

**ACCA7027 – CLOUD ENGINEERING PRINCIPLES
AND PRACTICE**

**CLOUD COMPUTING,
VIRTUALISATION AND DOCKER
CONTAINERISATION**

Sayem Akther Talukdar (2502919)

Table Of Contents

Introduction.....	1
2. Principles of Virtualization	1
2.1 Virtualization of Shared Hardware	1
2.2 Hypervisors, Isolation And Resource Control	2
2.3 Advantages and Disadvantages of Virtualization.....	2
3. Principles and Benefits of Containerization	3
3.1 Containerization as Operating-System-Level Isolation	3
3.2 Virtual machines vs Containers.....	3
3.3 Docker Images and their corresponding Dockerfile and containers	4
3.4 Advantages, Disadvantages and Correct Usage.....	4
4. Practical Docker Implementation	4
4.1 Docker Installation and Verification	5
4.2 Project files and building the Webpage.....	5
4.3 Creating and Testing the docker image.....	7
4.4 Run and Test the Console Container.....	9
4.5 For publishing image on Docker Hub.....	11
4.6 Docker volume and persistent data for websites.....	13
4.7 Custom networks and communication between containers	17
4.8 Video Demonstration	20
5. Load Balancing, Scalability and Application Gateways.....	20
5.1 The Suggested Azure Context for a Demonstrated Web Application	20
5.2 Load Balancing	20
5.3 Scalability	21
5.4 Application Gateway.....	21
5.5 Why Load Balancing and Scaling Are Complementary	21
5.6 Advantages of Using the Three Technologies Combine.....	22
6. Driving Forces Behind the Expansion of Cloud Computing.....	22
6.1 Model and Resource-Efficiency Cost	22

6.2 Flexibility, Elasticity and Speed of Provisioning.....	23
6.3 Availability, Worldwide Reach and Focus on Functionality	23
6.4 Serverless Functions	23
6.5 Serverless Functions vs Containerization	24
7. Conclusion	25
Appendix A: Dockerfile.....	28
Appendix B: HTML Source.....	28

Figure 1: Docker Engine installation test and running service status.	5
Figure 2: Dockerfile used to create the custom Nginx web application image.	6
Figure 3: HTML webpage source code to be deployed via Nginx.	7
Figure 4: Successfully built Docker image for the custom cloud-web-platform.	8
Figure 5: Verification of the locally created cloud-web-platform:v1 Docker image.	9
Figure 6: Running the custom Nginx container with host-to-container port mapping.	10
Figure 7: Custom HTML webpage served successfully from the Nginx Docker container.	11
Figure 8: Publishing the custom cloud-web-platform:v1 image to Docker Hub.	12
Figure 9: Published cloud-web-platform:v1 image available in the Docker Hub repository.	13
Figure 10: Creation and verification of the platform-storage Docker volume.	14
Figure 11: Nginx container configured to use the platform-storage Docker volume.	15
Figure 12: Persistence of website data after removal of the original Docker container.	16
Figure 13: Reuse of persistent website data by a newly created Nginx container.	17
Figure 14: Creation and inspection of the custom platform-network bridge network.	18
Figure 15: Successful communication between the client and web server containers through the custom Docker network.	19
Figure 16: Final overview of the running Docker containers used in the practical implementation.	20

Introduction

The core of cloud computing is a powerful model that allows on-demand access to and use of processing, storage, networking, and software. Cloud computing is defined by the National Institute of Standards and Technology as on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service [\[1\]](#). Such characteristics indicate an evolution of computing from static, owner-occupied infrastructure toward provisioned and released resources that match changing needs. Not only does this shift change where applications will run, but it also changes how organizations plan capacity, monetize operations, and deliver services.

Within this model, virtualization and containerization are the underlying enabling technologies. Virtualization segments physical hardware into independent Virtual Machines, each having its own guest OS. Containerization wraps applications and their dependencies into isolated processes that share the host operating-system kernel. Hence, both approaches are related, but not interchangeable. VMs offer strong OS separation, but containers usually start faster and provide smaller deployment units that use fewer host resources. These technologies are further boosted when combined with elastic capacity, automated provisioning, and billing based on usage [\[2\]](#), [\[3\]](#)

This paper discusses the principles of virtualization and containerization, assessing them during a hands-on Docker project. In the practical part, I used a Nginx image... to host a customized HTML page. The project consists of building a Dockerfile, creating the image cloud-web-platform:v1, running the web-platform container, pushing it to Docker hub, using the platform-storage volume for persistent storage across containers, and using the platform-network bridge for communication between containers. The report goes on to explore load balancing, scalability, and application gateways using Microsoft Azure for the cloud-based equivalent of this web service. Lastly, it looks at the major drivers of cloud growth and highlights serverless functions against container-based deployment.

2. Principles of Virtualization

2.1 Virtualization of Shared Hardware

Virtualization generates a software-defined representation of physical computing resources. In full virtualization, a hypervisor presents virtual hardware to one or more operating systems and their applications [\[4\]](#). This means that an entire server can contain several virtual machines, allocated with virtual processors, memory, disks, and network interfaces from the common underlying hardware. The workloads running on the host are isolated; each virtual machine acts as a separate computer.

The practical benefit of this shared hardware virtualization is that it enables better use. If you have a single physical server for a small application, much of your processing and memory may go untapped. If an organization can run many workloads on a host, it reduces the physical systems required by hosting more workloads per physical machine and utilizing its current

capacity better. This method also allows fast deployment, since a virtual machine can be provisioned from a template instead of being constructed as a brand-new physical server. This supports the resource pooling and elasticity features of cloud computing [\[1\]](#).

Nonetheless, resource sharing must be carefully coordinated. More than one virtual machine may compete for CPU time, memory bandwidth, storage input/output, or network capacity. Performance becomes unpredictable if capacity is overcommitted without sufficient monitoring. Physical host failures can, down the line, impact every VM on that host. Thus, virtualization enhances flexibility, but resilience still requires tedious measures such as redundant hosts, shared storage, backups, and workload migration.

2.2 Hypervisors, Isolation And Resource Control

Hypervisors are software layers that manage and create virtual machines. Type 1 hypervisors run directly on the server hardware, and Type 2 hypervisors run the host operating system as an application. Type 1 hypervisors are often used in data centers because they reduce the number of software layers in between and therefore provide you with direct control of physical resources. Type 2 designs are still beneficial for desktop testing, training, and development because you can deploy them on a typical workstation.

Isolation is one of the most powerful reasons for using VMs. Different guest operating systems can run on the same host in separate virtual machines, which allows applications with different operating-system requirements or security boundaries to coexist. Be that as it may, the hypervisor itself becomes an enormous security target. NIST (the National Institute of Standards and Technology) guidance calls for hardening hypervisor management interfaces, limiting administration, applying patches, and preserving separation between hosted systems [\[4\]](#), [\[5\]](#). This is because the hypervisor oversees many workloads, so if even a single flaw is introduced in this layer, it could have knock-on effects further afield than could an exploit within a single guest.

2.3 Advantages and Disadvantages of Virtualization

Virtualization provides four main benefits. For one, it can help lower the related requirements for hardware, power, cooling and physical space. Second, virtual machine provisioning and cloning can take place rapidly, which further helps in development, testing, and recovery. Third, every virtual machine that runs under a hypervisor can run a completely different operating system as well as a different software stack. Fourth, a configured physical server can be backed up or migrated more systematically than equivalent virtual disks. These characteristics are why virtual machines continue to be at the heart of infrastructure-as-a-service clouds.

The main limitation is overhead. A virtual machine has an entire guest OS, which consumes more battery and RAM than a container, which only needs the host kernel. It usually has a longer boot time as well. As the number of virtual machines grows, it can lead to more complex management with image maintenance, patching, licensing, monitoring, and security. Consequently, virtualization makes more sense when there is a greater need for operating-system isolation or

legacy compatibility, or where strong workload separation is required rather than the smallest possible deployment footprint.

3. Principles and Benefits of Containerization

3.1 Containerization as Operating-System-Level Isolation

Containerization compiles an application along with the files, libraries and configuration needed to run it but shares its host operating system kernel. Containers isolate application processes and file systems while sharing the host operating-system kernel, eliminating the need to include a full guest operating system in each deployment unit. Containerization is a lightweight virtualization technology that Pahl defines as the approach to pack, ship, and manage applications in platform-as-a-service [6].

This kind of design usually yields smaller images and quicker load times compared with complete VMs. It has also been shown that many workloads can achieve native or near-native performance when using Linux containers in comparison to virtual machines. However, image-intensive applications still need careful configuration [7]. So, the advantage is not that containers have less overhead to tolerate, but rather that their isolation model tends toward lower Resource duplication than a full guest.

Portability is another major benefit. Docker images offer a repeatable application package you can move between any environment that supports it. According to Merkel's talks, Docker is used to solve the dependency inconsistencies across systems for development, testing, and deployment [8]. In practice today, the image is the configuration of the application environment, and the container is a running instance started from that image. Docker has a concept of volumes and networks for expressing volume storage and communication as separate managed objects [9].

3.2 Virtual machines vs Containers

Containers and Virtual Machines are solutions to different isolation problems. Virtual machines isolate OSs, while containers isolate applications and processes with a shared kernel. That distinction determines density, start-up time, portability, and security. Summarized in table 1, the differences most relevant to our project are outlined here.

Table 1: Comparison of virtual machines and containers

Criterion	Virtual Machine	Container
Operating system	Includes a complete guest operating system	Shares the host kernel
Deployment size	Usually larger because the guest OS is included	Usually smaller because application layers are packaged
Start-up	Requires guest operating-system boot	Starts as an isolated process
Isolation boundary	Between guest operating systems	Between processes and namespaces

Criterion	Virtual Machine	Container
Typical strength	Strong OS separation and legacy compatibility	Rapid, repeatable and efficient application deployment
Main limitation	Higher resource and management overhead	Greater dependency on host-kernel security and compatibility

3.3 Docker Images and their corresponding Dockerfile and containers

Docker uses a client-server architecture in which commands are issued via the Docker client and processed by the Docker daemon. The daemon builds images, creates containers, volumes, and networks [9]. A Dockerfile gives us a repeatable way to specify how to build an image. Common directives in a Dockerfile include FROM to choose a base image, COPY to copy application files into the image, LABEL to store metadata, and EXPOSE to document the application port [10]. This minimizes manual variation since the same Dockerfile can be used to rebuild the image for applications.

Image: A template that cannot be updated, which is a combination of layer(s). Container: An image plus the writable runtime layer. The distinction is important. Rebuilding or recreating a container does not change the image from which it was created. - Conversely, all data written only to a container writable layer can be lost by removing the container. Your volumes or another external storage should provide persistent storage for application data.

3.4 Advantages, Disadvantages and Correct Usage

Containerization led to Fast Deployment, Consistency and Efficient Scale. You can start multiple containers based on the same image (from a service perspective, this may be useful if a load balancer must balance traffic between two or more instances of your image). Images are also versioned and can be published (eg: in a registry) for simple reuse across systems. This is one of the reasons containers work well with web services, application programming interfaces, pipelines, and modular cloud applications.

This approach also introduces risks. The container image may contain vulnerable or outdated software, and the host kernel is on the surface of the attack. Images should be trusted through trusted bases, updated regularly, and executed with the necessary privileges when many containers need to be scheduled, monitored, and networked; operational complexity increases. Containers are best when application portability and fast deployment are needed, but that does not eliminate the requirement for security, governance, persistent storage, and capacity planning.

4. Practical Docker Implementation

This section describes the practical implementation used to create a containerized Nginx service running on Ubuntu. The sequence covered the full application life cycle needed for the assessment: Docker verification, creating a Dockerfile and application, building a Docker image, executing a

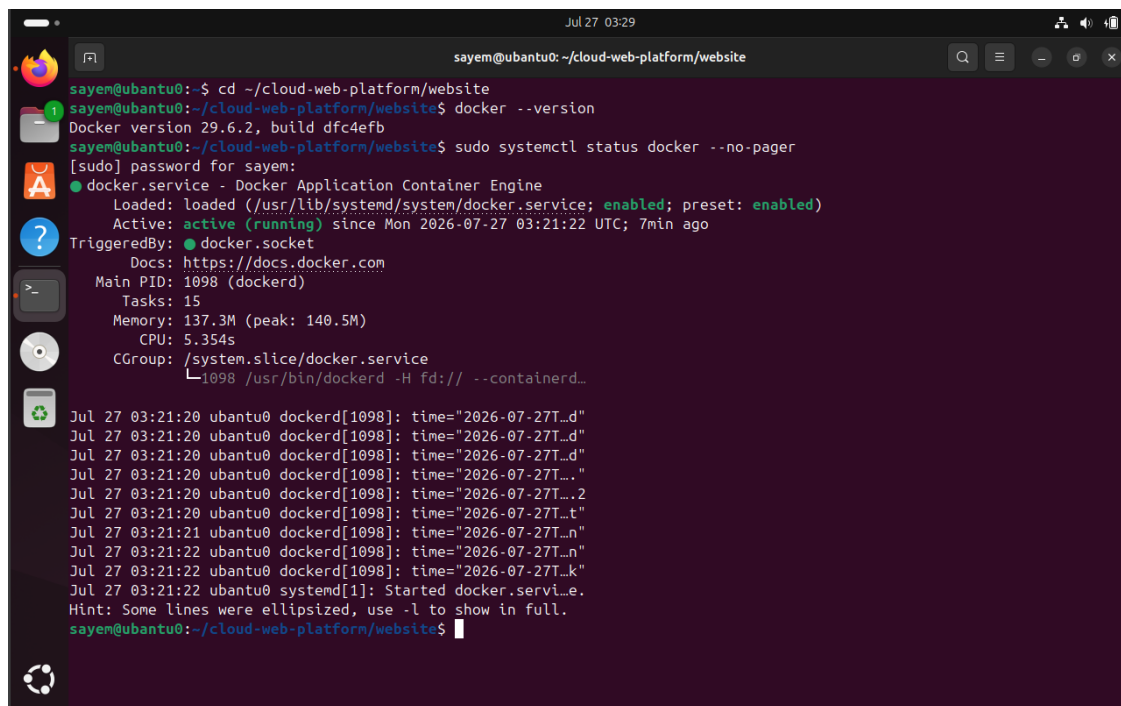
container, running browser tests using selenium, uploading to the registry, persisting volumes, and connecting containers. Each screenshot appears in only one place and is always found directly before its figure.

4.1 Docker Installation and Verification

The latest version of Docker was installed from the Ubuntu package repository, and the service was started and enabled. The installation involved updating the package repository, installing Docker, enabling the Docker service, and verifying that it was running correctly, `systemctl start docker`, and `io`. Finally, I verified the installation via Docker version, service status, and the official hello-world container. I verified that Docker version 29.6.2 was available, as shown in Figure 1, and that the service was active and running.

```
docker --version
```

```
sudo systemctl status docker --no-pager
```

A terminal window titled 'sayem@ubuntu0: ~/cloud-web-platform/website' showing the execution of two commands. The first command, 'docker --version', returns 'Docker version 29.6.2, build dfc4efb'. The second command, 'sudo systemctl status docker --no-pager', shows the status of the Docker service as 'active (running)' since Mon 2026-07-27 03:21:22 UTC, 7min ago. It also lists details like 'Main PID: 1098 (dockerd)', 'Tasks: 15', 'Memory: 137.3M (peak: 140.5M)', and 'CPU: 5.354s'. A series of log messages at the bottom show the service starting and running successfully.

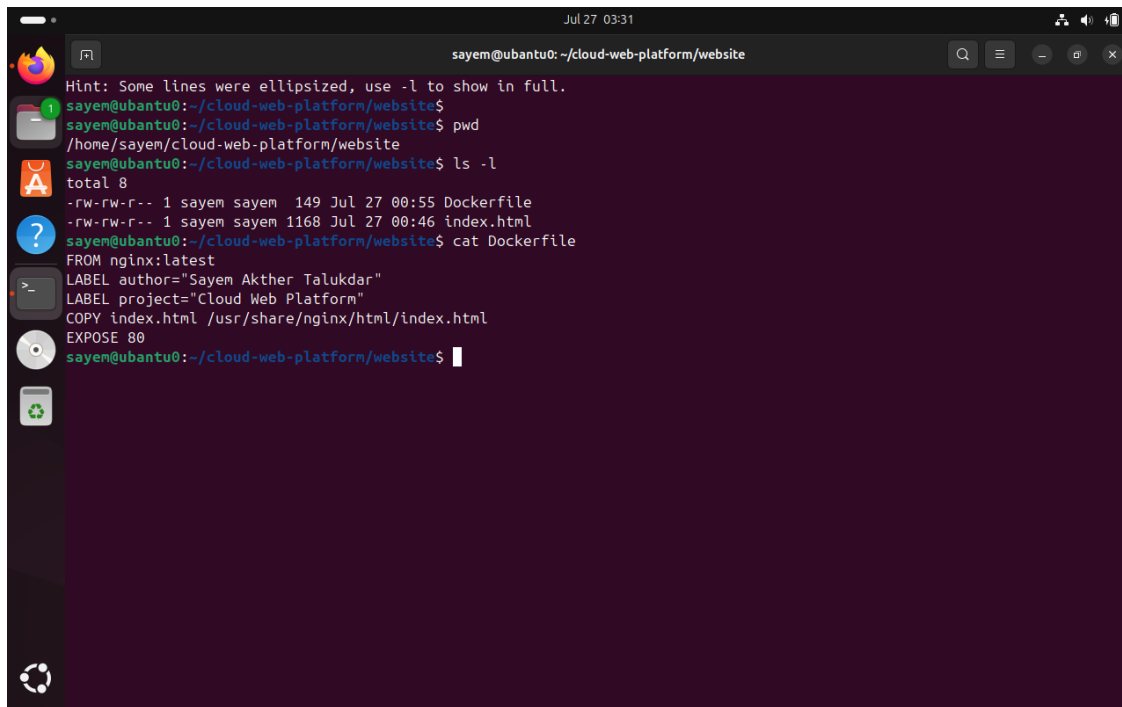
```
sayem@ubuntu0:~$ cd ~/cloud-web-platform/website
sayem@ubuntu0:~/cloud-web-platform/website$ docker --version
Docker version 29.6.2, build dfc4efb
sayem@ubuntu0:~/cloud-web-platform/website$ sudo systemctl status docker --no-pager
[sudo] password for sayem:
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; preset: enabled)
   Active: active (running) since Mon 2026-07-27 03:21:22 UTC; 7min ago
     TriggeredBy: ● docker.socket
     Docs: https://docs.docker.com
    Main PID: 1098 (dockerd)
      Tasks: 15
     Memory: 137.3M (peak: 140.5M)
        CPU: 5.354s
     CGroup: /system.slice/docker.service
            └─1098 /usr/bin/dockerd -H fd:// --containerd...

Jul 27 03:21:20 ubuntu0 dockerd[1098]: time="2026-07-27T...d"
Jul 27 03:21:20 ubuntu0 dockerd[1098]: time="2026-07-27T...d"
Jul 27 03:21:20 ubuntu0 dockerd[1098]: time="2026-07-27T...d"
Jul 27 03:21:20 ubuntu0 dockerd[1098]: time="2026-07-27T...2"
Jul 27 03:21:20 ubuntu0 dockerd[1098]: time="2026-07-27T...t"
Jul 27 03:21:21 ubuntu0 dockerd[1098]: time="2026-07-27T...n"
Jul 27 03:21:22 ubuntu0 dockerd[1098]: time="2026-07-27T...n"
Jul 27 03:21:22 ubuntu0 dockerd[1098]: time="2026-07-27T...k"
Jul 27 03:21:22 ubuntu0 systemd[1]: Started docker.servi.e.
Hint: Some lines were ellipsized, use -l to show in full.
sayem@ubuntu0:~/cloud-web-platform/website$
```

Figure 1: Docker Engine installation test and running service status.

4.2 Project files and building the Webpage

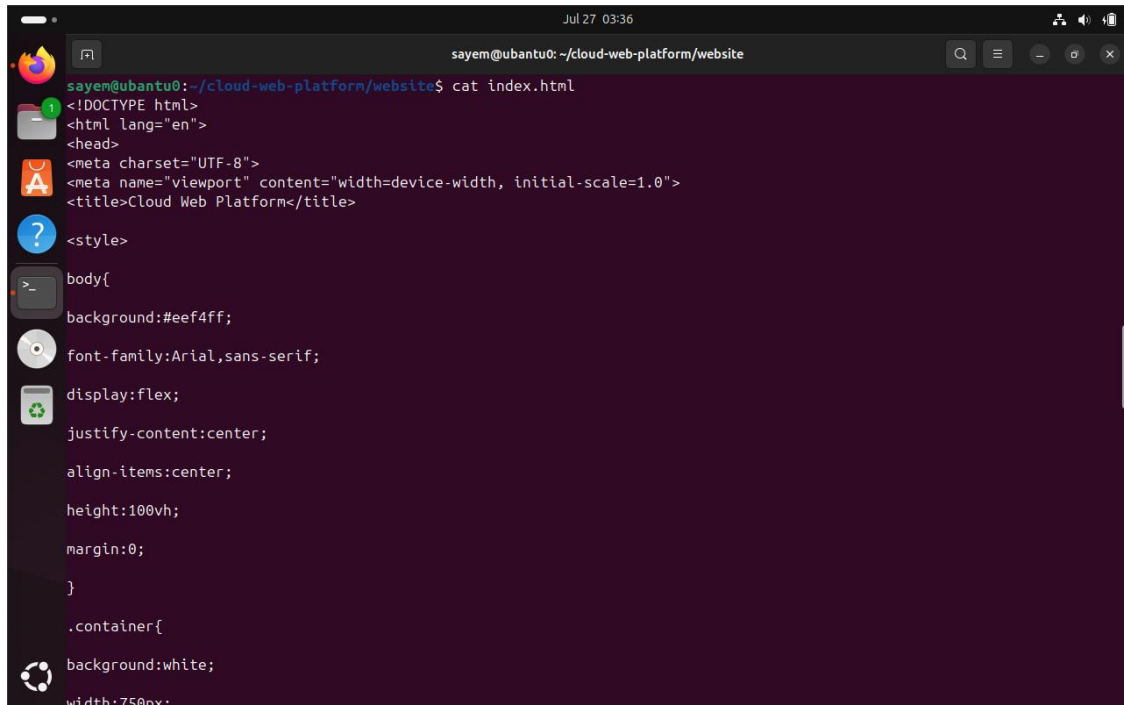
A project directory named `cloud-web-platform` was created, followed by a `website` subdirectory. The HTML file and Dockerfile are stored together in this directory so that the Docker build context contains both required files. Figure 2 shows the project files and the completed Dockerfile. The file uses `nginx:latest` as the base image, records author and project labels, copies `index.html` into the standard Nginx web directory, and exposes port 80. These instructions match the Dockerfile construction model described in the official documentation [\[10\]](#).

A terminal window titled 'sayem@ubuntu0: ~/cloud-web-platform/website' with a dark purple background. The terminal shows a series of commands and their outputs. The first command is 'pwd', which returns '/home/sayem/cloud-web-platform/website'. The second command is 'ls -l', which shows two files: 'Dockerfile' (149 bytes, dated Jul 27 00:55) and 'index.html' (1168 bytes, dated Jul 27 00:46). The third command is 'cat Dockerfile', which displays the contents of the Dockerfile. The Dockerfile starts with 'FROM nginx:latest', followed by two labels: 'LABEL author="Sayem Akther Talukdar"' and 'LABEL project="Cloud Web Platform"'. Then, it has a 'COPY' instruction: 'COPY index.html /usr/share/nginx/html/index.html', and finally an 'EXPOSE' instruction: 'EXPOSE 80'. The terminal ends with a prompt 'sayem@ubuntu0:~/cloud-web-platform/website\$' and a cursor.

```
Jul 27 03:31
sayem@ubuntu0: ~/cloud-web-platform/website
Hint: Some lines were ellipsized, use -l to show in full.
sayem@ubuntu0:~/cloud-web-platform/website$ pwd
/home/sayem/cloud-web-platform/website
sayem@ubuntu0:~/cloud-web-platform/website$ ls -l
total 8
-rw-rw-r-- 1 sayem sayem 149 Jul 27 00:55 Dockerfile
-rw-rw-r-- 1 sayem sayem 1168 Jul 27 00:46 index.html
sayem@ubuntu0:~/cloud-web-platform/website$ cat Dockerfile
FROM nginx:latest
LABEL author="Sayem Akther Talukdar"
LABEL project="Cloud Web Platform"
COPY index.html /usr/share/nginx/html/index.html
EXPOSE 80
sayem@ubuntu0:~/cloud-web-platform/website$
```

Figure 2: Dockerfile used to create the custom Nginx web application image.

The page is built as index.html with inline CSS styling. The viewable output shows the module name, student, and university, and indicates that the page runs in a Nginx Docker container. The source is validated with cat index.html. This was left as is, unchanged by the practical work, and used as a file whose contents are copied into the image for our application.

A terminal window on a Linux system (Ubuntu) showing the contents of a file named index.html. The window title is 'sayem@ubuntu0: ~/cloud-web-platform/website'. The command 'cat index.html' has been executed. The output is HTML code for a web page titled 'Cloud Web Platform'. It includes a DOCTYPE declaration, a head section with meta tags for charset (UTF-8) and viewport (width=device-width, initial-scale=1.0), and a body section with CSS styles. The body styles include a light blue background, Arial font, flex display, and centered content. A container class is also defined with a white background and a width of 750px.

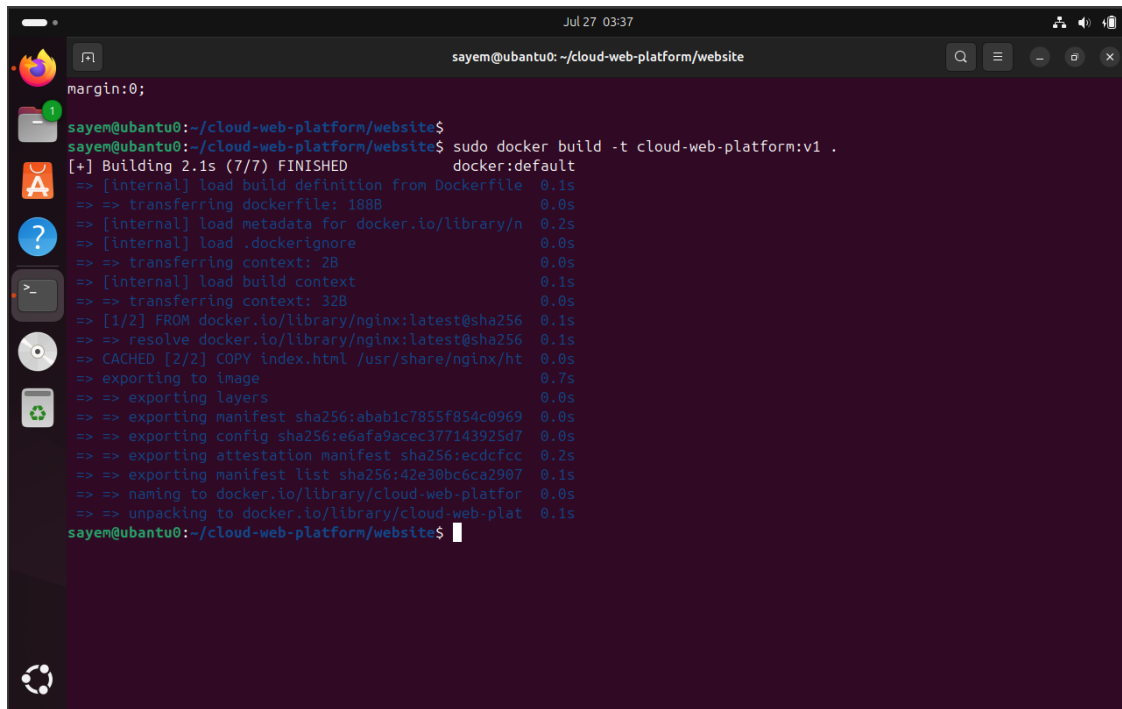
```
sayem@ubuntu0:~/cloud-web-platform/website$ cat index.html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Cloud Web Platform</title>
<style>
body{
background:#eef4ff;
font-family:Arial,sans-serif;
display:flex;
justify-content:center;
align-items:center;
height:100vh;
margin:0;
}
.container{
background:white;
width:750px;
```

Figure 3: HTML webpage source code to be deployed via Nginx.

4.3 Creating and Testing the docker image

The current directory changes also built and pushed the custom image, tagged cloud-web-platform:v1. This tag identifies both the project and the image's earliest release. Docker reads the Dockerfile, pulls down Nginx base layers, and copies your custom web page into the image. The successful build is recorded in Figure 4.

```
sudo docker build -t cloud-web-platform:v1 .
```

A terminal window on an Ubuntu system showing the successful build of a Docker image. The window title is 'sayem@ubuntu0: ~/cloud-web-platform/website'. The user has executed 'sudo docker build -t cloud-web-platform:v1 .' and the output shows the build process from Dockerfile to the final image export. The terminal output is as follows:

```
margin:0;
sayem@ubuntu0:~/cloud-web-platform/website$
sayem@ubuntu0:~/cloud-web-platform/website$ sudo docker build -t cloud-web-platform:v1 .
[+] Building 2.1s (7/7) FINISHED    docker:default
=> [internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 188B 0.0s
=> [internal] load metadata for docker.io/library/nginx:latest@sha256:42e30bc6ca2987 0.2s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load build context 0.1s
=> => transferring context: 32B 0.0s
=> [1/2] FROM docker.io/library/nginx:latest@sha256:42e30bc6ca2987 0.1s
=> => resolve docker.io/library/nginx:latest@sha256:42e30bc6ca2987 0.1s
=> CACHED [2/2] COPY index.html /usr/share/nginx/html 0.0s
=> exporting to image 0.7s
=> => exporting layers 0.0s
=> => exporting manifest sha256:abab1c7855f854c0969 0.0s
=> => exporting config sha256:e6afa9acec377143925d7 0.0s
=> => exporting attestation manifest sha256:ecdcfcc 0.2s
=> => exporting manifest list sha256:42e30bc6ca2987 0.1s
=> => naming to docker.io/library/cloud-web-platform:v1 0.0s
=> => unpacking to docker.io/library/cloud-web-platform:v1 0.1s
sayem@ubuntu0:~/cloud-web-platform/website$
```

Figure 4: Successfully built Docker image for the custom cloud-web-platform.

To verify that the new image existed, I filtered the local image list after the build. The output for cloud-web-platform:v1 is shown in Figure 5, including the image tag and local disk utilization. It validates that images are built successfully, as opposed to later running containers.

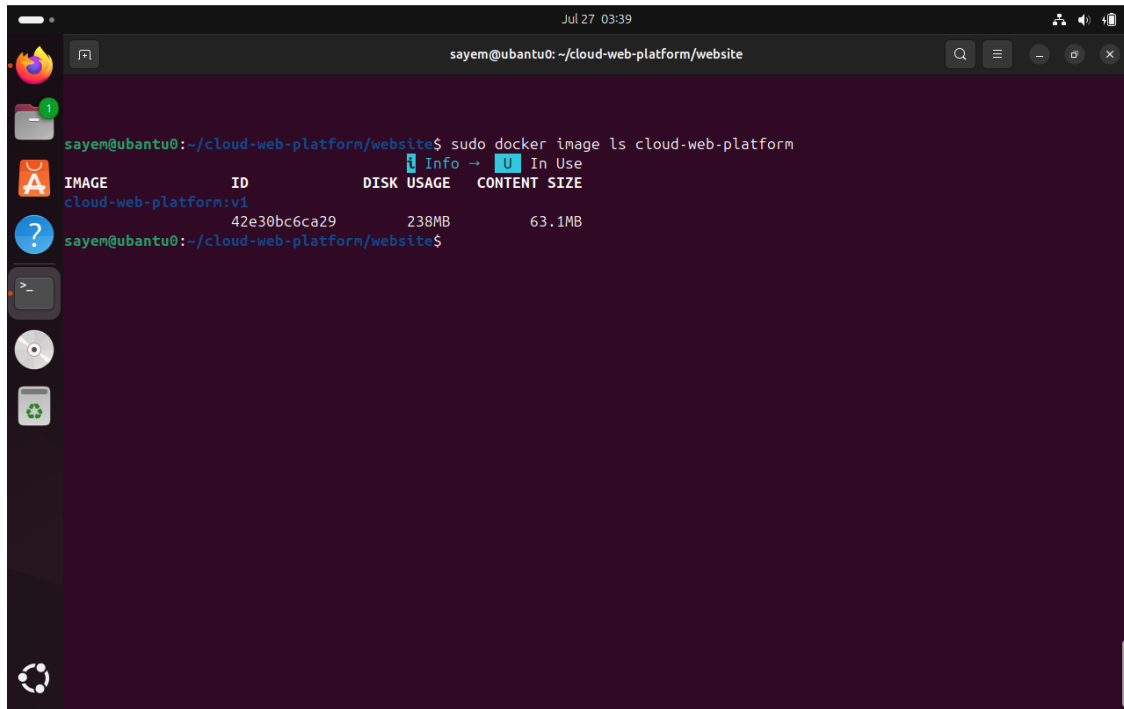
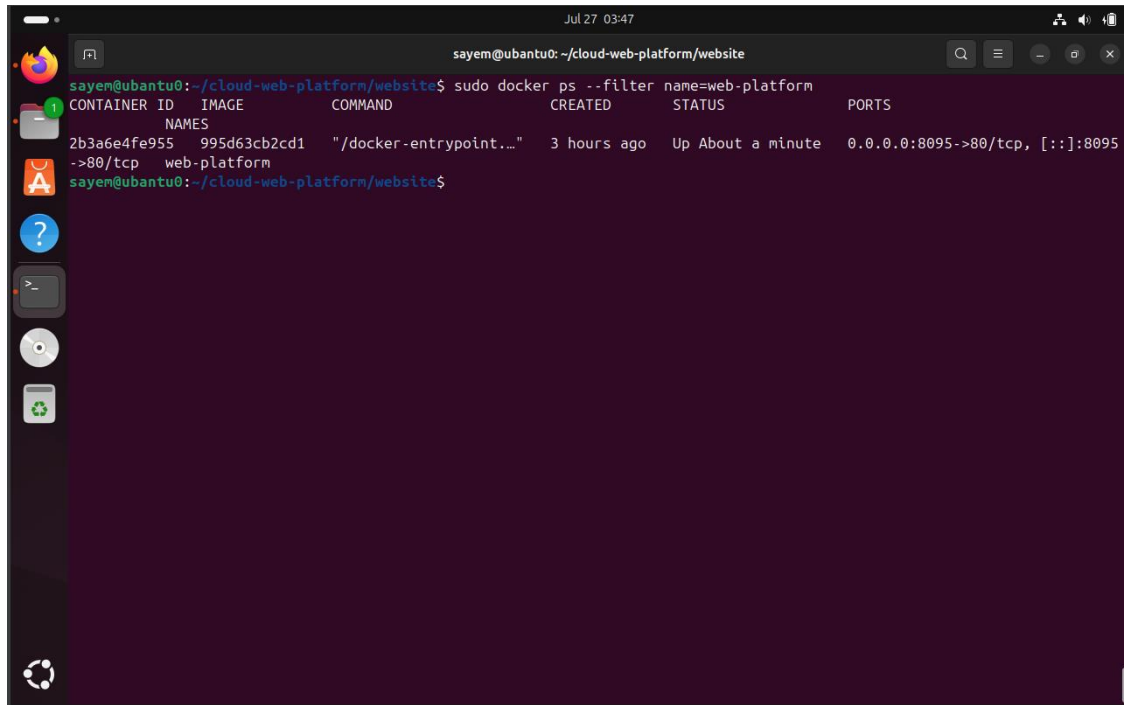


Figure 5: Verification of the locally created `cloud-web-platform:v1` Docker image.

4.4 Run and Test the Console Container

After successfully building the image, I created a container named `web-platform` from the custom image. The `-d` option put it in detached mode, and `-p 8095:80` mapped container port 80 to host port 8095. A service port that can be exposed on the host is accessible via docker port publishing to an internal service [\[11\]](#). As shown in Figure 6, the target container was running, and the expected port mapping was active.

```
sudo docker run -d -p 8095:80 --name web-platform cloud-web-platform:v1
```

A terminal window titled 'sayem@ubuntu0: ~/cloud-web-platform/website' showing the output of the command 'sudo docker ps --filter name=web-platform'. The output is a table with columns: CONTAINER ID, IMAGE, COMMAND, CREATED, STATUS, and PORTS. One container is listed with ID '2b3a6e4fe955', image '995d63cb2cd1', command '/docker-entrypoint...', created '3 hours ago', status 'Up About a minute', and ports '0.0.0.0:8095->80/tcp, [::]:8095'. The terminal also shows the prompt 'sayem@ubuntu0:~/cloud-web-platform/website\$' and a sidebar with application icons on the left.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
2b3a6e4fe955	995d63cb2cd1	"/docker-entrypoint..."	3 hours ago	Up About a minute	0.0.0.0:8095->80/tcp, [::]:8095

Figure 6: Running the custom Nginx container with host-to-container port mapping.

The webpage was then accessed through a web browser, on <http://127.0.0.1:8095>. This is an example of a custom page that was successfully served from the web-platform container (Figure 7). The first full circle was completed: source file → Dockerfile → image → running container and output access in the browser.

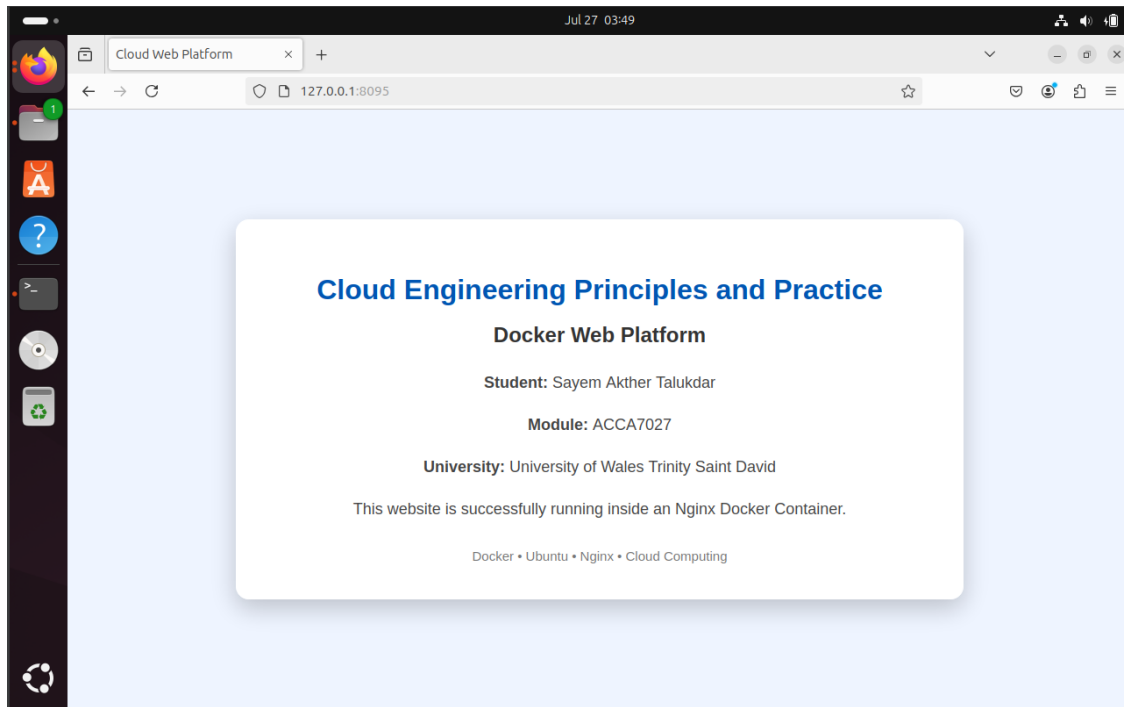
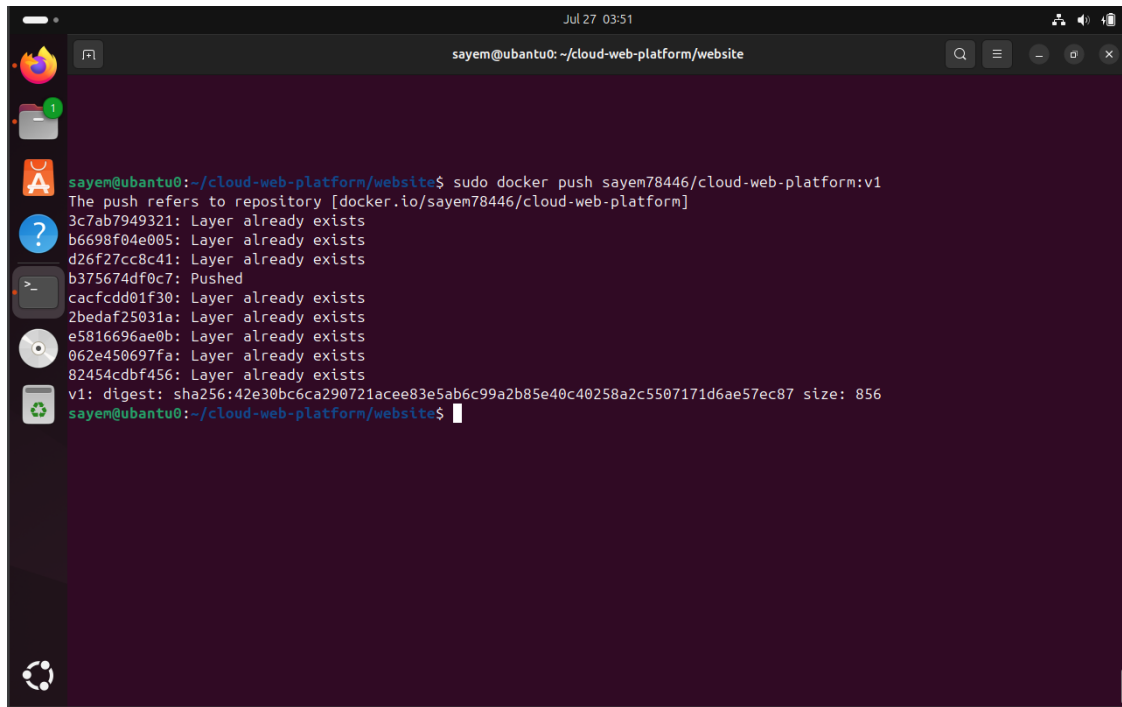


Figure 7: Custom HTML webpage served successfully from the Nginx Docker container.

4.5 For publishing image on Docker Hub

To publish the local image, I then logged into Docker Hub and tagged my repository as sayem78446/cloud-web-platform:v1. Pushing the image to the repository Docker pushes the image layers and saves the image manifest under the specified repository and tag [\[12\]](#). The push was successful, and a registry digest is returned (see Figure 8).

```
sudo docker tag cloud-web-platform:v1 sayem78446/cloud-web-platform:v1
sudo docker push sayem78446/cloud-web-platform:v1
```



```
sayem@ubuntu0: ~/cloud-web-platform/website
sayem@ubuntu0:~/cloud-web-platform/website$ sudo docker push sayem78446/cloud-web-platform:v1
The push refers to repository [docker.io/sayem78446/cloud-web-platform]
3c7ab7949321: Layer already exists
b6698f04e005: Layer already exists
d26f27cc8c41: Layer already exists
b375674df0c7: Pushed
cacfcdd01f30: Layer already exists
2bedaf25031a: Layer already exists
e5816696ae0b: Layer already exists
062e450697fa: Layer already exists
82454cdbf456: Layer already exists
v1: digest: sha256:42e30bc6ca290721acee83e5ab6c99a2b85e40c40258a2c5507171d6ae57ec87 size: 856
sayem@ubuntu0:~/cloud-web-platform/website$
```

Figure 8: Publishing the custom cloud-web-platform:v1 image to Docker Hub.

Figure 9 provides browser-based confirmation that the cloud-web-platform repository was present in the sayem78446 Docker Hub namespace. Publication is important because the image is no longer limited to the local Ubuntu system and can be retrieved by another authorized Docker environment.

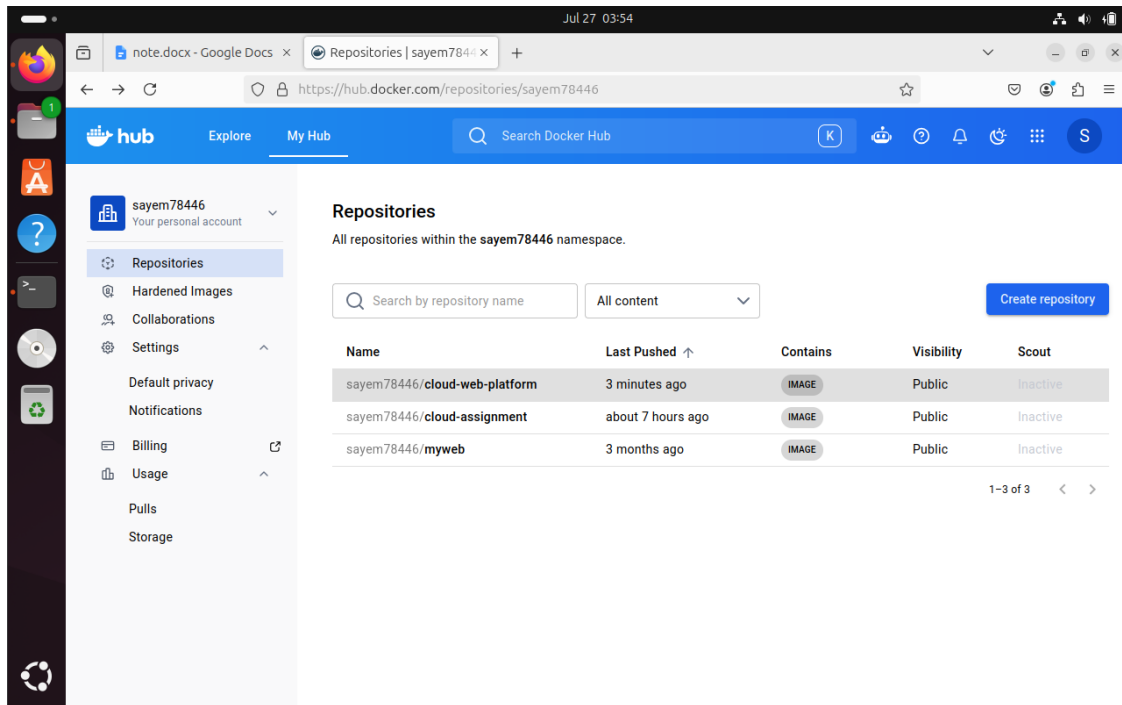
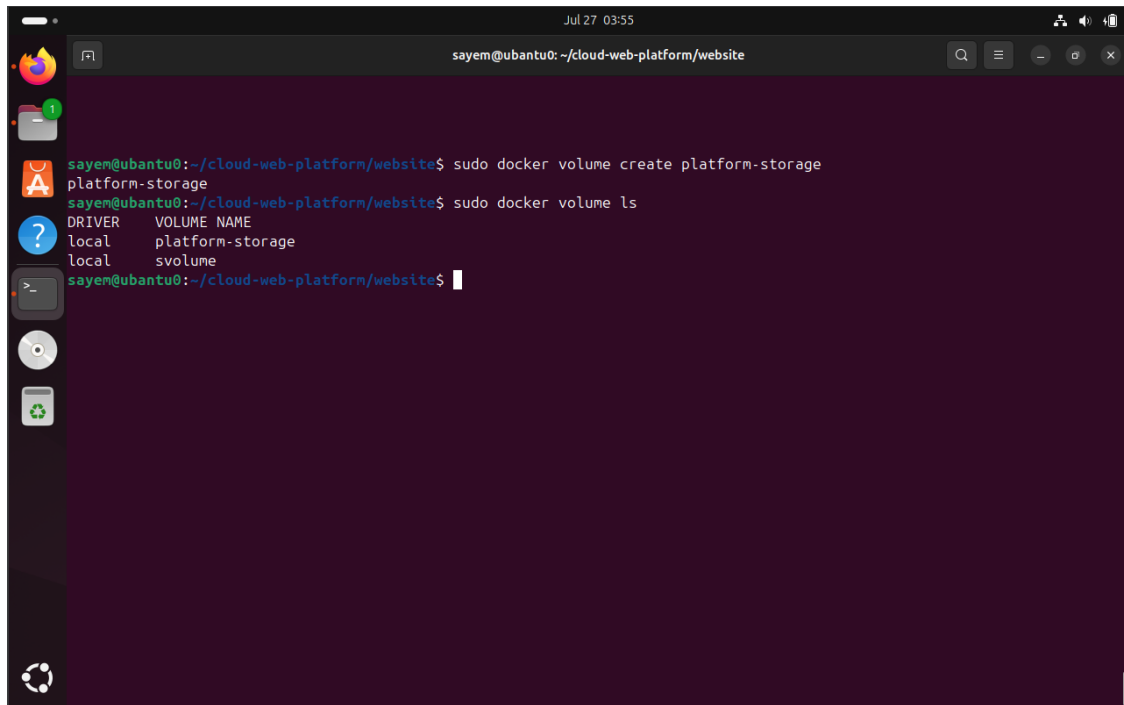


Figure 9: Published `cloud-web-platform:v1` image available in the Docker Hub repository.

4.6 Docker volume and persistent data for websites

So, a named volume called `platform-storage` has been created to separate the website data from the lifecycle of an individual container. Docker volumes provide persistent data storage and exist outside the writable layer of a removed container [13], [14]. Creating the volume and verifying using `docker volume ls` is shown in Figure 10.

```
sudo docker volume create platform-storage
sudo docker volume ls
```

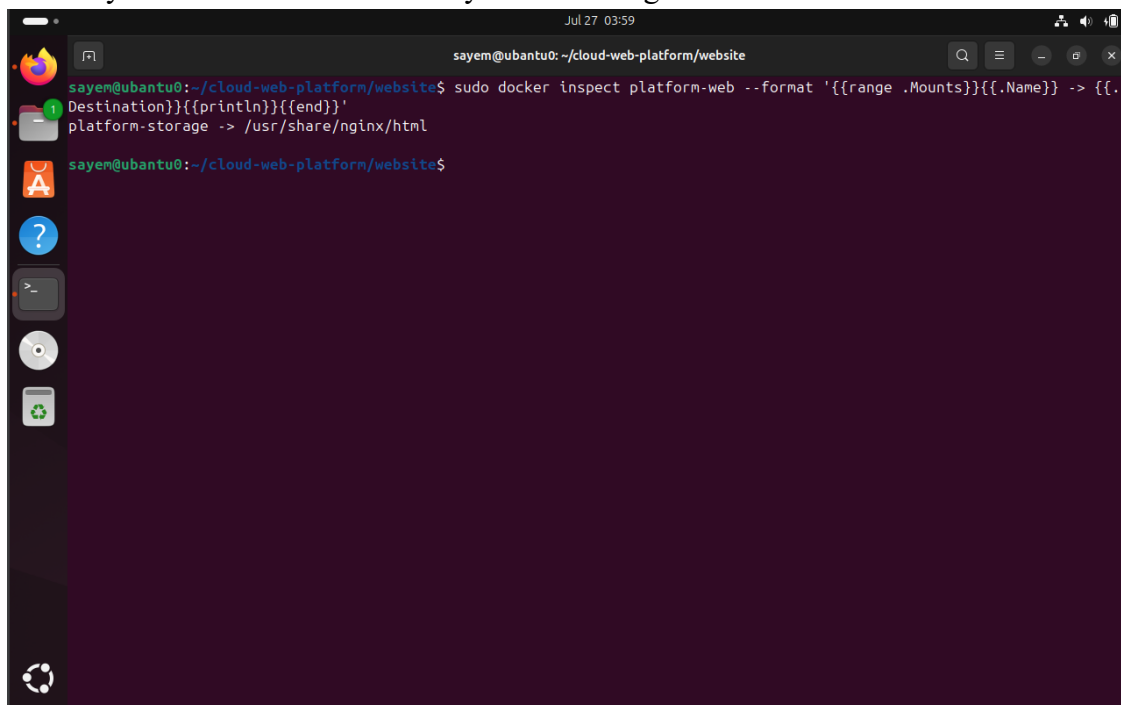
A terminal window titled 'sayem@ubuntu0: ~/cloud-web-platform/website' with a dark purple background. The terminal shows the following commands and output:

```
sayem@ubuntu0:~/cloud-web-platform/website$ sudo docker volume create platform-storage
platform-storage
sayem@ubuntu0:~/cloud-web-platform/website$ sudo docker volume ls
DRIVER      VOLUME NAME
local      platform-storage
local      svolume
sayem@ubuntu0:~/cloud-web-platform/website$
```

Figure 10: Creation and verification of the platform-storage Docker volume.

The index.html file was copied into the volume using a temporary Ubuntu container. An Nginx container named platform-web was then created, mounting the platform-storage volume at /usr/share/nginx/html and exposing the service through host port 8096. Figure 11 provides Docker inspect output, confirming that the volume was successfully mounted and that the destination

directory was correctly configured within the container.

A terminal window titled 'sayem@ubuntu: ~/cloud-web-platform/website' showing the command 'sudo docker inspect platform-web --format '{{range .Mounts}}{{.Name}} -> {{.Destination}}{{println}}{{end}}'' and its output 'platform-storage -> /usr/share/nginx/html'. The terminal has a dark purple background and a sidebar with application icons on the left.

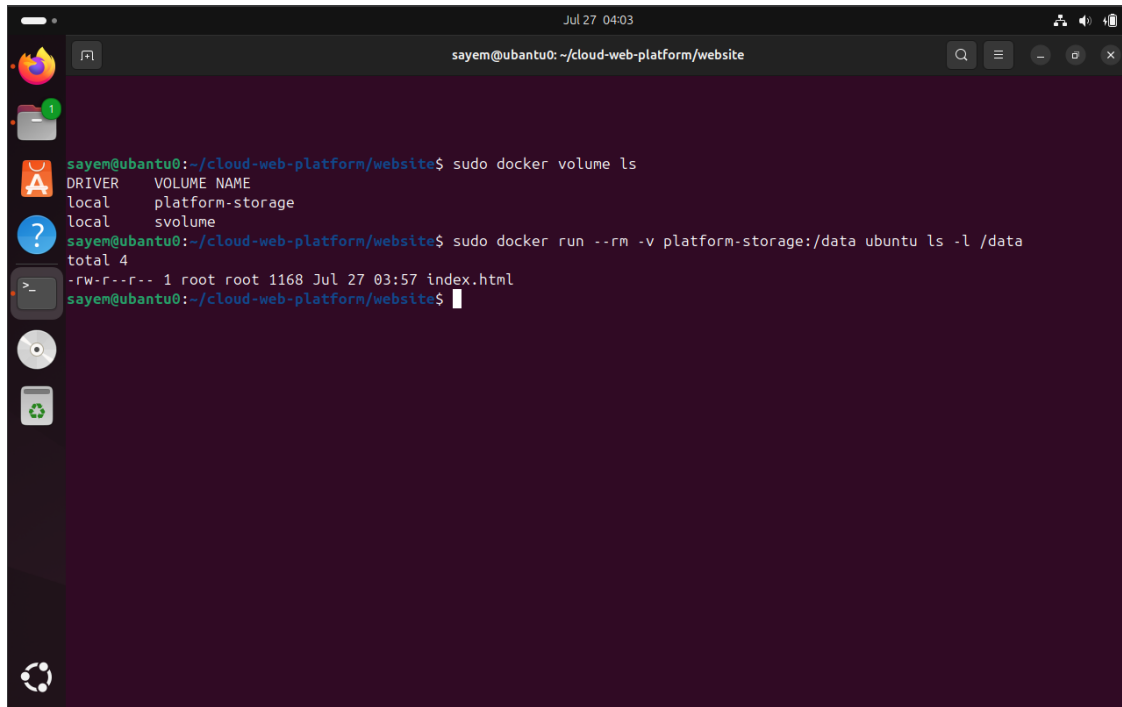
```
sayem@ubuntu0: ~/cloud-web-platform/website$ sudo docker inspect platform-web --format '{{range .Mounts}}{{.Name}} -> {{.Destination}}{{println}}{{end}}'
```

```
platform-storage -> /usr/share/nginx/html
```

```
sayem@ubuntu0: ~/cloud-web-platform/website$
```

Figure 11: Nginx container configured to use the platform-storage Docker volume.

The first volume-backed container was then stopped and removed for testing persistence. The volume continued to exist because its lifecycle was managed separately from that of the container you deleted. This platform-storage volume is shown in Figure 12 and serves to validate that the index.html file was still inside it. This is better proof than looking for the name of a volume, and it also proves that the file inside was still there.

A terminal window on an Ubuntu system. The window title is 'sayem@ubuntu0: ~/cloud-web-platform/website'. The terminal shows the following commands and output:

```
sayem@ubuntu0:~/cloud-web-platform/website$ sudo docker volume ls
DRIVER      VOLUME NAME
local       platform-storage
local       svolume

sayem@ubuntu0:~/cloud-web-platform/website$ sudo docker run --rm -v platform-storage:/data ubuntu ls -l /data
total 4
-rw-r--r-- 1 root root 1168 Jul 27 03:57 index.html

sayem@ubuntu0:~/cloud-web-platform/website$
```

Figure 12: Persistence of website data after removal of the original Docker container.

To replace it, I started a new nginx container with a volume and host port 8097. In the screenshot evidence, this replacement points to platform-web-new. In Figure 13, the new container has the same web page through port 8097. This also shows that the application content was restored from platform-storage, which means it was not restored from the deleted container.

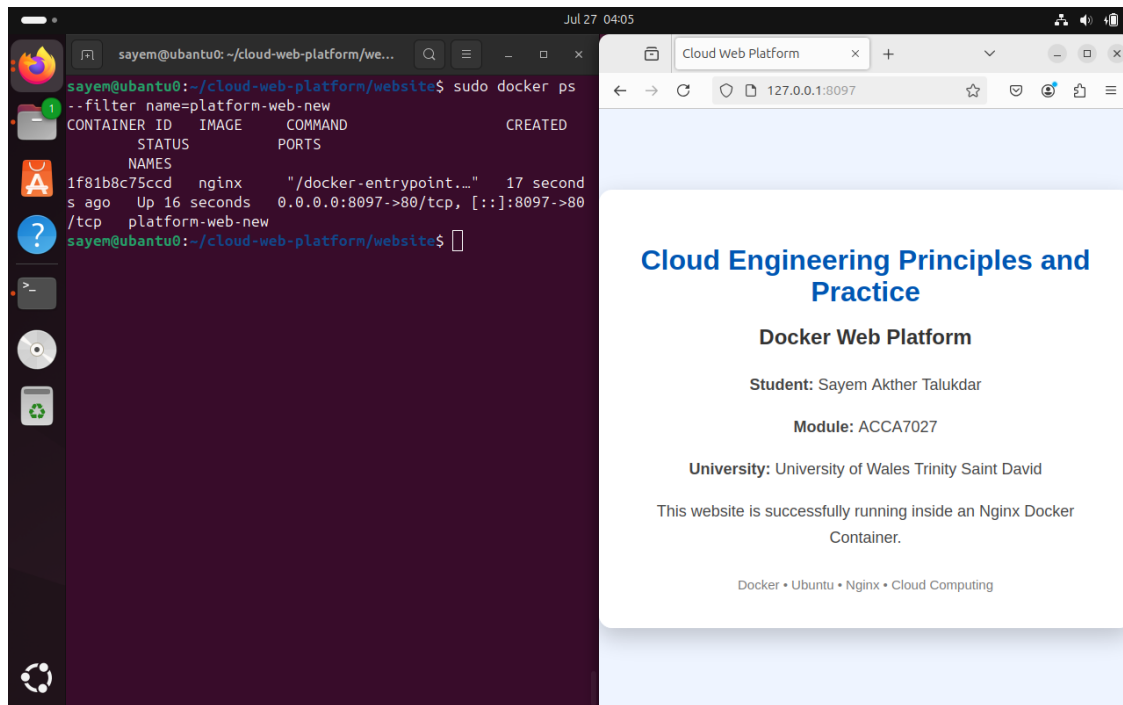
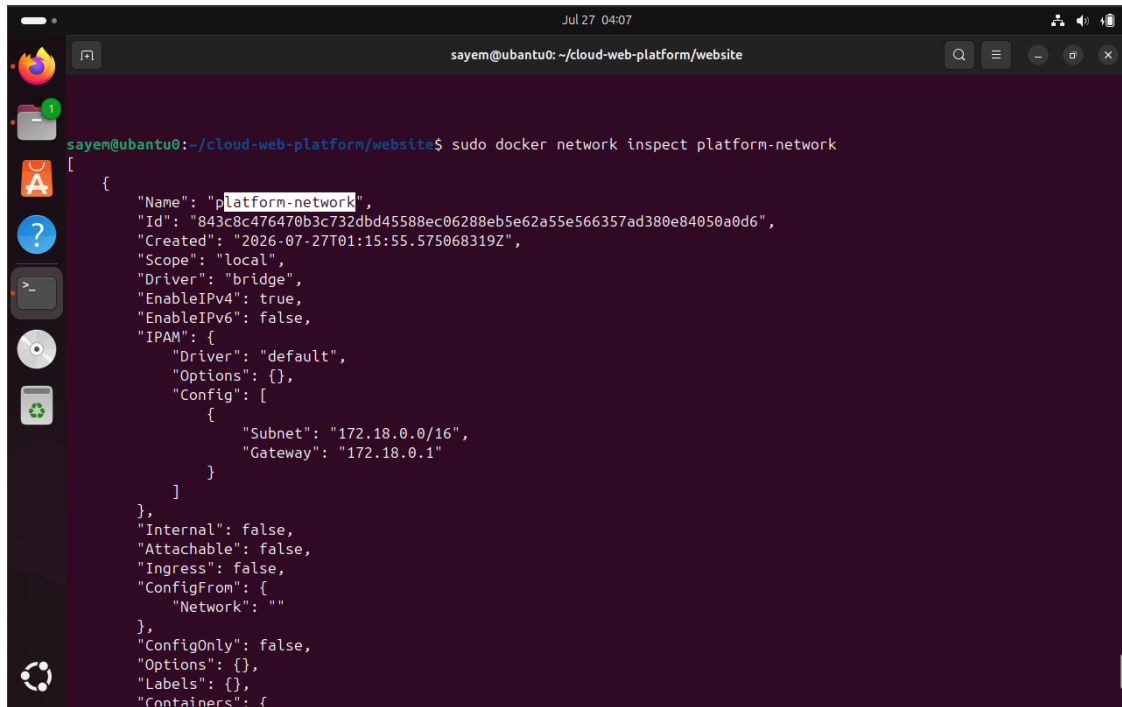


Figure 13: Reuse of persistent website data by a newly created Nginx container.

4.7 Custom networks and communication between containers

I created a user-defined bridge network called platform-network, to which I added the existing container web-platform. Finally, Docker user-defined bridge networks [15] are scoped for communication and provide auto-discovery of containers attached to them. The output of network inspection is shown in Figure 14, including the bridge driver, subnet, and gateway configuration.

```
sudo docker network create platform-network
sudo docker network connect platform-network web-platform
```

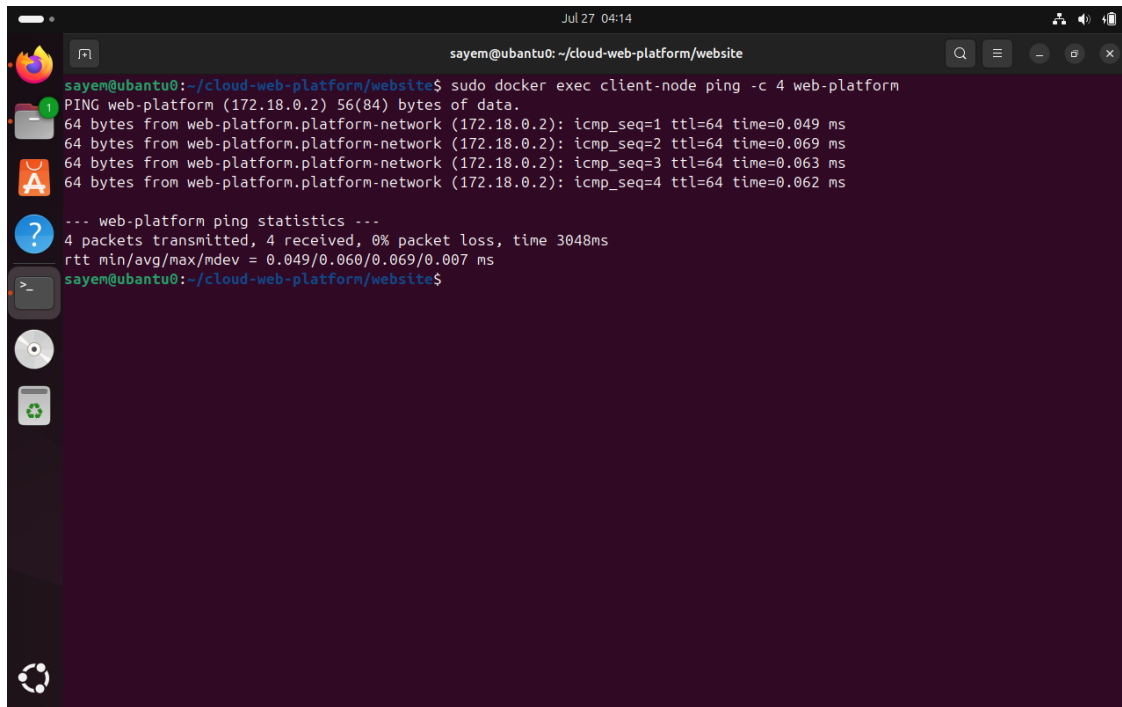
A terminal window titled 'sayem@ubuntu0: ~/cloud-web-platform/website' showing the command 'sudo docker network inspect platform-network'. The output is a JSON object describing the network configuration. The network is named 'platform-network', has a unique ID, and was created on 2026-07-27. It is a local bridge network with IPv4 enabled and IPv6 disabled. The IPAM configuration shows a default driver with a subnet of 172.18.0.0/16 and a gateway of 172.18.0.1. Other fields like Internal, Attachable, Ingress, ConfigFrom, ConfigOnly, Options, Labels, and Containers are also present.

```
sayem@ubuntu0:~/cloud-web-platform/website$ sudo docker network inspect platform-network
[
  {
    "Name": "platform-network",
    "Id": "843c8c476470b3c732dbd45588ec06288eb5e62a55e566357ad380e84050a0d6",
    "Created": "2026-07-27T01:15:55.575068319Z",
    "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": {},
    "Labels": {},
    "Containers": {}
  }
]
```

Figure 14: Creation and inspection of the custom platform-network bridge network.

A second container named client-node was created named client-node from ubuntu:24.04 on the same network. Once iputils-ping and curl are installed, clients perform a container resolution test via ping by name to the web container. Figure 15: Four Successful Ping Replies from web-platform with Zero Packet Loss. The fact that the container name (rather than a hard-coded address) worked proves that Docker network name resolution was working.

```
sudo docker run -dit --name client-node --network platform-network ubuntu:24.04 bash
sudo docker exec client-node ping -c 4 web-platform
```

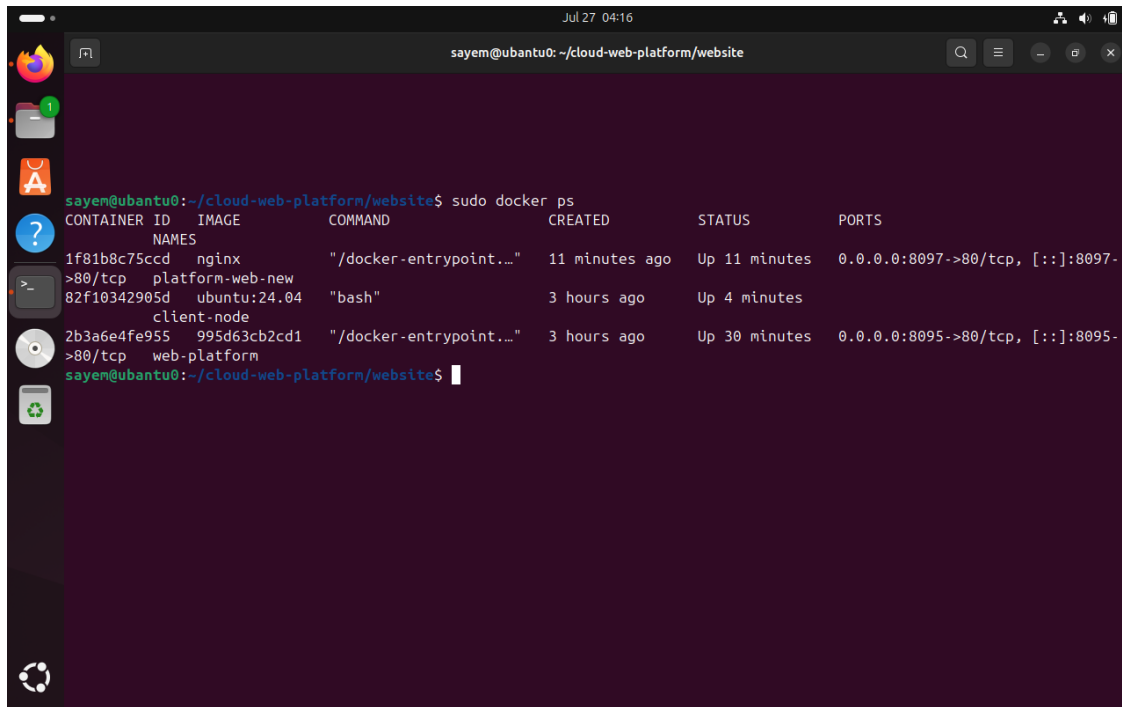


```
sayem@ubuntu0: ~/cloud-web-platform/website
sayem@ubuntu0:~/cloud-web-platform/website$ sudo docker exec client-node ping -c 4 web-platform
PING web-platform (172.18.0.2) 56(84) bytes of data:
64 bytes from web-platform.platform-network (172.18.0.2): icmp_seq=1 ttl=64 time=0.049 ms
64 bytes from web-platform.platform-network (172.18.0.2): icmp_seq=2 ttl=64 time=0.069 ms
64 bytes from web-platform.platform-network (172.18.0.2): icmp_seq=3 ttl=64 time=0.063 ms
64 bytes from web-platform.platform-network (172.18.0.2): icmp_seq=4 ttl=64 time=0.062 ms

--- web-platform ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3048ms
rtt min/avg/max/mdev = 0.049/0.060/0.069/0.007 ms
sayem@ubuntu0:~/cloud-web-platform/website$
```

Figure 15: Successful communication between the client and web server containers through the custom Docker network.

Figure 16: Final docker ps output. It has the earliest web-platform service, the test container for a client-node, and now the new Nginx container from volumes. The environment thus shows image-based deployment, published ports, persistent storage, and communication between containers. The practical sequence shown in the video evidence should be referenced in Section 4.8.

A terminal window titled 'sayem@ubuntu0: ~/cloud-web-platform/website' showing the output of the 'sudo docker ps' command. The output is a table with columns: CONTAINER ID, IMAGE, COMMAND, CREATED, STATUS, and PORTS. The table lists three containers: '1f81b8c75ccd' (nginx), '82f10342905d' (platform-web-new), and '2b3a6e4fe955' (web-platform).

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
1f81b8c75ccd	nginx	"/docker-entrypoint..."	11 minutes ago	Up 11 minutes	0.0.0.0:8097->80/tcp, [::]:8097->80/tcp
82f10342905d	ubuntu:24.04	"bash"	3 hours ago	Up 4 minutes	
2b3a6e4fe955	995d63cb2cd1	"/docker-entrypoint..."	3 hours ago	Up 30 minutes	0.0.0.0:8095->80/tcp, [::]:8095->80/tcp

Figure 16: Final overview of the running Docker containers used in the practical implementation.

4.8 Video Demonstration

The complete practical demonstration of the Docker implementation is available at: <https://youtu.be/wqh-eYmGEXk>

5. Load Balancing, Scalability and Application Gateways

5.1 The Suggested Azure Context for a Demonstrated Web Application

The local docker project only serves a single static web app from a single main web platform container. That means the deployment in our cloud would have to keep serving this page despite increased demand and even if one backend instance is down. Within this analysis, the same containerized web application is assumed to be an Azure workload (with multiple backend instances). This is a conceptual stretch the assessment needs and does not mean the actual environment itself was deployed in Azure.

5.2 Load Balancing

A load balancer is a type of computing resource that distributes requests across multiple healthy compute resources. Azure Load Balancer is a Layer 4 load balancer, used for TCP and UDP traffic distribution to replicate virtual machines or scale-set instances [16]. Health probes also lessen the impact of unavailable instances by excluding them from active traffic distribution, increasing resilience over the use of a single server. This allows multiple identical container hosts to share incoming connections for the Docker web service instead of one.

The biggest benefit is not just increasing throughput. Additionally, load balancing helps prevent a single instance from becoming the sole provider of service delivery and supports continuing service even with other running instances responding. The most important thing to note about the device is that it cannot create its own capacity. If all the backend instances are overloaded, then spreading out traffic does not fix the underlying shortage. You therefore must work with some sort of scaling policy as well with load balancing.

5.3 Scalability

Scalability refers to the ability to increase and/or decrease the capacity of an application. Vertical scaling involves adding CPU, memory, or other capacity to an existing resource, and horizontal scaling consists of adding more instances in use. Although Azure App Service and even Azure Virtual Machine services support both approaches, horizontal scaling is usually compatible with a stateless web application since equal instances can be added or removed without altering one consolidated server [\[17\]](#).

Azure autoscaling can react to schedules or runtime metrics for example CPU load, memory usage, or queue length [\[18\]](#). For the web service outlined, a scale-out rule might provision instances when sustained demand surpassed a mutually agreed threshold, then remove them once demand dipped. This promotes responsiveness and cost efficiency, but poorly chosen thresholds can lead to a leap in scale, delayed action, or unwanted costs. So, scaling rules must match the actual workload patterns, while specifying appropriate minimum and maximum instances, as well as cooldown periods.

5.4 Application Gateway

Azure Application Gateway is a web traffic load balancer for HTTP and HTTPS applications. It provides host-based and path-based routes, TLS termination, backend health checking, and integration with a web application firewall [\[19\]](#). These functions operate on the application layer and (unlike a transport-layer load balancer) make routing choices based on web-request information that is not available at the transport layer.

In the showcased demo, the Application Gateway could terminate HTTPS, and normal web traffic could be routed to the pool with cloud-web-platform. For example, if the application later added two paths, /api and /admin, they could be routed to completely different backend services. Or a web application firewall could inspect the requests before they even reach the container hosts. So, the gateway augments application awareness, but with caveats and prudence: routing rules or certificates can unintentionally block legitimate access.

5.5 Why Load Balancing and Scaling Are Complementary

Scaling modifies the amount of available capacity; load balancing makes sure that requests use that capacity. A load-balancing layer will need to identify a new instance (when scaling out, for example) as healthy before routing traffic to it. Of course, if a taken-away instance now stops receiving new connections from the balancer. Without load balancing, some instances may be

underutilized; without scaling, a load balancer can only spread the traffic among a bounded and possibly too small set of servers. Both mechanisms act on different time scales: while the load balancer decides for a back connection/request, the scaling systems adjust the backend pool based on demand. Their coordination is therefore essential. This means that the host is already available for processing requests even if the application itself is not ready, so Health probes, start-up time, and cooldown should be taken into consideration.

5.6 Advantages of Using the Three Technologies Combine

Microsoft defines the Azure load-balancing services as additional options that can be mixed and matched based upon transport, web-routing, and geographic needs [20]. Application Gateway provides a secure HTTP entry point, Azure Load Balancer distributes lower-level network traffic, and horizontal scaling increases the number of backend instances to improve performance and process increased traffic. Their joint contribution is summarized in Table 2.

Table 2: Combined contribution of load balancing, scalability and Application Gateway

Technology	Primary role	Customer benefit	Key consideration
Load balancing	Distributes traffic across healthy instances	More consistent response and reduced dependence on one server	Cannot create new capacity
Scalability	Adds or removes compute instances	Handles changing demand while limiting overprovisioning	Requires tested thresholds and start-up planning
Application Gateway	Controls HTTP(S) routing and web security	Secure, application-aware access to services	Rules, certificates and WAF policies require governance

The three concepts work together to enhance availability, performance, and operational control. Customers get one secure entry point, requests are routed only to healthy resources, and growth is possible during peak load periods. This only unifies the design process between pods and stateful sets, while also adding monitoring and config- and cost-management capabilities. So a good cloud architecture uses these services as orchestrated controls, not as discrete products.

6. Driving Forces Behind the Expansion of Cloud Computing

6.1 Model and Resource-Efficiency Cost

Cost is one of the biggest drivers of cloud adoption because cloud services eliminate the need for a large initial expenditure in infrastructure. NIST defines cloud computing as a model for enabling on-demand access to shared computing resources provided as services [1]. Armbrust et al. characterize cloud computing as an important financial shift that eliminates upfront commitments and enables rapid access to IT resources [2]. Similarly, Buyya et al. describe cloud computing as a utility-oriented model that delivers computing capacity on demand [3]. The economic benefit is

maximum when demand varies over time. This allows an organization to free resources after a temporary workload, instead of keeping hardware unused. But slack isn't automatically cheaper in the cloud. Recurring expenditure can remain high for years, driven by long-lived or poorly managed services with excessive data transfer and oversized resources. This hence requires monitoring, rightsizing, scaling and lifecycle control to achieve cost optimization. Clouds create value when dollars spent are linked to valuable consumption, rather than transferring that existing inefficiency to a provider bill.

6.2 Flexibility, Elasticity and Speed of Provisioning

Flexibility is the ability to choose and adapt computing resources without needing to reconstruct physical systems. Elasticity, on the other hand, is a more granular feature of scaling in and out based on demand. These features combined enable development teams to build environments rapidly, test modifications, and deploy services without the delay of hardware purchase. This principle is illustrated with Docker at the application level where a single image can have many equivalent containers, and a registry facilitates moving the package to another environment.

Speed also supports experimentation. Starting small with resources that can grow after a proof of demand. It assists in reducing the risk of unintentionally deciding to build something permanent too early on. In exchange, you are dependent on provider services, interfaces, and their pricing. While container-based portability mechanisms eliminate certain classes of application-level dependency, provider-specific networking, identity, database, and serverless services can add complexity when migrating among cloud providers.

6.3 Availability, Worldwide Reach and Focus on Functionality

Cloud platforms supply applications with the ability to leverage a mixture of instances, zones, and regions that an organization would have to build out on its own. Managed health monitoring, load balancing, and automatic replacement can improve service availability. It also helps customers and employees at different locations to access services. These benefits are especially valuable in the context of web applications, where demand and user locations can change dynamically.

Transferring some of the operational burden to the provider is another driver. Infrastructure maintenance, hardware replacement, and managed-service updates give application teams the freedom to focus on service logic. There is borderline liability for customers on this. Identity, access control, application security, data governance and proper service configuration still matter. Cloud expansion is then based on the mutual responsibility: the provider standardizes infrastructure operations, but customers must govern the selection and use of services.

6.4 Serverless Functions

Serverless computing extends the cloud model by allowing code to run without the user provisioning or maintaining the underlying server. Azure Functions is an event-driven serverless platform in which functions can respond to HTTP requests, database changes, queues, schedules and other triggers [\[21\]](#). AWS Lambda provides a comparable model and automatically manages

infrastructure, scaling and patching for function execution [22]. The term serverless does not mean that servers disappear; it means that server management is abstracted from the application developer.

Academic literature identifies serverless as an evolution of cloud programming models in which operational concerns and fine-grained billing are shifted towards the provider [23], [24]. Common benefits are rapid deployment, automatic scaling and payment based on execution. Studies of real serverless applications show that the model is particularly attractive for event processing, APIs, automation and workloads with variable or intermittent demand [25], [26].

The limitations are equally important. Functions are usually designed to be short-lived and stateless, with durable state stored in external services. Cold starts can add latency after an idle period, and provider limits may constrain execution time, memory, networking or runtime choice. Azure provides several hosting plans with different performance, scaling and cost characteristics, showing that serverless is not one uniform operating model [27].

6.5 Serverless Functions vs Containerization

Containers and serverless functions are both cloud-native, but they place responsibility at different levels. A container is a virtualized environment that can be run in isolation, and it usually provides the team with more control over dependencies, processes, ports, and execution time. A function is smaller, wraps your code, and lets a provider handle most infrastructure management and scaling. In Table 3, I compare the two approaches against the driving forces identified above.

Table 3: Comparison of serverless functions and containerisation

Criterion	Serverless Functions	Containers
Provisioning	Provider provisions execution capacity automatically	Team builds the image and selects the container platform
Scaling	Usually event-driven and can scale from zero	Depends on container runtime or orchestration configuration
Cost pattern	Fine-grained execution-based charging can suit intermittent workloads	Allocated hosts or managed container capacity may suit steady workloads
Runtime control	More provider constraints and supported runtime limits	Greater control of libraries, process model and image contents
Portability	Function code may depend on provider triggers and services	Images improve application packaging portability, subject to platform compatibility
Workload fit	Short, event-driven and stateless tasks	Web services, long-running processes and applications needing custom environments

Cost-wise, the serverless aspect can be very productive if there are few requests, as the function scales to zero when idle. For a constantly running service though, this model of either continuously

executing the function or having to find ways to support managed services may not be the cheapest design. From the perspective of performance for a long-running web server like the Nginx application I demonstrated in this report, containers may offer better deterministic performance and more control. So, the right choice then is determined by workload shape rather than by an overarching assertion that one model is always cheaper.

operational flexibility perspective, serverless reduces infrastructure decisions but increases reliance on provider-specific triggers, limits, and integrations. Containers themselves require more operational management, but they maintain a complete application environment and can shift between compatible hosts. A hybrid design is more often the right way to go. Serverless functions handle infrequent events like resizing an image, sending notifications, or running scheduled maintenance, and the main web application can run inside containers. It unites the portability and controlled runtime of containers with the event-driven scalability of functions.

7. Conclusion

Virtualization, containerization and serverless computing represent different levels of abstraction in cloud engineering. Virtual machines divide physical hardware into independent operating-system environments, while containers package applications more efficiently by sharing the host kernel. The practical project demonstrated the complete Docker process through a custom Nginx webpage, image construction, container execution, Docker Hub publication, persistent volume storage and communication over a user-defined bridge network.

For cloud deployment, the same service would require coordinated load balancing, scaling and application-layer routing. Azure Load Balancer, autoscaling and Application Gateway illustrate how these responsibilities can be combined to improve availability, responsiveness and security. Cost, flexibility and rapid provisioning continue to drive cloud expansion, but benefits depend on governance and workload suitability. Serverless functions reduce operational responsibility for event-driven code, whereas containers provide stronger runtime control for sustained services. The evidence therefore supports a selective approach in which each technology is applied according to the isolation, performance, portability and operating requirements of the workload.

References

- [1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” *NIST*, vol. 800, no. 145, Sept. 2011, doi: 10.6028/nist.sp.800-145.
- [2] M. Armbrust *et al.*, “A View of Cloud Computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010, doi: 10.1145/1721654.1721672.
- [3] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, “Cloud Computing and Emerging It Platforms: Vision, Hype, and Reality for Delivering Computing as the 5Th Utility,” *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, June 2009, doi: 10.1016/j.future.2008.12.001.
- [4] K. Scarfone, M. Souppaya, and P. Hoffman, “Guide to Security for Full Virtualization Technologies Recommendations of the National Institute of Standards and Technology Special Publication 800-125,” Jan. 2011. Accessed: July 27, 2026. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-125.pdf>
- [5] R. Chandramouli, “Security Recommendations for Server-Based Hypervisor Platforms,” June 2018, doi: 10.6028/nist.sp.800-125ar1.
- [6] C. Pahl, “Containerization and the PaaS Cloud,” *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May 2015, doi: 10.1109/mcc.2015.51.
- [7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An Updated Performance Comparison of Virtual Machines and Linux Containers,” *2015 IEEE International Symposium on Performance Analysis of Systems and Software (Ispass)*, pp. 171–172, Mar. 2015, doi: 10.1109/ispass.2015.7095802.
- [8] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment | Linux Journal,” June 2014. Accessed: July 27, 2026. [Online]. Available: <https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment>
- [9] Docker Inc, “What Is Docker?,” *Docker Documentation*, Feb. 11, 2026. <https://docs.docker.com/get-started/docker-overview/>. (accessed July 27, 2026).
- [10] *Docker.com*, 2026. <https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/>. (accessed July 27, 2026).
- [11] *Docker.com*, 2026. <https://docs.docker.com/get-started/docker-concepts/running-containers/publishing-ports/>. (accessed July 27, 2026).

- [12] Docker Inc, “Build, Tag, and Publish an Image,” *Docker Documentation*, Feb. 22, 2026. <https://docs.docker.com/get-started/docker-concepts/building-images/build-tag-and-publish-an-image/>. (accessed July 27, 2026).
- [13] Docker Inc, “Volumes,” *Docker Documentation*, July 21, 2026. <https://docs.docker.com/engine/storage/volumes/>. (accessed July 27, 2026).
- [14] Docker Inc, “Storage Drivers,” *Docker Documentation*, June 02, 2026. <https://docs.docker.com/engine/storage/drivers/>. (accessed July 27, 2026).
- [15] Docker Inc, “Bridge Network Driver,” *Docker Documentation*, Feb. 21, 2026. <https://docs.docker.com/engine/network/drivers/bridge/>. (accessed July 27, 2026).
- [16] Microsoft, “What Is Azure Load Balancer? - Azure Load Balancer,” Feb. 2025. Accessed: July 27, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview>
- [17] cephalin, “Scale Up Features and Capacities - Azure App Service.” Accessed: July 27, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/app-service/manage-scale-up>
- [18] claytonsiemens77, “Autoscaling Guidance - Azure Architecture Center,” *MicrosoftLearn*, Mar. 10, 2026. <https://learn.microsoft.com/en-us/azure/architecture/best-practices/autoscaling> (accessed July 27, 2026).
- [19] greg-lindsay, “What Is Azure Application Gateway,” Sept. 2023. Accessed: July 27, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/overview>
- [20] martinekuan, “Load-Balancing Options - Azure Architecture Center.” Accessed: July 27, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/load-balancing-overview>
- [21] “Azure Functions Overview.” Accessed: July 27, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview>
- [22] AWS, “What Is AWS Lambda? - AWS Lambda,” 2025. Accessed: July 27, 2026. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
- [23] I. Baldini *et al.*, “Serverless Computing: Current Trends and Open Problems,” *Research Advances in Cloud Computing*, pp. 1–20, Jan. 2017, doi: [10.1007/978-981-10-5026-8_1](https://doi.org/10.1007/978-981-10-5026-8_1).
- [24] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, “The Rise of Serverless Computing,” *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, Nov. 2019, doi: [10.1145/3368454](https://doi.org/10.1145/3368454).
- [25] S. Eismann *et al.*, “Serverless Applications: Why, When, and How?,” *IEEE Software*, vol. 38, no. 1, pp. 32–39, Jan. 2021, doi: [10.1109/ms.2020.3023302](https://doi.org/10.1109/ms.2020.3023302).

[26] M. Shahrad *et al.*, “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider,” 2020. Accessed: July 27, 2026. [Online]. Available: <https://www.usenix.org/system/files/atc20-shahrad.pdf>

[27] C. Lin, Y. Ma, and M. Shahrad, “Demystifying Serverless Costs on Public Platforms: Bridging Billing, Architecture, and OS Scheduling,” *Proceedings of the 21St European Conference on Computer Systems*, pp. 1964–1981, Apr. 2026, doi: [10.1145/3767295.3769374](https://doi.org/10.1145/3767295.3769374).

Appendix A: Dockerfile

The Dockerfile used in the practical implementation is reproduced below once for reference.

```
FROM nginx:latest
LABEL author="Sayem Akther Talukdar"
LABEL project="Cloud Web Platform"
COPY index.html /usr/share/nginx/html/index.html
EXPOSE 80
```

Appendix B: HTML Source

The HTML source used by the custom Nginx image is reproduced below.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Cloud Web Platform</title>
<style>
body{
background:#eef4ff;
font-family:Arial,sans-serif;
display:flex;
justify-content:center;
align-items:center;
height:100vh;
margin:0;
}
.container{
background:white;
width:750px;
padding:40px;
border-radius:15px;
box-shadow:0 10px 25px rgba(0,0,0,.2);
text-align:center;
}
h1 {color:#0056b3;margin-bottom:15px;}
h2 {color:#333;margin-bottom:25px;}
```

```

p{font-size:18px;line-height:30px;color:#444;}
.footer{margin-top:30px;font-size:15px;color:#777;}
</style>
</head>
<body>
<div
class="container">
<h1>Cloud Engineering Principles and Practice</h1>
<h2>Docker Web Platform</h2>
<p><strong>Student:</strong> Sayem Akther Talukdar</p>
<p><strong>Module:</strong> ACCA7027</p>
<p><strong>University:</strong> University of Wales Trinity Saint David</p>
<p>This website is successfully running inside an Nginx Docker Container.</p>
<div class="footer">Docker • Ubuntu • Nginx • Cloud Computing</div>
</div>
</body>
</html>

```