 - 3.5 Benefits of Containerization
4. Docker Images and Publishing
 - 4.1 Understanding Docker Images
 - 4.2 Image Layers and Caching
 - 4.3 Publishing Images to DockerHub
 - 4.4 Practical Implementation
5. Container Volumes
 - 5.1 The Need for Data Persistence
 - 5.2 Types of Docker Volumes
 - 5.3 Creating and Managing Volumes
 - 5.4 Practical Implementation
6. Container Networking
 - 6.1 Docker Networking Concepts
 - 6.2 Network Drivers
 - 6.3 Container Communication
 - 6.4 Practical Implementation
7. Complete Integration Example - Web Server Deployment
 - 7.1 Project Overview
 - 7.2 Implementation Steps
 - 7.3 Results and Analysis
8. Cloud Computing Concepts
 - 8.1 Load Balancing
 - 8.2 Scalability
 - 8.3 Application Gateways
 - 8.4 Integration of All Three Concepts
9. Serverless Functions vs Containerization
 - 9.1 Driving Forces in Cloud Computing
 - 9.2 Serverless Functions Explained
 - 9.3 Comparison with Containerization
 - 9.4 Use Case Analysis
10. Video Demonstration
11. Conclusion
12. References

1. Introduction

Containerization and virtualization are very important technologies for today's software development and cloud computing. Where containerization packs the application and its dependencies in a portable and consistent environment, On the other hand virtualization runs multiple OS on a single machine. Combining both improves Scalability, DevOps processes and deployment efficiency. This report focuses on containerization and virtualization, Docker images, Docker network, Docker volume and practical execution. Furthermore, the report examines essential cloud computing fundamentals such as load balancing application gateway and scalability, summarizing with an assessment of serverless computing as an upcoming model. In this implementation phase we are going through the deployment and development of a containerized web application, showing the complete workflow from development to upload on Docker Hub.

2. The Principles of Virtualization

2.1 Overview of Virtualization

Virtualization is a foundational technology in cloud computing that enables the abstraction and sharing of hardware resources among multiple users or environments [1]. It allows multiple tenants to share physical resources while maintaining a level of security and segregation that is difficult to achieve with containerization alone. This technology facilitates resource pooling, scalability, and workload separation, which are essential properties of cloud computing environments. [1][2][3]

2.2 Types of Virtualization

In full virtualization, each operating system runs as a guest in a separate, isolated virtual machine (VM) on virtual hardware [4]. Windows, Linux, and vice versa run on the same machine, each in its own isolated VM. without any change, because hypervisor provide interface of original hardware [4]. Paravirtualization provides a virtual machine setup that look like a physical system. However changes need in the OS for high speed [5]. For memory updates it use verified hypervisor calls and with low hypervisor control it gives permission to OS to manage hardware page tables [5]. As per studies, there are 3 types of x86 virtualization: full, paravirtualization and hardware-assisted in which Xen is an important paravirtualization model [6]. Hardware-assisted virtualization improves the performance and resource efficiency of virtual systems by leveraging built-in processor features [7]. Technologies like SR-IOV and Intel VT reduce overheads of network I/O, but disk I/O remains slow [7]. Xen and KVM tests showed that cloud-based HPCs have faster task switching, direct I/O access, and improve memory handling. [8]. For best efficiency and less overhead, OS-level virtualization means containerization runs several separate containers on a common OS [9] [10]. Unlike the Paravirtualization or Full, it allocates resources like memory, storage, and network without requiring any resources. [9]

2.3 Virtualization of Shared Hardware

Overview:

Systems and multiple users can operate together, enabling independent and effective resource management, thanks to virtualization of shared hardware, which segment and divide physical computer resources. Traditional cloud platforms such as VMware, Hyper-V, and Comcast extend the capabilities of virtual machines by allocating power, memory, and I/O, With a less processing and head of the vertical layer. [11]

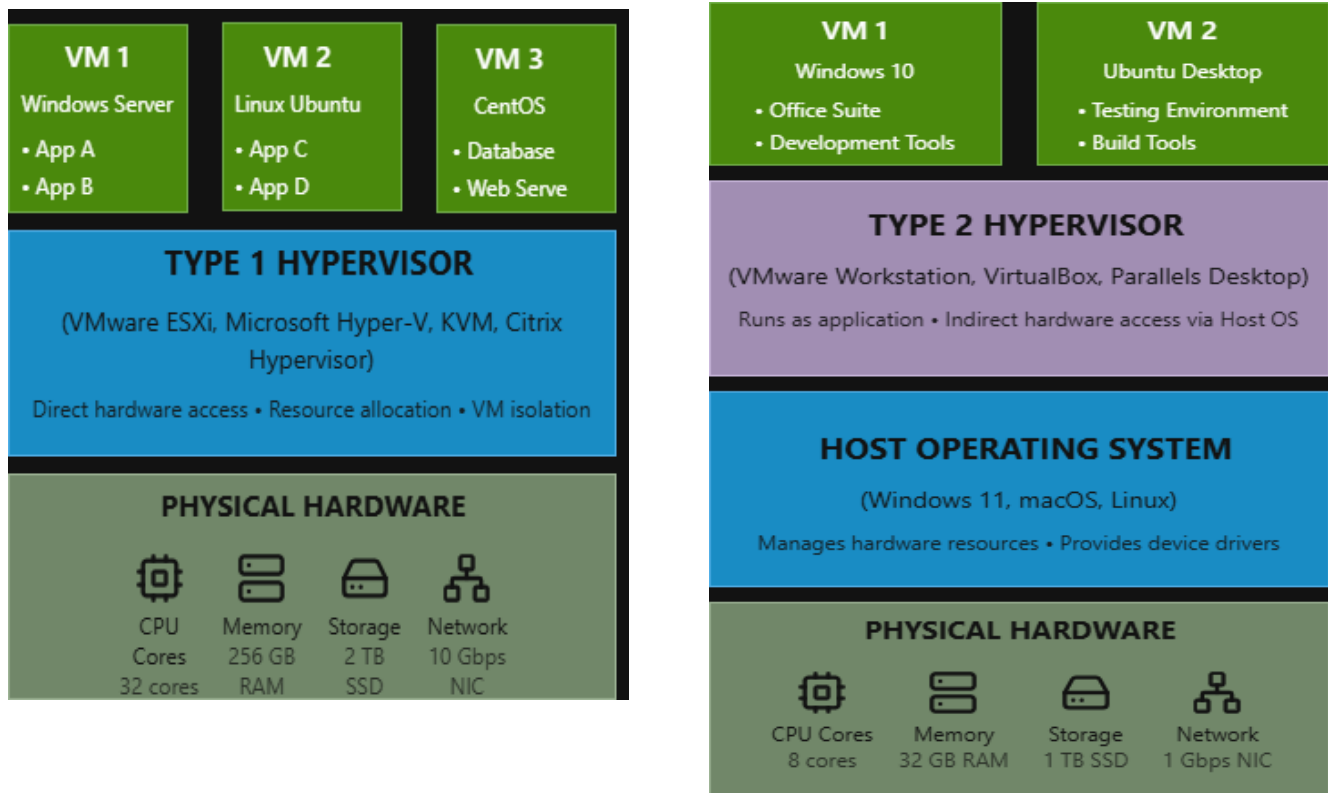
How Multiple VMs Share Physical Hardware:

Resources such as I/O, CPU, and RAM allocate devices, virtualization allow multiple VMs to to operate independently on a single host, Supporting physical hardware, providing strong security and restricted access. [12]

Type of Hypervisors:

There are different types of hypervisor, Type 1 (Bare Metal) and Type 2 (Hosted). Type 1 hypervisor operates directly on host hardware, control resources of host OS without host OS and provide superior VM isolation and security, But Mac incompatibility and 64-bit issues are still there [13] [14]. To improve security, the research team developed a bare-metal hypervisor that isolates corporate systems from consumer apps and hardware. Hosted hypervisors like VMware Player, Windows Virtual PC, and Parallels Desktop, operate on host OS and control hardware like a program [13] [14]. These provide an extra layer that can compromise security, but they are for setup and testing/development purposes. [13]. To fulfill performance requirements in the network, resource allocation communication, processing, and storage assets to segments, it divides into VNF's or slices; isolation ensures that one slice does not affect another. Thus the quality of service maintains in 5G network [16] [17] [18] [19]. The main goal of cloud engineering is cost effective resource utilization, flexibility, and cost reduction. SPACE4Cloud reduces 60% execution costs through VM mapping and architecture modeling [20]. In Microservices dynamic allocation, like Alibaba Cloud, increase the usage by 10-15% without latency compromise [21]. Simulation tools improve flexibility and performance by guiding best resource allocation and performance. [22]

Figure 1: Virtualization Architecture



2.4 Benefits and Challenges

Modern cloud design is based on virtualization, which is more useful in dependability and efficiency, through server consolidation, where multiple VMs or containers operate on a underused server, this promotes resource optimization and cost effectiveness [23][32]. Tools like Vshadow reduce migration time by over 90%, Improve speed by supporting multiple VMs on a single host [28] [30] [31]. Virtualization ensure isolation, like VMs operates independently and improves security [32]. and allow portability through which VMs switch easily between servers [33]. This flexibility converts fixed hardware into elastic resources, support scalable cloud services like IaaS and SaaS [26] [27] [34] Despite the benefits, there are also some drawbacks to virtual reality. Container-based solutions like LXC and Docker shows good performance, but hypervisor can create layer performance overhead, especially during peak loads or dynamic migrations [24] [29] [34]. Maintain and update Multiple VMs and guest OSs create management complexity, Which demands automation and skilled resource management [31], and implementing security in multi-tenant environments is even more challenging [32]. Lastly, virtualization helps in disaster recovery [25], But designing and maintaining robust virtualized systems requires considerable operational and architectural effort. All things considered, containerization offers a more effective, controllable, and scalable substitute for conventional virtualization techniques.

3. Containerization Principles and Benefits

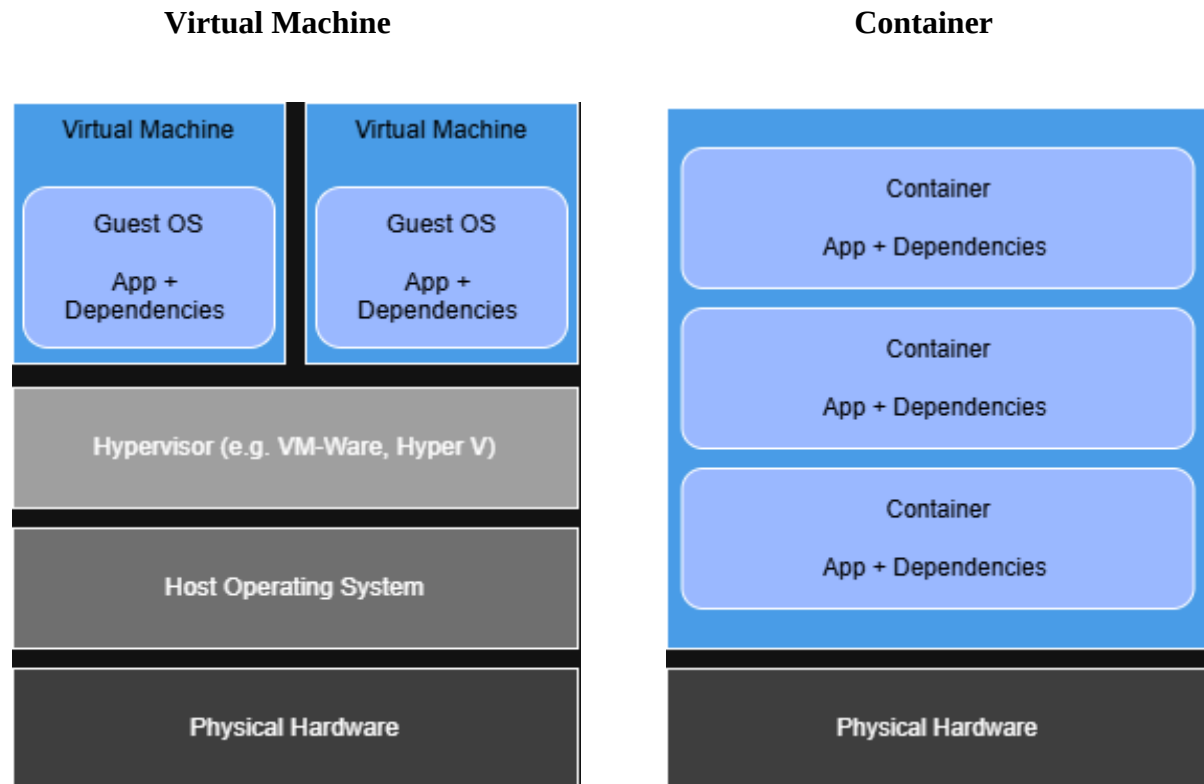
3.1 Introduction to Containerization

Applications and their dependencies pack in the container through containerization, they are lightweight virtualization technique. By relying on the host kernel, containers operate reliably in every environment and are portable, resource-efficient and fast [35][36][37]. Containerization fulfill the growing demand of adaptable, scalable and efficient computing, especially in cloud and distributed systems. It is also ideal for scientific and data-intensive applications due to its low overhead and fast setup. [38]

3.2 Containers vs Virtual Machines

There are two main virtualization techniques in cloud computing virtual machines and containers. Containers provide OS level virtualization by sharing host kernel, it fast the startup and low overhead, while VMs virtualize hardware and run full guest, it consumes more resources and time [39][40][41]. Containers are of two types application container and systems containers which supports microservices and CI/CD workflow with lightweight and fast deployment [39][35]. Containers like LXD and Docker gives better performance and scalability, and LXD provide live migration and OS level features that improves scalability [39]. Despite being bulky VMs provide strong isolation, that's why secure multi-tenant are more suitable for environments [40]. Containers fit better with PaaS and SaaS, while IaaS give similar performance for both [41]. Secure runtimes and micro-VMs are developed for addressing container security, that combines container speed and VM level isolation. [40] [42].

Figure 2:



3.3 Docker Architecture

Docker implements containerization through a client-server architecture consisting of several key components:

Docker Engine:

Docker engine works on client server architecture. It has CLI (it runs docker commands), REST API (it communicates with docker deamon), and Docker daemon (it manages networks, images and container). [43]

Docker Images:

Docker images are read only template it has libraries, code, dependencies and configuration. It is modified through dockerfile, And these are the blueprints for making containers. [44]

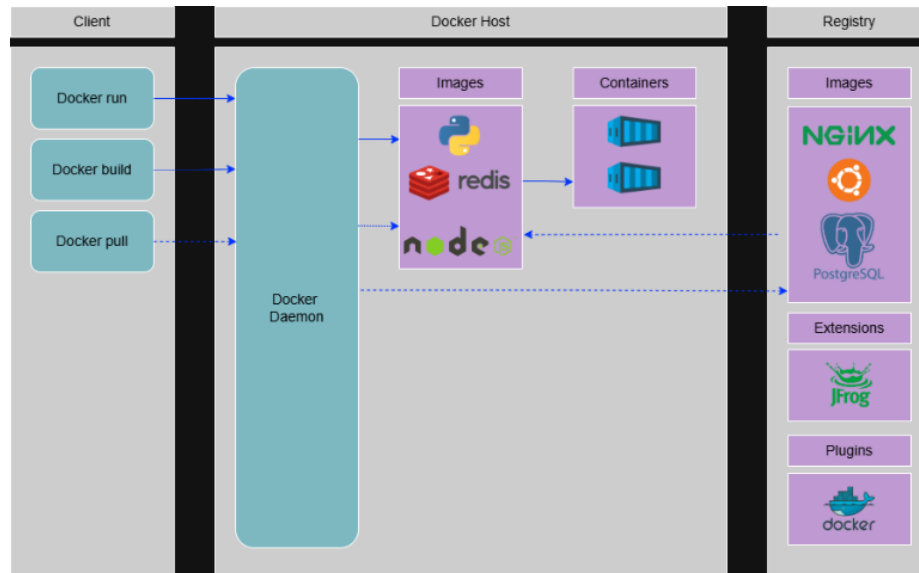
Docker Containers:

Docker containers built from images are lightweight and portable environments. These hosts share OS kernel and bundle app with their dependencies, which makes deployment fast and efficient. [45]

Docker Registry:

Docker registry is a place where Docker images are store, share and distribute. Docker hub is a famous registry, while companies also use their private registries. [46]

Figure 3:



3.4 Building Docker Images - Dockerfile Construction

A Dockerfile is a text file containing instructions for building a Docker image. Below is the Dockerfile created for this assignment:

dockerfile

Use official Nginx Alpine image as base

FROM nginx:alpine

LABEL maintainer="habibullah.sheikh351@gmail.com"

LABEL description="Cloud Engineering Portfolio Web Server"

LABEL version="1.0"

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

RUN mkdir -p /usr/share/nginx/html/data

EXPOSE 80

```
HEALTHCHECK --interval=30s --timeout=3s \
  CMD wget --quiet --tries=1 --spider http://localhost/ || exit 1
```

Dockerfile Instruction Explanation:

FROM nginx:alpine

This line specifies base image through which the image build. Alpine is a lightweight Linux distribution, this reduces the size of final image and makes pull fast. [47]

LABEL:

LABEL adds metadata like version, description or maintainer. This stores in image and tools can read it for automation or documentation.[48]

COPY:

COPY command copy files inside image using build context. Here we copy index.html to the web root of Nginx /usr/share/nginx/html/ so that the server can survey it. [47]

RUN:

RUN command executes during image build and every RUN creates new layer. This is use to create directories or install packages. It is better to combine run commands to keep the layers low. [47]

EXPOSE:

EXPOSE command tells on which port the container is listening (like port 80). But it doesn't open the port on the host unless you docker run -p it [49]

HEALTHCHECK :

HEALTHCHECK is a command that Docker run time to time to check the health of the container (healthy, starting, unhealthy). If the container gets unhealthy then Docker can restart it or trigger alert.[49]

Figure 4: Dockerfile Build Process

```
PS D:\project\docker-project> docker build -t docker-project .
[+] Building 2.7s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 148B
=> [internal] load metadata for docker.io/library/node:22
=> [auth] library/node:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/node:22@sha256:4f6d253e4ddf8407f79f9cd06a70272af5bf4f640d89241dac680f13a332dbad
=> => resolve docker.io/library/node:22@sha256:4f6d253e4ddf8407f79f9cd06a70272af5bf4f640d89241dac680f13a332dbad
=> [internal] load build context
=> => transferring context: 43.45kB
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY package*.json ./
=> CACHED [4/5] RUN npm install
=> CACHED [5/5] COPY . .
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:e1962a7e264452a60d275461bb5a6be561ff4af89333fa34bbe852867ef3f20a
=> => exporting config sha256:11f7d4e4791497cfb4d142596515f223695f54703732c6738eae96fad4821cd4
=> => exporting attestation manifest sha256:4ac25d7e54d4fc52d6c7307eca02e57de704659ed7529ebc67a58f3eaa9236fa
=> => exporting manifest list sha256:ffc15477a170efe9d832a93094d2c545dd7941bc81db3299b34e7d3d1bb4b927
=> => naming to docker.io/library/docker-project:latest
=> => unpacking to docker.io/library/docker-project:latest
PS D:\project\docker-project>
```

Figure 5: Docker Image

```
PS D:\project\docker-project> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
habibdocker2/docker-project   latest              86276d972dd1       3 days ago         1.64GB
docker-project          latest              ffc15477a170       3 days ago         1.64GB
tictoc.py              latest              6ba809cf8c64       6 days ago         1.61GB
tictoc                  latest              45ba488b5265       13 days ago        1.16GB
<none>                  <none>              23c7e48f2a2f       13 days ago        1.16GB
ubuntuimage            latest              0493220f016f       2 weeks ago        1.16GB
habibdocker2/my-project   v1                  04cb2a32dbec       2 weeks ago        274MB
habibdocker2/my-docker-project latest              e9e51acce94a       2 weeks ago        1.64GB
node                    latest              6ff78d6d45f2       3 weeks ago        237MB
alpine                  latest              4b7ce07002c6       3 weeks ago        12.8MB
nginx                    latest              3b7732505933       3 weeks ago        236MB
ubuntu                   latest              66468d557b25       4 weeks ago        117MB
```

3.5 Benefits of Containerization

Consistency Across Environments:

Docker and Kubernetes pack applications with all their dependencies, Therefore, apps run seamlessly on every machine or cloud. [50] [51]

Quick Deployment:

Containers are lightweight and portable, Therefore, apps can be deployed very quickly and reliably on cloud, fog, and other platforms. [52]

Resource Efficiency:

Containerization optimize resource usage, because it allows flexible deployment on multi cloud and makes performance and infrastructure utilization better. [53]

Microservices Architecture:

Every microservice runs in its different container, which makes the system more manageable, scalable, and modular. Tools like Kubernetes make multi-container systems easy to deploy. [54]

Version Control:

Containers improve automation and collaboration in DevOps, and reduce migration or versioning errors. [55]

Cloud Portability:

Containers isolate apps with their dependencies, Which makes the "turn them on, run blindly" approach possible across all clouds. [56]

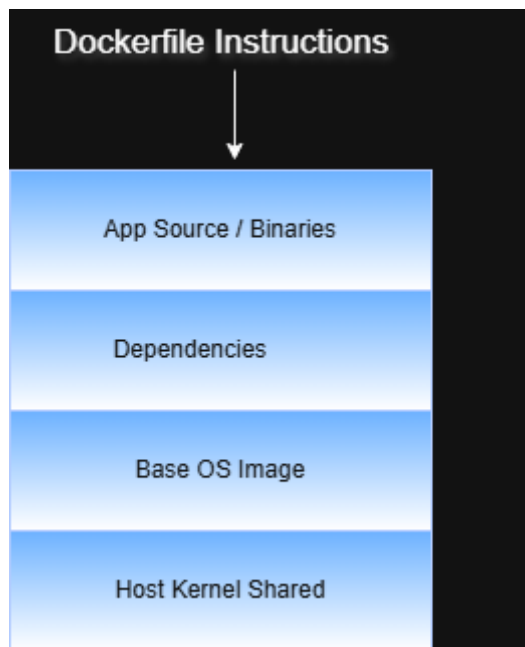
4. Docker Images and Publishing

4.1 Understanding Docker Images

A container is a standardized and unchangeable package that contains everything needed to run the application like binaries, data, libraries and configuration files. These images are build in layered structure so that efficiency can increase, and developers add their code easily on base image like Python and Node.js. Global marketplaces are use to store and distribute images, like Docker Hub, where developers easily find verified and reliable images so they can easily deploy. [44]

4.2 Image Layers and Caching

Docker images are created from different layered files that are executed by each Dockerfile instruction. Because of this layered structure, only the changed layers have to be saved and transferred, making data reuse and sharing more efficient. But after repeated rebuilds, when files have to be downloaded repeatedly, image quality can be slow [57]. Technologies like FastBuild use local caching so that remote downloads are not repeated, and overlap layer writing with instruction execution. Due to these optimizations, builds become very fast and network usage even slower. [57]



4.3 Publishing Images to DockerHub

DockerHub is a cloud-based registry service that allows developers to share Docker images [46].

Publication Process:

Step 1: Tag the Image

bash

```
docker tag my-portfolio:v1.0 habibdocker2/my-portfolio:v1.0
```

Tagging assigns a repository name under your Docker Hub account (habibdocker2/...) and a version tag (:v1.0). [58]

Step 2: Login to DockerHub

bash

docker login

This authenticates your Docker CLI with Docker Hub so you can push your image. [59]

Step 3: Push Image

bash

docker push username/my-portfolio:v1.0

This uploads your image to the repository. Only layers not already in the registry are sent, which makes the process efficient. [60]

4.4 Practical Implementation

Figure 7: Building the Image

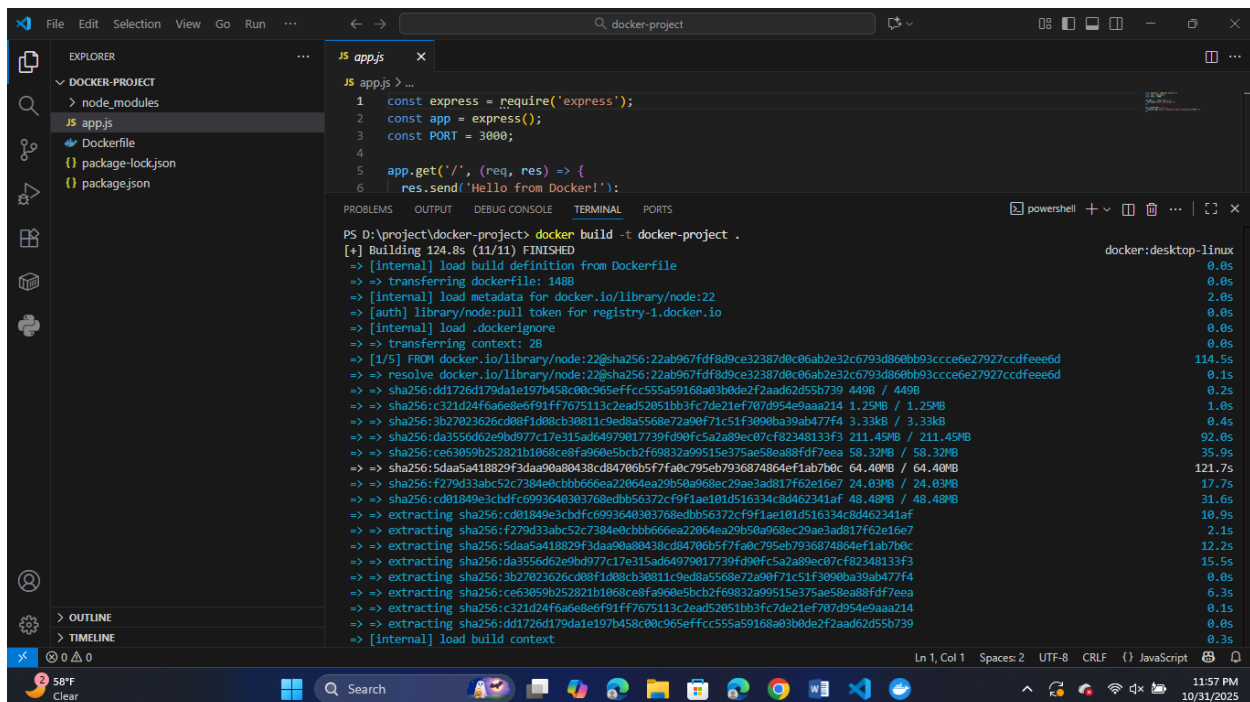


Figure 8: Local Docker Images

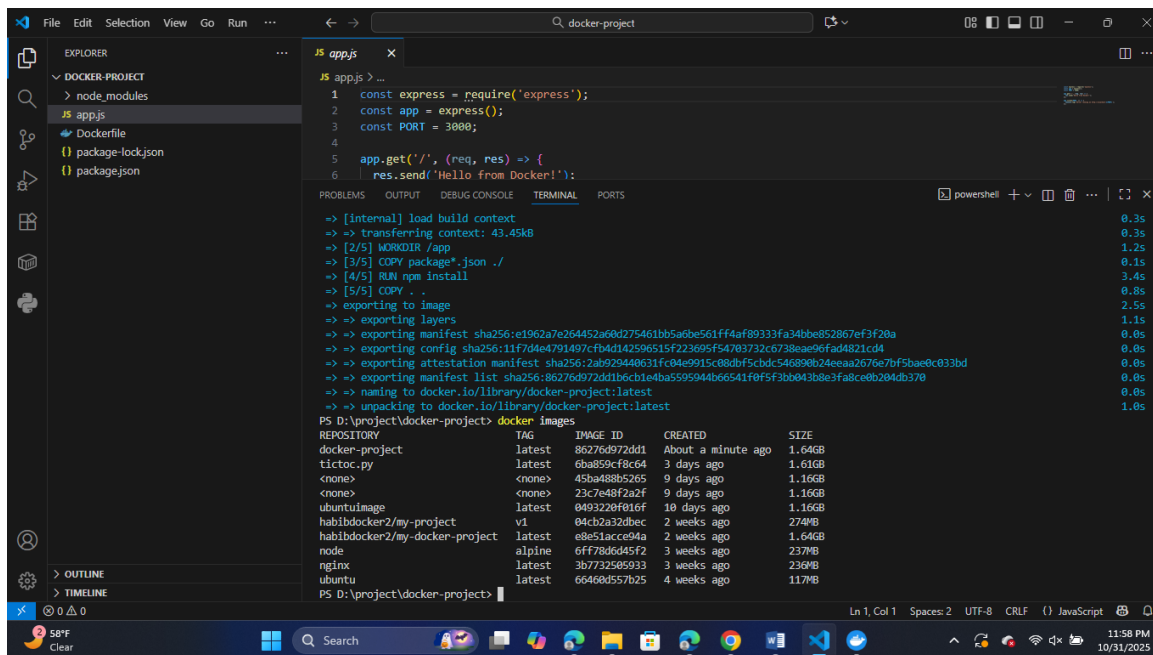


Figure 9: Pushing to DockerHub

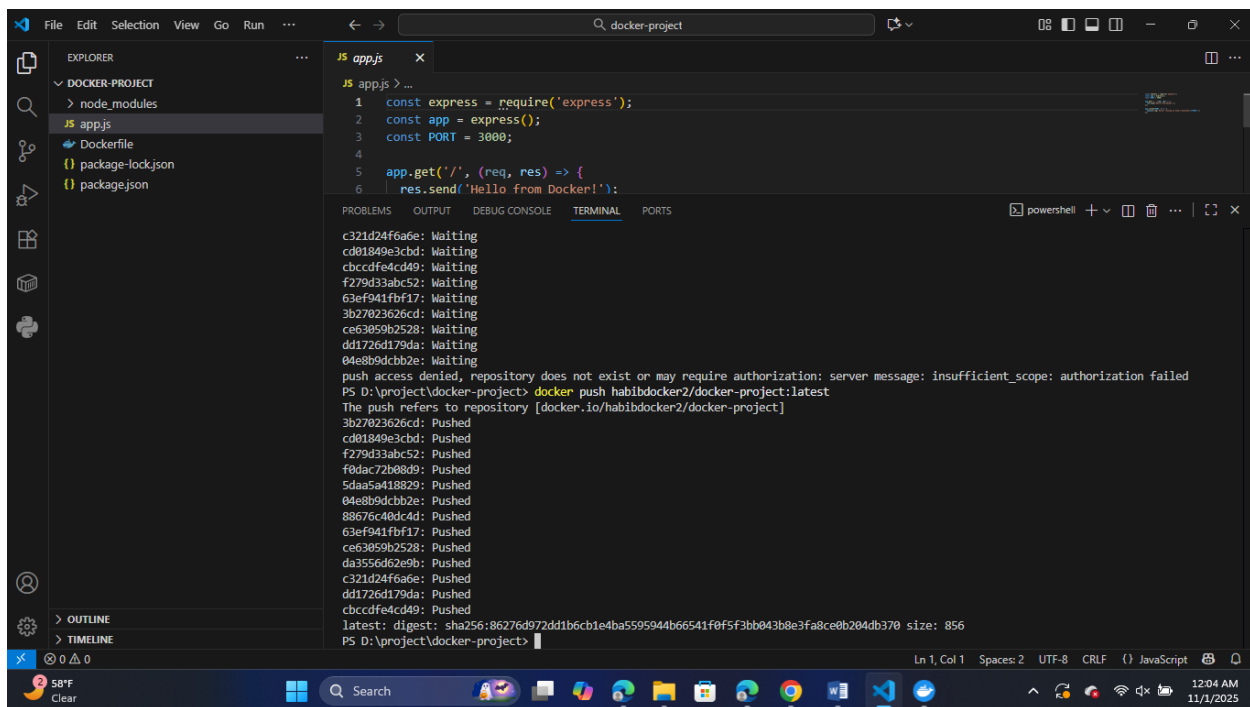
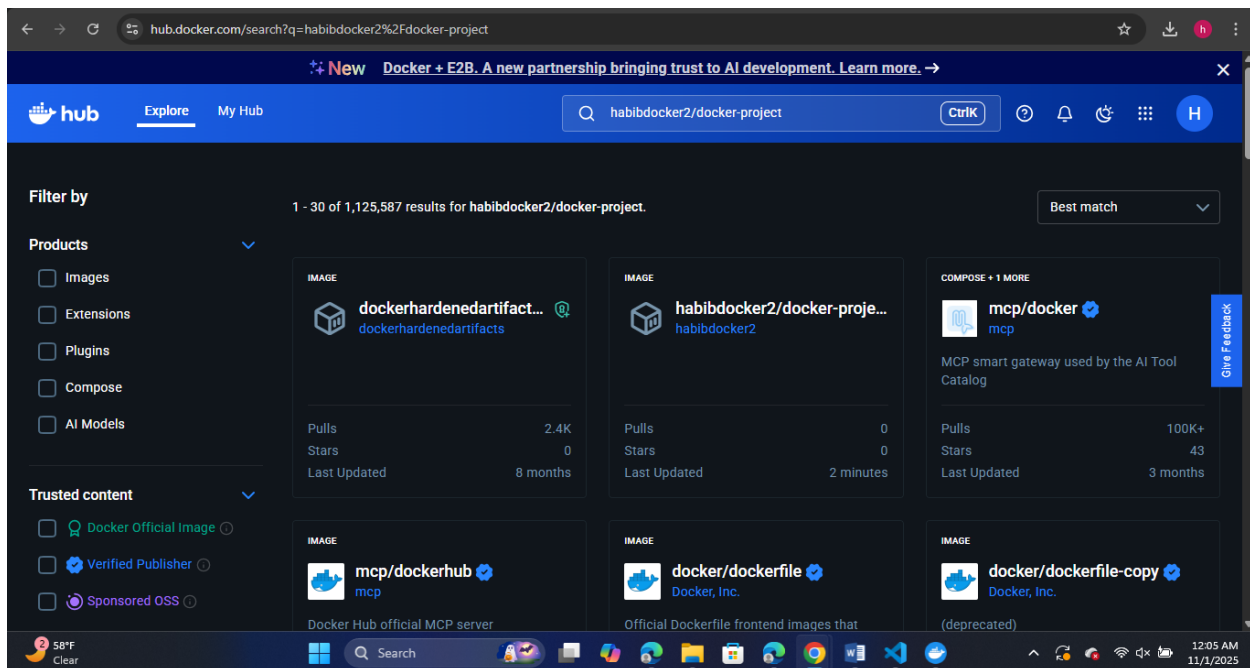


Figure 10: Image on DockerHub



5. Container Volumes

5.1 The Need for Data Persistence

Containers, like Docker, use temporary file systems, Therefore, the data stored inside them is lost when the container is broken or removed. That's why persistence mechanisms are important [61] [62]. Applications like Databases, logs and user uploads needs persistent storage so that the data will not loss during redeployment or restart. [63] [62]. If Docker volumes are not present, important data loss can occur, which negatively impacts reliability, migration, and recovery. [61] [63]

5.2 Types of Docker Volumes

Docker provide three primary storage methods.

Volumes: It us useful for most of the cases. They are manage through Docker and save in its docker storage are. [64]

Blind Mounts: Use for development, because they map a host machine's folder in a container. [65]

tmpfs Mounts: These store data only temporarily in memory, and the data is automatically deleted as soon as the container is closed. [66]

5.3 Creating and Managing Volumes

Creating a Named Volume:

bash

```
docker volume create portfolio-data
```

- ❑ Creates a persistent Docker volume named portfolio-data.
- ❑ Data remains even if the container is removed.

Mounting Volume to Container:

bash

```
docker run -d -p 8080:80 \  
-v portfolio-data:/usr/share/nginx/html/data \  
my-portfolio:v1.0
```

- ❑ -v <volume>:<container_path> mounts the volume inside the container.
- ❑ /usr/share/nginx/html/data now points to portfolio-data.
- ❑ Changes inside this path persist beyond the container lifecycle.

5.4 Practical Implementation

Figure 11: Creating Docker Volume

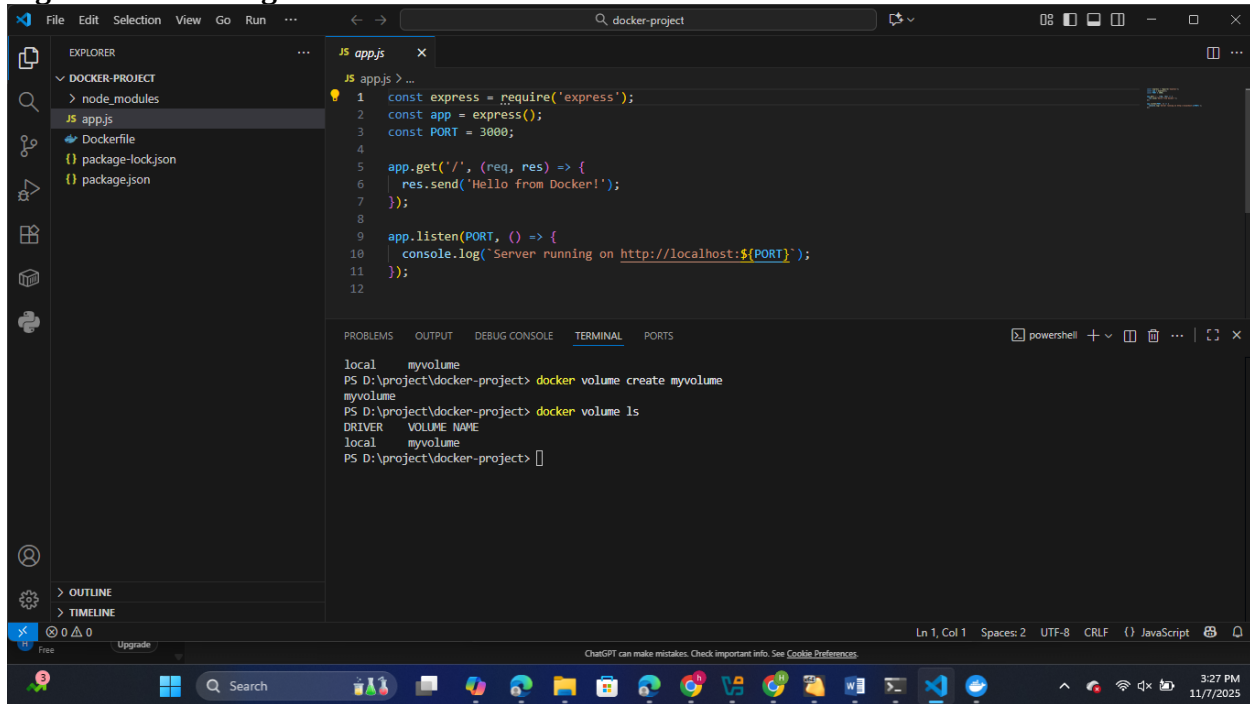


Figure 12: Volume Inspection

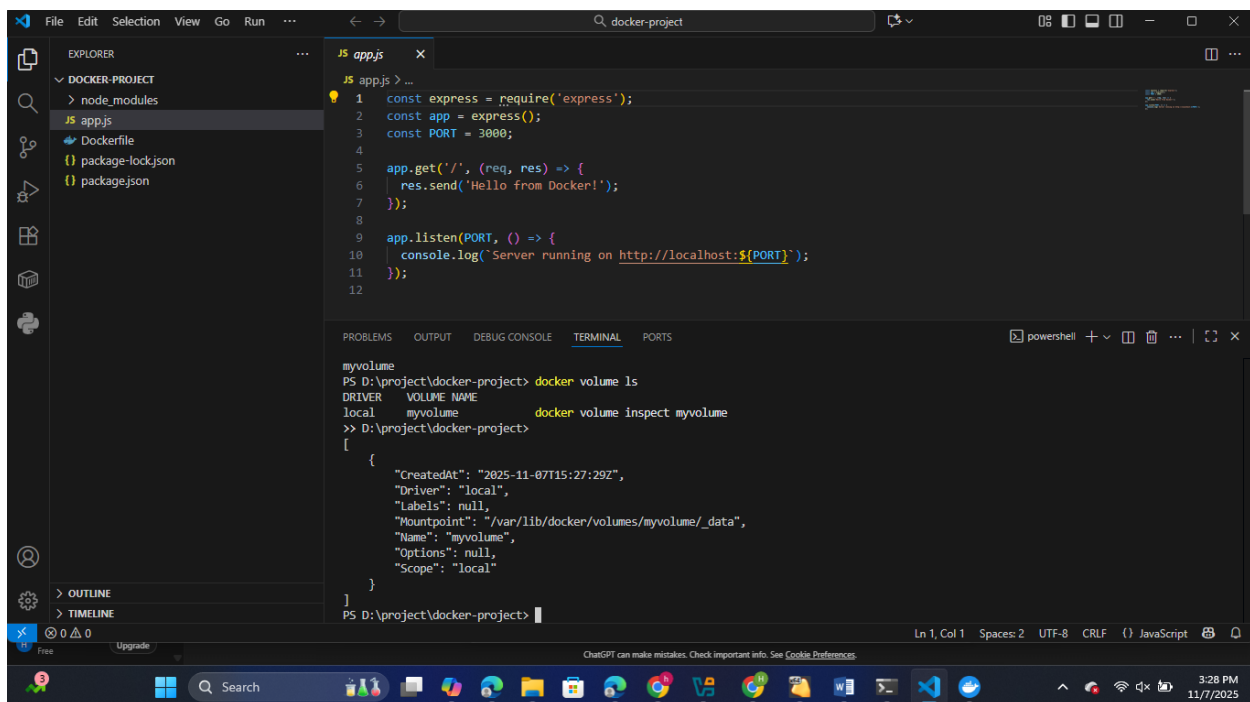
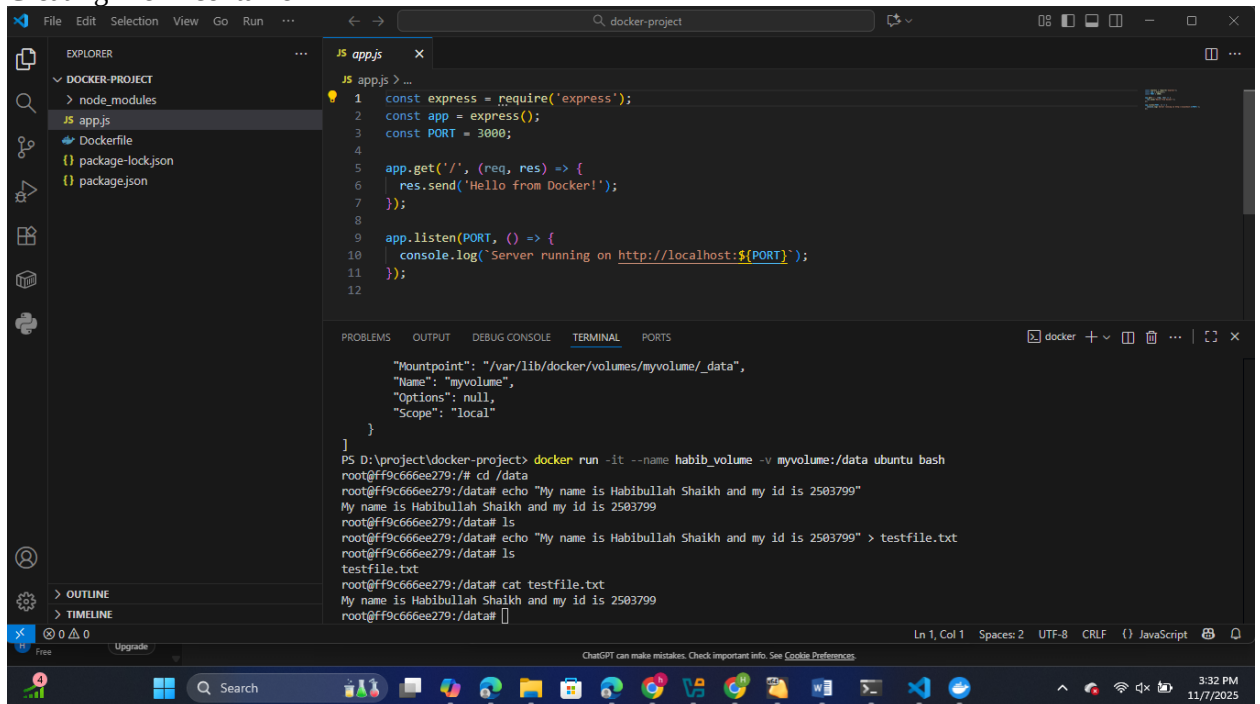


Figure 13: Testing Data Persistence

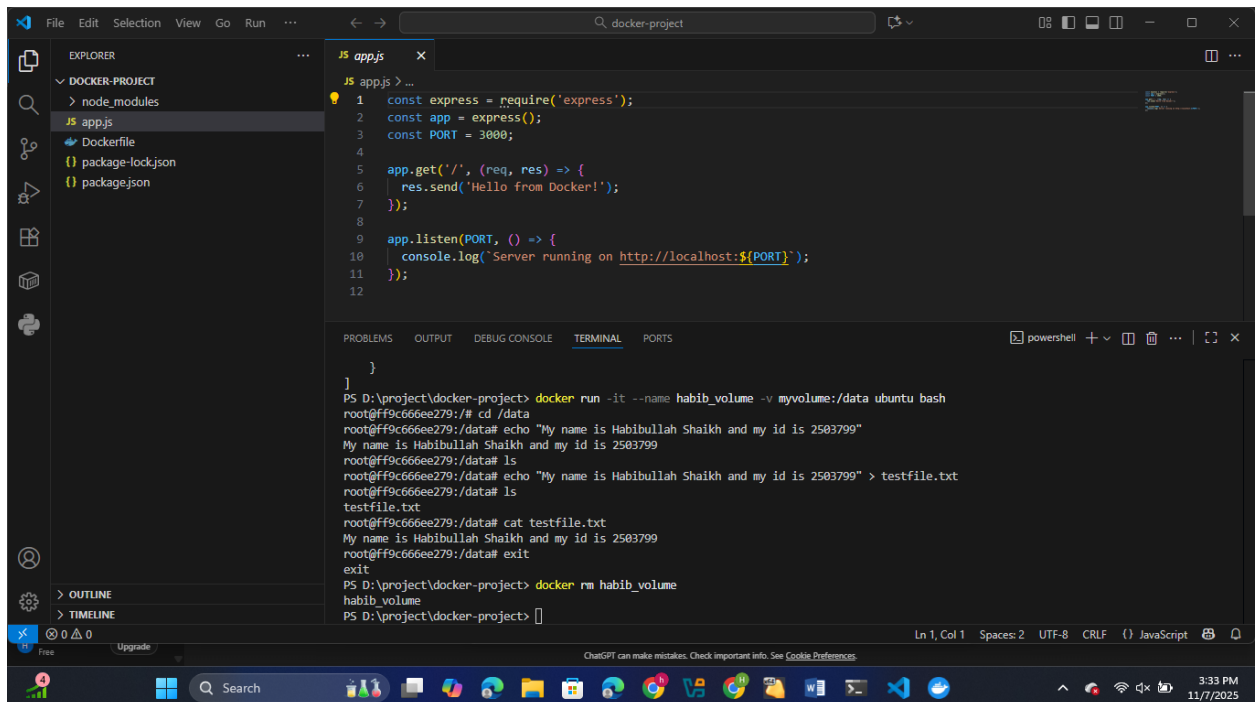
1. Creating file in container



```
JS app.js x
JS app.js > ...
1 const express = require('express');
2 const app = express();
3 const PORT = 3000;
4
5 app.get('/', (req, res) => {
6   res.send('Hello from Docker!');
7 });
8
9 app.listen(PORT, () => {
10   console.log(`Server running on http://localhost:${PORT}`);
11 });
12

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Mountpoint: "/var/lib/docker/volumes/myvolume/_data",
Name: "myvolume",
Options: null,
Scope: "local"
}
]
PS D:\project\docker-project> docker run -it --name habib_volume -v myvolume:/data ubuntu bash
root@ff9c666ee279:/# cd /data
root@ff9c666ee279:/data# echo "My name is Habibullah Shaikh and my id is 2503799"
My name is Habibullah Shaikh and my id is 2503799
root@ff9c666ee279:/data# ls
root@ff9c666ee279:/data# echo "My name is Habibullah Shaikh and my id is 2503799" > testfile.txt
root@ff9c666ee279:/data# ls
testfile.txt
root@ff9c666ee279:/data# cat testfile.txt
My name is Habibullah Shaikh and my id is 2503799
root@ff9c666ee279:/data#
```

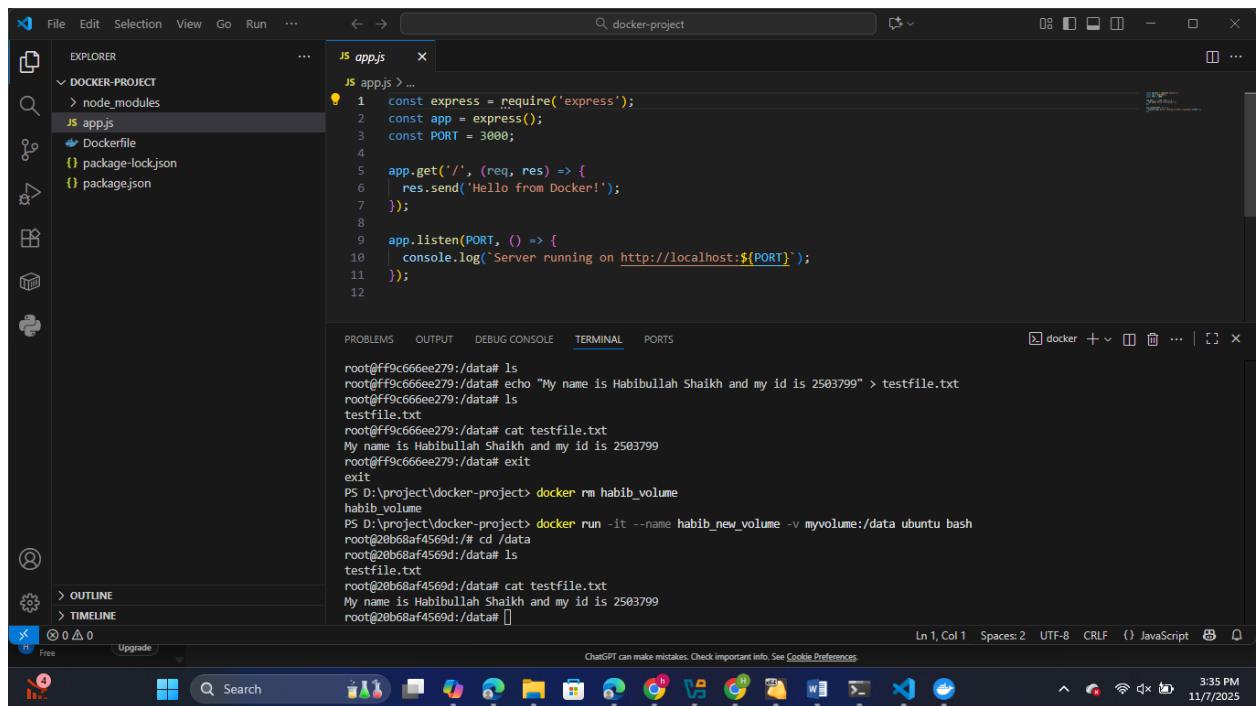
2. Stopping and removing container



```
JS app.js x
JS app.js > ...
1 const express = require('express');
2 const app = express();
3 const PORT = 3000;
4
5 app.get('/', (req, res) => {
6   res.send('Hello from Docker!');
7 });
8
9 app.listen(PORT, () => {
10   console.log(`Server running on http://localhost:${PORT}`);
11 });
12

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
}
]
PS D:\project\docker-project> docker run -it --name habib_volume -v myvolume:/data ubuntu bash
root@ff9c666ee279:/# cd /data
root@ff9c666ee279:/data# echo "My name is Habibullah Shaikh and my id is 2503799"
My name is Habibullah Shaikh and my id is 2503799
root@ff9c666ee279:/data# ls
root@ff9c666ee279:/data# echo "My name is Habibullah Shaikh and my id is 2503799" > testfile.txt
root@ff9c666ee279:/data# ls
testfile.txt
root@ff9c666ee279:/data# cat testfile.txt
My name is Habibullah Shaikh and my id is 2503799
root@ff9c666ee279:/data# exit
exit
PS D:\project\docker-project> docker rm habib_volume
habib_volume
PS D:\project\docker-project>
```

3. Starting new container with same volume



The screenshot shows the Visual Studio Code interface with a Docker project. The Explorer sidebar on the left shows the project structure: `DOCKER-PROJECT` containing `node_modules`, `app.js`, `Dockerfile`, `package-lock.json`, and `package.json`. The main editor displays the `app.js` file with the following code:

```
1 const express = require('express');
2 const app = express();
3 const PORT = 3000;
4
5 app.get('/', (req, res) => {
6   res.send('Hello from Docker!');
7 });
8
9 app.listen(PORT, () => {
10   console.log(`Server running on http://localhost:${PORT}`);
11 });
12
```

The bottom panel shows the TERMINAL with the following commands and output:

```
root@ff9c666ee279:/data# ls
root@ff9c666ee279:/data# echo "My name is Habibullah Shaikh and my id is 2583799" > testfile.txt
root@ff9c666ee279:/data# ls
testfile.txt
root@ff9c666ee279:/data# cat testfile.txt
My name is Habibullah Shaikh and my id is 2583799
root@ff9c666ee279:/data# exit
exit
PS D:\project\docker-project> docker rm habib_volume
habib_volume
PS D:\project\docker-project> docker run -it --name habib_new_volume -v myvolume:/data ubuntu bash
root@20b68af4569d:/# cd /data
root@20b68af4569d:/data# ls
testfile.txt
root@20b68af4569d:/data# cat testfile.txt
My name is Habibullah Shaikh and my id is 2583799
root@20b68af4569d:/data#
```

6. Container Networking

6.1 Docker Networking Concepts

Docker networking allows container to connect with an another host and external systems, which is very important for multi container applications [67][68]. By default, Docker uses cgroups and linux namespaces to isolate containers in a secure way. [68][69]. User defined networks, like overlay and bridge, provide controlled communication, which makes tight and selective connectivity possible. [67] [70].

6.2 Network Drivers

Docker provides network drivers for container connectivity:

- **Bridge (Default):** Use mostly in single host environments. This allows containers to communicate with each other on a same host. [71].
- **Host:** This share host's network stack, performance improves but isolations are removed. [72]
- **Overlay:** Connects container on multiple hosts, and Docker Swarm is best for environments. [73]

- **Macvlan:** It gives unique MAC address to every container, through which it looks like a physical device on a network. [74]
- **None:** It completely disables networking, provides full isolation [75].

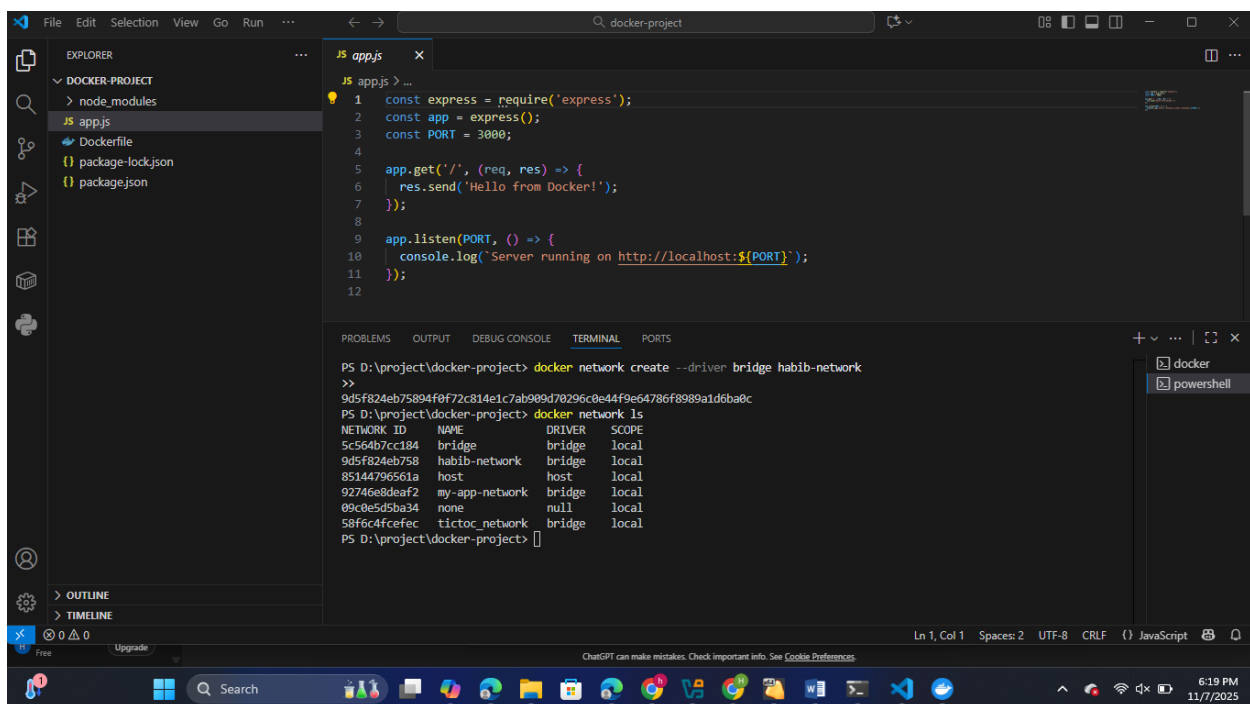
6.3 Container Communication

Docker's built-in DNS service makes discovery and communication within virtual networks possible, which allows containers to be interconnected by using names instead of IPs [76]. This communication is done through different network drivers, that provides different levels of isolation and performance [76]. Port mapping forwards host traffic to container ports by using NAT, which gives external access to internal services while remaining in isolation. [77].

6.4 Practical Implementation

Creating Custom Network:

Figure 14: Creating Docker Network

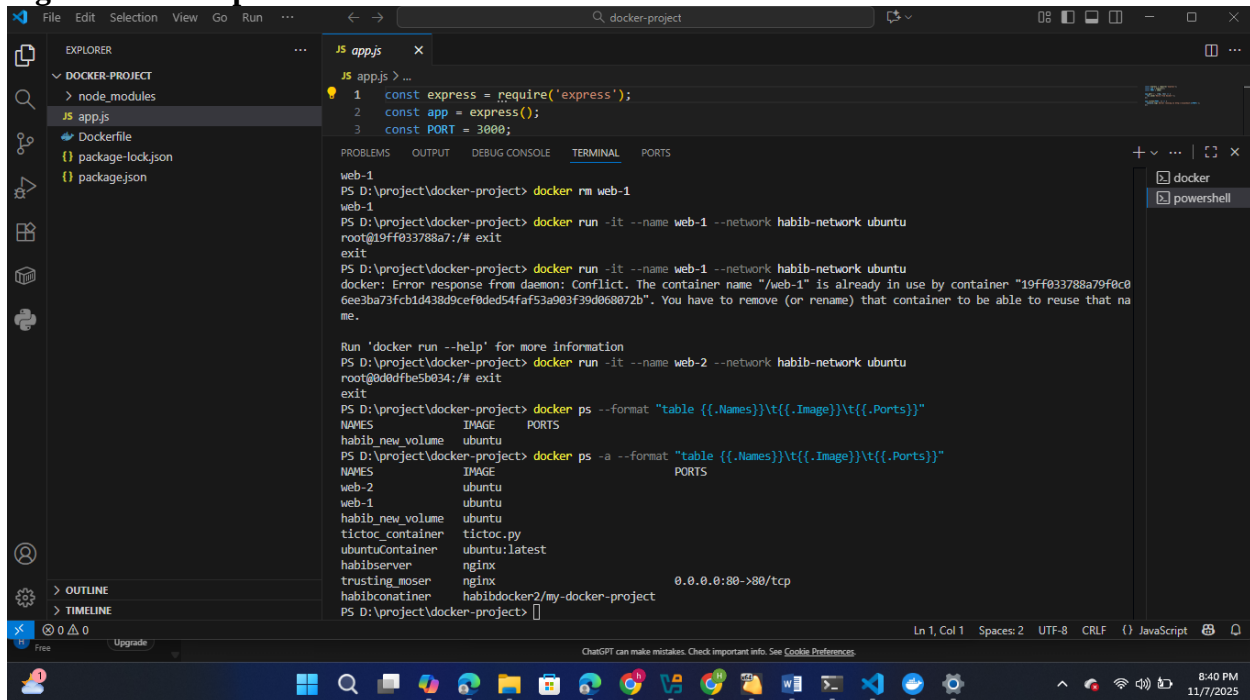


The screenshot shows a Visual Studio Code editor with a project named 'docker-project'. The Explorer sidebar on the left shows the file structure: 'DOCKER-PROJECT' containing 'node_modules', 'app.js', 'Dockerfile', 'package-lock.json', and 'package.json'. The main editor area displays the content of 'app.js', which is a simple Express.js server. Below the editor, the TERMINAL panel is open, showing the execution of Docker commands to create and list networks.

```
PS D:\project\docker-project> docker network create --driver bridge habib-network
>>
9d5f824eb75894f0f72c814e1c7ab909d70296c0e44f9e64786f8989a1d6ba0c
PS D:\project\docker-project> docker network ls
NETWORK ID        NAME                DRIVER              SCOPE
5c564b7cc184      bridge              bridge              local
9d5f824eb758      habib-network       bridge              local
85144796561a      host                host                local
92746e8dea2f      my-app-network      bridge              local
09c8e5d5ba34      none                null                local
58f6c4fcefc       tictoc_network      bridge              local
PS D:\project\docker-project>
```

Running Containers on Same Network:

Figure 15: Multiple Containers on Network

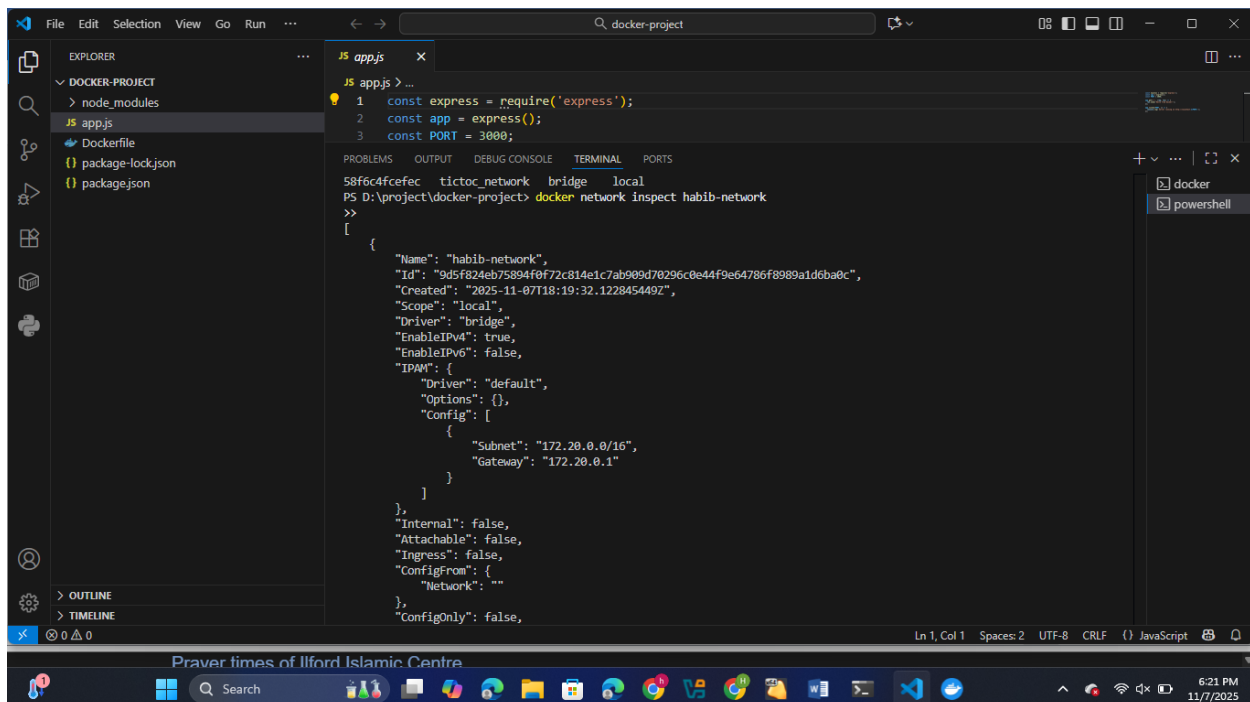


The screenshot shows a VS Code editor with a file explorer on the left and a terminal window on the right. The file explorer shows a project named 'docker-project' with files like 'Dockerfile', 'package-lock.json', and 'package.json'. The terminal window shows the following commands and output:

```
PS D:\project\docker-project> docker rm web-1
web-1
PS D:\project\docker-project> docker run -it --name web-1 --network habib-network ubuntu
root@19ff033788a7:/# exit
exit
PS D:\project\docker-project> docker run -it --name web-1 --network habib-network ubuntu
docker: Error response from daemon: Conflict. The container name "/web-1" is already in use by container "19ff033788a79f0c06ee3ba73fcb1d438d9cef0ded54faf53a903f39d068072b". You have to remove (or rename) that container to be able to reuse that name.

Run 'docker run --help' for more information
PS D:\project\docker-project> docker run -it --name web-2 --network habib-network ubuntu
root@80dfb5b034:/# exit
exit
PS D:\project\docker-project> docker ps --format "table {{.Names}}\t{{.Image}}\t{{.Ports}}"
TABLE      IMAGE      PORTS
habib_new_volume  ubuntu
PS D:\project\docker-project> docker ps -a --format "table {{.Names}}\t{{.Image}}\t{{.Ports}}"
TABLE      IMAGE      PORTS
web-2      ubuntu
web-1      ubuntu
habib_new_volume  ubuntu
tictoc_container  tictoc.py
ubuntuContainer  ubuntu:latest
habibserver      nginx
trusting_moser   nginx
habibcontainer   habibdocker2/my-docker-project
PS D:\project\docker-project>
```

Figure 16: Network Inspection

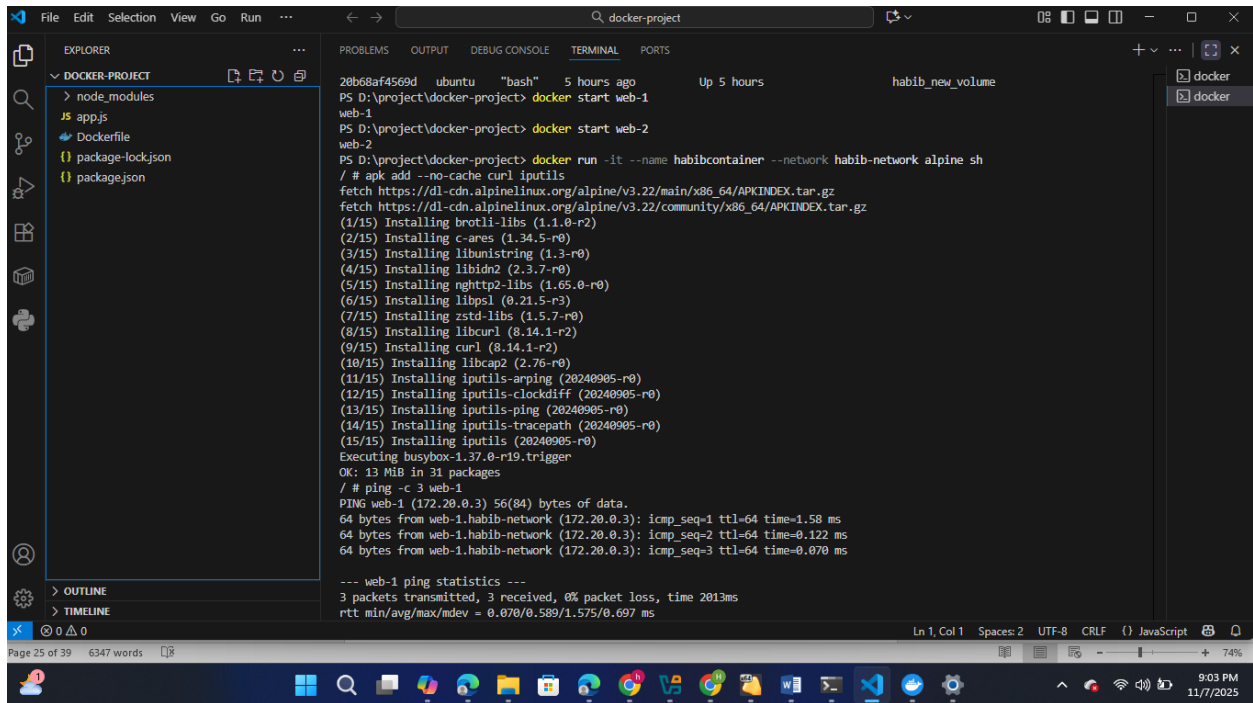


The screenshot shows a VS Code editor with a file explorer on the left and a terminal window on the right. The file explorer shows a project named 'docker-project' with files like 'Dockerfile', 'package-lock.json', and 'package.json'. The terminal window shows the following commands and output:

```
PS D:\project\docker-project> docker network inspect habib-network
[
  {
    "Name": "habib-network",
    "Id": "9d5f824eb75894f0f72c814e1c7ab909d70296c0e44f9e64786f8989a1d0ba0c",
    "Created": "2025-11-07T18:19:32.122845449Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv4": true,
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.20.0.0/16",
          "Gateway": "172.20.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
  }
]
```

Testing Container Communication:

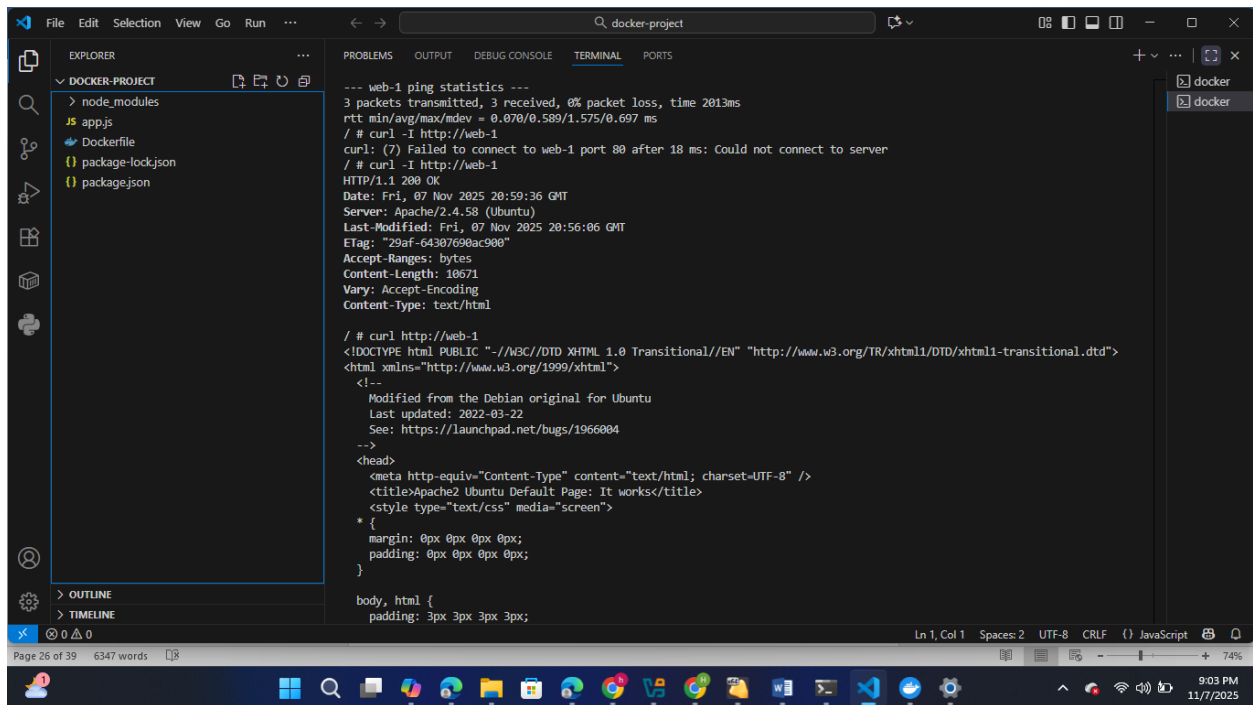
Figure 17: Container-to-Container Communication



The screenshot shows the Visual Studio Code interface with a Docker project. The Explorer pane on the left shows the project structure with files like `node_modules`, `app.js`, `Dockerfile`, `package-lock.json`, and `package.json`. The Terminal pane on the right shows the execution of Docker commands and the output of a ping test. The output indicates that the containers are successfully communicating.

```
PS D:\project\docker-project> docker start web-1
web-1
PS D:\project\docker-project> docker start web-2
web-2
PS D:\project\docker-project> docker run -it --name habibcontainer --network habib-network alpine sh
/ # apk add --no-cache curl iputils
fetch https://dl-cdn.alpinelinux.org/alpine/v3.22/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.22/community/x86_64/APKINDEX.tar.gz
(1/15) Installing brotli-libs (1.1.0-r2)
(2/15) Installing c-ares (1.34.5-r0)
(3/15) Installing libunistring (1.3-r0)
(4/15) Installing libidn2 (2.3.7-r0)
(5/15) Installing nghttp2-libs (1.65.0-r0)
(6/15) Installing libpsl (0.21.5-r3)
(7/15) Installing zstd-libs (1.5.7-r0)
(8/15) Installing libcurl (8.14.1-r2)
(9/15) Installing curl (8.14.1-r2)
(10/15) Installing libcap2 (2.76-r0)
(11/15) Installing iputils-arping (20240905-r0)
(12/15) Installing iputils-clockdiff (20240905-r0)
(13/15) Installing iputils-ping (20240905-r0)
(14/15) Installing iputils-tracepath (20240905-r0)
(15/15) Installing iputils (20240905-r0)
Executing busybox-1.37.0-r19.trigger
OK: 13 MiB in 31 packages
/ # ping -c 3 web-1
PING web-1 (172.20.0.3) 56(84) bytes of data.
64 bytes from web-1.habib-network (172.20.0.3): icmp_seq=1 ttl=64 time=1.58 ms
64 bytes from web-1.habib-network (172.20.0.3): icmp_seq=2 ttl=64 time=0.122 ms
64 bytes from web-1.habib-network (172.20.0.3): icmp_seq=3 ttl=64 time=0.070 ms

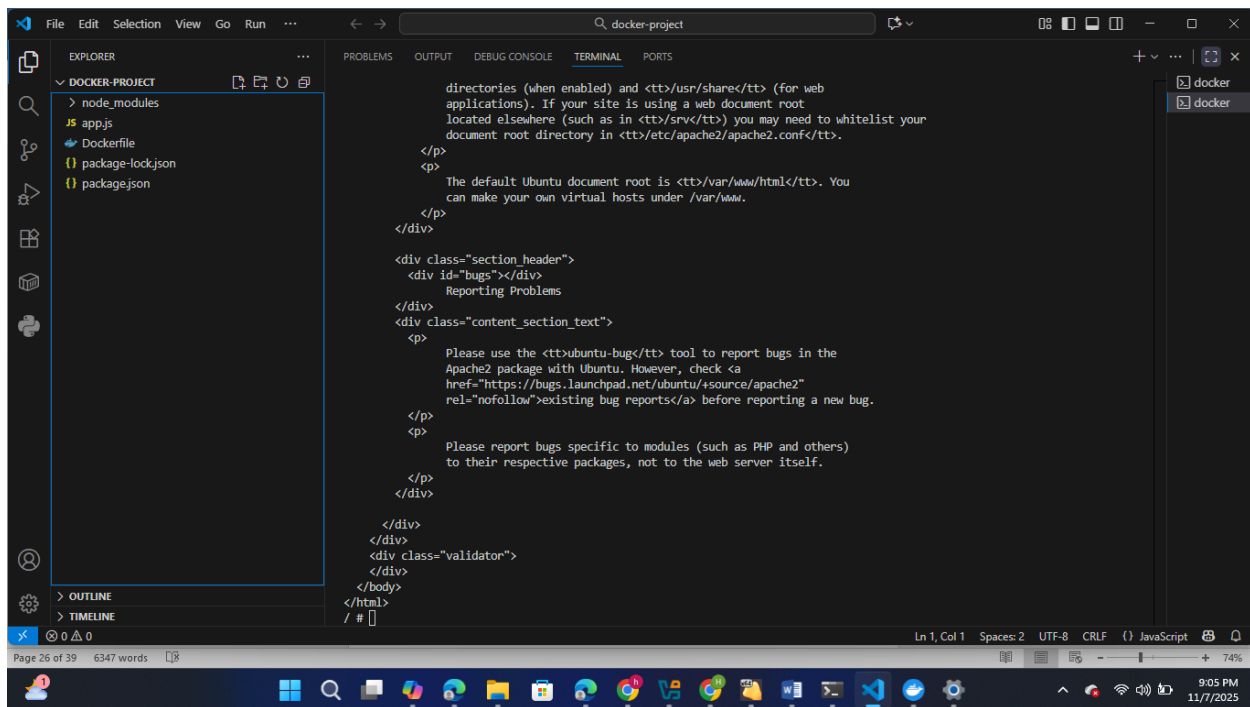
--- web-1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2013ms
rtt min/avg/max/mdev = 0.070/0.589/1.575/0.697 ms
```



The screenshot shows the Visual Studio Code interface with a Docker project. The Explorer pane on the left shows the project structure with files like `node_modules`, `app.js`, `Dockerfile`, `package-lock.json`, and `package.json`. The Terminal pane on the right shows the execution of Docker commands and the output of a curl test. The output indicates that the containers are successfully communicating.

```
--- web-1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2013ms
rtt min/avg/max/mdev = 0.070/0.589/1.575/0.697 ms
/ # curl -I http://web-1
curl: (7) Failed to connect to web-1 port 80 after 18 ms: Could not connect to server
/ # curl -I http://web-1
HTTP/1.1 200 OK
Date: Fri, 07 Nov 2025 20:59:36 GMT
Server: Apache/2.4.58 (Ubuntu)
Last-Modified: Fri, 07 Nov 2025 20:56:06 GMT
ETag: "29af-64307698ac900"
Accept-Ranges: bytes
Content-Length: 10671
Vary: Accept-Encoding
Content-Type: text/html

/ # curl http://web-1
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <!--
    Modified from the Debian original for Ubuntu
    Last updated: 2022-03-22
    See: https://launchpad.net/bugs/1966004
  -->
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Apache2 Ubuntu Default Page: It works</title>
    <style type="text/css" media="screen">
      * {
        margin: 0px 0px 0px 0px;
        padding: 0px 0px 0px 0px;
      }
    </style>
    <body, html {
      padding: 3px 3px 3px 3px;
    }
  </html>
```



7. Complete Integration Example - Web Server Deployment

7.1 Project Overview

This section shows a complete end to end example including Image creation using Dockerfile, Run container using volume and networking and finally publish image to DockerHub.

Project: Personal Portfolio Web Server

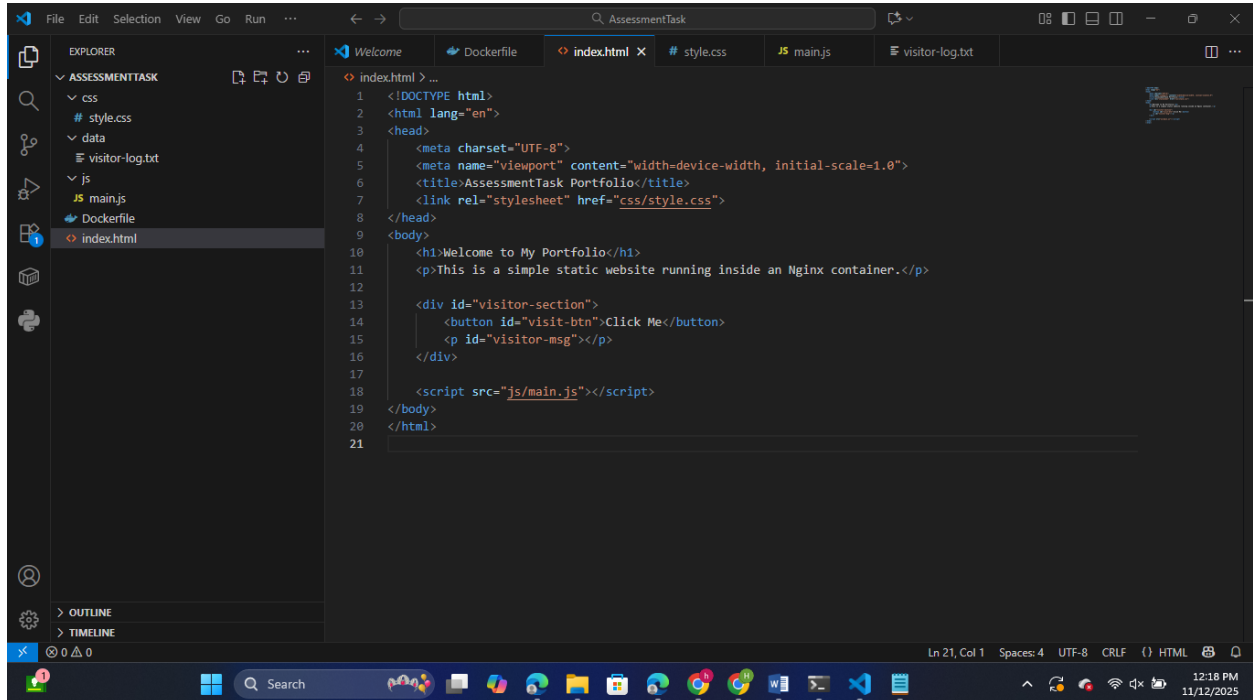
- **Technology:** Nginx web server
- **Content:** HTML/CSS portfolio page
- **Features:** Containerized, persistent storage, networked, published

7.2 Implementation Steps

Step 1: Application Development

I created a modern, responsive portfolio website with the following structure:

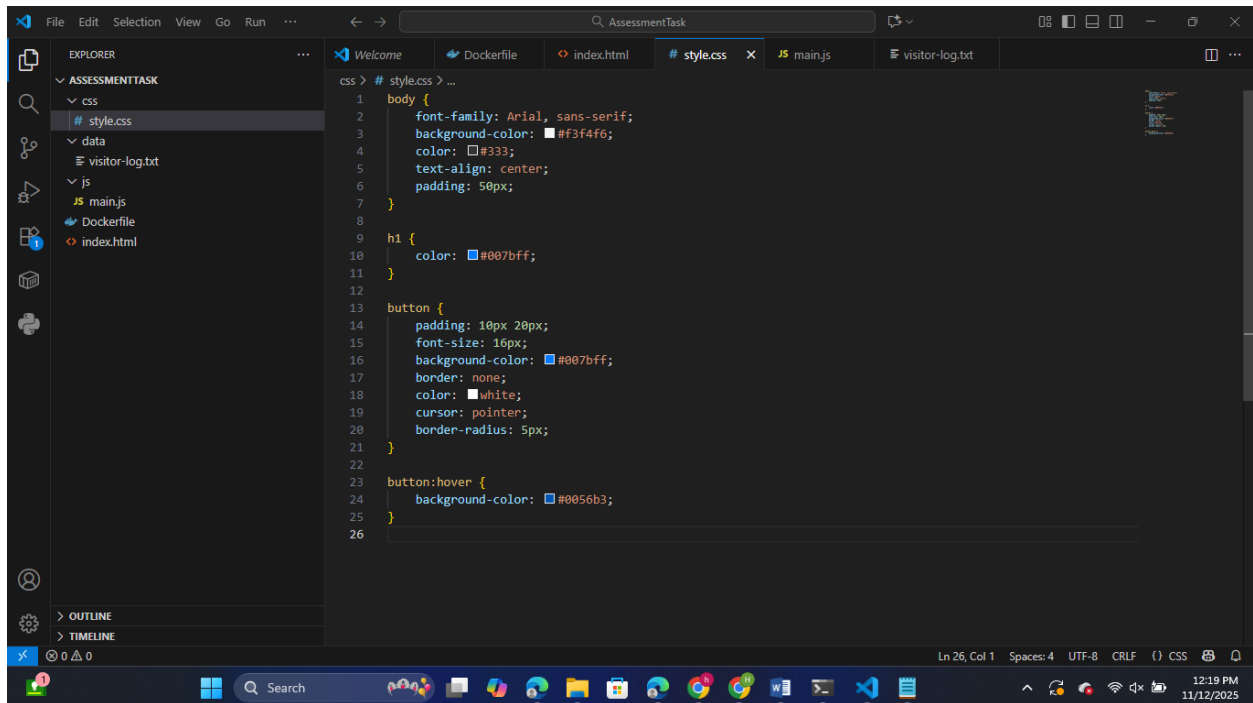
Index.html



The screenshot shows the Visual Studio Code editor with the 'index.html' file open. The Explorer sidebar on the left shows the project structure for 'ASSESSMENTTASK', including 'css', 'data', 'js', and 'Dockerfile'. The main editor area displays the HTML code for 'index.html'.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>AssessmentTask Portfolio</title>
7   <link rel="stylesheet" href="css/style.css">
8 </head>
9 <body>
10  <h1>Welcome to My Portfolio</h1>
11  <p>This is a simple static website running inside an Nginx container.</p>
12
13  <div id="visitor-section">
14    <button id="visit-btn">Click Me</button>
15    <p id="visitor-msg"></p>
16  </div>
17
18  <script src="js/main.js"></script>
19 </body>
20 </html>
21
```

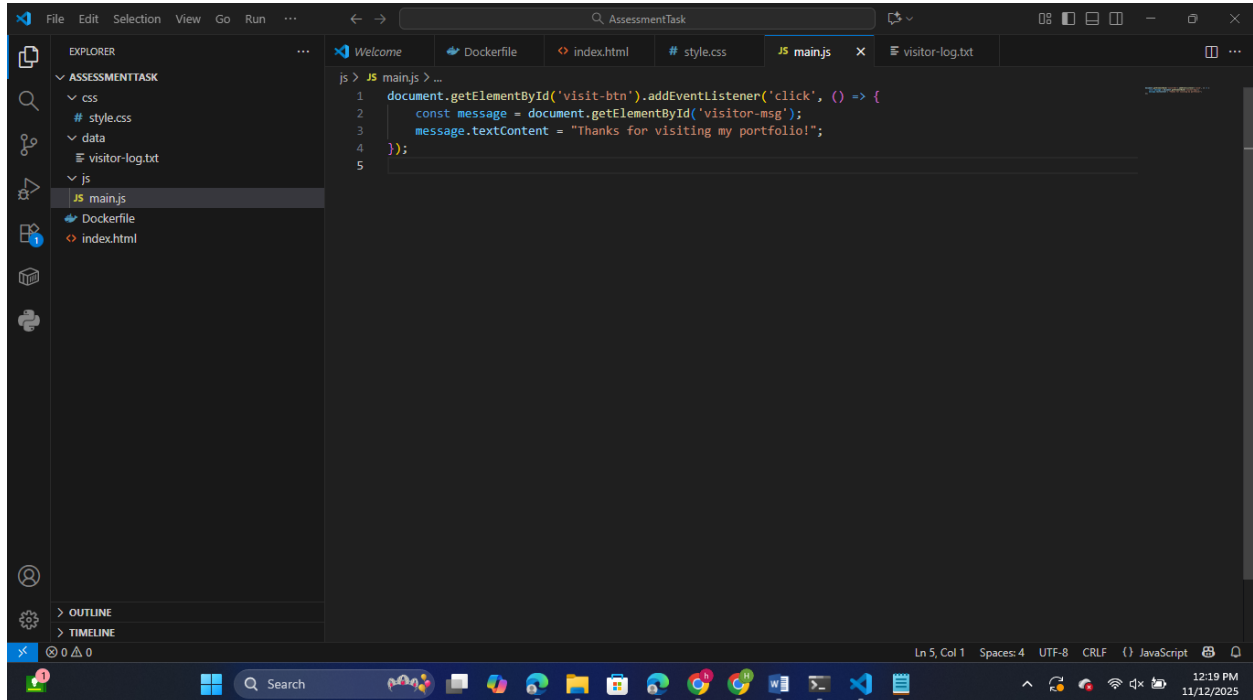
Style.css



The screenshot shows the Visual Studio Code editor with the 'style.css' file open. The Explorer sidebar on the left shows the project structure for 'ASSESSMENTTASK', including 'css', 'data', 'js', and 'Dockerfile'. The main editor area displays the CSS code for 'style.css'.

```
1 body {
2   font-family: Arial, sans-serif;
3   background-color: #f3f4f6;
4   color: #333;
5   text-align: center;
6   padding: 50px;
7 }
8
9 h1 {
10  color: #007bff;
11 }
12
13 button {
14   padding: 10px 20px;
15   font-size: 16px;
16   background-color: #007bff;
17   border: none;
18   color: white;
19   cursor: pointer;
20   border-radius: 5px;
21 }
22
23 button:hover {
24   background-color: #0056b3;
25 }
26
```

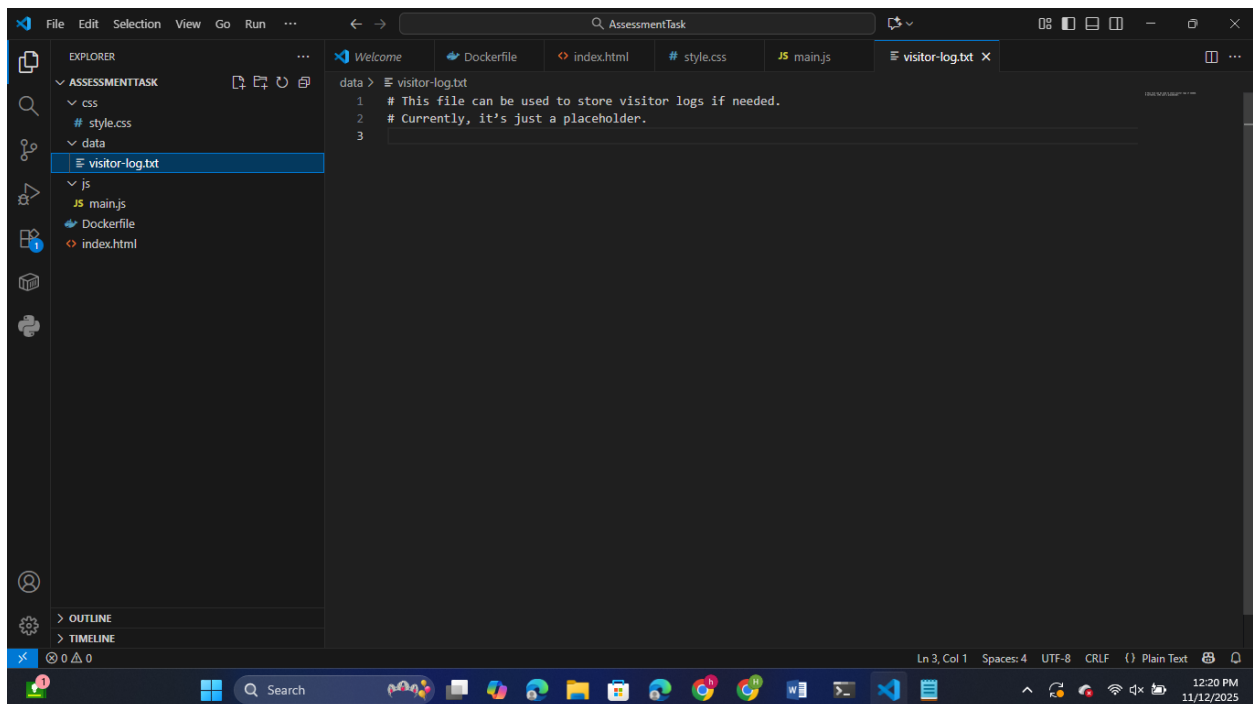
Main.js



A screenshot of the Visual Studio Code editor interface. The Explorer sidebar on the left shows a project named 'ASSESSMENTTASK' with folders for 'css', 'data', and 'js'. The 'js' folder is expanded, showing 'main.js' selected. The main editor area displays the content of 'main.js', which contains a single event listener for a button with id 'visit-btn'. The status bar at the bottom indicates 'Ln 5, Col 1', 'Spaces: 4', 'UTF-8', 'CRLF', and 'JavaScript'.

```
js > JS main.js > ...
1  document.getElementById('visit-btn').addEventListener('click', () => {
2      const message = document.getElementById('visitor-msg');
3      message.textContent = "Thanks for visiting my portfolio!";
4  });
5
```

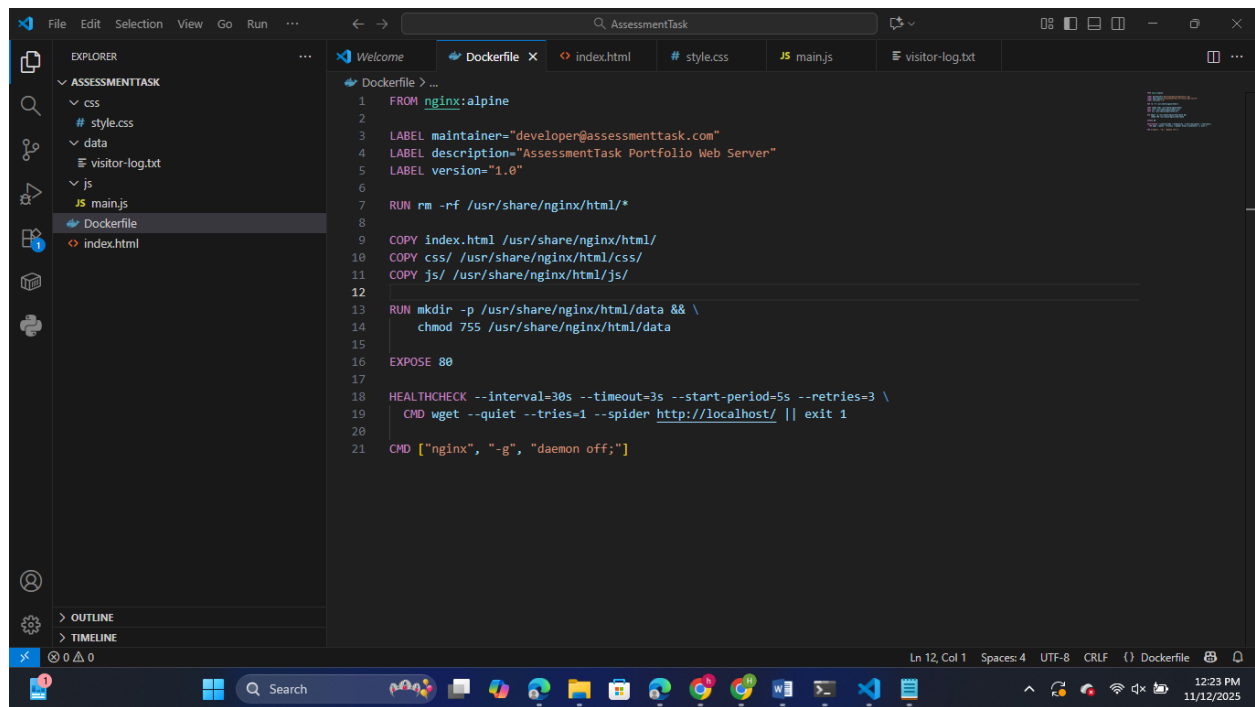
Visitor-log.txt



A screenshot of the Visual Studio Code editor interface. The Explorer sidebar on the left shows the 'ASSESSMENTTASK' project with the 'data' folder expanded, and 'visitor-log.txt' selected. The main editor area displays the content of 'visitor-log.txt', which contains two placeholder comments. The status bar at the bottom indicates 'Ln 3, Col 1', 'Spaces: 4', 'UTF-8', 'CRLF', and 'Plain Text'.

```
data > visitor-log.txt
1  # This file can be used to store visitor logs if needed.
2  # Currently, it's just a placeholder.
3
```

Step 2: Dockerfile Creation



The screenshot shows the Visual Studio Code editor with a Dockerfile open. The Explorer sidebar on the left shows a project named 'ASSESSMENTTASK' with files: 'css', 'style.css', 'data', 'visitor-log.txt', 'js', 'main.js', 'Dockerfile', and 'index.html'. The Dockerfile content is as follows:

```
1 FROM nginx:alpine
2
3 LABEL maintainer="developer@assessmenttask.com"
4 LABEL description="AssessmentTask Portfolio Web Server"
5 LABEL version="1.0"
6
7 RUN rm -rf /usr/share/nginx/html/*
8
9 COPY index.html /usr/share/nginx/html/
10 COPY css/ /usr/share/nginx/html/css/
11 COPY js/ /usr/share/nginx/html/js/
12
13 RUN mkdir -p /usr/share/nginx/html/data && \
14     chmod 755 /usr/share/nginx/html/data
15
16 EXPOSE 80
17
18 HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
19     CMD wget --quiet --tries=1 --spider http://localhost/ || exit 1
20
21 CMD ["nginx", "-g", "daemon off;"]
```

The status bar at the bottom indicates the cursor is at line 12, column 1, with 4 spaces, UTF-8 encoding, and CRLF line endings.

Step 3: Image Building

bash

```
docker build -t assessmenttask:v1.0 .
```

Figure 18: Image Build Process

The screenshot shows the Visual Studio Code interface with the Dockerfile open. The Dockerfile contains a single instruction: `FROM nginx:alpine`. The terminal shows the output of the `docker build -t assessmenttask:v1.0 .` command, which successfully builds the image. The output includes details about the build process, such as the image ID `7cc2f7fdc016` and the size `80MB`. The terminal also shows the output of the `docker images` command, which lists the built image and other images on the system.

```
PS D:\project\AssessmentTask> docker build -t assessmenttask:v1.0 .
[+] Building 3.5s (12/12) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] load metadata for docker.io/library/nginx:alpine
=> [internal] load metadata for registry-1.docker.io
=> [internal] load dockerignore
=> transferring context: 2B
=> [internal] load build context
=> transferring context: 140B
=> [1/6] FROM docker.io/library/nginx:alpine@sha256:b3c65d55d7ad751196f21b7fd2e8d4da9cb430e32f646adc92441b72f82b14
=> resolve docker.io/library/nginx:alpine@sha256:b3c65d55d7ad751196f21b7fd2e8d4da9cb430e32f646adc92441b72f82b14
=> CACHED [2/6] RUN rm -rf /usr/share/nginx/html/*
=> CACHED [3/6] COPY index.html /usr/share/nginx/html/
=> CACHED [4/6] COPY css/ /usr/share/nginx/html/css/
=> CACHED [5/6] COPY js/ /usr/share/nginx/html/js/
=> CACHED [6/6] RUN mkdir -p /usr/share/nginx/html/data && chmod 755 /usr/share/nginx/html/data
=> exporting to image
=> exporting layers
=> exporting manifest sha256:9455981499465939ceb660d23e5bb9b231586291063d1f33e4cbd9026bec8fd
=> exporting config sha256:d339bf31c0dc583eca4be1e6bd3a629202e15aa013fed09813e1ca7ae40ac7f
=> exporting attestation manifest sha256:2aca0f400462c4485bd0776b50f050313d92a1c461815954a3360b17d2e45c9
=> exporting manifest list sha256:7cc2f7fdc0169bb6952f31839c04d25306c25b0c4ae5fc9951db25a96b91529
=> naming to docker.io/library/assessmenttask:v1.0
=> unpacking to docker.io/library/assessmenttask:v1.0
PS D:\project\AssessmentTask> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
assessmenttask       v1.0               7cc2f7fdc016       10 hours ago       80MB
docker-project       latest             ffc15477a170       11 days ago        1.64GB
habibdocker2/docker-project latest             86276d972dd1       11 days ago        1.64GB
tictoc.py            latest             6ba859cf8c64       2 weeks ago        1.61GB
tictoc               latest             6ba859cf8c64       2 weeks ago        1.61GB
```

Step 4: Creating Network

The screenshot shows the Visual Studio Code interface with the Dockerfile open. The Dockerfile contains a single instruction: `FROM nginx:alpine`. The terminal shows the output of the `docker images` command, which lists the built image and other images on the system. The terminal also shows the output of the `docker network rm frontend-net` command, which successfully removes the network. The terminal then shows the output of the `docker network create frontend-net` command, which successfully creates the network. The terminal also shows the output of the `docker network ls` command, which lists the networks on the system.

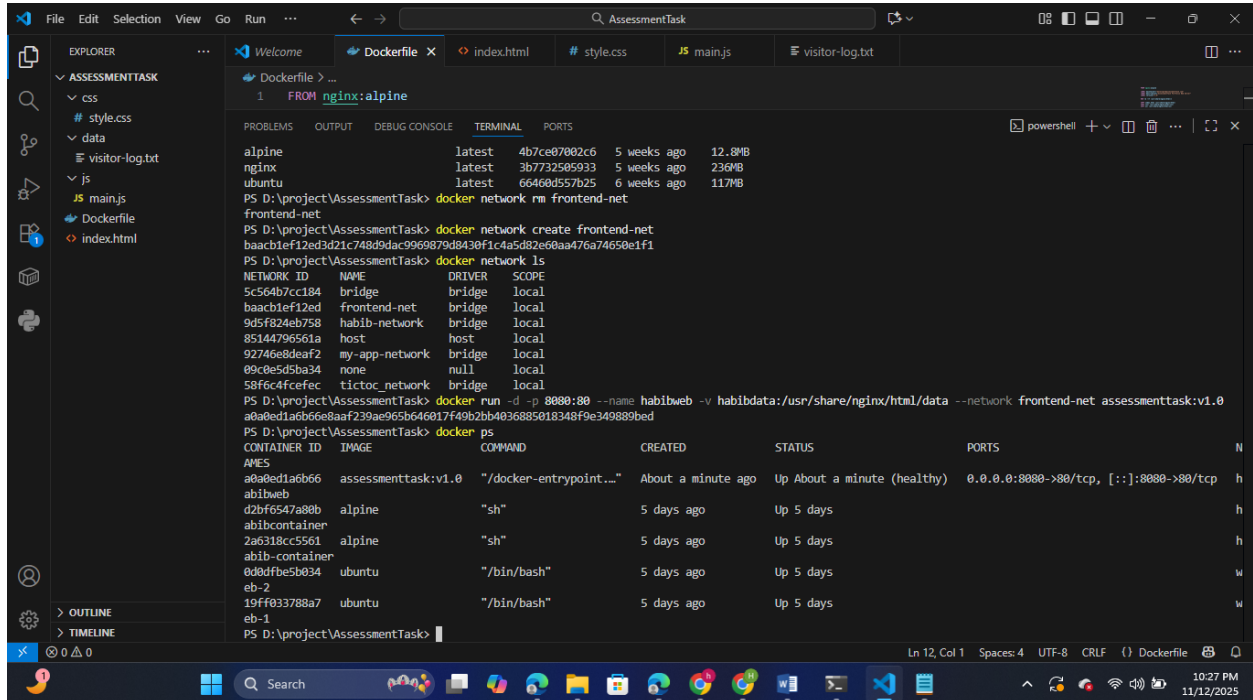
```
PS D:\project\AssessmentTask> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
assessmenttask       v1.0               7cc2f7fdc016       10 hours ago       80MB
docker-project       latest             ffc15477a170       11 days ago        1.64GB
habibdocker2/docker-project latest             86276d972dd1       11 days ago        1.64GB
tictoc.py            latest             6ba859cf8c64       2 weeks ago        1.61GB
tictoc               latest             6ba859cf8c64       2 weeks ago        1.61GB
<none>              <none>             458a480b5265       3 weeks ago        1.16GB
<none>              <none>             23c7e48f2a2f       3 weeks ago        1.16GB
ubuntuimage         latest             0493220f016f       3 weeks ago        1.16GB
habibdocker2/my-project v1                 04cb2a32dbec       4 weeks ago        274MB
habibdocker2/my-docker-project latest             e8e51acce94a       4 weeks ago        1.64GB
node                alpine             6ff78d6d45f2       4 weeks ago        237MB
alpine              latest             4b7ce07802c6       5 weeks ago        12.8MB
nginx               latest             3b7732505933       5 weeks ago        236MB
ubuntu              latest             6640ed557b25       6 weeks ago        117MB
PS D:\project\AssessmentTask> docker network rm frontend-net
PS D:\project\AssessmentTask> docker network create frontend-net
baacb1ef12ed3d21c748d9dac9969879d8430f1c4a5d82e60aa476a74650e1f1
PS D:\project\AssessmentTask> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
5c564b7cc184        bridge              bridge              local
baacb1ef12ed        frontend-net        bridge              local
9d5f804e4b758       habib-network       bridge              local
85144796561a        host                host                local
92746e8dea2f        my-app-network      bridge              local
09c0e5d5ba34        none                null                local
58f6c4fcefec        tictoc_network      bridge              local
PS D:\project\AssessmentTask>
```

Step 5: Container Deployment

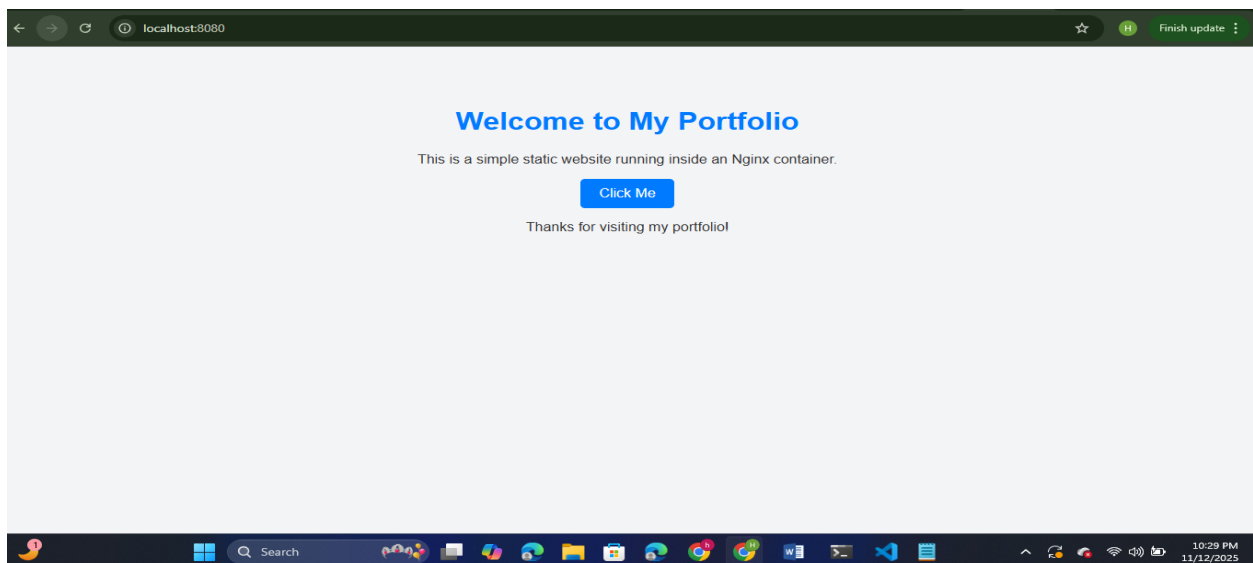
bash

```
docker run -d -p 8080:80 --name habibweb -v habib-data:/usr/share/nginx/html/data --network frontend-net assessmenttask:v1.0
```

Figure 19: Container Running

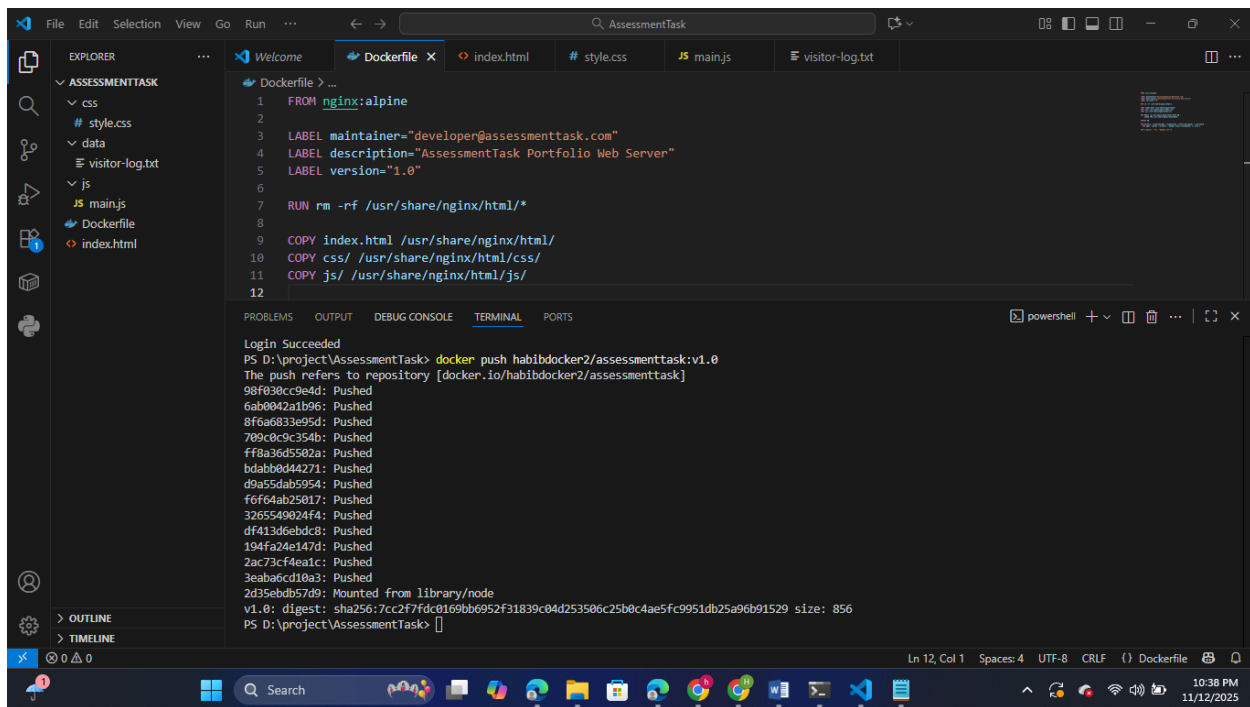


Step 6: Application Testing Figure 20:



Step 7: Publishing to DockerHub

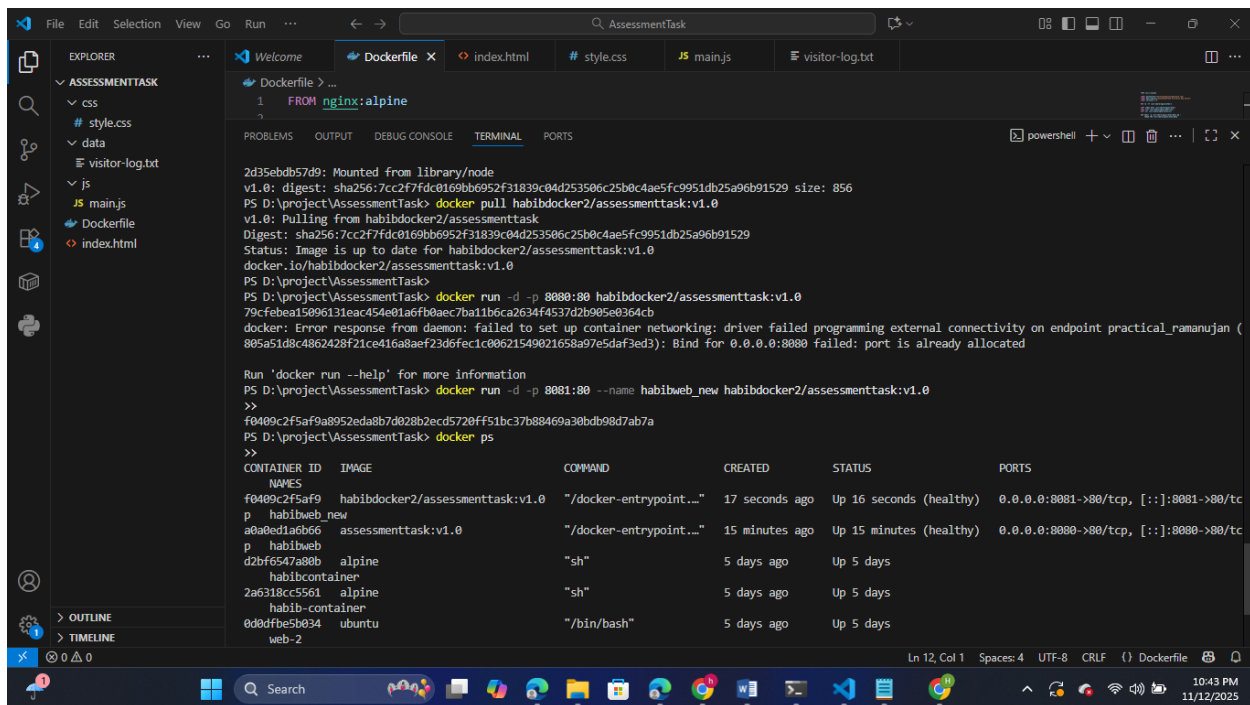
Figure 21: DockerHub Publication



```
1 FROM nginx:alpine
2
3 LABEL maintainer="developer@assessmenttask.com"
4 LABEL description="AssessmentTask Portfolio Web Server"
5 LABEL version="1.0"
6
7 RUN rm -rf /usr/share/nginx/html/*
8
9 COPY index.html /usr/share/nginx/html/
10 COPY css/ /usr/share/nginx/html/css/
11 COPY js/ /usr/share/nginx/html/js/
12
```

Problems OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Login Succeeded
PS D:\project\AssessmentTask> docker push habibdocker2/assessmenttask:v1.0
The push refers to repository [docker.io/habibdocker2/assessmenttask]
98f030c9e4d: Pushed
6ab0042a1b96: Pushed
8f6a6833e95d: Pushed
709c0c9c354b: Pushed
ff8a36d5502a: Pushed
bdabb0d44271: Pushed
d9a55dab5954: Pushed
f6f64ab25017: Pushed
3265549024f4: Pushed
df413d6ebdc8: Pushed
194fa24e147d: Pushed
2ac73c44ea1c: Pushed
3eaba6cd10a3: Pushed
2d35ebdb57d9: Mounted from library/node
v1.0: digest: sha256:7cc2f7fdc0169bb6952f31839c04d253506c25b0c4ae5fc9951db25a96b91529 size: 856
PS D:\project\AssessmentTask>
```



```
2d35ebdb57d9: Mounted from library/node
v1.0: digest: sha256:7cc2f7fdc0169bb6952f31839c04d253506c25b0c4ae5fc9951db25a96b91529 size: 856
PS D:\project\AssessmentTask> docker pull habibdocker2/assessmenttask:v1.0
v1.0: Pulling from habibdocker2/assessmenttask
Digest: sha256:7cc2f7fdc0169bb6952f31839c04d253506c25b0c4ae5fc9951db25a96b91529
Status: Image is up to date for habibdocker2/assessmenttask:v1.0
docker.io/habibdocker2/assessmenttask:v1.0
PS D:\project\AssessmentTask>
PS D:\project\AssessmentTask> docker run -d -p 8080:80 habibdocker2/assessmenttask:v1.0
79cfebea1596131eac454e01a6fb0a6c7ba11b6ca2634f4537d2b905e0364cb
docker: Error response from daemon: failed to set up container networking: driver failed programming external connectivity on endpoint practical_ramanujan (
805a5108c486248f21ce416a8aef23d6fec1c00621549021658a97e5daf3ed3): Bind for 0.0.0.0:8080 failed: port is already allocated

Run 'docker run --help' for more information
PS D:\project\AssessmentTask> docker run -d -p 8081:80 --name habibweb_new habibdocker2/assessmenttask:v1.0
f0409c2f5af9a8952eda8b7d028b2ecd5720ff51bc37b88469a30bd9b0d7ab7a
PS D:\project\AssessmentTask> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
f0409c2f5af9	habibdocker2/assessmenttask:v1.0	"/docker-entrypoint..."	17 seconds ago	Up 16 seconds (healthy)	0.0.0.0:8081->80/tcp, [::]:8081->80/tcp
a0a0ed1a6b66	assessmenttask:v1.0	"/docker-entrypoint..."	15 minutes ago	Up 15 minutes (healthy)	0.0.0.0:8080->80/tcp, [::]:8080->80/tcp
d2bf6547a80b	alpine	"sh"	5 days ago	Up 5 days	
2a6318cc5561	alpine	"sh"	5 days ago	Up 5 days	
0dddfbe5b034	ubuntu	"/bin/bash"	5 days ago	Up 5 days	

7.3 Results and Analysis

Application Deployment

- Docker image deployed successfully on port 8080
- Nginx web server hosted static HTML, JavaScript, and CSS files effectively.

- Because of HEALTHCHECK directive the health of the container stayed stable.

Persistence and Networking

- /usr/share/nginx/html/data directory was mounted through the portfolio data volume. Which makes data persistence possible.
- To replicate production style multi-container architecture, a custom Docker network named frontend net was used.

Image Publishing

- My Dockerhub username is habibdocker2, I uploaded the image successfully on DockerHub after tagging.
- Public image is reusable and accessible for any user, which improves portability and reusability in different environments.

Lessons Learned

- I got knowledge of refine and create Dockerfile, to deploy static websites.
- Understand the basics of volume persistence and network crates in Docker.
- Experience with container lifecycle management – build, run, deploy, and publish.
- Implemented health checks to ensure application uptime and container reliability.

8. Cloud Computing Concepts

Cloud computing is a significant evolution in computing paradigm, It provides IT resources as on demand, scalable, and device-independent services available via the Internet. It helps to deliver software, hardware, and data in the form of services, and promotes economies of scale in deployment and operation. This paradigm supports wide range of applications, such as Internet consumer services browsing, social networking, web email, and online shopping which today rely on cloud infrastructures. [124]

8.1 Load Balancing

Load balancing is an important strategy in cloud computing which prevents any single server from being overloaded and maximize scalability, availability, and performance. In cloud providers like AWS, Load balancing efficiently handles client requests by distributing traffic evenly across multiple servers or instances, which ensures high availability and fault tolerance [78]. AWS provide load balancing mainly through Elastic Load Balancing, which automatically distribute application traffic on multiple Amazon EC2 instances. ELB handles workloads by looking at real-time traffic changes, which keeps the system scalable and reliable. It provides different load balancing algorithms round robin, IP hash and least connections, So, that every application scenario can be suit. When traffic is low, ELB automatically adjusts the number of

active instances in conjunction with auto-scaling, which improves the usage of resources and reduce cost. [78] [79]

Elastic Load Balancer (ELB):

Types of Load Balancers

AWS Network Load Balancer works on Layer 4 provide high performance for TCP/UDP traffic, whereas Application Load Balancer handles HTTP/HTTPS on Layer 7 [80]. Gateway Load Balancer helps in deployment and manage firewalls and other virtual appliances [81].

Traffic Distribution

Round Robin is the default routing method of ALB, In which traffic is equally distributed among every target [82]. Least Outstanding Requests algorithm assigns requests to targets that have the fewest pending requests [83]. NLB use flow hash method, which routes traffic based on TCP sequence number, destination IP & port, source IP & port, and protocol. [82].

Health Checks

ELB only routes traffic to healthy instances. It regularly checks registered instances and marks their status as InService or OutOfService [84]. Load Balancer verify the condition of each instance at each health check interval by sending a request to the specified port, protocol, and path [84]. If Load Balancer thinks the instance is unhealthy, So, it blocks the traffic to that instance, and when it becomes healthy again, it starts getting traffic again [84].

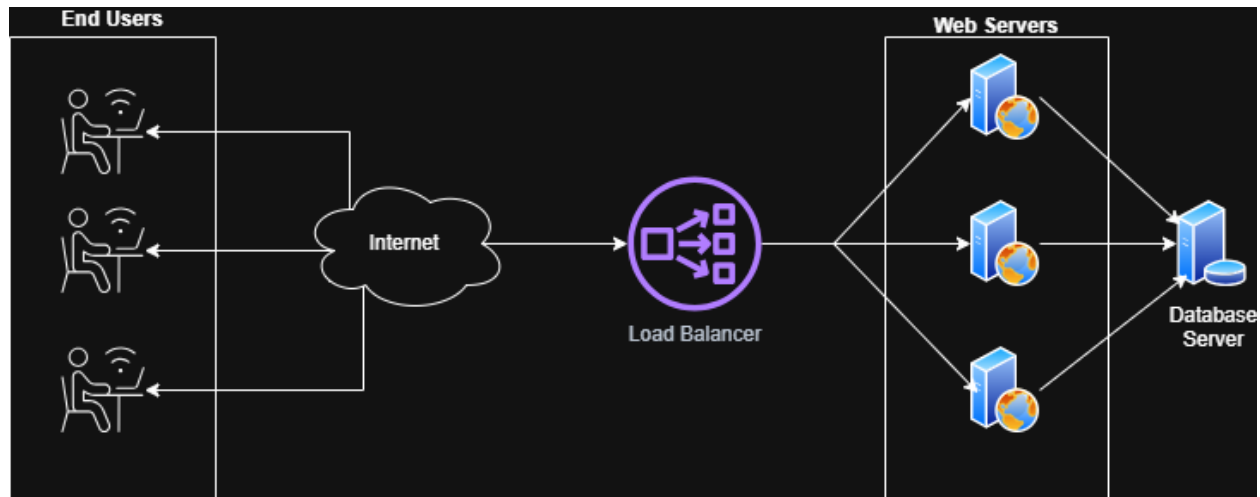
Automatic Failover

ELB ensures high availability across multiple Availability Zones within a Regional Boundary [85]. If any availability Zone got fail, then Network Load Balancer automatically switch to another healthy Availability Zones [85]. Whenever instances become unhealthy, Load balancer blocks traffic from being sent to them until their issues are fixed [84].

Auto Scaling Integration

When Elastic Load Balancing used with Auto Scaling groups, the instances that are started in the auto-scaling group are automatically registered with the load balancer, and the instances that are terminated are automatically deregister [86]. When load balancer is connected to an auto scaling group, this group can be configured to scale the number of instances according to demand, Elastic By using Load Balancing metrics (like ALB request count per target) [86]. Elastic Load Balancing health checks can be apply on Auto Scaling groups so that Amazon EC2 Auto Scaling detect unhealthy instances and replace them, on the bases of these extra health check [86].

Figure 22: Load Balancer Architecture



Azure Load Balancer:

Azure Load Balancer is designed to handle common cloud hosting issues, like server got crash because of multiple users request, or security issues in shared hosting environment. The method of Azure is to set up a load balancer that equally distribute incoming traffic on multiple virtual machines. In addition, this load balancer also works as a reverse proxy, this keeps the IP addresses of backend server protected from external threats, and the security of public cloud applications is also improved. [87]

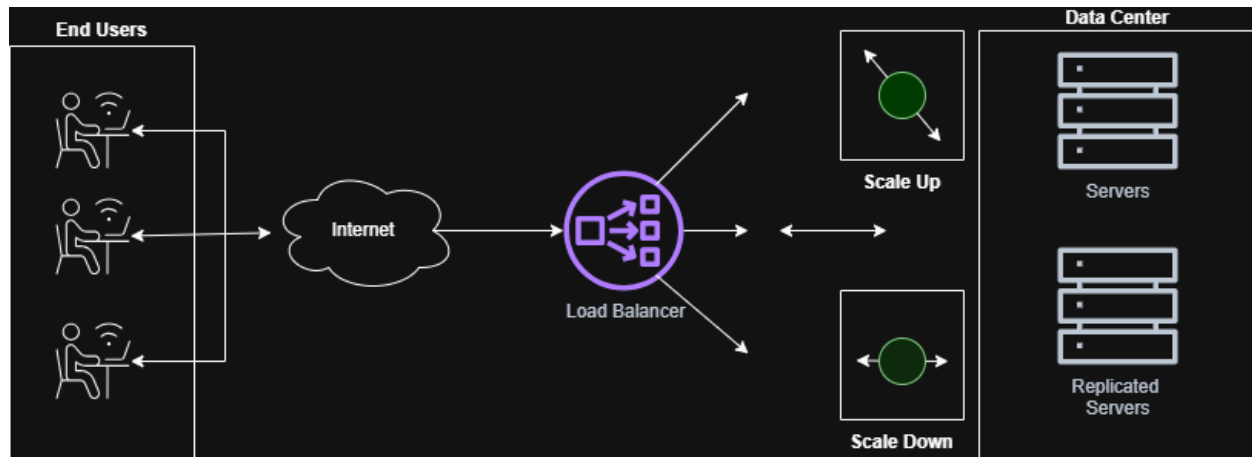
Google Cloud Load Balancing:

Google Cloud Platform focuses on dependability, load balancing scalability and optimal resource use. This provides horizontal auto-scaling for Kubernetes microservices, where components are distributed in different tiers and the scaling decision is based on message-queue length. This technique handles more than double request as compared to non scaling systems, which provides very good proof of GCP's fault tolerance and resource efficiency [88].

8.2 Scalability

In Cloud computing scalability is essential for service quality because it allows cloud services to efficiently handle fluctuating workloads without compromising performance. It is linked with elasticity and efficiency, which affects total performance and cost-effectiveness [89]. There are two methods to achieve auto scaling horizontal scaling (to add/remove instances) and vertical scaling (increase the capacity of existing instances), which have their own trade-offs according to workload and resource requirements [90][91]. Vertical scaling is fast and smooth but in a limited range, whereas horizontal scaling handles big changes but it is slow. Hybrid techniques do vertical scaling first and then horizontal scaling, to optimize resource consumption and flexibility [90]. Control theory helps to manage scaling to preserve stability despite delays [91]. Cloud providers like AWS support both scaling methods, and frameworks like SmartScale automate hybrid scaling, maximize resources and reduces reconfiguration costs [92].

Figure 23: Auto Scaling Process



8.3 Application Gateways

Application gateways controls data confidentiality, service reliability, and security in cloud computing. For example, an information gateway route data on the basis of confidentiality in SaaS framework, allows clients to specify data policies and it makes it possible to convert existing applications to SaaS [93].

AWS Application Load Balancer:

AWS Application Load Balancer (ALB) distributes traffic across multiple instances of the auto-scaling group, performs health check so unhealthy servers can be rejected and service reliability should be maintain [94].

Azure Application Gateway:

Azure Application Gateway is a layer-7 load balancer that distribute web traffic, provides geographic routing, and provides secure access through firewalls and managed load balancers [95][96].

Why Load Balancing and Scaling Are Complementary:

Load balancing and scaling are complementary in cloud computing because load balancing evenly distributes workloads, whereas auto scaling increase resources according to demand. Their integration improves performance. Minimize resource waste, and enable efficient response for traffic changes [97][98][99].

Benefits and Challenges of Combining All Three Technologies:

When scaling, application gateways and load balancers gets combine, improves resource utilization, reduces response times, and availability gets increase. Challenges includes potential bottlenecks, scaling and difficulty in balancing coordinate, and includes maintaining throughput for encrypted traffic [100][101][102][103].

Example:

Imagine an e-commerce website that frequently experiences slow response times and occasional downtime during high traffic events, especially when multiple users try to browse products, add items to the cart, or checkout simultaneously. These issues arise because a single server cannot handle sudden spikes in traffic, leading to overloaded resources and poor user experience. To solve this, the website is deployed on a cloud platform using load balancing, auto-scaling, and application gateways. Load balancers, such as AWS Application Load Balancer or Azure Application Gateway, distribute incoming user requests evenly across multiple server instances, preventing any single server from becoming a bottleneck. Auto-scaling dynamically adjusts the number of active servers based on real-time demand, adding more instances during traffic peaks and reducing them during low activity to optimize resource use and costs. Application gateways enhance security and data confidentiality by routing sensitive user data, such as payment information, through secure channels and firewalls. By combining these technologies, the website can handle high traffic smoothly, maintain fast response times, ensure continuous availability, and protect user data. This integrated solution transforms a previously overloaded, unreliable website into a scalable, high-performance, and secure online platform that can reliably serve users worldwide.

9. Serverless Functions vs Containerization

9.1 Driving Forces in Cloud Computing

1. Cost Efficiency

Compared to containerization, In cloud computing the biggest reason to adopt serverless functions is cost effectiveness. Serverless computing follow pay-by-usage model in which you only pay for invocation count, execution time, and memory usage. It is different from traditional cloud services, where fixed resources are allocate and billing periods are long. This serverless metering model provides very fine grained cost control, due to which this is more cost-effective for workloads that have variable or unpredictable demand [104].

2. Flexibility and Agility

Flexibility and agility are the most important factors in cloud computing, specially when comparing serverless activities with containerizations. Serverless computing's event-driven and highly scalable approach automatically scales applications on demand, which eliminates the usage of human infrastructure management and workloads can be adjusted very quickly. This model improve operational efficiency because the cost of idle resources is reduced and The time to bring new applications to market is also getting shorter. In this way organizations agility improves overall. [105] [106]

3. Scalability

Scalability is a basic driving force in cloud computing, whether it's serverless operations or containerizations, But the techniques and challenges of the two approaches are different.

Serverless computing, especially Function-as-a-Service (FaaS) model, allocate resources dynamically in response to event triggers, through which scaling gets smooth. This simplifies operations and pricing is also gets variable according to workload. The biggest benefit of this architecture is this that the applications scales automatically according to the demand without manual resource management, which is perfect for workloads where quick elasticity is required. [107] [108]

4. Innovation Speed

To understand innovation speed in cloud computing It is important to look at the operational models and research focus of serverless functions and containerizations. Containerization, is a lightweight virtualization technology, makes easy to create, delete and deploy containers by sharing operating system. This makes resource consumption more efficient as compared to traditional virtual machines. This approach led to the birth of container orchestration platforms like Kubernetes, There are constant updates coming in like auto-scaling, AI-driven resource optimization and workload scheduling. These developments show that the trend of innovation in container orchestration and management is continuously growing, which reduce the operational overhead and improves fault tolerance [109] [110].

5. Reliability and Availability

In cloud computing availability and reliability are both important for serverless functions and containerization, and they directly affect their adoption and working. Serverless computing systems, like Function-as-a-Service (FaaS), hide users from infrastructure like load balancing, containers and VMs which reduce operational complexity and It becomes easier to manage for microservices dynamically. In this way, serverless systems improve overall availability [111] [112]. But serverless environments also face some challenges, specially due infrastructure states provider cold, worm cold, worm states and container cold. Because of these states can cause multiple variations upto 15 times, that effect both reliability and response. [111]

9.2 Serverless Functions Explained

AWS Lambda:

AWS Lambda is a serverless service that runs code in response to events, without server setup and management's need. This automatically handles hardware resources, due to which, manual infrastructure operations are needed. That's why it is different from containerization, where resources are explicitly manage. [113] [114]

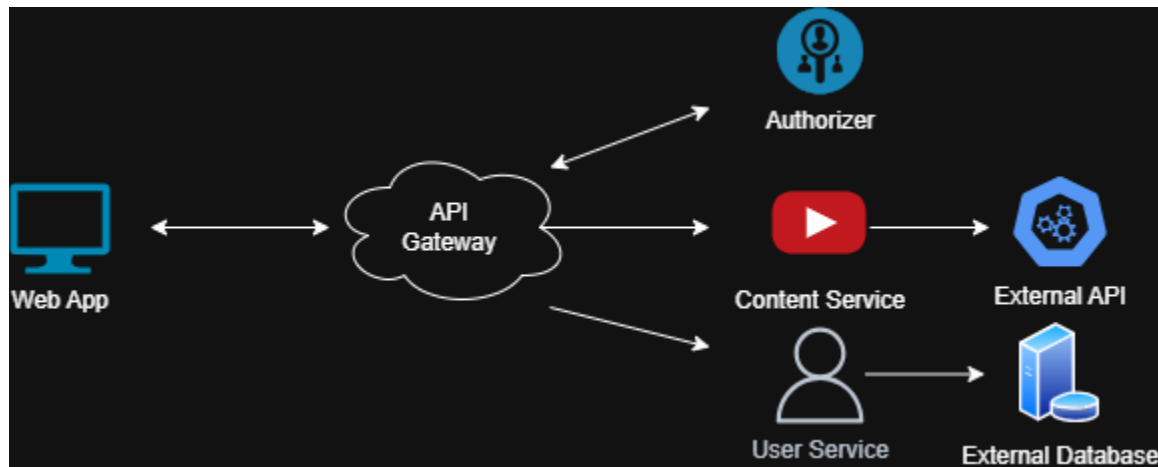
Azure Functions:

Azure Functions provides affordable and on-demand execution model in which developers can run event driven code without managing servers. This support language like Node.js, JavaScript and C#, and easily integrate with Azure services through triggers and bindings. [115][116][117]

How Serverless Works:

Serverless computing runs small, event triggered functions that automatically scale and only runs when needed because these functions are short lived and are randomly invoke, cloud platforms provisions resources very fast, and user only pay for the actual usage [118][119].

Figure 24:Serverless Architecture



9.3 Comparison with Containerization

Serverless is perfect for small or unpredictable workloads because this eliminates infrastructure management and scales automatically [120] [121]. Containers needs orchestration, but they gives more control and support full applications [120]. Serverless systems use pay-per-execution model, that is affordable for low usage, but there are also time limits and cold start issues [122][123]. In containers you only pay for running instances, but it runs stateful or long-running apps smoothly and no issue of cold start [121][122]. Vendor lock-in is high in serverless systems, while containers are portable in such different environments [120][123].

9.4 Use Case Analysis

When to Use Serverless:

Great for intermittent or event-driven activities, prototypes, and small microservices such as image processing, webhooks, and scheduled jobs [120][121].

When to Use Containers:

It is ideal for long-running services, complex dependencies, consistent traffic, or when full control is needed such as web apps, databases, and legacy systems [121][122].

Cost Analysis:

- ❑ Low traffic: Serverless is cheaper (no idle cost) [122].
- ❑ High constant traffic: Containers cost less due to fixed resource pricing [122].
- ❑ Spiky traffic: Serverless often wins due to elastic auto-scaling [120].

Flexibility Analysis:

Serverless has runtime limits, while containers allow any language/library and full customization. Hybrid designs combine both effectively [121][123].

Addressing Driving Forces:

- **Cost:** Serverless = no idle cost; Containers = predictable cost [122].
 - **Flexibility:** Serverless limited; Containers fully flexible [121].
 - **Scalability:** Both scale well using different mechanisms [120].
 - **Speed:** Serverless deploys fast; Containers avoid cold starts [123].
-

10. Video Demonstration

A comprehensive video demonstration has been prepared showing all practical implementations discussed in this report.

Video Link: <https://youtu.be/OzAriL4FoTA> **Duration:** 16:34

11. Conclusion

This research covers important cloud engineering concepts, includes containerization, virtualization and modern cloud computing models. Virtualizations provides strong isolation through hypervisors, but its startup time is slow and the resource overhead is high. In contrast, Docker based containerization provides lightweight and efficient deployment because containers share host OS kernel and maintain process level isolation. Practical work like creating docker images, set up networks and volumes, and push on DockerHub shows that how containerization allows fast, scalable, and reliable application deployments. Portfolio web server project demonstrated that how Dockerfile instructions, storage persistence, and controlled networking come together to create production-ready systems. The study also explored cloud scalability methods. where auto scaling, load balancer, and application gateways create robust and efficient cloud systems together. The comparison of serverless computing and containerizations told that serverless provides automatic scalability, simplicity, and cost-efficiency for intermittent

workloads, when containers provides flexibility, portability and continuous execution without cold starts. Hybrid approaches where for serverless event-driven tasks, for containers core services, and for virtual machines (VMs) legacy systems are use and more beneficial for organizations. Emerging technologies like micro VMs and serverless container runtime show that concepts will become even stronger in the future. Overall, the project combined theory and hands-on practice and show that how modern cloud technologies enable scalable, efficient and reliable application deployment, which prepare practitioners for rapidly evolving cloud landscape.

References

- [1] N. Jain and S. Choudhary, "Overview of virtualization in cloud computing," 2016. doi: 10.1109/CDAN.2016.7570950.
- [2] S. Bisman, T. Bhumika and K. Rakesh, "Cloud computing: Virtualization, service models, and deployment options," *I-manager's Journal on Cloud Computing*, vol. 11, no. 2, pp.28, 2024. doi: 10.26634/jcc.11.2.21197.
- [3] M. A. Khan and A. Sharma, "Deep Overview of Virtualization Technologies Environment and Cloud Security," 2023. doi: 10.1109/INOCON57975.2023.10101349.
- [4] https://csrc.nist.gov/glossary/term/full_virtualization
- [5] https://link.springer.com/chapter/10.1007/11942634_49
- [6] <https://www.sciencedirect.com/topics/computer-science/paravirtualization>
- [7] R. Ganesan, et al., "Empirical study of performance benefits of hardware assisted virtualization," 2013. doi: 10.1145/2522548.2522598.
- [8] H. N. Palit, et al., "Evaluating hardware-assisted virtualization for deploying HPC-as-a-service," 2013. doi: 10.1145/2465829.2465833.
- [9] R. Ansu, et al., "Efficient operating system level virtualization techniques for cloud resources," *IOP conference series.Materials Science and Engineering*, vol. 263, no. 4, pp.42002, 2017. doi: 10.1088/1757-899X/263/4/042002.
- [10] K. S. Ketha, G. Song and T. Zhu, "Analysis of Security in OS-Level Virtualization," 2025. doi: 10.48550/arxiv.2501.01334.
- [11] Chen-ming ZHENG, et al., "Design and implementation of a cloud server based on hardware virtualization," *工程科学学报*, vol. 44, no. 11, pp.1935–1945, 2022. doi: 10.13374/j.issn2095-9389.2022.01.12.005.
- [12] A. Traistaru and R. Ciobanu, "A Comprehensive Study on Live Migration of Virtual Machines and Techniques for Migration Optimization," 2025. doi: 10.1109/CSCS66924.2025.00047.
- [13] J. Brodtkin, "Bare-metal desktop hypervisors: A primer: Client virtualization holds long-term potential but is in its infancy," *Network World (Online)*, 2010.
- [14] J. Brodtkin, "A look at bare-metal hypervisor basics," *Network World*, vol. 27, no. 14, pp.1, 2010.
- [15] J. Brodtkin, "Citrix targets laptop security with new virtualization software: Automatic encryption, bare-metal hypervisor added to XenDesktop," *Network World (Online)*, 2010.
- [16] S. Qian, et al., "Computing and Storage Resources Allocation of UPF Based on Isolation in Private 5G Networks," 2022. doi: 10.1109/VTC2022-Spring54318.2022.9860618.
- [17] Y. K. Tun, et al., "Wireless Network Slicing: Generalized Kelly Mechanism-Based Resource Allocation," *J-SAC*, vol. 37, no. 8, pp.1794–1807, 2019. doi: 10.1109/JSAC.2019.2927100.

- [18] X. Lu, et al., "An elastic resource allocation algorithm enabling wireless network virtualization," *Wireless communications and mobile computing*, vol. 15, no. 2, pp.295–308, 2015. doi: 10.1002/wcm.2342.
- [19] V. R. Chintapalli, et al., "NFVPermit: Toward Ensuring Performance Isolation in NFV-Based Systems," *T-NSM*, vol. 20, no. 2, pp.1717–1732, 2023. doi: 10.1109/TNSM.2023.3278731.
- [20] M. Ciavotta, et al., "Architectural Design of Cloud Applications: A Performance-Aware Cost Minimization Approach," *TCC*, vol. 10, no. 3, pp.1571–1591, 2022. doi: 10.1109/TCC.2020.3015703.
- [21] M. Xu, et al., "Practice of Alibaba cloud on elastic resource provisioning for large-scale microservices cluster," *Software, practice & experience*, vol. 54, no. 1, pp.39–57, 2024. doi: 10.1002/spe.3271.
- [22] M. Tourchi Moghaddam and M. Møller Hansen, "Architecting Cloud Solutions for Improved Consulting Practices: Leveraging Simulation for Cost, Performance, and Energy Efficiency," 2023. doi: 10.1145/3603166.3632549.
- [23] W. Vogels, "Beyond Server Consolidation," *ACM queue*, vol. 6, no. 1, pp.20, 2008. doi: 10.1145/1348583.1348590.
- [24] C. Arango, R. Dernat and J. Sanabria, "Performance evaluation of container-based virtualization for high performance computing environments," *Revista UIS ingenierías (En línea)*, vol. 18, no. 4, pp.31–42, 2019. doi: 10.18273/revuin.v18n4-2019003.
- [25] D. Guster and O. F. Lee, "Enhancing the Disaster Recovery Plan Through Virtualization," *Journal of information technology research*, vol. 4, no. 4, pp.18–40, 2011. doi: 10.4018/jitr.2011100102.
- [26] M. Keller, et al., "eScience Cloud Infrastructure," 2011. doi: 10.1109/SEAA.2011.38.
- [27] J. Park, H. Kim and Y. Jeong, "Efficiency Sustainability Resource Visual Simulator for Clustered Desktop Virtualization Based on Cloud Infrastructure," *Sustainability*, vol. 6, no. 11, pp.8079–8091, 2014. doi: 10.3390/su6118079.
- [28] S. Wu, et al., "Vshadow: Promoting Physical Servers into Virtualization World," *Int J Parallel Prog*, vol. 45, no. 1, pp.45–66, 2017. doi: 10.1007/s10766-015-0385-2.
- [29] A. Traistaru and R. Ciobanu, "A Comprehensive Study on Live Migration of Virtual Machines and Techniques for Migration Optimization," 2025. doi: 10.1109/CSCS66924.2025.00047.
- [30] L. S. Subhash and R. Udayakumar, "Sunflower Whale Optimization Algorithm for Resource Allocation Strategy in Cloud Computing Platform," *Wireless Pers Commun*, vol. 116, no. 4, pp.3061–3080, 2021. doi: 10.1007/s11277-020-07835-9.
- [31] J. Xu, et al., "Autonomic resource management in virtualized data centers using fuzzy logic-based approaches," *Cluster Comput*, vol. 11, no. 3, pp.213–227, 2008. doi: 10.1007/s10586-008-0060-0.
- [32] W. Vogels, "Beyond Server Consolidation," *ACM queue*, vol. 6, no. 1, pp.20, 2008. doi: 10.1145/1348583.1348590.

- [33] B. Chatras and F. F. Ozog, "Network functions virtualization: the portability challenge," *NET-M*, vol. 30, no. 4, pp.4–8, 2016. doi: 10.1109/MNET.2016.7513857.
- [34] S. Shirinbab, "Performance Implications of Virtualization," 2019.
- [35] S. B. Verma, B. Pandey and B. Kumar Gupta, "Containerization and its Architectures: A Study," *Advances in distributed computing and artificial intelligence journal*, vol. 11, no. 4, pp.395–409, 2022. doi: 10.14201/adcaij.28351.
- [36] H. Sturley, et al., "Virtualization vs. Containerization, a Comparative Approach for Application Deployment in the Computing Continuum Focused on the Edge," *Future internet*, vol. 16, no. 11, pp.427, 2024. doi: 10.3390/fi16110427.
- [37] C. Pahl, "Containerization and the PaaS Cloud," *IEEE-CC*, vol. 2, no. 3, pp.24–31, 2015. doi: 10.1109/MCC.2015.51.
- [38] O. Bentaleb, et al., "Containerization technologies: taxonomies, applications and challenges," *J.Supercomput.*, vol. 78, no. 1, pp.1144–1181, 2022. doi: 10.1007/s11227-021-03914-1.
- [39] D. Silva, J. Rafael and A. Fonte, "Toward Optimal Virtualization: An Updated Comparative Analysis of Docker and LXD Container Technologies," *Computers (Basel)*, vol. 13, no. 4, pp.94, 2024. doi: 10.3390/computers13040094.
- [40] I. Mavridis and H. Karatza, "Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud," *Concurrency and computation*, vol. 35, no. 11, pp.n/a, 2023. doi: 10.1002/cpe.6365.
- [41] A. M, et al., "Comparison of Containerization and Virtualization in Cloud Architectures," 2021. doi: 10.1109/CONECCT52877.2021.9622668.
- [42] Z. Li, et al., "Lightweight and Holistic-Scalable Serverless Secure Container Runtime for High-Density Deployment and High-Concurrency Startup," *TC*, vol. 74, no. 8, pp.2621–2634, 2025. doi: 10.1109/TC.2025.3566912.
- [43] <https://docs.docker.com/get-started/docker-overview/>
- [44] <https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/>
- [45] <https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/>
- [46] <https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-registry/>
- [47] <https://docs.docker.com/build/concepts/dockerfile/>
- [48] <https://docs.docker.com/engine/manage-resources/labels/>
- [49] <https://docs.docker.com/reference/dockerfile/>
- [50] S. Agrawal and D. Singh, "Study Containerization Technologies like Docker and Kubernetes and their Role in Modern Cloud Deployments," 2024. doi: 10.1109/I2CT61223.2024.10543986.

- [51] M. J. Al Qausar, et al., "Software Metrics for Container-Based Applications: Systematic Literature Review," 2023. doi: 10.1109/ICIMTech59029.2023.10277948.
- [52] I. Hamzaoui, et al., "A Topical Review on Container-Based Cloud Revolution: Multi-Directional Challenges, and Future Trends," *SN COMPUT.SCI*, vol. 5, no. 4, pp.416, 2024. doi: 10.1007/s42979-024-02763-y.
- [53] S. S. M. and M. Karumudi, "Efficient Workload Portability and Optimized Resource Utilization using Containerization in a Multi-Cloud Environment," 2024. doi: 10.1109/ICDICI62993.2024.10810796.
- [54] F. Eyvazov, et al., "Beyond Containers: Orchestrating Microservices with Minikube, Kubernetes, Docker, and Compose for Seamless Deployment and Scalability," 2024. doi: 10.1109/ICRITO61523.2024.10522382.
- [55] V. Singh, et al., "Advantages of using Containerization Approach for Advanced Version Control System," 2022. doi: 10.1109/ICERECT56837.2022.10059738.
- [56] S. S. VP, CTO and a. M. Modernization Tribe Leader, "Ready...Set...Start Your Containers," *CIO*, 2022.
- [57] Z. Huang, et al., "FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container," 2019. doi: 10.1109/MSST.2019.00-18.
- [58] <https://docs.docker.com/reference/cli/docker/image/tag/>
- [59] <https://docs.docker.com/get-started/introduction/build-and-push-first-image/>
- [60] <https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/>
- [61] B. Dordevic, A. Stefanovic and V. Timcenko, "File system performance comparison in the case of Docker container-based virtualization with Volumes and Bind mounts," 2022. doi: 10.1109/TELFOR56187.2022.9983751.
- [62] J. S. Chelladurai, V. Singh and P. Raj, *Learning Docker: faster app development and deployment with Docker containers, second edition*, 2nd Ed. PACKT Publishing, 2017.
- [63] S. Hong, D. Li and X. Huang, "Database Docker persistence Framework based on Swarm and Ceph," 2019. doi: 10.1145/3341069.3342985.
- [64] <https://docs.docker.com/engine/storage/volumes/>
- [65] <https://docs.docker.com/engine/storage/bind-mounts/>
- [66] <https://docs.docker.com/engine/storage/tmpfs/>
- [67] *Networking overview.* <https://docs.docker.com/network/>

- [68] *Network tutorial (Standalone)*.
<https://docs.docker.com/network/network-tutorial-standalone/>
- [69] *Docker network drivers overview* (alternative to “Understanding Docker networking”).
<https://docs.docker.com/engine/network/drivers/>
- [70] *Bridge network driver*. <https://docs.docker.com/network/bridge/>
- [71] *Bridge network driver*. <https://docs.docker.com/network/bridge/>
- [72] *Host network driver*. <https://docs.docker.com/network/host/>
- [73] *Overlay network driver*. <https://docs.docker.com/network/overlay/>
- [74] *Macvlan network driver*. <https://docs.docker.com/network/macvlan/>
- [75] *None network driver*. <https://docs.docker.com/network/none/>
- [76] L. L. Mentz, W. J. Loch and G. P. Koslovski, "Comparative experimental analysis of Docker container networking drivers," 2020. doi: 10.1109/CloudNet51028.2020.9335811.
- [77] N. Verma, M. kashyap and A. Jha, "Extending Port Forwarding Concept to IOT," 2018. doi: 10.1109/ICACCCN.2018.8748430.
- [78] S. V. J and C. Bagyalakshmi, "An Improved Performance Analysis for AWS Cloud Load Balancing Using Weighted Round Robin (WRR) Technique," 2025. doi: 10.1109/ICOCT64433.2025.11118409.
- [79] S. Sharma, et al., "Advanced Deployment Strategies for Elastic Load Balancing in AWS: A Comprehensive Study on Multi-Tier Architecture Optimization," 2024. doi: 10.1109/I3CEET61722.2024.10993779.
- [80] IOD - The Content Engineers, "ELB vs. ALB vs. NLB: Choosing the Best AWS Load Balancer for Your Needs," 2022.
- [81] AWS, "Application, Network, and Gateway Load Balancing - Difference Between Load Balancing Types," 2024.
- [82] AWS, "How Elastic Load Balancing works."
- [83] AWS, "Application Load Balancer now supports Least Outstanding Requests algorithm for load balancing requests," 2019.
- [84] AWS, "Health checks for the instances for your Classic Load Balancer."
- [85] Stack Overflow, "Does AWS ELB handle HA by itself?"

- [86] AWS, "Use Elastic Load Balancing to distribute incoming application traffic in your Auto Scaling group."
- [87] T. Sheela, et al., "Secure Online Examination System on Azure Cloud with Load Balancer," 2021. doi: 10.1109/ICCCT53315.2021.9711889.
- [88] С. Теленик, В. Войналович and Д. Смаковський, "Web-application Architecture for the Kubernetes Cluster at the Google Cloud Platform with Horizontal Automatic Scaling," *Adaptivni sistemi avtomatičnogo upravlinnâ (Online)*, vol. 2, no. 39, 2021. doi: 10.20535/1560-8956.39.2021.247417.
- [89] K. Hwang, et al., "Cloud Performance Modeling with Benchmark Evaluation of Elastic Scaling Strategies," *TPDS*, vol. 27, no. 1, pp.130–143, 2016. doi: 10.1109/TPDS.2015.2398438.
- [90] M. Daraje and J. Shaikh, "Hybrid Resource Scaling for Dynamic Workload in Cloud Computing," 2021. doi: 10.1109/ICMNWC52512.2021.9688556.
- [91] V. Millnert and J. Eker, "HoloScale: horizontal and vertical scaling of cloud resources," 2020. doi: 10.1109/UCC48980.2020.00038.
- [92] S. Dutta, et al., "SmartScale: Automatic Application Scaling in Enterprise Clouds," 2012. doi: 10.1109/CLOUD.2012.12.
- [93] K. Nishikawa, K. Oki and A. Matsuo, "SaaS Application Framework Using Information Gateway Enabling Cloud Service with Data Confidentiality," 2012. doi: 10.1109/APSEC.2012.133.
- [94] S. Sharma, et al., "Advanced Deployment Strategies for Elastic Load Balancing in AWS: A Comprehensive Study on Multi-Tier Architecture Optimization," 2024. doi: 10.1109/I3CEET61722.2024.10993779.
- [95] S. Bisson, "Understanding Azure Virtual Networks," *InfoWorld.com*, 2021.
- [96] M. Waly, *Hands-on networking with Azure: build large-scale, real-world apps using Azure networking solutions*, 1st Ed. Birmingham: PACKT Publishing, 2018. doi: 10.0000/9781788993944
- [97] A. R and J. Agarkhed, "Evaluation of Auto Scaling and Load Balancing Features in Cloud," *International journal of computer applications*, vol. 117, no. 6, pp.30–33, 2015. doi: 10.5120/20561-2949.
- [98] S. K. Rout, et al., "A Dynamic Scalable Auto-Scaling Model as a Load Balancer in the Cloud Computing Environment," *EAI endorsed transactions on scalable information systems*, 2023. doi: 10.4108/eetsis.3356.
- [99] G. S. Sharvani and K. S. Resma, "An Auto-Scaling Approach to Load Balance Dynamic Workloads for Cloud Systems," *Turkish journal of computer and mathematics education*, vol. 12, no. 11, pp.515–531, 2021. doi: 10.17762/turcomat.v12i11.5915.
- [100] M. Rahman, S. Iqbal and J. Gao, "Load Balancer as a Service in Cloud Computing," 2014. doi: 10.1109/SOSE.2014.31.

- [101] G. J. Mirobi and L. Arockiam, "Dynamic Load Balancing Approach for Minimizing the Response Time Using An Enhanced Throttled Load Balancer in Cloud Computing," 2019. doi: 10.1109/ICSSIT46314.2019.8987845.
- [102] Y. Zhang, et al., "Cloud Load Balancers Need to Stay Off the Data Path," *TCC*, vol. 13, no. 3, pp.1078–1090, 2025. doi: 10.1109/TCC.2025.3595172.
- [103] G. Sridevi and Midhunchakkravarthy, "A Hybrid Multi-level Statistical Load Balancer-Based Parameters Estimation Model in Realtime Cloud Computing Environment," *Ingénierie des systèmes d'Information*, vol. 25, no. 6, pp.771–782, 2020. doi: 10.18280/isi.250607.
- [104] Z. Li, et al., "A Survey of Cost Optimization in Serverless Cloud Computing," *Journal of physics.Conference series*, vol. 1802, no. 3, pp.32070, 2021. doi: 10.1088/1742-6596/1802/3/032070.
- [105] A. K. Bayya, "Leveraging Advanced Cloud Computing Paradigms to Revolutionize Enterprise Application Infrastructure," *Asian Journal of Mathematics and Computer Research*, vol. 32, no. 1, pp.133–154, 2025. doi: 10.56557/ajomcor/2025/v32i19067.
- [106] A. P. Rajan, "A review on serverless architectures - function as a service (FaaS) in cloud computing," *Telkomnika*, vol. 18, no. 1, pp.530–537, 2020. doi: 10.12928/telkomnika.v18i1.12169.
- [107] C. Calavaro, et al., "Beyond Cloud: Serverless Functions in the Compute Continuum," *SN COMPUT.SCI*, vol. 6, no. 3, pp.194, 2025. doi: 10.1007/s42979-025-03699-7.
- [108] T. V. Sarathi, et al., "A Preliminary Study of Serverless Platforms for Latency Sensitive Applications," 2022. doi: 10.1109/CONECCT55679.2022.9865790.
- [109] S. K. Mondal, et al., "Kubernetes in IT administration and serverless computing: An empirical study and research challenges," *J.Supercomput.*, vol. 78, no. 2, pp.2937–2987, 2022. doi: 10.1007/s11227-021-03982-3.
- [110] V. Sharma, "AI-DRIVEN CLOUD INFRASTRUCTURE: ADVANCES IN KUBERNETES AND SERVERLESS COMPUTING," *International journal of advanced research in computer science*, vol. 16, no. 2, pp.65, 2025. doi: 10.26483/ijarcs.v16i02.7234.
- [111] W. Lloyd, et al., "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," 2018. doi: 10.1109/IC2E.2018.00039.
- [112] W. Lloyd, et al., "Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads," 2018. doi: 10.1109/UCC-Companion.2018.00056.
- [113] M. Klems, *AWS Lambda Quick Start Guide: Learn How to Build and Deploy Serverless Applications on AWS*, 1st Ed. Birmingham: Packt Publishing, Limited, 2018. doi: 10.0000/9781789340600
- [114] I. Bluemke and A. Zdanowski, "Evaluation of configurations of AWS Lambda functions," *International Journal of Electronics and Telecommunications*, vol. 71, no. 3, pp.1–9, 2025. doi: 10.24425/ijet.2025.153619.
- [115] S. Malik, *Azure Functions*. New York, NY, USA: Springer Nature, 2019.
- [116] S. Bisson, "Microsoft Azure Functions locks in on serverless computing," *InfoWorld.com*, 2017.
- [117] M. Yadav and M. Soni, *Learning Azure Functions*, 1st Ed. Packt Publishing, 2017.
- [118] Ustiugov, et al., "Benchmarking, analysis, and optimization of serverless function snapshots," 2021. doi: 10.1145/3445814.3446714.

- [119] P. Martínez Palacios, et al., "Towards Digital Transformation: Serverless Function Deployment Strategies," *Revista FI-UPTC*, vol. 30, no. 56, 2021. doi: 10.19053/01211129.v30.n56.2021.12776.
- 120] Cloudflare, "Serverless Computing vs. Containers."
- [121] Datadog, "Serverless vs. Containers: Which Is Right for You?"
- [122] M. Hamza, "Understanding Cost Dynamics of Serverless Computing," Springer, 2023.
- [123] DNSStuff, "Serverless vs Containers: Pros, Cons, and How to Choose."
- [124] M. D. Dikaiakos, et al., "Cloud Computing: Distributed Internet Computing for IT and Scientific Research," *MIC*, vol. 13, no. 5, pp.10–13, 2009. doi: 10.1109/MIC.2009.103.
- [125] D. G. Harkut, K. N. Kasat and S. A. Shah, Cloud Computing: Technology and Practices, IntechOpen, 2019. doi: 10.5772/intechopen.72088

