

ACCA7027: Cloud Engineering Principles and Practice

*Static Cloud Engineering Assignment Portfolio Website
Deployed using Docker and Nginx*

Submitted by:
Md Imran Biswas

Module Code: ACCA7027
Docker Hub: hmdimran

Academic Year 2025–2026

Contents

1. Introduction.....	3
2. Principles of Virtualisation	3
2.1 Definition and Role of Virtualisation	3
2.2 Type 1 and Type 2 Hypervisors	4
2.3 Virtualisation in Cloud Computing	4
2.4 Limitations of Virtual Machines.....	4
3. Principles and Benefits of Containerisation.....	5
3.1 Definition of Containerisation.....	5
3.2 Docker Architecture and Components	5
3.3 Benefits of Containerisation	6
3.4 Limitations and Risks	6
4. Practical Docker Demonstration: Imran's Static Web Page with Nginx	7
4.1 Practical Scenario	7
4.2 Project Folder and HTML Page Preparation	7
4.3 Dockerfile Construction	8
4.4 Building the Docker Image.....	9
4.5 Running the Docker Container	10
4.6 Browser Testing.....	10
4.7 Nginx Logs	11
4.8 Docker Hub Publishing	12
4.9 Docker Volume Demonstration.....	12
4.10 Docker Network Demonstration.....	14
4.11 Evaluation of the Practical Work.....	15
5. Cloud Computing Concepts: Load Balancing, Scalability and Application Gateways.....	16
5.1 Load Balancing.....	16
5.2 Scalability	16
5.3 Application Gateways.....	17
5.4 Why Load Balancing and Scalability Are Complementary	17
5.5 Benefits of Combining Load Balancing, Scalability and Application Gateways.....	18
6. Cloud Expansion Drivers and Serverless Functions.....	18
6.1 Main Driving Forces Behind Cloud Expansion	18
6.2 Serverless Functions: AWS Lambda and Azure Functions.....	19
6.3 Serverless Functions vs Containerisation.....	19
7. Conclusion	20
8. References.....	21
Appendix A: Video Demonstration Link.....	23

1. Introduction

The essence of cloud computing has radically changed how organizations design, deploy and manage their digital infrastructure. The move towards cloud-based, containerized deployments, as opposed to the traditional on-premises server environments is indicative of a move toward greater agility, cost-efficiency, and resiliency at scale [1]. This report has been prepared as part of the ACCA7027 Cloud Engineering Principles and Practice module and explores the theoretical and practical aspects of that transformation through the prism of a practical Docker and Nginx deployment by Md Imran Biswas.

It starts by looking at the principle behind the concept of virtualization and how it is used to facilitate the process of cloud computing before looking at the concept of containerization as a lightweight and more portable version of the concept of virtualization. One of the practical demonstrations is a detailed account of the entire workflow that the student underwent: writing a Dockerfile, building a custom Nginx image, deploying a container to serve a simple HTML page, continuing to update the content with Docker volumes, isolating services with custom Docker networks, and finally publishing the image to Docker Hub under the account hmdimran.

The subsequent sections place these practical skills into a broader context of cloud engineering, such as load balancing, horizontal and vertical scaling, application gateways, and the new paradigm of serverless computing. Throughout, the discussion is critical and takes into account not only the positive aspects of these technologies but also their architectural limitations, and in what conditions they can be used. The references are written in the IEEE numeric citation style [2].

2. Principles of Virtualization

2.1 Definition and Role of Virtualization

Virtualization refers to the process of developing a software based (virtual) representation of physical computing resources including processors, memory, storage and networking over the internet [3]. Virtualization enables the existence and operation of multiple, independent operating system environments, known as virtual machines (VMs) to coexist and run concurrently on the same physical host. Every VM is run with the understanding that they have exclusive use of the underlying hardware, although that hardware is in reality shared and controlled by the hypervisor [4].

It cannot be overemphasized that virtualization is important to cloud computation. In its absence, the dynamically scalable, reconfigurable and allocate-able compute resources would be a technical impossibility, which is a defining characteristic of a cloud platform such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Cloud providers can provide multi-tenancy, i.e. the co-hosting of multiple customers on the same physical infrastructure in an isolated and secure manner with the help of virtualization [3].

2.2 Type 1 and Type 2 Hypervisors

The key software component, which makes virtualization possible, is the hypervisor, which controls the distribution of physical hardware resources to virtual machines. It has two major types, known as Type 1 (bare-metal) and Type 2 (hosted) hypervisors, each of which is appropriate to various deployment scenarios [4].

Table 1. Type 1 vs Type 2 Hypervisors.

Feature	Type 1 Hypervisor (Bare-metal)	Type 2 Hypervisor (Hosted)
Definition	Runs directly on the physical hardware (no host OS required)	Runs on top of an existing host operating system
Performance	High – direct hardware access; minimal overhead	Lower – additional OS layer introduces latency
Typical Use Case	Enterprise data centres, cloud infrastructure	Developer workstations, testing, local labs
Examples	VMware ESXi, Microsoft Hyper-V, Xen, KVM	VMware Workstation, Oracle VirtualBox, Parallels
Security Isolation	Stronger – fewer software layers between VM and hardware	Weaker – host OS vulnerabilities may affect VMs
Deployment Cost	Higher upfront infrastructure cost	Lower cost; leverages existing OS

Type 1 hypervisors are massively favored due to their excellent performance and security features in the environment of cloud data centers [4]. The AWS EC2 instances, such as the Xen hypervisor, are customized and run on a dedicated hardware accelerator [5].

2.3 Virtualization in Cloud Computing

All major models of cloud service are based on the concept of virtualization: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The IaaS model sees cloud providers like AWS, Azure and GCP delivering virtualized compute instances to their clients allowing them to select CPU, memory and storage configurations without purchasing physical hardware [1]. This elasticity is the core of the value proposition of the cloud: organizations are able to deploy resources within minutes and release them when not in need and pay only what they use [3].

Moreover, virtualization allows pooling resources and live migration - the possibility to move a running VM between physical hosts without interrupting services - which is essential during maintenance procedures and high-availability systems [4]. All these features make up the tech base on which container-based deployments, including those illustrated in this report, have been constructed.

2.4 Limitations of Virtual Machines

Although useful, virtual machines have an inescapable architectural overhead that limits their appropriateness in high-density, rapid-deployment environments [6]. A full operating system kernel is included in each VM, and usually consumes gigabytes of memory and several minutes to boot. This heavyweight characteristic renders VMs less appropriate to situations demanding

rapid autoscaling like in the case of a sudden increase in traffic. Also, the resource duplication that is introduced by running multiple full OS stacks on the same host leads to a reduction in the consolidation efficiency of the resource loads compared to containerized workloads. The very same limitations that drove the creation of container technology which is discussed in the next section.

3. Principles and Benefits of Containerization

3.1 Definition of Containerization

Containerization is a type of virtualization technique that operates at the operating-system-level, and is the packaging of an application along with all its dependencies libraries, configuration files, runtime environment into a self-contained, portable unit known as a container [6]. A container is not a complete operating system, as opposed to a virtual machine. Rather, it shares the host operating system kernel, which is much lighter, faster to boot and more resource-efficient [7].

Released in 2013, Docker is the most popular platform to create and manage containers. It presented a friendly toolchain to developers, and a standardized image format that enabled containerization to scale [7]. Containers developed with Docker can be run with exactly the same code on development laptops, test servers and production cloud infrastructure, addressing the long-standing issue of environment inconsistency: the it works on my machine syndrome familiar to software engineers.

3.2 Docker Architecture and Components

Docker is based on a client-server structure. Docker client commands are sent to the Docker daemon (dockerd) which is in control of building images, running containers as well as managing networks and volumes. Photos can be stored on to the local drive or they can be fetched off of a registry like Docker Hub [7].

Table 2. Docker Components and Roles.

Component	Role
Docker Daemon (dockerd)	Background service that manages containers, images, volumes, and networks on the host
Docker Client (CLI)	Command-line interface through which the user sends instructions to the daemon
Docker Image	A read-only, layered template containing the application and its dependencies; basis for containers
Docker Container	A runnable instance of an image; isolated, stateless, and ephemeral by default
Dockerfile	A script of sequential instructions used to build a Docker image layer by layer
Docker Hub	Cloud-based public/private registry for sharing and distributing Docker images
Docker Volume	A persistent storage mechanism that survives container deletion; managed by the Docker daemon

Docker Network	A virtual network that enables controlled communication between containers
----------------	--

3.3 Benefits of Containerization

Containerization brings a number of benefits compared to both bare-metal and VM-based strategies of deployment [6][7]:

- **Portability:** Container images bundle together all of the runtime dependencies. The image developed on a local workstation of a developer will work exactly in the same way when it is deployed in a cloud instance despite the fact that it is built on an underlying operating system distribution.
- **Speed:** Containers are usually initiated in milliseconds as opposed to minutes of VM boot. This allows quicker horizontal scaling and shortened recovery times in case of failure.
- **Density:** Since containers share the host kernel, a much greater number of containers can be run per physical host than equivalent VMs, improving infrastructure utilisation and reducing cost.
- **Immutability and reproducibility:** A tagged Docker image (e.g., v1) is immutable. Deployments are deterministic, and can be versioned, audited and rolled back.
- **Developer-to-production consistency:** The identical container image as in development is then promoted to production, avoiding configuration drift and environment-specific bugs.

3.4 Limitations and Risks

Not without challenges is containerization. Security wise, the host OS kernel is shared by all the containers. In theory, a kernel-level vulnerability can impact all containers on that host, whereas a VM that has been compromised is not subjected to any interactions at the hypervisor level [8]. Moreover, Docker images obtained via public registries can contain out-of-date or susceptible software layers, and frequent image scanning and updates should be a critical practice in operation.

Moreover, containers are by definition volatile: once a container is removed, all data written in the container are lost unless an external volume is mounted. At scale, operationally unsustainable are managing container lifecycles, networking, and scaling manually. Applications like Kubernetes and Docker Swarm are hence commonly used in production to coordinate containers [6]. The practical demonstration in this report willingly employs manual Docker commands to demonstrate the underlying concepts; it is important to note that a production-grade deployment would require orchestration.

4. Practical Docker Demonstration: Imran's Static Web Page with Nginx

4.1 Practical Scenario

To illustrate the concept of containerization in action, Md Imran Biswas created a static HTML portfolio webpage and containerized the webpage with the help of Docker and Nginx. The entire workflow includes image creation, container execution, log inspection, volume-based persistence, custom network communication and Docker Hub publishing. All artefacts were built on a local workstation, which was running Docker Desktop, and the final image published to Docker Hub under the account hmdimran.

4.2 Project Folder and HTML Page Preparation

The project was structured in a well defined directory structure in which the Dockerfile and the static HTML file index.html were housed. Such segregation of issues is an industry best practice in containerised deployments [7]. As shown in figure 1, the structure of the folders to be used in the practical demonstration is as follows.

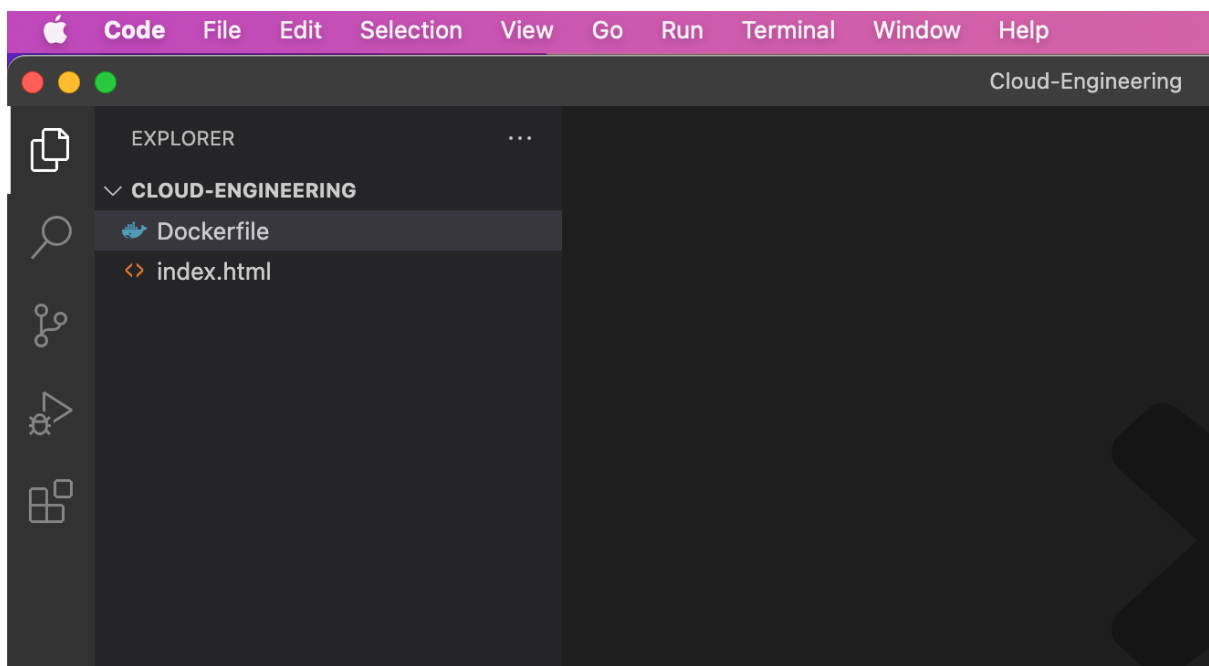


Figure 1: Folder structure of Imran Docker and Nginx practical demonstration.

The index.html file is the only content that the Nginx web server within the container serves. Figure 2 demonstrates that it was edited with the help of Visual Studio Code.

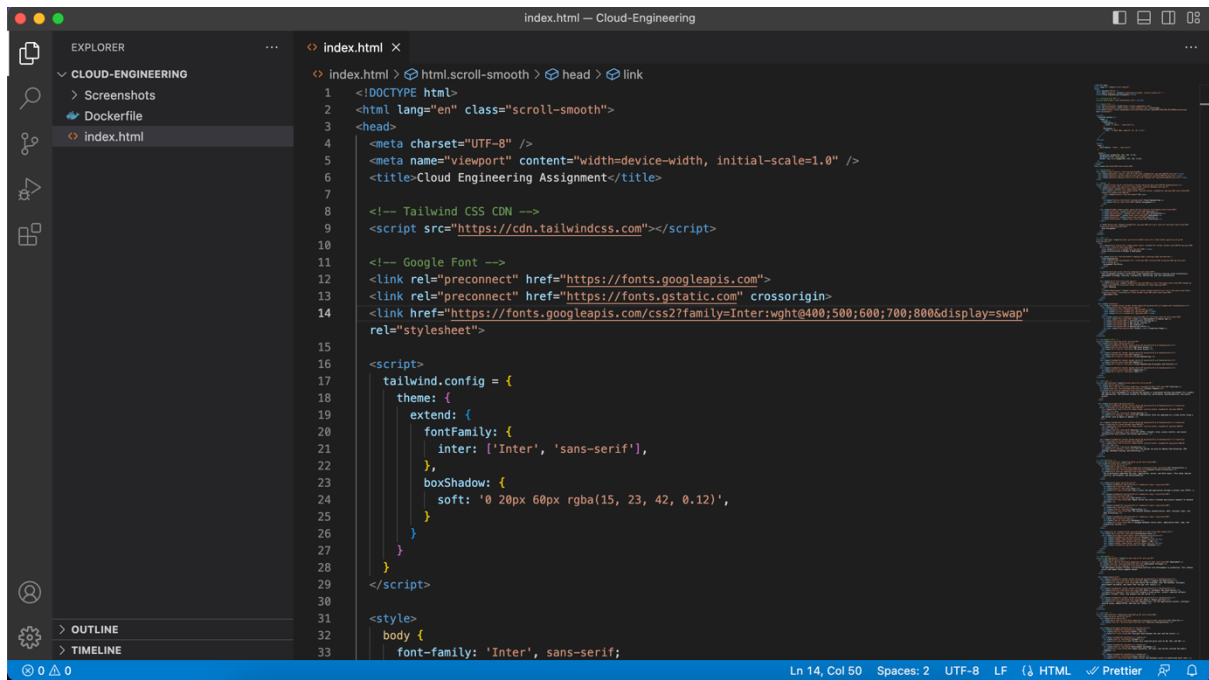


Figure 2: Imran cloud engineering assignment webpage opened in VS Code.

4.3 Dockerfile Construction

The declarative script out of which Docker constructs an image is called the Dockerfile. Every command in the Dockerfile builds a new layer in the resulting image, getting added to the final artefact [7]. The Dockerfile to be followed in this practical is the following:

FROM nginx:alpine

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

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

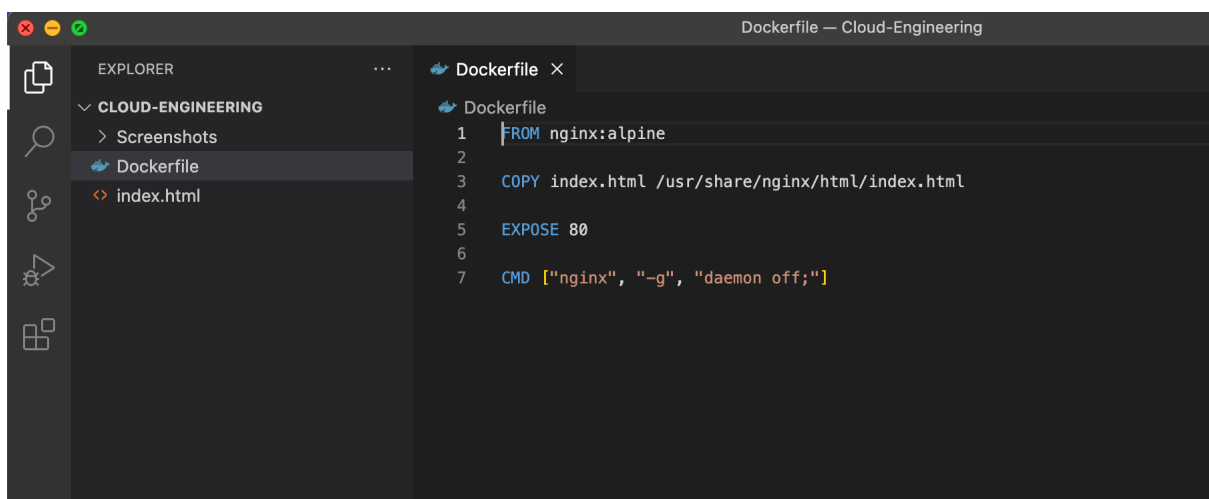
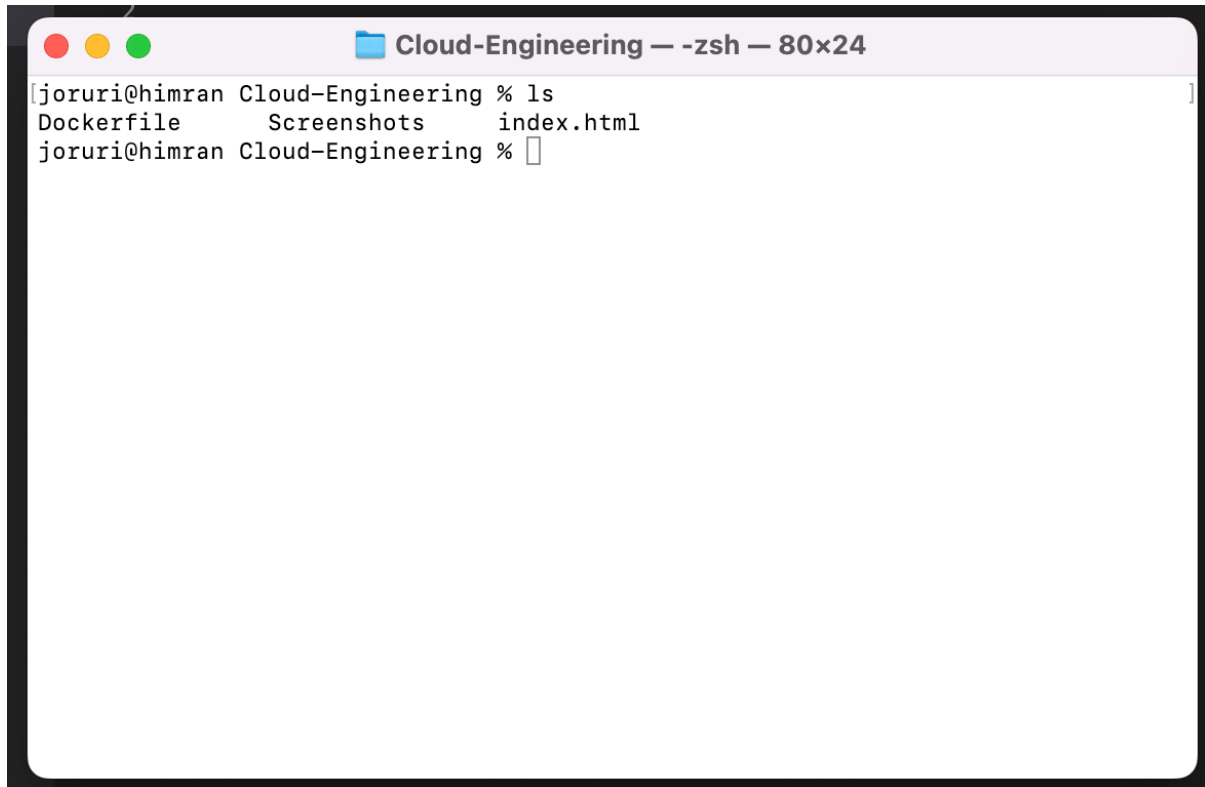


Figure 3: Dockerfile used to containerise the Imran static web page.

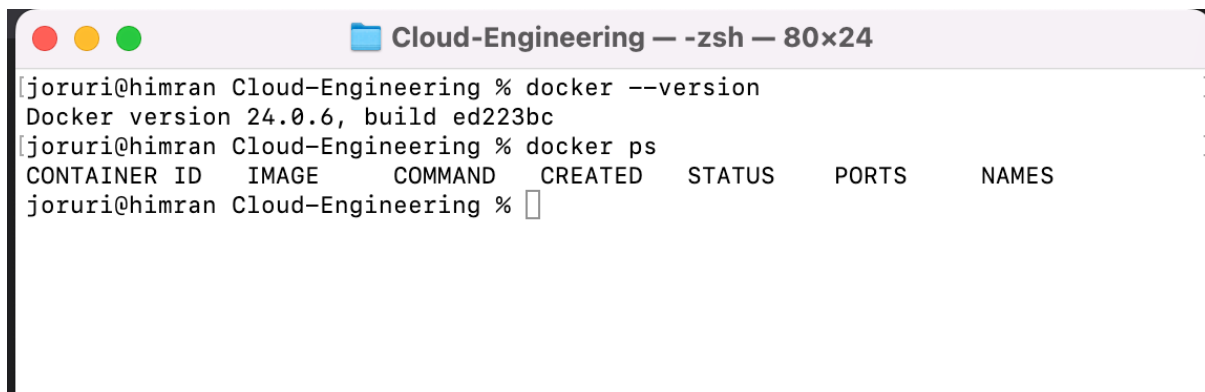
4.4 Building the Docker Image

Prior to constructing the image, the project files had been checked and the Dockerfile and index.html were both present in the working directory as indicated in Figure 4 Docker installation was also verified (Figure 5).

A terminal window titled "Cloud-Engineering — -zsh — 80x24" showing the output of the 'ls' command. The output lists three files: Dockerfile, Screenshots, and index.html.

```
[joruri@himran Cloud-Engineering % ls  
Dockerfile      Screenshots      index.html  
joruri@himran Cloud-Engineering % ]
```

Figure 4: Project files verified before Docker image build.

A terminal window titled "Cloud-Engineering — -zsh — 80x24" showing the output of 'docker --version' and 'docker ps'. The first command shows Docker version 24.0.6. The second command shows a table of running containers, which is currently empty.

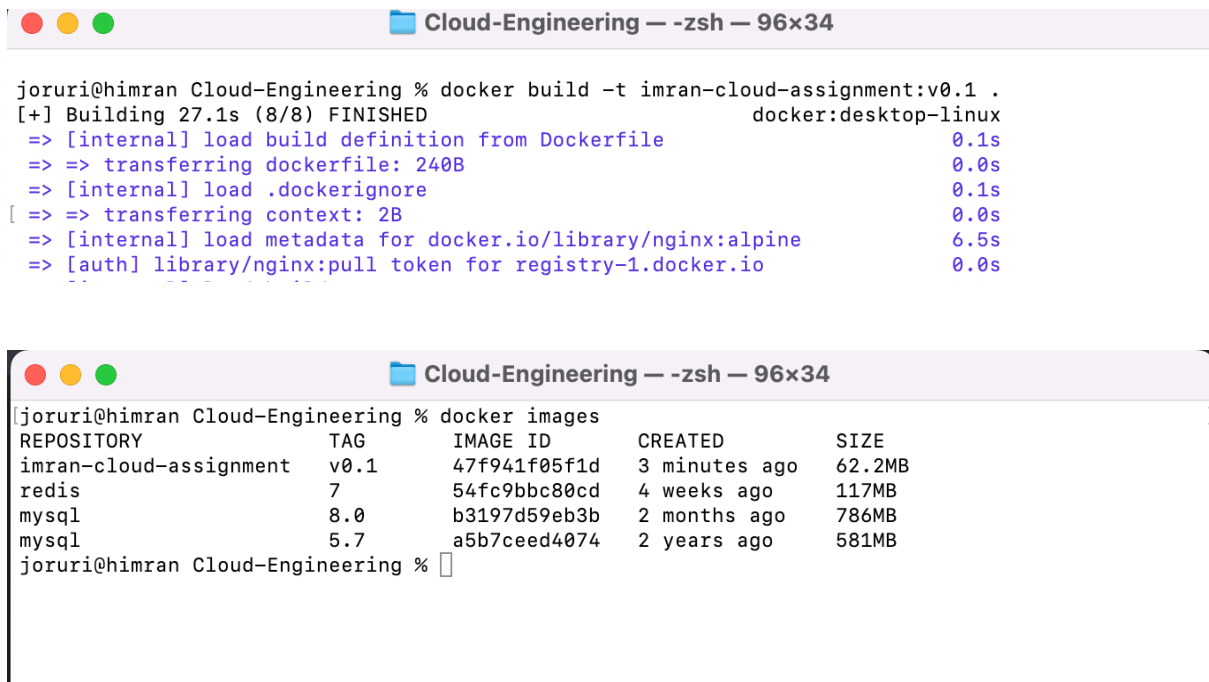
```
[joruri@himran Cloud-Engineering % docker --version  
Docker version 24.0.6, build ed223bc  
[joruri@himran Cloud-Engineering % docker ps  
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES  
joruri@himran Cloud-Engineering % ]
```

Figure 5: Docker installation and running environment verified.

The image was built using the following command:

```
docker build -t imran-cloud-assignment:v0.1
```

This command tells Docker to read the Dockerfile in the current directory (.) and build an image with a name of imran-cloud-assignment:v0.1. Each instruction is carried out one instruction at a time, and the result of the instruction is an intermediate layer. The resulting image was verified by use of docker images as illustrated by Figure 6.



```
joruri@himran Cloud-Engineering % docker build -t imran-cloud-assignment:v0.1 .
[+] Building 27.1s (8/8) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile              0.1s
=> => transferring dockerfile: 240B                             0.0s
=> [internal] load .dockerignore                                0.1s
=> => transferring context: 2B                                   0.0s
=> [internal] load metadata for docker.io/library/nginx:alpine  6.5s
=> [auth] library/nginx:pull token for registry-1.docker.io     0.0s

joruri@himran Cloud-Engineering % docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
imran-cloud-assignment  v0.1        47f941f05f1d     3 minutes ago   62.2MB
redis               7           54fc9bbc80cd     4 weeks ago     117MB
mysql               8.0         b3197d59eb3b     2 months ago    786MB
mysql               5.7         a5b7ceed4074     2 years ago     581MB
joruri@himran Cloud-Engineering %
```

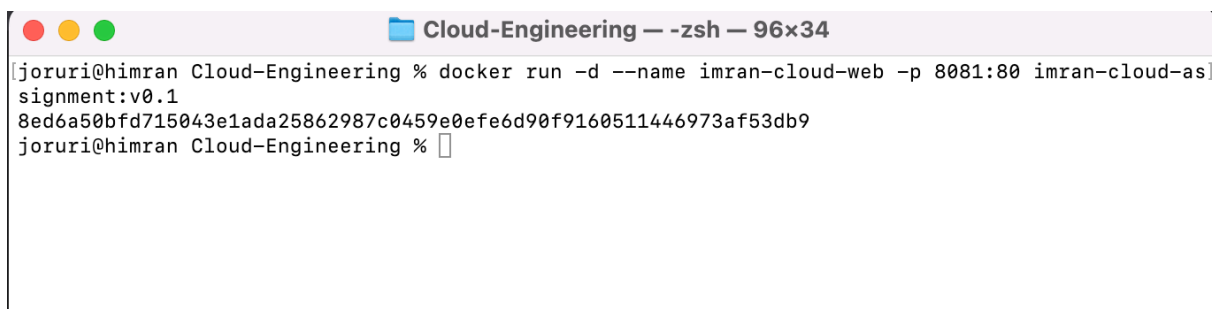
Figure 6: Docker image built for the Imran static assignment webpage.

It is notable that the layer caching mechanism of Docker implies that instructions that remain unchanged (such as FROM nginx:alpine) do not need to be re-executed on a later build, which can dramatically decrease build times when developing iteratively [7].

4.5 Running the Docker Container

```
docker run -d --name imran-cloud-web -p 8081:80 imran-cloud-assignment:v0.1
```

The -d flag puts the container in the background. The --name option gives the name imran-cloud-web. The flag maps of host port 8081 to container port 80 are defined by the -p 8081:80 flag. Figure 7 confirmed that imran-cloud-web was running, and this was confirmed using docker ps.



```
joruri@himran Cloud-Engineering % docker run -d --name imran-cloud-web -p 8081:80 imran-cloud-assignment:v0.1
8ed6a50bfd715043e1ada25862987c0459e0efe6d90f9160511446973af53db9
joruri@himran Cloud-Engineering %
```

Figure 7: Imran Nginx container running from the custom Docker image.

4.6 Browser Testing

The container was started and the portfolio webpage was opened in a web browser at the address: <http://localhost:8081>. Figure 8 indicates that the build pipeline - Dockerfile to image to container to HTTP response - worked as intended, end to end.

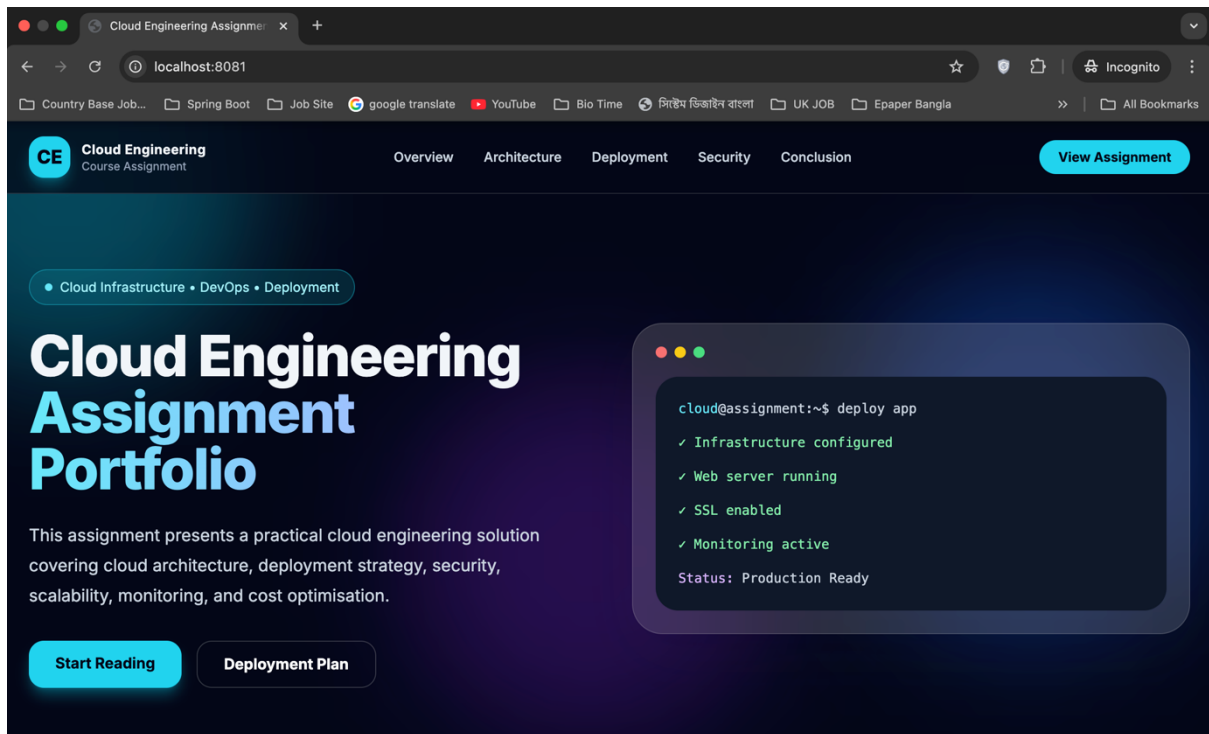


Figure 8: Imran cloud engineering webpage served from the Docker container.

4.7 Nginx Logs

Container logs were inspected using:

```
docker logs imran-cloud-web
```

Figure 9 illustrates the access log entries of the browser at Nginx generated by the browser through the HTTP GET requests. In cloud environments, log monitoring is crucial: logs offer visibility of operations, aid debugging, and serve as the source of alerting and audit trails [8]. Container logs would be sent to a centralised log management system like AWS CloudWatch, Azure Monitor or the ELK stack in production deployments.

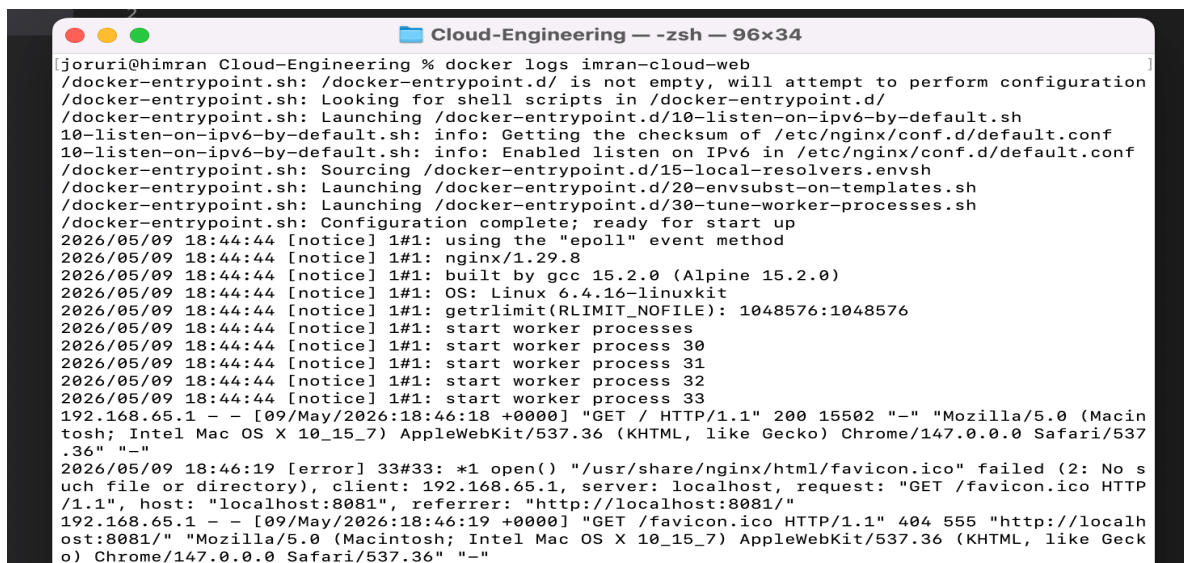


Figure 9: Nginx access logs generated by browser requests to the Docker container.

4.8 Docker Hub Publishing

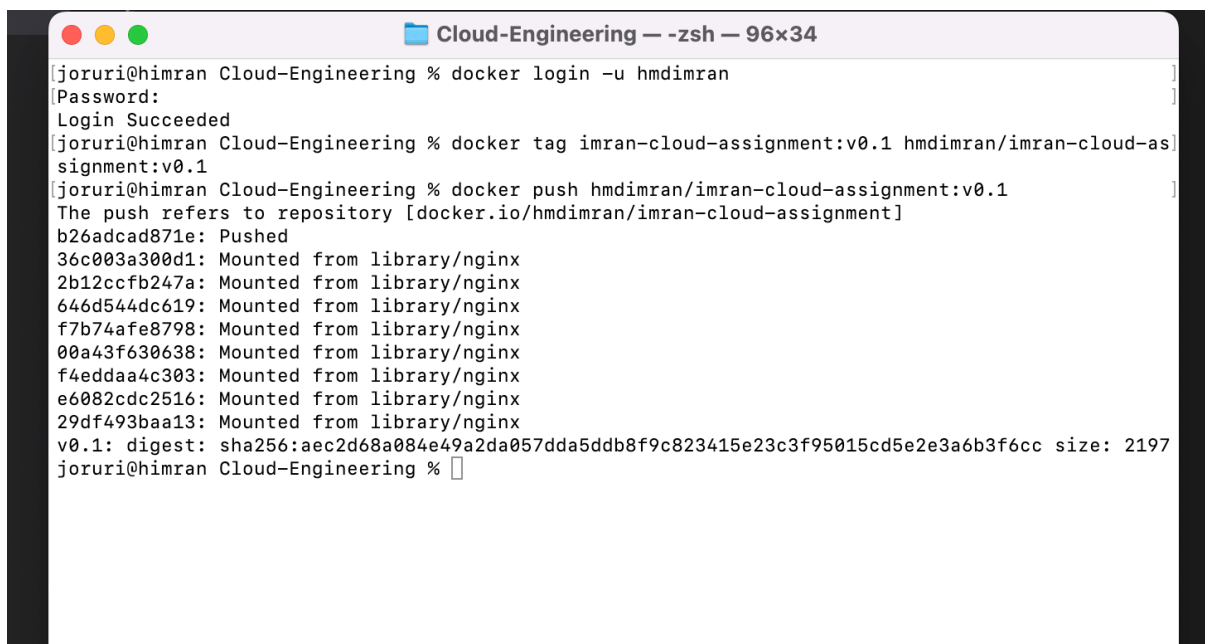
To enable the image to be used remotely, it was posted to Docker Hub under the account hmdimran. The following were the commands which were used:

```
docker login -u hmdimran
```

```
docker tag imran-cloud-assignment:v0.1 hmdimran/imran-cloud-assignment:v0.1
```

```
docker push hmdimran/imran-cloud-assignment:v0.1
```

The docker tag command is used to give an alias to the local image with a namespace prefix of the Docker Hub username. The docker push command is then used to push the tagged image layers into the registry. The fact that the image was uploaded successfully to the public Docker Hub repository is confirmed in figure 16.

A terminal window titled "Cloud-Engineering — zsh — 96x34" showing the execution of Docker commands. The user logs in as hmdimran, tags the local image imran-cloud-assignment:v0.1 with the namespace hmdimran/imran-cloud-assignment:v0.1, and pushes it to Docker Hub. The output shows the image being pushed layer by layer, with the final digest and size (2197) displayed.

```
joruri@himran Cloud-Engineering % docker login -u hmdimran
Password:
Login Succeeded
joruri@himran Cloud-Engineering % docker tag imran-cloud-assignment:v0.1 hmdimran/imran-cloud-assignment:v0.1
joruri@himran Cloud-Engineering % docker push hmdimran/imran-cloud-assignment:v0.1
The push refers to repository [docker.io/hmdimran/imran-cloud-assignment]
b26adcad871e: Pushed
36c003a300d1: Mounted from library/nginx
2b12ccfb247a: Mounted from library/nginx
646d544dc619: Mounted from library/nginx
f7b74afe8798: Mounted from library/nginx
00a43f630638: Mounted from library/nginx
f4eddaa4c303: Mounted from library/nginx
e6082cdc2516: Mounted from library/nginx
29df493baa13: Mounted from library/nginx
v0.1: digest: sha256:aec2d68a084e49a2da057dda5ddb8f9c823415e23c3f95015cd5e2e3a6b3f6cc size: 2197
joruri@himran Cloud-Engineering %
```

Figure 16: Imran Docker image published to Docker Hub.

By introducing the image to a registry, any authorized system - be it a remote cloud VM, a CI/CD pipeline runner, or a Kubernetes cluster - is able to push and pull the image with a single command, which shows the portability benefit of containerization [7].

4.9 Docker Volume Demonstration

By default, data written within a container is volatile: once the container has been removed, its filesystem is destroyed. Docker volumes solve this by offering a storage solution, which is independent of the lifecycle of any container [7]. The commands listed below were used to create and use a named volume:

```
docker volume create imran-nginx-html-volume
```

```
docker volume ls
```

```
docker run -d --name imran-volume-web -p 8082:80 -v imran-nginx-html-volume:/usr/share/nginx/html nginx:alpine
```

```
docker cp index.html imran-volume-web:/usr/share/nginx/html/index.html
```

Figure 10 shows the volume creation. Figure 11 shows the container running with the volume mounted.

```
Cloud-Engineering — -zsh — 96x34
joruri@himran Cloud-Engineering % docker volume create imran-nginx-html-volume
imran-nginx-html-volume
joruri@himran Cloud-Engineering % docler volume ls
zsh: command not found: docler
joruri@himran Cloud-Engineering % docker volume ls
DRIVER      VOLUME NAME
local       9c8a5af6fdebe8611f2234ca2c5cf5ee09f712ab0e2bd548c9082380700f78e3
local       55ad120efb78a0527484ef11584756a352a767a375086dd42ec68c9c10bc9208
local       7849711cfbde2aa6560980991a40c82717b2185b9a84038ff40cea9f4e4886c8
local       d95c391c908b351c4356e8475f626aa5fccb1cde98c6fa827c6fea493835db46
local       imran-nginx-html-volume
local       jobportal_my-db
```

Figure 10: Docker volume created for serving web content persistently.

```
Cloud-Engineering — -zsh — 117x31
joruri@himran Cloud-Engineering % docker run -d --name imran-volume-web -p 8082:80 -v imran-nginx-html-volume:/usr/share/nginx/html nginx:alpine
Unable to find image 'nginx:alpine' locally
alpine: Pulling from library/nginx
6a0ac1617861: Already exists
82736a35d0e7: Already exists
583599bb7d38: Already exists
aee4e54b3865: Already exists
781ff50d2644: Already exists
453da7dbc73e: Already exists
4a8b0b2a5b19: Already exists
612c0c1df4c5: Already exists
Digest: sha256:5616878291a2eed594aee8db4dade5878cf7edcb475e59193904b198d9b830de
Status: Downloaded newer image for nginx:alpine
6e524bf8cf6521cf550e0ac827e12b33bbdbb496a1c80811a8a981b60d54133f
joruri@himran Cloud-Engineering % docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS                NAMES
6e524bf8cf65   nginx:alpine  "/docker-entrypoint. ..."  19 seconds ago Up 17 seconds  0.0.0.0:8082->80/tcp  imran-volume-web
joruri@himran Cloud-Engineering %
```

Figure 11: Nginx container running with a Docker volume mounted to the web root.

The HTML file was copied into the volume of the running container, which was volume-mounted to the web root which was subsequently volume-mounted to the web root in a second container:

```
docker rm -f imran-volume-web
```

```
docker run -d --name imran-volume-web2 -p 8083:80 -v imran-nginx-html-volume:/usr/share/nginx/html nginx:alpine
```

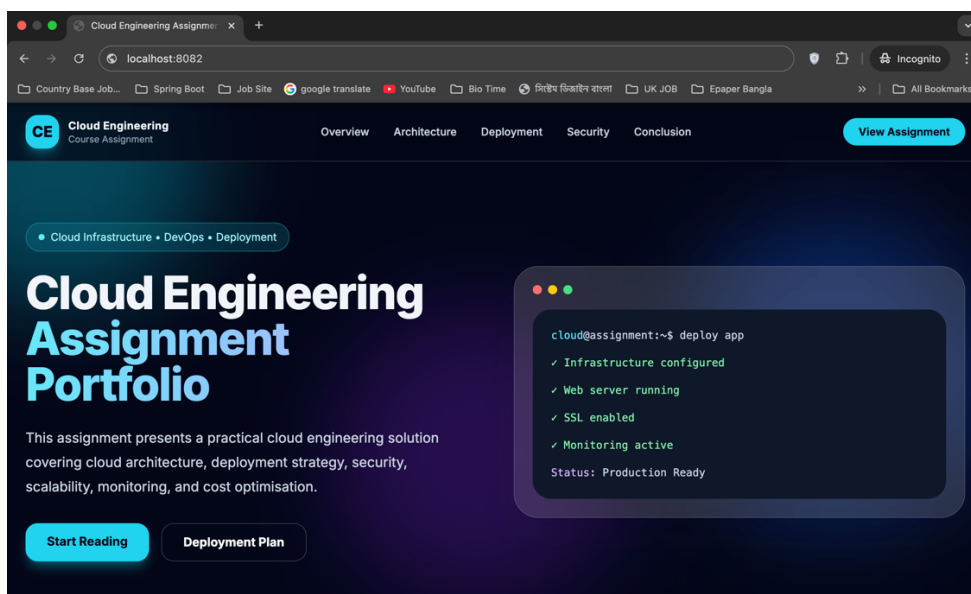


Figure 12: Imran HTML page served through a Docker volume mounted into Nginx.

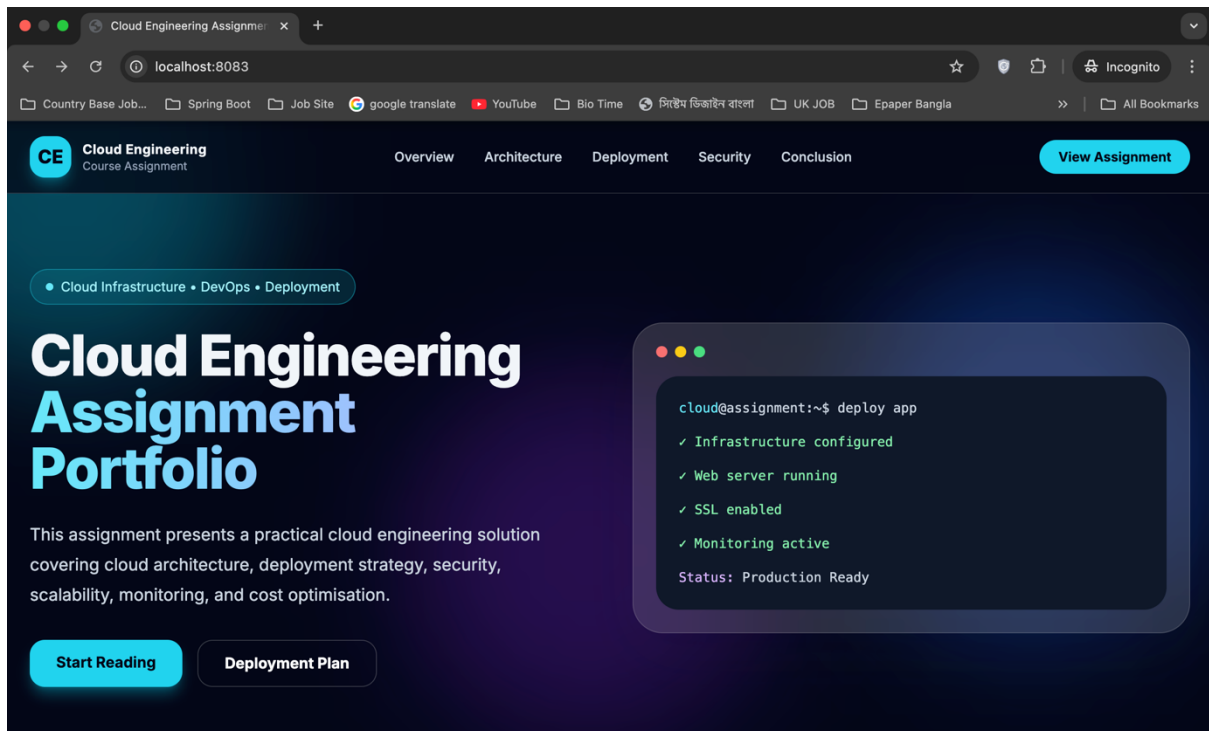


Figure 13: Web page content persisting after Nginx container recreation using Docker volume.

Figure 13 validates the assertion that the HTML page can still be accessed through the new container and this is the indication of data persistence regardless of the new container lifecycle. This trend is highly reminiscent of continued storage policies in cloud architecture as applied to application configurations, user uploads and database files.

4.10 Docker Network Demonstration

The Docker networks allow the controlled and isolated communication between the containers. In the absence of a common network, containers are isolated and have no name access to each other [7]. This was done by a custom bridge network made with:

```
docker network create imran-cloud-network
```

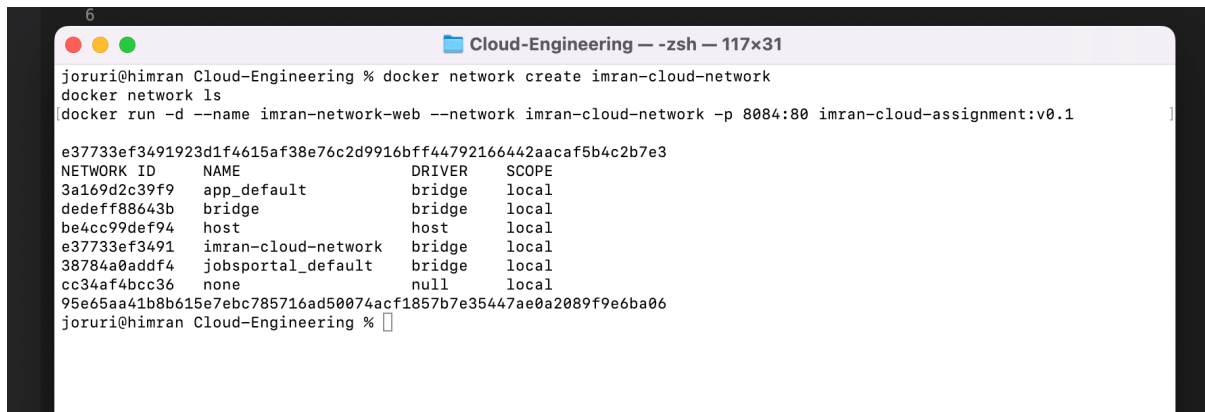
```
docker network ls
```

```
docker run -d --name imran-network-web --network imran-cloud-network -p 8084:80 imran-cloud-assignment-nginx:v1
```

The custom network that was developed is depicted in figure 14. A temporary container of Alpine Linux was then started on the same network to test container-to-container HTTP connectivity:

```
docker run -it --rm --network imran-cloud-network alpine sh
```

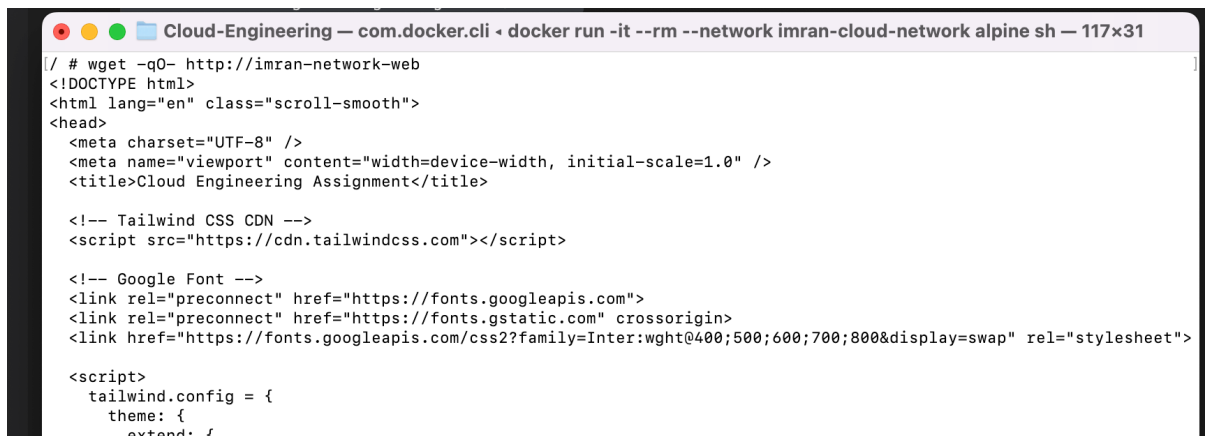
```
wget -qO- http://imran-network-web
```



```
Cloud-Engineering — -zsh — 117x31
joruri@himran Cloud-Engineering % docker network create imran-cloud-network
docker network ls
docker run -d --name imran-network-web --network imran-cloud-network -p 8084:80 imran-cloud-assignment:v0.1

e37733ef3491923d1f4615af38e76c2d9916bff44792166442aacf5b4c2b7e3
NETWORK ID      NAME                DRIVER  SCOPE
3a169d2c39f9    app_default         bridge  local
dedeff88643b    bridge              bridge  local
be4cc99def94    host                host    local
e37733ef3491    imran-cloud-network bridge  local
38784a0addf4    jobsportal_default  bridge  local
cc34af4bcc36    none                null    local
95e65aa41b8b615e7ebc785716ad50074acf1857b7e35447ae0a2089f9e6ba06
joruri@himran Cloud-Engineering %
```

Figure 14: Custom Docker network created for Imranweb containers.



```
Cloud-Engineering — com.docker.cli • docker run -it --rm --network imran-cloud-network alpine sh — 117x31
[/ # wget -qO- http://imran-network-web
<!DOCTYPE html>
<html lang="en" class="scroll-smooth">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Cloud Engineering Assignment</title>

  <!-- Tailwind CSS CDN -->
  <script src="https://cdn.tailwindcss.com"></script>

  <!-- Google Font -->
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link href="https://fonts.googleapis.com/css2?family=Inter:wght@400;500;600;700;800&display=swap" rel="stylesheet">

  <script>
    tailwind.config = {
      theme: {
        extend: {
```

Figure 15: Container-to-container communication tested through a custom Docker network.

Figure 15 confirms that the Alpine container was able to resolve the DNS built-in in Docker of the hostname-only HTML response of imran-network-web. This ability forms the basis of multi-container architectures, in which services — such as a web frontend, an API server, and a database - are isolated, and must communicate with each other without any ports exposed publicly.

4.11 Evaluation of the Practical Work

The real world experience demonstrated was able to accomplish its intended scope: it addressed the entire container workflow including image construction, running containers, persistent storage, networking, and registry publishing. It should be noted, though it is necessary to contextualise these steps in the manual with the overall picture of the production cloud engineering.

Container management using manual management is neither scalable nor manageable. In a real-world implementation with concurrent users and needing high availability, an orchestration platform, like Kubernetes, would be used to schedule containers, health check, rolling updates, and autoscaling automatically [6]. Moreover, the Docker image that will be used in this demonstration has not been scanned to identify any known vulnerabilities. To

identify the outdated or vulnerability packages in the Alpine or Nginx layers, docker scan or an integrated tool such as Trivy would be required in a professional setting [8].

Nonetheless, the practical effectively demonstrates the main principles of containerization and offers a replicable base of more complex, production-oriented deployments.

5. Cloud Computing Concepts: Load Balancing, Scalability and Application Gateways

In order to bring these ideas into a tangible form, we will use the example of an online learning portal - a system where thousands of students have access to lecture videos, deliver assignments and sit examinations. Such a platform could have a few hundred simultaneous users during quiet times. It can handle tens of thousands of concurrent requests at a time, during an examination window or in the beginning of a semester. Such situation predisposes load balancing, scalability and application gateways not only as desirable but also operationally important.

5.1 Load Balancing

A load balancer is a network device that directs a set of incoming client request to a pool of backend servers (or containers) based on a predefined algorithm [9]. The aim is to ensure that no one server becomes a bottleneck or failure point. Round-robin (requests are used in a sequence), least connections (request is sent to a server with the fewest active connections), and IP hash (request is sent to the same server to which the request has already been sent) are common load-balancing algorithms [9].

In the case of online learning portal, a load balancer placed in front of several Nginx application servers will ensure that the platform is responsive even when there are several instances of the server application that are experiencing high load or have failed. Cloud providers provide managed load-balancing services, namely AWS Elastic Load Balancing (ELB), Azure Load Balancer, and GCP Cloud Load Balancing which is built into autoscaling groups and container orchestration systems [10].

5.2 Scalability

Scalability is the ability of a system to accommodate more workload by increasing the resources of the system. Two basic methods include scaling up (vertical scaling) and scaling out (horizontal scaling) [11].

Table 4. Comparison of Vertical and Horizontal Scaling.

Attribute	Vertical Scaling (Scale Up)	Horizontal Scaling (Scale Out)
Mechanism	Upgrade existing server (more CPU, RAM)	Add more server instances or containers
Downtime	Often requires restart; potential downtime	Can be done live without interruption

Cost Model	High cost; diminishing hardware returns	Pay-as-you-grow; more granular cost control
Failure Impact	Single point of failure risk	Redundancy built in; failure of one node is tolerated
Use in Cloud	Useful for databases; limited scalability ceiling	Preferred approach for stateless web applications
Example	EC2 t3.medium → t3.large	Add 3 more EC2 instances behind a load balancer

In the case of the online learning portal, horizontal scaling is the strategy of choice: new containers or virtual machine instances are automatically provisioned when either CPU utilisation or request latency exceeds a threshold. This pattern is implemented natively by AWS Auto Scaling Groups and Azure Virtual machine Scale Sets [10][11].

5.3 Application Gateways

An application gateway is a routing system based on smart, content-sensitive routing of HTTP and HTTPS traffic to Layer 7 of the OSI model (the application layer) [12]. Compared to a simple load balancer, which is a Layer 4 (transport layer) load balancer that balances traffic based on IP and port only) an application gateway may inspect the content of requests, and may base routing decisions on URL path, HTTP headers, or cookies.

Within the online learning portal, requests to /videos/ + could be sent to a cluster of media streaming services, requests to /api/ + could be sent to a cluster of REST API backends, and requests to + could be sent to the main web frontend cluster. These features are available to Azure Application Gateway and AWS Application Load Balancer (ALB) to block unwanted traffic [10][12].

5.4 Why Load Balancing and Scalability Are Complementary

An architectural dependency exists between load balancing and scalability, in that neither can be used in isolation to achieve its full value [9]. A load balancer can distribute load among the available pool of servers, but the pool size is fixed, resulting in no possible way to avoid service degradation when total demand exceeds combined capacity. Horizontal scaling, that is, adding more servers, does not provide any performance advantage in a situation where all incoming traffic is still being directed to one overloaded server.

Scalability together with load balancing gives the dynamic capacity and intelligent distribution mechanism respectively. Since the online learning portal autoscales between five and twenty container instances during an examination event, the load balancer automatically adds the new instances to its routing pool, making sure that the additional capacity is immediately used and no instance is idle [9][11]. This demand-driven scaling and traffic distribution feedback loop is the key to resilient and elastic cloud architecture.

5.5 Benefits of Combining Load Balancing, Scalability and Application Gateways

Table 5. Benefits of Combining Load Balancing, Scalability and Application Gateways.

Benefit	Explanation
High Availability	If one container or instance fails, the load balancer automatically routes traffic to healthy instances, eliminating single points of failure.
Elastic Cost Management	Resources scale up during peak demand (e.g., examination periods) and scale down afterwards, aligning expenditure with actual usage.
Performance Optimization	Intelligent Layer 7 routing directs requests to the most appropriate microservice, reducing latency and improving user experience.
Security Enhancement	Application gateways with integrated WAF filter SQL injection, cross-site scripting, and other attacks before they reach application servers.
Operational Simplicity	Managed cloud services handle health checks, SSL termination, and traffic distribution automatically, reducing DevOps overhead.
Geographic Distribution	Combining global load balancers with regional autoscaling groups allows content to be served from the closest data centre to the end user.

6. Cloud Expansion Drivers and Serverless Functions

6.1 Main Driving Forces Behind Cloud Expansion

The cloud computing market in the world has since expanded out of a small technical offer to the major model of enterprise IT infrastructure. The growth has been fuelled by a number of converging forces [1][13]:

Table 6. Main Driving Forces Behind Cloud Computing Adoption.

Driver	Explanation
Cost Efficiency	Cloud is a substitute of the high capital expenditure (CapEx) on physical servers and data centers with operational expenditure (OpEx) on metered services. The organizations only pay what they use, which minimizes unutilized capacity.
Flexibility and Agility	New environments can be created by teams in a couple of minutes instead of weeks. This helps to shorten software development cycles, enables experimentation, and enables a quick response to business opportunities or threats.
Elastic Scalability	The resources of cloud platforms can be scaled on demand, making it technically and financially feasible to have start-ups manage an enterprise-grade workload without pre-buying infrastructure.
Global Reach	The providers of these services, including AWS, Azure, and GCP, run data centers in dozens of geographic locations, allowing organizations to serve customers with low latency, irrespective of their location.
Reliability and Resilience	Cloud networks have inherent redundancy, multi-zone failover and Service Level Agreements (SLAs), which guarantee uptime percentages, which would otherwise be prohibitively expensive to achieve in a private data center.

Innovation Acceleration	Managed services AI / ML API, managed databases, message queues and analytics platforms enable development teams to consume advanced functionality without deep expert knowledge.
Remote Work Enablement	The COVID-19 pandemic has caused an increased adoption of clouds as organizations need secure, globally available infrastructure to support distributed workforces [13].

6.2 Serverless Functions: AWS Lambda and Azure Functions

Serverless computing is an additional innovation to containers in the separation of infrastructure issues to the application developer. The idea of a serverless architecture is something of a misnomer, in that servers still exist, but the developer does not have any visibility of, or any responsibility towards, the underlying infrastructure [14]. The cloud provider takes complete control of the execution environment, such as provisioning, scaling, patching and decommissioning.

Under the serverless model, developers will deploy discrete units of code functions which are executed in response to triggers such as an HTTP request, a message queue event, a file upload or a scheduled time [14]. The first popularly adopted serverless platform was AWS Lambda, which was launched in 2014. Microsoft has a similar implementation, Azure Functions, which provides similar functionality and full integration with the Azure ecosystem [15]. Both platforms support a variety of programming languages (Python, Node.js, Java, C#) and can scale down to zero when not in use, which imposes no cost, and can help run thousands of instances of the functions it supports in parallel and on-demand.

One of the practical uses of online learning portal could be an AWS lambda function invoked every time a student submits an assignment. The role retrieves metadata in the submission in DynamoDB, and directly sends an acknowledgement email to the user via Amazon SES, without a permanent server or container to use between submissions.

6.3 Serverless Functions vs Containerization

Serverless and containerization are not mutually exclusive technologies but rather complementary to each other based on the nature of the workload [14][15]:

An evaluation of the serverless technology should be conducted critically considering the weaknesses and the strengths of the technology. The issue of vendor lock-in is also a major concern: code written to run on AWS Lambda employs AWS-specific APIs and triggers, and cannot be easily ported to Azure or GCP [14]. Latency introduced by cold starts Cold starts The latency introduced by cold starts as the cloud provider must provide a new execution environment to execute a user-facing application that has been idle. Moreover, serverless is not suitable when dealing with long-term workloads of data processing [15].

By contrast, containerization with Docker and orchestration with Kubernetes are more portable and flexible but at the cost of increased operational complexity. In the case of a system like the one that Imran has built, a portfolio web server, which is an old, stateful HTTP server, containerization is the more suitable architectural solution. In the case of event-driven tasks

that are not vital to the core functionality, serverless functions tend to be much more graceful and economical.

7. Conclusion

This report has discussed the theoretical basis as well as practical uses of the principles of virtualization, containerization as well as cloud computing in the context of the ACCA7027 module. Starting with the role of hypervisors to facilitate multi-tenant cloud infrastructure, the discussion can be continued by analyzing the architecture of Docker and the practical process of containerization which is carried out by Md Imran Biswas: the construction of the Dockerfile, the construction of a fully functional Nginx web server, the content volume persistence, the customization of the network communication, and the publicly available Docker Hub image of hmdimran/imran-cloud-assignment:v0.1.

The report then put containerization into context in broader cloud engineering concepts by illustrating using an online learning portal scenario how load balancing, horizontal scaling, and application gateways are complementary, interdependent layers in a resilient cloud architecture. Last, the drivers of cloud adoption were also surveyed and serverless computing (as exemplified by AWS Lambda and Azure Functions) was critically compared with the container-based solution as illustrated in the practical.

One of the themes that can be traced all through this analysis is the conflict between abstraction and control. Each new step in cloud technology - physical server to VMs, VMs to containers, containers to serverless - removes a degree of infrastructure complexity in the developer and introduces new constraints: less portability, less duration to execute, or reliance on managed services. The choice of the appropriate level of abstraction is thus not a technical binary but a subtle choice in architectural decisions based on the workload characteristics, operational capacity, cost models and organizational risk tolerance.

The practical exercise illustrations in this report are a valuable initial step in that path, and form the basis of the fundamental skills in container technology, on which more advanced cloud-native architectures can be built.

8. References

- [1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology,” National Institute of Standards and Technology, Sep. 2011. Available: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>
- [2] I. Periodicals, “IEEE REFERENCE GUIDE,” 2018. Available: <https://ieeauthorcenter.ieee.org/wp-content/uploads/IEEE-Reference-Guide.pdf>
- [3] R. Buyya, C. S. Yeo, and S. Venugopal, “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, Jun. 2009, doi: <https://doi.org/10.1016/j.future.2008.12.001>.
- [4] VMware, “What is a Hypervisor? | VMware Glossary,” *www.vmware.com*, 2024. <https://www.vmware.com/topics/hypervisor>
- [5] Amazon Web Services, “AWS Nitro System,” *Amazon Web Services, Inc.*, 2024. <https://aws.amazon.com/ec2/nitro/>
- [6] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” 2014. Available: <https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf>
- [7] Docker, “Docker Documentation,” *Docker Documentation*, Oct. 31, 2019. <https://docs.docker.com/>
- [8] OWASP, “Docker Security - OWASP Cheat Sheet Series,” *cheatsheetseries.owasp.org*, 2023. https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
- [9] F5 Networks, “What is a Load Balancer?,” *www.f5.com*, 2023. <https://www.f5.com/glossary/load-balancer>
- [10] Greg Lindsay, “What is Azure Application Gateway,” *learn.microsoft.com*, Sep. 27, 2023. <https://learn.microsoft.com/en-us/azure/application-gateway/overview>
- [11] Amazon Web Services, “AWS Auto Scaling,” *Amazon Web Services, Inc.*, 2018. <https://aws.amazon.com/autoscaling>
- [12] AWS, “What Is an Application Load Balancer? - Elastic Load Balancing,” *Amazon.com*, 2019. <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html>
- [13] Gartner, “Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly \$600 Billion in 2023,” *Gartner*, Apr. 19, 2023. <https://www.gartner.com/en/newsroom/press-releases/2023-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023>

[14] AWS, “What Is AWS Lambda? - AWS Lambda,” *Amazon.com*, 2025.
<https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>

[15] Microsoft, “Azure Functions Overview,” *learn.microsoft.com*, 2024.
<https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview>

Appendix A: Video Demonstration Link

Video Title: ACCA7027 Cloud Engineering - Docker and Nginx Practical Demonstration by Md Imran Biswas

Video Link: https://youtu.be/i93_V_ERZ_M

Duration: 7 minutes 15sec

Platform: Youtube