



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

Cloud Engineering Principles and Practice (ACCA7027)

Submitted by

Md Eyamin Molla

Student ID: 2425028

MSc in Cybersecurity & Digital Forensics

University Of Wales Trinity Saint David

Submission Date: 11-05-2026

Table of content

1. Introduction	3
2. Principles of Virtualisation.....	3
2.1 Virtualisation and Shared Infrastructure.....	3
2.2 Hypervisors and Resource Abstraction	4
2.3 Role of Virtualisation in Cloud Service Models	4
2.4 Constraints of VMs Compared with Containers	5
3. Containerisation and Docker in Cloud Engineering	5
3.1 What Containerisation Changes Compared with VMs	5
3.2 Docker Components and Architecture	5
3.3 Benefits for Repeatable Deployment and DevOps.....	6
3.4 Risks and Operational Limitations	6
4. Practical Docker Demonstration: Apache Cloud Operations Control Panel.....	7
4.1 Practical Objective and Artefact Overview	7
4.2 Dockerfile and Apache Image Definition.....	8
4.3 Build Verification and Local Image Creation	9
4.4 Container Execution and Browser Validation	10
4.5 Observability Through Apache Access Logs	11
4.6 Registry Publishing with Docker Hub	12
4.7 Persistent Content Using Docker Volumes.....	14
4.8 Service Connectivity Using Docker Networks.....	15
4.9 Critical Evaluation of the Practical Demonstration	16
5. Cloud Concepts Applied to a Smart Clinic Patient Portal	17
5.1 Load Balancing	17
5.2 Scalability.....	Error! Bookmark not defined.
5.3 Application Gateways	18
5.4 Why Load Balancing and Scaling Are Complementary	18
5.5 Benefits of Combining All Three.....	18
6. Cloud Expansion Drivers and Serverless Functions	19
6.1–6.4 Driving Forces Behind Cloud Expansion.....	19
6.5 Serverless Functions: AWS Lambda and Azure Functions	19
6.6 Comparison with Containerisation	20
7. Conclusion	20
8. References	21
Appendix A — Video Demonstration Link.....	22

1. Introduction

Cloud computing has essentially changed the way organizations are designing, deploying and managing their online infrastructure [1]. Within the last ten years, the need to switch on-premises physical servers to elastic, cloud-native deployments have been driven by an increasing demand of operational agility, cost efficiency, as well as the ability to dynamically scale services based on changing demand [2]. At the core of this revolution is virtualization - the technology which allows multiple isolated workloads to share a single physical host - and its more efficient successor, containerization, which eliminates the overhead of running a full operating system per workload [3].

The given report has been created in the context of ACCA7027 Cloud Engineering Principles and Practice module. It analyses the theoretical foundations of virtualization and containerization as well as proves these principles through an operation validation exercise which is practically carried out: the deployment of a Cloud Operations Control Panel using Docker and Apache HTTP Server [4], [5]. The word operations validation is not chosen randomly - every step of the deployment process, including image construction, container networking, etc., is not described but verified. This practice is a show of how cloud engineers test services in the workplace [4].

The report is designed in a way that it goes through theory, practice and then to wider application. The principles of virtualization and containerization are laid down in Section 2 and 3, including a critical discussion of the architecture of Docker, its benefits and limitations [3], [4]. Section 4 records the entire Docker process of deploying Apache: Docker file creation, image creation, container execution, access log monitoring, Docker Hub publishing, volume persistence and custom network connectivity [4], [5]. In section 5 and 6, these concepts are applied to a smart clinic patient portal scenario and to assess the load balancing, scalability, application gateways, drivers of cloud expansion, and serverless functions comparing the latter with containerization to evaluate the trade-offs involved in making decisions about cloud architecture [1], [6], [7]. In the text, the IEEE numeric referencing is applied following the IEEE Reference Guide [12].

2. Principles of Virtualization

2.1 Virtualization and Shared Infrastructure

- Explain what virtualization is: a software-based abstraction of physical hardware that enables more than one VM on a single host [1].
- Discuss multi-tenancy - multiple customers using the same physical infrastructure is safe [2].
- Without virtualization, the cloud providers had no way to provide elastic compute resources on-demand [1], [2].

Virtualization is the process of designing a software model of the physical computing resources - processors, memory, storage and networking [1]. An abstracted hardware, such as a software layer, allows a single physical host to support multiple independent virtual machines (VMs) running simultaneously, each with the illusion of exclusive access to the underlying hardware [2].

This is the feature which the use of cloud computing rests on its core value proposition: without virtualization, cloud providers like Amazon Web Services, Microsoft Azure, and Google Cloud Platform will not be able to offer the elastic, on-demand compute allocation which define cloud computing [1], [2].

2.2 Hypervisors and Resource Abstraction

Type 1 (bare-metal): operates directly on hardware - VMware ESXi, KVM, Xen; used in cloud data centers [4].

Type 2 (hosted): is implemented on the top of an OS VirtualBox, VMware Workstation; can be used in local testing [4].

AWS EC2 is based on the Nitro System - hardware-accelerated virtualization [4].

The hypervisor is in charge of physical hardware resource allocation to the virtual machines and comes in two major forms [4]. Type 1 (bare-metal) hypervisors directly run on bare hardware and are favored in enterprise cloud data centers due to their high performance and security features [4]. Hosted (type 2) hypervisors are installed on an existing operating system, and are suitable to developer workstations and local test environments.

Feature	Type 1 Hypervisor (Bare-metal)	Type 2 Hypervisor (Hosted)
Definition	Runs directly on 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
Use Case	Enterprise data centres, cloud infrastructure	Developer workstations, testing, local labs
Examples	VMware ESXi, Microsoft Hyper-V, KVM, Xen	Oracle VirtualBox, VMware Workstation, Parallels
Security	Stronger — fewer software layers between VM and hardware	Weaker — host OS vulnerabilities may affect VMs
Cost	Higher upfront infrastructure investment	Lower — leverages existing OS

Table 1. Type 1 vs Type 2 Hypervisors [4].

2.3 Role of Virtualization in Cloud Service Models

- IaaS: AWS EC2, Azure VMs -customers pay to rent virtualized compute without purchasing hardware [1].
- Live migration: A live VM can be migrated between physical hosts without causing any downtime [4].
- Resource pooling and multi-tenancy that the hypervisor layer makes possible [1], [2].

All the significant models of cloud services are based on virtualization [1]. The Infrastructure as a Service (IaaS) model is a type of cloud computing whereby, cloud providers present virtualized compute instances to their customers, allowing them to select CPU, memory and storage configurations without having to purchase physical hardware [1], [2]. Live migration - moving a running VM between physical hosts without disrupting the service of the VM being migrated - further supports high-availability architectures and planned maintenance [4].

2.4 Constraints of VMs Compared with Containers

- Both VMs boot a complete operating system, a gigabyte of storage, and several minutes to boot up [3].
- Duplication of resources: having several OS stacks on the same host wastes both CPU and memory [3].
- These shortcomings encouraged the use of container technology, which is explained in Section 3.

Irrespective of their usefulness, virtual machines have a non-negligible architectural overhead. Each VM is a full OS kernel, with typical storage requirements of several gigabytes, and booting times of several minutes [3]. This extremely heavyweight character renders VMs ill-adapted to situations where quick autoscaling is essential. It is these restrictions that drove the creation of container technology, or rather discussed in the next section [3].

3. Containerization and Docker in Cloud Engineering

3.1 What Containerization Changes Compared with VMs

- Containers all share the host OS kernel - no individual OS per container - only milliseconds to startup [3].
- Contains the application + libraries + config but NOT a full OS [6]
- This means that Docker (2013) standardized and introduced container technology and the image format [3].

Containerization is a virtualization mechanism at the OS-level that packages an application with all of its dependencies, such as libraries, configuration files, and its runtime environment, into a self-contained, portable unit [6]. In contrast to a virtual machine, a container shares the host OS kernel, which is much lighter, quicker to startup, and resource-efficient [3]. It was standardized by Docker, which was first released in 2013, resolving the long-standing issue of inconsistency in the environments [3].

3.2 Docker Components and Architecture

Docker operates through a client-server architecture: the Docker client sends commands to the Docker daemon (dockerd), which manages images, containers, networks, and volumes on the host [4]. Images are stored locally or retrieved from a registry such as Docker Hub [4].

Component	Role in Docker Architecture
Docker Daemon (dockerd)	Background service managing containers, images, volumes and networks on the host [4]
Docker Client (CLI)	Command-line interface through which the engineer sends build, run and push instructions [4]
Docker Image	Read-only, layered snapshot containing the application and its dependencies [4]
Docker Container	A running instance of an image — isolated, ephemeral by default [4]
Dockerfile	Script of sequential build instructions; each line creates a new image layer [4]

Docker Hub	Public/private cloud registry for distributing and versioning Docker images [4]
Docker Volume	Persistent storage mechanism that survives container deletion; managed by the daemon [4]
Docker Network	Virtual network for controlled, isolated communication between containers [4]

Table 2. Docker Components and Roles [4].

3.3 Benefits for Repeatable Deployment and DevOps

- Portability: same image performs the same way on laptop as on cloud [3], [6].
- Speed: startup of milliseconds containers compared to minutes in VMs - ideal in autoscaling [3].
- Immutability: tagged image (v1) does not change, it is predictable, and can be rolled back deployments [4].
- Devops consistency: the same image that has been tested is promoted to production without changes [3].

Containerization provides a number of benefits compared to deployments based on VM [6]. Portability guarantees that an image that is developed on a workstation of a developer behaves in the same manner when deployed in a cloud instance, irrespective of the underlying OS distribution [3]. An immutable tagged Docker image, like `eyamin-cloud-ops-apache:v1`, is one that cannot be changed in any manner (immutable). An immutable tagged Docker image, such as `eyamin-cloud-ops-apache:v1`, can not be changed in any way (immutable).

3.4 Risks and Operational Limitations

- Shared kernel: a vulnerability at the kernel level can impact all the containers on the host at the same time [10].
- Image vulnerabilities: images in the public registry can include out-of-date software layers [10].
- Ephemeral storage: the data within a container is lost when it is deleted unless a volume is mounted [4].
- The level of production has to be orchestrated with Kubernetes or Docker Swarm [11].

Containerization is not risk free. The shared kernel model has a security implication: since a kernel-level vulnerability can, in theory, affect all containers on that host, it cannot affect all containers on a host simultaneously, as VMs do [10]. Images obtained by means of public registries might also contain old-fashioned or unprotected software layers, and regular image scanning is an important operation practice [10]. On a production scale, the manual execution of Docker commands becomes operationally unsustainable; tools like Kubernetes are typically used to coordinate the activity of the container lifecycle management, scaling and self-healing [11].

4. Practical Docker Demonstration: Apache Cloud Operations Control Panel

4.1 Practical Objective and Artefact Overview

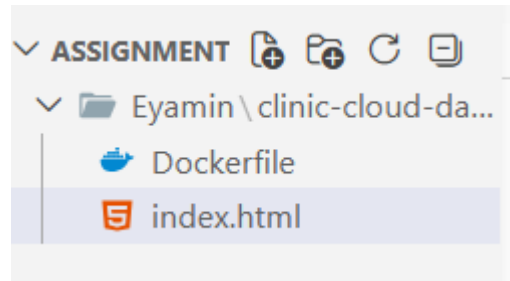


Figure 1: Folder structure for Md Eyamin Molla's Docker and Apache practical demonstration.

```
index.html x Dockerfile
Eyamin > clinic-cloud-dashbaord > index.html > html > body.bg-slate-100.text-slate-800.min-h-screen > main.max-w-5xl.mx-auto.px-4.py-8.space-y-8
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8" />
5 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6 <title>Cloud Operations Control Panel - ACCA7027</title>
7 <script src="https://cdn.tailwindcss.com"></script>
8 </head>
9 <body class="bg-slate-100 text-slate-800 min-h-screen">
10
11 <!-- HEADER -->
12 <header class="bg-slate-900 text-white px-6 py-6 border-b-4 border-amber-400">
13 <div class="text-amber-400 text-xs font-semibold uppercase tracking-widest mb-1">ACCA7027 - Cloud Engineering</div>
14 <div class="text-2xl font-bold mb-1">Cloud Operations Control Panel</div>
15 <div class="text-sm mb-4">Docker-Based Web Deployment for ACCA7027 Cloud Engineering</div>
16 <div class="flex flex-wrap gap-2 text-xs font-semibold">
17 <span class="bg-green-600 text-white px-3 py-1 rounded-full">✓ Container Ready</span>
18 <span class="bg-sky-600 text-white px-3 py-1 rounded-full">✓ Web Server Active</span>
19 <span class="bg-amber-500 text-white px-3 py-1 rounded-full">✓ Cloud Deployment Practice</span>
20 <span class="bg-violet-600 text-white px-3 py-1 rounded-full">✓ Monitoring Enabled</span>
21 </div>
22 </header>
23
24 <main class="max-w-5xl mx-auto px-4 py-8 space-y-8">
25
26 <!-- STUDENT INFO -->
27 <section class="bg-white rounded-xl shadow p-6 border border-slate-200">
28 <div class="text-xs font-bold uppercase tracking-widest text-slate-400 mb-4">Student Information</div>
29 <div class="grid grid-cols-2 sm:grid-cols-4 gap-4 text-center">
30 <div class="bg-slate-50 rounded-lg p-4 border border-slate-200">
31 <div class="text-xs text-slate-400 mb-1">Student Name</div>
32 <div class="font-bold text-blue-000 text-sm">Md Eyamin Molla</div>
33 </div>
34 <div class="bg-slate-50 rounded-lg p-4 border border-slate-200">
35 <div class="text-xs text-slate-400 mb-1">Module Code</div>
36 <div class="font-bold text-blue-000 text-sm">ACCA7027</div>
37 </div>
38 </div>
39 </section>
40
41 <!-- PRACTICAL AIM -->
42 <div class="bg-white rounded-xl shadow p-6 border border-slate-200">
43 <div class="text-sm font-bold text-blue-000 mb-4">Practical Aim</div>
44 <div class="text-xs text-slate-400">This page is a static web application prepared for containerised deployment using Docker. The practical covers building a Docker image, running a container, browser testing, publishing to Docker Hub, and using volumes and networks for persistent storage and inter-container communication.</div>
45 </div>
46
47 <!-- DOCKER DEPLOYMENT WORKFLOW -->
48 <div class="bg-white rounded-xl shadow p-6 border border-slate-200">
49 <div class="text-sm font-bold text-blue-000 mb-4">Docker Deployment Workflow</div>
50 <div class="flex justify-between text-xs">
51 <div class="bg-blue-50 text-blue-000 font-semibold px-4 py-2">HTML Page</div>
52 <div class="bg-yellow-50 text-yellow-000 font-semibold px-4 py-2">Dockerfile</div>
53 <div class="bg-orange-50 text-orange-000 font-semibold px-4 py-2">Image Build</div>
54 <div class="bg-green-50 text-green-000 font-semibold px-4 py-2">Container</div>
55 <div class="bg-purple-50 text-purple-000 font-semibold px-4 py-2">Browser</div>
56 <div class="bg-pink-50 text-pink-000 font-semibold px-4 py-2">Docker Hub</div>
57 </div>
58 </div>
59 </main>
60 </body>
61 </html>
```

Figure 2: Cloud Operations Control Panel HTML page opened in VS Code.

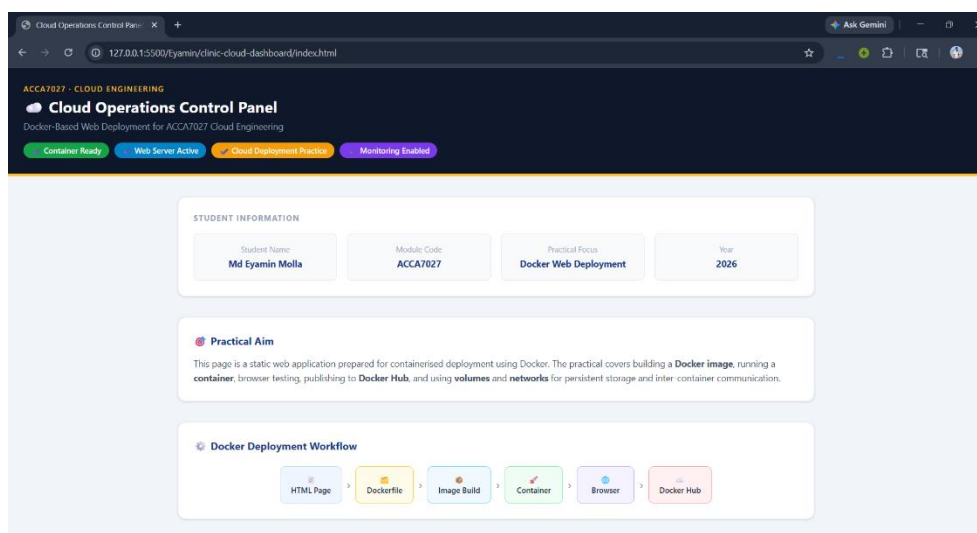


Figure 3: Cloud Operations Control Panel tested locally before Docker containerization.

- Explain the Cloud Operations Control Panel - a fixed HTML dashboard which is served by Apache HTTP Server.
- Explain: the HTML file was originally tested locally prior to containerization - it is essential to demonstrate the content correctness first.
- Two artefacts: the HTML page (index.html), and the Dockerfile.

In a bid to illustrate how containerization can be applied in practice, a simple HTML dashboard the Cloud Operations Control Panel was created and containerized using Docker and Apache HTTP Server [4], [5]. The project was structured in an easily understood directory hierarchy, as shown in Figure 1, which has the Dockerfile and index.html in the same working directory. Before containerization, the dashboard was initially tested in a local browser as indicated in Figure 3 making sure that the HTML content was valid before being packaged [4].

4.2 Dockerfile and Apache Image Definition

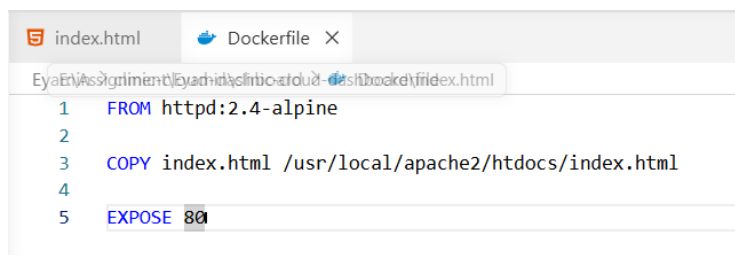


Figure 4: Docker file used to containerize Md Eyamin Molla’s Apache web page.

- httpd:2.4-alpine: the official Apache image on Alpine Linux - minimal size, reduced attack surface [5].
- Apache document root:/user/local/apache2/htdocs - default directory served by httpd [5].
- EXPOSE 80: declarative documentation alone -p at runtime with -p [4]
- None of the CMD instructions were necessary - the official image of Apache, used in the front, will automatically start Apache in the foreground.

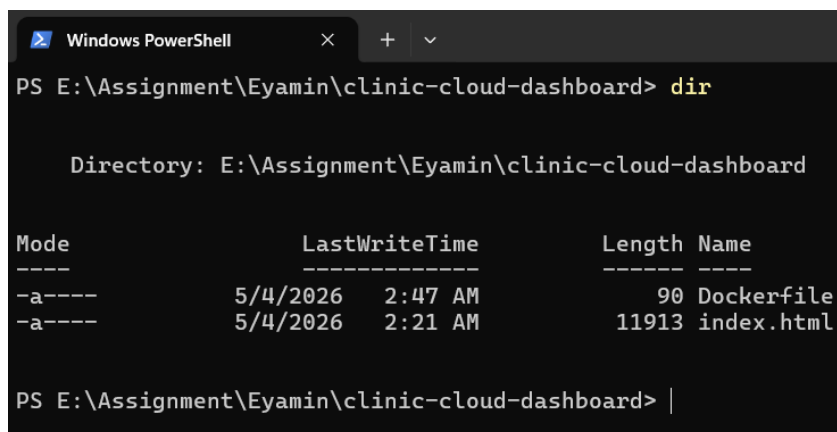
Instruction	Explanation
FROM httpd:2.4-alpine	Establishes the baseline image for Apache HTTP Server 2.4 on Alpine Linux. Alpine uses a minimal footprint (~50 MB) to produce a web server with a smaller footprint, including build time and attack surface. httpd:2.4 is a production-grade, mature, web server with a smaller footprint, including build time and attack surface. httpd:2.4 is a production-grade, mature web server that has a smaller footprint, including both build time and attack surface.

COPY	index.html /usr/local/apache2/htdocs/index.html	Copies the Cloud Operations Control Panel HTML file into the Apache document root in the container, overwriting the default welcome page [5].
EXPOSE 80		Documents that the container listens on port 80. This is declarative - it does not make the port visible to the host. Port mapping is set at the run time with -p 8091:80 [4].

Table 3. Explanation of the Apache Docker file [4], [5].

The Docker file is based on the official image of the Apache Software Foundation, which is the default htdocs directory containing the default static content: [5]. Contrary to an image based on Nginx, the Apache httpd image does not need to have an explicit CMD instruction in it - the official image is configured to run the Apache daemon in the foreground automatically, which is the most correct behavior of a Docker container [4], [5].

4.3 Build Verification and Local Image Creation



```

Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> dir

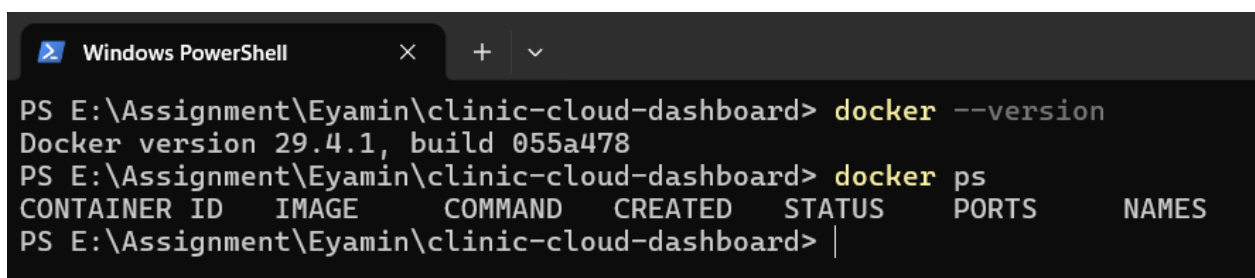
Directory: E:\Assignment\Eyamin\clinic-cloud-dashboard

Mode                LastWriteTime         Length Name
----                -
-a-----         5/4/2026   2:47 AM             90 Dockerfile
-a-----         5/4/2026   2:21 AM          11913 index.html

PS E:\Assignment\Eyamin\clinic-cloud-dashboard> |

```

Figure 5: Project files verified before building the Apache Docker image.



```

Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker --version
Docker version 29.4.1, build 055a478
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS   NAMES
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> |

```

Figure 6: Docker installation and running environment verified.

```

PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker build -t eyamin-cloud-ops-apache:v1 .
[+] Building 17.5s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 127B
=> [internal] load metadata for docker.io/library/httpd:2.4-alpine
=> [auth] library/httpd:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/2] FROM docker.io/library/httpd:2.4-alpine@sha256:968c8b4098fcec473762b45f6c541a
=> => resolve docker.io/library/httpd:2.4-alpine@sha256:968c8b4098fcec473762b45f6c541a
=> sha256:b990da80e2107a23042852f28ec7a4b255081f2e48da22ee89466 11.07MB / 11.07MB
=> sha256:4f4fb708ef54461cfa02571ae8db9a8dc1e0cd5577484a6d75e68dc38eacc1 32B / 32B
=> sha256:7b3c5d639c1194129f70ca528c7c5aa08d1ada88fc8d8c16423b51de3e 5.90MB / 5.90MB
=> sha256:4ca6f6797e45f405f90851d91732e15b07e8990f66fd0aee8106ace2cebb37 286B / 286B
=> sha256:719e56c8c985fd8fb6194fe85949486f3e41afc09e04ab9abaffcbbb5df9c 144B / 144B
=> sha256:41c15f5ba5843d4eed3ceeb41954062982e98c0a43aa6783d25dc27d4c194 930B / 930B
=> extracting sha256:41c15f5ba5843d4eed3ceeb41954062982e98c0a43aa6783d25dc27d4c194
=> extracting sha256:719e56c8c985fd8fb6194fe85949486f3e41afc09e04ab9abaffcbbb5df9c
=> extracting sha256:4f4fb708ef54461cfa02571ae8db9a8dc1e0cd5577484a6d75e68dc38eacc
=> extracting sha256:b990da80e2107a23042852f28ec7a4b255081f2e48da22ee89466e6362c2
=> extracting sha256:7b3c5d639c1194129f70ca528c7c5aa08d1ada88fc8d8c16423b51de3e979ac
=> extracting sha256:4ca6f6797e45f405f90851d91732e15b07e8990f66fd0aee8106ace2cebb37e
=> [internal] load build context
=> => transferring context: 11.95kB
=> [2/2] COPY index.html /usr/local/apache2/htdocs/index.html
=> exporting to image
=> exporting layers
=> exporting manifest sha256:f9af4d4fb5983d7545fb7c2a5da0e55ff73e7934595a98ed1d25c49
=> exporting config sha256:8b5d944711e1249794f38b454c17444f2377e9cf084add7aa3073a17
=> exporting attestation manifest sha256:3dd0ae8fbfb8d8376818cd992380b83693ab5f1b175
=> exporting manifest list sha256:fa0b33f3b3c3497bd1268d2ff208d4de38cb2b7ee129bd6afa
=> naming to docker.io/library/eyamin-cloud-ops-apache:v1
=> unpacking to docker.io/library/eyamin-cloud-ops-apache:v1
PS E:\Assignment\Eyamin\clinic-cloud-dashboard>

```

Figure 7: Docker image built for Md Eyamin Molla’s Apache cloud operations dashboard.

```

PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker images

```

IMAGE	ID	DISK USAGE	CONTENT SIZE	EXTRA
eyamin-cloud-ops-apache:v1	fa0b33f3b3c3	96.7MB	20.8MB	

```

PS E:\Assignment\Eyamin\clinic-cloud-dashboard>

```

Figure 8: Apache Docker image listed after successful build.

- Docker build executes the instructions in the Docker file sequentially, creating a new layer each time an instruction is executed [4].
- The trailing dot (.) defines the build context - the directory that contains the Docker file and index.html.
- -t flag is used to set a versioned artefact of deployment [4]--v1.
- Docker images will ensure that the image has been created and capture its ID and size.

After building, it was confirmed that the project files had been verified prior to building, as can be seen in Figure 5, and that the Docker system was operational, as can be seen in Figure 6. The command displayed above was used to build the image as shown in Figure 7 [4]. The resulting image was verified using Docker images (Figure 8), to establish a reproducible, versioned deployment artefact tagged eyamin-cloud-ops-apache:v1 [4].

4.4 Container Execution and Browser Validation

```

PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker run -d --name eyamin-apache-web -p 8091:
80 eyamin-cloud-ops-apache:v1
43d6f279e9e107b05bd1d52173100f8f5602f80e37af11de73137b45063c9daf
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker ps

```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
43d6f279e9e1	eyamin-cloud-ops-apache:v1	"httpd-foreground"	20 seconds ago	Up 18 seconds

```

PS E:\Assignment\Eyamin\clinic-cloud-dashboard>

```

Figure 9: Apache container running from the custom Docker image.

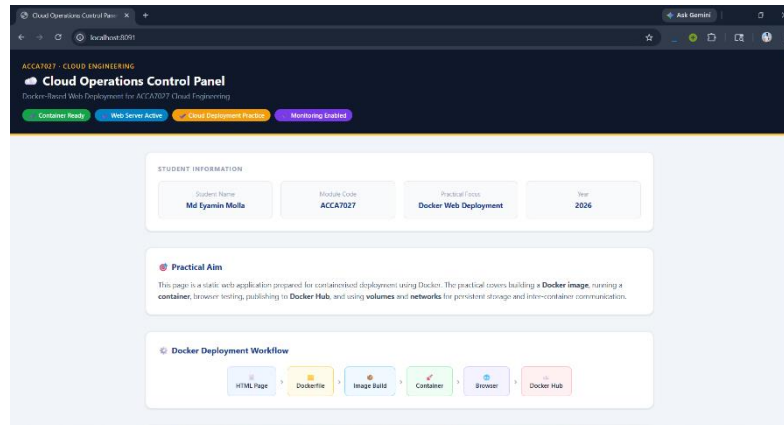


Figure 10: Md Eyamin Molla's Cloud Operations Control Panel served from an Apache Docker container.

- -d: detached mode -d: container operates in the background and frees up the terminal to issue further commands [4].
- --name eyamin-Apache-web: attaches a human-readable name to identify it and retrieve logs.
- -p 8091:80: maps host port 8091 to container port 80 — browser accesses `http://localhost:8091`
- as of 4: the docker ps command confirms that the container is in the state of Up and that it has the right port binding [4].

The container was started in detached mode (-d), with the port 8091 on the host mapped to the port 80 inside the container (-p 8091:80) [4]. As seen in Figure 9, docker ps was used to verify that the container was running and that it had port binding as expected. The Cloud Operations Control Panel (Fig. 10) is correctly rendering in the browser at the URL of `http://localhost:8091`, which validates the fact that Apache was indeed serving the containerized HTML dashboard [4], [5].

4.5 Observability Through Apache Access Logs

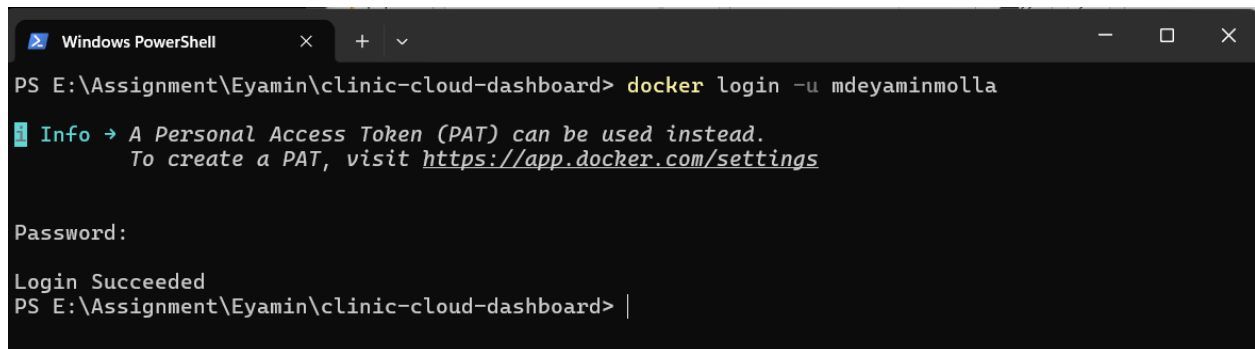
```
Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker logs eyamin-apache-web
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 172.17.0.2. Set the 'ServerName' directive globally to suppress this message
[Sun May 03 21:07:41.940414 2026] [mpm_event:notice] [pid 1:tid 1] AH00489: Apache/2.4.66 (Unix) configured -- resuming normal operations
[Sun May 03 21:07:41.940870 2026] [core:notice] [pid 1:tid 1] AH00094: Command line: 'httpd -D FOREGROUND'
172.17.0.1 - - [03/May/2026:21:09:09 +0000] "GET / HTTP/1.1" 200 11913
172.17.0.1 - - [03/May/2026:21:09:11 +0000] "GET /favicon.ico HTTP/1.1" 404 236
172.17.0.1 - - [03/May/2026:21:10:01 +0000] "-" 408 -
PS E:\Assignment\Eyamin\clinic-cloud-dashboard>
```

Figure 11: Apache access logs generated by browser requests.

- Apache logs all requests in Combined Log Format: client IP, time, HTTP method, path, status code, bytes [5].
- docker logs: Retrieves the stdout of the container, where access logs are written by default in Apache.
- The logs can be used to show that the browser request has physically arrived at the containerized Apache service.
- In production, logs would be sent to a centralized system like AWS CloudWatch or ELK stack.

Apache HTTP Server automatically records all incoming HTTP requests in Combined Log Format [5]. According to the analysis of Docker logs eyamin-Apache-web the entry of the browser request at localhost:8091 into the server indicated that the server received, processed, and responded to the browser request with an HTTP 200 status code. This log evidence is vital in operational validation: it does not just show that the container is in operation, but also that the Apache service that is within the container is actively accepting requests [4], [5].

4.6 Registry Publishing with Docker Hub



```

Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker login -u mdeyaminmolla

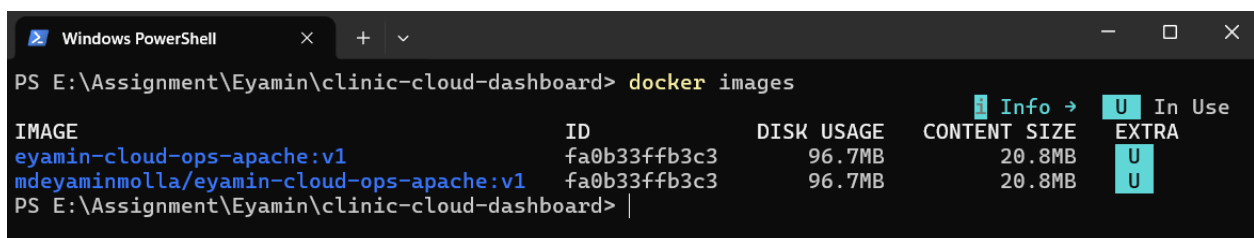
Info → A Personal Access Token (PAT) can be used instead.
        To create a PAT, visit https://app.docker.com/settings

Password:

Login Succeeded
PS E:\Assignment\Eyamin\clinic-cloud-dashboard>

```

Figure 12: Docker Hub login completed for publishing the Apache image.



```

Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker images

IMAGE                                ID                                DISK USAGE  CONTENT SIZE  EXTRA
eyamin-cloud-ops-apache:v1           fa0b33ffb3c3                     96.7MB      20.8MB        U
mdeyaminmolla/eyamin-cloud-ops-apache:v1 fa0b33ffb3c3                     96.7MB      20.8MB        U
PS E:\Assignment\Eyamin\clinic-cloud-dashboard>

```

Figure 13: Docker image tagged for Docker Hub publishing.

```
Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker push mdeyaminmolla/eyamin-cloud-ops-apache:v1
The push refers to repository [docker.io/mdeyaminmolla/eyamin-cloud-ops-apache]
719e56c8c985: Pushed
4f4fb700ef54: Pushed
7b3c5d639c11: Pushed
e1a4731f5772: Pushed
b989da80e210: Pushed
352f015f534a: Pushed
4ca6f6797e45: Pushed
6a0ac1617861: Mounted from mdimranbiswas/imran-cloud-assignment-nginx
41c15f5ba584: Pushed
v1: digest: sha256:fa0b33ffb3c3497bd1268d2ff208d4de38cb2b7ee129bd6afaa24781c2198072 size: 856
PS E:\Assignment\Eyamin\clinic-cloud-dashboard>
```

Figure 14: Apache Docker image pushed to Docker Hub.

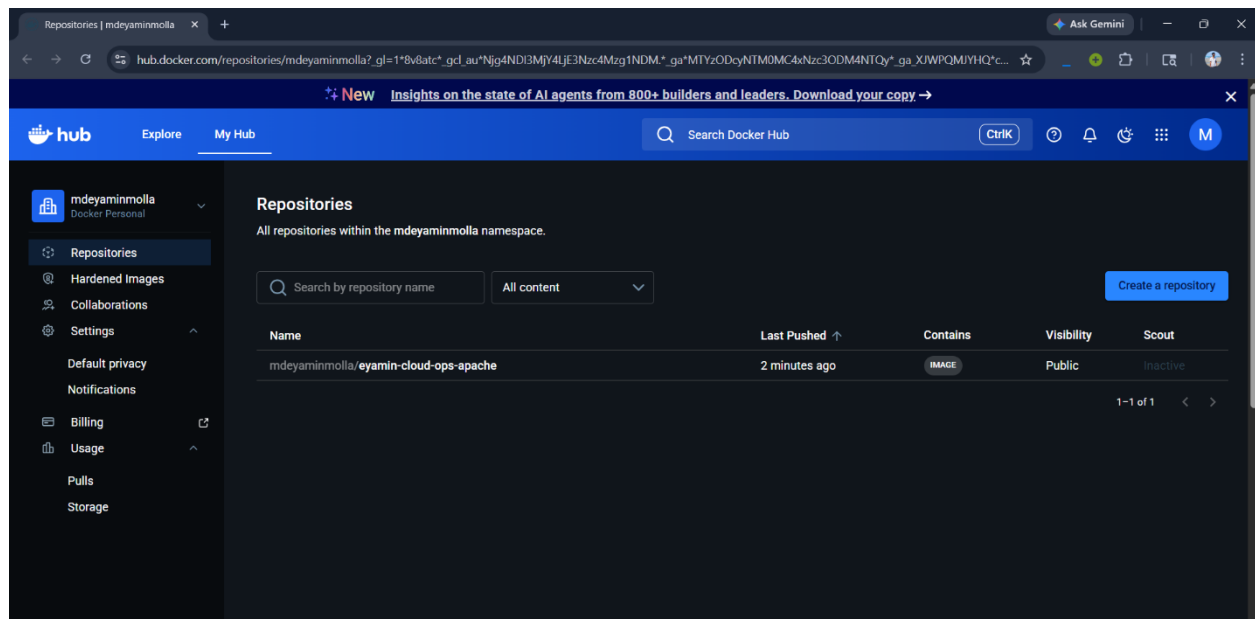


Figure 15: Docker Hub repository for Md Eyamin Molla’s Apache image.

- docker logout ensures no stale credentials before fresh authentication [4]
- docker tag creates a registry-path reference required for pushing to Docker Hub [4]
- docker push uploads each image layer — Figure 14 shows each layer being transmitted [4]
- Figure 15 (Docker Hub repository) confirms the image is publicly accessible from any cloud platform [4]

Publishing the image to Docker Hub transforms it from a local build artefact into a globally accessible deployment resource [4]. Following authentication (Figure 12), the image was tagged with the Docker Hub username prefix (Figure 13) and pushed layer-by-layer, as shown in Figure 14. Figure 15 confirms the image is now available in the Docker Hub registry under the account [INSERT USERNAME], enabling reproducible deployment on any cloud instance or CI/CD pipeline that can reach Docker Hub [4].

4.7 Persistent Content Using Docker Volumes

```
Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker volume create eyamin-apache-html-volume
eyamin-apache-html-volume
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker volume ls
DRIVER      VOLUME NAME
local       eyamin-apache-html-volume
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> |
```

Figure 16: Docker volume created for persistent Apache web content.

```
Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker rm -f eyamin-apache-web
eyamin-apache-web
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker run -d --name eyamin-volume-web -p 8092:
80 -v eyamin-apache-html-volume:/usr/local/apache2/htdocs httpd:2.4-alpine
Unable to find image 'httpd:2.4-alpine' locally
2.4-alpine: Pulling from library/httpd
Digest: sha256:968c8b4098fcec473762b45f6c541a3b2b2cfab2caccb1edbd2cece071ef160
Status: Downloaded newer image for httpd:2.4-alpine
172e767ae3e6323b42dd3a656a51bcd8df87800a267d9b4fb92e7be5464c0197
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker cp index.html eyamin-volume-web:/usr/loc
al/apache2/htdocs/index.html
Successfully copied 11.9kB (transferred 13.8kB) to eyamin-volume-web:/usr/local/apache2/htdocs/
index.html
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> |
```

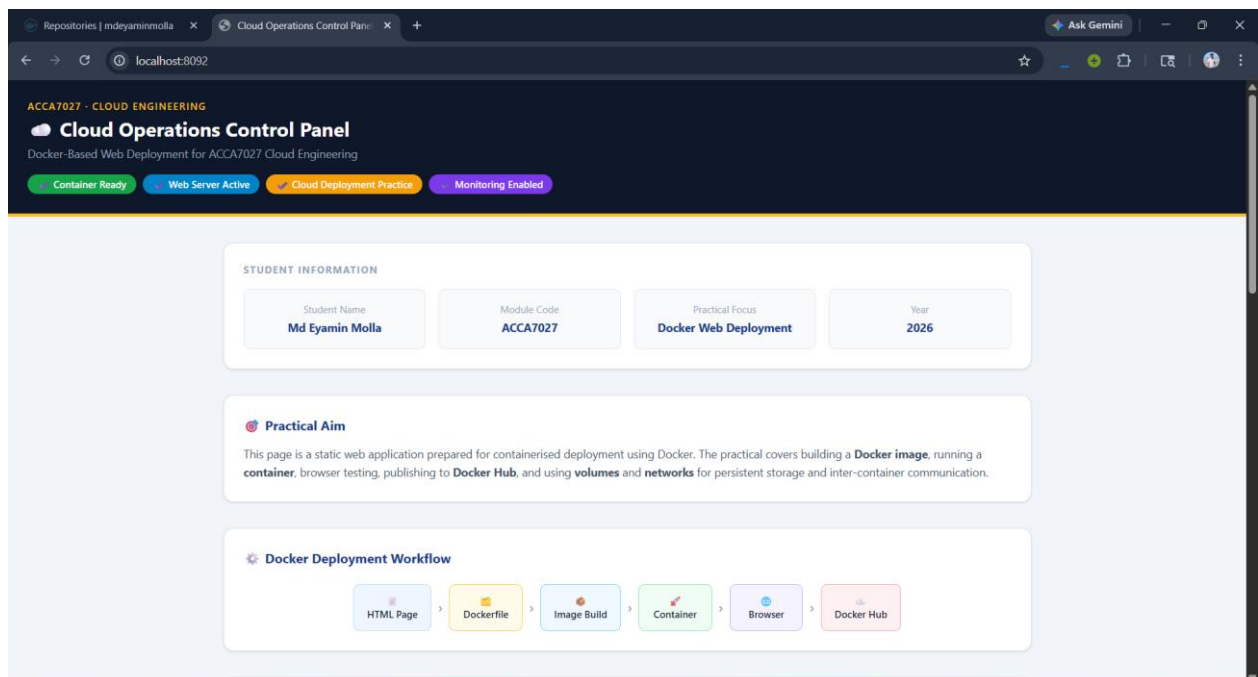


Figure 17: Cloud Operations Control Panel served through a Docker volume at localhost:8092.

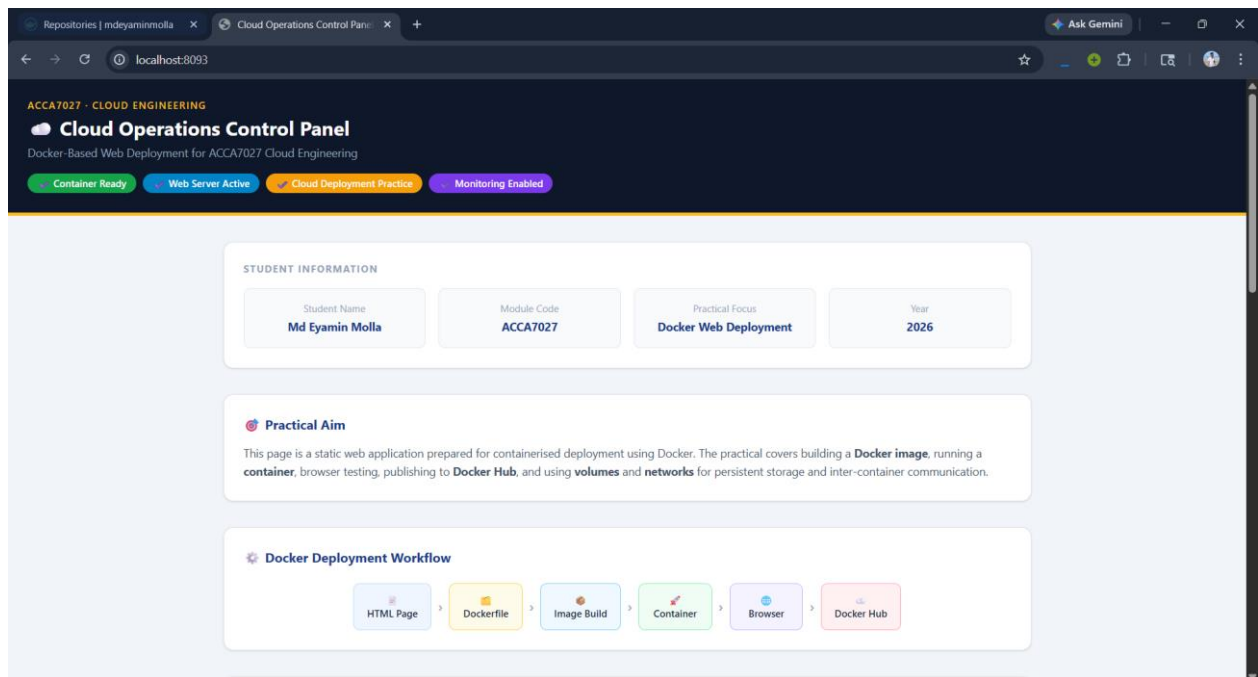


Figure 18: Web page content persisting after Apache container recreation.

Docker volumes are a persistent storage system that is operated by the Docker daemon; volume data is stored external to the container filesystem and is also resilient to container destruction [4]. After making the named volume eyamin-Apache-html-volume, and copying index.html into the mount path, the container was intentionally deleted with `docker rm -f`. Another new container (eyamin-volume-web2) was then created, mounting the same volume, and the Cloud Operations Control Panel continued to be accessible at localhost:8093, as shown in Figure 18 - confirming that the volume continued to persist data independently of the container lifecycle [4].

4.8 Service Connectivity Using Docker Networks

```

Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker network create eyamin-cloud-network
249425b414accfa76d5ebe4b13c26509191fb05c555bcb5b190905a45091194d
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
69cdcc0745c9        bridge              bridge              local
249425b414ac        eyamin-cloud-network bridge              local
43e56c7b8bdb        host                host                local
4de3af798eb4        none                null                local
PS E:\Assignment\Eyamin\clinic-cloud-dashboard>

```

Figure 19: Custom Docker network created for Md Eyamin Molla's Apache demonstration.

```
Windows PowerShell
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker run -d --name eyamin-network-web --network eyamin-cloud-network -p 8094:80 eyamin-cloud-ops-apache:v1
f9914be9c8e57280a69089d7d69c72bde77d379d2e373bf415c7319c9a95e19f
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                               NAMES
f9914be9c8e5   eyamin-cloud-ops-apache:v1         "httpd-foreground"      6 seconds ago Up 4 seconds  0.0.0.0:8094->80/tcp               eyamin-network-web
PS E:\Assignment\Eyamin\clinic-cloud-dashboard> docker run -it --rm --network eyamin-cloud-network alpine sh
/ # wget -qO- http://eyamin-network-web
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title>Cloud Operations Control Panel - ACCA7027</title>
<script src="https://cdn.tailwindcss.com"></script>
</head>
<body class="bg-slate-100 text-slate-800 min-h-screen">
<!-- HEADER -->
<header class="bg-slate-900 text-white px-6 py-6 border-b-4 border-amber-400">
<p class="text-amber-400 text-xs font-semibold uppercase tracking-widest mb-1">ACCA7027 · Cloud Engineering</p>
<h1 class="text-2xl font-bold mb-1">☛ Cloud Operations Control Panel</h1>
<p class="text-slate-400 text-sm mb-4">Docker-Based Web Deployment for ACCA7027 Cloud Engineering</p>
<div class="flex flex-wrap gap-2 text-xs font-semibold">
<span class="bg-green-600 text-white px-3 py-1 rounded-full">✔ Container Ready</span>
<span class="bg-sky-600 text-white px-3 py-1 rounded-full">✔ Web Server Active</span>
<span class="bg-amber-500 text-white px-3 py-1 rounded-full">✔ Cloud Deployment Practice</span>
<span class="bg-violet-600 text-white px-3 py-1 rounded-full">✔ Monitoring Enabled</span>
</div>
</header>
<main class="max-w-5xl mx-auto px-4 py-8 space-y-8">
<!-- STUDENT INFO -->
<section class="bg-white rounded-xl shadow p-6 border border-slate-200">
<h2 class="text-xs font-bold uppercase tracking-widest text-slate-400 mb-4">Student Information</h2>
<div class="grid grid-cols-2 sm:grid-cols-4 gap-4 text-center">
<div class="bg-slate-50 rounded-lg p-4 border border-slate-200">
<p class="text-xs text-slate-400 mb-1">Student Name</p>
<p class="font-bold text-blue-900 text-sm">Md Eyamin Molla</p>
</div>
<div class="bg-slate-50 rounded-lg p-4 border border-slate-200">
<p class="text-xs text-slate-400 mb-1">Module Code</p>
</div>
</div>
</section>
</main>
```

Figure 20: Container-to-container communication tested using Alpine and Apache.

The embedded DNS service in Docker assigns name to IP address by mapping container names to IP addresses within a custom network, which allows containers to interact using name instead of an IP address [4]. An Alpine test container was launched on the same eyamin-cloud-network and used wget to request content on the http://eyamin-network-web. The successful response in HTML proves the existence of the Apache container accessible by its DNS name [4].

4.9 Critical Evaluation of the Practical Demonstration

What Was Successfully Demonstrated

- Repeatable Apache image created based on a minimal and declarative Docker file [4], [5].
- The observability of operations using Apache access logs [5].
- Registry publishing Image is available worldwide using Docker Hub [4].
- Persistence of Data across container lifecycle with a named Docker volume [4].
- Container-to-container DNS-based communication over a bespoke bridge network [4].

What Remains Limited or Absent

- None of the orchestration layers: manual commands are not scalable to workloads of production processes [11].
- No automatic health checks, restart policies or liveness checks [11]
- None of the image vulnerability scanning of the Apache base image [10].
- None of the HTTPS/TLS termination, all the traffic is plaintext HTTP.
- There is no centralized logging pipeline - logs are only present within the running container.

The hands-on experiment was effective in validating the entire Docker workflow to deploy an Apache-based deployment. The Apache container, deployed in a production cloud environment, would be managed by Kubernetes [11], exposed using an Application Gateway with TLS termination [7], scanned for vulnerabilities before deployment [10], and monitored using a centralized logging service. The manual Docker commands that are carefully used in this

demonstration have an educational value - each command makes an otherwise invisible operation transparent and verifiable.

5. Cloud Concepts Applied to a Smart Clinic Patient Portal

5.1 Load Balancing

- Allocates requests presented to a container across several container instances or VMs [6].
- Smart clinic situation: thousands of patients all at once, making appointments in the mornings or during health campaigns.
- Azure Load Balancer / AWS ELB: health checks identify unhealthy instances and automatically eliminate them [6].

Load balancing spreads the received network load among several backend servers, so that no one server becomes a bottleneck [6]. Demand peaks are predictable in the smart clinic patient portal: the appointment booking windows in each morning, the launch of public health campaigns, the deadlines of document uploads can cause sudden spikes in the number of visits to the portal services. Azure Load Balancer allocates requests to a pool of unhealthy backend instances by configurable algorithms (e.g., round-robin or least-connections) and uses health probes to automatically eliminate unhealthy backend instances in the pool - maintaining availability without human intervention [6].

5.2 Scalability

- Vertical scaling: scale CPU/RAM of an established instance - implementable hardware limit [6].
- Horizontal scaling: more replicas of containers - theoretically infinite, cost-effective [6].
- Examples in the clinic: the campaign to vaccinate against the flu: a boost in appointment booking traffic; the end of the quarter: the spike of document uploads.

Dimension	Vertical Scaling (Scale Up)	Horizontal Scaling (Scale Out)
Approach	Increase CPU/RAM of existing instance	Add more instances of the same service
Cloud example	Upgrade Azure VM from B2s to D4s tier	Increase container replicas via Kubernetes HPA or Azure VMSS
Limit	Physical hardware ceiling — cannot scale indefinitely	Theoretically unlimited — limited only by load balancer
Cost model	Higher cost per instance; idle when demand drops	Pay-as-you-scale; instances removed when demand drops
Risk	Single point of failure if instance fails	Resilient — traffic redistributed if one instance fails
Best for clinic	Database servers needing high single-thread performance	Patient portal web tier during appointment booking peaks

Table 6. Vertical vs Horizontal Scaling.

5.3 Application Gateways

- Layers 7 (HTTP/HTTPS) - routing decisions are made based on URL path, hostname, headers [7].
- TLS termination: decrypts HTTPS at the gateway, and forwards plain HTTP internally [7].
- WAF: filters SQL injection, XSS, and other attacks before they get to portal services [7].

URL Path	Backend Service	Purpose
/Appointments	Appointment booking container pool	Schedule and manage patient appointments
/Records	Medical records service	Upload and retrieve patient documents securely
/Billing	Billing and payments service	Invoice access and payment processing
/Notifications	Notification service	Appointment reminders and public health alerts

Azure Application Gateway works on Layer 7, so routing by URL path can be used to route /appointments requests to the booking service, /records to the medical records service, and /billing to the payment service [7]. The decryption of incoming HTTPS traffic at the gateway means that the processing load is centrally distributed, and the individual containers on the backend do not have to process the entire load. The embedded Web Application Firewall (WAF) blocks malicious traffic - such as SQL injection and cross-site scripting attacks - before they can access patient data [7].

5.4 Why Load Balancing and Scaling Are Complementary

Scaling adds new capacity (more instances); load balancing allows the capacity to be used.

WITHOUT load balancing, New instances will not be seen by the incoming traffic.

They make up an elastic system together: capacity is automatically expanded/contracted, and traffic is always distributed.

Scalability and load balancing are not conflicting strategies, but they are mutually reliant mechanisms [6]. Scaling introduces a new capacity - more container replicas - but without a load balancer, requests will still flow to the original instance, and the new capacity will be idle. However, a load balancer running on a fixed pool of servers is unable to accommodate permanent growth in demand. The two technologies, together, comprise an elastic system: the capacity scales automatically in response to demand, and the load balancer ensures that all the provisioned instances are actively used [6], [7].

5.5 Benefits of Combining All Three

Benefit	How the Combination Delivers It	Clinic Impact
High availability	Load balancer removes failed instances; scaling replaces them [6]	Patients access the portal continuously — no downtime during failures
Security	Application gateway WAF + TLS filters threats centrally [7]	Patient records protected from attacks and interception

Low response time	Requests distributed evenly; no instance overloaded [6]	Appointment booking responds in milliseconds at peak load
Fault tolerance	Traffic rerouted instantly if a container or region fails [6][7]	Regional outage does not prevent patient access
Patient trust	Secure, fast, always-available portal builds confidence in digital health [1]	Higher portal adoption and better health outcomes

Table 7. Combined Benefits of Load Balancing, Scalability, and Application Gateway.

6. Cloud Expansion Drivers and Serverless Functions

6.1–6.4 Driving Forces Behind Cloud Expansion

- Cost/pay-as-you-go: no initial investment in hardware; only pay as you use [1], [2].
- Flexibility: deliver services within minutes; reconfigure with no change in hardware [1].
- Elastic scaling: the resources automatically scale with demand [2].
- International coverage: global clouds cover the entire world, allowing access to low-latency access globally [2].
- What is Reliability: 99.9%+ SLAs are supported by redundant infrastructure [1].

6.5 Serverless Functions: AWS Lambda and Azure Functions

- Serverless: functions that are executed in response to an event; the provider maintains all infrastructure [9].
- The developer writes the logic of the functions only; there is no provisioning of the server needed [8].

Examples of a smart clinic include:

1. The event of S3/Azure Blob storage triggers a Lambda or an Azure Function [9], [8].
2. Function checks on the type of file (PDF, JPEG only)
3. Function metadata: time of upload, file size, patient ID.
4. Function inserts metadata record to the database.
5. Function sends notification to the patient that the upload is successful [9].

AWS Lambda runs function code based on events like file uploads, HTTP requests, or execution environments, with all underlying infrastructure being managed by AWS Lambda [9]. Azure Functions provides the same serverless executability coupled with the Azure ecosystem [8]. In a smart clinic scenario, a patient file upload triggers a serverless function that authenticates the file, extracts metadata, updates the database, and sends a notification all within a single event-based execution that costs nothing other than the time taken to execute the event-driven function [9].

6.6 Comparison with Containerization

Dimension	Serverless (Lambda / Azure Functions)	Containerization (Docker / Apache)
Execution model	Event-driven; function runs only when triggered [9]	Long-running; container serves requests continuously [4]
Infrastructure management	Fully managed by cloud provider [8][9]	Developer manages Docker file, image updates, container lifecycle [4]
Cost model	Pay per execution (millisecond billing) — zero idle cost [9]	Pay per running container — idle containers still incur cost [4]
Scaling	Automatic, near-instantaneous [9]	Manual or orchestrated via Kubernetes HPA [11]
Cold starts	First invocation after idle: 100ms–2s delay [9]	No cold start — running container responds immediately [4]
Vendor lock-in	High — code uses provider-specific SDKs and triggers [8][9]	Low — Docker images run on any cloud, on-prem, or locally [4]
Best for clinic	Short-lived event tasks: document validation, notifications [9]	Long-running web services: the patient portal front-end [4]
Portability	Limited — tied to AWS/Azure APIs [8][9]	High — OCI-compatible image runs anywhere [4]

Table 9. Serverless Functions vs Containerization

Containers and serverless functions do not represent competing paradigms but are complementary tools that are appropriate for various workload characteristics [4], [9]. The front-end of the patient portal of the smart clinic is a long-running, always-available web service, which is well suited to the concept of containerization: it needs consistent response times, persistent state using volumes and network isolation using Docker networks [4]. However, document processing a short-lived, event-driven task, is better served by AWS Lambda or Azure Functions: no idle infrastructure cost, automatic scaling, and event-based triggering [8], [9]. The most significant trade-off with serverless is vendor lock-in and cold start latency; containers are more portable and have more control at the expense of increased operational responsibility [10], [11].

7. Conclusion

This report has explored the concept of cloud engineering both theoretically and practically [1], [2]. The Apache Cloud Operations Control Panel showed that Docker makes it possible to build operationally tested, and validated deployments: all the steps between image building and network connectivity were proven by the numbers, logs, and browser testing [4], [5]. The case of smart clinic patient portal scenario demonstrated a combination of load balancing, scale, and application gateways to provide availability, security, and scalability [6], [7]. The workload peculiarities should inform the choice of cloud technology: containers with long-running services that have to be portably and controllably deployed [4], serverless functions with event-driven tasks where cost-efficiency and scalability under control should be given priority [9]. Due to the constant evolution of cloud platforms, engineering decisions need to balance the performance, security, cost, and operational complexity [1].

8. References

- [1] P. Mell and T. Grance, "The NIST definition of cloud computing," *NIST*, vol. 800, no. 145, Sep. 2011, doi: <https://doi.org/10.6028/nist.sp.800-145>.
- [2] 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, Jun. 2009, doi: <https://doi.org/10.1016/j.future.2008.12.001>.
- [3] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," 2014. Available: <https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf>
- [4] Docker Inc., "Docker Documentation," *Docker Documentation*, Jul. 15, 2020. <https://docs.docker.com>
- [5] Apache Software Foundation, "Documentation: Apache HTTP Server - The Apache HTTP Server Project," *Apache.org*, 2019. <https://httpd.apache.org/docs/>
- [6] Microsoft, "What is Azure Load Balancer? - Azure Load Balancer," *learn.microsoft.com*, Feb. 20, 2025. <https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview>
- [7] Microsoft, "What is Azure Application Gateway," *learn.microsoft.com*, Sep. 27, 2023. <https://learn.microsoft.com/en-us/azure/application-gateway/overview>
- [8] Microsoft, "Azure Functions Overview," *learn.microsoft.com*. <https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview>
- [9] Amazon Web Services, "What Is AWS Lambda? - AWS Lambda," *Amazon.com*, 2025. <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
- [10] OWASP, "Docker Security - OWASP Cheat Sheet Series," *cheatsheetseries.owasp.org*. https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
- [11] Kubernetes, "Kubernetes Documentation," *Kubernetes.io*, 2019. <https://kubernetes.io/docs/home/>
- [12] I.Periodicals, "IEEE REFERENCE GUIDE," 2018. Available: <https://ieeauthorcenter.ieee.org/wp-content/uploads/IEEE-Reference-Guide.pdf>

Appendix A — Video Demonstration Link

Field	Your Content
Video Title	ACCA7027 Cloud Engineering – Docker and Apache Practical Demonstration
Video Link	https://youtu.be/4DRvou4a4Dc
Duration	8 minutes 12 sec
Platform	Youtube
Content Covered	Folder structure → Docker file → docker build → docker run → browser test → docker logs → Docker Hub push → volume persistence → custom network and container-to-container communication