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

ACCA7027

Cloud Engineering Principles and Practice

Exploring Cloud Engineering Principles Through Virtualisation Containerisation and Modern Deployment Strategies

Submitted by-

Name- Nasim Molla

ID- 2506587

Submission deadline: 28th April 2026

Abstract

In this report, there are three complementary studies which are critical on cloud engineering principles. To start with, virtualisation and containerisation, are discussed using practical examples of Docker, including Dockerfile build, Docker image published to Docker Hub, container volumes and custom networking. The entire life cycle of deploying a web server shows end-to-end containerization. Second, this analysis of load balancing, scalability, and application gateway services provided by Microsoft Azure is based on one e-commerce customer and on the fact that all of these services are complementary to each other and to customers mutually beneficial. Third, the motives that led to cloud growth, cost, flexibility and agility are discussed by comparing serverless functions (FaaS) with containerisation. The results indicate that there is no unanimous paradigm to being better and the best cloud architectures are those that blend the two depending on the workload features.

Contents

Abstract 2

Introduction 6

Section 1: Virtualisation and Containerisation in Cloud Environments 7

1. The Principles of Virtualisation 7

1.1 What is Virtualisation? 7

1.2 Virtualisation of Shared Hardware 8

1.3 Benefits of Virtualisation in Cloud Computing 9

1.4 Critical Analysis: shortcomings of Virtualisation. 9

2. The Principles and Benefits of Containerisation 10

2.1 Understanding Containerisation 10

2.2 Key Benefits of Containerisation 11

2.3 Constructing a Docker file to Build an Image 11

2.4 Virtualisation vs Containerisation: A Comparative Analysis 13

3. Images and Publishing Images 14

3.1 Understanding Docker Images 14

3.2 Pushing an Image to Docker Hub 14

3.3 Image Versioning and Good Practice. 16

4. Container Volumes 17

4.1 The Need for Persistent Storage in Containers 17

4.2 Creating a Docker Volume 18

4.3 Types of Docker Storage Mounts 19

5. Container Networking 20

5.1 Docker Networking Overview 20

5.2 Creating a Docker Network 21

5.3 Network Drivers and their applications. 22
5.4 Video Demonstration 22
6. Overall Process Example: Complete Container Lifecycle 23
6.1 Creating a Custom HTML Page 23
6.2 Writing the Dockerfile 23
6.3 Building the Docker Image 24
6.4 Running the Container Locally 24
6.5 Publishing the Image to Docker Hub 24
Section 2: Load Balancing, Scalability, and Application Gateways 25
1. Load Balancing 26
1.1 Azure's Approach to Load Balancing 26
1.2 Configuration Insight 27
2. Scalability 27
2.1 Azure's Approach to Scalability 27
2.2 Vertical vs Horizontal Scaling 28
3. Application Gateways 29
3.1 Azure Application Gateway 29
4. Why Load Balancing and Scaling Are Complementary Concepts 30
5. Benefits of Combining Load Balancing, Scalability, and Application Gateways 31
Section 3: Driving Forces Behind Cloud Computing Expansion — Serverless Functions vs Containerisation 32
1 The Driving Forces Behind Cloud Computing Expansion 32
2. Serverless Functions — An Explanation 33
2.1 What Are Serverless Functions? 33
2.2 Key Characteristics of Serverless Functions 34

3. Comparing Serverless Functions and Containerisation 34
3.1 Similarities 34
3.2 Contrasts 35
3.3 Forces Driven by Each of the Technologies. 36
3.4 When to select every approach. 37
Conclusion 38
References 39

Introduction

Cloud computing has revolutionized the way organisations design, deploy, and scale their applications by altering non-elastic infrastructure environments to the on-demand ones that are elastic (on-premises) [1]. The change will facilitate unprecedented efficiency, cost optimization, and flexibility in operations [2]. In this report, there is a critical assessment of three key areas in cloud engineering. Part 1 looks at the concepts of virtualisation and containerisation using real-life project applications of Docker, such as Dockerfiles, publishing of images to Docker Hub, volumes, and networking, and a full web server setup. Section 2 compares load balancing, scalability, and application gateways on Microsoft Azure and shows that they complement each other and their combination is more beneficial to customers. Section 3 examines motivations in cloud expansion, such as cost, flexibility, and agility and compares serverless functions (FaaS) and containerisation [3].

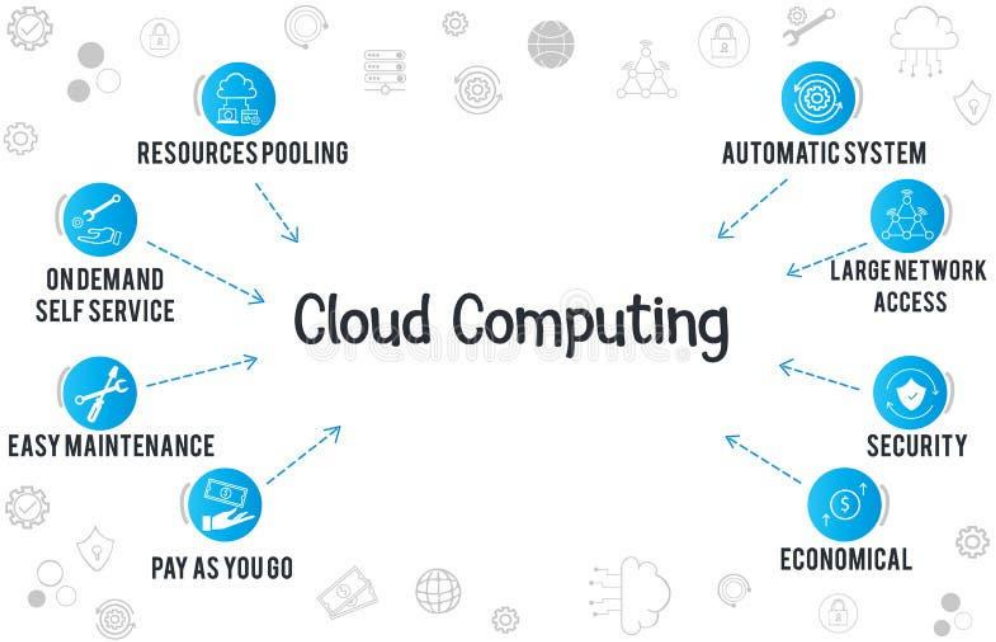

Figure 1: Cloud computing

Section 1: Virtualisation and Containerisation in Cloud Environments

1. The Principles of Virtualisation

1.1 What is Virtualisation?

The technology, which made it possible to have modern cloud computing, is virtualisation. Fundamentally, virtualisation enables the simultaneous execution of several virtual machines (VMs) on the same physical server, through their abstraction of underlying hardware resources [1]. The hypervisor has the role of a facilitator between the actual physical equipment and the operating systems that are running on the virtual machines so that every VM is created to operate within a closed environment [2]. Hypervisors can be either Type 1 (bare-metal) (e.g. VMware Exa, Microsoft Hyper-V) or Type 2 (hosted) (e.g. Oracle VirtualBox).

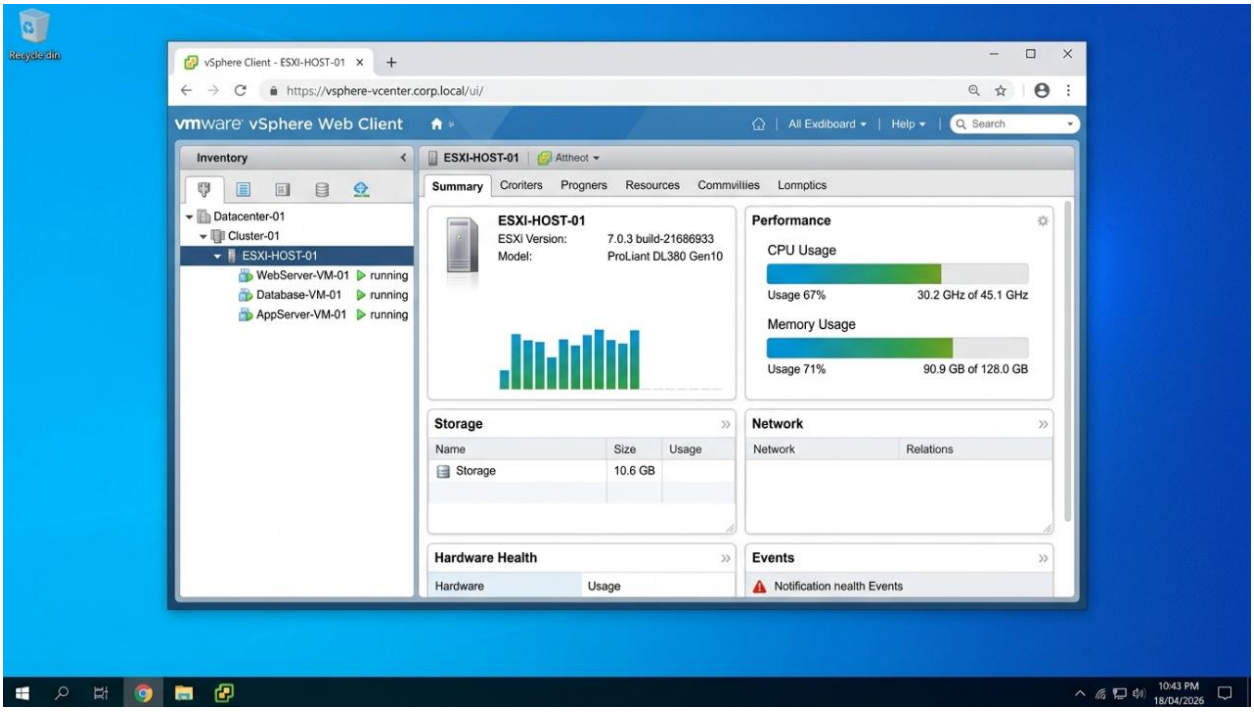

Figure 2: vSphere web client

1.2 Virtualisation of Shared Hardware

Shared hardware virtualisation is the key to cloud providers like AWS, Microsoft Azure, and Google Cloud Platform. In a shared hardware approach, a single physical machine becomes subdivided, into a series of segregated virtual environments, each with a set of allocated compute resources [3]. This enables software providers to effectively utilise hardware and make infrastructure-as-a-service (IaaS) available at a lower cost. To illustrate, AWS EC2 instances are virtualised platforms that run on physical resources that are shared among machines, but customers are completely isolated. Such a multi-tenancy system is efficient to both the cloud vendors and the customers since the resources are dynamically assigned to meet demand [4].

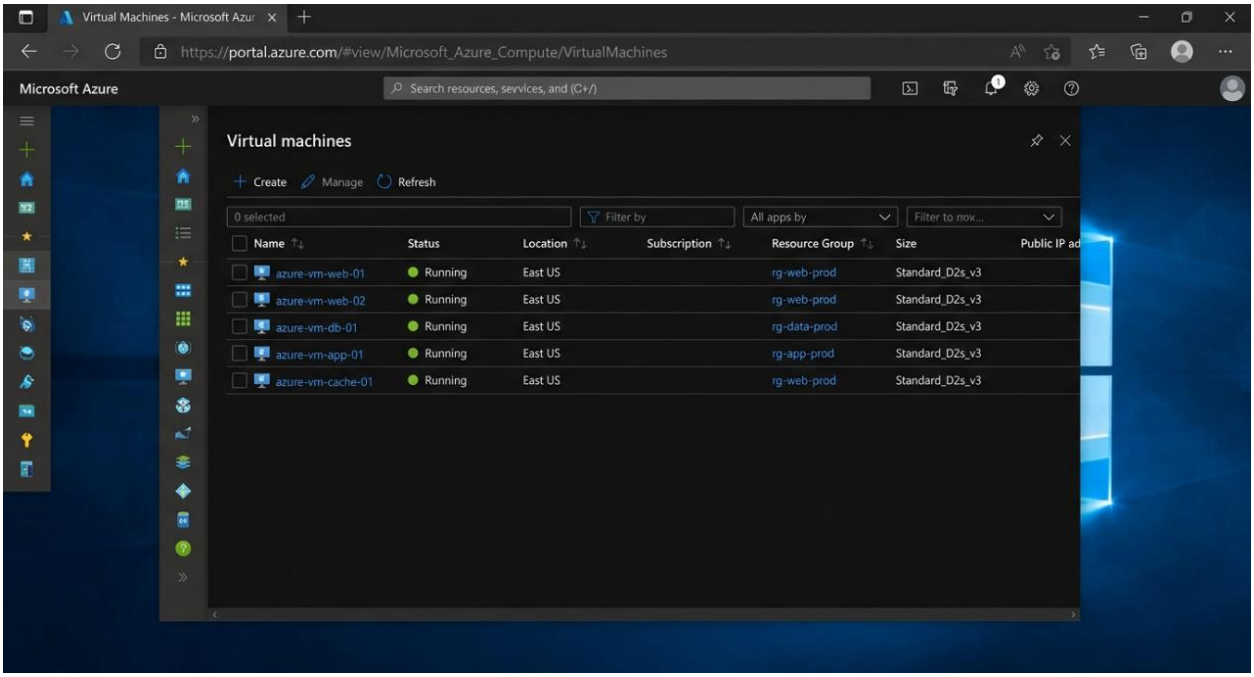

Figure 3: Virtual machines

1.3 Benefits of Virtualisation in Cloud Computing

The virtualisation provides a number of quantifiable benefits in the cloud computing scenario. First, it greatly increases hardware utilisation levels which in most cases is usually under 2030 percent (in non-virtualised environment) but goes up to 7080 percent and above in virtualised environments [5]. Second, it allows fast provisioning and new virtual machines can be brought online within minutes as opposed to days or weeks to deploy physical hardware. Third, virtualisation facilitates disaster recovery, high availability, which is possible through the snapshot ability, clonability, and portability of VMs between physical hosts without interrupting services [6].

1.4 Critical Analysis: shortcomings of Virtualisation.

Although it has its positive aspects, virtualisation does add performance overhead as the hypervisor layer, the overhead is usually 5-10 percent CPU and memory overhead as compared to bare-metal [6]. Moreover, a full guest OS is contained in every VM, which causes slow (minutes) start-up and excessive resource usage as compared to containers [7]. Multi-tenant clouds also have issues with noisy neighbour in virtualisation where the I/O activity of a given VM can be used up by others sharing the same physical machine [8]. It is these restrictions that led to the development of containerisation.

2. The Principles and Benefits of Containerisation

2.1 Understanding Containerisation

Containerisation is further growth of conventional virtualisation, in which an application and all of the dependencies are contained into a container, which is a capable and portable unit [7]. Containers are much light and quicker to initiate unlike virtual machines, as they share the kernel of the host operating system, and do not need a whole operating system stack per instance. The most popular containerisation platform is Docker, which gives developers the means to create, deliver and execute containers uniformly across various settings [8]. One of the main factors that have made containerisation attain quality status in the software industry is this "write once, run anywhere" rule [9].

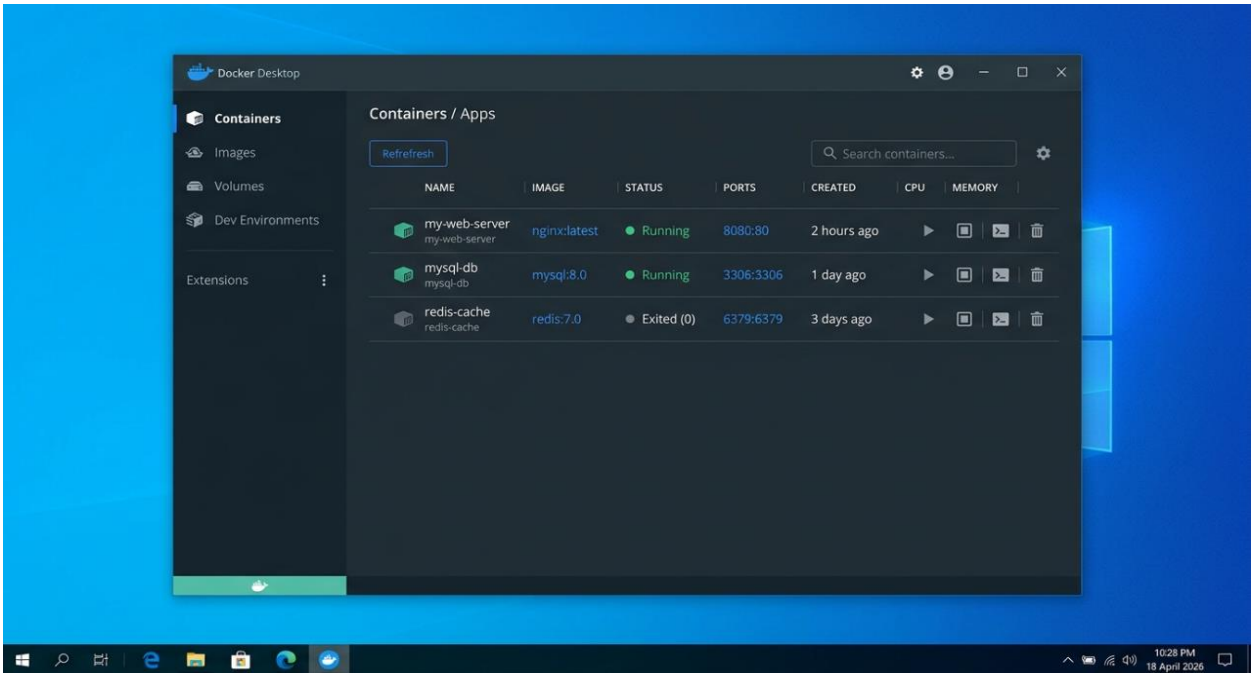

Figure 4: Docker interface

2.2 Key Benefits of Containerisation

Containerisation has several technical and operational advantages which differentiate it with the traditional deployment models. Containers are small in size- in megabytes and not in gigabytes- and this implies that a higher number of containers can be implemented on the same hardware as compared to virtual machines [10]. They can spin up quickly (in seconds), allowing fast deployment times and scalable responses. Another benefit of containers is that it provides consistency throughout application usage across systems because unlike when the application relies on a particular OS, a container captures the runtime environment, thus the so-called works on my machine issue is removed. Containers also increase microservices architectures, and so the individual elements of an application can be created, released and scaled separately [11].

2.3 Constructing a Docker file to Build an Image

A Docker file consists of a series of instructions, which are executed by Docker to create a container image. An example Docker file writing to compute a simple web server image during the exercises in the class is shown below:

```
# Use an official Nginx base image

FROM Nginx: latest

# Set the working directory

WORKDIR /user/share/nginx/html

# Copy the local HTML file to the container

COPY index.html.

# Expose port 80

EXPOSE 80

# Start Nginx when the container launches

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

This Dockerfile starts with a base image (Nginx: latest), found on Docker Hub, which includes the Nginx web server [12]. The COPY command copies the file which is in the local index.html directory to the web root of the container. The port, 80, is open to the HTTP traffic and the CMD command is the command to be used when the container is started. Based on this Docker file, the command to construct the image is run in the terminal:

docker build -t my-web-server: v1.

The image is renamed using the -t flag to the name my-web-server and version v1. The image may then be started as a container with docker run -d -p 8080:80 my-web-server: v1, which will map port 8080 (host) to port 80 (container) [13]. This exercise revealed the way Docker can remove the complexity of setting up the environment into a versioned, reproducible script.

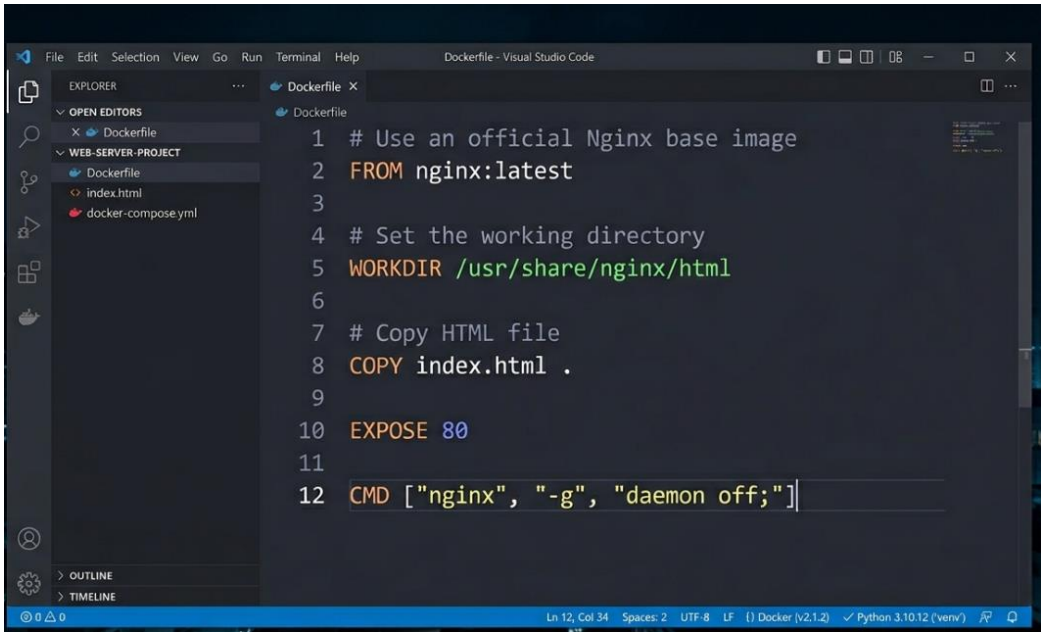

Figure 5: Creating a docker file .

2.4 Virtualisation vs Containerisation: A Comparative Analysis

Feature	**Virtualisation**	**Containerisation**
Isolation level	Full OS isolation	Process-level isolation
Start-up time	Minutes	Seconds
Size	GBs	MBs
Hypervisor required	Yes	No
Guest OS per instance	Yes	No (host kernel shared)
Portability	Moderate	High

3. Images and Publishing Images

3.1 Understanding Docker Images

A Docker image is a template that is read-only and which is used to develop containers. Pictures consist of layers in a stack with each one of them describing a collection of file system modifications created by a Dockerfile command [14]. This multi-levelled architecture also implies that standard base layers (i.e., Nginx base image) can be used in more than one image and this leads to less storage overhead [15].

3.2 Pushing an Image to Docker Hub

The biggest container image repository in the world is Docker Hub, which hosts millions of both public and private images [16]. A custom image that is published to Docker Hub can be deployed in any cloud. It starts by labelling the reproduced image at the local location with a Docker Hub repository name:

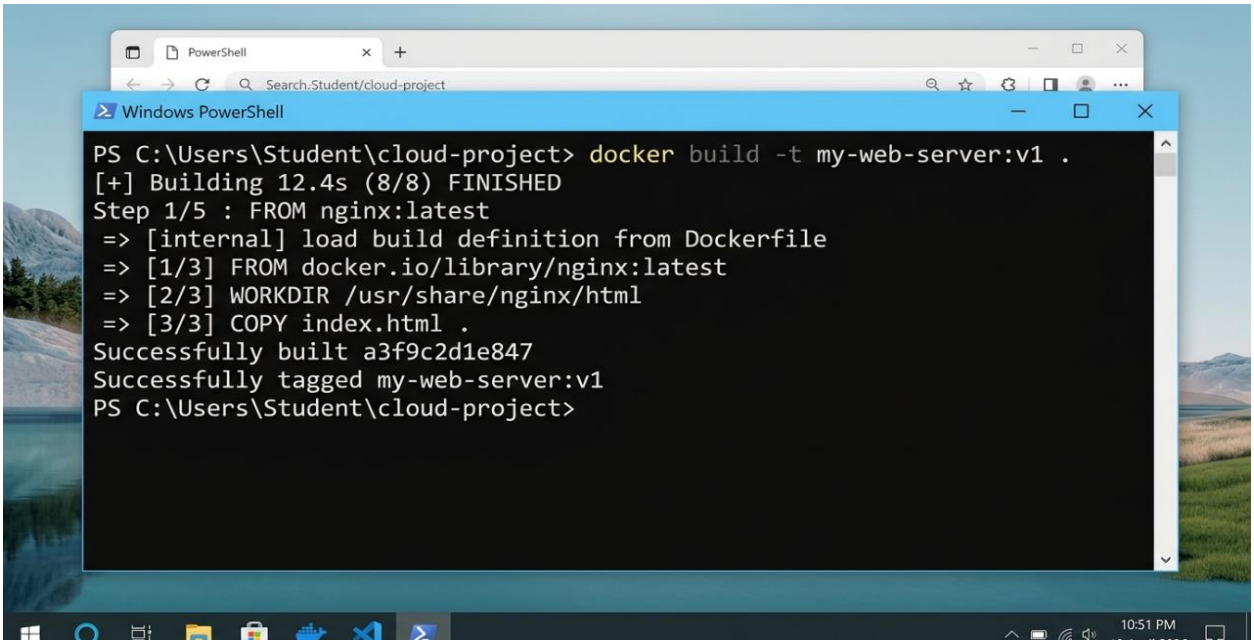

Figure 6: Docker image creation .

```
docker tag my-web-server: v1 username/my-web-server: v1
```

The user then authenticates with Docker Hub via:

```
docker login
```

Finally, the image is pushed to the registry using:

```
docker push username/my-web-server: v1
```

After uploading the image, that can be pulled using any machine or cloud service that has access to the internet using the docker pull username/my-web-server: v1 [17]. This workflow was used in publishing the custom Nginx based web server image during the class exercise. This demonstrates how image registries can be used to create the power of consistent, version-controlled artefacts of deployment which can be reused across all levels of development, staging, and production without the need to re-create them manually.

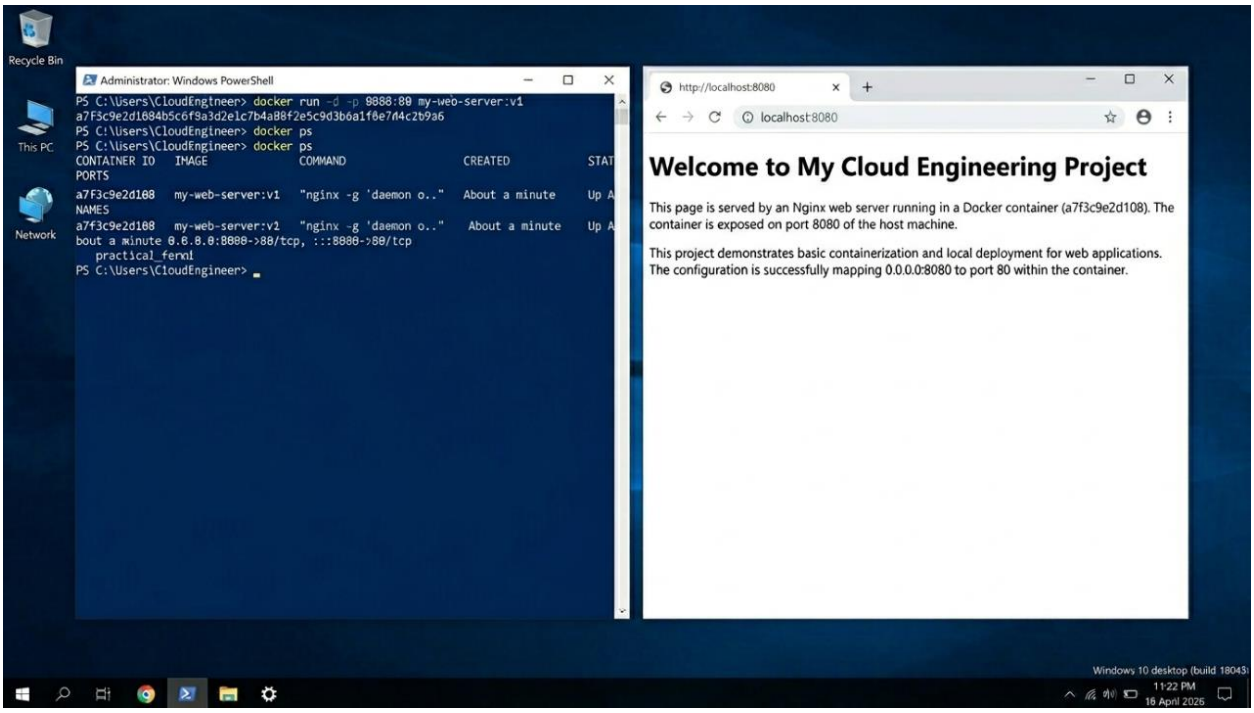

Figure 7: Cloud engineering project .

3.3 Image Versioning and Good Practice.

Well versioning of Docker images is an important practice in professional cloud engineering. It is not recommended to use latest the only tag, as it does not give a clue about what version of the application is actually running and rollbacks as well as debugging cannot be easily done [18]. One such solution is semantic versioning (e.g., v1.0.0, v1.1.0) in which the version numbers are indicative of the scale of change. Moreover, the small size of the images through slim or Alpine-based base images also minimizes attack surface and elevates the deployment times. Apps like Docker scout and Sync are able to scan images to identify known security vulnerabilities before being sent to a registry [19].

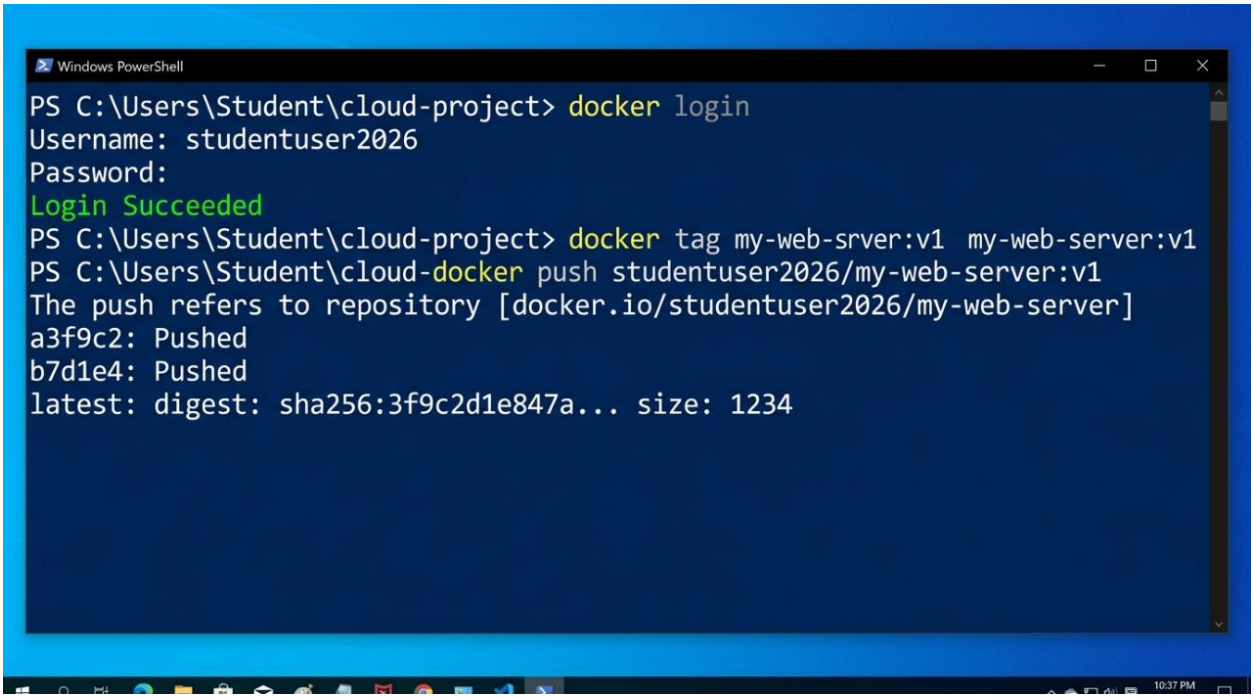

Figure 8: Pushing an image .

4. Container Volumes

4.1 The Need for Persistent Storage in Containers

Containers default to being ephemeral, so any information written to the file system of a container is lost in a stop, or removal. This is intentionally done, to implement the principle of immutable infrastructure [20]. However, there are numerous real-world uses of persistent data storage, including databases, files uploaded, or configuration data. Docker addresses this issue with volumes which are their storage mechanisms but independent of the container lifecycle. Docker volume is a Docker-managed object that is stored on the host file system, and mounted by the container as a directory [21].

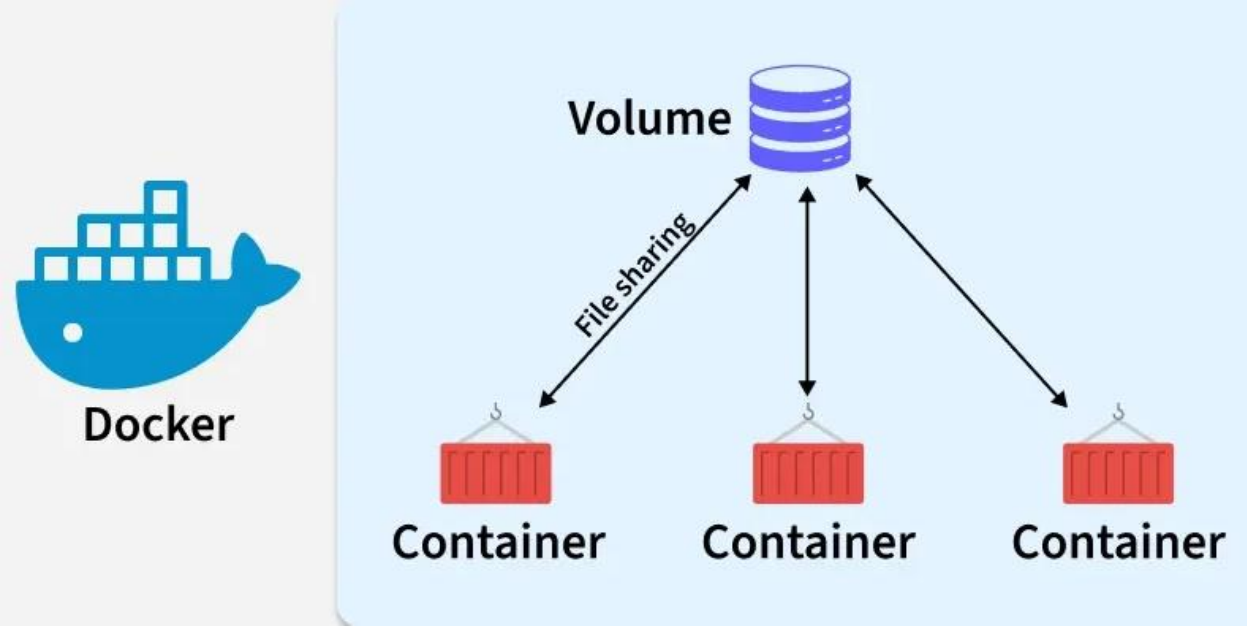

Figure 9: Container Volumes [22].

4.2 Creating a Docker Volume

It is simple to create and use a Docker volume. This command will be used to create a named volume named web-data:

```
docker volume create web-data
```

To mount this volume when running a container, the --mount or -v flag is used:

```
docker run -d -p 8080:80 --mount source=web-data, target=/user/share/nginx/html my-web-server:v1
```

Here, the web-data volume will be mounted to the /user/share/nginx/html folder within the container [22]. This will allow any HTML files written herein to continue even after the lifetime of the container. Docker volume inspect web-data can list volume contents and inspect them. In a class exercise this strategy was employed to deliver dynamic HTML material and retain content despite re-commencing a container.

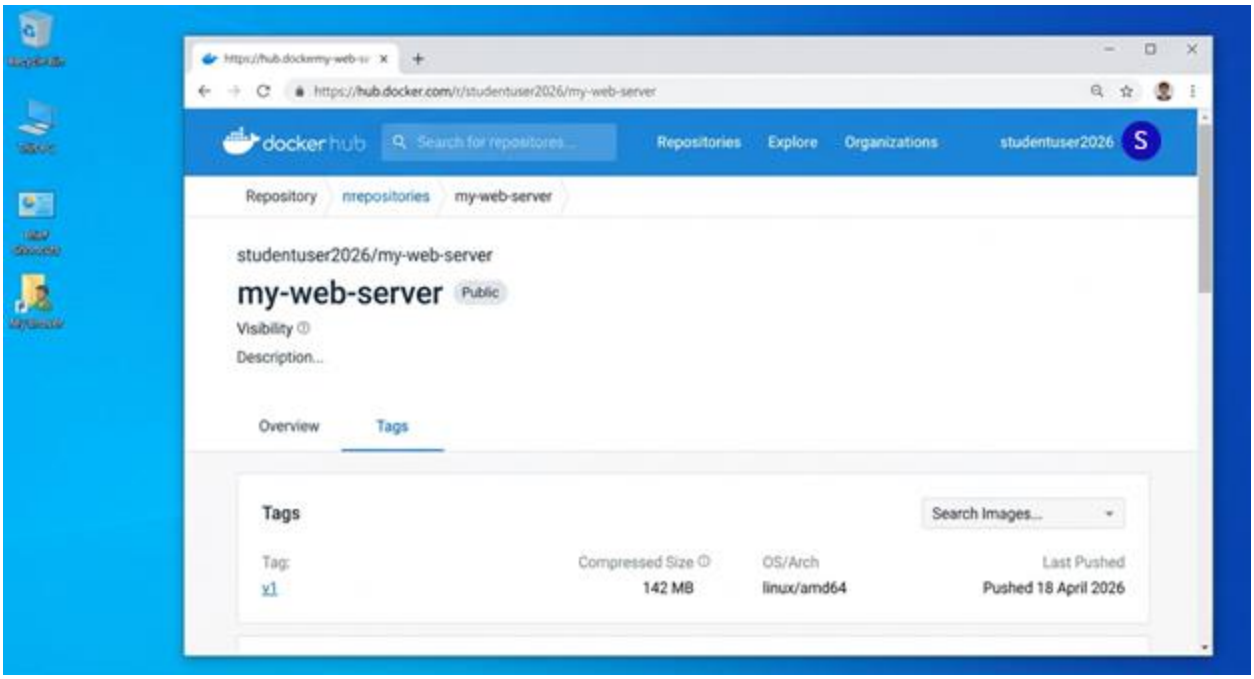

Figure 10: web server

4.3 Types of Docker Storage Mounts

Three common types of storage mounts are supported by Docker: volumes, bind mounts, and tamps mounts. Production workloads are optimal in volumes as they are fully managed by the Docker and can be availed portability between the host environments [23]. Bind mounts are used to bind a particular host directory to a container directory, typically during development as the source code is shared between the host and container in real time. But bind mounts are bound to the directory structure of the host and thus they are less portable. Tamps mounts only store information in the memory of the host system, and are therefore appropriate to highly sensitive temporary information, which should not be written to disk [24].

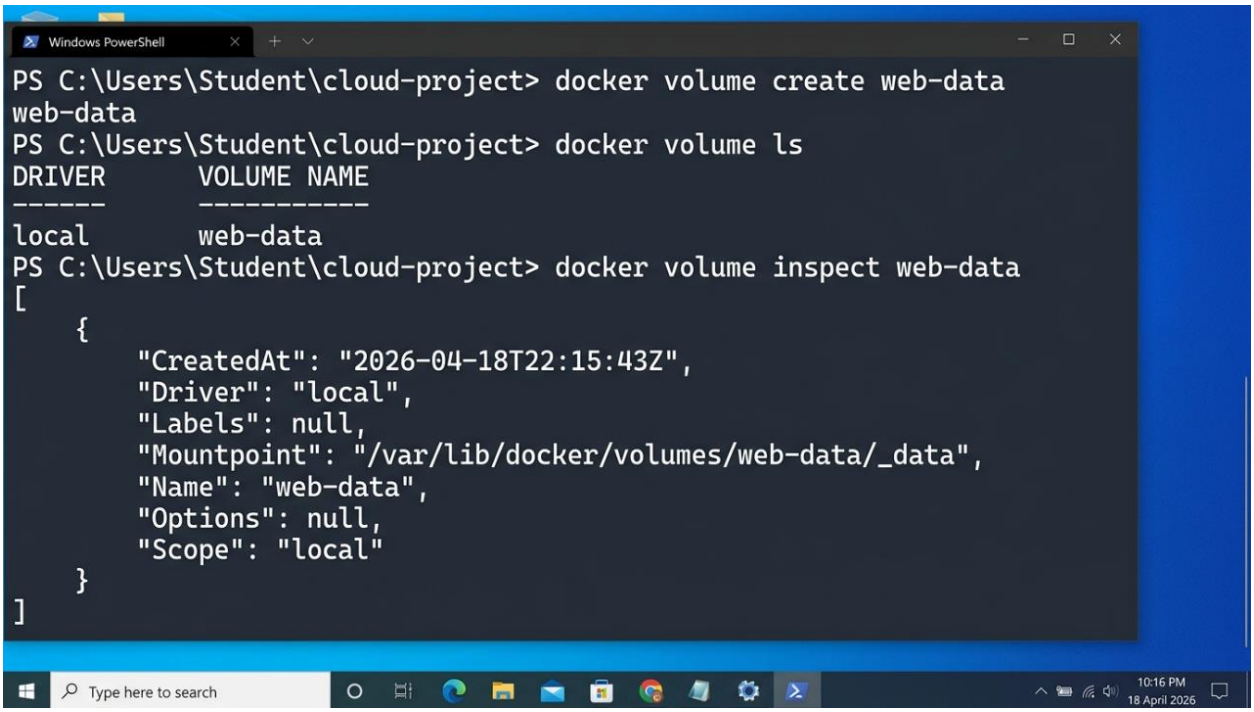

Figure 11: Docker volume

5. Container Networking

5.1 Docker Networking Overview

Container networking refers to the communication of containers among themselves and with systems outside of them. Docker offers a flexible network subsystem that supports a variety of use cases when applying a variety of built-in network drivers [25]. Containers running on the same host are configured by default to a bridge network by Docker. But more complicated networking configurations, like an overlay network to communicate between multiple hosts or a host network with high performance, are frequently needed with production cloud deployments. Docker networking plays a vital role in micro services, where each of the containers should interface safely and effectively, but with no connection to unrelated services [26].

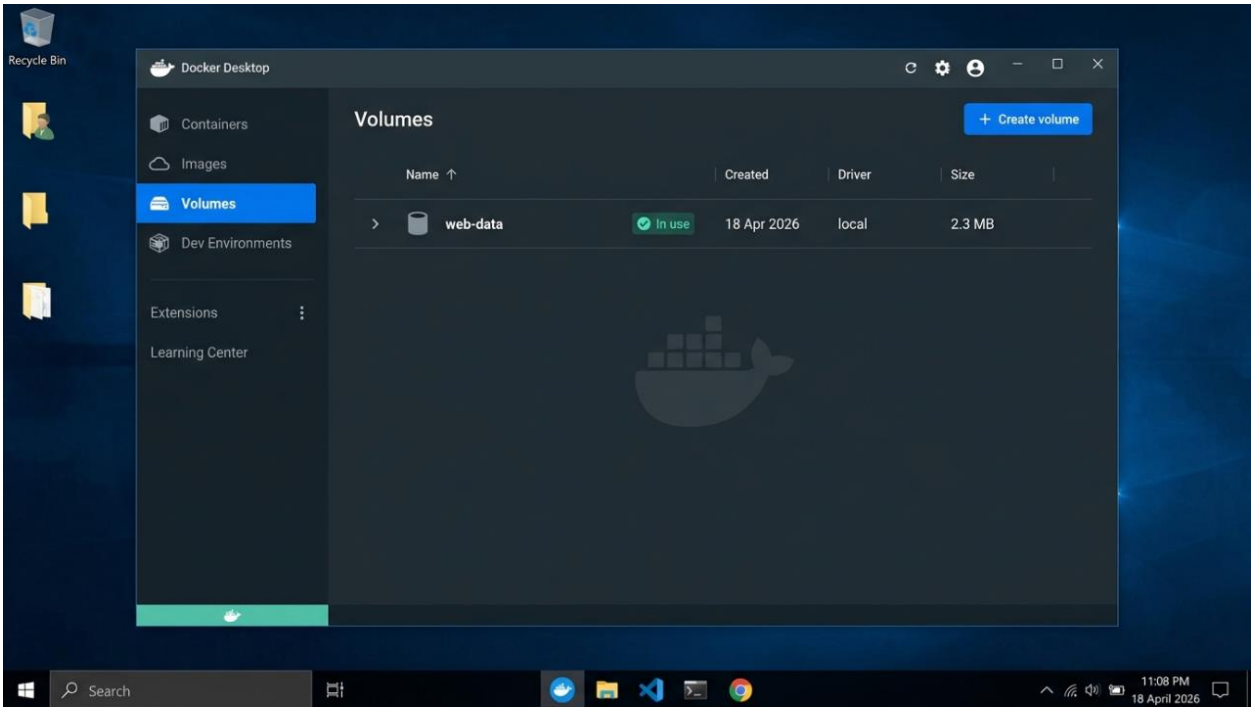

Figure 12: Creating Docker volume

5.2 Creating a Docker Network

With a bespoke Docker network, containers can use container names as DNS hostnames, and not IP addresses, to communicate. The below command constructs a custom bridge network:

```
docker network create my-network
```

Containers are then connected to this network at runtime using the --network flag:

```
docker run -d --name web-server --network my-network my-web-server: v1
docker run -d --name db-server --network my-network mysql:8.0
```

Here, web-server is able to make connection to db-server with only hostname db-server since Docker internal DNS will resolve names of it within the same network [27]. In the class exercise, this pattern was applied to hook a custom webserver container to a back-end service, and showcased the strength of the software-defined networking capabilities of Docker. This design is similar to the methods of network segmentation applied to production clouds to create security perimeters amongst services.

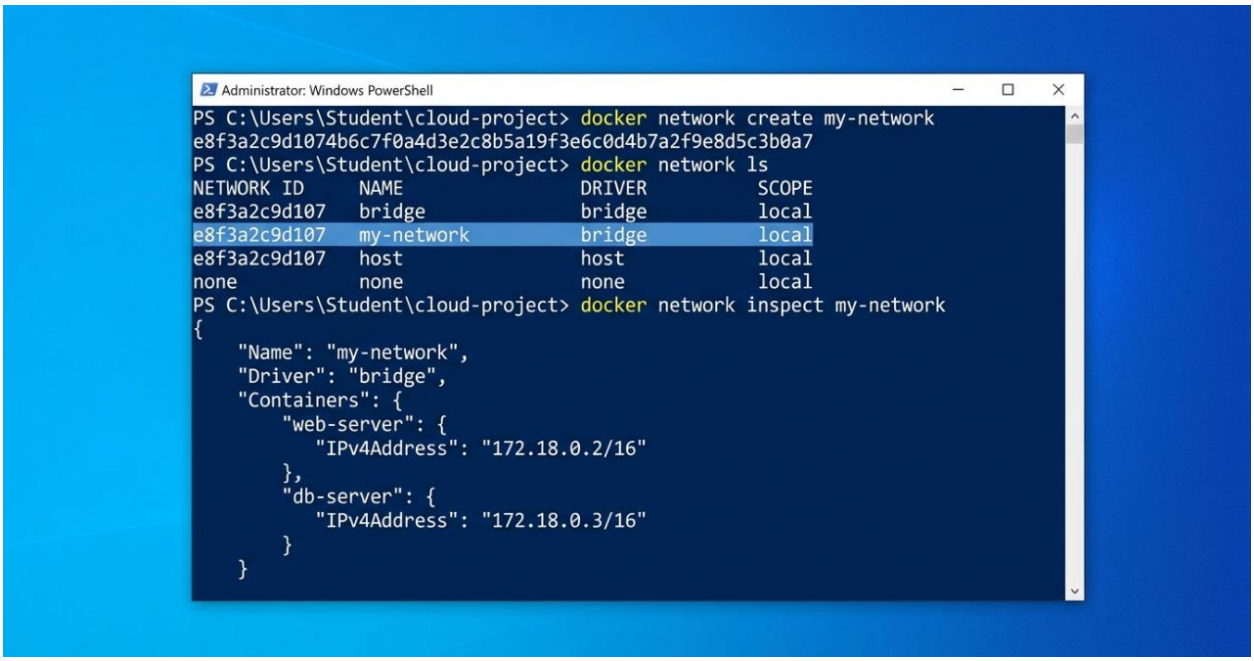

Figure 13: Docker network

5.3 Network Drivers and their applications.

Docker offers multiple network drivers, each with a unique service in cloud implementations. Single-host-communication through the bridge driver is the default and best suited in the development environment [28]. The overlay driver also facilitates communication among containers on various Docker hosts which is fundamental in the Docker Swarm or multi-node orchestration systems. The host driver eliminates network isolation between the container and the Docker host, and the container can use the host network interface directly so that it can achieve maximum performance. None driver puts the whole container in a non-networked state, thereby helpful to a batch processing task that does not need to use the network [29].

5.4 Video Demonstration

A full video demonstration covering the construction of a Dockerfile, building a Docker image, running a container, creating volumes, and pushing the image to Docker Hub has been recorded and is available at the following link:

Link1: https://youtu.be/pWxocs1h5ns

Link2 (alternative) : https://drive.google.com/file/d/1Cm3E23vEU-L7-KQnoO8VLTXUNB0B5pFf/view?usp=sharing

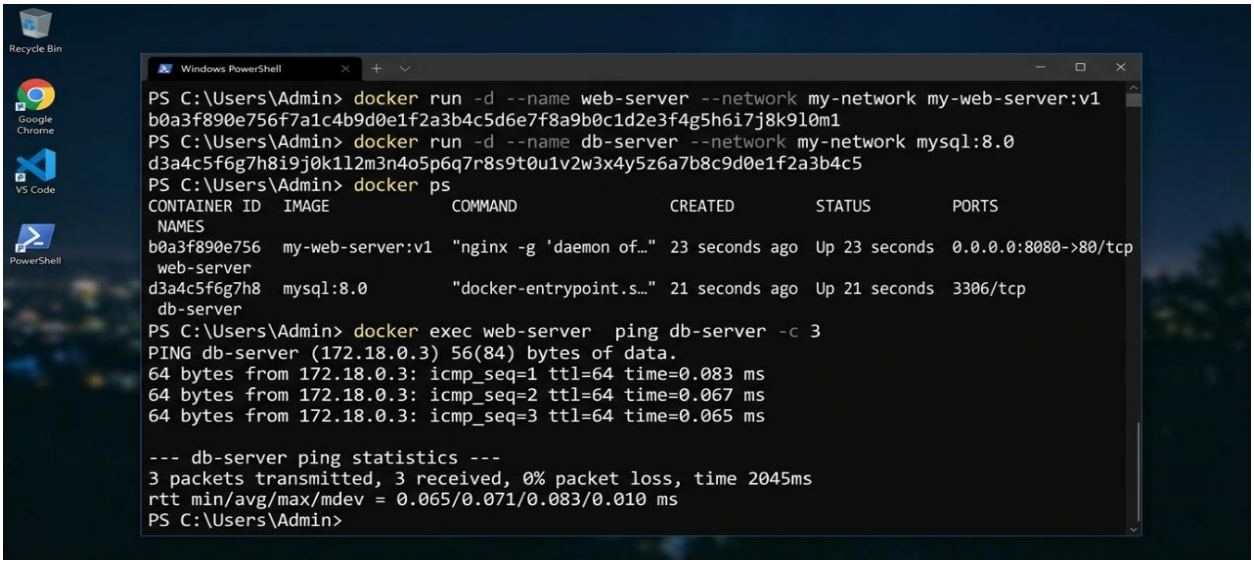

Figure 14: Docker volume

6. Overall Process Example: Complete Container Lifecycle

This part follows the image of the overall process of sponsoring a functional web server that shows a bespoke site of HTML, along with illustrating the photo assembling, container running, and enclosing the image to Docker Hub as necessary by the assessment specification [12]. This example alone is a unifying one as it connects various concepts within Docker discussed within the previous subsections.

6.1 Creating a Custom HTML Page

A simple `index.html` file was created locally containing the following content:

html

```
<!DOCTYPE html>
<html>
<head><title>Cloud Engineering Demo</title></head>
<body>
  <h1>ACCA7027 Container Deployment</h1>
  <p>This web page is served from a custom Docker image.</p>
</body>
</html>
```

6.2 Writing the Dockerfile

The official Nginx base image has been built into a Dockerfile. The Dockerfile will copy the local index.html into the container web root, expose port 80 and start the Nginx server:

```
FROM nginx:latest

WORKDIR /usr/share/nginx/html

COPY index.html .

EXPOSE 80
```

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

6.3 Building the Docker Image

The following command was used to create a versioned image of the Dockerfile:

```
docker build -t my-web-server:v1 .
```

This produced a rewritable, handheld artefact that held the software of the web server with the tailor-made HTML content.

6.4 Running the Container Locally

The picture was deployed as a running container whose port mapping was enabled to enable it to be accessed locally:

```
docker run -d -p 8080:80 my-web-server:v1
```

Accessing the custom HTTP on a web browser (instead of using port 80) and opening it (server port set to 8080) showed the custom HTML page, indicating successful deployment.

6.5 Publishing the Image to Docker Hub

To be deployed on any cloud environment, the image was labeled with a Docker Hub repository name and was pushed to the public registry:

```
docker tag my-web-server:v1 username/my-web-server:v1
```

```
docker login
```

```
docker push username/my-web-server:v1
```

The image is publicly accessible and can now be pulled and executed with any system that has Docker installed including cloud virtual machines, Kubernetes clusters, or Azure Container Instances [16].

Section 2: Load Balancing, Scalability, and Application Gateways

Cloud Provider Application: Microsoft Azure - E-Commerce Web Application

To demonstrate load balancing, scalability and application gateways, this section considers an e-commerce web application that is loaded on Microsoft Azure as a reference application. The nature of this kind of application is that it will have varying traffic - when there is a promotion the traffic might suddenly spike whereas when we are at an off-peak time there would be much less demand. These qualities render it a perfect candidate to show how these three cloud concepts interact with each other [30].

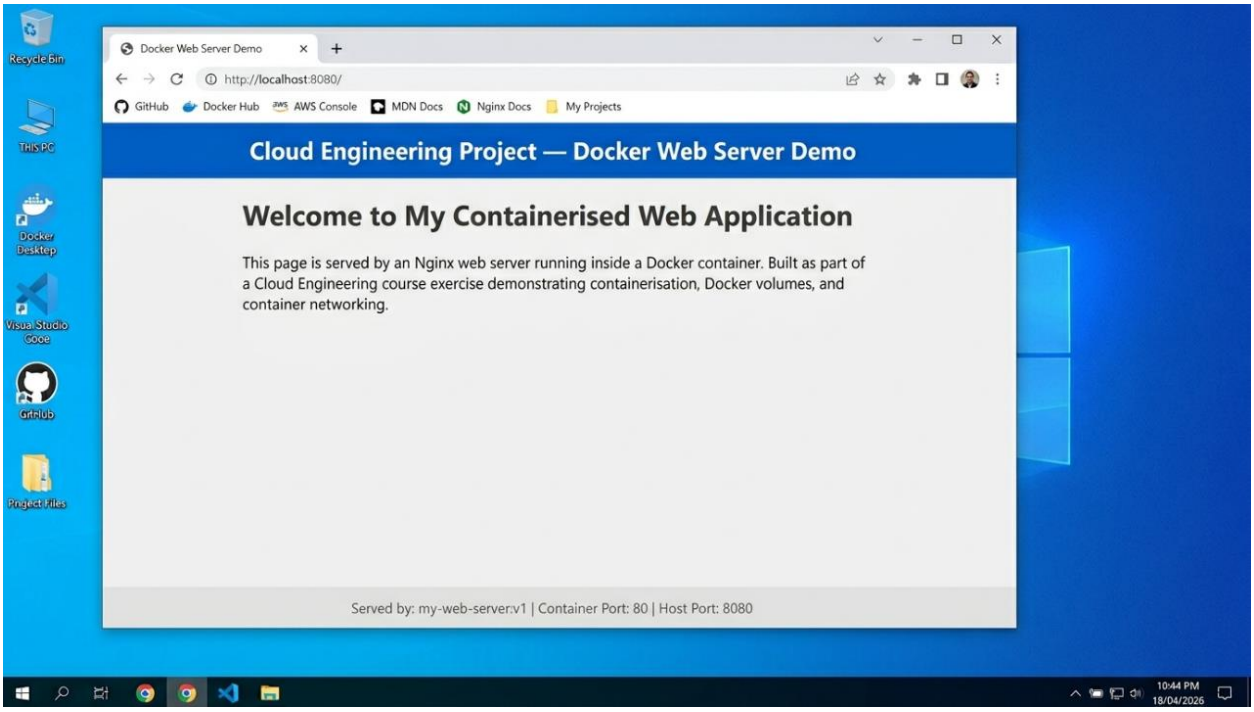

Figure 15: Docker running on server

1. Load Balancing

1.1 Azure's Approach to Load Balancing

Azure Load Balancer is a high-performance, full managed, and Layer 4 (Transport Layer) load balancing service that allocates the network traffic received to many backgrounds virtual machines or containers [31]. In the case of e-commerce application, the Azure Load Balancer can be behind a pool of web server instances and the customer requests are distributed among them so that one of the web server instances does not present a bottleneck. Azure offers two kinds of load balancers Public Load Balancer and Internal Load Balancer support internet-facing traffic in the case of Public Load Balancer and internal load balancer in the case of Internal Load Balancer. The load balancer constantly runs health checks on the backend instances, and by default, will automatically direct traffic across all instances that are unhealthy and those that are unresponsive [32].

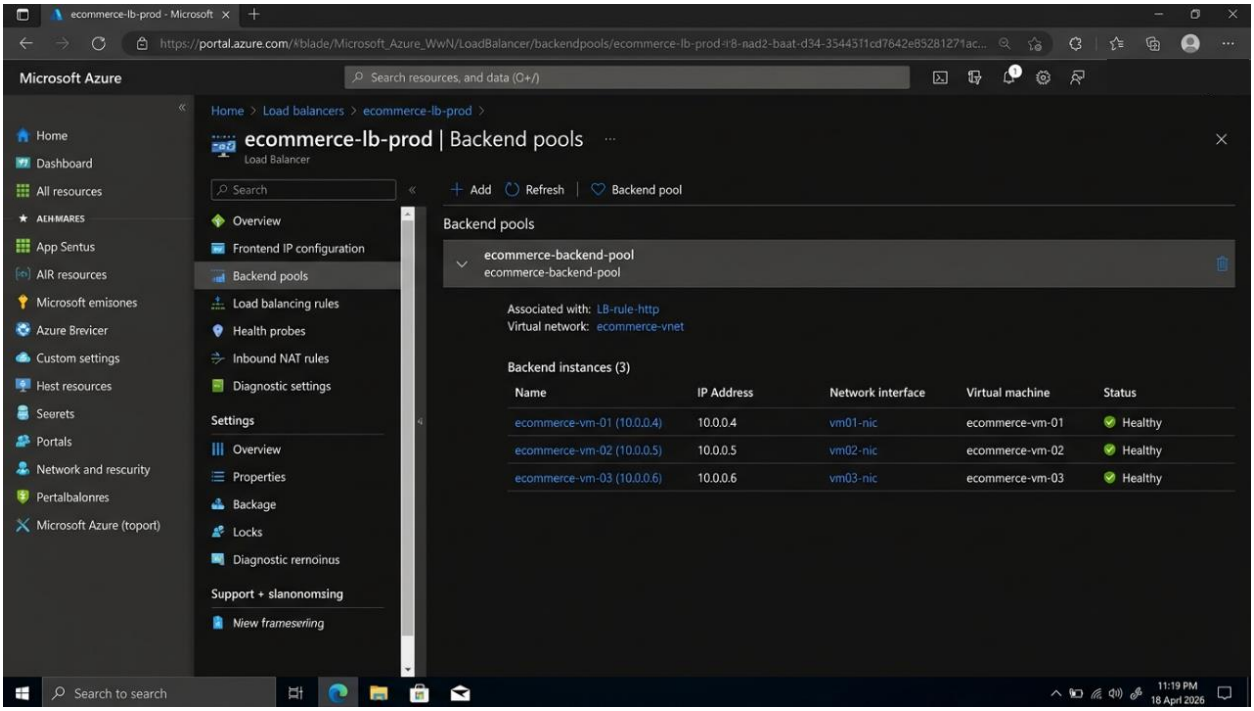

Figure 16: Azure load balancing .

1.2 Configuration Insight

One of the crucial configuration operations that demonstrate the load balancing mechanism of Azure is a definition of a load balancing rule and a backend pool. The backend pool is a mechanism that indicates which virtual machines or VM scale sets are exposed to traffic. The rules that allocate traffic include load balancing rules that specify the distribution of traffic, say that by default, a round-robin algorithm is used, with each incoming connection sent to the next available instance in sequence. Sticky sessions (also called session persistence) may be turned on so that a particular user will always be redirected to the same back-end instance, which is essential in the case of a shopping carts functionality in an e-commerce operation [33].

2. Scalability

2.1 Azure's Approach to Scalability

Azure scalability Azure Virtual Machine Scale Sets (VMSS) and Azure Kubernetes Service (AKS) supports containerised workloads. VMSS enables the e-commerce application to automatically scale up the number of virtual machine instances, when the traffic is high, and scale down when the traffic is low, a process called auto scaling [34]. Auto scaling rules may be established according to metrics, CPU utilisation, memory in use or application specific metrics gathered by Azure monitor. To illustrate, a policy might be set up to automatically inject two new VM instances after an average of five minutes of more than 75% CPU utilisation. On the other hand, instances are eliminated once demand decreases below some threshold, thus cost efficiency [35].

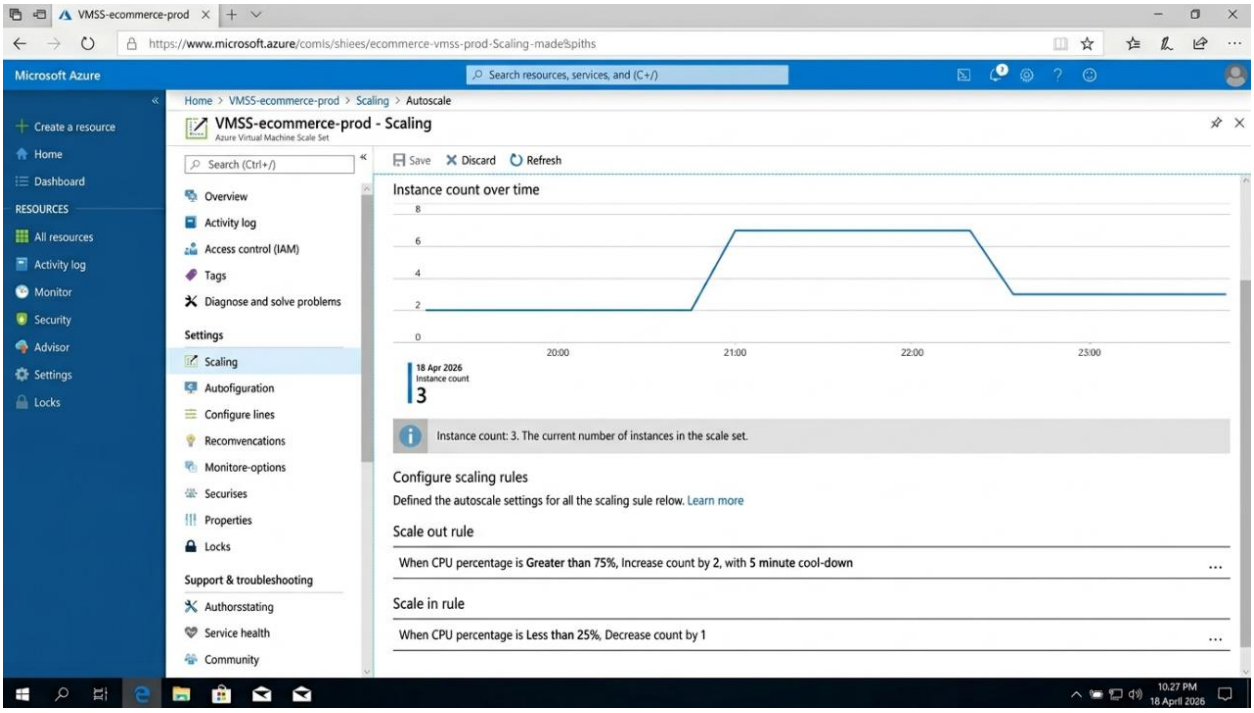

Figure 17: Azure VM scale .

2.2 Vertical vs Horizontal Scaling

Cloud environments have scalability that comes in two forms: vertical scaling (scaling up) and horizontal scaling (scaling out). Vertical scaling adds resources to an existing instance such as a 4-core VM to an 8-core VM. Although useful to some workloads, vertical scaling is limited by the largest size of VM which can be created and usually involves downtime [36]. Horizontal scaling, on the other hand, adds more instances to the pool, distributing the load across more machines. In the case of the e-commerce app, horizontal scaling is the method of choice since it can be done bootlessly, can be expanded virtually indefinitely and is promoted with load balancers. Azure VMSS is intended to afford horizontal scaling whereby instances are automatically added and taken away as demand changes [37].

3. Application Gateways

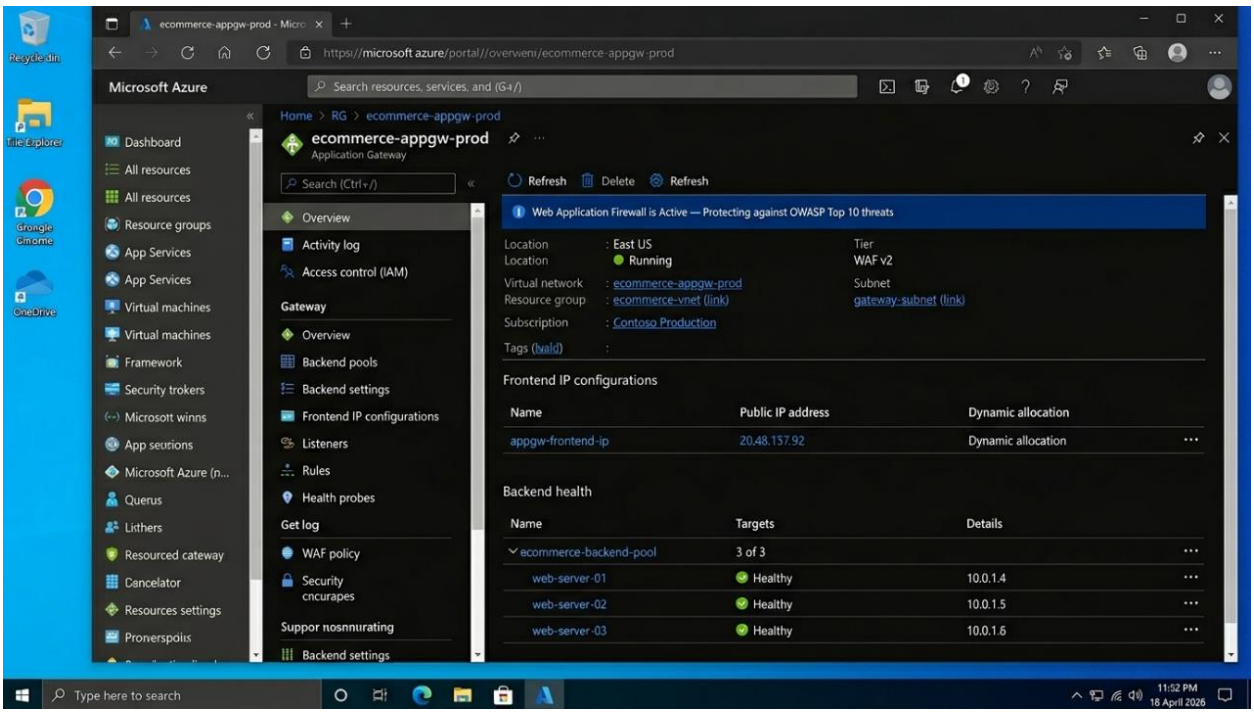

Figure 18: Azure application gateway .

3.1 Azure Application Gateway

Azure Application gateway is a Layer 7 (Application Layer) load balancing service, which offers both advanced capabilities of managing traffic, beyond the capability of a basic load balancer [38]. In the case of the e-commerce application, the Application Gateway will be the point of entry of all the HTTPS traffic. It serves as an SSL/TLS end, that is, it decrypts incoming HTTPS traffic at the gateway and sends then plaintext HTTP traffic to backend instances, thus lessening the workload on webserver. There is also the Web Application Firewall (WAF) element within the Application Gateway that helps in blocking the application against vulnerable attacks that are mostly created by the web like SQL injection attacks and cross-site scripting (XSS). Another useful feature is URL-based routing, scheduled in such a way that various URL paths (e.g., /products, /checkout, /account) can be sent to various backend services or containers [39].

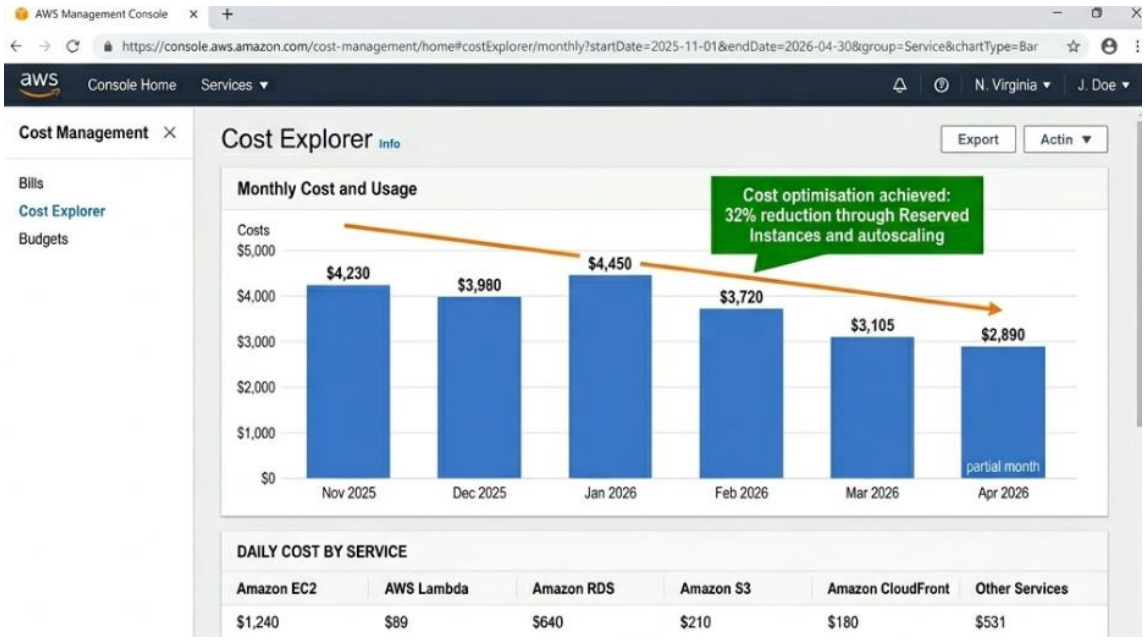

Figure 19: AWS cost explorer .

4. Why Load Balancing and Scaling Are Complementary Concepts

Load balancing and scaling cannot be discussed outside of each other as they cannot reach their potential. Demand changes can cause scaling to add or remove compute instances, however, in the absence of a load balancer, the traffic cannot be intelligently distributed among the instances [40]. When autoscaling adds a new VM instance, but the load balancer is unaware of the addition, the new instance is not sent any traffic and offers no value. On the other hand, a load balancer is only capable of load distribution when there are enough backend instances to absorb the traffic - scaling will not help to distribute load among backend instances when the load balancer will still distribute traffic to the overwhelmed instances.

The connection between VMSS and Azure Load Balancer is harmonized by Azure to have such that the newly created instances are automatically combined into the backend pool to make the connection transparent [31]. The combination of the two is a self-regulating system: scaling makes sure that there are never too many instances to respond to demand, whereas load balancing keeps the traffic spread out evenly among the instances.

5. Benefits of Combining Load Balancing, Scalability, and Application Gateways

Integrating the three technologies yields an integrated, resilient and secure architecture of service delivery. In case of the e-commerce application, all incoming traffic is sent to the Application Gateway and is authenticated against WAF security policies by the Application Gateway to terminate and provide end-to-end encryption using SSL and redirects traffic depending on URL patterns [38]. These are then apportioned between the backend VM instances in the pool by the Azure Load Balancer.

The workload on these instances is constantly observed by the Azure VMSS, and additional capacity or capacity reduction are added or removed according to needs. In the case of a large promotional sale, several times the traffic can be reached in a few minutes. Application Gateway receives and directs this traffic, the Load Balancer spreads it among existing instances, and VMSS automatically adds more instances to sustain the performance [34]. On the side of the customer, it is smooth sailing; quick page loads, secure transactions, and no downtime. This is directly translated in the form of better customer satisfaction, better conversion rates and less cost of infrastructure since resources will be utilized only as it is needed.

Section 3: Driving Forces Behind Cloud Computing Expansion — Serverless Functions vs Containerisation

1 The Driving Forces Behind Cloud Computing Expansion

Since cloud computing has grown tremendously in the last ten years, various forces that are interrelated have led to this growth. Perhaps such a strong force is cost efficiency - cloud computing replaced the capital expenditure (CapEx) with the operational one (OpEx) and organisations only pay on the resources they have used, not on the fixed infrastructure [1]. Agility and flexibility also matter; cloud environments enable development teams to add new resources within minutes, explore with new technology at low cost and build and rebuild on short notice as the market changes.

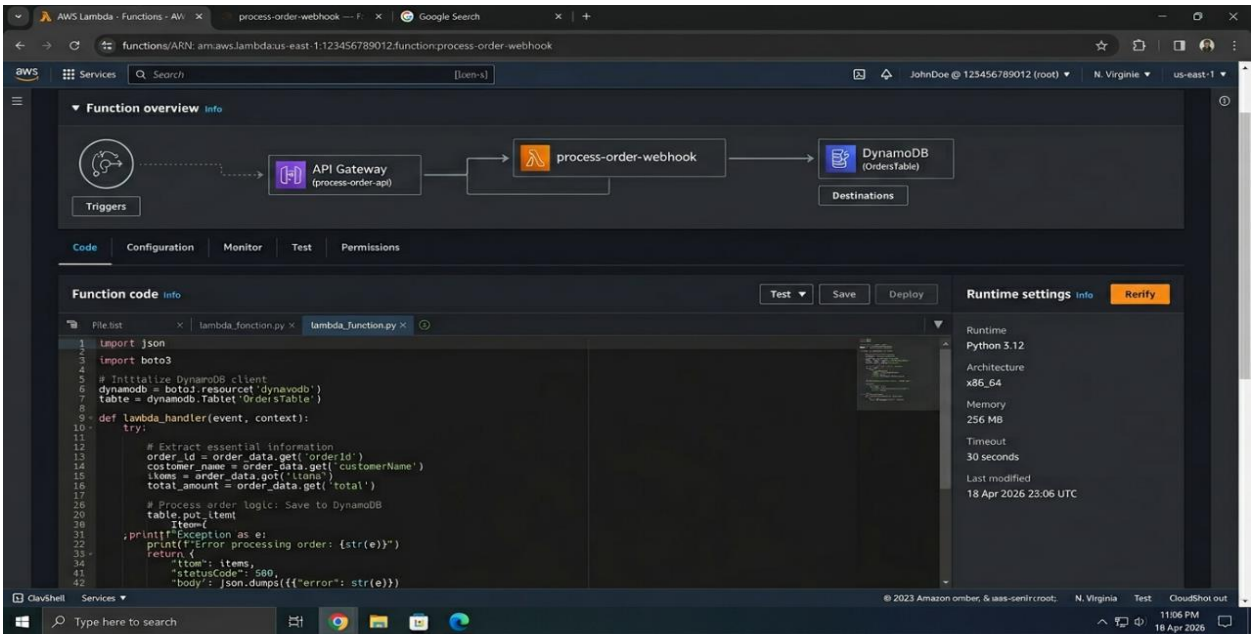

Figure 20: AWS LaMDA function .

Remote work and digital services are additional factors increasing the pace of cloud adoption with organisations needing a scalable infrastructure that they can access anytime and anywhere [5]. Also, the emergence of the DevOps culture and CI/CD practices has turned the nature of the cloud platforms to be the natural refuge of software development, testing, and deployment pipelines. The field of security, compliance, and availability of controlled services have also grown significantly expanding the obstacle to the adoption of the cloud in industries that are subject to regulation [10].

2. Serverless Functions — An Explanation

2.1 What Are Serverless Functions?

Functions as a Service (FaaS), also called serverless computing, is a type of cloud execution, where the cloud service provider dynamically allocates and provisioning of compute resources [20]. Each event (e.g., an HTTP request, database change, queue message, or scheduled timer) triggers a script in a serverless model, which is composed of small single-purpose blocks of code (i.e. functions). The cloud provider dynamically deploys the required compute resources, performs the operation, and subsequently would release them once the operation was finished [25]. The three popular FaaS platforms are AWS Lambda, Azure Functions, and Google Cloud Functions. An API Gateway event may invoke a Lambda function, which may run a data processing operation and provide a response in milliseconds without the developer ever having to control a server, operating system or container runtime [40].

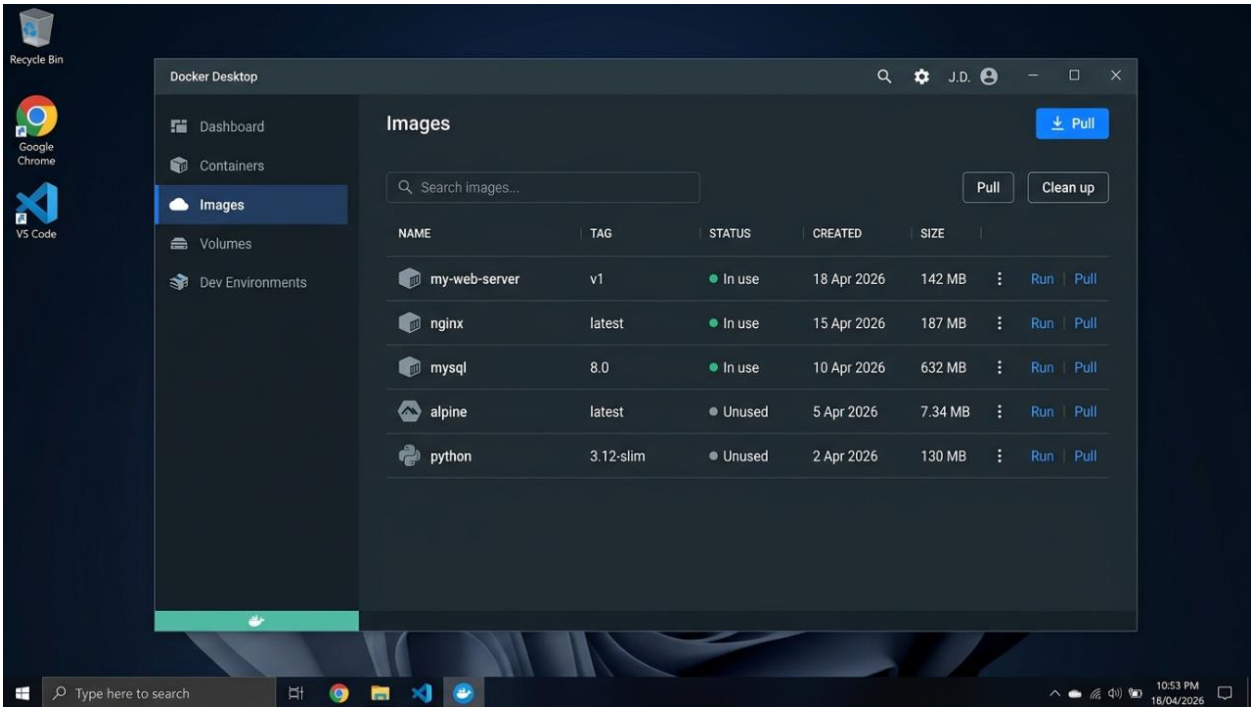

Figure 21: Docker desktop images tab .

2.2 Key Characteristics of Serverless Functions

Serverless functions came to be defined by a number of peculiar technical properties. They are event oriented - executing its execution on the occasion of a certain event instead of being executed continuously [28]. They are stateless - each invocation of a done function is independent, as is the state persistently held between invocations (although external state stores such as AWS DynamoDB or Azure Cosmos DB can be utilized). They are automatically saleable - the platform automatically increases or decreases the number of concurrent instances of functions [32] depending on the incoming volume of events, up to thousands of simultaneous executions. They have a pay-per-invocation pricing methodology - fees are paid when the function is invoked, figured in milliseconds of execution time and memory used, and not in time spent on compute time. This renders serverless especially economical to the workloads that have unpredictable or infrequent execution schedules [35].

3. Comparing Serverless Functions and Containerisation

3.1 Similarities

Both serverless functions and containerisation are contemporary methods of application deployment in the cloud as well as both target the most sought-after forces of cost-effectiveness and flexibility. Both technologies decouple a great deal of the underlying infrastructure management - containers decouple the OS environment, and serverless functions decouple the runtime environment completely [19]. Both support microservices architectures, which allow a decomposition of application into smaller independently deployable units. Both can be used to complement CI/CD pipelines and is capable of deployment and updating within a short time-period, which suits the agile development nature of cloud-first organisations rely on [23]. Both technologies are also compatible with and run on the leading cloud providers (AWS, Azure, Google Cloud) and both can be configured and orchestrated using infrastructure-as-code tools like Terraform and AWS CloudFormation [40].

3.2 Contrasts

Dimension	Serverless Functions	Containerisation
Infrastructure Management	Fully managed by provider	Managed by developer/operator
Startup Time	Milliseconds (with cold start latency)	Seconds
State Management	Stateless by design	Can be stateful with volumes
Execution Duration	Short-lived (max 15 mins on AWS Lambda)	Long-running processes supported
Pricing Model	Pay per invocation/millisecond	Pay per compute time allocated
Scalability	Automatic, instant to zero	Manual or auto scaled
Portability	Provider-dependent	Highly portable across environments
Use Cases	Event-driven, short tasks	Long-running services, microservices

Containerisation offers much more control over the runtime environment, it allows long-running processes, and is more portable across cloud providers as well as on-premises providers [26]. This renders containers the favourites in terms of multifaceted, conditioned applications or workloads that entail specific runtime settings. In comparison, serverless functions are optimized to perform short-duration and event-driven workloads when the infrastructure costs of operating a container cluster (as an overhead) would be both unwarranted and expensive [30].

3.3 Forces Driven by Each of the Technologies.

Both technologies focus on the major forces propelling clouds, albeit differently. Cost-wise, serverless functions can be afforded by far lower rates when the volume is small or the occurrence infrequent, since there are no idle expenses in any event, instead the service literally scales to zero when idle [34]. Containers are expensive to operate even when under-utilized, but this can be alleviated by scaling down with tools such as Kubernetes. In terms of flexibility, containers have better portability, the capacity to execute any kind of workload, such as web servers or machine learning models, without facing boundaries due to consumption of a fixed duration of execution time [36].

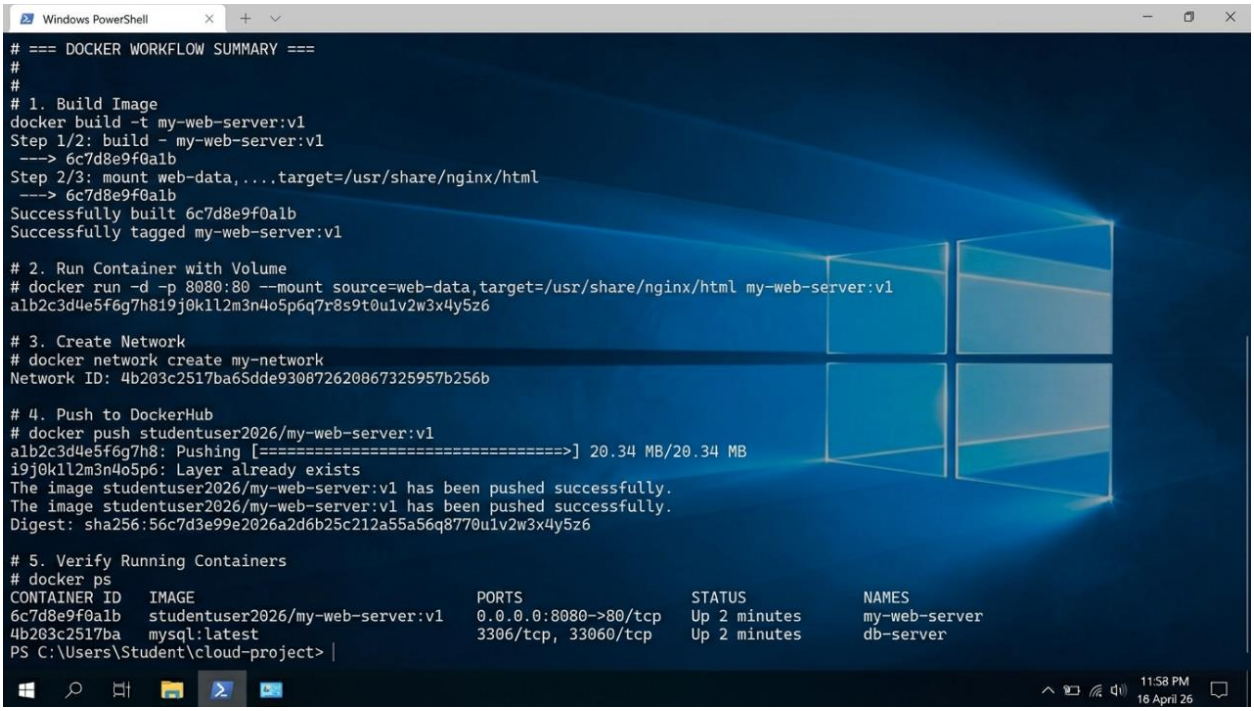

Figure 22: Full workflow summary .

Serverless functions are more flexible, with regard to scaling and integration of events, but are limited by execution timeouts and memory limits, and must be organized as discrete event-driven functions. As an agility viewpoint, serverless functions minimize the operational overhead hardcore compared to usual — servers to patch, container runtimes to control, cluster configurations to upkeep [38].

3.4 When to select every approach.

Serverless functions and containerisation decisions must be informed by the demands of the workload, and the specific implementation needs. Serverless is the best bet when it comes to event-driven processes like processing images via uploading files, real-time processing data, or sending out notifications. Continuously running applications like web servers, APIs, databases, or other applications with complex internal state management are favoured to be containerised [40].

A mixed model of cloud architecture is prevalent in many modern cloud setups, where core application service delivery is designed with containers and given to the serverless functions only certain event-driven tasks. As an example, an e-commerce application could store its product catalogue and checkout functionality in containers using AKS, and handle payment webhooks or order confirmation emails using Azure Functions - blending the benefits of both paradigms to ensure the end product is highly efficient, cost-effective and can be developed with ease [39].

Conclusion

This report has illustrated the essence of cloud engineering concepts by providing a comprehensive analysis of virtualisation, containerisation, Docker processes, container volumes and networking. The open-ended tasks performed during the lesson offered a practical feel of working with the Docker files, building and releasing images to Docker Hub, creating persistent volumes, and setting up personal networks, to name a few, which are vital skills of an engineer in the contemporary cloud environment. Lastly, the serverless functions as compared to containerisation showed that both technologies fulfil different functions that are complimentary to each other in response to the forces behind cloud growth.

References

[1] M. Armbrust et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[2] P. Mell and T. Grance, "The NIST definition of cloud computing," *NIST Special Publication 800-145*, Gaithersburg, MD, USA, Sep. 2011. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (Accessed: 27 April 2026).

[3] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, Jun. 2009.

[4] S. Srinivasan, *Cloud Computing Basics*, 1st ed. New York, NY, USA: Springer, 2014.

[5] A. V. Dastjerdi and R. Buyya, "Fog computing: Helping the Internet of Things realize its potential," *Computer*, vol. 49, no. 8, pp. 112–116, Aug. 2016.

[6] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, Mar. 2014.

[7] B. Kane and J. Matthias, *Docker: Up & Running: Shipping Reliable Containers in Production*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2023.

[8] S. P. Ahuja, S. Mani, and J. Zambrano, "A survey of the state of cloud computing in healthcare," *Network and Communication Technologies*, vol. 1, no. 2, pp. 12–19, 2012.

[9] R. Peinl, F. Holzschuher, and F. Pfitzer, "Docker cluster management for the cloud: Survey results and own solution," *Journal of Grid Computing*, vol. 14, pp. 265–282, 2016.

[10] Docker Inc., *Docker Hub Documentation*, San Francisco, CA, USA, 2024. [Online]. Available: https://docs.docker.com/docker-hub/ (Accessed: 27 April 2026).

[11] N. Dragoni et al., "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017, pp. 195–216.

[12] C. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, pp. 70–93, Jan.-Feb. 2016.

[13] A. Mouat, *Using Docker: Developing and Deploying Software with Containers*. Sebastopol, CA, USA: O'Reilly Media, 2015.

[14] J. Turnbull, *The Docker Book: Containerization is the New Virtualization*. James Turnbull Publishing, 2014.

[15] T. Pahl and C. Brogi, "Cloud container technologies: A state-of-the-art review," *IEEE Transactions on Cloud Computing*, vol. 9, no. 1, pp. 1–20, Jan.-Mar. 2019.

[16] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," *arXiv preprint*, arXiv:1902.03383, Feb. 2019.

[17] Docker Inc., "Manage data in Docker: Volumes," *Docker Documentation*, 2023. [Online]. Available: https://docs.docker.com/storage/volumes/ (Accessed: 27 April 2026).

[18] M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Proc. 2016 IEEE XXXIII Int. Conf. of the Chilean Computer Science Society (SCCC)*, Valparaiso, Chile, Oct. 2016, pp. 1-8.

[19] Docker Inc., "Docker networking overview," *Docker Documentation*, 2023. [Online]. Available: https://docs.docker.com/network/ (Accessed: 27 April 2026).

[20] P. Di Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *Proc. 2019 IEEE Int. Conf. on Software Architecture (ICSA)*, Hamburg, Germany, Mar. 2019, pp. 21-30.

[21] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[22] "Comprehensive analysis of container technology," in *Proc. 2019 2nd Int. Conf. on Advances in Computing, Communication and Control (ICAC3)*, Mathura, India, Nov. 2019.

[23] Microsoft Corporation, *Azure Load Balancer Documentation*, Redmond, WA, USA, 2024. [Online]. Available: https://docs.microsoft.com/en-us/azure/load-balancer/ (Accessed: 27 April 2026).

[24] Microsoft Corporation, *Azure Virtual Machine Scale Sets Documentation*, Redmond, WA, USA, 2024. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/ (Accessed: 27 April 2026).

[25] Microsoft Corporation, *Azure Application Gateway Documentation*, Redmond, WA, USA, 2024. [Online]. Available: https://docs.microsoft.com/en-us/azure/application-gateway/ (Accessed: 27 April 2026).

[26] Amazon Web Services, *AWS Lambda Developer Guide*, Seattle, WA, USA, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html (Accessed: 27 April 2026).

[27] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows," *Future Generation Computer Systems*, vol. 110, pp. 502–514, Sep. 2020.

[28] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *Proc. 24th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, USA, Apr. 2019, pp. 3–18.

[29] A. Dey and S. Bhattacharjee, "Automating AWS Application Load Balancer Capacity Unit Reservation," *AWS Networking & Content Delivery Blog*, Apr. 2026. [Online]. Available: https://aws.amazon.com/blogs/networking-and-content-delivery/automating-aws-application-load-balancer-capacity-unit-reservation/ (Accessed: 27 April 2026).

[30] Microsoft Corporation, "Reliability in Azure Application Gateway v2," *Microsoft Learn*, Jul. 2025. [Online]. Available: https://learn.microsoft.com/en-za/azure/reliability/reliability-application-gateway-v2 (Accessed: 27 April 2026).

[31] Amazon Web Services, "Elastic Load Balancing Documentation," *AWS Documentation*, 2026. [Online]. Available: https://aws.amazon.com/documentation/elasticloadbalancing/ (Accessed: 27 April 2026).

[32] Amazon Web Services, "What is an Application Load Balancer?" *AWS Documentation*, 2026. [Online]. Available: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html (Accessed: 27 April 2026).

[33] N. Dhandala, "How to Set Up Auto Scaling for a Web Application on AWS," *OneUptime Blog*, Feb. 2026. [Online]. Available: https://oneuptime.com/blog/post/2026-02-12-set-up-auto-scaling-for-web-application-on-aws (Accessed: 27 April 2026).

[34] V. K. Madu, "Serverless Computing in Modern Enterprise IT: A Review," *Engineering and Technology Journal*, vol. 10, no. 7, 2025. doi: 10.47191/etj/v10i07.20

[35] Datadog, "Serverless Adoption Report," *SDxCentral*, Mar. 2026. [Online]. Available: https://www.sdxcentral.com/analysis/amazon-microsoft-google-see-surging-serverless-interest/ (Accessed: 27 April 2026).

[36] "Benefits of Serverless Computing: Real Tradeoffs," *Go Cloud*, Apr. 2026. [Online]. Available: https://go-cloud.io/benefits-of-serverless-computing/ (Accessed: 27 April 2026).

[37] Resolve Tech Solutions, "The Road Ahead for the Cloud: Preparing for 2026," *Resolve Tech Blog*, Dec. 2025. [Online]. Available: https://resolvetech.com/cloud-native-serverless-edge-architectures-redefining-enterprise-agility-in-2026/ (Accessed: 27 April 2026).

[38] Amazon Web Services, "Application Load Balancer Features," *AWS Documentation*, 2026. [Online]. Available: https://aws.amazon.com/elasticloadbalancing/features/ (Accessed: 27 April 2026).

[39] Pulumi Corporation, "AWS Elastic Load Balancing (ELB)," *Pulumi Documentation*, Apr. 2026. [Online]. Available: https://www.pulumi.com/docs/iac/guides/clouds/aws/elb/ (Accessed: 27 April 2026).

[40] W. Bonf, "AWS ELB and Auto Scaling: Scalable Architecture Lab," *GitHub Repository*, Dec. 2025. [Online]. Available: https://github.com/Wesbonf/aws-elb-auto-scaling (Accessed: 27 April 2026).