



Assignment Title: Implementing Cloud Infrastructures- From Docker Containerisation to Serverless Functions

Student Name: Md Ahasanul Kobir Opy

Student ID number: 2522648

Programme: MSc Computing Portfolio

Module: ACCA7027 Cloud Engineering Principles and Practice

Lecturer: Renuga Jayakumar

Submission Date: 27 April 2026

TABLE OF CONTENTS

Executive Summary	3
1. Introduction	4
1.1 The Architectural Shift	4
1.1.1 The Five Essential Characteristics	4
1.2 The Shift from On-Premise to Cloud Infrastructure	5
1.3 Purpose and Scope of this Report	5
1.4 Video Demonstration Link	6
2. Cloud Technology and Examples (Docker).....	7
2.1 The Foundations of Virtualisation	7
2.2 Installation and Verification of the Docker Engine	7
2.3 Core Principles and Benefits of Containerisation.....	10
2.4 Developing the Application and the Dockerfile Blueprint.....	10
2.5 Image Construction and the Web Server Lifecycle	11
2.6 Publishing to the Cloud: Docker Hub Registry.....	13
2.7 Data Persistence: Managing Container Volumes	15
2.8 Container Communication: Virtual Networking	16
3. Cloud Computing Concepts	18
3.1 Practical Application: The E-Commerce Use Case	18
3.2 Scalability: Managing Dynamic Workloads.....	18
3.3 Load Balancing: Intelligent Traffic Distribution.....	18
3.4 Application Gateways: Smart Routing and Security.....	18
3.5 The Symbiotic Relationship: Scaling and Balancing.....	19
3.6 Customer Impact: Performance, Reliability, and Trust	19
4. The Future of Cloud Services.....	20
4.1 Driving Forces: The Economics of Cloud Expansion	20
4.2 Flexibility: Accelerating Innovation and Speed to Market	21
4.3 Serverless Functions: The Next Evolution in Abstraction	21
4.4 Comparative Analysis: Containers vs. Serverless	21
5. Conclusion	23
6. References.....	24

Executive Summary

This is a detailed report that gives a systematic overview of the concepts of contemporary cloud engineering and its application in contemporary software development. The elastic scale is relevant As the industry is developing to shift to the decentralized, microservices based, architecture, the focus on the principles of resource abstraction and distribution[1]. The report is split into two sections to fill the gap between the design and implementation. The former is a hands-on tour of containerization using Docker, showing the full application lifecycle - starting with the very process of creating a virtualized Linux environment, continuing with the creation of effective Dockerfiles, and the management of persistent volumes and intricate inter-container networking. The last step of pushing images to a global cloud registry (Docker Hub) finally completes this part and demonstrates the idea of build once and run everywhere of DevOps [2].

The second section of the report takes a different turn to the more abstract principles that enable cloud infrastructures to scale globally. It applies the engineering concepts of Layer 4 and Layer 7 load balancing, the complexities of horizontal and vertical scalability as a prime example and the application gateways to handle the incoming traffic. It also examines the economic and operational motivations behind the industry change to cloud services, which specifically, is the shift to the capital-intensive hardware investments into a more flexible utility-based pricing model [6]. The paper wraps up by critically evaluating the container-based and event-driven serverless functions to examine the trade-offs in the form of latency, statefulness and cost-efficiency in order to give a holistic view of the existing cloud-native environment [7].

1. Introduction

Cloud computing is not just a change in the location of data, but it is also a change in the design, delivery and economics of many services. It decouples hardware from software, converting capital expenditure (CapEx) to operational expenditure (OpEx). This enables organisations to use computing resources as needed, then to avoid significant upfront hardware investments and adjust business demands more rapidly [6].

This is particularly relevant in today's world as organisations rely on internet services that need to be accessible, responsive and secure despite varying demands. Historically, IT infrastructure often involved lengthy planning, hardware procurement and over-provisioning to meet peak demand [8]. Cloud computing overcomes these constraints via resource pooling, on-demand provisioning and service models that enable experimentation, continuous delivery and business agility [6].

1.1 The Architectural Shift

Underlying cloud computing is virtualization - the ability to create software representations of physical resources [8]. This layer of indirection enables a physical server to be split into multiple virtual machines (VMs) with each having its own operating system, dedicated memory, virtual disk and network configuration. This allows better utilization of hardware while providing isolation between applications [10].

The key element that allows this to happen is the hypervisor. A hypervisor is a piece of software that runs between the hardware and the guest operating systems, manages access to the CPU, memory and device input/output, and presents each VM as a separate computer. Type 1 hypervisors are bare-metal hypervisors that run directly on the hardware and are commonly found in enterprise and cloud data centres, while Type 2 hypervisors are hosted hypervisors that run on top of a host operating system and are more suitable for desktop and lab environments [10].

Functionally, a hypervisor is comprised of a number of parts: a CPU scheduler to control processing time, a memory manager to isolate and map the random access memory (RAM), a device and storage virtualisation part to present virtual disks and network adapters, and a virtual networking part to control communication between VMs and the outside world. These components enable isolation, sharing and consistent presentation of virtual hardware, which enables virtualisation in today's cloud infrastructure [10].

Cloud architecture is commonly delivered through three major service models:

- I. **Infrastructure as a Service (IaaS):** It offers the bare bones such as servers, storage and networking.
- II. **Platform as a Service (PaaS):** This is a system that provides a construct through which developers are able to create and run applications without having to manage the hardware.
- III. **Software as a Service (SaaS):** Provides the end-user with fully functional applications over a web browser [8].

1.1.1 The Five Essential Characteristics

National Institute of Standards and Technology (NIST) states that the characteristics that distinguish true cloud computing, compared with traditional data centers, are five in nature: [8]

- I. **On-demand Self-service:** Users can provision resources independently without the need for human intervention from the service provider.
- II. **Broad Network Access:** The capabilities are network-based and can be accessed via conventional mechanisms (mobile phones, tablets, laptops).
- III. **Resource Pooling:** The computing resources of the provider are pooled and are shared with multiple consumers using a multi-tenant model.
- IV. **Rapid Elasticity:** The resources may be elastically scaled in and out and re-provisioned at scale very quickly depending on the demand.
- V. **Measured Service:** Cloud systems automatically control and optimize resource use by taking advantage of a metering capability (e.g. storage, processing, bandwidth).

The following discussion considers how these characteristics are being enabled by complex backend engineering approaches, specifically focusing on distributed systems, load balancing and high availability protocols that ensure "five nines" (\$99.999%) of uptime in an increasingly digital-first world [5].

1.2 The Shift from On-Premise to Cloud Infrastructure

In order to understand why Docker and serverless functions are so famous these days, it is important to keep in mind the previous on-premise infrastructure. On-premise IT infrastructure was costly to run and also slow to scale [6]. Expansion of the organisation often involved the addition of more servers, space, power and cooling and hire of additional IT support personnel [6]. This approach led to high fixed costs and limited flexibility to adapt to changing technical and business needs [6].

With cloud computing, the paradigm has shifted to a more software-based infrastructure management process [6]. Where previously, the organisation could only use a static set of physical hardware, it can now create and manage the compute, storage and networking resources purely through software interfaces and tools [6]. This also facilitates Infrastructure as Code (IaC), whereby environments can be defined, duplicated and managed through configuration files, rather than hardware [6]. This results in more consistent applications development, testing and deployments to local environments, virtual machines and cloud computing environments [2].

This change also accounts for the rise of containerisation and serverless. Containers like Docker bundle applications and their dependencies into standardised environments [2]. Serverless computing enables functions to run on demand [7]. Both trends are part of the broader shift towards cloud computing and on-demand service delivery [6].

1.3 Purpose and Scope of this Report

To link the theoretical concepts of cloud computing to engineering practice is the purpose of this report. It outlines the main concepts identified in the module brief, focusing on virtualisation, containerisation, image building and publication, persistent storage, networking, scalability, load balancing, application gateways and serverless computing.

In Section 2, the emphasis is on Docker. The discussion covers the installation and verification of Docker Engine, the development of a simple web application, the creation of a Dockerfile, the building and running of a container image, and the publication of the image to Docker Hub and the use of volumes and virtual networks. This process illustrates how containerisation enables efficient

deployment to the cloud, and also the link between virtualisation and application delivery.

Section 3 takes a more abstract look at cloud systems by applying the concepts of scalability, load balancing and application gateways to an e-commerce example. This section demonstrates how a cloud provider can implement each idea, explains why scaling and load balancing go hand in hand, and illustrates how the three technologies complement each other to enhance reliability, performance and trust.

Section 4 discusses the key drivers of cloud computing growth (particularly cost and flexibility), and compares containerisation to serverless functions. It takes into account control, portability, automation and cost efficiency to give a fair assessment of the advantages and disadvantages of each approach. The report aims to stay focused on the assessment criteria, while showing understanding and practical applicability.

1.4 Video Demonstration Link

As part of the assessment requirements, to walk through the terminal exercises shown in this report ,a video demonstration was taken. Which consists of explanations of the Docker commands and the final output of the web page.

YouTube link : <https://www.youtube.com/watch?v=X1qZcifdF4>

2. Cloud Technology and Examples (Docker)

2.1 The Foundations of Virtualisation

Virtualisation is a fundamental aspect of cloud computing. It decouples software from hardware and enables several operating systems to run simultaneously in separate virtual machines. Typically, a computer would run a single operating system. Virtualization allows multiple virtual machines to run on a single physical server, each with allocated resources for CPU, memory, storage and network [10].

This is achieved through a software component called a hypervisor. In essence, the hypervisor accepts requests from the guest operating systems and manages the hardware. The key components of a hypervisor are a CPU scheduler, memory manager, storage and device emulation and a virtual switch or network. These components manage resource allocation, provide isolation and provide a virtual hardware abstraction to the guest operating systems [10].

Hypervisors are generally classified into Type 1 and Type 2. Type 1 hypervisors run on the bare hardware and are often associated with enterprise servers and cloud computing due to improved performance and security. Type 2 hypervisors are installed on top of an operating system and are more likely to be found in classroom, testing and desktop environments where flexibility is more important than efficiency [10].

Although VMs were a big step forward, virtual machines are relatively heavy. VMs require a copy of an operating system like Windows or Linux. So deploying ten VMs on a server uses up a lot of memory and disk space because the operating system is duplicated many times. This was one of the motivating factors for the advent of containerisation, which provides a less heavy form of isolation for running applications [10].

2.2 Installation and Verification of the Docker Engine

The first step after the VM had booted was to update the package list. The `sudo apt update` command was used, as shown in Figure 1. The `sudo` command in Linux allows a user to run commands with elevated or administrative privileges. The `apt update` command is not for installing software, but rather for updating the list of software packages available from the Ubuntu software repositories. This ensures the latest Docker packages and security updates are available for installation.

```
ubuntu@user: ~  
ubuntu@user:~$ sudo apt update  
[sudo: authenticate] Password:  
Hit:1 http://ports.ubuntu.com/ubuntu-ports questing InRelease  
Get:2 http://ports.ubuntu.com/ubuntu-ports questing-updates InRelease [136 kB]  
Get:3 http://ports.ubuntu.com/ubuntu-ports questing-backports InRelease [133 kB]  
Get:4 http://ports.ubuntu.com/ubuntu-ports questing-security InRelease [136 kB]  
Get:5 http://ports.ubuntu.com/ubuntu-ports questing-updates/main arm64 Packages [341 kB]  
Get:6 http://ports.ubuntu.com/ubuntu-ports questing-updates/main arm64 Components [47.1 kB]  
Get:7 http://ports.ubuntu.com/ubuntu-ports questing-updates/main Icons (48x48) [15.1 kB]  
Get:8 http://ports.ubuntu.com/ubuntu-ports questing-updates/main Icons (64x64) [18.3 kB]  
Get:9 http://ports.ubuntu.com/ubuntu-ports questing-updates/main arm64 c-n-f Metadata [7,776 B]  
Get:10 http://ports.ubuntu.com/ubuntu-ports questing-updates/restricted arm64 Components [212 B]  
Get:11 http://ports.ubuntu.com/ubuntu-ports questing-updates/universe arm64 Packages [194 kB]  
Get:12 http://ports.ubuntu.com/ubuntu-ports questing-updates/universe arm64 Components [42.1 kB]  
Get:13 http://ports.ubuntu.com/ubuntu-ports questing-updates/universe Icons (48x48) [23.6 kB]  
Get:14 http://ports.ubuntu.com/ubuntu-ports questing-updates/universe Icons (64x64) [35.2 kB]  
Get:15 http://ports.ubuntu.com/ubuntu-ports questing-updates/universe arm64 c-n-f Metadata [5,220 B]  
Get:16 http://ports.ubuntu.com/ubuntu-ports questing-updates/multiverse arm64 Components [212 B]  
Get:17 http://ports.ubuntu.com/ubuntu-ports questing-backports/main arm64 Components [212 B]  
Get:18 http://ports.ubuntu.com/ubuntu-ports questing-backports/restricted arm64 Components [216 B]  
Get:19 http://ports.ubuntu.com/ubuntu-ports questing-backports/universe arm64 Components [216 B]  
Get:20 http://ports.ubuntu.com/ubuntu-ports questing-backports/multiverse arm64 Components [216 B]  
Get:21 http://ports.ubuntu.com/ubuntu-ports questing-security/main arm64 Packages [236 kB]  
Get:22 http://ports.ubuntu.com/ubuntu-ports questing-security/main arm64 Components [448 B]  
Get:23 http://ports.ubuntu.com/ubuntu-ports questing-security/main arm64 c-n-f Metadata [5,216 B]  
Get:24 http://ports.ubuntu.com/ubuntu-ports questing-security/restricted arm64 Components [212 B]  
Get:25 http://ports.ubuntu.com/ubuntu-ports questing-security/universe arm64 Packages [127 kB]  
Get:26 http://ports.ubuntu.com/ubuntu-ports questing-security/universe arm64 Components [10.8 kB]  
Get:27 http://ports.ubuntu.com/ubuntu-ports questing-security/universe arm64 c-n-f Metadata [3,868 B]  
Get:28 http://ports.ubuntu.com/ubuntu-ports questing-security/multiverse arm64 Components [212 B]  
Fetched 1,518 kB in 5s (326 kB/s)  
111 packages can be upgraded. Run 'apt list --upgradable' to see them.  
ubuntu@user:~$
```

Figure 1: Updating the local package repositories on the terminal.

With the system updated, Docker Engine installation could begin. Figure 2 shows that the installation process required several modules to be downloaded. The unpacking of containerd and runc is displayed on the terminal. These are basic runtime components. Containerd manages the container lifecycle, while runc is the low-level utility that instructs the Linux kernel to create the isolated environments in which containers run [10].

```
ubuntu@user: ~  
Get:9 http://ports.ubuntu.com/ubuntu-ports questing/universe arm64 ubuntu-fan all 0.12.17 [34.3 kB]  
Fetched 64.2 MB in 35s (1,858 kB/s)  
Preconfiguring packages ...  
Selecting previously unselected package pigz.  
(Reading database ... 194805 files and directories currently installed.)  
Preparing to unpack .../0-pigz_2.8-1_arm64.deb ...  
Unpacking pigz (2.8-1) ...  
Selecting previously unselected package bridge-utils.  
Preparing to unpack .../1-bridge-utils_1.7.1-4ubuntu2_arm64.deb ...  
Unpacking bridge-utils (1.7.1-4ubuntu2) ...  
Selecting previously unselected package runc.  
Preparing to unpack .../2-runc_1.3.3-0ubuntu1-25.10.3_arm64.deb ...  
Unpacking runc (1.3.3-0ubuntu1-25.10.3) ...  
Selecting previously unselected package containerd.  
Preparing to unpack .../3-containerd_2.1.3-0ubuntu3.1_arm64.deb ...  
Unpacking containerd (2.1.3-0ubuntu3.1) ...  
Selecting previously unselected package docker.io.  
Preparing to unpack .../4-docker.io_28.2.2-0ubuntu1_arm64.deb ...  
Unpacking docker.io (28.2.2-0ubuntu1) ...  
Selecting previously unselected package liberror-perl.  
Preparing to unpack .../5-liberror-perl_0.17030-1_all.deb ...  
Unpacking liberror-perl (0.17030-1) ...  
Selecting previously unselected package git-man.  
Preparing to unpack .../6-git-man_1%3a2.51.0-1ubuntu1_all.deb ...  
Unpacking git-man (1:2.51.0-1ubuntu1) ...  
Selecting previously unselected package git.  
Preparing to unpack .../7-git_1%3a2.51.0-1ubuntu1_arm64.deb ...  
Unpacking git (1:2.51.0-1ubuntu1) ...  
Selecting previously unselected package ubuntu-fan.  
Preparing to unpack .../8-ubuntu-fan_0.12.17_all.deb ...  
Unpacking ubuntu-fan (0.12.17) ...  
Setting up runc (1.3.3-0ubuntu1-25.10.3) ...  
Setting up liberror-perl (0.17030-1) ...  
Setting up bridge-utils (1.7.1-4ubuntu2) ...  
Setting up pigz (2.8-1) ...  
Setting up git-man (1:2.51.0-1ubuntu1) ...  
Setting up containerd (2.1.3-0ubuntu3.1) ...  
Created symlink '/etc/systemd/system/multi-user.target.wants/containerd.service' -> '/usr/lib/systemd/system/containerd.service'.  
Setting up ubuntu-fan (0.12.17) ...  
Created symlink '/etc/systemd/system/multi-user.target.wants/ubuntu-fan.service' -> '/usr/lib/systemd/system/ubuntu-fan.service'.  
Setting up docker.io (28.2.2-0ubuntu1) ...  
Created symlink '/etc/systemd/system/multi-user.target.wants/docker.service' -> '/usr/lib/systemd/system/docker.service'.  
Created symlink '/etc/systemd/system/sockets.target.wants/docker.socket' -> '/usr/lib/systemd/system/docker.socket'.  
Setting up git (1:2.51.0-1ubuntu1) ...  
Processing triggers for man-db (2.13.1-1) ...  
ubuntu@user:~$
```

Figure 2: Finishing the installation of the Docker Engine and its basic dependencies.

Before application development began, the engine was verified. As shown in Figure 3, the hello-world test was run. This is an essential step in cloud engineering termed as "smoke testing". It verifies that Docker was able to connect to the internet and download a small image, create a container and print a message. The output starting with "Hello from Docker!", is confirmation to continue with the actual project [2].

```
ubuntu@user: ~  
Unpacking git (1:2.51.0-1ubuntu1) ...  
Selecting previously unselected package ubuntu-fan.  
Preparing to unpack .../8-ubuntu-fan_0.12.17_all.deb ...  
Unpacking ubuntu-fan (0.12.17) ...  
Setting up runc (1.3.3-0ubuntu1-25.10.3) ...  
Setting up liberror-perl (0.17030-1) ...  
Setting up bridge-utils (1.7.1-4ubuntu2) ...  
Setting up pigz (2.8-1) ...  
Setting up git-man (1:2.51.0-1ubuntu1) ...  
Setting up containerd (2:1.3-0ubuntu3.1) ...  
Created symlink '/etc/systemd/system/multi-user.target.wants/containerd.service' -> '/usr/lib/systemd/system/containerd.service'.  
Setting up ubuntu-fan (0.12.17) ...  
Created symlink '/etc/systemd/system/multi-user.target.wants/ubuntu-fan.service' -> '/usr/lib/systemd/system/ubuntu-fan.service'.  
Setting up docker.io (28.2.2-0ubuntu1) ...  
Created symlink '/etc/systemd/system/multi-user.target.wants/docker.service' -> '/usr/lib/systemd/system/docker.service'.  
Created symlink '/etc/systemd/system/sockets.target.wants/docker.socket' -> '/usr/lib/systemd/system/docker.socket'.  
Setting up git (1:2.51.0-1ubuntu1) ...  
Processing triggers for man-db (2.13.1-1) ...  
ubuntu@user:~$ sudo docker run hello-world  
Unable to find image 'hello-world:latest' locally  
latest: Pulling from library/hello-world  
198f93fd5094: Pull complete  
Digest: sha256:85404b3c53951c3ff5d40de0972b1bb21fafa2e8daa235355baf44f33db9dbdd  
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.  
   (arm64v8)  
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/  
  
ubuntu@user:~$ ^C  
ubuntu@user:~$
```

Figure 3: Successfully running 'hello-world' test of Docker to check system health.

2.3 Core Principles and Benefits of Containerisation

Efficiency and portability are the main reasons for using containers instead of virtual machines. While a VM may take minutes to boot because it must start an entire operating system, a Docker container can start in less than a second because it shares the host operating system kernel. Containers are therefore isolated processes running on the main operating system rather than full machines with separate kernels [2].

The second advantage is portability. One of the most common issues encountered in the field of software engineering is the inability of a program to perform in a certain way in a local development environment and fail when it is deployed to a production server due to the inconsistency of the environmental context. This typically occurs when the developer's laptop has different settings to the server. Docker achieves this by bundling the application code and required dependencies in an environment that suits it. This means that the code will behave exactly the same way on a student's laptop, on a testing server or from within a giant AWS data center [2].

2.4 Developing the Application and the Dockerfile Blueprint

To demonstrate this, a custom web server was created. First, content was required. As illustrated in Figure 4, the directory, cloud-demo, was created and a simple index.html file was written. The file contains a header and two short paragraphs confirming that the page is running inside a Docker container.

```
ubuntu@user: ~/cloud-demo
ubuntu@user:~$ mkdir cloud-demo
ubuntu@user:~$ cd cloud-demo
ubuntu@user:~/cloud-demo$ nano index.html
ubuntu@user:~/cloud-demo$ ls
index.html
ubuntu@user:~/cloud-demo$ cat index.html
<!DOCTYPE html>
<html>
<head>
  <title>Demo of Cloud Engineering</title>
</head>
<body>
  <h1>Hello from my Docker container</h1>
  <p>Module: ACCA7027 Cloud Engineering Principles and Practice</p>
  <p>This webpage is running inside a Docker container.</p>
</body>
</html>
ubuntu@user:~/cloud-demo$
```

Figure 4: Developing the homepage, or index.html, of the web server.

This was followed by the instructions used to build the server in Docker. This is done through a Dockerfile. As shown in Figure 5, the Dockerfile contains three simple lines. The image uses nginx:alpine as its base. Nginx is a widely used web server and Alpine is an ultra-lightweight version of Linux. This is followed by copying the HTML file into the container and exposing port 80 so that web traffic can access it [2].

```
ubuntu@user:~/cloud-demo$ nano Dockerfile
ubuntu@user:~/cloud-demo$ ls
Dockerfile  index.html
ubuntu@user:~/cloud-demo$ cat Dockerfile
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/index.html
EXPOSE 80
ubuntu@user:~/cloud-demo$
```

Figure 5: The creation of the Dockerfile to define the container environment.

2.5 Image Construction and the Web Server Lifecycle

In principle, an image is a blueprint of an application. It is a template that does not change, while a container is the running form of that template. As shown in Figure 6, the docker build command was used to convert the Dockerfile into an image. The terminal displays the downloading of the Nginx base layers and the addition of the HTML file as the topmost layer. The resulting image was labelled cloud-demo:v1 [2].

```
ubuntu@user: ~/cloud-demo
~/cloud-demo

Step 1/3 : FROM nginx:alpine
alpine: Pulling from library/nginx
d8ad8cd72600: Pulling fs layer
1595112a18c2: Pulling fs layer
0855a7fcacab: Pulling fs layer
96522e0e3029: Pulling fs layer
26a9266e72a1: Pulling fs layer
c0700f204f65: Pulling fs layer
21c8549f3077: Pulling fs layer
b15a4549c6e6: Pulling fs layer
96522e0e3029: Waiting
c0700f204f65: Waiting
21c8549f3077: Waiting
26a9266e72a1: Waiting
b15a4549c6e6: Waiting
0855a7fcacab: Verifying Checksum
0855a7fcacab: Download complete
1595112a18c2: Verifying Checksum
1595112a18c2: Download complete
d8ad8cd72600: Download complete
d8ad8cd72600: Pull complete
1595112a18c2: Pull complete
0855a7fcacab: Pull complete
96522e0e3029: Verifying Checksum
96522e0e3029: Pull complete
26a9266e72a1: Download complete
26a9266e72a1: Pull complete
c0700f204f65: Verifying Checksum
c0700f204f65: Download complete
c0700f204f65: Pull complete
21c8549f3077: Verifying Checksum
21c8549f3077: Download complete
21c8549f3077: Pull complete
b15a4549c6e6: Verifying Checksum
b15a4549c6e6: Download complete
b15a4549c6e6: Pull complete
Digest: sha256:5bad1ddd1ae0bc2cf5d90a6141566cc3f0e05c6deca4106ef04ff5d8a94b280
Status: Downloaded newer image for nginx:alpine
--> 2f3ea53bb0bb
Step 2/3 : COPY index.html /usr/share/nginx/html/index.html
--> cd511f769c33
Step 3/3 : EXPOSE 80
--> Running in 92275b02b2d2
--> Removed intermediate container 92275b02b2d2
--> 9f66773e5333
Successfully built 9f66773e5333
Successfully tagged cloud-demo:v1
ubuntu@user:~/cloud-demo$
```

Figure 6: Constructing the Docker image by using the Dockerfile blueprint.

Once the image is created, it must be executed as a running container. As shown in Figure 7, the docker run command was used with two important settings. The -d flag runs the container in background mode, while the -p 8080:80 option maps the container port to the VM. This forms a network path through which the website can be accessed [2].

```
ubuntu@user:~/cloud-demo
~/cloud-demo

ubuntu@user:~/cloud-demo$ sudo docker run -d --name cloud-demo-container -p 8080:80 cloud-demo:v1
d4685a17f3ebf6cc8450a048c0f42333893ec6b45f34181cd060e4cf26c3828
ubuntu@user:~/cloud-demo$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                               NAMES
d4685a17f3eb   cloud-demo:v1   "/docker-entrypoint..." 23 seconds ago Up 22 seconds 0.0.0.0:8080->80/tcp, [::]:8080->80/tcp   cloud-demo-container
ubuntu@user:~/cloud-demo$
```

Figure 7: Launching container and map network ports

Finally, as shown in Figure 8, a web browser was opened at localhost:8080. The screen shows the custom webpage, confirming that the Docker container is active, healthy and serving web traffic as expected.

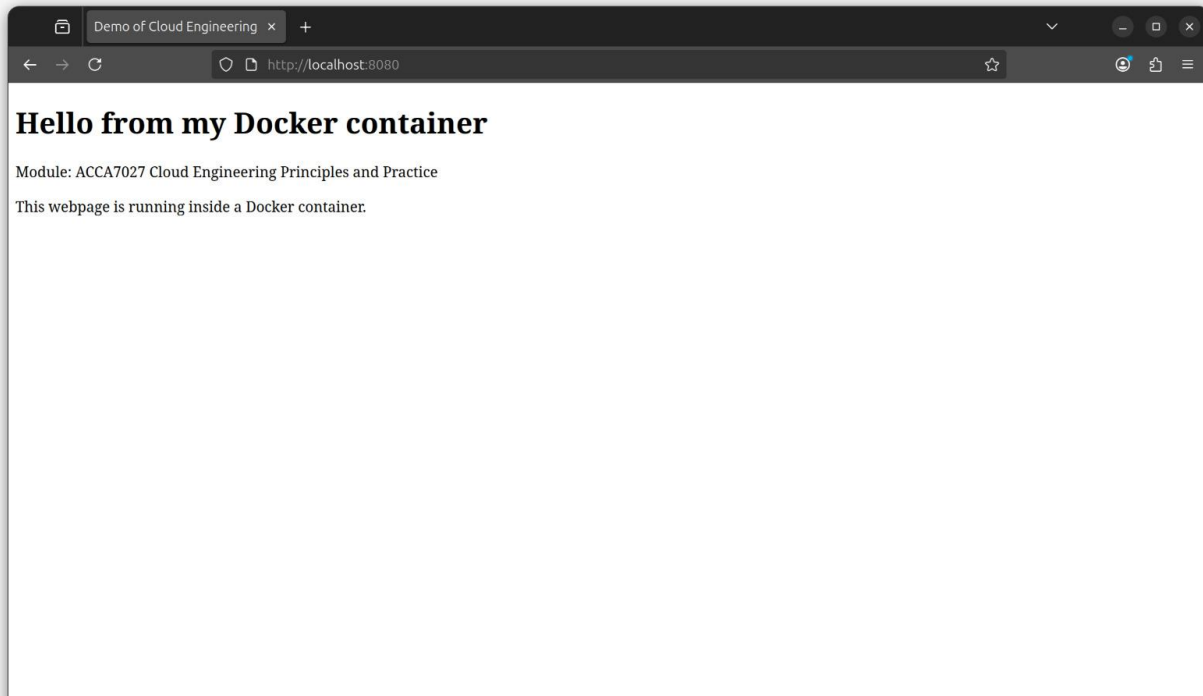


Figure 8: Testing the web server is running in a browser.

2.6 Publishing to the Cloud: Docker Hub Registry

A container stored only on a local machine has limited value in a cloud context. The image must be uploaded to a registry so that it can be downloaded by colleagues or deployed on cloud servers. The most widely used public registry is Docker Hub [2].

First, as shown in Figure 9, authentication to Docker Hub was required with the `docker login` command and a web-based authentication token. This ensures that only authorised users can upload images to the relevant account [2].

```
ubuntu@user: ~/cloud-demo
~/cloud-demo

ubuntu@user:~/cloud-demo$ sudo docker login
[sudo: authenticate] Password:

USING WEB-BASED LOGIN

Info -> To sign in with credentials on the command line, use 'docker login -u <username>'

Your one-time device confirmation code is: LFZN-VQFB
Press ENTER to open your browser or submit your device code here: https://login.docker.com/activate
Waiting for authentication in the browser...

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

Login Succeeded
ubuntu@user:~/cloud-demo$
```

Figure 9: Successfully logging in to the Docker Hub registry.

Next, as demonstrated in Figure 10, the image had to be tagged with the Docker Hub username and then pushed to the internet. The terminal screen displays Docker uploading the unique layers of the image to global cloud servers [2].

```
ubuntu@user: ~/cloud-demo
ubuntu@user:~/cloud-demo$ sudo docker login
[sudo: authenticate] Password:

USING WEB-BASED LOGIN

Info → To sign in with credentials on the command line, use 'docker login -u <username>'

Your one-time device confirmation code is: LFZN-VQFB
Press ENTER to open your browser or submit your device code here: https://login.docker.com/activate
Waiting for authentication in the browser...

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

Login Succeeded
ubuntu@user:~/cloud-demo$ sudo docker tag cloud-demo:v1 mdahasaulkobiropy/cloud-demo:v1
ubuntu@user:~/cloud-demo$ sudo docker push mdahasaulkobiropy/cloud-demo:v1
The push refers to repository [docker.io/mdahasaulkobiropy/cloud-demo]
0b3831ed1533: Pushed
b7a361813e4b: Mounted from library/nginx
8d82fe761162: Mounted from library/nginx
7af56d46d250: Mounted from library/nginx
9590d4725929: Mounted from library/nginx
02f406f31f40: Mounted from library/nginx
67252101b42f: Mounted from library/nginx
d34aa5681408: Mounted from library/nginx
45f3ea5848e8: Mounted from library/alpine
v1: digest: sha256:a00cb906a5e0d0b67edce2fdcc02d5050697563f719818ab26c0adaf36b526d1 size: 2196
ubuntu@user:~/cloud-demo$
```

Figure 10: Pushing the image to the Docker Hub repository.

Verification was completed on the Docker Hub website. As seen in Figure 11, the image of the cloud-demo is now clearly visible. This means that anyone in the world can download the web server with a single command.

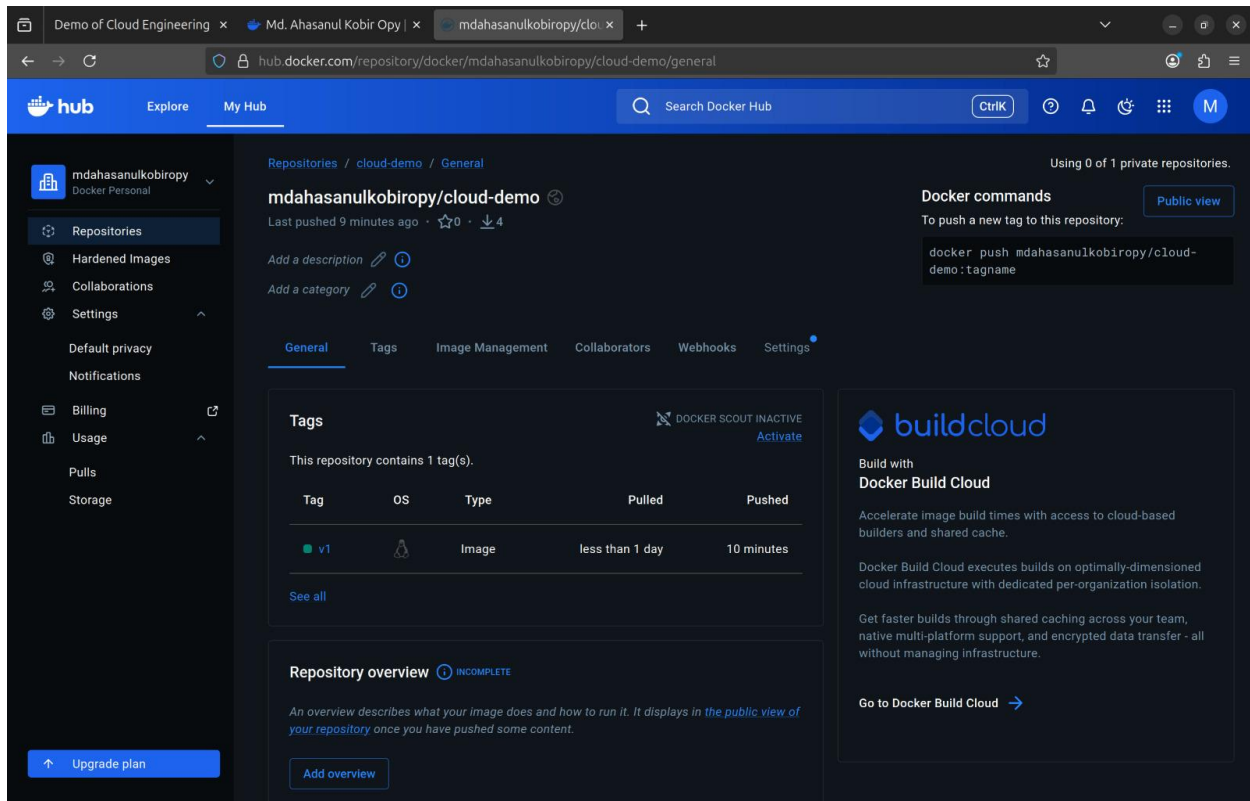


Figure 11: Verifying that the image is publicly available in Docker Hub.

2.7 Data Persistence: Managing Container Volumes

One of the most important characteristics of containers is that container environments are ephemeral. If a container crashes or is deleted, the data stored inside that container can be lost permanently. This creates a problem for stateful applications, including databases, that require persistent data storage. Docker addresses this issue through volumes. A volume is a safe storage space on the host computer that can be accessed by the container [2].

As shown in Figure 12, a volume named `my-demo-volume` was created. Next, as shown in Figure 13, a test container was run and the volume was attached. A text file was written within the volume, which was then deleted. Even though the container was gone, the information remained safe on the host machine. This is how to store user passwords, bank details in the cloud.

```
ubuntu@user:~/cloud-demo$ sudo docker volume create my-demo-volume
my-demo-volume
ubuntu@user:~/cloud-demo$ sudo docker volume ls
DRIVER      VOLUME NAME
local       my-demo-volume
ubuntu@user:~/cloud-demo$
```

Figure 12: The development of a persistent Docker volume.

```
ubuntu@user: ~/cloud-demo
~/cloud-demo
ubuntu@user:~/cloud-demo$ sudo docker run --rm -v my-demo-volume:/data alpine sh -c "echo 'This file is stored in Docker volume' > /data/test.txt && cat /data/test.txt"
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
d8ad8cd72600: Already exists
Digest: sha256:25109184c71bdad752c8312a8623239686a9a2071e8825f20acb8f2198c3f659
Status: Downloaded newer image for alpine:latest
This file is stored in Docker volume
ubuntu@user:~/cloud-demo$
```

Figure 13: Testing Data Persistence by Writing to Volume

2.8 Container Communication: Virtual Networking

A real-world application may use one container for the website and another for the database. These two containers must communicate with each other. Docker uses virtual networks to enable this communication in a more secure way than exposing every service directly to the public internet [2].

As shown in Figure 14, a new virtual network, my-demo-network, was created. Finally, as shown in Figure 15, the web server was joined to this network, and the docker inspect command was used to confirm that it was given a private IP address. This confirms that the container is operating within a safe, internal cloud network.

```
ubuntu@user: ~/cloud-demo
~/cloud-demo
ubuntu@user:~/cloud-demo$ sudo docker network create my-demo-network
457179605b5a0f2913fbb34bf6f1051518909c797cc80835de9039bc3166336
ubuntu@user:~/cloud-demo$ sudo docker network ls


| NETWORK ID   | NAME            | DRIVER | SCOPE |
|--------------|-----------------|--------|-------|
| fe8829c3a649 | bridge          | bridge | local |
| c66fae063776 | host            | host   | local |
| 457179605b5a | my-demo-network | bridge | local |
| ed8925c4922d | none            | null   | local |


ubuntu@user:~/cloud-demo$
```

Figure 14: How to build a private and isolated Docker network.

```
ubuntu@user: ~/cloud-demo
~/cloud-demo
ubuntu@user:~/cloud-demo$ sudo docker network connect my-demo-network cloud-demo-container
ubuntu@user:~/cloud-demo$ sudo docker inspect cloud-demo-container
[
  {
    "Id": "d4685a17f3ebfccb8450a048c0f4233893ec6b45f34181cd060e4cf26c3828",
    "Created": "2026-03-11T09:14:52.705368476Z",
    "Path": "/docker-entrypoint.sh",
    "Args": [
      "nginx",
      "-g",
      "daemon off;"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 8028,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2026-03-11T09:14:52.737891583Z",
      "FinishedAt": "0001-01-01T00:00:00Z"
    },
    "Image": "sha256:9f66773e5333b2df3600987be83435300520181e9d9f833962c6176c115d3a08",
    "ResolvConfPath": "/var/lib/docker/containers/d4685a17f3ebfccb8450a048c0f4233893ec6b45f34181cd060e4cf26c3828/resolv.conf",
    "HostnamePath": "/var/lib/docker/containers/d4685a17f3ebfccb8450a048c0f4233893ec6b45f34181cd060e4cf26c3828/hostname",
    "HostsPath": "/var/lib/docker/containers/d4685a17f3ebfccb8450a048c0f4233893ec6b45f34181cd060e4cf26c3828/hosts",
    "LogPath": "/var/lib/docker/containers/d4685a17f3ebfccb8450a048c0f4233893ec6b45f34181cd060e4cf26c3828/d4685a17f3ebfccb8450a048c0f4233893ec6b45f34181cd060e4cf26c3828-jso
n.log",
    "Name": "/cloud-demo-container",
    "RestartCount": 0,
    "Driver": "overlay2",
    "Platform": "linux",
    "MountLabel": "",
    "ProcessLabel": "",
    "AppArmorProfile": "docker-default",
    "ExecIDs": null,
    "HostConfig": {
      "Binds": null,
      "ContainerIDFile": "",
      "LogConfig": {
        "Type": "json-file",
        "Config": {}
      },
      "NetworkMode": "bridge",
      "PortBindings": {
        "80/tcp": [
          {
            "HostIP": "0.0.0.0",
            "HostPort": "80",
            "ContainerPort": 80,
            "Protocol": "tcp"
          }
        ]
      }
    }
  }
]

ubuntu@user:~/cloud-demo$ sudo docker inspect cloud-demo-container | grep my-demo-network
      "my-demo-network": {
        "IPAMConfig": {},
        "Links": null,
        "Aliases": [],
        "MacAddress": "ce:a7:8e:95:62:4e",
        "DriverOpts": {},
        "QwPriority": 0,
        "NetworkID": "457179605b5a0f2913fbb34fb6f10515189b9c797cc80835de9039bc3166336",
        "EndpointID": "8f12d195d16630416bb0d4f0f0814658689da8dc425885d24e64f5789930a19b",
        "Gateway": "172.18.0.1",
        "IPAddress": "172.18.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "DNSNames": [
          "cloud-demo-container",
          "d4685a17f3eb"
        ]
      }
    ]
  }
]
ubuntu@user:~/cloud-demo$ sudo docker inspect cloud-demo-container | grep my-demo-network
      "my-demo-network": {
ubuntu@user:~/cloud-demo$
```

Figure 15: Examining the container setup to check for network connectivity.

3. Cloud Computing Concepts

3.1 Practical Application: The E-Commerce Use Case

To grasp the concepts of high-level clouds, it is best to consider an example of a real-world situation. Imagine having a large E-commerce Website (similar to an online clothing store). During a normal day maybe 500 people will be browsing the website. But in a "Black Friday" sale, that number may rise to 500,000 within a few minutes.

If the website is only on a single server, the server will immediately run out of memory and crash. The business will lose millions of dollars in sales. To prevent this, cloud engineers have three major tools: Scalability, Load Balancing, and Application Gateways.

3.2 Scalability: Managing Dynamic Workloads

Scalability is the ability of a system to do more work as resources are added. Traditionally, vertical scaling involved purchasing a bigger server with more memory. This has obvious limitations as the server can only be so large. So in cloud computing, horizontal scaling is used extensively, where many smaller servers, or instances, are grouped together to handle the load.

The Cloud Provider Approach: Amazon Web Services (AWS) has a feature called Auto Scaling Groups. The cloud provider monitors the utilisation of the website's CPU. During peak shopping times, when the utilisation is too high, AWS can spin up many new Docker containers or instances. When utilisation is low, AWS can shut down extra instances so that the company does not waste money [4].

3.3 Load Balancing: Intelligent Traffic Distribution

When multiple servers are operating simultaneously, traffic needs to be distributed across multiple servers. If all 500,000 customers try to connect to "Server #1" at the same time, this server will be overwhelmed while the others will remain underutilised. To prevent this, a traffic distribution layer is needed [3].

The Cloud Provider Approach: This traffic control layer is known as a Load Balancer (for instance, AWS Elastic Load Balancer or Azure Load Balancer). When a customer enters the URL of a website, the traffic first goes to the load balancer. The load balancer scans the servers and routes the customer to the server with the lowest traffic. This allows the capacity to be efficiently used and no server is overloaded [3].

3.4 Application Gateways: Smart Routing and Security

Load balancers are commonly used at the OSI Layer 4, where traffic is routed based on IP addresses and TCP ports but not the content. An Application Gateway, on the other hand, is at Layer 7. At this layer, HTTP and HTTPS traffic can be analysed and decisions can be made on how to route traffic based on request type, path or any other application-specific information [5].

The Cloud Provider Approach: AWS and Azure both have application gateway services. For example, a request to get an image for a product can be directed to an image server, and a request to make a payment can be directed to an encrypted payment server. The gateway can also provide security, such as a Web Application Firewall, to prevent malicious code from reaching the web site [5].

3.5 The Symbiotic Relationship: Scaling and Balancing

It is very crucial to understand that in cloud systems scaling and load balancing are symbiotic [3]. Scaling gives the capacity [4]. Load Balancing gives the direction [3].

- I. Scaling gives the capacity (it builds the extra servers).
- II. Load Balancing gives the direction (it tells the traffic where to go).

If scaling is increased by adding 10 new servers but no load balancer is present, the website may still go down because traffic will still go to the original server. Therefore, scaling and load balancing go hand in hand to create a highly available system that is less likely to experience outages.

3.6 Customer Impact: Performance, Reliability, and Trust

When an e-commerce company is able to adopt new cloud-based architecture, the customer benefits. Load balancing allows for better load distribution and service availability [3]. Elastic scaling enhances performance with variable loads [4]. Security gateways enhance security and trust [5].

Performance (Speed and Responsiveness):	A. Through horizontal scaling, the website can keep the response time low even during peak traffic periods (e.g. Black Friday, flash sales). B. The customer experiences a fast user interface because the resources are allocated dynamically based on demand (no delays as with conventional servers that have a fixed capacity).
Reliability (The "Always-On" Experience):	A. The load balancer is employed as a sort of a traffic cop, and it periodically health-checks the backends. B. In the event that one of the servers goes down, the load balancer will automatically redirect traffic to the next available server. As a result, the customer is never presented with a "service unavailable" or "request failed" message and a seamless purchase process is ensured.
Trust (Security and Data Integrity):	A. The Application Gateway is a security shield that scans traffic entering for suspicious traffic patterns before the database is reached. B. This helps to protect against malicious attacks such as SQL injection or Cross Site Scripting (XSS) and ensures that data (credit card data, home addresses, etc.) is safe. This robust security helps to foster the brand and customer loyalty.

4. The Future of Cloud Services

4.1 Driving Forces: The Economics of Cloud Expansion

The rapid growth of cloud computing has been caused by a fundamental change in financial strategy away from physical assets being fixed and towards a utility-based model. This shift can be explained by the use of two main economic approaches: [6]

- I. **Capital Expenditure (CapEx) – The Traditional Risk:**
 - A. **Upfront Investment:** Needs a lot of "sunk costs" to purchase physical servers, networking hardware and cooling systems before one line of code is ever run.
 - B. **Risk of Failure:** If a project or business fails, it is impossible to get all of the \$10,000+ back on the hardware, creating a massive barrier to entry.
 - C. **Inflexible Scaling:** Organizations have to over-provision hardware to accommodate peak demand, meaning that costly hardware may remain idle during normal operations.
- II. **Operational Expenditure (OpEx) – The Cloud Advantage:**
 - A. **Pay-As-You-Go Pricing:** Cloud providers use a utility-based model similar to home utilities. Organisations are charged only for the amount of CPU, storage and bandwidth used.
 - B. **Low Barrier to Entry:** Cloud pricing models lower entry barriers by allowing organisations to rent infrastructure as needed, which enables competition with much larger institutions on a more even playing field.
 - C. **Elasticity and Efficiency:** Resources can be switched off immediately when no longer needed, eliminating the cost of idle hardware and aligning spending more closely with current business growth.
- III. **Strategic Impact on Innovation:**
 - A. **Rapid Experimentation:** Since the price of failure is lowered to the cost of a few hours of server time, fast teams are able to fail fast and repeat without compromising the financial stability of the company.
 - B. **Agility:** A shift to OpEx gives the business the ability to switch the strategy immediately and redirect investment funds towards product development and customer experience instead of infrastructure maintenance [6].

4.2 Flexibility: Accelerating Innovation and Speed to Market

The second key factor is Flexibility. In previous generations of IT infrastructure, obtaining a new server often required approval and a lead time for the server to be delivered and installed. In the cloud, a developer can create 1,000 servers in 60 seconds with a script [6].

This agility enables experimentation. A new feature can be deployed for just one day and, if the result is unsuccessful, the servers can be shut down with a minimal cost. This is one of the reasons why firms like Netflix and Spotify can release new versions of their apps so often.

4.3 Serverless Functions: The Next Evolution in Abstraction

After virtual machines and containers, the next abstraction is serverless. The most recent step is "Serverless" (also known as FaaS - Functions as a Service). Examples are AWS Lambda and Azure Functions [7].

In a "Serverless" model the developer does not have to deal with servers (or containers). The developer uploads a piece of code (or function). The function is not executed until it is invoked and the cost is incurred only when the function is executed [7].

As an example, one serverless function might resize an uploaded photo into a thumbnail. If no photos are uploaded during the day, there is no compute charge for execution. If 1,000 photos are uploaded at the same time, the cloud provider can execute the code 1,000 times and then scale back automatically. This is a high level of automation [7].

4.4 Comparative Analysis: Containers vs. Serverless

However, this is only possible if the workload allows for such an approach.

I. Cost Comparison:

- A. **Containers (Docker):** Ideal in the case of applications that operate throughout the day. Since the server is constantly running, a constant and predictable hourly fee is charged [7].
- B. **Serverless:** Ideal for activities that occur occasionally. Serverless can be nearly free when code runs for only 10 seconds a day. However, when an application runs 24/7, serverless can actually cost more than Docker [7].

II. Flexibility and Control:

- A. **Containers (Docker):** Containers give developers more control over where code runs, including choices about the operating system and the way application components work together. This reduces vendor lock-in and makes migration of the Docker container to another cloud platform easier [7].
- B. **Serverless Platforms:** By virtue of the cloud provider's management of the execution environment, this necessarily entails a reduction in the control available over the underlying infrastructure. The cloud provider (like Amazon) has the freedom to decide how the environment works. This creates "Vendor Lock-in"-It is very hard to move a Lambda function from Amazon to Microsoft [7].

To conclude, Docker is to be used in large and complicated websites that require 24/7 availability. Serverless functions excel when used for short, reactive tasks or for handling background jobs that run occasionally.

Another important issue is that of latency, and the so-called cold starts of serverless architectures. Since serverless functions scale down to zero during idle times for cost optimization, the initial request after an idle period will experience a time penalty. There will be a startup delay for the cloud computing provider, as it will take some resources and bootstrapping code to be executed. As a result, even a small delay can make serverless architectures inappropriate in highly time-sensitive applications that are real-time and where each millisecond matters [6].

5. Conclusion

The notion of modern cloud engineering represents a fundamental change in the design and operation of information systems. As illustrated in this report with practical experience of using Docker, containerization enhances portability, consistency, data persistence and secure inter-service communication. The report also investigated the effect of scalability, load balancing and application gateways on the reliability of cloud services during the peak load (such as e-commerce applications). Finally, the report has shown that cost and flexibility are the key factors behind the adoption of cloud computing, while the decision to use containerization or serverless technology is based on the application requirements.

6. References

- [1] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.
- [2] Docker Inc., "Docker Documentation: Get Started," docs.docker.com, 2024. [Online]. Available: <https://docs.docker.com/get-started/>. [Accessed: 10-Mar-2026].
- [3] Amazon Web Services, "What is Load Balancing?," aws.amazon.com, 2023. [Online]. Available: <https://aws.amazon.com/what-is/load-balancing/>. [Accessed: 10-Mar-2026].
- [4] Amazon Web Services, "What is Amazon EC2 Auto Scaling?," AWS Documentation, 2026. [Online]. Available: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html>. [Accessed: 12-Mar-2026].
- [5] Microsoft, "What is Azure Application Gateway?," Microsoft Learn, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/overview>. [Accessed: 12-Mar-2026].
- [6] D. Taibi, J. Spillner, and K. Wawruch, "Serverless Computing—Where Are We Now, and Where Are We Heading?," IEEE Software, vol. 38, no. 1, pp. 22-27, Jan.-Feb. 2021.
- [7] P. Sbarski, Serverless Architectures on AWS. Shelter Island, NY: Manning Publications, 2017.
- [8] P. Mell and T. Grance, The NIST Definition of Cloud Computing, NIST Special Publication 800-145. Gaithersburg, MD: National Institute of Standards and Technology, 2011.
- [9] J. Turnbull, The Docker Book: Containerization is the new virtualization. Self-published, 2014.
- [10] N. Poulton, Docker Deep Dive. Leanpub, 2020.