



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

Cloud Engineering Principles and Practice

ACCA7027 Accessment Report

Student Name: Gandhi Raj Rakeshkumar

Student ID: 2503898

Program: MSC Cybersecurity and Digital Forensics

Submission To: Renuga Jayakumar

Table of Contents

1. Introduction.....	1
2. Cloud Technology (Docker) and Examples.....	1
2.1 Principles of Virtualisation	1
2.2 Principles and Benefits of Containerisation.....	3
2.3 Docker Images and Publishing Images.....	4
2.4 Container Volumes	6
2.5 Container Networking.....	7
2.6 Full End-to-End Workflow Example.....	11
2.7 Video Link	12
3. Cloud Computing Concepts.....	13
3.1 Load Balancing, Scalability, and Application Gateways.....	13
3.2 Complementarity of Load Balancing and Scaling	17
3.3 Benefits of Combining Load Balancing, Scaling, and Application Gateways	18
4. Main Driving Forces Leading to the Expansion of Cloud and Serverless vs Containerisation	19
4.1 Driving Forces for Cloud Expansion	19
4.2 Serverless Functions (Functions as a Service).....	20
4.3 Compare & Contrast Serverless vs Containerisation.....	21
4.4 Influence of Driving Forces on Deployment Choice.....	21
5. Conclusion	22
References.....	23

1. Introduction

Cloud computing is now an essential part of the digital infrastructure today, where organisations are able to implement applications more efficiently, flexibly and reliably. With the continued growth of cloud environments, engineering methods that enable the deployment of applications at scale and in a portable manner have become increasingly important. Containerisation is one of these practices, which has become one of the most important technologies because it enables applications and dependencies to be packaged into lightweight, consistent, and isolated units. One of the most popular container platforms is Docker, which is a key driver in making application deployment simpler, regardless of the environment type, whether it is a local development machine or a large-scale cloud-based system. The report examines the concepts of containerisation and associated cloud engineers, and how technologies like Docker can help in efficient resource utilisation and reliability in the system. The discussion also covers the basics of virtualisation, building and sharing of container images, and providing a utilisation of volumes to provide persistence and networking to support multiple containers. Also, larger cloud computing ideas, i.e. load balancing, scalability and redirecting to applications, are also looked at to realise the way in which cloud providers can provide high availability and performance on contemporary applications. The main factors that have contributed to cloud computing growing as rapidly as it has, such as cost and architectural flexibility, are also examined, as well as a comparison of the container-based architecture to the serverless architecture of cloud computing.

2. Cloud Technology (Docker) and Examples

2.1 Principles of Virtualisation

Virtualisation is one of the biggest changes in the current computing architecture, which serves as a base for the cloud environment. In its most fundamental form, virtualisation is a process that abstracts physical hardware and allows a plurality of isolated operating systems to be executed on one physical machine. This is achieved with the assistance of a software layer, which is referred to as a hypervisor. There are two broad categories of Hypervisors: Type 1, which are run on the actual physical hardware, and Type 2, which are run on a host operating system. Both technologies require and implement virtual machines (VMs), where all of them act as a fully functional and autonomous computing system that has resources of

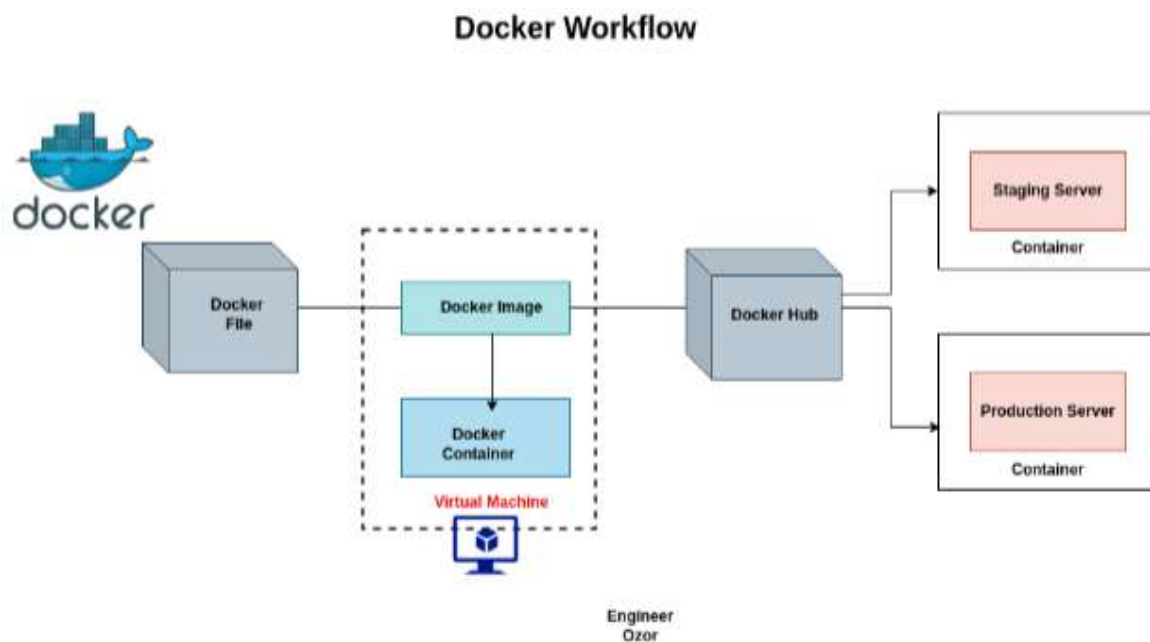
its own, including a CPU, memory, storage, and network interfaces (Agrawal and Singh, 2024).

The advantage of virtualisation consists of the ability to pack hardware assets together. Virtualisation enables multiple applications to coexist in distinct VMs on the same server instead of using one whole server dedicated to a specific application of a server. This significantly enhances hardware usage, and the number of physical servers needed in a data centre will be minimised. Moreover, virtualisation offers great isolation of guest operating systems. When a fault, crash or security breach occurs in a single VM, it does not impact anyone directly, which is a key requirement given the multi-tenant cloud environment. Virtualisation also helps in the efficiency of operations by the use of snapshots, cloning and migration features. The snapshots enable the saving of the entire VM state at any moment, which is why it is possible to roll back or duplicate a VM state. Live migration will allow you to transfer a running VM between physical hosts with minimal interruption to allow maintenance, like work, to take place without interruption.

One application of virtualisation in the practical sense is that hardware resources are shared among several VMs that run on the same server. One physical server with a big CPU and a large amount of memory can have many VMs operating at the same time, with each having different workloads or even completely different operating systems (Kaur, Josan and Neeru, 2022). The hypervisor uses a set of policies in the distribution of CPU cycles, memory and network bandwidth, in ensuring that resource distribution is stable and performance predictable across all VMs.

Virtualisation has overheads, although it offers isolation and flexibility since a VM has a complete operating system. This results in quite a slow boot time and huge CPU and memory usage. Because of these shortages, containers were proposed to provide a solution to these limitations, which is lightweight, yet preserves high application isolation. The containers, in contrast to the VMs, use the same host operating system kernel but separate user-space environments. This gives them greater efficiency, more start-up speed, and simpler scaling.

2.2 Principles and Benefits of Containerisation



Containerisation is a new and dominant technology in the cloud engineering sphere because of its efficiency, consistency, and support of contemporary practices in development like microservices and DevOps. One can define a container as an isolated user-space owning an application and all of the necessary libraries and dependencies. Containers do not include a complete operating system, but they use the kernel of the host system, which makes them more differentiated than virtual machines and consumes significantly less overhead.

Principles of Containerisation:

Containerisation is based technologically on OS-level virtualisation. It relies on capabilities of the host operating system, such as namespaces and cgroups, in order to isolate processes and/or regulate access to system resources and give safe execution environments (Baloni et al., 2023). Every container is equivalent to running on a dedicated system, although it has the same underlying kernel as other containers.

There is another principle of immutability. Container images are built up out of layers of filesystems, where each layer is a change, e.g. adding a library or moving application files. These layers are formed permanently. Such immutability guarantees that there is consistency in the environment; the same image should be used on both the machine of developer and a machine in the cloud.

Containerisation is also based on the layer filesystems, where several images can share common layers. Today, an example of such an application is a base Linux image that personifies a number of applications. Due to the shared base layers, the storage needs are

minimised, and the process of image updating is more efficient. It is only the modified layers that need to be rebuilt whenever modifications are necessary.

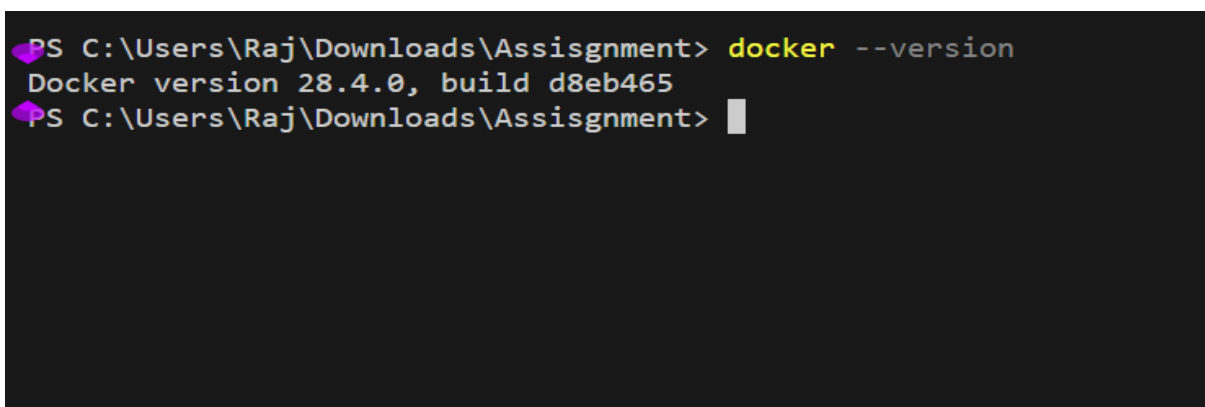
Another benefit is a quick time to start and an elevated density. Containers start very fast due to the fact that no complete operating system is required to be booted during the start. This is a very desirable feature, especially in the cloud environment where scaling is a must. With the utilisation of many containers, the host does not waste resources, thus improving utilisation and lowering the cost of infrastructure (Brooks, Vance and Ames, 2025).

Benefits of Containerisation:

Containerisation has several qualities that render it attractive to cloud engineering:

- **Portability:** Container images are containerised. A packaged application inside some type of container will run the same regardless of the local, test or cloud environment and eliminate compatibility problems.
- **Resource efficiency:** Containers use fewer system resources since they share the host kernel compared to virtual machines. This productivity is what allows organisations to operate more services using the same hardware.
- **Suitability for CI/CD:** Automated pipelines can be combined with containers. Images are created in one of the standardised manner, they are subject to testing, and are released as immutable objects, both factors contributing to the minimisation of errors and high deployment reliability.
- **Support for microservices architectures:** Containers give natural frontiers to small, independent services. Microservices may be installed in separate containers, which allows them to develop at a modular level, scale independently, and provide fault separation..

2.3 Docker Images and Publishing Images

A screenshot of a Windows command prompt window with a black background and white text. The prompt shows the user's location as 'C:\Users\Raj\Downloads\Assignment'. The command 'docker --version' has been entered, and the output is 'Docker version 28.4.0, build d8eb465'. The prompt is ready for the next command.

```
PS C:\Users\Raj\Downloads\Assignment> docker --version
Docker version 28.4.0, build d8eb465
PS C:\Users\Raj\Downloads\Assignment> █
```

Figure 1. Docker Image

1. Build Image: docker build –t rajgandhi06/docker-web-demo .

```

118
119 -> Running in a Docker Container /hi>
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
=> sha256:abdece946283a31d98ef184559f417a33c3a8936a80153b2f0ffa208af4a0d48: 954B / 954B 0.3s
=> sha256:5d2cc344426d3d91284b457a772efc975de834e185506f5cce5d6b863da1ca9: 629B / 629B 0.3s
=> sha256:dfad290a5c259f8d1ec1938529f8ef602a335a26080497ad56d38e0727e1a10a: 1.06MB / 1.06MB 0.7s
=> sha256:034e56e613966f73cce0858124ca5fbc601d7888099969a4aea69f31dc071a51: 3.86MB / 3.86MB 2.0s
=> extracting sha256:034e56e613966f73cce0858124ca5fbc601d7888099969a4aea69f31dc071a51: 0.3s
=> extracting sha256:dfad290a5c259f8d1ec1938529f8ef602a335a26080497ad56d38e0727e1a10a: 0.3s
=> extracting sha256:5d2cc344426d3d91284b457a772efc975de834e185506f5cce5d6b863da1ca9: 0.0s
=> extracting sha256:abdece946283a31d98ef184559f417a33c3a8936a80153b2f0ffa208af4a0d48: 0.0s
=> extracting sha256:51c30493937c33bd5b568d8aed08d9556f558d8877b95a5e185816aba85e1f: 8.8s
=> extracting sha256:ad5b65da02cfbd43daa87443b87051f3816a10eb7719938d8cc9a96ee828d471: 0.1s
=> extracting sha256:fc13532503d72b70e47dd276ae52f743b14326b83c31935c86d7477c08019dea: 0.0s
=> extracting sha256:136bc6976c2023a363a86b88187d08819faca3e756c162c58754a3029bf4863: 0.7s
=> [2/2] COPY index.html /usr/share/nginx/html/index.html 0.2s
=> exporting to image 0.7s
=> exporting layers 0.2s
=> exporting manifest sha256:2267faaa6ad613d5a71db7d1c92dabcbaf15b0b445f1ff92d9866efa8393c3a 0.0s
=> exporting config sha256:933aec7d88471dc0c38fe8e3dfde5821a6696955be92c9c7d73ae7ea500e565c 0.0s
=> exporting attestation manifest sha256:034ff745fd055af8960eccdd6953652935aeac123ccf89ac54fb6ac736d7f997 0.1s
=> exporting manifest list sha256:a5e2264fb387908fde38cf63c50252b2bc7c72c26ce7542a48e4de393af4f8d0 0.0s
=> naming to docker.io/rajgandhi06/docker-web-demo:latest 0.0s
=> unpacking to docker.io/rajgandhi06/docker-web-demo:latest 0.1s
PS C:\Users\Raj\Downloads\Assignment>

```

Figure 2. Docker Image Build

2. Push to DockerHub: docker push rajgandhi06/docker-web-demo

```

=> sha256:2267faaa6ad613d5a71db7d1c92dabcbaf15b0b445f1ff92d9866efa8393c3a
=> exporting config sha256:933aec7d88471dc0c38fe8e3dfde5821a6696955be92c9c7d73ae7ea500e565c
=> exporting attestation manifest sha256:034ff745fd055af8960eccdd6953652935aeac123ccf89ac54fb6ac736d7f997
=> exporting manifest list sha256:a5e2264fb387908fde38cf63c50252b2bc7c72c26ce7542a48e4de393af4f8d0
=> naming to docker.io/rajgandhi06/docker-web-demo:latest
=> unpacking to docker.io/rajgandhi06/docker-web-demo:latest
PS C:\Users\Raj\Downloads\Assignment> docker push rajgandhi06/docker-web-demo
Using default tag: latest
The push refers to repository [docker.io/rajgandhi06/docker-web-demo]
dfad290a5c25: Pushed
5d2cc344426d: Pushed
136bc6976c20: Pushed
54978284da34: Pushed
abdece946203: Pushed
ad5b65da02cf: Pushed
51c30493937c: Pushed
fc13532503d7: Pushed
034e56e61396: Pushed
014e56e61396: Pushed
latest: digest: sha256:a5e2264fb387908fde38cf63c50252b2bc7c72c26ce7542a48e4de393af4f8d0 size: 856
PS C:\Users\Raj\Downloads\Assignment>

```

Figure 3. Docker Push Image

The artefacts of containerised application deployment consist of container images. A picture has all the needed information to execute that; it is a set of application programs, resources, system tools, and configuration documentation. An image may be stored either on demand or publicly on a registry after it has been created.

Image publication is crucial to co-operation, automation and cloud deployment. Docker Hub, Azure Container Registry and Amazon Elastic Container Registry are registries where the images are stored and can be made available in various environments. Image uploading usually goes through the authentication with the registry, the appropriate tag that signifies the version of the image is applied, and the image is pushed to the remote repository (Raftopoulos, 2025).

After publication, other systems get access to the image. It is possible to have cloud platforms automatically draw the image when using containerised workloads. Build pipelines.

Continuous Integration/Continuous Delivery pipelines often rely on published images to make sure that all the environments are utilising the same checked and verified form of the application. Teams in the organisation share images with one another in a bid to ensure that there is a sense of uniformity in the environment of development, testing, and production.

2.4 Container Volumes

The fact that containers are generally temporary is one of their limitations. In case of a stopped container or removal, all data stored in the filesystem is destroyed. Persistent data is needed in many applications such as databases, content management systems and web servers.

Container volumes solve this restriction by offering storage that lives outside of the container lifecycle. One can create a volume on the host system, and this volume can be handled by the container engine. After it has been created, the volume may be installed in a single or several containers. It enables data to be stored even when containers are restarted, changed or replaced (Bandaru, 2023).

One common application is web server content or database file storage. Data within an application, configuration files, logs or uploaded by users may be contained in a volume. After being attached to a container, the volume acts similarly to a normal directory within the container, although it is not lost in case the container is destroyed.

One use of volumes will be to prepare a storage space with an HTML file and thereafter attach the volume to a container of a web server. The content to be stored in the volume is provided by the container, and the container has the contents even in case it needs to be re-created. This is basic in ensuring that stateful information is stored in stateless container spaces.

```
PS C:\Users\Raj\Downloads\Assignment> docker volume create mydata
mydata
PS C:\Users\Raj\Downloads\Assignment> docker inspect mydata
[
  {
    "CreatedAt": "2025-12-12T00:05:00Z",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/mydata/_data",
    "Name": "mydata",
    "Options": null,
    "Scope": "local"
  }
]
PS C:\Users\Raj\Downloads\Assignment> █
```

Figure 4. Docker Volume Creation

```
Run 'docker run --help' for more information
PS C:\Users\Raj\Downloads\Assignment> docker run -d -p 8080:80 -v mydata:/usr/share/nginx/html --name web-volume nginx:latest
081c6eff177c189c585f011cb6c548a1d88bde76d0dcd4d2e3770074893d33c
PS C:\Users\Raj\Downloads\Assignment> █
```

Figure 5. Volume Attached to a Container

2.5 Container Networking

Multi-container applications depend on networking because the containers are required to communicate with each other or with the external system. Containers are initially in isolation, although container engines do allow the definition of networks that define how containers associate.

The most frequent solution is the user-defined bridge network, where the containers can communicate among themselves using expected naming conventions. On entering the same custom network, the containers are automatically provided with Domain Name System (DNS) resolution (Antonova et al., 2024). This can enable one container to reach another just with their names and with no need to have hard-coded IP addresses.

Host networking is also another networking mode whereby a container shares the network stack of the host system. This will be able to boost performance or make some form of networking easier, at the expense of isolation. Continuing to more complex scenarios, there are overlay networks, whereby containers of various physical hosts can flexibly communicate safely- it is common in distributed orchestration systems, including Docker Swarm or Kubernetes.

One of the simplest illustrations of container networking is to create a custom network, start a backend service in that network and subsequently start a second container which will interact with the backend. Since the network is shared by both containers, the second container can request the backend using the container name. This illustrates the fact that container networking aids in service discovery and microservice communication.

```
PS C:\Users\Raj\Downloads\Assignment> docker network create mynetwork
bfff572bc7d5bda78bc71700a8b05ae0069b0dfff885b7e489bbd8c24950f51b4
PS C:\Users\Raj\Downloads\Assignment>
```

Figure 6. Docker Network Creation

```
Run 'docker run --help' for more information
PS C:\Users\Raj\Downloads\Assignment> docker run -d --network mynetwork -p 8081:80 --name web1 rajgandhi06/docker-web-demo
f0abb233a907bcc51497133f7f6ca031168e8593289a47ff7d253920aa57032a
PS C:\Users\Raj\Downloads\Assignment>
```

Figure 7. Docker Network Run

```
PS C:\Users\Raj\Downloads\Assignment> docker run -d --network mynetwork --name db -- MYSQL_ROOT_PASSWORD=Password123 mysql:lates
t
Unable to find image 'mysql:latest' locally
latest: Pulling from library/mysql
f6a972aa365b: Pull complete
7a5e1e917526: Pull complete
55f85a7d691e: Pull complete
aff72f8f4e98: Pull complete
e4f376e797b1: Pull complete
98d18218e1bb: Pull complete
b843491434c1: Pull complete
ac5a3aa7003a: Pull complete
57bd38a2d740: Pull complete
ba8d858f8b56: Pull complete
Digest: sha256:fe036967257bf11aab7184e371920c5b629f0dd36cbefdf4ccd2ae18cc908dbd
Status: Downloaded newer image for mysql:latest
663a86e33fd65e76960963011d36d6340baa58037c0eee9c318a4f15aa0c640
PS C:\Users\Raj\Downloads\Assignment>
```

Figure 8. Docker Custom Network

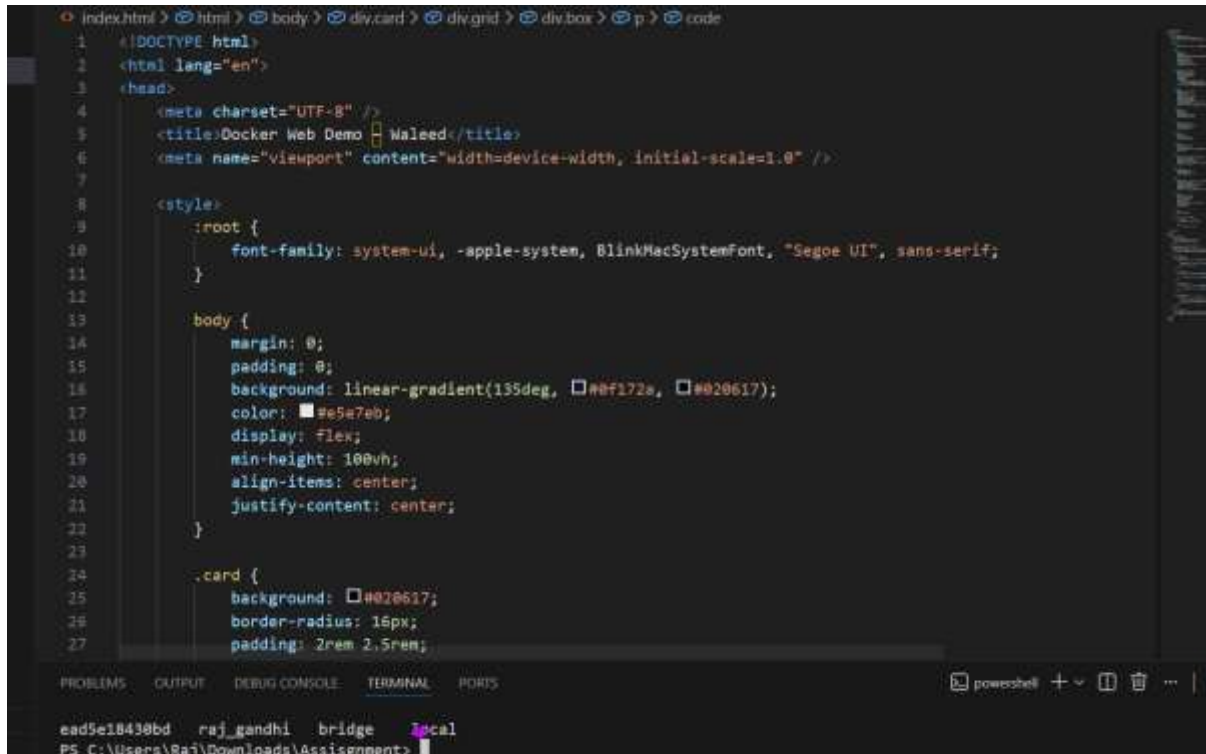
```
PS C:\Users\Raj\Downloads\Assignment> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
d7460d5d0dbb        bridge              bridge              local
44a67322085c        host                host                local
bfff572bc7d5b        mynetwork           bridge              local
134951122bbf        none                null                local
ead5e18430bd        raj_gandhi          bridge              local
PS C:\Users\Raj\Downloads\Assignment>
```

Figure 9. Docker Networks List

Example: Static Web Application Image

An example of a simple practical implementation will be having an image that is used to populate a web page using a simple web server. This is done by beginning with an effective base image, like the basic Linux system, which has a web service (Silva, Rafael and Fonte,

2024). The default web server content is lost with project-specific content, usually an HTML file in a project folder. The image opens the correct port of application to permit incoming web traffic, and then the container can be executed on any host of the compatible type. After creating the image, it may be deployed as a container, which subsequently repackages the HTML page by use of the built-in web server

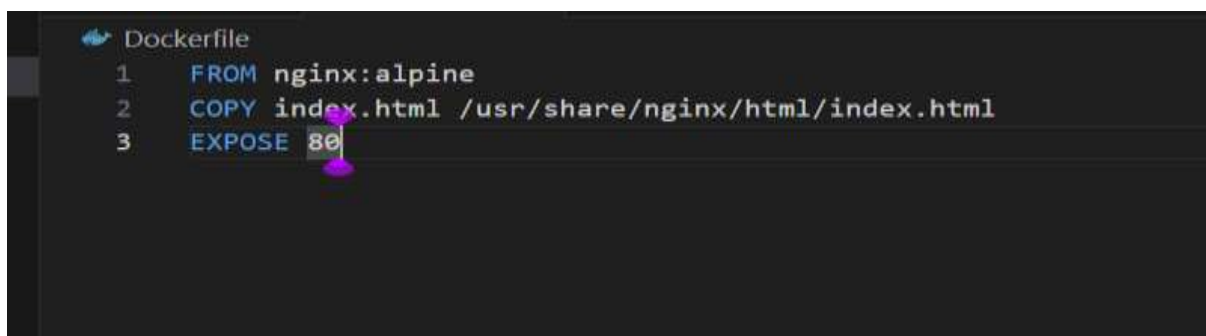


```
index.html > html > body > div.card > div.grid > div.box > p > code
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8" />
5   <title>Docker Web Demo Waleed</title>
6   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7
8   <style>
9     :root {
10       font-family: system-ui, -apple-system, BlinkMacSystemFont, "Segoe UI", sans-serif;
11     }
12
13     body {
14       margin: 0;
15       padding: 0;
16       background: linear-gradient(135deg, #f1f2f8, #e2e6e1);
17       color: #e5e7eb;
18       display: flex;
19       min-height: 100vh;
20       align-items: center;
21       justify-content: center;
22     }
23
24     .card {
25       background: #e2e6e1;
26       border-radius: 16px;
27       padding: 2rem 2.5rem;
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

ead5e18430bd raj_gandhi bridge local
PS C:\Users\Raj\Downloads\Assignment>

Figure 10. Docker index file



```
Dockerfile
1 FROM nginx:alpine
2 COPY index.html /usr/share/nginx/html/index.html
3 EXPOSE 80
```

Figure 11. Dockerfile

```

128
129 <hi>Running in a Docker Container</hi>

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
powerlshet + - - - - -

=> => sha256:abdece946203a31d98ef184559f417a33c3a8936a80153b2f0ffa208af4a0d48 954E / 954B 0.3s
=> => sha256:5d2cc344426d3d91200b457a771ecfe976de824e165506f5cce5d0b863da1ca9 629B / 629B 0.3s
=> => sha256:dfad290a5c259f8d1ec1938529f8ef02e335a26600497ad56d38e0727e1a10a 1.06MB / 1.06MB 0.7s
=> => sha256:014e56e61396e6f73cra0858124ca5fbc601d7888099968a4eaa09f31dcd71a53 3.86MB / 3.86MB 2.0s
=> => extracting sha256:814e56e61396e6f73cra0858124ca5fbc601d7888099968a4eaa09f31dcd71a53 0.3s
=> => extracting sha256:dfad290a5c259f8d1ec1938529f8ef02e335a26600497ad56d38e0727e1a10a 0.3s
=> => extracting sha256:5d2cc344426d3d91200b457a771ecfe976de824e165506f5cce5d0b863da1ca9 0.0s
=> => extracting sha256:abdece946203a31d98ef184559f417a33c3a8936a80153b2f0ffa208af4a0d48 0.0s
=> => extracting sha256:51c30493937c33bd8b568d8ead09d9556f558d88077b85a5a185516abe85a1f 8.0s
=> => extracting sha256:ad5b65da02cfbd43dae87443b87051f3816a10eb771993838cb9a96ee828d471 0.1s
=> => extracting sha256:fc13532509d72b70e7dd276ee52f2743b14326b83c13935c86d7477c00019dee 0.0s
=> => extracting sha256:136bc6976c2023e3363e66b88167d008819faca9e756c161c58754e3819bf4863 0.7s
=> [2/2] COPY index.html /usr/share/nginx/html/index.html 0.2s
=> => exporting to image 0.7s
=> => exporting layers 0.2s
=> => exporting manifest sha256:2257faaa6a06131d5a71db7d1c92dabcba615b0b445f1ff92d9866efa8393c3a 0.0s
=> => exporting config sha256:913aec7d88471cdcc38fe863dfde5821a6696953be92c9c7d73ae7ea500e565c 0.0s
=> => exporting attestation manifest sha256:034ff745fd055af8960eccdd6953652935aeac123ccf89ac54fb6ac736d7f997 0.1s
=> => exporting manifest list sha256:a5e2264fb387908fcd38cf63c50252b2bc7c72c26ce7542a48e4de393af4f8d0 0.0s
=> => naming to docker.io/rajgandhi06/docker-web-demo:latest 0.0s
=> => unpacking to docker.io/rajgandhi06/docker-web-demo:latest 0.1s

PS C:\Users\Raj\Downloads\Assignment>

```

Figure 12. Docker Image Build

```

=> => exporting manifest sha256:2257faaa6a06131d5a71db7d1c92dabcba615b0b445f1ff92d9866efa8393c3a
=> => exporting config sha256:913aec7d88471cdcc38fe863dfde5821a6696953be92c9c7d73ae7ea500e565c
=> => exporting attestation manifest sha256:034ff745fd055af8960eccdd6953652935aeac123ccf89ac54fb6ac736d7f997
=> => exporting manifest list sha256:a5e2264fb387908fcd38cf63c50252b2bc7c72c26ce7542a48e4de393af4f8d0
=> => naming to docker.io/rajgandhi06/docker-web-demo:latest
=> => unpacking to docker.io/rajgandhi06/docker-web-demo:latest

PS C:\Users\Raj\Downloads\Assignment> docker push rajgandhi06/docker-web-demo
Using default tag: latest
The push refers to repository [docker.io/rajgandhi06/docker-web-demo]
dfad290a5c25: Pushed
5d2cc344426d: Pushed
136bc6976c20: Pushed
54978284da34: Pushed
abdece946203: Pushed
ad5b65da02cf: Pushed
51c30493937c: Pushed
fc13532509d7: Pushed
04b1a564c80a: Pushed
014e56e61396: Pushed
latest: digest: sha256:a5e2264fb387908fcd38cf63c50252b2bc7c72c26ce7542a48e4de393af4f8d0 size: 856
PS C:\Users\Raj\Downloads\Assignment>

```

Figure 13. Docker Image Push

```

PS C:\Users\Raj\Downloads\Assignment> docker run -d -p 8082:80 --name web3 rajgandhi06/docker-web-demo
3a5ad2c7a8d78fd5ef0b9b08ba35bae7df1825b4d301b1f71e359f5b28fbd1e0
PS C:\Users\Raj\Downloads\Assignment>

```

Figure 14. Docker Container Run

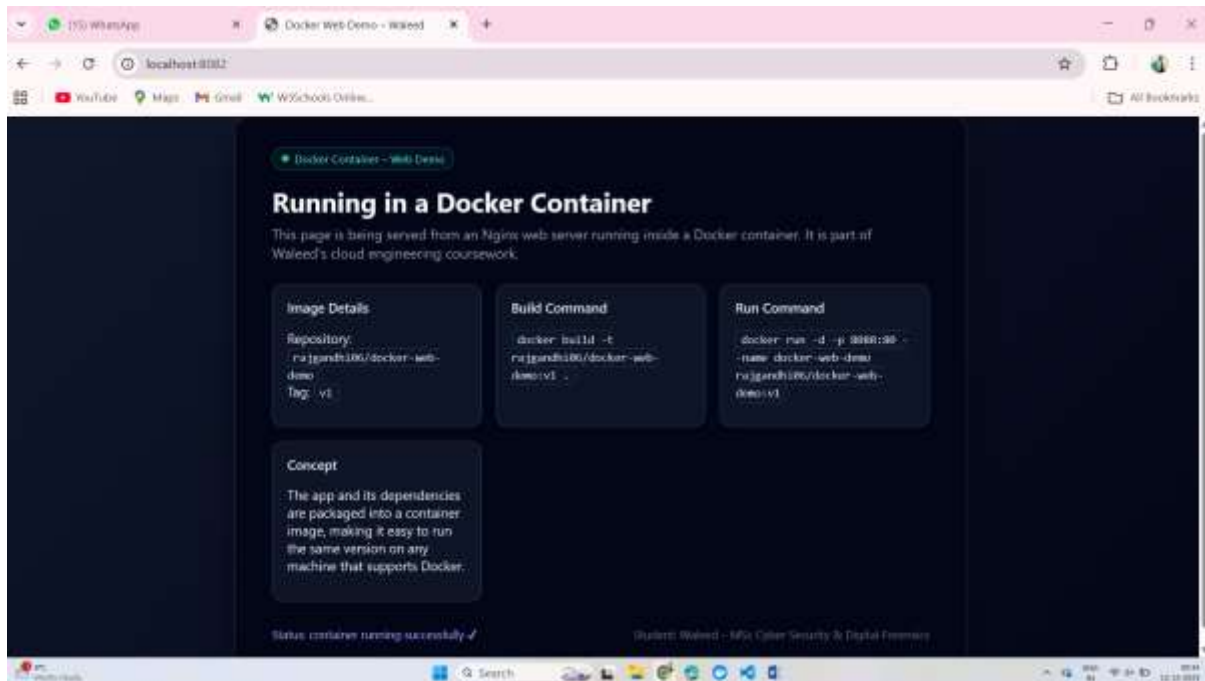


Figure 15. Docker Web Demo

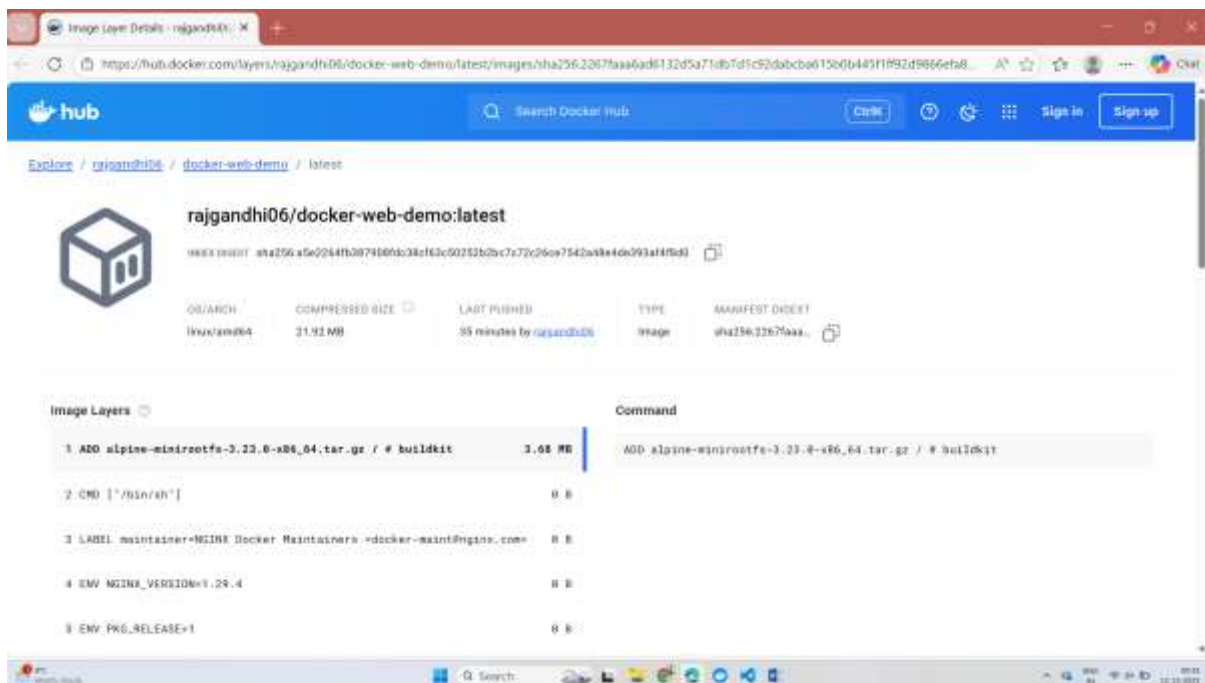


Figure 16: Docker Hub

2.6 Full End-to-End Workflow Example

A full show of container technology is an integration of various concepts to explain the entire lifecycle of developing, operating, and distributing a containerised application.

The creation of a project directory with files of the application starts a typical workflow. A configuration file is then written, giving the specifications of how the application

environment would be built (Hoel, Myrhaug and Svelland, 2024). The base system is defined in this file, along with the application data or dependencies, and the necessary network ports are brought out.

When it is configured, the image is assembled. The container engine reads the configuration file, constructs the layered filesystem, and builds an image that cannot be modified. That image is a mobile artefact which could be executed on clouds, placed in registries or shared on an inter-team basis.

Once the image is created, more resources like networks and volumes can be created. Custom network enables containers to communicate with each other by name, and a volume is used to ensure dynamic or persistent data is not lost when containers are restarted.

Launching is then done with the image made beforehand. The web server is contained in the running container, and it is used to serve the HTML page within the project, and it is used to reply to the received requests via the mapped port. This only shows how the application is running in a completely isolated environment.

A command-line request tool or a web browser accesses a webpage served on the host system to ensure that the software works. When one shows the correct content, then the containerised application is running smoothly (Bhardwaj and Krishna, 2021).

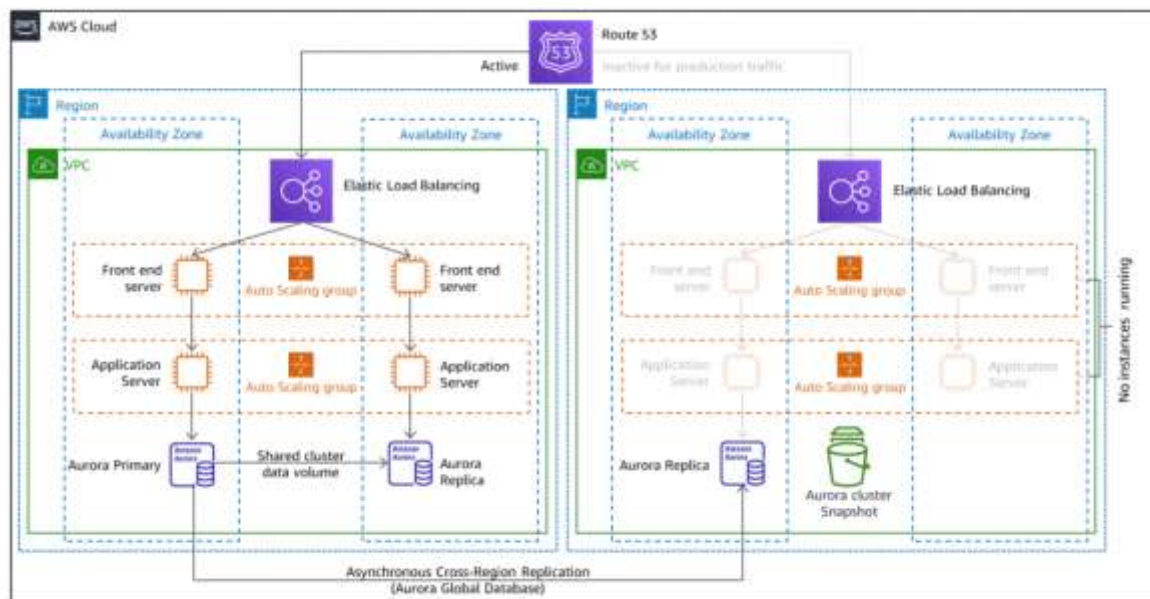
Lastly, the picture is posted to an off-site registry. Saving the image would enable universal accessibility to cloud deployment. Stored in the registry, models Orchestration systems (e.g. Amazon ECS, Kubernetes, or Azure Container Instances) can later access it and deploy the image in scalable cloud environments. This makes it complete development to cloud-ready distribution.

2.7 Video Link

<https://youtu.be/8Bb5xcotSKg>

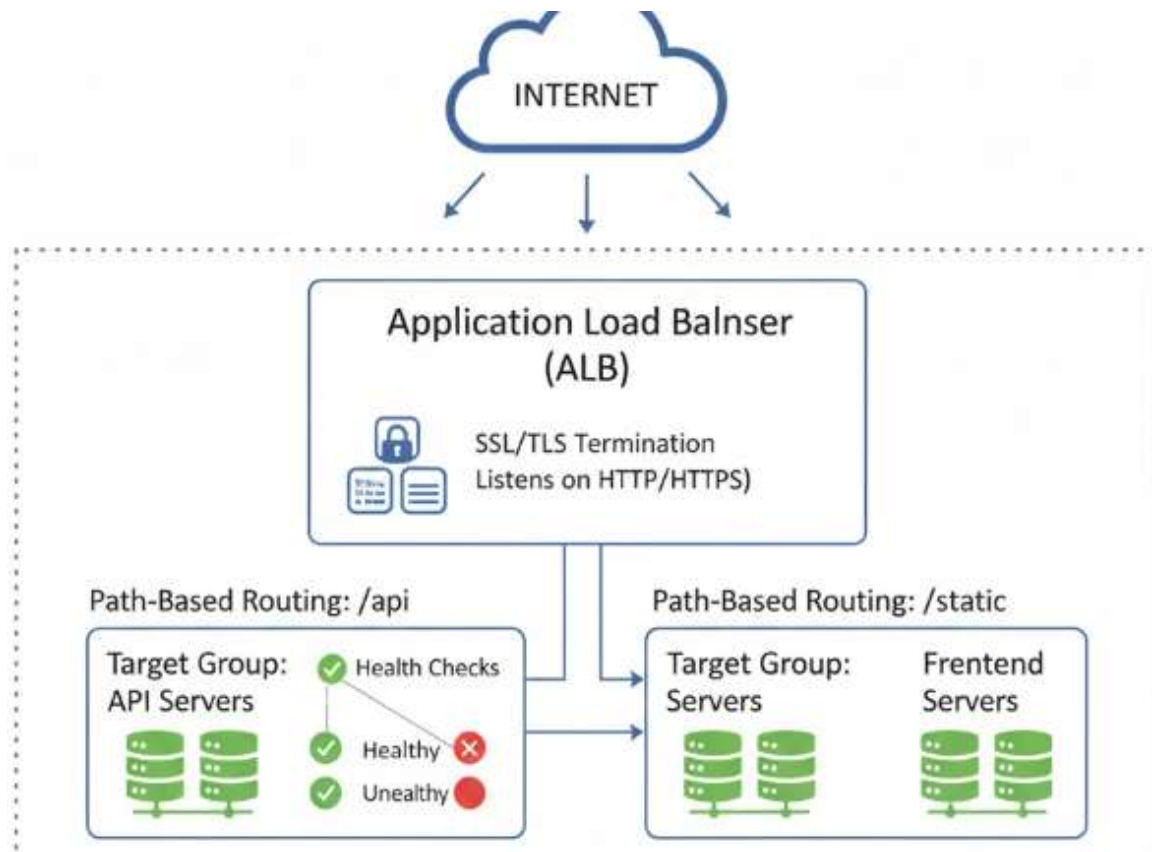
3. Cloud Computing Concepts

3.1 Load Balancing, Scalability, and Application Gateways



Contemporary cloud applications are based on three core principles, which include load balancing, scalability, and application gateways to provide scale, performance, and security. The mentioned components collaborate with each other to enable applications that are horizontally distributed, including a web platform providing both static and dynamically generated content as well as API endpoints. In this case, the application will be composed of a set of Web servers, which operate simultaneously and can service user requests, and cloud-native solutions provide a means of intelligent traffic distribution, dynamically changing capacity, and maintaining a secure entry point.

Load Balancing in Cloud Environments:



Load balancing will help guarantee that there is an effective distribution of traffic entering the application instances. This eliminates the possibility of overloading a single server, hence enhancing the stability, responsiveness and fault tolerance (Arogundade and Palla, 2023). Cloud providers have managed load-balancing services that can be used to sustain large amounts of traffic and do not require much configuration effort.

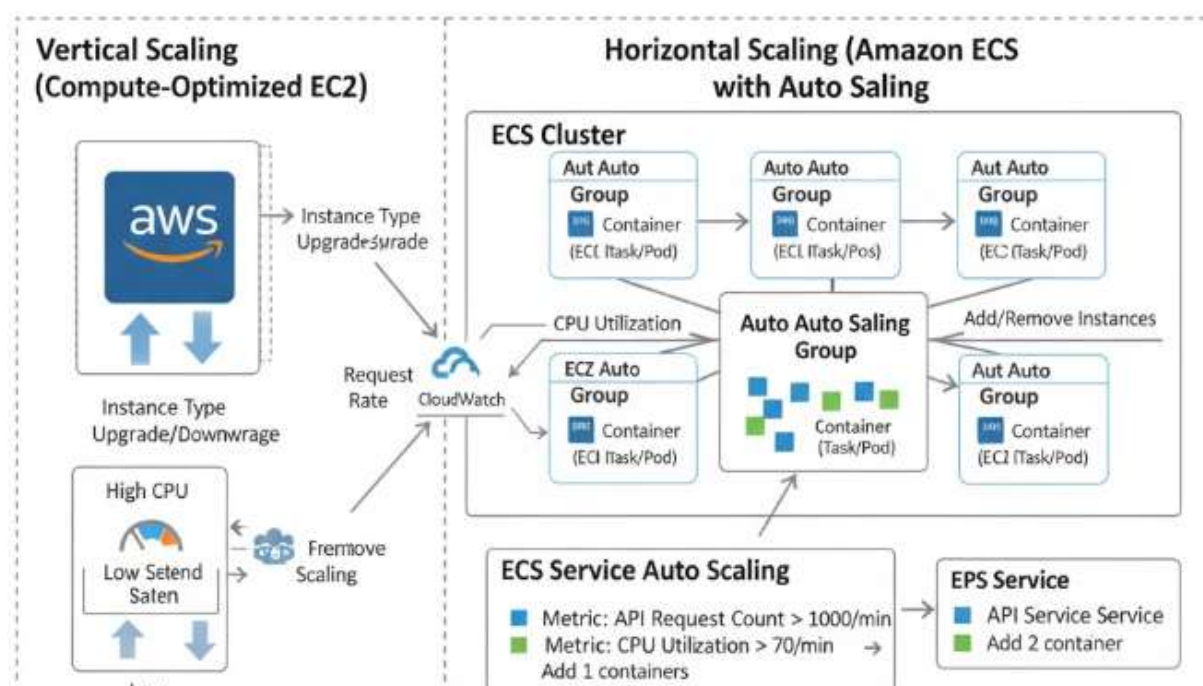
In the context of Amazon Web Services (AWS), there are three types of load balancers provided within the family of Elastic Load Balancing (ELB), of which the most applicable is the Application Load Balancer (ALB). The ALB works at Layer 7 on the OSI model, and it is optimised to support HTTP and HTTPS traffic. It supports the advanced features like host-based routing and path-based routing, which enable various services to be redirected to various backend targets based on the structure of a request. As an illustration, the traffic to the /api route can be sent to the group of API servers, whereas requests to the static pages can be sent to another group of frontend servers. The ALB also handles the termination of web

resources (SSL/TLS), which makes it easier to handle certificates, and it also shifts the heavy cryptographic workload of the backend server.

Health checks are another necessary operation. The ALB constantly evaluates the health of the backend instances and automatically excludes the unhealthy ones from the rotation of instances in the traffic (Cherukuri, 2024). This enables the application to resume even when single instances are having failures. Also, stickies, WebSocket and automatic scaling integration of the ALB make it fit dynamically and distributed web applications.

Scaling of Cloud Applications:

Amazon Web Services (AWS Cloud)

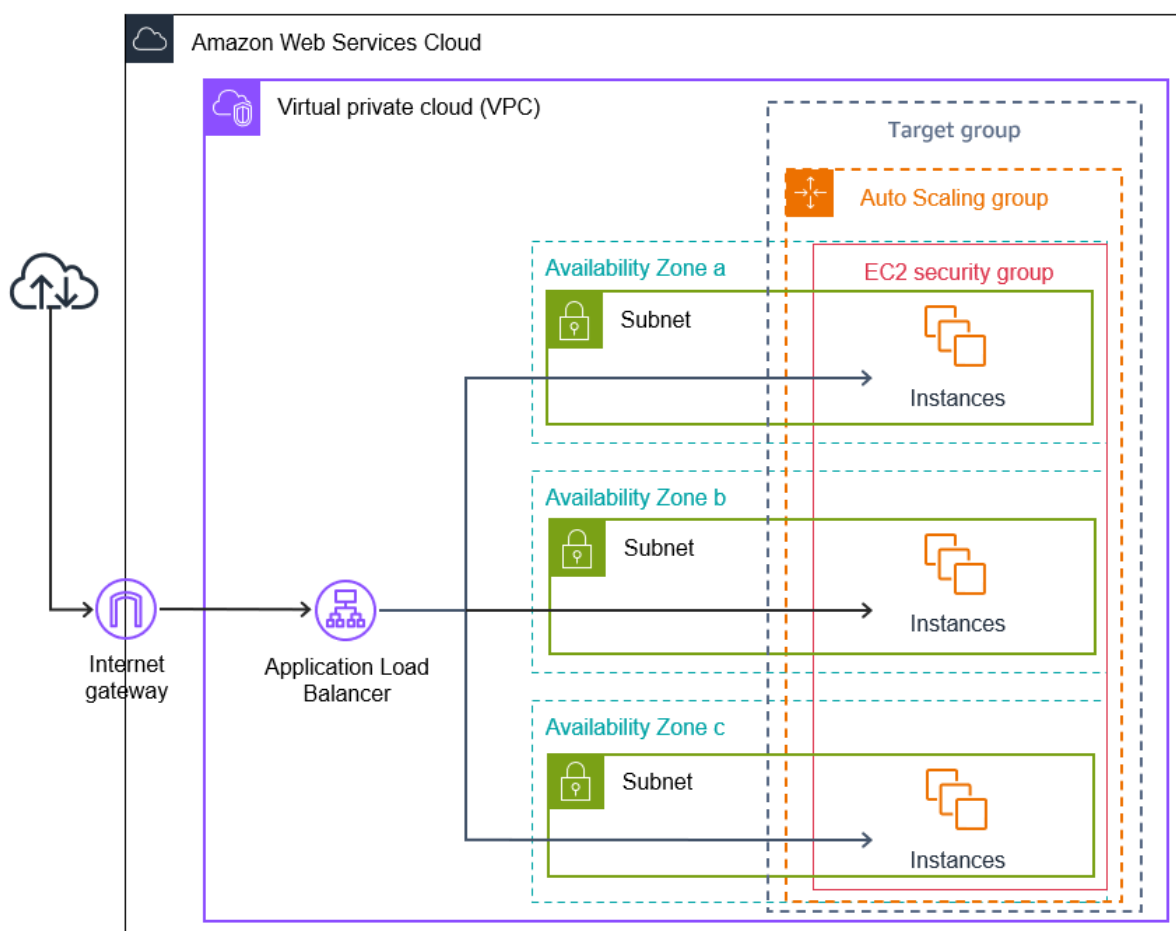


Scalability is the capacity of a system to scale its capacity depending on fluctuating demand. Scalability in cloud environments is usually of two types: vertical scaling, which is the expansion of resources of one instance, and horizontal scaling, which refers to the addition or removal of instances. Cloud native systems have more frequent horizontal scalable architectures due to superior fault tolerance and reduced geographical distribution of architectural components.

AWS offers scalability machines like Auto Scaling Groups (ASG) of virtual machines and scalability of features of container orchestrators, including Elastic Container Service (ECS) and Elastic Kubernetes Service (EKS). To achieve this, Auto Scaling Groups check its key indicators, e.g. CPU utilisation, request rate, or latency, and use these signals to add instances

of the application automatically or to remove them. This will keep the system with the right capacity at any given time to satisfy the demand of the user (Ajasa, Chizari and Alam, 2025). These principles are larger within containerised environments. ECS and EKS offer service-level auto-scaling which automatically changes containers or pods into the running state depending on the appropriate metrics or application-defined rules. This will allow the scaling of services at a granular level, so individual segments of an application can be scaled. Indicatively, the API service that has peaks in the request volume can be scaled in minutes without the need to have an impact on the quantity of frontend containers. Scalability allows for preventing bottlenecks during periods when demand is high, and at the same time, it allows being cost-efficient when demand is low since it can be scaled down then. Scaling, when applied in the case of a horizontally scalable web application, is what ensures that the application's scale is adequate enough to satisfy user demand without compromising the responsiveness and reliability of the application.

Application Gateways:



An application gateway is a clever and secure point of entry to traffic of clients. It usually includes functionality like request routing, authentication, content filtering, threat protection,

caching and traffic shaping. Application gateways are essential in current architectures, especially where the applications provide multiple endpoints to APIs, or they need edge security (Alam et al., 2023).

In AWS, API Gateway is a managed service that provides an expose to the HTTP endpoints and orchestrates with the back-up services, such as Lambda functions, containerised applications, and old-fashioned servers. The access control is done by API Gateway through integrations with identity providers like AWS Cognito or IAM permissions. It also uses throttling policies, which prevent the misuse or overuse of the backend services. The request transformation feature enables the gateway to alter the headers, parameters or payloads and forward them to downstream systems in order to achieve compatibility between the client and the backup services.

Other websites provide similar services. Azure offers Application Gateway and Front Door, both of which are based on Layer 7 routing, Web Application Firewall (WAF), as well as global load balancing. The functionality that Google Cloud provides is similar in the form of Cloud Endpoints and Cloud Armour. Application gateways are built to consolidate and protect the points of application entry and remove the burden of the back-end components, regardless of the provider.

Application gateways are usually utilised in the event of controlling large-scale APIs, performing consistent verification, securing services against assault, or directing demands to various microservices according to course or domain (Divya and Harshitha, 2024). Gateways are the initial level of a carefully designed, multi-layered architecture used together with load balancers and scaling systems.

3.2 Complementarity of Load Balancing and Scaling

Both load balancing and scaling are related to cloud architecture, although they work together to deliver resilience and performance. Despite the different functions that they execute, they require each other to exist.

Load balancing is used to make sure that some traffic is evenly distributed among a pool of instances. In the absence of load balancing, new or existing instances would be inefficient, especially when the demand is high. To take an example, when there are ten such there are ten instances, but the traffic can only reach one or two, then the skewed load can form bottlenecks, unnecessary latency, or even result in the crash of the application. This issue is removed by a load balancer, which redirects traffic to various functional instances.

Scaling, on the other hand, guarantees that traffic can be distributed to enough instances that the load balancer will have. At some point of a sudden rush in demand, the given instances might not be adequate to support the load. This is automatically detected by autoscaling mechanisms and is automatically provided with new instances (Rehaimi et al., 2024). After becoming healthy, the new instances are registered with the load balancer, which, in turn, routes the traffic to them with no need for human intervention.

The complementation stands out particularly in times of variation in workloads. Scaling is used to add capacity when it is needed, and load balancing makes sure that the new capacity is used effectively. Scaling down of instances will save cost during quiet periods by decreasing the number of active instances, and the load balancer will change the distribution of traffic. They offer a dynamic, self-adjusting architecture that is able to sustain performance, improve reliability, and lower the cost of operation.

3.3 Benefits of Combining Load Balancing, Scaling, and Application Gateways

A combination of all three of these components, load balancing, scalability, and application gateways, creates a complete cloud architecture, which can serve secure, responsive, and resilient services. An example of one such situation, which demonstrates why this combination is crucial, is the presence of an e-commerce platform that has consistent surges when there are promotions or sales by flash.

The application gateway will serve as the gateway in this case. It has essential roles of terminating TLS servers, routing of requests depending on the path of the URL, blocking traffic containing malicious data with the help of a Web Application Firewall, and enforcing rules on throttling of abusive traffic (Beena et al., 2025). The gateway ensures that the infrastructure supporting the backend is not burdened with unnecessary loads or attacked at all by taking these responsibilities to the edge.

The load balancer is fed with the filtered and correctly directed traffic of the gateway and distributes it across the existing application servers. This makes sure that any individual server does not become overloaded, and also lowers latency, and guarantees high availability. When the servers stop responding, the load balancer redirects the traffic to avoid any disruption in the services.

Scalability mechanisms check in the system when demand is increasing. When it comes to a flash sale, the activity of customers might increase exponentially. Auto-scaling monitors the increase in CPU, requests, or response times and creates more instances of an application to handle the surged workload. These new cases are automatically registered by the load

balancer, but this enables the system to sustain the surge demand without the system slowing down.

The demand usually decreases after the event is over. Auto-scaling then downscales the number of operating instances to enable the organisation to save avoidable expenses that would have been incurred because of idle resources (Melo et al., 2025).

The cumulative advantages to the customer are:

- **Improved security:** Application gateways offer WAF protection, enforcement of authentication, as well as traffic filtering, which protects the application against threats.
- **High resilience:** Scaling mechanisms, as well as load balancers, are combined to eliminate outages, make up failures, and ensure that the application stays operational even in the case of infrastructure disruption.
- **Cost efficiency:** Auto-scaling makes sure that only the necessary number of resources is expanded and the unnecessary resources are shrunk accordingly.
- **Enhanced user experience:** Even when there is heavy traffic, visitors are offered the same performance, good response time and service interruption.

These integrated cloud services allow organisations to provide stable and quality services at a high cost-security-performance optimisation.

4. Main Driving Forces Leading to the Expansion of Cloud and Serverless vs Containerisation

4.1 Driving Forces for Cloud Expansion

The increase of cloud computing has been so high because it possesses economic, operational and strategic advantages that respond to the current organisational requirements. The main benefits are the reduction of costs, as in the case of conventional on-premise infrastructure, very large amounts of capital are required to procure hardware and maintain it, whereas cloud services are a pay-as-you-use model (Rehan, 2024). This enables organisations to incur costs on the resources utilised only, while enjoying the economies of scale enjoyed by global cloud data centres.

Scalability and elasticity are also essential; the ability of the cloud to dynamically provision resources driven by real-time requirements eliminates over-provisioning and towards performance at peak times. This is flexible, particularly for an organisation that has fluctuating traffic patterns.

Cloud computing also increases nimbleness and accelerates the development cycles. Quick deployment of resources, including servers, databases and analytics services, enhances innovation and allows teams to experiment and launch features quickly, making it easy to transform digitally.

Another notable benefit is operational simplicity since the cloud operators handle infrastructure processes, like patching and security measures, hence the technical departments do not have to focus on maintenance but on building applications.

Moreover, cloud services have a crucial impact on the global access and reliability, which is ensured by several regional data centres that minimise latency and enhance the resilience of the applications by means of multi-region replication and redundancy (Sapkal and Kusi, 2024).

Finally, cloud platforms are also compatible with DevOps and continuous integration/continuous deployment (CI/CD) practices, which justify automated pipelines and infrastructure-as-code tools, which are critical towards ensuring high release speed and quality in development.

4.2 Serverless Functions (Functions as a Service)

Serverless computing is one of the changes to application deployment and execution. Within this model, functions are operated as a reaction to events like HTTP requests, some messages on a queue or scheduled events. There are also AWS Lambda, Azure Functions and Google Cloud Functions. The key characteristic of serverless functions is that the cloud vendor takes care of the entire infrastructure.

Serverless functions are relatively short-lived and stateless, i.e. they are designed to provide explicit tasks in a short period of time without any long-term state. The necessary persistence should be stored in other services like the database or object storage. This architecture promotes loose, open architectures.

Serverless functions exhibit one of their key effectiveness attributes in the form of a lack of server management. The application developers just provide the logic behind the functions, and the cloud platform takes control of the runtime provisioning, scaling and execution (Taherdoost, 2025). In serverless systems, scaling is provided by default because the system will run multiple processes in parallel when it handles many events at the same time. This renders the model efficient, especially for workloads that have unpredictable demand.

The billing system is also incidence-based. Serverless is very cost-effective when it comes to intermittent workloads, as organisations pay on a per-function invocation basis and also for

the duration in which a worker runs. Serverless functions can, however, have cold-start performance delays when initialised following idle time, which can be an implementation factor in applications with latency sensitivity.

4.3 Compare & Contrast Serverless vs Containerisation

Aspect	Serverless (Lambda/Functions)	Containerisation (Docker + orchestrator)
Operational overhead	Very low; provider manages runtime	More control; you manage container images, runtime, orchestration
Startup time	Potential cold-start latency	Fast (containers start quickly), often faster than cold serverless setups
Billing model	Per-invocation (often very cost-efficient for sporadic workloads)	Per-provisioned resource (VM/cluster); more cost for always-on resources
Use cases	Event handlers, simple APIs, ETL, cron jobs	Long-running services, complex apps, stateful services (with volumes)
Control & customization	Limited runtimes and environment control	Full control over environment, libraries, OS packages
Scaling model	Seamless per-request scaling (managed)	Orchestrator scales services; needs configuration (but more flexible)
Portability	Portable code but may tie to specific provider triggers/APIs	Highly portable images; can run anywhere Docker/K8s supported

4.4 Influence of Driving Forces on Deployment Choice

The motivators for the use of the clouds affect the adoption of serverless or container-based clouds. Serverless computing can be the most cost-effective choice for workloads that are both cost-sensitive and unpredictable, and event-driven due to granular billing and automatic scaling. It is also capable of lowering the time to market of simple services, in which case no infrastructure configuration is required (Katal, Dahiya and Choudhury, 2023).

In contrast, applications with prolonged execution circumstances, customised operating environments, foreseeable performance, or use of an enduring storage are more adaptable to containerisation. The platforms used to orchestrate containers allow a stable deployment, granular control of the deployment, and high-scale system-supporting microservice stacks. The issues of flexibility and lock-in to vendors are also of significance. Containerised workloads can be deployed to any cloud service or on-prem on any environment and are more portable. Serverless has an efficient approach but tends to couple well with a selected cloud ecosystem.

The combination of these driving forces influences the architectural decisions so that organisations will select the version that fits most effectively the performance requirements, cost and strategic objectives (Scrivano, 2025).

5. Conclusion

Cloud computing still keeps evolving the way contemporary application development is carried out, as this offers an opportunity to deploy any type of services in a scalable environment with efficiency and resilience. The technologies discussed in this report, especially Docker and containerisation, have the effect of enabling lightweight, portable and consistent application environments to be very useful in enhancing the deployment processes in the development, testing, and production initiatives. The practical Grounding The concepts of virtualisation, container networking, volumes, and image distribution demonstrate the real basis of cloud-native architectures. These technologies are shared with the core cloud computing principles such as load balancing, scalability and application gateways that show how cloud providers sustain performance, security and reliability to applications with fluctuating demand. The analysis of the similarities and distinctions between serverless functions and containerisation shows that the two technologies respond to major goals of cloud deployment, including the decrease of operational overhead, cost-effectiveness, and greater agility. The two models have their own advantages based on the nature of the workload, and serverless is best in event-driven types, with containers offering the aspects of control and flexibility necessary in complex and long-running applications. These technologies and architectural solutions, as a whole, exhibit how cloud platforms can help organisations to provide high-quality services, optimise the use of resources, and reduce resource utilisation costs. The further development of cloud-native tools and practices will be an important part of the future of scalable, distributed computing.

References

- [1] S. Agrawal and D. Singh, “Study containerization technologies like Docker and Kubernetes and their role in modern cloud deployments,” in *Proc. IEEE 9th Int. Conf. Convergence in Technology (I2CT)*, Apr. 2024, pp. 1–5.
- [2] P. Kaur, J. K. Josan, and N. Neeru, “Performance analysis of Docker containerization and virtualization,” in *Proc. Third Int. Conf. Communication, Computing and Electronics Systems (ICCCES 2021)*. Singapore: Springer, Mar. 2022, pp. 863–877.
- [3] D. Baloni, C. Bhatt, S. Kumar, P. Patel, and T. Singh, “The evolution of virtualization and cloud computing in the modern computer era,” in *Proc. Int. Conf. Communication, Security and Artificial Intelligence (ICCSAI)*, Nov. 2023, pp. 625–630.
- [4] N. Brooks, C. Vance, and D. Ames, “Cloud computing: A review of evolution, challenges, and emerging trends,” *J. Computer Science and Software Applications*, vol. 5, no. 4, 2025.
- [5] 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.
- [6] C. Raftopoulos, “Cloud security with Docker and Kubernetes,” M.S. thesis, Univ. of Piraeus, Piraeus, Greece, 2025.
- [7] S. P. Bandaru, *Cloud Computing for Software Engineers: Building Serverless Applications*, 2023.
- [8] V. M. Antonova, M. A. Egorov, V. P. Blinov, E. E. Malikova, and A. Y. Malikov, “Studying the principles of infocommunication network virtualisation using the Docker platform,” in *Proc. Systems of Signals Generating and Processing in the Field of On-Board Communications*, Mar. 2024, pp. 1–5.
- [9] B. D. Hoel, E. A. Myrhaug, and L. N. Svelland, “Cloud computing: A comparative study,” B.S. thesis, Norwegian Univ. of Science and Technology (NTNU), Trondheim, Norway, 2024.
- [10] A. Bhardwaj and C. R. Krishna, “Virtualization in cloud computing: Moving from hypervisor to containerization—A survey,” *Arabian J. Science and Engineering*, vol. 46, no. 9, pp. 8585–8601, 2021.
- [11] O. R. Arogundade and K. Palla, “Virtualization revolution: Transforming cloud computing with scalability and agility,” *Int. Adv. Res. J. Sci. Eng. Technol. (IARJSET)*, 2023.
- [12] B. R. Cherukuri, “Containerization in cloud computing: Comparing Docker and Kubernetes for scalable web applications,” *Int. J. Scientific Research and Application*, vol. 13, no. 1, pp. 3302–3315, 2024.

- [13] A. D. Ajasa, H. Chizari, and A. Alam, "Database security and performance: A case of SQL injection attacks using Docker-based virtualisation and its effect on performance," *Future Internet*, vol. 17, no. 4, p. 156, 2025.
- [14] M. Alam, S. Mustajab, M. Shahid, and F. Ahmad, "Cloud computing: Architecture, vision, challenges, opportunities, and emerging trends," in *Proc. Int. Conf. Computing, Communication, and Intelligent Systems (ICCCIS)*, Nov. 2023, pp. 829–834.
- [15] C. D. Divya and K. Harshitha, "Tools and open source platforms for cloud computing," in *Advanced Computing Techniques for Optimization in Cloud*. Boca Raton, FL, USA: Chapman and Hall/CRC, 2024, pp. 226–244.
- [16] A. Rehaimi, Y. Sadqi, Y. Maleh, G. S. Gaba, and A. Gurtoev, "Towards a federated and hybrid cloud computing environment for sustainable and effective provisioning of cyber security virtual laboratories," *Expert Systems with Applications*, vol. 252, p. 124267, 2024.
- [17] B. M. Beena *et al.*, "Adaptive energy optimization in cloud computing through containerization," *IEEE Access*, 2025.
- [18] P. Melo *et al.*, "Performance evaluation of containerized virtualization platforms," *SN Computer Science*, vol. 6, no. 7, p. 771, 2025.
- [19] H. Rehan, "Revolutionizing America's cloud computing: The pivotal role of AI in driving innovation and security," *J. Artificial Intelligence General Science (JAIGS)*, vol. 2, no. 1, pp. 239–240, 2024.
- [20] A. Sapkal and S. S. Kusi, "Evolution of cloud computing: Milestones, innovations, and adoption trends," 2024.
- [21] H. Taherdoost, "Cloud computing essentials," in *Innovating and Integrating Emerging Technologies*. Cham, Switzerland: Springer Nature, 2025, pp. 41–54.
- [22] A. Katal, S. Dahiya, and T. Choudhury, "Energy efficiency in cloud computing data centers: A survey on software technologies," *Cluster Computing*, vol. 26, no. 3, pp. 1845–1875, 2023.
- [23] A. Scrivano, "Innovative approaches in cloud computing: Balancing efficiency, scalability, and sustainability," *Authorea Preprints*, 2025..