

**ACCA7027 – CLOUD ENGINEERING PRINCIPLES AND
PRACTICE**

*An Investigation into Docker
Containerisation, Azure Cloud Services, and
Serverless Computing in Modern Cloud
Environments*

2505249 - Sanjay kumar Unkili

UNIVERSITY OF WALES TRINITY SAINT DAVID - Wales Institute of Science and Arts - School of
Applied Computing

Table of Contents

1. Introduction	4
2. Virtualisation	4
2.1 What is Virtualisation, Hypervisors, and Virtual Machines	4
2.2 How Shared Hardware is Virtualised	5
2.3 Type 1 vs Type 2 Hypervisors	5
3. Containerisations	5
3.1 What are Containers and How They Differ from VMs.....	5
3.2 Benefits of Containerisation	6
3.3 Dockerfile Example with Explanation	6
4. Docker Images and Publishing	7
4.1 Understanding Docker Images, Layers, and Registries	7
4.2 Building the Docker Image	7
4.3 Pushing to DockerHub	9
5. Container Volumes.....	10
5.1 The Problem of Data Persistence in Containers	10
5.2 Volume Demonstration	10
6. Container Networking	12
6.1 Docker Networking Modes	12
6.2 Custom Network Demonstration.....	12
7. End-to-End Demonstration.....	13
8. Cloud Computing Concepts	18
8.1 Application Scenario and Cloud Provider.....	18
8.2 Load Balancing	18
8.3 Scalability.....	18
8.4 Application Gateway.....	19
8.5 Why Load Balancing and Scaling are Complementary	19
8.6 Benefits of Combining All Three Technologies	20
9. Driving Forces of Cloud Adoption.....	20
10. Serverless Functions.....	21
11. Serverless vs Containerisation: A Comparative Analysis.....	21
11.1 Cost Efficiency	22
11.2 Scalability.....	22
11.3 Flexibility and Control	22
11.4 Vendor Lock-in	23
11.5 Cold Starts and Performance	23
12. Conclusion.....	24

References	25
------------------	----

Table of Figures

Figure 1: Dockerfile for the custom Nginx web server image.....	6
Figure 2: Custom HTML page (index.html) served by the container	7
Figure 3: Building the Docker image from the Dockerfile	8
Figure 4: Verifying the built image.....	8
Figure 5: Detailed image inspection showing metadata and layers	9
Figure 6: Logging into DockerHub and pushing the image.....	9
Figure 7: DockerHub repository showing the published image.....	10
Figure 8: Deleting the local image and pulling it back from DockerHub	10
Figure 9: Creating and inspecting the Docker volume.....	11
Figure 10: Diagnosing the log symlink issue and resolving it for volume logging	11
Figure 11: Volume persistence - logs survive container deletion	12
Figure 12: Custom Docker network listing and detailed inspection	13
Figure 13: System update on Kali Linux	14
Figure 14: Installing required dependencies	14
Figure 15: Adding Docker's GPG key and repository.....	14
Figure 16: Installing Docker Engine packages	15
Figure 17: Docker installation verification - version check and hello-world container.....	15
Figure 18: Docker login and project directory setup.....	16
Figure 19: Running the container with network and volume, then testing with curl.....	16
Figure 20: Custom web page served by the Dockerised Nginx container in the browser	17
Figure 21: Running a container from the pulled DockerHub image and verifying with curl	17
Figure 22: Cleanup - removing container, network, and volume.....	17

1. Introduction

Cloud computing has revolutionized the IT infrastructure by providing computing resources through on demand internet services which provide flexibility, scalability and cost effectiveness as alternative to owning physical hardware. One of the most important enabling technologies in this change is containerisation.

This report discusses some of the important technologies used in the modern cloud environment. The former is a discussion of virtualisation and containerisation, which addresses Docker using real world examples such as image creation with a Dockerfile, publishing to DockerHub, volume operations, networking, and deploying a containerised Nginx web server.

The second section discusses the load balancing, scaling, application gateways in Microsoft Azure and how these complementary systems are deployed as a unified architecture with an e-commerce application.

The last part touches on the drivers of cloud adoption, cost effectiveness, flexibility and scaling and then compares serverless computing and containerisation, assessing the strengths of each model and the appropriate situations to use them.

2. Virtualisation

2.1 What is Virtualisation, Hypervisors, and Virtual Machines

Virtualisation refers to the act of developing software-based devices of the physical computing resources, which allows a number of separated environments to operate concurrently on the same hardware platform. This is the technology that makes cloud computing possible because it enables cloud providers to effectively share physical infrastructure across tenants. At the heart of virtualisation is the hypervisor, which is a software layer that lies in between the physical hardware and the virtual machines and is responsible in assigning resources and isolation amongst concurrent workloads [1].

A virtual machine (VM) is an entire software simulation of a physical computer, with its own operating system, virtualised hardware (CPU, memory, storage, network interfaces), and applications. There is no knowledge of other VMs on the same host and thus every VM is isolated and thus secure enough to be used in a multi-tenant cloud deployment [2].

2.2 How Shared Hardware is Virtualised

Physical servers in a cloud data centre have hypervisors that virtualize the underlying CPU, memory, storage, and network resources into pools which can be dynamically allocated to virtual machines. A request by a cloud customer to spin up a compute instance causes the hypervisor to slice the requested resources off the physical host and make them available to the VM as dedicated hardware. This model can be used to have several VMs in one physical server with a potential utilisation of 10-15% which can be extended to 60-80% and therefore save a lot of costs on infrastructure and consumptions [3].

The techniques used to manage resource sharing include CPU time-slicing, memory ballooning and I/O virtualisation. The hypervisor maintains a fair scheduling to ensure that no individual VM can hog the host resources, and still creates an illusion to each VM that it has exclusive access to the hardware assigned to it.

2.3 Type 1 vs Type 2 Hypervisors

Hypervisors are of two major categories. Type 1 (bare-metal) hypervisors like VMware ESXi, Microsoft Hyper-V and Xen operate directly on the physical hardware without a host operating system. They have better performance, reduced latency and better security isolation and are the default option in production cloud environments and enterprise data centres [3].

Hosted hypervisors (type 2) like Virtualbox and VMware Workstation are applications over a more traditional operating system. Although they are easier to install and are appropriate in development and testing, they add overhead of the host OS layer. Type 1 hypervisors are practically the only type of hypervisor used by cloud providers to address the high-performance and security demands of a large-scale multi-tenant infrastructure [4].

3. Containerisations

3.1 What are Containers and How They Differ from VMs

Containerisation is more advanced than conventional virtualisation in the sense that it is based on the operating system level as opposed to the hardware level. Each container has a copy of the host OS kernel and isolated user space of process, filesystem, and network stacks, in contrast to VMs where each guest operating system requires its own complete operating

system [5]. This basic architectural distinction causes containers to be vastly lighter weight, with common sizes in megabytes as opposed to the gigabytes of VM pictures.

Docker is the most popular containerisation platform that bundles up an application along with all its dependencies, libraries and configuration files into a standardised package known as a container image. This image may then be run as a running container on any system, with Docker installed, no matter the underlying operating system or hardware configuration.

3.2 Benefits of Containerisation

The main advantages of containerisation are that startup time is in seconds, compared to minutes with VMs, resource usage is low as containers do not run their own operating system but instead share the common kernel of the host, and consistency in the environment, so that applications do not tend to resist or behave differently in development, testing, and production environments. Containers are also used to facilitate microservices architectures in which complex applications are broken down into smaller deployable services (each in its own container) [6].

3.3 Dockerfile Example with Explanation

In order to illustrate how to use containerisation in practice, a personalised HTML page, an image of an Nginx web server was built using a custom Dockerfile. The figure 1 illustrates the Dockerfile developed in this assignment.

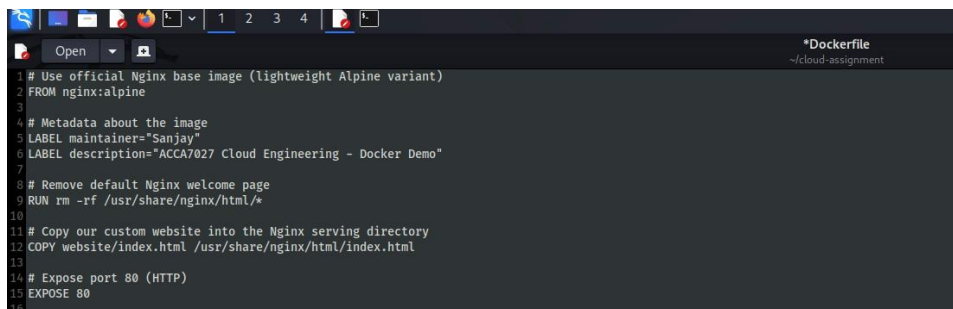
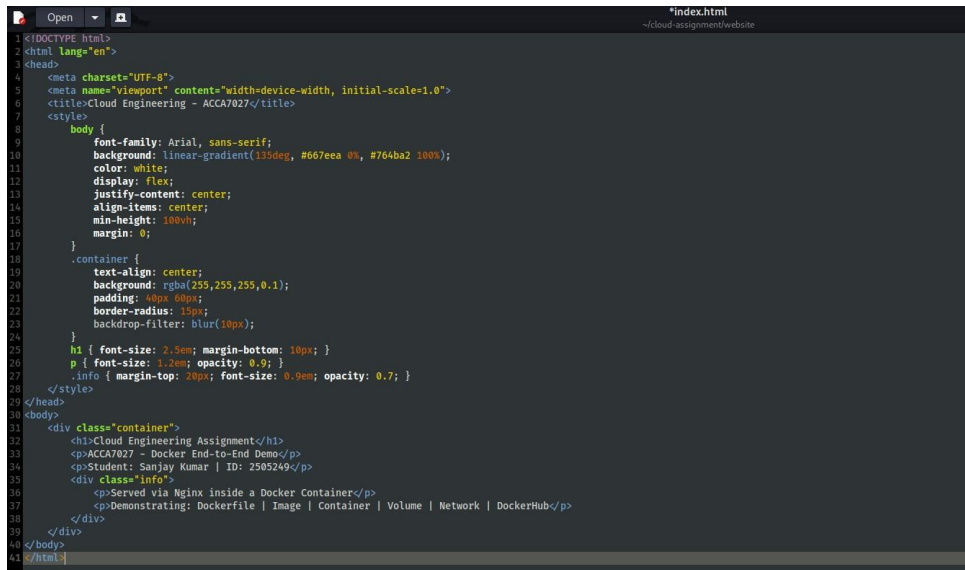


Figure 1: Dockerfile for the custom Nginx web server image

The FROM instruction in line 2 uses a basic image in nginx:alpine, which is a minimal Linux distribution designed to be used in containers, around 5MB. The metadata of identification and maintainability is added to the LABEL instructions on lines 5-6. Lines 9 and 12 respectively delete the standard Nginx welcome page and then copy the custom web pages to the serving directory of the container. Lastly, EXPOSE 80 on line 15 records that the

container is listening on the port 80 of the HTTP. Figure 2 illustrates the tailor made HTML page which was to be served by this container.



```
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 Engineering - ACCA7027</title>
7   <style>
8     body {
9       font-family: Arial, sans-serif;
10      background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
11      color: white;
12      display: flex;
13      justify-content: center;
14      align-items: center;
15      min-height: 100vh;
16      margin: 0;
17    }
18    .container {
19      text-align: center;
20      background: rgba(255,255,255,0.1);
21      padding: 40px 60px;
22      border-radius: 15px;
23      backdrop-filter: blur(10px);
24    }
25    h1 { font-size: 2.5em; margin-bottom: 10px; }
26    p { font-size: 1.2em; opacity: 0.9; }
27    .info { margin-top: 20px; font-size: 0.9em; opacity: 0.7; }
28  </style>
29 </head>
30 <body>
31   <div class="container">
32     <h1>Cloud Engineering Assignment</h1>
33     <p>ACCA7027 - Docker End-to-End Demo</p>
34     <p>Student: Sanjay Kumar | ID: 2505249</p>
35     <div class="info">
36       <p>Served via Nginx inside a Docker Container</p>
37       <p>Demonstrating: Dockerfile | Image | Container | Volume | Network | DockerHub</p>
38     </div>
39   </div>
40 </body>
41 </html>
```

Figure 2: Custom HTML page (index.html) served by the container

4. Docker Images and Publishing

4.1 Understanding Docker Images, Layers, and Registries

A Docker image is a read only template which is a layered filesystem that includes all that is required to execute an application, both the code, runtime, libraries and system tools. Every command in the Dockerfile forms a new layer and Docker in turn caches the layers to optimize future builds. Once an image is altered, the altered layers just have to be reassembled, which is much more efficient in terms of development than reassembling the entire image. Docker registries, like DockerHub are centralised stores where images may be stored, versioned, and distributed to a team or to others [7].

4.2 Building the Docker Image

The docker build command was used to construct the image with the tag of sanjay0608/cloud-assignment:v1. The entire build process can be seen in figure 3.

```

(kali@kali)~[/cloud-assignment]
$ docker build -t sanjay0608/cloud-assignment:v1 .
[+] Building 29.1s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 458B
=> [internal] load metadata for docker.io/library/nginx:alpine
=> [auth] library/nginx:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/3] FROM docker.io/library/nginx:alpine@sha256:5616878291a2eed594aee8db4d4de5878cf7edcb475e59193904b198d9b830de
=> => resolve docker.io/library/nginx:alpine@sha256:5616878291a2eed594aee8db4d4de5878cf7edcb475e59193904b198d9b830de
=> sha256:612c0c1df4c55a0bf145f84df03cb28de505e6a52fb8e49c9da7b143fc00ad2d 20.25MB / 20.25MB
=> sha256:4a8b0b2a5b1937755263ac7ac4ee26db2906c13a5d70f1fb1875ba8e370cdd8f 1.40kB / 1.40kB
=> sha256:453da7dbc73e7338c4e201ec3f3c4a5f7751adb5a6f127792c1cd001a780721 1.21kB / 1.21kB
=> => sha256:aee4e54b3865ee4d98545b6b49f9e8ab3b9b6ed114a2eafedc7a22e984ea5a68 956B / 956B
=> sha256:781ff50d2644f74714c0d5a4ffa3447bbfd4b293cfb1dc6a63772ae24c89241 404B / 404B
=> sha256:583599bb7d382e9e986a9ff65204981307bfa71670a29f4190eedae93c4858be 627B / 627B
=> sha256:82736a35d0e7f1309edc13d09115410b81542cf8b91afff80deace02b6f2f26 1.87MB / 1.87MB
=> sha256:6e0ac1617861a677b045b7ff88545213ec31c0ff08763195a70a4a5adda577bb 3.86MB / 3.86MB
=> => extracting sha256:6a0ac1617861a677b045b7ff88545213ec31c0ff08763195a70a4a5adda577bb
=> extracting sha256:82736a35d0e7f1309edc13d09115410b81542cf8b91afff80deace02b6f2f26
=> extracting sha256:583599bb7d382e9e986a9ff65204981307bfa71670a29f4190eedae93c4858be
=> => extracting sha256:aee4e54b3865ee4d98545b6b49f9e8ab3b9b6ed114a2eafedc7a22e984ea5a68
=> => extracting sha256:781ff50d2644f74714c0d5a4ffa3447bbfd4b293cfb1dc6a63772ae24c89241
=> extracting sha256:453da7dbc73e7338c4e201ec3f3c4a5f7751adb5a6f127792c1cd001a780721
=> extracting sha256:4a8b0b2a5b1937755263ac7ac4ee26db2906c13a5d70f1fb1875ba8e370cdd8f
=> => extracting sha256:612c0c1df4c55a0bf145f84df03cb28de505e6a52fb8e49c9da7b143fc00ad2d
=> [internal] load build context
=> => transferring context: 1.41kB
=> [2/3] RUN rm -rf /usr/share/nginx/html/*
=> [3/3] COPY website/index.html /usr/share/nginx/html/index.html
=> exporting to image
=> => exporting layers
=> exporting manifest sha256:c69ff1bd6accfcc3b0449266d0e62abb8a8a60af355c02104b964331a1f31d05
=> => exporting config sha256:56072d295dda9ee4f6b933044995458ea50f3a365be123e45091740148f08cfc
=> => exporting attestation manifest sha256:b87f2565a0f178e3c01d0dd0e9b1377161cb5b96f4bb14b80fcaaf3c2e2cef8
=> => exporting manifest list sha256:362f5c03ad4cb6ea759f6a8af8beaf7e43621225d5fe4a5ce3cd1c03af0f9e
=> => naming to docker.io/sanjay0608/cloud-assignment:v1
=> => unpacking to docker.io/sanjay0608/cloud-assignment:v1
(kali@kali)~[/cloud-assignment]
$

```

Figure 3: Building the Docker image from the Dockerfile

The build output in action, shows the layered architecture of Docker. It starts by loading the Dockerfile, and pulling the nginx:alpine base image at Docker Hub, followed by running each command in its own build step. The last image was labeled as sanjay0608/cloud-assignment:v1, and validated at 92.6MB through docker image images as indicated in Figure 4.

```

(kali@kali)~[/cloud-assignment]
$ docker images | grep cloud-assignment
WARNING: This output is designed for human readability. For machine-readable output, please use --format.
sanjay0608/cloud-assignment:v1 362f5c03ad4c 92.6MB 26MB
(kali@kali)~[/cloud-assignment]
$

```

Figure 4: Verifying the built image

Metadata was analyzed with docker image inspect, as in Figure 5, and revealed the image architecture (amd64), operating system (linux), the version of Nginx 1.29.8, custom labels, entryptoint configuration, and the layered filesystem structure with each layer having its own SHA256 hash.

```
(kali@kali) ~/cloud-assignment
$ docker image inspect sanjay0608/cloud-assignment:v1
[
  {
    "Id": "sha256:362f5c03ad4cb6ea759fba68afd8beaf7e43621225d5fe4a5ce3cd1c03af0f9e",
    "RepoTags": [
      "sanjay0608/cloud-assignment:v1"
    ],
    "RepoDigests": [
      "sanjay0608/cloud-assignment@sha256:362f5c03ad4cb6ea759fba68afd8beaf7e43621225d5fe4a5ce3cd1c03af0f9e"
    ],
    "Comment": "buildkit.dockerfile.v0",
    "Created": "2026-04-23T02:21:41.310295683-04:00",
    "Config": {
      "ExposedPorts": {
        "80/tcp": {}
      },
      "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "NGINX_VERSION=1.29.8",
        "PKG_RELEASE=1",
        "DYNPKG_RELEASE=1",
        "NJS_VERSION=0.9.6",
        "NJS_RELEASE=1",
        "ACME_VERSION=0.3.1"
      ],
      "Entrypoint": [
        "/docker-entrypoint.sh"
      ],
      "Cmd": [
        "nginx",
        "-g",
        "daemon off;"
      ],
      "WorkingDir": "/",
      "Labels": {
        "description": "ACCA7027 Cloud Engineering - Docker Demo",
        "maintainer": "Sanjay"
      },
      "StopSignal": "SIGQUIT"
    },
    "Architecture": "amd64",
    "Os": "linux",
    "Size": 26005571,
    "RootFS": {
      "Type": "layers",
      "Layers": [
        "sha256:29df493baa13de438d6d2ece3a8333032e0b7b9b9d8cce4ee82194da255f61e1",
        "sha256:e6082cdc2516c48531390c6f369268148ed5189748b4cdcb30d8de04d4ef146",
        "sha256:f4eaddaa4c30384a3db34177c1132ca20f44c1216fe41ba7f607dcee58b908104"
      ]
    }
  }
]
```

Figure 5: Detailed image inspection showing metadata and layers

4.3 Pushing to DockerHub

Once the authentication was made with the docker login, an image was pushed to DockerHub with docker push as illustrated in Figure 6. The individual layers were uploaded, and a distinct SHA256 digest was allocated to check the integrity.

```
(kali@kali) ~/cloud-assignment
$ docker login -u sanjay0608

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

Password:
Login Succeeded

(kali@kali) ~/cloud-assignment
$ docker push sanjay0608/cloud-assignment:v1
The push refers to repository [docker.io/sanjay0608/cloud-assignment]
92fc9333459a: Pushed
aee4e54b3865: Pushed
3d5ae3e01e12: Pushed
35817508f230: Pushed
453da7dbc73e: Pushed
4a8b0b2a5b19: Pushed
82736a35d0e7: Pushed
583599bb7d38: Pushed
6a0ac1617861: Pushed
781ff50d2644: Pushed
612c0c1df4c5: Pushed
v1: digest: sha256:362f5c03ad4cb6ea759fba68afd8beaf7e43621225d5fe4a5ce3cd1c03af0f9e size: 856
```

Figure 6: Logging into DockerHub and pushing the image

The successful publication was checked using the DockerHub web interface, which confirmed the repository sanjay0608/cloud-assignment that has a v1 tag and a compressed repository size of 24.8MB, as illustrated in Figure 7.

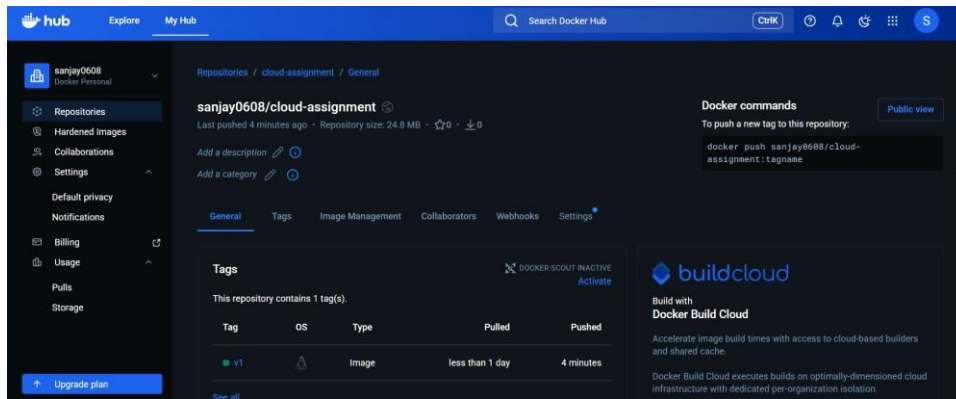


Figure 7: DockerHub repository showing the published image

To prove image portability, the local image was deleted with `docker rmi` and re-downloaded with DockerHub with `docker pull` as illustrated in Figure 8. The pulled image was then used to start a new container to test that it was fully functional and could be deployed independently of the registry.

```
(kali@kali)~/cloud-assignment
$ docker rmi sanjay0608/cloud-assignment:v1
Untagged: sanjay0608/cloud-assignment:v1
Deleted: sha256:362f5c03ad4cb6ea759fba68afd8beaf7e43621225d5fe4a5ce3cd1c03af0f9e

(kali@kali)~/cloud-assignment
$ docker pull sanjay0608/cloud-assignment:v1
v1: Pulling from sanjay0608/cloud-assignment
92fc9333459a: Pull complete
35817508f230: Pull complete
3d5ae3e01e12: Download complete
Digest: sha256:362f5c03ad4cb6ea759fba68afd8beaf7e43621225d5fe4a5ce3cd1c03af0f9e
Status: Downloaded newer image for sanjay0608/cloud-assignment:v1
docker.io/sanjay0608/cloud-assignment:v1
```

Figure 8: Deleting the local image and pulling it back from DockerHub

5. Container Volumes

5.1 The Problem of Data Persistence in Containers

Any data which is written into a Docker container is by default stored in an ephemeral writable layer [8]. Once the container is decrypted, the data in the container is lost forever. This poses a big problem to applications that require to save data, e.g. databases, log files and content posted by the user. Docker volumes address this issue by offering a storage system which exists outside the container lifecycle, controlled by Docker on the host filesystem [9].

5.2 Volume Demonstration

Docker volume create was used to create a named volume named `nginx-logs`. Its creation was verified by the `docker volume ls` command and the `docker volume inspect` command indicated its mount point at `/var/lib/docker/volumes/nginx-logs/_data` on the host machine as shown in Figure 9.

```

(kali@kali)-[~/cloud-assignment]
$ docker volume create nginx-logs
nginx-logs

(kali@kali)-[~/cloud-assignment]
$ docker volume ls
DRIVER      VOLUME NAME
local       nginx-logs

(kali@kali)-[~/cloud-assignment]
$ docker volume inspect nginx-logs
[
  {
    "CreatedAt": "2026-04-23T02:25:10-04:00",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/nginx-logs/_data",
    "Name": "nginx-logs",
    "Options": null,
    "Scope": "local"
  }
]

(kali@kali)-[~/cloud-assignment]
$

```

Figure 9: Creating and inspecting the Docker volume

During the first run of the container, it turned out that the nginx:alpine image symlinks its log files to /dev/stdout and /dev/stderr instead of writing them to real files, and thus the volume was unable to capture logs. The solution to this was to override the startup command of the container to delete the symlinks, generate actual log files, and subsequently to start Nginx. The diagnosis and resolution of this issue is illustrated in figure 10, after which the volume was able to capture access logs.

```

(kali@kali)-[~/cloud-assignment]
$ docker exec cloud-web-server ls -la /var/log/nginx/
total 8
drwxr-xr-x 2 root root 4096 Apr 23 06:12 .
drwxr-xr-x 1 root root 4096 Apr 15 20:12 ..
lrwxrwxrwx 1 root root    11 Apr 15 20:12 access.log -> /dev/stdout
lrwxrwxrwx 1 root root    11 Apr 15 20:12 error.log -> /dev/stderr

(kali@kali)-[~/cloud-assignment]
$ docker exec cloud-web-server sh -c "cat /var/log/nginx/access.log || echo 'No log file found'"
No log file found

(kali@kali)-[~/cloud-assignment]
$ docker stop cloud-web-server
cloud-web-server

(kali@kali)-[~/cloud-assignment]
$ docker ps cloud-web-server
cloud-web-server

(kali@kali)-[~/cloud-assignment]
$ docker volume rm nginx-logs
nginx-logs

(kali@kali)-[~/cloud-assignment]
$ docker volume create nginx-logs
nginx-logs

(kali@kali)-[~/cloud-assignment]
$ docker run --name cloud-web-server -d -p 8080:80 --network cloud-demo-network -- nginx-logs:/var/log/nginx sanjay8086/cloud-assignment:v1 sh -c "rm -f /var/log/nginx/access.log /var/log/nginx/error.log 00 touch /var/log/nginx/access.log /var/log/nginx/error.log 00 sleep 60"
fa22bcff0b0e447a837a7d3f1d3d3151f2b5232d9e9a9a0b0a1f4387f

(kali@kali)-[~/cloud-assignment]
$ docker exec cloud-web-server cat /var/log/nginx/access.log
172.18.0.1 - - [23/Apr/2026:06:16:31 +0000] "GET / HTTP/1.1" 200 3336 "-" "curl/8.19.8" "-"
172.18.0.1 - - [23/Apr/2026:06:16:32 +0000] "GET / HTTP/1.1" 200 3336 "-" "curl/8.19.8" "-"
172.18.0.1 - - [23/Apr/2026:06:16:32 +0000] "GET / HTTP/1.1" 200 3336 "-" "curl/8.19.8" "-"

```

Figure 10: Diagnosing the log symlink issue and resolving it for volume logging

To show that volumes retain data even after the container lifecycle, the container was halted, and totally removed with docker stop and docker rm. Temporary Alpine container was subsequently started which mounted the same nginx-logs volume, and was able to retrieve the access logs of the deleted container, as illustrated in Figure 11. The three GET requests that have been issued previously were still displayed, which confirms that the volume retains information without the container.

```
(kali@kali)~/cloud-assignment
$ docker stop cloud-web-server
cloud-web-server

(kali@kali)~/cloud-assignment
$ docker rm cloud-web-server
cloud-web-server

(kali@kali)~/cloud-assignment
$ docker run --rm -v nginx-logs:/logs alpine cat /logs/access.log

Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
a63813f19ccb: Download complete
4e0cb4ac63e7: Download complete
Digest: sha256:5b10f432ef3da1b8d4c7eb6c487f2f5a8f096bc91145e68878dd4a5019afde11
Status: Downloaded newer image for alpine:latest
172.18.0.1 - - [23/Apr/2026:06:38:31 +0000] "GET / HTTP/1.1" 200 1336 "-" "curl/8.19.0" "-"
172.18.0.1 - - [23/Apr/2026:06:38:32 +0000] "GET / HTTP/1.1" 200 1336 "-" "curl/8.19.0" "-"
172.18.0.1 - - [23/Apr/2026:06:38:32 +0000] "GET / HTTP/1.1" 200 1336 "-" "curl/8.19.0" "-"

(kali@kali)~/cloud-assignment
$
```

Figure 11: Volume persistence - logs survive container deletion

This is a necessary behaviour in production cloud setups, where data in applications, database files and logs need to be preserved despite container restart, updates, and redeployments. In the absence of volumes, regular tasks like rolling updates or container crashes would lead to irreversible loss of data.

6. Container Networking

6.1 Docker Networking Modes

Docker comes with various networking drivers that allow connecting containers and other systems [10]. The default of single-host standalone containers is the bridge driver. The host driver eliminates network isolation between the container and the host, providing the container with access to host networking. The none driver completely disables networking and gives total isolation. In a multi-host setup, the overlay driver allows containers on multiple Docker hosts to communicate in a secure manner, which is critical when an orchestrated environment is being implemented using Docker Swarm or Kubernetes.

6.2 Custom Network Demonstration

A docker network create was used to create a custom bridge network called cloud-demo-network. Its presence was confirmed using the docker network ls command along with the default networks, and the detailed configuration was shown using the docker network inspect command, including the bridge driver, subnet 172.18.0.0/16 and the gateway 172.18.0.1 as shown in Figure 12.

```
(kali@kali)~/cloud-assignment
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
94273b08262e        bridge             bridge              local
0341d255422f        cloud-demo-network bridge              local
2820f7e23b94        host               host                local
7dd25d849ba1        none              null                local

(kali@kali)~/cloud-assignment
$ docker network inspect cloud-demo-network
[
  {
    "Name": "cloud-demo-network",
    "Id": "0341d255422ffe8c9e8bf57878501350a967575b1742fdaec146bf0e77596790",
    "Created": "2026-04-23T02:24:18.276167358-04:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv4": true,
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Options": {},
    "Labels": {},
    "Containers": {},
    "Status": {
      "IPAM": {
        "Subnets": [
          {
            "172.18.0.0/16": {
              "IPsInUse": 3,
              "DynamicIPsAvailable": 65533
            }
          }
        ]
      }
    }
  }
]
```

Figure 12: Custom Docker network listing and detailed inspection

The custom networks offer some great benefits as compared to the default bridge network. Multi-container architectures are made easier by containers on a custom network resolving each other by name using the built-in DNS server of Docker. Custom networks also offer an improved isolation, because containers on separate networks do not interact except when linked explicitly, improving the security of microservice deployments.

7. End-to-End Demonstration

This part will bring together all the ideas mentioned above into one single workflow and illustrate the entire pipeline of Docker setup to the deployment of a containerised web application. It was run on Kali Linux and in logical order.

It started by updating the system packages with apt update and apt upgrade as demonstrated in Figure 13, then it installed the necessary requirements such as certificates, curl, and GPG tools as demonstrated in Figure 14.

```

(kali@kali)~[~/Desktop]
$ sudo apt update && sudo apt upgrade -y
[sudo] password for kali:
Get:1 http://kali.download/kali kali-rolling InRelease [34.0 kB]
Get:2 http://kali.download/kali kali-rolling/main amd64 Packages [21.1 MB]
Get:3 http://kali.download/kali kali-rolling/main amd64 Contents (deb) [53.4 MB]
Get:4 http://kali.download/kali kali-rolling/contrib amd64 Packages [118 kB]
Get:5 http://kali.download/kali kali-rolling/contrib amd64 Contents (deb) [274 kB]
Get:6 http://kali.download/kali kali-rolling/non-free amd64 Packages [188 kB]
Get:7 http://kali.download/kali kali-rolling/non-free amd64 Contents (deb) [907 kB]
Get:8 http://kali.download/kali kali-rolling/non-free-firmware amd64 Packages [15.3 kB]
Get:9 http://kali.download/kali kali-rolling/non-free-firmware amd64 Contents (deb) [38.7 kB]
Fetched 76.1 MB in 14s (5,534 kB/s)
2247 packages can be upgraded. Run 'apt list --upgradable' to see them.
The following packages were automatically installed and are no longer required:
amass-common libcrypt-dev libpgpme11t64 libpocketsphinx3 libsoup-2.4-1
curlftpfs libdisplay-info2 libpgmepp6t64 libpoppler145 libsoup2.4-common
firmware-ti-connectivity libdlt2 libhdf4-0-alt libportmidi0 libspinxbase3t64
gir1.2-girepository-2.0 libflac12t64 libicu-dev libpostproc58 libsqlcipher1
icu-devtools libfuse2t64 libinstpatch-1.0-2 libpython3.12-minimal libswscale8
libarmadillo14 libfuse3-3 libjs-jquery-ui libpython3.12-stdlib libteamctl0
libaudio2 libgav1-1 libjs-underscore libpython3.12t64 libtheora0
libavfilter10 libgdal36 liblbfsgsb0 libqt5ct-common1.8 libtsk19t64
libavformat61 libgdata-common libmongoc-1.0-0t64 libradare2-5.0.0t64 libudfread0
libblosc2-4 libgdata22 libmupdf25.1 librubberband2 libutempter0
libbluray2 libgdk-pixbuf2.0-bin libnet1 libframe1 libvdpau-v4l2
libbson-1.0-0t64 libgeos3.13.0 libogdi4.1 libsigsegv2 mesa-vaapi-drivers
libconfig-inifiles-perl libgirepository-1.0-1 libplacebo349 libsnmp40t64 openjdk-21-jre
Use 'sudo apt autoremove' to remove them.

```

Figure 13: System update on Kali Linux

```

(kali@kali)~[~/Desktop]
$ sudo apt install -y ca-certificates curl gnupg lsb-release
Upgrading:
ca-certificates dirmngr gnupg-l10n gpg gpg-wks-client gpgsm libcurl3t64-gnutls
curl gnupg gnupg-utils gpg-agent gpgconf gpgv libcurl4t64

Installing dependencies:
libngtcp2-crypto-openssl0

Summary:
Upgrading: 24, Installing: 1, Removing: 0, Not Upgrading: 2223
Download size: 246 kB / 10.3 MB
Space needed: 1,480 kB / 62.8 GB available

Get:1 http://kali.download/kali kali-rolling/main amd64 gpgv amd64 2.4.9-4 [240 kB]
Get:2 http://kali.download/kali kali-rolling/main amd64 lsb-release all 12.1-2 [6,072 B]
Fetched 246 kB in 1s (238 kB/s)
Preconfiguring packages ...
(Reading database ... 408445 files and directories currently installed.)
Preparing to unpack .../libssl3t64_3.6.1-3_amd64.deb ...
Unpacking libssl3t64:amd64 (3.6.1-3) over (3.4.0-2) ...
Setting up libssl3t64:amd64 (3.6.1-3) ...
(Reading database ... 408444 files and directories currently installed.)
Preparing to unpack .../openssl-provider-legacy_3.6.1-3_amd64.deb ...
Unpacking openssl-provider-legacy (3.6.1-3) over (3.4.0-2) ...
Setting up openssl-provider-legacy (3.6.1-3) ...
(Reading database ... 408444 files and directories currently installed.)
Preparing to unpack .../00-openssl_3.6.1-3_amd64.deb ...
Unpacking openssl (3.6.1-3) over (3.4.0-2) ...
Preparing to unpack .../01-ca-certificates_20260223_all.deb ...
Unpacking ca-certificates (20260223) over (20241223) ...
Preparing to unpack .../02-libnghttp3-9_1.15.0-1_amd64.deb ...
Unpacking libnghttp3-9:amd64 (1.15.0-1) over (1.6.0-2) ...
Preparing to unpack .../03-libngtcp2-16_1.21.0-1_amd64.deb ...
Unpacking libngtcp2-16:amd64 (1.21.0-1) over (1.9.1-1) ...
Selecting previously unselected package libngtcp2-crypto-openssl0:amd64.
Preparing to unpack .../04-libngtcp2-crypto-openssl0_1.21.0-1_amd64.deb ...

```

Figure 14: Installing required dependencies

The official GPG key and repository of Docker was then added to the debian Bookworm stable channel, as Figure 15 shows, and the Docker Engine packages (docker-ce, docker-ce-cli, containerd.io and plugins) were installed, as Figure 16 shows.

```

(kali@kali)~[~/Desktop]
$ sudo install -o root -g root -m 0755 -c /etc/apt/keysring
(kali@kali)~[~/Desktop]
$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o /etc/apt/keysring/docker.gpg
(kali@kali)~[~/Desktop]
$ sudo chmod a+r /etc/apt/keysring/docker.gpg
(kali@kali)~[~/Desktop]
$ echo "deb [arch=amd64] signed-by=/etc/apt/keysring/docker.gpg https://download.docker.com/linux/debian bookworm stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
(kali@kali)~[~/Desktop]

```

Figure 15: Adding Docker's GPG key and repository

```
(kali@kali)-[~/Desktop]
└─$ sudo apt update
Get:1 https://download.docker.com/linux/debian bookworm InRelease [46.6 kB]
Get:2 https://download.docker.com/linux/debian bookworm/stable amd64 Packages [67.8 kB]
Hit:3 http://http.kali.org/kali kali-rolling InRelease
Get:4 https://download.docker.com/linux/debian bookworm/stable amd64 Contents (deb) [1,719 B]
Fetched 116 kB in 2s (64.1 kB/s)
2223 packages can be upgraded. Run 'apt list --upgradable' to see them.

(kali@kali)-[~/Desktop]
└─$ sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
Installing:
  containerd.io  docker-buildx-plugin  docker-ce  docker-ce-cli  docker-compose-plugin

Installing dependencies:
  docker-ce-rootless-extras  libslirp0  pigz  slirp4netns

Suggested packages:
  cgroupfs-mount | cgroup-lite  docker-model-plugin

Summary:
  Upgrading: 0, Installing: 9, Removing: 0, Not Upgrading: 2223
  Download size: 94.9 MB
  Space needed: 365 MB / 62.8 GB available

Get:1 https://download.docker.com/linux/debian bookworm/stable amd64 containerd.io amd64 2.2.3-1-debian.12-bookworm [23.6 MB]
Get:2 http://kali.download.kali kali-rolling/main amd64 pigz amd64 2.8-1 [62.7 kB]
Get:3 http://kali.download.kali kali-rolling/main amd64 libslirp0 amd64 4.9.1-1 [72.8 kB]
Get:4 http://kali.download.kali kali-rolling/main amd64 slirp4netns amd64 1.3.3-1 [39.9 kB]
Get:5 https://download.docker.com/linux/debian bookworm/stable amd64 docker-ce-cli amd64 5:29.4.1-1-debian.12-bookworm [17.0 MB]
Get:6 https://download.docker.com/linux/debian bookworm/stable amd64 docker-ce amd64 5:29.4.1-1-debian.12-bookworm [22.7 MB]
Get:7 https://download.docker.com/linux/debian bookworm/stable amd64 docker-buildx-plugin amd64 0.33.0-1-debian.12-bookworm [16.9 MB]
Get:8 https://download.docker.com/linux/debian bookworm/stable amd64 docker-ce-rootless-extras amd64 5:29.4.1-1-debian.12-bookworm [6,558 kB]
Get:9 https://download.docker.com/linux/debian bookworm/stable amd64 docker-compose-plugin amd64 5.1.3-1-debian.12-bookworm [8,023 kB]
Fetched 94.9 MB in 15s (6,468 kB/s)
```

Figure 16: Installing Docker Engine packages

The Docker service was then started and the user was added to the docker group to have non-root access and the installation was then verified by checking the version of Docker 29.4.1 and successfully running the hello-world test container as shown in Figure 17.

```
(kali@kali)-[~/Desktop]
└─$ sudo systemctl start docker

(kali@kali)-[~/Desktop]
└─$ sudo systemctl enable docker
Synchronizing state of docker.service with SysV service script with /usr/lib/systemd/systemd-sysv-install.
Executing: /usr/lib/systemd/systemd-sysv-install enable docker

(kali@kali)-[~/Desktop]
└─$ sudo usermod -aG docker $USER

(kali@kali)-[~/Desktop]
└─$ newgrp docker

(kali@kali)-[~/Desktop]
└─$ docker --version
Docker version 29.4.1, build 055a478

(kali@kali)-[~/Desktop]
└─$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
4f55086f7dd0: Pull complete
d5e71e642bf5: Download complete
Digest: sha256:f9078146db2e05e794366b1bfe584a14ea6317f44027d10ef7dad65279026885
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/

(kali@kali)-[~/Desktop]
└─$
```

Figure 17: Docker installation verification - version check and hello-world container

The project workspace and DockerHub authentication were set up, which is presented in Figure 18, with Docker running.

```

(kali@kali)-[~/Desktop]
$ docker login -u sanjay0608

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

Password:

WARNING! Your credentials are stored unencrypted in '/home/kali/.docker/config.json'.
Configure a credential helper to remove this warning. See
https://docs.docker.com/go/credential-store/

Login Succeeded

(kali@kali)-[~/Desktop]
$ mkdir ~/cloud-assignment && cd ~/cloud-assignment

(kali@kali)-[~/cloud-assignment]
$ mkdir website

(kali@kali)-[~/cloud-assignment]
$

```

Figure 18: Docker login and project directory setup

Dockerfile and HTML page were written (see above in Figures 1 and 2), the image was built (Figure 3), and the custom network and volume was created (Figures 9 and 12). The custom network, the volume, and the port mapping (8080:80) were attached to the container and launched at the same time. The output of `docker ps` confirmed the successful running of the container, and the output of `curl` was the HTML content, as seen in Figure 19.

```

(kali@kali)-[~/cloud-assignment]
$ docker run --name cloud-web-server -d -p 8080:80 --network cloud-demo-network --v nginx-logs:/var/log/nginx sanjay0608/cloud-assignment:v1
(b1d92b278a3ab7e42057d941807e0d93f7b1ec582e0ea7b1ce0bcc35af)

(kali@kali)-[~/cloud-assignment]
$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS                               NAMES
(b1d92b278a3 sanjay0608/cloud-assignment:v1   "/docker-entrypoint..."  16 seconds ago Up 14 seconds  0.0.0.0:8080->80/tcp, [::]:8080->80/tcp   cloud-web-server

(kali@kali)-[~/cloud-assignment]
$ curl http://localhost:8080
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Cloud Engineering - ACCA7827</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background: linear-gradient(135deg, #6677aa 0%, #77a2 100%);
      color: white;
      display: flex;
      justify-content: center;
      align-items: center;
      min-height: 100vh;
      margin: 0;
    }
    .container {
      text-align: center;
      background: rgba(255,255,255,0.1);
      padding: 40px 60px;
      border-radius: 15px;
      backdrop-filter: blur(10px);
    }
    h1 { font-size: 2.5em; margin-bottom: 10px; }
    p { font-size: 1.2em; opacity: 0.9; }
    .info { margin-top: 20px; font-size: 0.9em; opacity: 0.7; }
  </style>
</head>
<body>
  <div class="container">
    <h1>Cloud Engineering Assignment</h1>
    <p>ACCA7827 - Docker End-to-End Demos</p>
    <p>Student: Sanjay Kumar | ID: 2585249</p>
    <div class="info">
      <p>Served via Nginx inside a Docker Container</p>
      <p>Demonstrating: Dockerfile | Image | Container | Volume | Network | DockerHub</p>
    </div>
  </div>

```

Figure 19: Running the container with network and volume, then testing with curl

The custom HTML page containing the student details and assignment information was accessed at the web server via a browser using the localhost:8080 port successfully and it showed the information, as seen in Figure 20. This verifies the entire pipeline between Dockerfile and a running web service.

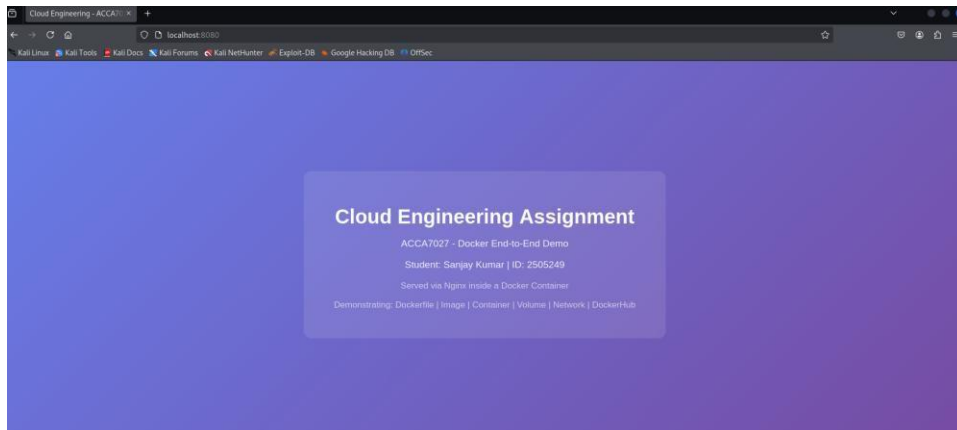


Figure 20: Custom web page served by the Dockerised Nginx container in the browser

Once the image had been deposited to DockerHub (Figure 6) and the image checked on the web interface (Figure 7), the image was once again pulled in and a new container launched on the image hosted on the registry as demonstrated in Figure 21.

```
(kali@kali): ~/cloud-assignment
$ docker run --name cloud-web-pulled -d -p 8080:80 sanjay8080/cloud-assignment:v1 sh -c 'rm -f /var/log/nginx/access.log /var/log/nginx/error.log 00 touch /var/log/nginx/access.log /var/log/nginx/error.log 00 nginx -g "daemon off;"'
ec62e5e15b0b05238e5b2b0b474722c49d5d897221261fe0388535a77f7b62

(kali@kali): ~/cloud-assignment
$ curl http://localhost:8080
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Cloud Engineering - ACCA7027</title>
<style>
body {
font-family: Arial, sans-serif;
background: linear-gradient(135deg, #067ee5 #0, #70baf2 100%);
color: white;
display: flex;
justify-content: center;
align-items: center;
margin: 0;
}
.container {
text-align: center;
background: rgba(255,255,255,0.1);
padding: 40px 80px;
border-radius: 15px;
background-filter: blur(10px);
}
h1 {
font-size: 2.5em; margin-bottom: 10px;
}
p {
font-size: 1.2em; opacity: 0.9;
}
p {
margin-top: 10px; font-size: 0.9em; opacity: 0.7;
}
</style>
</head>
<body>
<div class="container">
<div>Cloud Engineering Assignment</div>
<div>ACCA7027 - Docker End-to-End Demo</div>
<div>Student: Sanjay Kumar | ID: 2505249</div>
<div>Served via Nginx inside a Docker Container</div>
<div>Demonstrating: Dockerfile | Image | Container | Volume | Network | DockerHub</div>
</div>
</body>
</html>
```

Figure 21: Running a container from the pulled DockerHub image and verifying with curl

Lastly, the volume, the custom network, and the container were stopped and deleted to clean up all the resources as illustrated in Figure 22. This shows good resource management and lifecycle practices vital in the cloud.

```
(kali@kali): ~/cloud-assignment
$ docker stop cloud-web-pulled
cloud-web-pulled

(kali@kali): ~/cloud-assignment
$ docker rm cloud-web-pulled
cloud-web-pulled

(kali@kali): ~/cloud-assignment
$ docker network rm cloud-demo-network
cloud-demo-network

(kali@kali): ~/cloud-assignment
$ docker volume rm nginx-logs
nginx-logs

(kali@kali): ~/cloud-assignment
$
```

Figure 22: Cleanup - removing container, network, and volume

8. Cloud Computing Concepts

8.1 Application Scenario and Cloud Provider

This section explores the implementation of an e-commerce web application on Microsoft Azure to demonstrate the key cloud computing principles of load balancing, scalability and application gateways. The case study is of an online retail platform with noticeable changes in traffic where the usual traffic of about 5,000 simultaneous users is typical during non-promotional times and over 50,000 simultaneous users during promotions like Black Friday sales. This kind of application requires a cloud architecture with the ability to dynamically scale to the unpredictable demand and still provide the customers with a consistent performance and high availability. Azure as the cloud provider was chosen due to its provision of a set of integrated services that meet each of these requirements and its documentation and tooling are more inclined towards enterprise deployments [11].

8.2 Load Balancing

Azure Load Balancer is an Layer 4 (transport layer) load balancer that distributes internet traffic to a number of backend servers to avoid overloading [12]. It operates at TCP/UDP protocol level and makes routing choices based on source/destination IP addresses, port numbers, and protocol type, through a hash distribution algorithm.

The load balancer is constantly checking the backend instances using configurable health probes which periodically test the responsiveness of the servers. In case of failures during health checks, traffic is automatically diverted to the healthy servers, making it fault-tolerant and reducing downtime.

It enables both public and internal modes: public load balancers redirect internet traffic to the backend VMs; internal ones redirect service-to-service traffic between private virtual networks [13]. In the case of an e-commerce application, a public load balancer at the entry point spreads customer requests to web server instances such that the unavailability of a server does not impact the availability of the application.

8.3 Scalability

Azure Virtual machine scale sets (VMSS) also has the advantage of horizontal scaling which dynamically allocates VM instances to meet real-time demands [14]. Autoscale policies are configured by administrators based on such metrics as CPU utilisation or memory usage: to

reduce costs, they add instances when CPU utilisation is more than 70 percent and removes instances when it is less than 30 percent.

This is the core of elasticity of cloud computing, where applications can respond to unexpected traffic surges, like flash sales, without provisioning costly idle servers in advance. VMSS generates new instances based on predefined templates of images which makes all instances the same in terms of configuration.

Although Azure can also be scaled vertically by resizing single VMs, a horizontal scale via VMSS is typically favored in web applications because it offers performance as well as redundancy, and is therefore more appropriate to keeping availability when workload varies [15].

8.4 Application Gateway

Azure Application Gateway uses Layer 7 (application layer), which also allows intelligent routing in HTTP-based routing in addition to simple traffic distribution [16]. It is used in place of the Layer 4 Load Balancer and, unlike it, inspects the contents of HTTP requests, routing by URL path, hostnames, and headers. In the case of e-commerce, /api/, /images/ and /checkout/ requests can be sent to dedicated specialised backend pools.

The main characteristics are built-in Web Application Firewall (WAF) that guards against popular attacks such as SQL injection, cross-site scripting and other vulnerabilities described in the OWASP Top 10. SSL/TLS termination removes encryption processing on backend servers, allowing them to be used in application logic. Session affinity based on cookies allows a user to have consistent connection to the same back end server (such as shopping cart continuity across sessions).

8.5 Why Load Balancing and Scaling are Complementary

Scalability and load balancing are complementary terms that solve the same problem; the changing demand [17]. Scalability will add or delete server instances depending on the demand whereas load balancing will spread the traffic evenly across the instances available. In the absence of load balancing, other servers would be idle without a way to divert the traffic. On its own, the load balancing without scaling can only spread traffic over a set number of servers that will get congested.

The e-commerce case is different because when VMSS reaches 25 instances, during Black Friday traffic, the load balancer will automatically identify new instances through health probes and allocate requests respectively. With traffic reduced and cases eliminated, the load balancer scales up or down, so that no requests are sent to decommissioned servers, which is a dynamically elastic infrastructure in response to real-time demands.

8.6 Benefits of Combining All Three Technologies

Integrating the load balancing, scalability and application gateways gives a layered architecture which provides better performance, security and cost efficiency [18]. The Application Gateway is the smart gateway which checks the requests, WAF policies, terminating the SSL, and directing the traffic to the right backend pools. The Load Balancer spreads traffic among the VM instances evenly whereas VMSS adjusts the number of instances given the real-time demand.

This approach provides a number of benefits: high availability due to multi-level redundancy and automatic failover; optimisation of performance due to intelligent routing, even distribution of loads, and scale according to demand, and enhanced security due to WAF, termination of SSL, and network isolation; and cost effectiveness due to the transformation of fixed infrastructure costs to the variable operational costs according to actual usage. When combined, such technologies allow cloud providers to achieve service level agreements of 99.95 or above.

9. Driving Forces of Cloud Adoption

The accelerated pace of cloud computing can be attributed to the intertwining of economic, operational and technological factors that render the adoption of cloud computing an appealing requirement by organisations of any size.

Cost reduction remains the primary driver. Traditional on-premise systems mean huge capital investment in hardware, data centre, power, and cooling. Cloud computing transfers this into operational expenditure model where organisations only pay to use it, as opposed to the costly idle infrastructure that was previously deployed to meet peak demand [23].

Flexibility and agility enable organisations to launch environments and applications in minutes, not weeks, enabling quicker time-to-market, low-risk experimentation, and quicker iteration on customer feedback.

Scalability allows applications to dynamically scale resources as demand changes in real-time - critical when businesses need to scale with variable workloads such as e-commerce during sales events or streaming platforms during live shows [21]. Horizontal and vertical scaling provide steady performance irrespective of load.

The less operational load is due to cloud providers being in charge of hardware maintenance, security patches and physical infrastructure, which allows IT teams to concentrate on application development and business value rather than managing commodity infrastructure.

Global accessibility via provider data centres networks around the world enables organisations to implement applications in geographic locations, which lowers the end-user latency and offers disaster recovery through geographic redundancy without using physical data centres in each area [24].

10. Serverless Functions

Functions as a Service (FaaS) or serverless computing is a cloud implementation model in which the provider handles all server administration and provisioning [19]. Although naming is not an issue, servers remain, only they are completely abstracted in terms of the developers, who only write discrete functions that are activated on certain events.

On-demand services such as AWS Lambda and Azure Functions allow developers to deploy code modules that are automatically run in response to an event such as an HTTP request, file upload, queue message, or database update. The cloud provider takes care of all infrastructure issues such as provisioning of resources, scaling of instances of the demands and decommissioning of the idle resources [25].

The serverless price is granular as it only charges on an execution time in milliseconds and invocation counts instead of the reserved capacity. Azure functions, example, provides a million free executions per month. This renders serverless especially economical in applications that have sporadic or unpredictable traffic which would otherwise be charged by conventional server-based infrastructure when it does not have any work.

11. Serverless vs Containerisation: A Comparative Analysis

Both containerisation and serverless functions focus on the forces behind cloud adoption mentioned above but in vastly different architectural concepts, both with their own pros and

cons. Comparing these two models with the major driving forces could inform significant considerations when selecting the right deployment strategy.

11.1 Cost Efficiency

Serverless functions are even more cost efficient than workloads with irregular or low-volume traffic patterns since the charge is only incurred at the time of the actual function execution, and zero when the function is not running [22]. Even with autoscaling, containerised applications will often have a set minimum of instances in operation to reduce the delay associated with cold starts, which is a baseline cost irrespective of demand. But in cases of high-throughput workloads that remain high over time, containers prove more economical since serverless invocation charges grow exponentially with scale, and the price per milliseconds pricing can be higher than the cost of a continuously-running container with the same traffic.

11.2 Scalability

Serverless services offer automatic and virtually instant scale, and all scaling decisions are handled by the cloud provider, openly and transparently. Functions are able to scale to zero to thousands of running instances without any setup. Container-based scaling, e.g., using orchestrators like Kubernetes or Docker Swarm, similarly allows scaling to be automated, but scaling policies, resource limits, and health checks must be explicitly configured. Container scaling is usually measured on a slower scale of seconds to minutes than the sub-second scaling possible with serverless, but containers can be more predictable when scaling occurs because there are no cold starts once initial provisioning has been completed [26].

11.3 Flexibility and Control

Containers give much more freedom and control over the execution environment. The developers have the option of any programming language, framework, operating system, and version of the runtime, installing custom dependencies at the system level, and configuring the environment to exactly match their needs. Serverless functions are limited to the runtimes of the cloud provider, have a time constraint (usually 5-15 minutes), have limited memory and temporary storage, and limited control over the underlying execution environment. Containers are more suitable to use in complex applications that need long running processes or persistent connections and/or special system libraries [27].

11.4 Vendor Lock-in

Serverless functions are closely bound to the proprietary event model, API gateway, and services of the cloud provider, and bind heavily on vendor lock-in [20]. Migration of an application that runs on AWS Lambda to Azure functions involves significant code refactoring because of differences in the signature of functions, event formats and the supporting service integrations. Containers, in turn, are portable in nature since the Docker image format and runtime is standardised across all major cloud providers and on-premises environments. Containerised application can be deployed on AWS, Azure, or Google Cloud, or a private data centre with little or no changes, which offers strategic flexibility and lessens reliance on a particular provider [28].

11.5 Cold Starts and Performance

The cold start problem is a major serverless weakness: the first invocation after a period of idle time incurs an additional latency due to spin-up execution environments, runtimes loading, and code initialisation by the provider. This may contribute hundreds of milliseconds to seconds of delay, which is unacceptable to latency sensitive applications. Containers have predictable low-latency response time, as applications are always present in memory. Solutions such as provisioned concurrency reduce cold starts at the expense of serverless solutions, which keep warm instances [29].

Finally, it will be a decision based on the workload aspects. Serverless is event-driven, low-frequency, stateless workloads that focus on cost minimisation and operational simplicity. Containerisation is more suited to the complex, long-running, stateful applications that need environmental control, portability and predictable performance. Increasingly, new cloud architectures use hybrid methods, with serverless used to run peripheral event-driven tasks and containers used to run core services [30].

12. Conclusion

Cloud computing is based on scalable, reliable, and cost-effective services provided by layered technologies. Virtualisation abstracts the physical hardware, with Docker containerisation providing a more portable deployment method. Hands-on examples proved that Docker has a simplified workflow, starting with Dockerfile creation and continuing with image creation, and DockerHub publication, volumes and networking allows preserving data and interacting between containers.

The load balancing, scalability, and application gateway services of Azure are not independent but complementary features. The distribution of traffic load balancing, the scaling of capacity, and intelligent routing and security are offered by gateways- all of which guarantee high availability, performance, and security.

The major drivers of cloud adoption are still cost savings, flexibility, scalability and less burden on operations. There was no universal superiority found between serverless functions and containerisation, and each will be appropriate to various workloads. Serverless is better at event-driven, sporadic work that focuses on minimising costs, whereas containers are better adapted to complex, long-running applications that need to be run in an environment and be portable. Hybrid strategies are becoming more popular among modern architectures, where serverless is used to perform peripheral tasks and containers are used to perform core services in order to maximise the benefits of both.

References

- [1] A. K. Dass, A. Parida, S. Panigrahi, and S. K. Moharana, “Virtualization in cloud computing: Transforming infrastructure and enhancing efficiency,” *Zenodo*, Dec. 2023. doi: 10.5281/zenodo.10300506.
- [2] D. Silva, J. Rafael, and A. Fonte, “Virtualization maturity in creating system VM: An updated performance evaluation,” *International Journal of Electrical and Computer Engineering Research*, vol. 3, no. 2, pp. 7–17, Jun. 2023. doi: 10.53375/ijecer.2023.341.
- [3] M. Almoudane, “A comprehensive study on virtualization techniques and their role in cloud computing,” *ResearchGate*, May 2025. doi: 10.13140/RG.2.2.14067.64800.
- [4] C. Hayhurst, “Type 1 hypervisor vs. Type 2: What’s the difference?,” *BizTech Magazine*, 2024. [Online]. Available: <https://biztechmagazine.com/article/2024/08/type-1-vs-type-2-hypervisors-whats-difference-perfcon> (Accessed: Apr. 24, 2026).
- [5] N. Zhou, H. Zhou, and D. Hoppe, “Containerization for high performance computing systems: Survey and prospects,” *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2722–2740, 2022. [Online]. Available: <https://arxiv.org/pdf/2212.08717>
- [6] V. R. Gudelli, “Containerization technologies: ECR and Docker for microservices architecture,” *International Journal of Innovative Research in Management, Pharmacy and Sciences (IJIRMPs)*, vol. 11, no. 3, 2023. [Online]. Available: https://www.researchgate.net/profile/Venkata-Gudelli/publication/389590950_Containerization_Technologies_ECR_and_Docker_for_Microservices_Architecture/links/67c8888fe62c604a0dd5257c/Containerization-Technologies-ECR-and-Docker-for-Microservices-Architecture.pdf
- [7] I. Kansanen, “Automating Docker registry version control using Jenkins,” 2024. [Online]. Available: <https://trepo.tuni.fi/bitstream/handle/10024/162263/KansanenIlmari.pdf?sequence=2>
- [8] D. Moreau, K. Wiebels, and C. Boettiger, “Containers for computational reproducibility,” *Nature Reviews Methods Primers*, vol. 3, no. 1, p. 50, 2023. [Online]. Available: <https://par.nsf.gov/servlets/purl/10501743>

- [9] P. Muzumdar, A. Bhosale, G. P. Basyal, and G. Kurian, “Navigating the Docker ecosystem: A comprehensive taxonomy and survey,” *arXiv preprint arXiv:2403.17940*, 2024. [Online]. Available: <https://arxiv.org/pdf/2403.17940>
- [10] D. Silva, J. Rafael, and A. Fonte, “Toward optimal virtualization: An updated comparative analysis of Docker and LXD container technologies,” *Computers*, vol. 13, no. 4, p. 94, 2024. [Online]. Available: <https://www.mdpi.com/2073-431X/13/4/94>
- [11] V. Bieger, “A decision support framework for multi-cloud service composition,” Master’s thesis, 2023. [Online]. Available: <https://studenttheses.uu.nl/bitstream/handle/20.500.12932/44605/A%20decision%20support%20framework%20for%20multi-cloud%20service%20composition%20-%20Valentijn%20Bieger%201966839%20-%20anonymized.pdf?sequence=1&isAllowed=y>
- [12] A. B. Kasture, “A study and implementation of load balancing services provided by Microsoft Azure cloud service provider,” *SSRN*, 2023. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4549044
- [13] J. Kankaanpää, “On-premises data center load balancer service delivery model based on infrastructure as...,” 2024. [Online]. Available: <https://www.utupub.fi/server/api/core/bitstreams/b0f18693-675a-4afc-9eec-c2facce9962f/content>
- [14] T. L. Betti Pillippuge, Z. Khan, and K. Munir, “Horizontal autoscaling of virtual machines in hybrid cloud infrastructures: Current status, challenges, and opportunities,” *Encyclopedia*, vol. 5, no. 1, p. 37, 2025. [Online]. Available: <https://www.mdpi.com/2673-8392/5/1/37>
- [15] G. S. Kopparthi, “Cost optimization in Azure and AWS cloud,” *Journal of Marketing & Social Research*, vol. 2, pp. 177–182, 2025. [Online]. Available: <https://jmsr-online.com/article/cost-optimization-in-azure-and-aws-cloud-372/>
- [16] M. S. Krishna and K. Padmanaban, “Application gateway implementation with Azure,” 2024. [Online]. Available: <https://shisrrj.com/paper/SHISRRJ247273.pdf>
- [17] M. V. der Boor, S. C. Borst, J. S. Van Leeuwen, and D. Mukherjee, “Scalable load balancing in networked systems: A survey of recent advances,” *SIAM Review*, vol. 64, no. 3, pp. 554–622, 2022. [Online]. Available: <https://arxiv.org/pdf/1806.05444>

- [18] J. Olatunde-Thorpe, S. E. Aifuwa, T. Oshoba, E. Ogbuefi, and D. Akokodaripon, "Integrating load balancing strategies: Conceptual frameworks ensuring optimized performance across enterprise and service provider networks," *Journal of Frontiers in Multidisciplinary Research*, vol. 3, no. 2, pp. 170–181, 2022. [Online]. Available: https://www.researchgate.net/profile/Jennifer-Olatunde-Thorpe/publication/396543579_Integrating_load_Balancing_Strategies_Conceptual_Frameworks_Ensuring_Optimized_Performance_Across_Enterprise_and_Service_Provider_Networks/links/69063a82a2b691617b68c205/Integrating-load-Balancing-Strategies-Conceptual-Frameworks-Ensuring-Optimized-Performance-Across-Enterprise-and-Service-Provider-Networks.pdf
- [19] H. M. Zangana, Z. B. Sallow, and M. Omar, "Cloud architectures for distributed serverless computing: A review of event-driven and function-as-a-service paradigms," *International Journal of Artificial Intelligence & Robotics (IJAIR)*, vol. 6, no. 2, pp. 57–64, 2024. [Online]. Available: <https://ejournal.unitomo.ac.id/index.php/ijair/article/download/8597/4309>
- [20] S. Anwar, "Mitigating vendor lock-in in serverless cloud computing," Master's thesis, 2025. [Online]. Available: https://lutpub.lut.fi/bitstream/handle/10024/170879/Mastersthesis_Sheraz_Anwar.pdf?sequence=1&isAllowed=y
- [21] M. Sharma, "Streaming queries: Enabling real-time elastic scaling in modern applications," *Journal of Computer Science and Technology Studies*, vol. 7, no. 3, pp. 319–326, 2025. [Online]. Available: <https://al-kindipublishers.org/index.php/jcsts/article/view/9348>
- [22] H. Ahmad, "Performance evaluation of serverless cloud-native applications: FaaS vs. container-based in AWS," 2025. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/907601/Ahmad_Hamaad.pdf?sequence=2
- [23] Ayaphila Mkhize, K. D. Mokhothu, Mukhoden Tshikhotho, and B. A. Thango, "Evaluating the Impact of Cloud Computing on SME Performance: A Systematic Review," *Businesses*, vol. 5, no. 2, pp. 23–23, May 2025, doi: <https://doi.org/10.3390/businesses5020023>.
- [24] R. n Al-Dwairi and W. Jditawi, "The Role of Cloud Computing on the Governmental Units Performance and EParticipation (Empirical Study)," *International Journal of Advances*

in *Soft Computing and its Applications*, vol. 14, no. 3, pp. 79–93, Nov. 2022, doi: <https://doi.org/10.15849/ijasca.221128.06>.

[25] P. Borra and Hanuman Prasad Pamidipoola, “Serverless Computing: The Future of Scalability and Efficiency with AWS, Azure, and GCP,” *International Journal of Advanced Research in Science Communication and Technology*, vol. 5, no. 2, pp. 505–514, Feb. 2025, doi: <https://doi.org/10.48175/IJARSCT-23373>.

[26] M. GOLEC, G. K. WALIA, M. KUMAR, F. CUADRADO, S. S. Gill, and S. UHLIG, “Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions,” *ACM Computing Surveys*, Oct. 2024, doi: <https://doi.org/10.1145/3700875>.

[27] L. Vara, “Serverless Kubernetes: The Evolution of Container Orchestration,” *European Journal of Computer Science and Information Technology*, vol. 13, no. 30, May 2025, Accessed: Apr. 24, 2026. [Online]. Available: <https://ejournals.org/ejesit/vol13-issue30-2025/serverless-kubernetes-the-evolution-of-container-orchestration/>

[28] D. Mo, R. Cordingly, D. Chinn, and W. Lloyd, “Addressing Serverless Computing Vendor Lock-In through Cloud Service Abstraction,” Dec. 2023, doi: <https://doi.org/10.1109/cloudcom59040.2023.00040>.

[29] A. Joosen *et al.*, “Serverless Cold Starts and Where to Find Them,” pp. 938–953, Mar. 2025, doi: <https://doi.org/10.1145/3689031.3696073>.

[30] S. Saxena, “Serverless Architectures: Redefining Scalability and Cost Optimization in Cloud Computing,” *Journal of Information Systems Engineering and Management*, vol. 10, no. 58s, pp. 625–632, Aug. 2025, doi: <https://doi.org/10.52783/jisem.v10i58s.12642>.

Practical demonstration link : [cloud engineering docker video.MP4](#)