



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

Programme: MSc Cybersecurity and Digital Forensics

Module: Cloud Engineering Principles and Practice

Module code: ACCA7027

**Title: Cloud Computing, Containerisation and Serverless
Technologies: A Critical Evaluation of Docker and Cloud-Native
Service Delivery**

Student Name: Poshia Janvi Vivekkumar

Student ID: 2543705

Table of Contents

1. Introduction.....	6
1.1 Background to Cloud Computing.....	6
1.2 Aim of the Report.....	6
1.3 Objectives.....	6
1.4 Report Structure.....	6
2. Cloud Technology: Docker and Practical Examples.....	7
2.1 Principles of Virtualisation.....	7
2.1.1 Definition and Concept of Virtualisation.....	7
2.1.2 Virtualisation of Shared Hardware.....	7
2.1.3 Type 1 and Type 2 Hypervisors.....	9
2.2 Containerisation.....	10
2.2.1 Principles of Containerisation.....	10
2.2.2 Docker as a Containerisation Platform.....	10
2.2.3 Containers versus Virtual Machines.....	11
2.3 Docker file and Building a Docker Image.....	13
2.3.1 Docker file.....	13
2.3.2 Creating the HTML Web Application.....	13
2.3.3 Building the Docker Image.....	14
2.3.4 Running the Container.....	15
2.4 Docker Images and Publishing to Docker Hub.....	15
2.4.1 Docker Image Lifecycle.....	15
2.4.2 Tagging the Image.....	16
2.4.3 Publishing to Docker Hub.....	17
2.5 Container Volumes.....	18
2.5.1 Purpose of Docker Volumes.....	18
2.5.2 Types of Docker Storage.....	18
2.5.3 Practical Docker Volume.....	19
2.6 Container Networking.....	20
2.6.1 Docker Networking Principles.....	20
2.6.2 Docker Network Types.....	21
2.6.3 Creating a Docker Network.....	21
2.7 Integrated Docker Demonstration.....	23
3. Cloud Computing Concepts.....	23
3.1 Load Balancing.....	23
3.1.1 Definition.....	23
3.1.2 Benefits.....	23
3.1.3 Example: AWS Elastic Load Balancer.....	24
3.2 Scalability.....	25
3.2.1 Horizontal Scaling.....	25
3.2.2 Vertical Scaling.....	26
3.2.3 AWS Auto Scaling.....	26
3.3 Application Gateway.....	27
3.4 Importance of Load Balancing and Scaling.....	29
3.5 Combining Load Balancing, Scalability and Application Gateway.....	31
4. Main Driving Forces Behind the Expansion of Cloud Computing.....	32
4.1 Cost Efficiency.....	32
4.2 Flexibility and Agility.....	33
4.3 Scalability and Elasticity.....	33
4.4 Availability and Global Reach.....	33

4.5 DevOps and Continuous Delivery.....	33
4.6 Serverless Computing.....	34
4.7 AWS Lambda and Azure Functions.....	35
4.8 Serverless versus Containerisation.....	37
4.9 Critical Evaluation of Cloud Adoption Drivers.....	40
5. Video Demonstration.....	40
6. Discussion and Critical Evaluation.....	48
7. Conclusion and Recommendation.....	51
References.....	52

List of Figures

Figure 1: Diagram of Virtualisation	9
Figure 2: Test Diagram of Virtualisation Coding	10
Figure 3: Type 1 versus Type 2 Hypervisor Architecture	11
Figure 4: Docker Architecture	12
Figure 5: Virtual Machines versus Docker Containers	13
Figure 6: HTML Page/Application Created for the Docker Demonstration	15
Figure 7: Docker Image Build Process	15
Figure 8: Running Docker Container	16
Figure 9: Docker Image Lifecycle	17
Figure 10: Docker Hub Repository/Published Image	18
Figure 11: Docker Volume Architecture	19
Figure 12: Docker Volume Practical Demonstration	20
Figure 12: Practical Docker Volume Diagram	21
Figure 13: Docker Networking practical Diagram	21
Figure 14: Docker Network Creation	22
Figure 15: Container Network Communication	23
Figure 16: Cloud Load Balancing Architecture	25
Figure 17: AWS ALB Diagram	26
Figure 18: Cloud Auto Scaling Architecture	28
Figure 19: Application Gateway / Layer 7	29
Figure 20: Application Gateway / Layer 7 Routing	30
Figure 21: Load Balancing + Auto Scaling	31
Figure 22: Flow Diagram of Load Balancing, Scalability and Application Gateway	32
Figure 23: Integrated Cloud Application Architecture	33
Figure 24: Diagram of Serverless Computing	35
Figure 25: Serverless Function Architecture	36
Figure 26: AWS Lambda / Azure Functions Comparison	37
Figure 27: Containers versus Serverless	40
Figure 28: Cloud Technology Coding	51

List of Tables

Table 1: Comparison summary of Type 1 and Type 2 Hypervisors	10
Table 2: comparison of Containers versus Virtual Machines	13
Table 3: AWS Lambda vs. Azure Functions	37
Table 4: Containers vs. Serverless	38

1. Introduction

1.1 Background to Cloud Computing

Cloud computing has become a major approach to delivering computing resources, applications, storage, and networking through the help of scalable infrastructures. Its growth is supported through the help of virtualisation, which allows physical hardware to be shared among multiple virtual machines, along with containerisation, which presents both lightweight as well as portable environments for applications. The scale of adoption of this advanced technology is showcased by Gartner, which revealed that the worldwide public cloud end-user spending to maximise from 595.7 billion in the FY 2024 to \$723.4 billion in FY 2025, which representing 21.5% growth [1].

1.2 Aim of the Report

The aim of this report is to critically examine the visualisation, Docker containerisation and finally the cloud-native application deployment, incorporating their significance for effective and scalable cloud services.

1.3 Objectives

- To investigate virtualisation along with containerisation.
- To showcase Docker deployment and publishing.
- To critically analyse Docker volumes and networking.
- To investigate load balancing in combination with scalability.
- To compare and contrast containers and server less computing.

1.4 Report Structure

The report examines Docker and practical cloud technologies, followed by load balancing, scalability, as well as application gateways. Then it critically discusses the drivers behind cloud adoption and server less computing, compares server less with containerisation, and ultimately concludes with the video demonstration, overall findings and recommendations as well.

2. Cloud Technology: Docker and Practical Examples

2.1 Principles of Virtualisation

2.1.1 Definition and Concept of Virtualisation

Virtualisation is a foundational technology across cloud computing which allows a single physical computing system to offer multiple isolated virtual environments [2]. Instead of allocating one physical server to one application, a hypervisor abstracts the underlying hardware and also allocates the different computing resources such as CPU, memory and storage to multiple virtual machines (VMs). As a result, organisations can enhance the utilisation of the hardware while lowering the physical infrastructure required for operating the applications. Therefore, virtualisation supports the cloud model by allowing providers to share expensive physically infrastructures among multiple customers effectively.

2.1.2 Virtualisation of Shared Hardware

The practical principles can be represented as physical server → hypervisor → multiple virtual machines, where each VM can run its own operating systems as well as applications independently. However, though VMs offer strong isolation and flexibility, they also require separate operating system instances [3]. Therefore, it consumes more resources compared to lightweight containers. Virtualisation consequently presents the underlying infrastructure abstraction upon which modern cloud platforms as well as containerised services can operate.

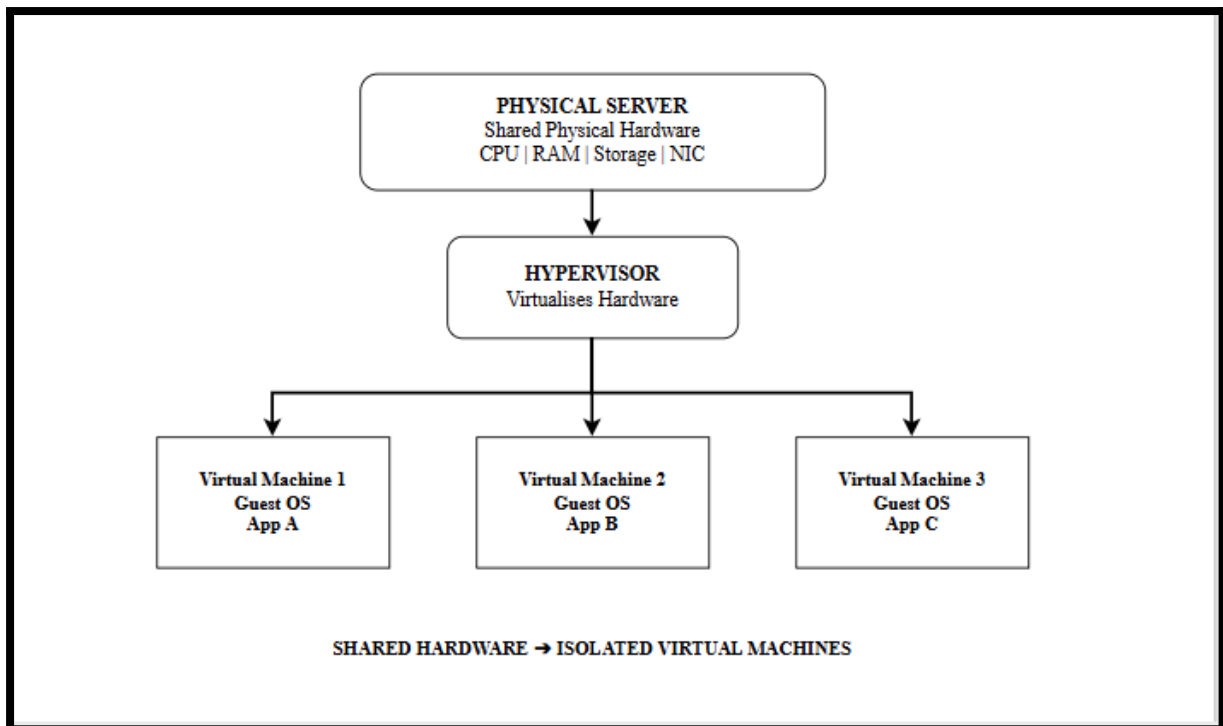


Figure 1: Diagram of Virtualisation

(Source: Self-developed)

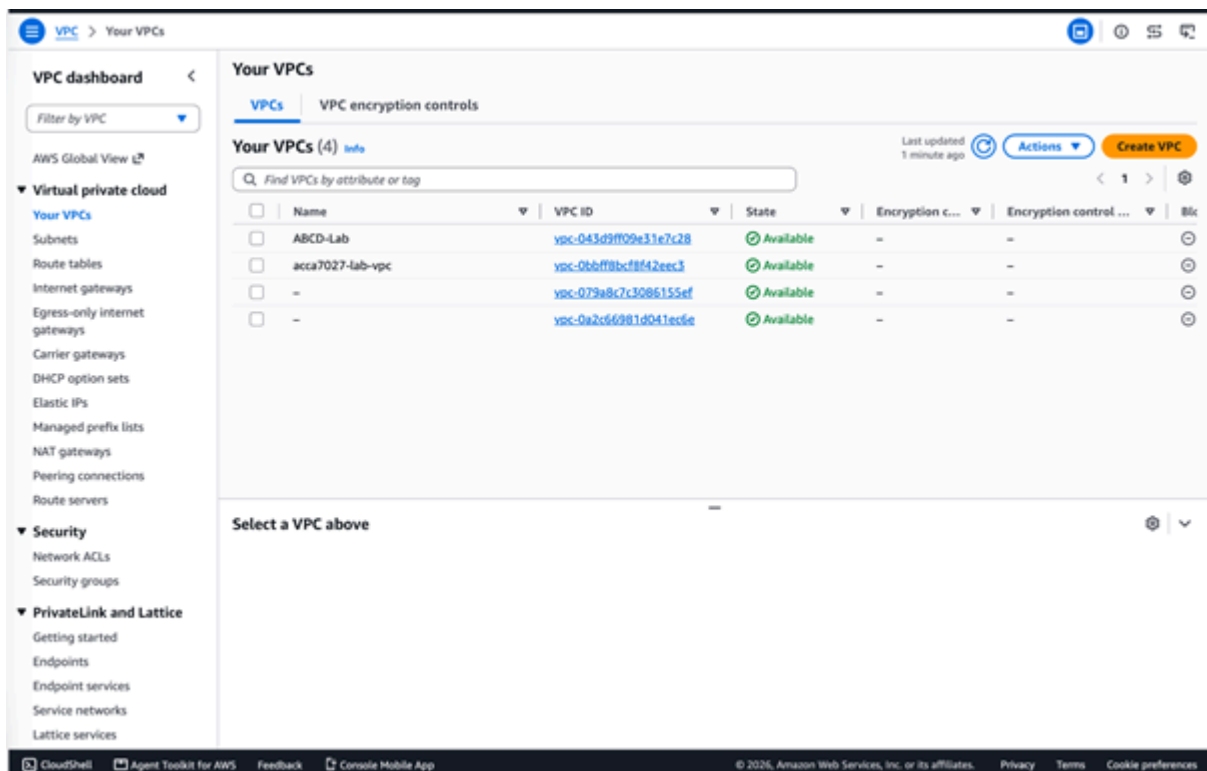


Figure 2: Test Diagram of Virtualisation Coding

(Source: Self-developed)

2.1.3 Type 1 and Type 2 Hypervisors

There are two prior forms of hypervisor, which include: **Type 1 (bare-metal)** and **Type 2 (hosted)** [3]. Type 1 hypervisor operates directly on physical hardware and are commonly related to enterprise as well as cloud environments, while Type 2 hypervisors operate above a conventional host operating system [4]. With respect to cloud data centres, Type 1 visualisation is specifically valuable as it presents effective allocation in combination with workload isolation and thus enhanced the overall scalability.

Type 1	Type 2
Bare-metal	Hosted
Direct hardware access	Runs over host OS
Enterprise/cloud environments	Desktop/lab environments
Higher efficiency	Additional overhead

Table 1: Comparison summary of Type 1 and Type 2 Hypervisors

(Source: Self-developed)

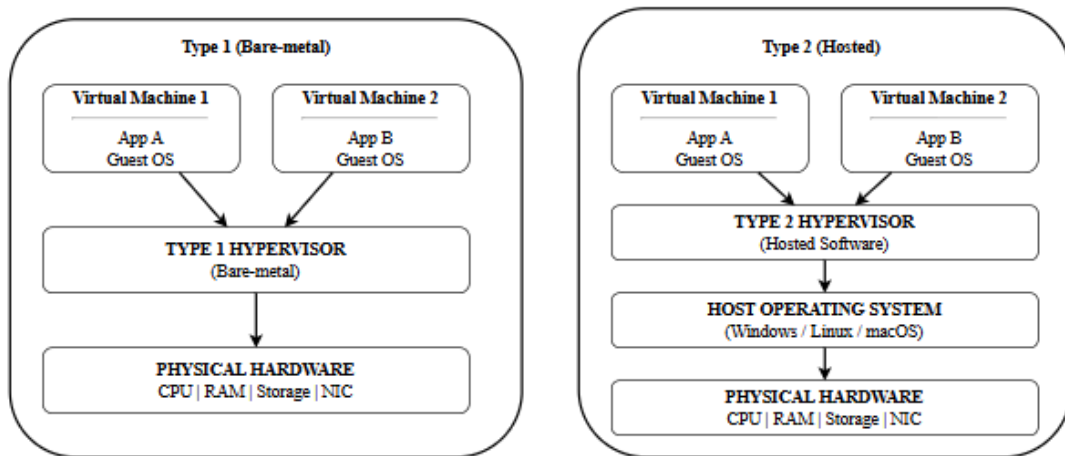


Figure 3: Type 1 versus Type 2 Hypervisor Architecture

(Source: Self-developed)

2.2 Containerisation

2.2.1 Principles of Containerisation

Containerisation is a lightweight form of operating system-level virtualisation in which an application along with its required dependencies is packaged into an isolated container. Unlike virtual machines, which basically require a separate guest operating system, containers share the host operating system kernel while effectively managing the processes, file systems and network isolation as well [5]. This not only reduces the resource overhead but also allows the applications to start rapidly while operating consistently across development, testing and production environments. Docker has become a preferred containerisation platform since it offers tools for not only creating as well as managing the container images.

2.2.2 Docker as a Containerisation Platform

Docker presents an integrated platform for not only creating but also managing and distributing containers [6]. The Docker client is the interface through which users issue any kind of commands, while the Docker Engine offers the core container-management environment. On the flip side, the Docker Daemon helps in managing the images, containers,

networks and finally the volumes. Moreover, a Docker image is a read-only package containing the application code as well as dependencies from which the containers have been developed. The container is the executable instance of an image. Not only that, Docker also supports registries which both store and distribute images, with Docker Hub presenting a widely utilised public registry [7].

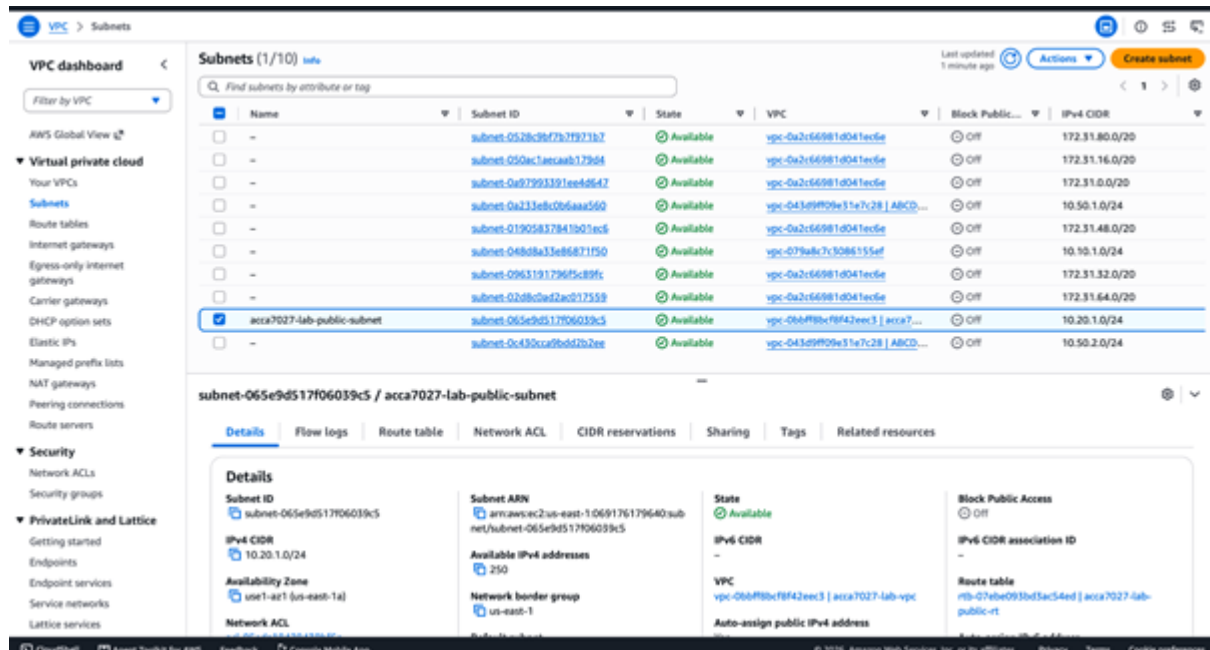


Figure 4: Docker Architecture

(Source: Self-developed)

2.2.3 Containers versus Virtual Machines

Factor	Virtual Machine	Docker Container
Virtualisation	Hardware	OS-level
Kernel	Separate	Shared
Start-up	Slower	Faster
Size	Larger	Smaller

Isolation	Strong	Process-level
Portability	Moderate	High
Resource consumption	Higher	Lower

Table 2: comparison of Containers versus Virtual Machines

(Source: Self-developed)

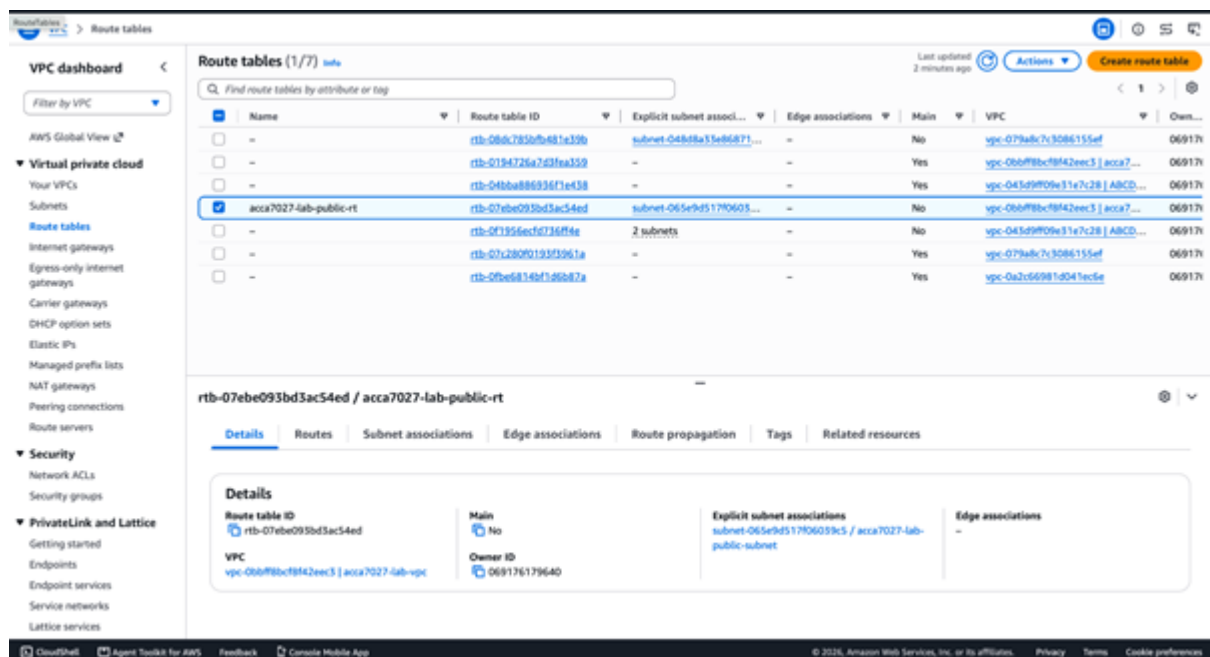


Figure 5: Virtual Machines versus Docker Containers

(Source: Self-developed)

The comparison showcases that VMs provides stronger hardware-level isolation but it consumes more resources because of the separate operating systems [8]. On the flip side, containers are lightweight, faster and more scalable because they share the host kernel. However, VMs remain preferable for the security-sensitive worlds requires a sense of stronger isolation. Therefore, the choice needs to depends on the workload requirements in combination with security needs as well as operational priorities.

2.3 Docker file and Building a Docker Image

2.3.1 Docker file

A Docker file presents a set of instructions for constructing a Docker image which allowing an application and its reliance to be packaged into a reproducible deployment unit. In the practical demonstrations, the Docker file leverages the following instructions:

FROM nginx:alpine

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

EXPOSE 80

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

FROM nginx:alpine presents a lightweight based image while the COPY places index.html across the Nginx's web directory. Besides that, EXPOSE 80 documents HTTP access, and CMD keeps Nginx running. Concurrently, these crucial instructions help to create a portable, reproducible container image while excluding the manual web-server configuration across the individual hosts.

2.3.2 Creating the HTML Web Application

The screenshot displays the AWS Management Console's Security Groups page. On the left, a navigation pane shows 'Virtual private cloud' and 'Security' sections. The main area shows a table of security groups. The 'acc7027-lab-sg' group is selected, and its details are shown below the table.

Name	Security group ID	Security group name	VPC ID	Description
-	sg-03f40b7600992648	default	vpc-043d9f09e31e7c28	default VPC security group
-	sg-089e78f6a9e1810c	RedTeamLab	vpc-0a2c66981d041ec5e	lab
-	sg-00efcda1ebc0417b6	default	vpc-0a2c66981d041ec5e	default VPC security group
-	sg-05596e0502e0e1992	default	vpc-0bbff8bcf8f42ec3	default VPC security group
-	sg-04bed2308e24d046e	abcd-lab-sg	vpc-043d9f09e31e7c28	lab
acc7027-lab-sg	sg-003a9af9b34f891dd	acc7027-lab-sg	vpc-0bbff8bcf8f42ec3	Security group for ACCA7027 Doc
-	sg-0586e38ab77af93d5	default	vpc-079a8c7c3086155ef	default VPC security group
-	sg-0122f7401300e827	ABCD-Lab-SG	vpc-079a8c7c3086155ef	Isolated penetration testing lab

sg-003a9af9b34f891dd - acc7027-lab-sg

Details

Security group name	acc7027-lab-sg	Security group ID	sg-003a9af9b34f891dd	Description	Security group for ACCA7027 Docker cloud lab	VPC ID	vpc-0bbff8bcf8f42ec3
Owner	069176179640	Inbound rules count	3 Permission entries	Outbound rules count	1 Permission entry		

Figure 6: HTML Page/Application Created for the Docker Demonstration

(Source: Self-developed)

The practical implementation begins through effectively developing an HTML webpage which is subsequently copied into the Nginx image [9]. This showcases a significant advantages of containerisation: the application along with its required execution environment can be packaged together, thereby lowering the inconsistencies between both development as well as deployment environments.

2.3.3 Building the Docker Image

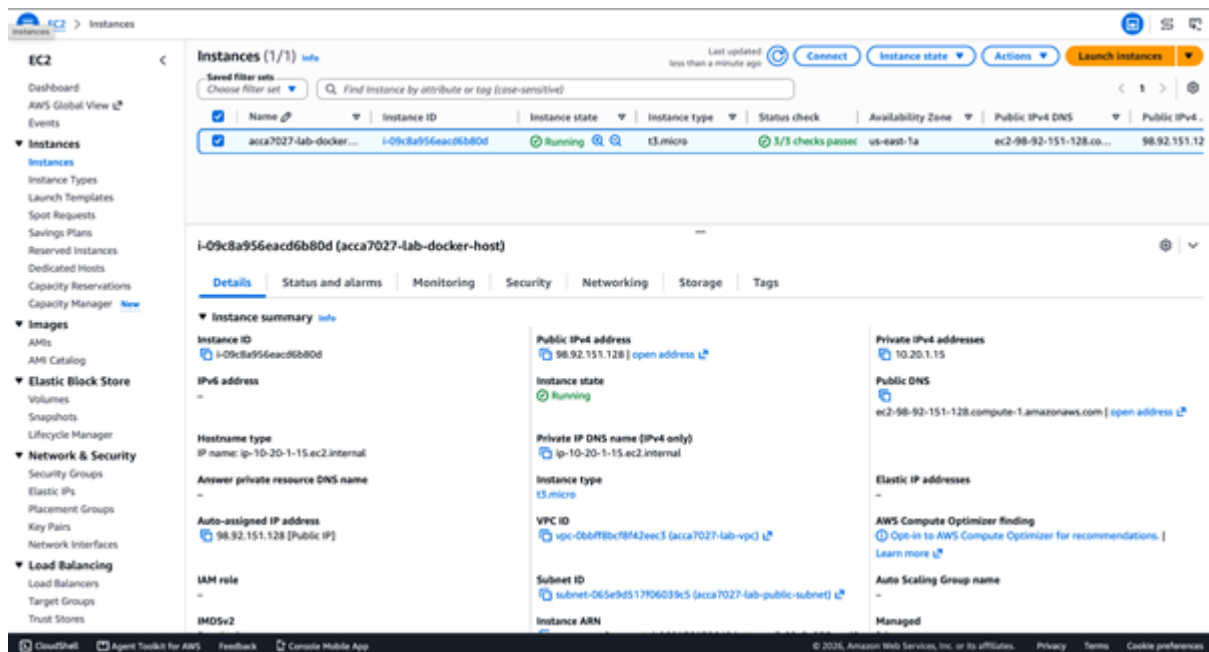


Figure 7: Docker Image Build Process

(Source: Self-developed)

The image is constructed using:

`docker build -t nginx-demo.`

Docker subsequently processes the Docker file, which creates reusable image layers. It also retrieves the Nginx base image, adds the HTML and configuration files while caching the unchanged layers for faster subsequent builds.

2.3.4 Running the Container

The completed image is executed using:

```
docker run -d -p 8080:80 nginx-demo
```

Here, -d run the container in a detached mode while the p 8080:80 maps port 8080 on the host to port 80 inside the container. As a result, the Nginx webpage can be accessed through a browser while the container continues to run independently.

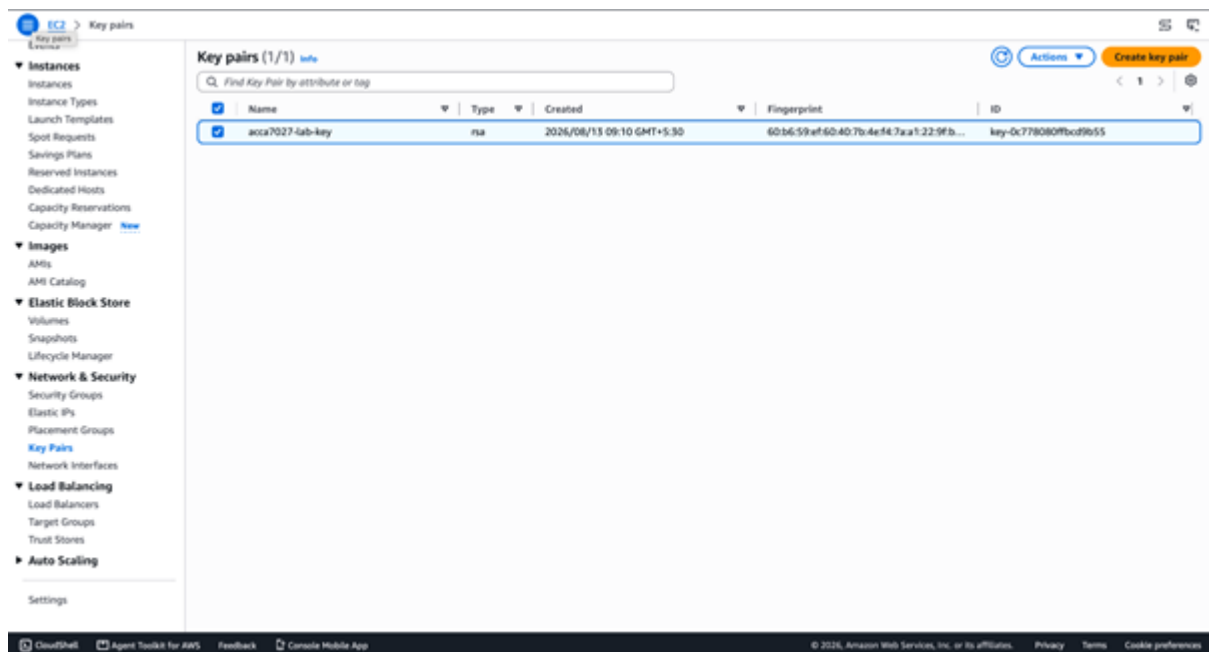


Figure 8: Running Docker Container

(Source: Self-developed)

2.4 Docker Images and Publishing to Docker Hub

2.4.1 Docker Image Lifecycle

A Docker image is a portable, read-only package which contains an application, its dependencies, along with the instructions required to develop a container [10, 11]. The practical lifecycle starts with a Docker file which defines the application environment environments as well as build instructions. Moreover, the Docker build command effectively transforms these instructions into an image which can subsequently be instantiated as a running container [12]. The completed image can then be represented as the Docker file →

Build → Image → Container → Registry. This separation between the application packaging and the execution enhances the portability which supports the consistent cloud deployment.

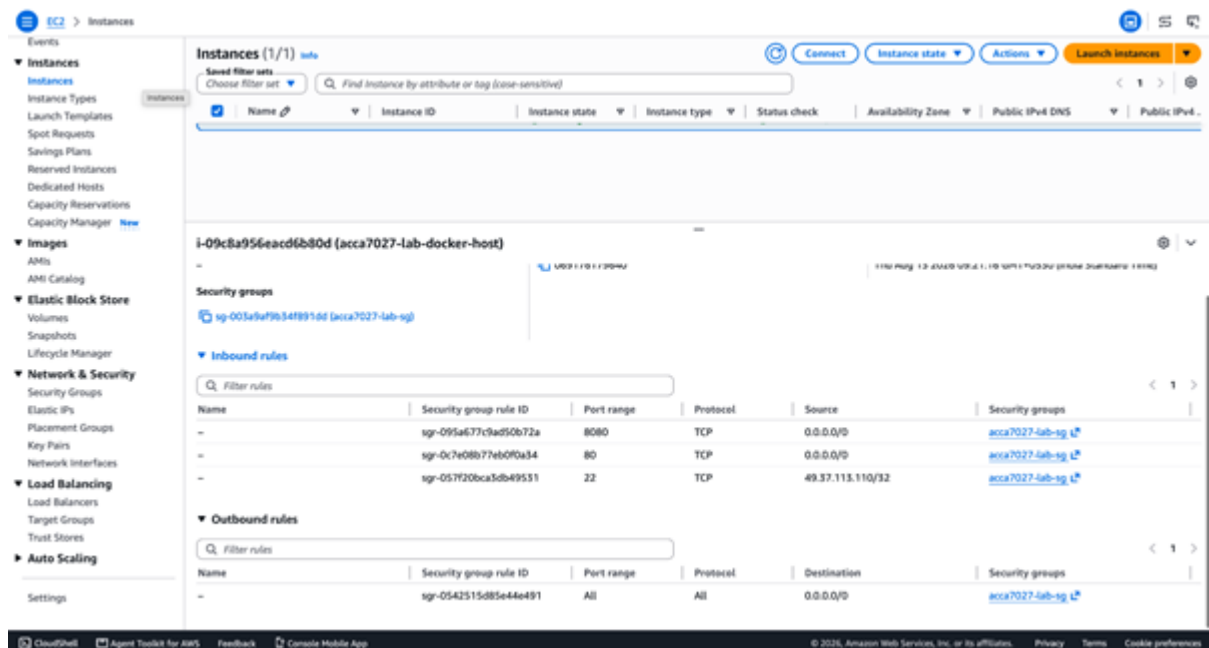


Figure 9: Docker Image Lifecycle

(Source: Self-developed)

2.4.2 Tagging the Image

Before publication, the image can be assigned a repository name along with the version tag:

`docker tag nginx-demo username/nginx-demo: v1`

The tag not only identifies the repository but also the image version, which supports organised image management in combination with reproducible deployments.

2.4.3 Publishing to Docker Hub

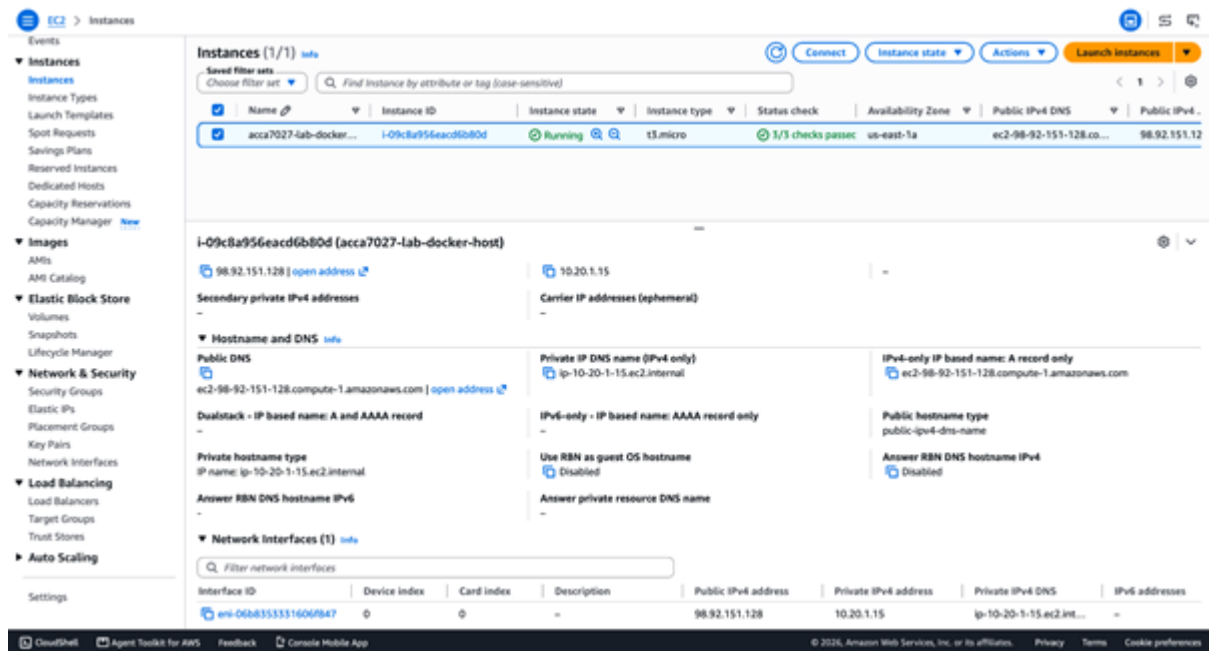


Figure 10: Docker Hub Repository/Published Image

(Source: Self-developed)

Publishing to Docker Hub enables images to be centrally accessed as well as reused, and thus supports the collaboration, version management, automated CI/CD pipelines and finally the faster cloud-native deployment.

2.5 Container Volumes

2.5.1 Purpose of Docker Volumes

The data contained in a container's writable layer is generally considered ephemeral and may be lost when the container is deleted. Docker volumes are specialized directories used by containers to store data [13]. It is important for data persistence when containers stopped, recreated, or deleted. Volumes can maintain application state, configuration, and user-generated data between container deployments. Databases and data must survive application-container replacement; stateful applications particularly benefit from using this technology [14]. Volumes aid in backup and recovery strategies by separating persistent data from the application container. As a result, Docker volumes allow you to combine the portability of containers with the persistence applications need in the real world.

2.5.2 Types of Docker Storage

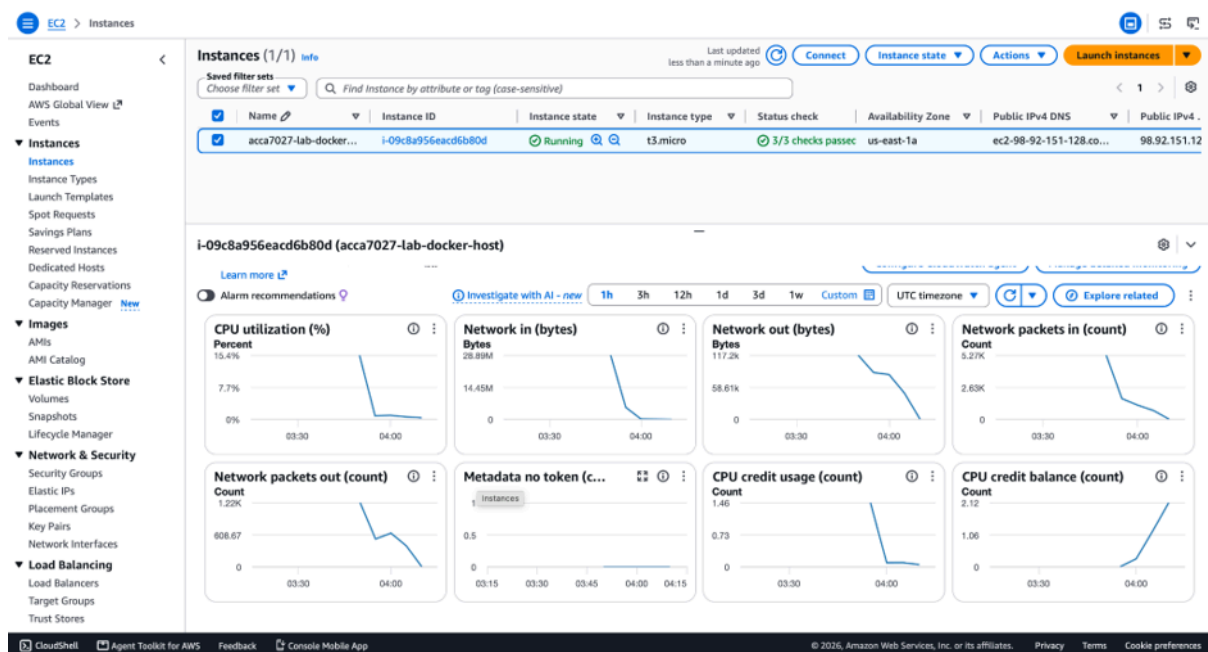


Figure 11: Docker Volume Architecture

(Source: Self-developed)

Docker allows several options to manage application data. Docker manages these volumes. They are stored in Docker's own area so they are used for applications that require persistent data. And we don't want to worry about using host paths for them. With bind mounts, we can choose to bind mount the host's directory or file either to a directory or file within a container. I can enhance the Docker storage functionality by using volume drivers to allow your

volumes to use external or specialized storage systems. The suitable method relies on use case specifications, portability, control and storage architecture.

2.5.3 Practical Docker Volume

```

janvi@Mac janvi-docker-demo % docker volume create janvi-volume
janvi-volume
janvi@Mac janvi-docker-demo % docker volume ps
docker: unknown command: docker volume ps
Usage: docker volume COMMAND
Run 'docker volume --help' for more information
janvi@Mac janvi-docker-demo % docker volume ls
DRIVER      VOLUME NAME
local       janvi-volume
local       my-volume
local       my-volume
janvi@Mac janvi-docker-demo % docker run -d --name janvi-volume-container -v janvi-volume:/data nginx
3173e087f18ef77ed688b565c3a55e38813f305da919069972e3e3ff20968a5
janvi@Mac janvi-docker-demo % docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED      STATUS      PORTS                  NAMES
3173e087f18e   nginx     "/docker-entrypoint..." 4 seconds ago Up 4 seconds 80/tcp                janvi-volume-container
28d3f2ac089f   janvi-docker-web   "/docker-entrypoint..." 4 minutes ago Up 4 minutes 0.0.0.0:8080->80/tcp   janvi-web-container
janvi@Mac janvi-docker-demo % docker network

```

Figure 12: Docker Volume Practical Demonstration

(Source: Self-developed)

The following command can be used to create a usable Docker volume.

“Docker volume creates volume-demo”

We can mount the resulting named volume into a Nginx container so application content is stored separately from the container. Such As.

“Docker run -d -p 8080:80 -v volume-demo:/usr/share/nginx/html nginx”

Here, Nginx's web-root directory is mounted on volume-demo. Files recorded here are kept in the docker volume and not just in the writable layer of the container. As a result, the data can still be available after a container is removed and a new container is created using the same volume. See how Docker supports best practices for persistent application storage.

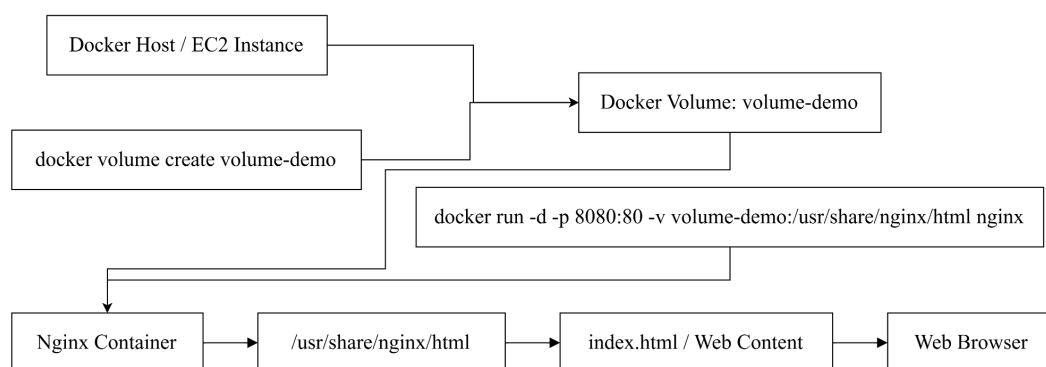


Figure 12: Practical Docker Volume Diagram

(Source: Self-developed)

2.6 Container Networking

2.6.1 Docker Networking Principles

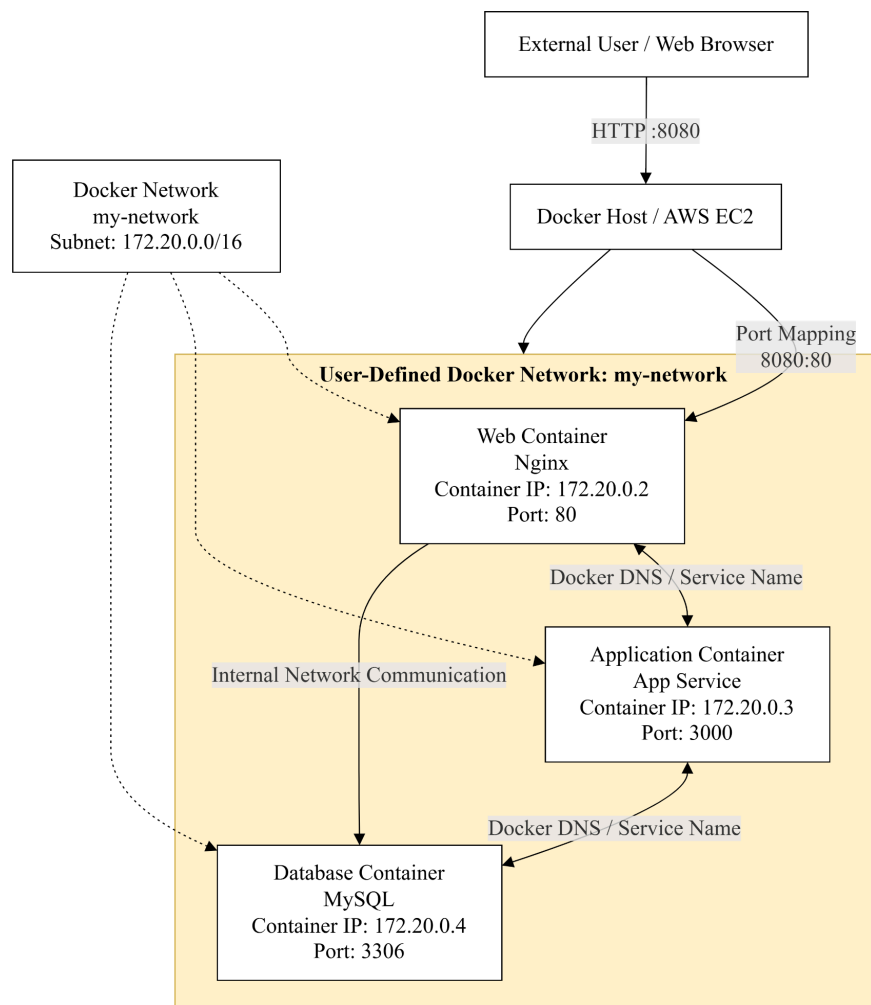


Figure 13: Docker Networking practical Diagram

(Source: Self-developed)

Docker's networking allows for the connection of containers to each other and the outside world, while still maintaining network isolation. Container-to-container communication refers to the process of exchanging data between multiple application components, including web servers and databases, over a Docker network [15]. Every container has its own IP address, which is used for identification over the network. In containerization, port mapping

enables one or more container ports to be connected to a host port. Consequently, an external user can access the services that are running within that container. Essentially, DNS-based service that docker provides helps provide insight to various containers that are on a user-defined same network. It helps identify each other using a name rather than an IP address.

2.6.2 Docker Network Types

The bridge network is often used for containers on the same Docker host to communicate with each other. By giving a container access to the network stack of the host, that container will behave as if it was running on the host itself [16].

2.6.3 Creating a Docker Network

Developing a custom Docker network.

“Docker network creates my-network”

Containers can create this network.

For example, “docker run -d --name web --network my-network nginx”

Another container.

“Docker run -it --name client --network my-network alpine”

Using DNS service within Docker offers a practical way of connecting application components without exposing every service directly to the outside world. This shows how Docker networking enables organized communication among several containers.

```
$ aws configure list
name      : VALUE
profile   : <not set>
type      : None
location  : None
access_key : *****6dN2
secret_key : *****7h18
region    : us-east-1
config_file : ~/.aws/config

$ aws ec2 describe-regions \
  --query 'Regions[?OptInStatus==`opt-in-not-required`].RegionName' \
  --output text
ap-south-1 eu-north-1 eu-west-3 eu-west-2 eu-west-1 ap-northeast-3 ap-northeast-2 ap-northeast-1 ca-central-1 sa-east-1 ap-southeast-1 ap-southeast-2 eu-central-1 us-east-1 us-east-2
```

Figure 14: Docker Network Creation

(Source: Self-developed)

```

$ aws ec2 describe-key-pairs \
  --region us-east-1 \
  --query 'KeyPairs[].KeyName' \
  --output table
$ aws ec2 create-key-pair \
  --region us-east-1 \
  --key-name acca7027-lab-key \
  --query 'KeyMaterial' \
  --output text > ~/Downloads/acca7027-lab-key.pem
$ chmod 400 ~/Downloads/acca7027-lab-key.pem
$ aws ec2 describe-key-pairs \
  --region us-east-1 \
  --query 'KeyPairs[].KeyName' \
  --output table
+-----+
| DescribeKeyPairs |
+-----+
| acca7027-lab-key |
+-----+
$ export AWS_REGION="us-east-1"

export PROJECT="acca7027"
export ENV="lab"

export KEY_NAME="${PROJECT}-lab-key"

export VPC_CIDR="10.20.0.0/16"
export PUBLIC_SUBNET_CIDR="10.20.1.0/24"
export PRIVATE_SUBNET_CIDR="10.20.2.0/24"

export AZ="us-east-1a"

export INSTANCE_TYPE="t3.micro"

export VPC_NAME="${PROJECT}-${ENV}-vpc"
export PUBLIC_SUBNET_NAME="${PROJECT}-${ENV}-public-subnet"
export PRIVATE_SUBNET_NAME="${PROJECT}-${ENV}-private-subnet"
export SG_NAME="${PROJECT}-${ENV}-sg"
$ printf '%s\n' \
  "AWS_REGION=$AWS_REGION" \
  "PROJECT=$PROJECT" \
  "ENV=$ENV" \
  "KEY_NAME=$KEY_NAME" \
  "VPC_CIDR=$VPC_CIDR" \
  "PUBLIC_SUBNET_CIDR=$PUBLIC_SUBNET_CIDR" \
  "AZ=$AZ" \
  "INSTANCE_TYPE=$INSTANCE_TYPE"
AWS_REGION=us-east-1
PROJECT=acca7027
ENV=lab
KEY_NAME=acca7027-lab-key
VPC_CIDR=10.20.0.0/16
PUBLIC_SUBNET_CIDR=10.20.1.0/24
AZ=us-east-1a
INSTANCE_TYPE=t3.micro
$ export PRIVATE_SUBNET_CIDR="10.20.2.0/24"
export VPC_NAME="${PROJECT}-${ENV}-vpc"
export PUBLIC_SUBNET_NAME="${PROJECT}-${ENV}-public-subnet"
export PRIVATE_SUBNET_NAME="${PROJECT}-${ENV}-private-subnet"
export SG_NAME="${PROJECT}-${ENV}-sg"
export IGW_NAME="${PROJECT}-${ENV}-igw"
export PUBLIC_RT_NAME="${PROJECT}-${ENV}-public-rt"
$ printf '%s\n' \

```

Figure 15: Container Network Communication

(Source: Self-developed)

The figure shows the communication between two containers connected on the same docker network. This practical implementation shows how networking is involved in the multi-container architecture. The services can work independent of each other and still share information via an isolated and logically organized network.

2.7 Integrated Docker Demonstration

We first create an HTML webpage and Dockerfile which is based on the Nginx image. When it is built into an image, the Dockerfile is highly reusable. This image is then run as a container and accessed via a web browser. You can tag and publish the resulting image to Docker Hub for use and sharing. This workflow exhibits how the process of containerization provides consistent applications. Likewise, while images are application templates that can be easily transported, containers are executables [17].

3. Cloud Computing Concepts

3.1 Load Balancing

3.1.1 Definition

Load balancing is a technique that cloud computing uses to distribute the incoming application requests. Load balancing increases service availability, performance and reliability by preventing over-concentration of traffic on a single server [18]. It helps organizations to use computing resources efficiently, while it also supports applications that experience fluctuating workloads and demand.

3.1.2 Benefits

Load balancing offers numerous important advantages for applications in the cloud. When one instance goes down the traffic will be redirected to the healthy server making it more available. The performance enhances as the heavy workloads are distributed instead of concentrated on a server. Redundancies empower applications to keep functioning despite the failure of certain instances [19]. Load balancing improves user experience by minimising delays, avoiding server overload and ensuring applications are responsive all the time [20]. In cloud services where traffic levels may vary widely and erratically, these benefits are particularly salient.

3.1.3 Example: AWS Elastic Load Balancer

```
$ aws ec2 modify-vpc-attribute \
  --region "$AWS_REGION" \
  --vpc-id "$VPC_ID" \
  --enable-dns-support '{"Value":true}'
$ aws ec2 modify-vpc-attribute \
  --region "$AWS_REGION" \
  --vpc-id "$VPC_ID" \
  --enable-dns-hostnames '{"Value":true}'
$ aws ec2 describe-vpc-attribute \
  --region "$AWS_REGION" \
  --vpc-id "$VPC_ID" \
  --attribute enableDnsSupport
{
  "EnableDnsSupport": {
    "Value": true
  },
  "VpcId": "vpc-0bbff8bcf8f42eec3"
}
$ aws ec2 describe-vpc-attribute \
  --region "$AWS_REGION" \
  --vpc-id "$VPC_ID" \
  --attribute enableDnsHostnames
{
  "EnableDnsHostnames": {
    "Value": true
  },
  "VpcId": "vpc-0bbff8bcf8f42eec3"
}
$ aws ec2 describe-vpc-attribute \
  --region "$AWS_REGION" \
  --vpc-id "$VPC_ID" \
  --attribute enableDnsHostnames
{
  "EnableDnsHostnames": {
    "Value": true
  },
  "VpcId": "vpc-0bbff8bcf8f42eec3"
}
$ IGW_ID=$(aws ec2 create-internet-gateway \
  --region "$AWS_REGION" \
  --query 'InternetGateway.InternetGatewayId' \
  --output text)
echo "IGW_ID=$IGW_ID"
IGW_ID=igw-0bde67422633439e0
$ aws ec2 create-tags \
  --region "$AWS_REGION" \
  --resources "$IGW_ID" \
  --tags Key=Name,Value="$IGW_NAME" Key=Project,Value="$PROJECT" Key=Environment,Value="$ENV"
$ aws ec2 attach-internet-gateway \
  --region "$AWS_REGION" \
  --internet-gateway-id "$IGW_ID" \
  --vpc-id "$VPC_ID"
$ aws ec2 describe-internet-gateways \
  --region "$AWS_REGION" \
  --internet-gateway-ids "$IGW_ID" \
  --query 'InternetGateways[0].{ID:InternetGatewayId,State:Attachments[0].State,VPC:Attachments[0].VpcId,Name:Tags[?Key==`Name`][0].Value}' \
  --output table
+-----+-----+-----+-----+
| DescribeInternetGateways |
+-----+-----+-----+-----+
| ID | Name | State | VPC |
+-----+-----+-----+-----+
| igw-0bde67422633439e0 | accs7827-lab-igw | available | vpc-0bbff8bcf8f42eec3 |
+-----+-----+-----+-----+
$ PUBLIC_SUBNET_ID=$(aws ec2 create-subnet \
```

Figure 16: Cloud Load Balancing Architecture

(Source: Self-developed)

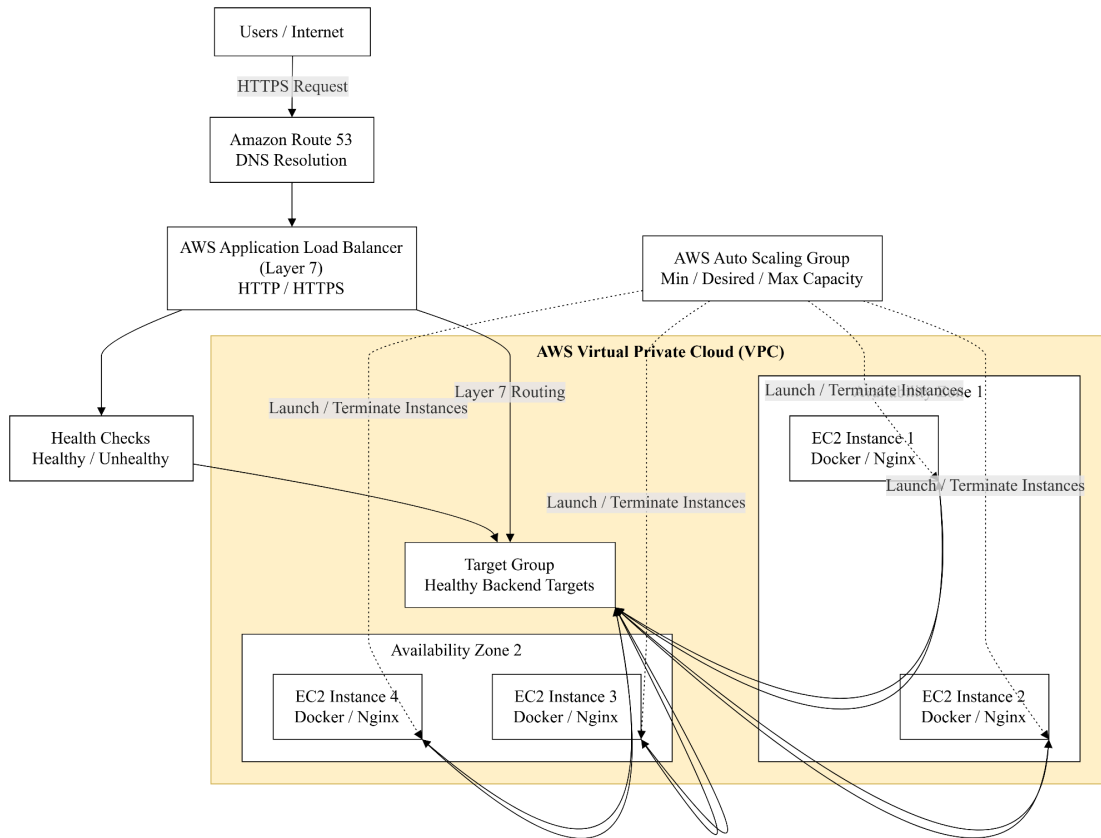


Figure 17: AWS ALB Diagram

(Source: Self-developed)

The Elastic Load Balancing (ELB) feature offered by Amazon Web Services (AWS) can distribute application traffic across a set of multiple targets. The Application Load Balancer (ALB) integrates on Layer 7, which allows routing on HTTP and HTTPS requests. The ALB helps to manage EC2 (Elastic Compute Cloud) instances and any traffic that comes to them by using target groups. Health checks continuously assess the availability of targets and prevent sending requests to unhealthy targets [21]. As a result, the traffic can be routed to EC2 instances that are healthy. By integrating with EC2 and other AWS services, ALB can work within scalable cloud architectures.

3.2 Scalability

3.2.1 Horizontal Scaling

Horizontal scaling means adding or subtracting application-delivering computing instances. It is not made that one server more powerful rather introduced with other servers. The process of horizontal scaling involves increasing the number of connections of the same type to a

computing resource. In cloud environments, it is particularly useful because organisations are able to provision more virtual machines or application instances when the demand increases [22]. They are also able to remove those instances when the demand falls. Multi-access edge computing method enhances access, facilitates a greater number of simultaneous users, and lessens reliance on any one server [23]. In conjunction with a load balancer, any incoming requests are sent to the additional instances; this enables the application to keep up with demand and efficiently respond to varying customer traffic.

3.2.2 Vertical Scaling

Vertical scaling refers to the addition of resources like CPU or memory to an existing server. An organisation can augment the CPU capacity, memory or other resources available to an EC2 instance when an application requires more processing power [24]. It can be easier than scaling horizontally since the application environment will not have to change much. Nonetheless, there are practical limits to vertical scaling, since the base instance type has a cap on resources. Based on the configuration and application design, it might also need changes or downtime. For a particular work-load, vertical scaling works. For highly available (highly available) cloud-native applications, horizontal scaling of multiple instances is more effective.

3.2.3 AWS Auto Scaling

AWS Auto Scaling automates the adjustment of compute capacity for applications based on demand. An ASG can specify a minimum capacity ensuring that a certain number of instances are available at all times [25]. Desired capacity is the number of running instances you want at all times, while the maximum capacity is the upper limit of running instances in your application when demand increases. Scaling policies define when to add or remove instances [26]. Demand-based scaling responds to indicators such as CPU utilization and application demand, for example.

```

$ PUBLIC_SUBNET_ID=$(aws ec2 create-subnet \
  --region "$AWS_REGION" \
  --vpc-id "$VPC_ID" \
  --cidr-block "$PUBLIC_SUBNET_CIDR" \
  --availability-zone "$AZ" \
  --query 'Subnet.SubnetId' \
  --output text)

echo "PUBLIC_SUBNET_ID=$PUBLIC_SUBNET_ID"
PUBLIC_SUBNET_ID=subnet-065e9d517f06039c5
$ aws ec2 create-tags \
  --region "$AWS_REGION" \
  --resources "$PUBLIC_SUBNET_ID" \
  --tags \
  Key=Name,Value="$PUBLIC_SUBNET_NAME" \
  Key=Project,Value="$PROJECT" \
  Key=Environment,Value="$ENV" \
  Key=Tier,Value=public
$ aws ec2 modify-subnet-attribute \
  --region "$AWS_REGION" \
  --subnet-id "$PUBLIC_SUBNET_ID" \
  --map-public-ip-on-launch
$ aws ec2 describe-subnets \
  --region "$AWS_REGION" \
  --subnet-ids "$PUBLIC_SUBNET_ID" \
  --query 'Subnets[0].{ID:SubnetId,CIDR:CidrBlock,AZ:AvailabilityZone,VPC:VpcId,PublicIP:MapPublicIpOnLaunch,Name:Tags[?Key==`Name`][0].Value}' \
  --output table

```

DescribeSubnets					
AZ	CIDR	ID	Name	PublicIP	VPC
us-east-1a	10.20.1.0/24	subnet-065e9d517f06039c5	acca7027-lab-public-subnet	True	vpc-0bbff8bcf8f42eec3

```

$ PUBLIC_RT_ID=$(aws ec2 create-route-table \
  --region "$AWS_REGION" \
  --vpc-id "$VPC_ID" \
  --query 'RouteTable.RouteTableId' \
  --output text)

echo "PUBLIC_RT_ID=$PUBLIC_RT_ID"
PUBLIC_RT_ID=rtb-07ebe093bd3ac54ed
$ aws ec2 create-tags \
  --region "$AWS_REGION" \
  --resources "$PUBLIC_RT_ID" \
  --tags \
  Key=Name,Value="$PUBLIC_RT_NAME" \
  Key=Project,Value="$PROJECT" \
  Key=Environment,Value="$ENV" \
  Key=Tier,Value=public
$ aws ec2 create-route \
  --region "$AWS_REGION" \
  --route-table-id "$PUBLIC_RT_ID" \
  --destination-cidr-block "0.0.0.0/0" \
  --gateway-id "$IGW_ID"
{
  "Return": true
}

```

Figure 18: Cloud Auto Scaling Architecture

(Source: Self-developed)

Figure 16 shows how the automatic adjustment of a cloud computing capacity meets application demand. Through proper instance capacity management, Auto Scaling can manage changing workloads without a continuous manual effort.

3.3 Application Gateway

The application gateway is a Layer 7 traffic-management service at the application layer of the networking stack. It can inspect various layers of an application, such as the content of HTTP and HTTPS requests, and make smart routing decisions. This is url based routing which means route requests to different backend services based on their paths like /products or /orders [27]. In host-based routing, traffic is directed based on the requested domain/hostname.

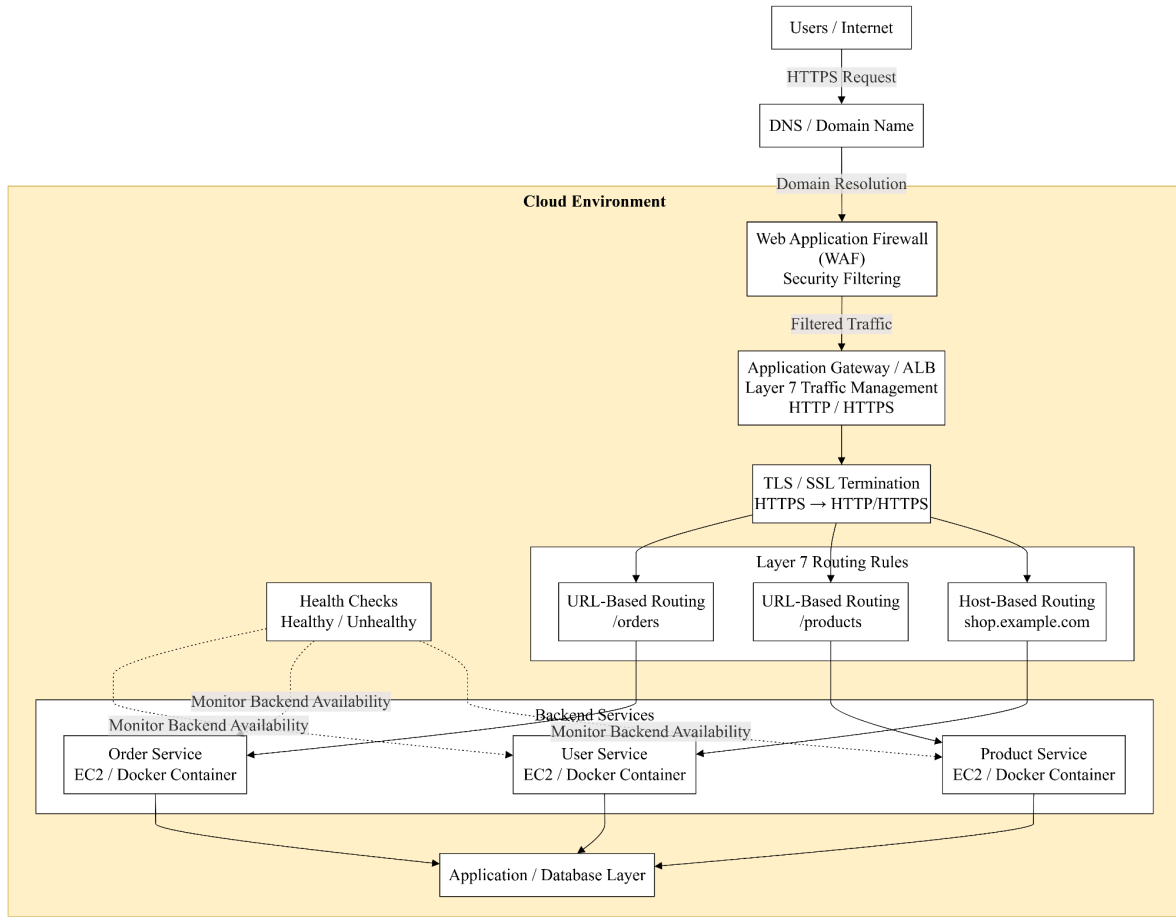


Figure 19: Application Gateway / Layer 7

(Source: Self-developed)

Application gateways also perform TLS/SSL termination, meaning TLS/SSL encrypted connections from clients to the gateway can be terminated and requests sent to back-end services. Connecting with a WAF can help to safeguard against common web-based attacks. The backend service routing facilitates the redirection of requests to the appropriate application servers or services by the gateway. Likewise, health monitoring can assist in avoiding non-available backends [28]. As a result, application gateways provide both security and availability with optimized request routing, traffic management and security controls. In the cloud architecture, they can work in conjunction with load balancers and auto-scaling services. It makes the applications reliable and responsive.

```

}
$ PUBLIC_RT_ASSOC_ID=$(aws ec2 associate-route-table \
--region "$AWS_REGION" \
--route-table-id "$PUBLIC_RT_ID" \
--subnet-id "$PUBLIC_SUBNET_ID" \
--query 'AssociationId' \
--output text)

(echo "PUBLIC_RT_ASSOC_ID=$PUBLIC_RT_ASSOC_ID"
PUBLIC_RT_ASSOC_ID=rtbassoc-0a325d89d9a78803e
$ aws ec2 describe-route-tables \
--region "$AWS_REGION" \
--route-table-ids "$PUBLIC_RT_ID" \
--query 'RouteTables[0].{RouteTable:RouteTableId,VPC:VpcId,Routes:Routes,Associations:Associations}' \
--output json
{
  "RouteTable": "rtb-07ebe093bd3ac54ed",
  "VPC": "vpc-0bbff8bcf8f42eec3",
  "Routes": [
    {
      "DestinationCidrBlock": "10.20.0.0/16",
      "GatewayId": "local",
      "Origin": "CreateRouteTable",
      "State": "active"
    },
    {
      "DestinationCidrBlock": "0.0.0.0/0",
      "GatewayId": "igw-0bde67422633439e0",
      "Origin": "CreateRoute",
      "State": "active"
    }
  ],
  "Associations": [
    {
      "Main": false,
      "RouteTableAssociationId": "rtbassoc-0a325d89d9a78803e",
      "RouteTableId": "rtb-07ebe093bd3ac54ed",
      "SubnetId": "subnet-065e9d517f06039c5",
      "AssociationState": {
        "State": "associated"
      }
    }
  ]
}
$ MY_IP=$(curl -s https://checkip.amazonaws.com)

(echo "MY_IP=$MY_IP"
MY_IP=49.37.113.110
$ SG_ID=$(aws ec2 create-security-group \
--region "$AWS_REGION" \
--group-name "$SG_NAME" \
--description "Security group for ACCA7027 Docker cloud lab" \
--vpc-id "$VPC_ID" \
--query 'GroupId' \
--output text)

(echo "SG_ID=$SG_ID"
SG_ID=sg-003a9af9b34f891dd
$ aws ec2 create-tags \
--region "$AWS_REGION" \

```

Figure 20: Application Gateway / Layer 7 Routing

(Source: Self-developed)

As shown in figure 17, application gateway decides on the HTTP/HTTPS requests before sending them to the back end. Through routing rules, the Gateway makes users reach to services

3.4 Importance of Load Balancing and Scaling

Load balancing and scaling are two different but related concepts in cloud computing that handle different aspects of the application. Scaling adjusts computer capacity up or down depending on need. Load balancing distributes requests throughout available capacity. As a result, scaling without load balancing could lead to uneven workload distribution, while load balancing without sufficient capacity cannot respond to a substantial increase in demand.

For example, if application traffic spikes, a group of EC2 instances can be launched to offer greater processing capacity through AWS Auto Scaling groups [29]. The Application Load Balancer (ALB) can start sending requests to the new instance as soon as it passes the health checks [30]. This allows the application to increase the user capacity quickly without a single instance suffering from a performance bottleneck. Auto Scaling can remove instances from the Availability Zone as demand reduces. Subsequently, the ALB continues routing the traffic to the remaining healthy ones.

```
$ aws ec2 authorize-security-group-ingress \
  --region "$AWS_REGION" \
  --group-id "$SG_ID" \
  --ip-permissions \
  {
    "IpProtocol": "tcp", "FromPort": 22, "ToPort": 22, "IpRanges": [{"CidrIp": "${MY_IP}/32", "Description": "Administration"}]
  }
{
  "Return": true,
  "SecurityGroupRules": [
    {
      "SecurityGroupRuleId": "sgr-057f20bca3db49531",
      "GroupId": "sg-003a9af9b34f891dd",
      "GroupOwnerId": "069176179640",
      "IsEgress": false,
      "IpProtocol": "tcp",
      "FromPort": 22,
      "ToPort": 22,
      "CidrIpv4": "49.37.113.110/32",
      "Description": "Administration",
      "SecurityGroupRuleArn": "arn:aws:ec2:us-east-1:069176179640:security-group-rule/sgr-057f20bca3db49531"
    }
  ]
}
$ aws ec2 authorize-security-group-ingress \
  --region "$AWS_REGION" \
  --group-id "$SG_ID" \
  --protocol tcp \
  --port 80 \
  --cidr 0.0.0.0/0
{
  "Return": true,
  "SecurityGroupRules": [
    {
      "SecurityGroupRuleId": "sgr-0c7e08b77eb0f0a34",
      "GroupId": "sg-003a9af9b34f891dd",
      "GroupOwnerId": "069176179640",
      "IsEgress": false,
      "IpProtocol": "tcp",
      "FromPort": 80,
      "ToPort": 80,
      "CidrIpv4": "0.0.0.0/0",
      "SecurityGroupRuleArn": "arn:aws:ec2:us-east-1:069176179640:security-group-rule/sgr-0c7e08b77eb0f0a34"
    }
  ]
}
$ aws ec2 authorize-security-group-ingress \
  --region "$AWS_REGION" \
  --group-id "$SG_ID" \
  --protocol tcp \
  --port 8080 \
  --cidr 0.0.0.0/0
{
  "Return": true,
  "SecurityGroupRules": [
    {
      "SecurityGroupRuleId": "sgr-095a677c9ad50b72a",
      "GroupId": "sg-003a9af9b34f891dd",
      "GroupOwnerId": "069176179640",
      "IsEgress": false,
      "IpProtocol": "tcp",
      "FromPort": 8080,
      "ToPort": 8080,
      "CidrIpv4": "0.0.0.0/0",
      "SecurityGroupRuleArn": "arn:aws:ec2:us-east-1:069176179640:security-group-rule/sgr-095a677c9ad50b72a"
    }
  ]
}
```

Figure 21: Load Balancing + Auto Scaling

(Source: Self-developed)

Combining these factors ensures flexibility, accessibility and effective resource use. When viewed through the lens of business, they boost customer experience during heavy demand

while avoiding the cost of always-on huge infrastructure. As a result, scalability provides capacity and load balancing ensures that it is used efficiently.

3.5 Combining Load Balancing, Scalability and Application Gateway

By integrating application gateway, load balancing and scalability creates an architecture so that cloud services are reliable for customers.

Overall flow

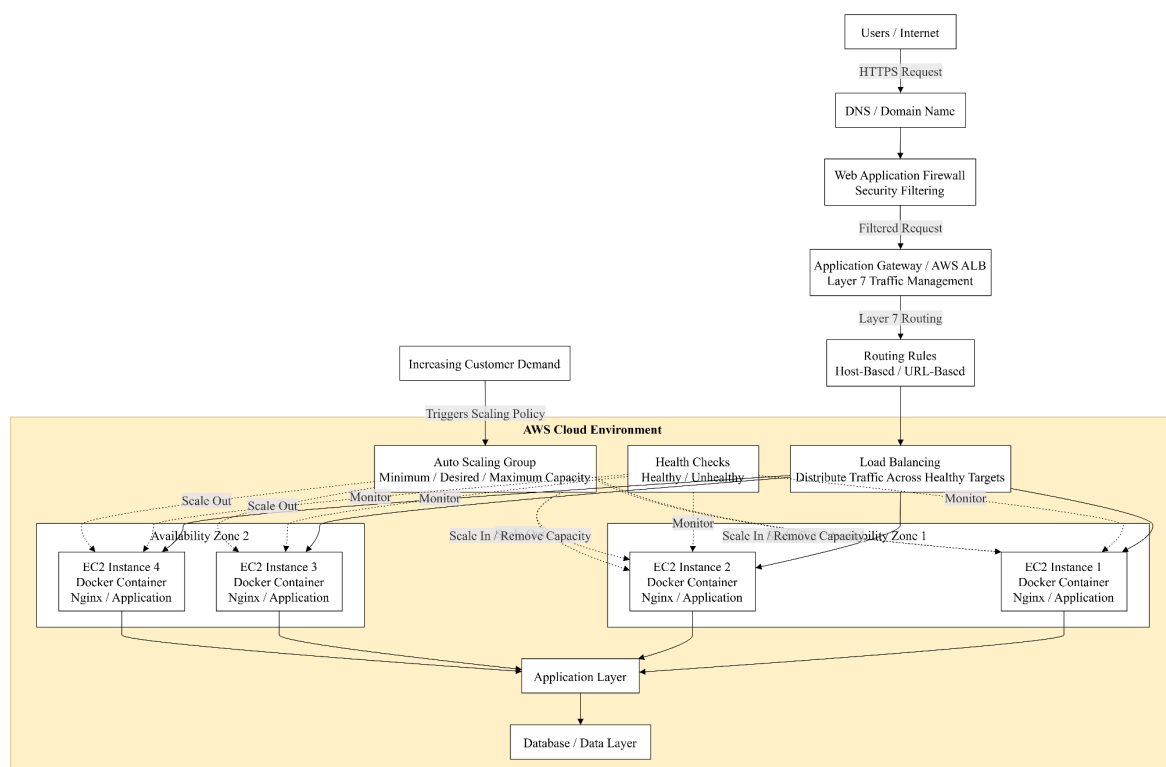


Figure 22: Flow Diagram of Load Balancing, Scalability and Application Gateway

(Source: Self-developed)

When customers request something, it first goes to the application gateway or the Application Load Balancer (ALB) that can look at application software traffic and apply routing rules. Subsequently, various healthy backend resources hosted either on EC2 instances or in Docker containers will receive the requests.

```

$ aws ec2 describe-security-groups \
  --region "$AWS_REGION" \
  --group-ids "$SG_ID" \
  --query 'SecurityGroups[0].{ID:GroupId,Name:GroupName,VPC:VpcId,Rules:IpPermissions}' \
  --output json
{
  "ID": "sg-003a9af9b34f891dd",
  "Name": "acca7827-lab-sg",
  "VPC": "vpc-8bbff8bcf8f42eac3",
  "Rules": [
    {
      "IpProtocol": "tcp",
      "FromPort": 80,
      "ToPort": 80,
      "UserIdGroupPairs": [],
      "IpRanges": [
        {
          "CidrIp": "0.0.0.0/0"
        }
      ],
      "Ipv6Ranges": [],
      "PrefixListIds": []
    },
    {
      "IpProtocol": "tcp",
      "FromPort": 8080,
      "ToPort": 8080,
      "UserIdGroupPairs": [],
      "IpRanges": [
        {
          "CidrIp": "0.0.0.0/0"
        }
      ],
      "Ipv6Ranges": [],
      "PrefixListIds": []
    },
    {
      "IpProtocol": "tcp",
      "FromPort": 22,
      "ToPort": 22,
      "UserIdGroupPairs": [],
      "IpRanges": [
        {
          "Description": "Administration",
          "CidrIp": "49.37.113.110/32"
        }
      ],
      "Ipv6Ranges": [],
      "PrefixListIds": []
    }
  ]
}
$ AMI_ID=$(aws ec2 describe-images \
  --region "$AWS_REGION" \
  --owners amazon \
  --filters \
    "Name=name,Values=al2023-ami-2023*-x86_64" \
    "Name=state,Values=available" \
    "Name=root-device-type,Values=efs" \
    "Name=virtualization-type,Values=hvm" \
  --query 'sort_by(Images,&CreationDate)[-1].ImageId' \
  --output text)
echo "AMI_ID=$AMI_ID"
AMI_ID=ami-0bdc7d025135d7b49

```

Figure 23: Integrated Cloud Application Architecture

(Source: Self-developed)

The combination enhances availability as traffic can be redirected to functioning resources when a particular one fails. Auto Scaling is used for scalability because it can add and remove EC2 instances according to demand. The burden on a single server highlights the Load balancing enhances performance by allocating requests across a range of resources rather than placing [31]. As a result, a cloud service architecture that is flexible, resilient and efficient.

4. Main Driving Forces Behind the Expansion of Cloud Computing

4.1 Cost Efficiency

A key reason why organisations are adopting the cloud is sustainability and cost efficiency. They can replace significant CapEx on physical servers and data-centre infrastructure with OpEx based on actual consumption. Cloud providers such as AWS and Microsoft Azure offer pay-as-you-go models of consumption, letting organisations avoid buying infrastructure to meet occasional peaks in demand [32].

4.2 Flexibility and Agility

The development environment can be created in minutes with rapid provisioning. Elastic resources help increase or decrease capacity as workload demands. The global deployment of cloud infrastructures enables applications to be closer to the widely located customers. Allowing businesses to experiment with more applications and services faster, this offers improved organisational agility. In addition, DevOps and CI/CD integration supports automated development, testing and deployment.

4.3 Scalability and Elasticity

Scalability and elasticity are two key drivers for the adoption of cloud computing, because the cloud infrastructure can automatically react to dynamic application demands. Scalability allows organizations to enhance computing capacity as workloads increase, by deploying additional instances or increasing current capacity. Elasticity extends this principle to automatically extend or reduce the capacity according to the requirements in real time. An e-commerce app might need extra EC2 instances for a big sale, and fewer when demand normalizes [33]. This flexibility maintains performance quality during peak times and saves unnecessary investment in infrastructure during slack times.

4.4 Availability and Global Reach

Cloud computing enhances availability and global reach by enabling applications to run on geographically dispersed infrastructure. Cloud providers (AWS for example) group infrastructure into Regions and Availability Zones, allowing organisations to distribute workloads across different locations and make them less dependent on a single building [34]. This architecture facilitates disaster recovery, as its applications and data may be replicated on independent infrastructure. With a global deployment, businesses can provide services closer to customers in multiple geographical markets, thereby improving responsiveness. Also, content delivery networks (CDNs) can cache and deliver frequently accessed content, closer to a user's location.

4.5 DevOps and Continuous Delivery

The applications that are packaged, tested and deployed, in the same way, will be supported by the docker and cloud infrastructure for the DevOps and the CI/CD. The discrepancies in development, test and production environments are minimised thanks to Docker containers. Incorporating coding and testing of new image deployments into CI/CD pipelines could

drastically reduce human interference. Infrastructure as Code, or IAC, is a method through which cloud components can be coded, defined, and reproduced [35]. The use of containers helps break up large applications based on the microservices architecture. As a result, organizations are increasingly adopting cloud-native software development practices to stay competitive in changing environments.

4.6 Serverless Computing

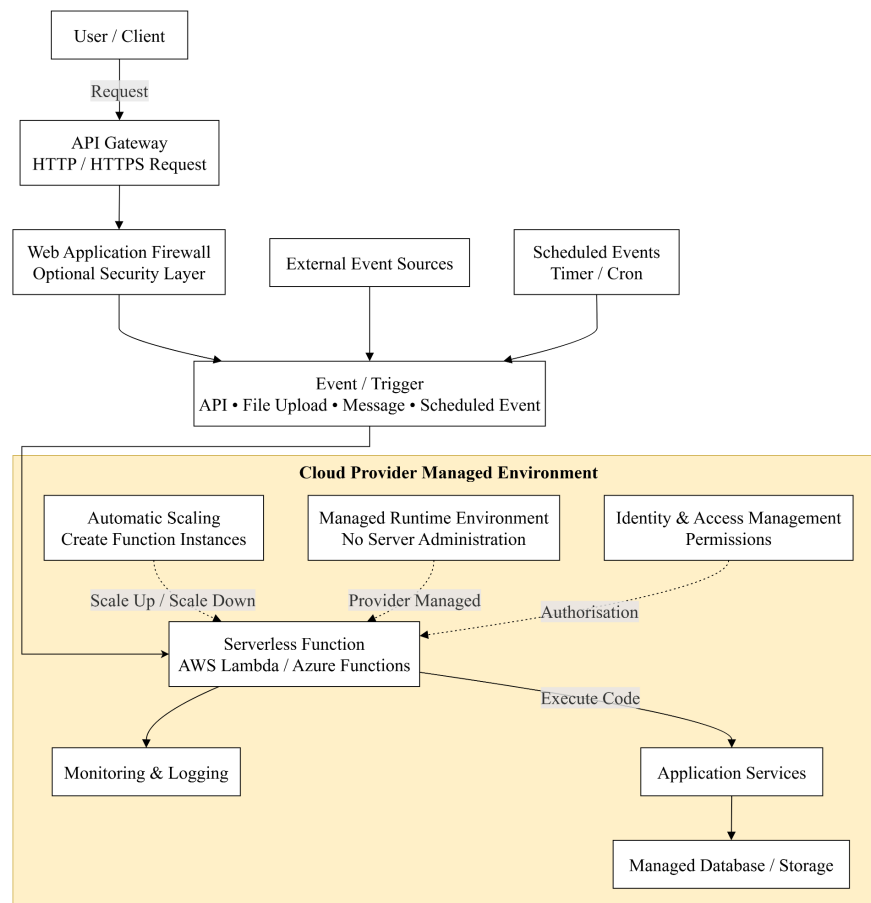


Figure 24: Diagram of Serverless Computing

(Source: Self-developed)

Serverless computing refers to a cloud service model that allows running and deploying program code without having to manage the underlying infrastructure directly. Notably, “serverless” doesn’t imply the physical or virtual server do not exist, it just implies that the onus to manage these servers is with the cloud. The infrastructure, provisioning, scaling, the runtime environment, and availability is managed by the provider while developers mainly focus on the application logic.

Billing	Consumption/execution-based options	Consumption/execution-based options
Triggers	APIs, events and cloud services	APIs, events and cloud services
Use cases	APIs, automation, event processing	APIs, automation, event processing

Table 3: AWS Lambda vs. Azure Functions

(Source: Self-developed)

```
ec2-user@ip-10-20-1-15:~$ cat /etc/os-release | head
NAME="Amazon Linux"
VERSION="2023"
ID="amazon"
ID_LIKE="fedora"
VERSION_ID="2023"
PLATFORM_ID="platform:al2023"
PRETTY_NAME="Amazon Linux 2023.12.28268883"
ANSI_COLOR="0;33"
CPY_NAME="cp2.3:amazon:amazon-linux-2023"
HOME_URL="https://aws.amazon.com/linux/amazon-linux-2023/"
ec2-user@ip-10-20-1-15:~$ docker --version
Docker version 25.0.14, build 9000007
ec2-user@ip-10-20-1-15:~$ sudo systemctl status docker --no-pager
* docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; preset: disabled)
   Active: active (running) since Thu 2026-08-13 03:52:29 UTC; 36s ago
   TriggeredBy: * docker.socket
   Docs: https://docs.docker.com
   Process: 6280 ExecStartPre=/usr/libexec/docker/docker-setup-runtimes.sh (code=exited, status=0/SUCCESS)
   Main PID: 6368 (dockerd)
   Tasks: 9
   Memory: 30.8M
   CPU: 438ms
   CGroup: /system.slice/docker.service
           └─568 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock --default-ulimit nofile=32768:65536

Aug 13 03:52:28 ip-10-20-1-15.ec2.internal systemd[1]: Starting docker.service - Docker Application Container Engine...
Aug 13 03:52:28 ip-10-20-1-15.ec2.internal dockerd[6368]: time="2026-08-13T03:52:28.955646877Z" level=info msg="Starting up"
Aug 13 03:52:29 ip-10-20-1-15.ec2.internal dockerd[6368]: time="2026-08-13T03:52:29.833727278Z" level=info msg="Loading containers: start."
Aug 13 03:52:29 ip-10-20-1-15.ec2.internal dockerd[6368]: time="2026-08-13T03:52:29.487336928Z" level=info msg="Loading containers: done."
Aug 13 03:52:29 ip-10-20-1-15.ec2.internal dockerd[6368]: time="2026-08-13T03:52:29.526146745Z" level=info msg="Docker daemon" commit=dfd8a6 containerd-snapshots=false storage-driver=overlay2 version=25.0.14
Aug 13 03:52:29 ip-10-20-1-15.ec2.internal dockerd[6368]: time="2026-08-13T03:52:29.526322698Z" level=info msg="Daemon has completed initialization"
Aug 13 03:52:29 ip-10-20-1-15.ec2.internal dockerd[6368]: time="2026-08-13T03:52:29.569833415Z" level=info msg="API listen on /run/docker.sock"
Aug 13 03:52:29 ip-10-20-1-15.ec2.internal systemd[1]: Started docker.service - Docker Application Container Engine.
ec2-user@ip-10-20-1-15:~$
ec2-user@ip-10-20-1-15:~$ docker info
Client:
 Version: 25.0.14
 Context: default
 Debug Mode: false
 Plugins:
  buildx: Docker Buildx (Docker Inc.)
   Version: 0.12.1
   Path: /usr/libexec/docker/cli-plugins/docker-buildx
Server:
 ERROR: permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get "http://N2FvarN2FrunN2Fdocker.sock/v1.44/info": dial unix /var/run/docker.sock: connect: permission denied
errors pretty printing info
ec2-user@ip-10-20-1-15:~$ sudo usermod -s0 docker ec2-user
ec2-user@ip-10-20-1-15:~$ getent group docker
docker:x:993:ec2-user
ec2-user@ip-10-20-1-15:~$ newgrp docker
ec2-user@ip-10-20-1-15:~$ getent group docker
docker:x:993:ec2-user
ec2-user@ip-10-20-1-15:~$ docker info
Client:
 Version: 25.0.14
 Context: default
 Debug Mode: false
 Plugins:
  buildx: Docker Buildx (Docker Inc.)
   Version: 0.12.1
   Path: /usr/libexec/docker/cli-plugins/docker-buildx
```

Figure 26: AWS Lambda / Azure Functions Comparison

(Source: Self-developed)

AWS Lambda and Azure Functions, which are serverless Function as Service (FaaS) platforms allow developers to run application code without the bother of managing the underlying server infrastructure. Amazon Web Services offers AWS Lambda, and Microsoft Azure offers Azure Functions. Both platforms have automatic scaling, which allows executing capacity to scale in response to application demand. According to them, consumption-based approaches can lessen the need for having infrastructure always-on, whereas the pricing model depends on execution and resource consumption. Both services allow the use of event-driven workloads, APIs and automation.

4.8 Serverless versus Containerisation

Containerisation provides developers and DevOps teams with better control over the hosting environment than serverless computing does. Containers allow for more flexible management of the application runtime and underlying environment, making it portable in the cloud and on-premises. Nonetheless, organizations are accountable for the management of some element of the container environment. By contrast, in a serverless system, the infrastructure management, provisioning and scaling is done by the cloud provider, thus reducing operational overhead, but potentially increasing lock-in with a particular provider.

<i>Factor</i>	<i>Containers</i>	<i>Serverless</i>
Infrastructure management	Some responsibility	Provider-managed
Control	High	Lower
Portability	High	Provider-dependent
Startup	Fast	Potential cold starts
Scaling	Configured/automatic	Usually automatic
Pricing	Running resources	Execution/invoke
Long-running workload	Suitable	Less suitable
Microservices	Excellent	Excellent
Operational overhead	Moderate	Low

Table 4: Containers vs. Serverless

(Source: Self-developed)

```
Context: default
Debug Mode: false
Plugins:
  builder: Docker Buildx (Docker Inc.)
    Version: 0.12.1
    Path: /usr/libexec/docker/cli-plugins/docker-buildx

Server:
  Containers: 0
    Running: 0
    Paused: 0
    Stopped: 0
  Images: 0
  Server Version: 25.0.16
  Storage Driver: overlay2
  Backing Filesystem: xfs
  Supports d_type: true
  Using metacopy: false
  Native Overlay Diff: true
  userxattr: false
  Logging Driver: json-file
  Cgroup Driver: systemd
  Cgroup Version: 2
  Plugins:
    Volume: local
    Network: bridge host ipvs macvlan null overlay
    Log: awslogs fluentd gcplogs gelf journald json-file local splunk syslog
  Swarm: inactive
  Runtimes: io.containerd.runc.v2 runc
  Default Runtime: runc
  Init Binary: docker-init
  containerd version: 4861b172d3373c36d829fd72c98e47dc5e8
  runc version: de44ad8
  init version: de44ad8
  Security Options:
    seccomp
      Profile: builtin
    cgroupsv2
  Kernel Version: 6.18.39-79.141.amzn2023.x86_64
  Operating System: Amazon Linux 2023.12.20260803
  OSType: linux
  Architecture: x86_64
  CPUs: 2
  Total Memory: 913.2MiB
  Name: ip-10-20-1-15.ec2.internal
  ID: a8c4495e48b4dc9e3a9e4e3d3d1998c33
  Docker Root Dir: /var/lib/docker
  Debug Mode: false
  Experimental: false
  Insecure Registries:
    127.0.0.0/8
  Live Restore Enabled: false

[ec2-user@ip-10-20-1-15 ~]$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS        NAMES
```

```
[ec2-user@ip-10-20-1-15 ~]$ mkdir -p ~/docker-lab/webapp
[ec2-user@ip-10-20-1-15 webapp]$ export APP_NAME="cloud-webapp"
export APP_VERSION="1.0"
export CONTAINER_NAME="${APP_NAME}-container"
export HOST_PORT="8080"
export CONTAINER_PORT="80"
[ec2-user@ip-10-20-1-15 webapp]$ printf '%s\n' \
  "APP_NAME=${APP_NAME}" \
  "APP_VERSION=${APP_VERSION}" \
  "CONTAINER_NAME=${CONTAINER_NAME}" \
  "HOST_PORT=${HOST_PORT}" \
  "CONTAINER_PORT=${CONTAINER_PORT}"
APP_NAME=cloud-webapp
APP_VERSION=1.0
CONTAINER_NAME=cloud-webapp-container
HOST_PORT=8080
CONTAINER_PORT=80
[ec2-user@ip-10-20-1-15 webapp]$ cat > index.html <<'EOF'
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Docker Cloud Lab</title>
</head>
<body>
  <h1>Docker Web Application</h1>
  <p>The application is running inside a Docker container on AWS EC2.</p>
  <p>Containerisation separates the application environment from the underlying host system.</p>
</body>
</html>
EOF
[ec2-user@ip-10-20-1-15 webapp]$ cat index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Docker Cloud Lab</title>
</head>
<body>
  <h1>Docker Web Application</h1>
  <p>The application is running inside a Docker container on AWS EC2.</p>
  <p>Containerisation separates the application environment from the underlying host system.</p>
</body>
</html>
[ec2-user@ip-10-20-1-15 webapp]$ cat > Dockerfile <<'EOF'
FROM nginx:alpine

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

EXPOSE 80
EOF
[ec2-user@ip-10-20-1-15 webapp]$ cat Dockerfile
FROM nginx:alpine

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

EXPOSE 80
[ec2-user@ip-10-20-1-15 webapp]$ cat Dockerfile
```

```

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

EXPOSE 80
[ec2-user@ip-10-20-1-15 webapp]$ cat Dockerfile
FROM nginx:alpine

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

EXPOSE 80
[ec2-user@ip-10-20-1-15 webapp]$ cat Dockerfile
FROM nginx:alpine

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

EXPOSE 80
[ec2-user@ip-10-20-1-15 webapp]$ docker build \
  --tag "${APP_NAME}:${APP_VERSION}" \
  .

[+] Building 2.2s (7/7) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] load metadata for docker.io/library/nginx:alpine
=> [internal] load .dockerignore
=> [internal] transfer context: 28
=> [internal] load build context
=> [internal] transferring context: 510B
=> [1/2] FROM docker.io/library/nginx:alpine@sha256:4a73873bd567c65b79980d6e37998b61f1be6ccc3c23aeac22d2a478c1752
=> resolve docker.io/library/nginx:alpine@sha256:4a73873bd567c65b79980d6e37998b61f1be6ccc3c23aeac22d2a478c1752
=> sha256:4a73873bd567c65b79980d6e37998b61f1be6ccc3c23aeac22d2a478c1752 18.33MB / 18.33MB
=> sha256:10a6ebbbbf1f186a1d6d719372a8399c9a3a5a1f8a0f5b6a30e1a3c1a0d5 2.58MB / 2.58MB
=> sha256:f0ba7779e657c6f8f9a77af4da61a6566f30d6e3721153512842a33779959e 12.32MB / 12.32MB
=> sha256:15afaf1acc2122b5d6e8d4f32dfe64272f898964mac26f6c32123439af26aa 3.89MB / 3.89MB
=> sha256:30c3cf854e44d4f1f1c1b0a8d8c47cfc0b0a8923892012a4253da 1.89MB / 1.89MB
=> sha256:62bec8d7c31c48a19081284d5f7748e6469ec83799450cc1b0c924b142 957B / 957B
=> sha256:1223f816a6aa42c217fc495a83770d6a7a90a43a6d511698ca1a582fc2818d59 627B / 627B
=> sha256:4d9779e6214399208721af63d48a6e8a4f8a8f3c43a1d130882c8 444B / 444B
=> extracting sha256:10a6ebbbbf1f186a1d6d719372a8399c9a3a5a1f8a0f5b6a30e1a3c1a0d5 444B / 444B
=> extracting sha256:15afaf1acc2122b5d6e8d4f32dfe64272f898964mac26f6c32123439af26aa 1.48MB / 1.48MB
=> sha256:4d9779e6214399208721af63d48a6e8a4f8a8f3c43a1d130882c8 28.32MB / 28.32MB
=> sha256:62bec8d7c31c48a19081284d5f7748e6469ec83799450cc1b0c924b142 1.21MB / 1.21MB
=> extracting sha256:30c3cf854e44d4f1f1c1b0a8d8c47cfc0b0a8923892012a4253da 0.18MB / 0.18MB
=> extracting sha256:1223f816a6aa42c217fc495a83770d6a7a90a43a6d511698ca1a582fc2818d59 0.08MB / 0.08MB
=> extracting sha256:62bec8d7c31c48a19081284d5f7748e6469ec83799450cc1b0c924b142 0.08MB / 0.08MB
=> extracting sha256:4d9779e6214399208721af63d48a6e8a4f8a8f3c43a1d130882c8 0.08MB / 0.08MB
=> extracting sha256:10a6ebbbbf1f186a1d6d719372a8399c9a3a5a1f8a0f5b6a30e1a3c1a0d5 0.08MB / 0.08MB
=> extracting sha256:15afaf1acc2122b5d6e8d4f32dfe64272f898964mac26f6c32123439af26aa 0.08MB / 0.08MB
=> extracting sha256:4d9779e6214399208721af63d48a6e8a4f8a8f3c43a1d130882c8 0.08MB / 0.08MB
=> [2/2] COPY index.html /usr/share/nginx/html/index.html
=> exporting to image
=> exporting layers
=> writing image sha256:f0b265ab8a9eb88d8d1d5eb8c38a5e93f67fa8e617695b15be424c36468
=> naming to docker.io/library/cloud-webapp:1.0
[ec2-user@ip-10-20-1-15 webapp]$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
cloud-webapp 1.0 fab265ab8a9e 5 seconds ago 62.4MB
[ec2-user@ip-10-20-1-15 webapp]$ mkdir -p ~/docker-lab/webapp
cd ~/docker-lab/webapp
[ec2-user@ip-10-20-1-15 webapp]$

```

```

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

EXPOSE 80
[ec2-user@ip-10-20-1-15 webapp]$ docker build \
  --tag "${APP_NAME}:${APP_VERSION}" \
  .

[+] Building 0.2s (7/7) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] load metadata for docker.io/library/nginx:alpine
=> [internal] load .dockerignore
=> [internal] transfer context: 28
=> [internal] load build context
=> [internal] transferring context: 510B
=> [1/2] FROM docker.io/library/nginx:alpine@sha256:4a73873bd567c65b79980d6e37998b61f1be6ccc3c23aeac22d2a478c1752
=> CACHED [2/2] COPY index.html /usr/share/nginx/html/index.html
=> exporting to image
=> exporting layers
=> writing image sha256:f0b265ab8a9eb88d8d1d5eb8c38a5e93f67fa8e617695b15be424c36468
=> naming to docker.io/library/cloud-webapp:1.0
[ec2-user@ip-10-20-1-15 webapp]$ docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
cloud-webapp 1.0 fab265ab8a9e 3 minutes ago 62.4MB
[ec2-user@ip-10-20-1-15 webapp]$ cat Dockerfile
FROM nginx:alpine

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

EXPOSE 80
[ec2-user@ip-10-20-1-15 webapp]$ ~/docker-lab/webapp
bash: /home/ec2-user/docker-lab/webapp: is a directory
[ec2-user@ip-10-20-1-15 webapp]$ CONTAINER_ID=$(docker run \
  --detach \
  --publish "${HOST_PORT}:${CONTAINER_PORT}" \
  --name "${CONTAINER_NAME}" \
  "${APP_NAME}:${APP_VERSION}")

echo "CONTAINER_ID=${CONTAINER_ID}"
CONTAINER_ID=37ba4642fbc224d63a5c0e8dca83e32a9d34eb1fb251bc8589bd049cf4ef23a
[ec2-user@ip-10-20-1-15 webapp]$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
37ba4642fbc2 cloud-webapp:1.0 "/docker-entrypoint..." 6 seconds ago Up 6 seconds 0.0.0.0:8080->80/tcp, :::8080->80/tcp cloud-webapp-container
[ec2-user@ip-10-20-1-15 webapp]$
[ec2-user@ip-10-20-1-15 webapp]$ curl http://localhost:8080
<!DOCTYPE html>
<html lang=en>
<head>
  <meta charset=UTF-8>
  <meta name=viewport content=width=device-width, initial-scale=1.0>
  <title>Docker Cloud Lab</title>
</head>
<body>
  <h1>Docker Web Application</h1>
  <p>The application is running inside a Docker container on AWS EC2.</p>
  <p>Containerisation separates the application environment from the underlying host system.</p>
</body>
</html>
[ec2-user@ip-10-20-1-15 webapp]$ docker inspect "${CONTAINER_NAME}" \
  --format 'Name:({.Name}) Status:({.State.Status}) Image:({.Config.Image}) Ports:({(json .NetworkSettings.Ports)})'
Name:cloud-webapp-container Status:running Image:cloud-webapp:1.0 Ports:["80/tcp":[{"HostIp":"0.0.0.0","HostPort":"8080"}],{"HostIp":"","HostPort":"8080"}]]
[ec2-user@ip-10-20-1-15 webapp]$ docker logs "${CONTAINER_NAME}"
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, Will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
20-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf

```

Figure 27: Containers versus Serverless

(Source: Self-developed)

A critical analysis of containers indicates that they are more suited to long-running applications, complex workloads and with more control over runtime. The benefits of serverless architecture have the biggest impact for event-driven workloads that are short-lived

and variable. The automatic scaling and reduced need for infrastructure administration are very beneficial. However, serverless can create provider lock-in and has execution constraints, while containers are more operationally heavy. The decision should be based on the characteristics of the workload, the portability requirement, cost, control and the organisational skill rather than being superior.

4.9 Critical Evaluation of Cloud Adoption Drivers

With the many advantages offered by the cloud, including cost efficiency, flexibility, scalability, availability, automation, DevOps and serverless, Usage by IT firms is increasing. Containers offer enhanced control, portability, and consistency, making them ideal for complex long-running workloads. Serverless can lessen your operational burden and provides convenient automatic scaling for event-driven applications, but it may add more vendor lock-in. The workloads that require dedicated operating system environments still benefit from strong isolation, flexibility and control in virtual machines. In the meantime, managed cloud services reduce infrastructure management, allowing companies to focus on business applications. The result may be superior performance from cloud providers and better outcomes from existing computing clusters.

5. Video Demonstration

Video Link : <https://youtu.be/xDePr0PzJO8>

6. Discussion and Critical Evaluation

Cloud alternatives shouldn't be considered one and the same as each tackle's different infrastructure/application requirements. Traditional virtual machines (VMs) deliver excellent isolation, OS control and flexibility, making them well suited for legacy apps, special environments and workloads needing dedicated OS-level configs.

Containers enable lightweight, portable, quick to deploy, and efficient use of resources. These tools are ideal for microservices, DevOps pipelines and applications that need consistent environments across dev, test and prod. Cloud load balancing spreads the load across containers or VMs. Using healthy resources improves operational capacity or gives better performance.


```

"Options": {},
"Labels": {}
}
}

[ec2-user@ip-10-20-1-15 webapp]$ export TEST_CONTAINER_NAME="${APP_NAME}-tester"
[ec2-user@ip-10-20-1-15 webapp]$ docker run \
--detach \
--name "$TEST_CONTAINER_NAME" \
--network "$NETWORK_NAME" \
alpine \
sleep infinity
31aae889ebc47700257972051089e3fe32f9d1d63df6d22c364fa7227ef4
[ec2-user@ip-10-20-1-15 webapp]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
31aae889ebc         alpine             "sleep infinity"    6 seconds ago       Up 5 seconds       0.0.0.0:8080->80/tcp, :::8080->80/tcp    cloud-webapp-tester
72a2e29f1c4         cloud-webapp:1.0   "/docker-entrypoint..."  6 minutes ago       Up 4 minutes              cloud-webapp-container
[ec2-user@ip-10-20-1-15 webapp]$ docker exec "$TEST_CONTAINER_NAME" \
| wget -qO- "http://$(cat /dev/urandom | tr -dc 'a-z0-9' | fold -w 40 | xargs | shasum | cut -d ' ' -f 1)"
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Docker Cloud Lab</title>
</head>
<body>
<h1>Docker Web Applications</h1>
<p>The application is running inside a Docker container on AWS EC2.</p>
<p>Containerisation separates the application environment from the underlying host system.</p>
</body>
</html>
[ec2-user@ip-10-20-1-15 webapp]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
31aae889ebc         alpine             "sleep infinity"    28 seconds ago     Up 27 seconds       0.0.0.0:8080->80/tcp, :::8080->80/tcp    cloud-webapp-tester
72a2e29f1c4         cloud-webapp:1.0   "/docker-entrypoint..."  6 minutes ago     Up 4 minutes              cloud-webapp-container
[ec2-user@ip-10-20-1-15 webapp]$ docker network inspect "$NETWORK_NAME" \
| _format '{{range .Containers}}{{.Name}} {{.IPv4Address}}{{"\n"}}{{end}}'
cloud-webapp-tester 172.18.0.3/16
cloud-webapp-container 172.18.0.2/16
[ec2-user@ip-10-20-1-15 webapp]$ docker exec "$TEST_CONTAINER_NAME" \
| wget -qO- "http://$(cat /dev/urandom | tr -dc 'a-z0-9' | fold -w 40 | xargs | shasum | cut -d ' ' -f 1)"
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Docker Cloud Lab</title>
</head>
<body>
<h1>Docker Web Application</h1>
<p>The application is running inside a Docker container on AWS EC2.</p>
<p>Containerisation separates the application environment from the underlying host system.</p>
</body>
</html>
[ec2-user@ip-10-20-1-15 webapp]$ docker network create
docker volume create
docker run
docker network connect
"docker network create" requires exactly 1 argument.
See 'docker network create --help'.

Usage: docker network create [OPTIONS] NETWORK

Create a network
4c82959f1c4a457c44ece0f5930f6d8e0fa7faa7e9ce20f889a8460282011
"docker run" requires at least 1 argument.

```

Figure 28: Cloud Technology Coding

An Application Gateway is most suitable when intelligent Layer 7 traffic management functions are needed, like URL-based routing or host-based routing or TLS termination or web security control integration. When minimizing management of infrastructure is more important than control of run time and portability an event driven short duration unpredictable serverless workloads are to be preferred.

There is no universally superior technology, therefore. Where isolation and control are important, VMs are ideal; where portable and efficient application deployment is key, containers will do; where load balancing and Auto Scaling is important, then why not; where sophisticated management of HTTP/HTTPS traffic is required, application gateways can help; serverless is great for event-driven applications.

7. Conclusion and Recommendation

Cloud computing offers a flexible platform for creating and delivering software applications. It allows efficient sharing of physical infrastructure in data centers using virtualisation technology and builds on that with containerisation. The practical implementation of Docker highlighted the significance of how dockerfiles, images, containers, volumes, and networks combine to execute applications. At the cloud infrastructure level, load balancing and ability (scalability) improve availability, performance and resource utilization while application gateways provide intelligent application-level traffic management and further security capabilities. Cloud adoption will continue to be fuelled by cost efficiency, flexibility, elasticity, globally availability and DevOps automation.

Organizations have to do applications on Load Balancers with Auto Scaling to achieve resilience. The load balancer routes requests to healthy instances and can also create instances based on demand. Organisations have to focus on a hybrid cloud-native architecture for long-running services that require portability across event-driven workloads.

References

1. Gartner (2024). Gartner Forecasts Worldwide Public Cloud End-User Spending to Total \$723 Billion in 2025. Available from <https://www.gartner.com/en/newsroom/press-releases/2024-11-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-total-723-billion-dollars-in-2025>. Accessed on 13.8.2026
2. Arogundade, O.R. and Palla, K., 2023. Virtualization revolution: Transforming cloud computing with scalability and agility. IARJSET. <https://www.academia.edu/download/115018998/iarjset.2023.pdf>
3. Sartakov, V.A., Vilanova, L., Eysers, D., Shinagawa, T. and Pietzuch, P., 2022. {CAP-VMs}:{Capability-Based} isolation and sharing in the cloud. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22) (pp. 597-612). <https://www.usenix.org/system/files/osdi22-sartakov.pdf>
4. Thyagaturu, A.S., Shantharama, P., Nasrallah, A. and Reisslein, M., 2022. Operating systems and hypervisors for network functions: A survey of enabling technologies and research studies. IEEE Access, 10, pp.79825-79873. <https://ieeexplore.ieee.org/iel7/6287639/9668973/09844719.pdf>
5. Martin, E., 2022. Virtualization and containerization: a new concept for data center management to optimize resources distribution. <https://unitesi.unive.it/bitstream/20.500.14247/14491/1/815635-84509.pdf>
6. Moreau, D., Wiebels, K. and Boettiger, C., 2023. Containers for computational reproducibility. Nature Reviews Methods Primers, 3(1), p.50. <https://par.nsf.gov/servlets/purl/10501743>
7. Muzumdar, P., Bhosale, A., Basyal, G.P. and Kurian, G., 2024. Navigating the docker ecosystem: A comprehensive taxonomy and survey. arXiv preprint arXiv:2403.17940. <https://arxiv.org/pdf/2403.17940>
8. Tara, A. and Khan, T.U., 2025, April. A comparative study of hardware-based and software-based secure virtualization technologies. In Computer Science On-line Conference (pp. 69-95). Cham: Springer Nature Switzerland. https://link.springer.com/chapter/10.1007/978-3-032-04581-2_6
9. Oikonomou, E., 2025. Development of web-based content management software using open technologies.

- https://dspace.lib.ntua.gr/xmlui/bitstream/handle/123456789/62026/Thesis_Oikonomou_Eleftherios.pdf?sequence=1.
10. Doan, T.P. and Jung, S., 2022. DAVS: Dockerfile analysis for container image vulnerability scanning. *Computers, Materials and Continua*, 72(1), pp.1699-1711. <https://www.sciencedirect.com/org/science/article/pii/S1546221822010955>
 11. Madhavapeddy, A., Scott, D.J. and Cormack, J., 2026. A Decade of Docker Containers. *Communications of the ACM*, 69(3), pp.56-65. <https://dl.acm.org/doi/pdf/10.1145/3761803>
 12. Kane, S.P. and Matthias, K., 2023. Docker: Up & Running. " O'Reilly Media, Inc.". https://imaster.academy/contenidos-tematicos/2022/ciclo4/Programacion/Ciclo%204A/Semana_6/01_Material_Apoyo/Docker.pdf
 13. Kane, S.P. and Matthias, K., 2023. Docker: Up & running. " O'Reilly Media, Inc.". https://imaster.academy/contenidos-tematicos/2022/ciclo4/Programacion/Ciclo%204A/Semana_6/01_Material_Apoyo/Docker.pdf
 14. Carpenter, J. and McFadin, P., 2022. Managing cloud native data on Kubernetes: architecting cloud native data services using open source technology. " O'Reilly Media, Inc.". <https://books.google.com/books?hl=en&lr=&id=G3ufEAAQBAJ&oi=fnd&pg=PT18&dq=Databases+and+data+must+survive+application-container+replacement+stateful+application+particularly+beneficial+using+this+technology&ots=85UluVgIYD&sig=WDsHS9qDauX2UTTHtDn7y3Midg0>
 15. Shelke, C.J., Raza, A., Verma, S., Suthar, A.G. and Tanveer, A.S., 2026. Containerization and Docker. BR Publications. <https://books.google.com/books?hl=en&lr=&id=asfYEQAAQBAJ&oi=fnd&pg=PA1&dq=Container-to-container+communication+refers+to+the+process+of+exchanging+data+between+multiple+application+components,+including+web+servers+and+databases,+over+a+Docker+network&ots=rSppxf2nnS&sig=1RWrwFKT-IV97MwTd340kXvqaXg>
 16. Wendt, A. and Schüppstuhl, T., 2022, January. Proxying ROS communications—enabling containerized ROS deployments in distributed multi-host environments. In 2022 IEEE/SICE International Symposium on System Integration (SII) (pp. 265-270). IEEE. <https://arxiv.org/pdf/2201.01613>
 17. Chen, L., Xia, Y., Ma, Z., Zhao, R., Wang, Y., Liu, Y., Sun, W. and Xue, Z., 2022. SEAF: A Scalable, Efficient, and Application-independent Framework for container security

- detection. *Journal of Information Security and Applications*, 71, p.103351. <https://www.sciencedirect.com/science/article/pii/S221421262200196X>
18. Oikonomou, E. and Rouskas, A., 2026. Workload-aware edge node orchestration and dynamic resource scaling in MEC. *Future Internet*, 18(4), p.184. <https://www.mdpi.com/1999-5903/18/4/184>
 19. Trump, B.D., Mitoulis, S.A., Argyroudis, S., Kiker, G., Palma-Oliveira, J., Horton, R., Pescaroli, G., Pinigina, E., Trump, J. and Linkov, I., 2025. Threat-agnostic resilience: Framing and applications. *International Journal of Disaster Risk Reduction*, 124, p.105535. <https://www.sciencedirect.com/science/article/pii/S2212420925003590>
 20. Zafer, M. and Aslam, S., 2024. Integrating Dynamic Load Balancing for Scalable Network Optimization in Recommender Systems. https://www.researchgate.net/profile/Saleem-Aslam-9/publication/385383454_Integrating_Dynamic_Load_Balancing_for_Scalable_Network_Optimization_in_Recommender_Systems/links/67226a0edb208342dede659f/Integrating-Dynamic-Load-Balancing-for-Scalable-Network-Optimization-in-Recommender-Systems.pdf
 21. Selby, K., Banatvala, N., Bovet, P. and Cornuz, J., 2023. Screening and health checks for NCD prevention and control. <https://iris.unil.ch/bitstreams/21bc48c6-a4fe-4971-93c4-c93d6a6417ef/download>
 22. Shang, Y., 2024. Prevention and detection of DDOS attack in virtual cloud computing environment using Naive Bayes algorithm of machine learning. *Measurement: Sensors*, 31, p.100991. <https://www.sciencedirect.com/science/article/pii/S2665917423003276>
 23. Malazi, H.T., Chaudhry, S.R., Kazmi, A., Palade, A., Cabrera, C., White, G. and Clarke, S., 2022. Dynamic service placement in multi-access edge computing: A systematic literature review. *Ieee Access*, 10, pp.32639-32688. <https://ieeexplore.ieee.org/iel7/6287639/9668973/09738624.pdf>
 24. Mishra, A., 2024. Balancing performance and cost in EC2 instance selection for energy-efficient cloud computing. https://assets-eu.researchsquare.com/files/rs-5376742/v1_covered_b5b139f7-cd0b-464a-8cd5-d79bc2d2b899.pdf?c=1732531629
 25. Tan, J., Zhang, J., Tang, Z., Xiao, X., Jiang, B., Zhao, J. and Li, K., 2025. ASSG: Enhanced workload balancing via adaptive state scheduling granularity approach for stateful distributed stream processing. *ACM Transactions on Architecture and Code Optimization*, 22(4), pp.1-26. <https://dl.acm.org/doi/pdf/10.1145/3776583>

26. Russo Russo, G., Cardellini, V. and Lo Presti, F., 2023. Hierarchical auto-scaling policies for data stream processing on heterogeneous resources. *ACM transactions on autonomous and adaptive systems*, 18(4), pp.1-44. <https://dl.acm.org/doi/pdf/10.1145/3597435>
27. Hu, W., Lu, J., Chen, J., Li, F., Shi, C., Shi, X., Wang, J. and Huo, W., 2026. SpecWeaver: End-to-End HTTP API Specification Inference across Multi-layer Routing in Production Web Services. *Proceedings of the ACM on Software Engineering*, 3(FSE), pp.4437-4458. <https://dl.acm.org/doi/pdf/10.1145/3808202>
28. Mabotha, E., Mabunda, N.E. and Ali, A., 2025. A dockerized approach to dynamic endpoint management for RESTful application programming interfaces in internet of things ecosystems. *Sensors*, 25(10), p.2993. <https://www.mdpi.com/1424-8220/25/10/2993>
29. Kesavan, B., 2024. Mastering Amazon EC2: Unravel the complexities of EC2 to build robust and resilient applications. Packt Publishing Ltd. <https://books.google.com/books?hl=en&lr=&id=A0ACEQAAQBAJ&oi=fnd&pg=PP1&dq=+if+application+traffic+spikes,+a+group+of+EC2+instances+can+be+launched+to+offer+greater+processing+capacity+through+AWS+Auto+Scaling+groups&ots=0dbBGs4Rd4&sig=mKOASq8iw4mVbW8WRc1FQKLnmFw>
30. Aqeel, I., Khormi, I.M., Khan, S.B., Shuaib, M., Almusharraf, A., Alam, S. and Alkhaldi, N.A., 2023. Load balancing using artificial intelligence for cloud-enabled internet of everything in healthcare domain. *Sensors*, 23(11), p.5349. <https://www.mdpi.com/1424-8220/23/11/5349>
31. Reddy, G.S., Kesavan, M.E., Jadhav, M.S. and Ram, C.S., 2025. Mastering Cloud Computing: Strategies for the Digital Age. Academic Guru Publishing House. https://books.google.com/books?hl=en&lr=&id=Ib5qEQAAQBAJ&oi=fnd&pg=PA1&dq=+Cloud+providers+such+as+AWS+and+Microsoft+Azure+offer+pay-as-you-go+models+of+consumption,+letting+organisations+avoid+buying+infrastructure+to+meet+occasional+peaks+in+demand&ots=t21pILILO9&sig=NL7C0-WkE_2rioxXgoo-18jhNPk
32. Banala, S., 2022. Exploring the cloudscape-A comprehensive roadmap for transforming IT infrastructure from on-premises to cloud-based solutions. *International Journal of Universal Science and Engineering*, 8(1), pp.35-44. https://www.academia.edu/download/119813862/Exploring_the_Cloudscape_A_Comprehensive.pdf
33. Bhalla, J.B., 2025. Understanding cloud growth strategies: Elasticity vs. scalability in modern cloud infrastructure. *International Journal of Scientific Research in Computer*

- Science, Engineering and Information Technology, 11(2), pp.1311-1319.
<https://www.academia.edu/download/121916880/CSEIT25112497.pdf>
34. Vankayalapati, R.K. and Dubey, S., 2022. Ai clusters and elastic capacity management: Designing systems for diverse computational demands. Available at SSRN, 5115889.
<https://www.academia.edu/download/121076650/1.pdf>
 35. Chintale, P., 2023. DevOps Design Pattern: Implementing DevOps best practices for secure and reliable CI/CD pipeline (English Edition). Bpb Publications.
<https://reference-global.com/download/chapter/9789355519924/10.0000/9789355519924-001.pdf>
 36. Mkhize, A., Mokhothu, K.D., Tshikhotho, M. and Thango, B.A., 2025. Evaluating the impact of cloud computing on SME performance: A systematic review. Businesses, 5(2), p.23. <https://www.mdpi.com/2673-7116/5/2/23>
 37. Rehan, H., 2024. Revolutionizing America's cloud computing the pivotal role of AI in driving innovation and security. Journal of Artificial Intelligence General science (JAIGS) ISSN, pp.3006-4023.
https://www.researchgate.net/profile/Hassan-Rehan/publication/379414015_Revolutionizing_America's_Cloud_Computing_the_Pivotal_Role_of_AI_in_Driving_Innovation_and_Security/links/66b05b212361f42f23b4aa8a/Revolutionizing-Americas-Cloud-Computing-the-Pivotal-Role-of-AI-in-Driving-Innovation-and-Security.pdf

