



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

Cloud Engineering Principles and Practice

ACCA7027 Assessment Report

Student Name: Muhammad Waleed

Student ID: 2430361

Program: MSc Cybersecurity & Digital Forensics

Submission To: Renuga Jayakumar

Table of Contents

1. Introduction.....	1
2. Principles of Virtualisation.....	1
3. Principles & Benefits of Containerisation	2
3.1 Benefits of Containerisation.....	2
3.2 Portability in Docker	3
3.3 Docker Images and Publishing Images	4
3.5 Docker Networking.....	6
4. Combined Example: Building, Running & Publishing a Web Server	7
5. Video Presentation	11
6. Cloud Provider Approaches to Major Cloud Concepts.....	11
6.1 Load Balancing.....	11
6.2 Scalability.....	12
6.3 Application Gateways	12
7. Why Load Balancing & Scaling Are Complementary.....	13
8. Benefits of Combining Load Balancing, Scaling & Gateways.....	14
9. Driving Forces Behind Cloud Expansion	14
10. Serverless vs Containerisation	15
10.1 Serverless Functions	15
10.2 Comparison: Serverless vs Containerisation	15
References.....	17

1. Introduction

Cloud computing has become a standard approach for how most organisations operate their applications. Rather than depending on a single physical machine for a single task, modern cloud platforms allow systems to expand, reduce, and support multiple workloads simultaneously. Two key technologies that enable this flexibility are virtualisation and containerisation. Virtualisation allows multiple operating systems to run on one physical server, while containerisation enables applications to be packaged and executed consistently across different environments[1].

This assignment provides an overview of how these technologies function and the benefits they offer. It includes practical demonstrations using Docker Desktop, such as creating images, running containers, pushing and pulling images, working with volumes, and configuring networks. The report also examines broader cloud concepts, including load balancing, scalability, and application gateways, explaining how these components help maintain reliable and efficient application performance[2]. Additionally, it outlines the factors driving the rapid growth of cloud computing and compares serverless architectures with container-based solutions.

2. Principles of Virtualisation

Virtualisation refers to using one physical computer to host several operating systems at the same time. The hardware of the machine is divided into virtual components—such as CPUs, memory, and disk space. Each virtual machine (VM) behaves like a separate computer of its own, even though all VMs share the same underlying physical server.

One key principle of virtualisation is resource abstraction. This means that the physical hardware is hidden behind a virtual layer, so each VM sees its own set of resources. This helps improve hardware utilisation, as multiple users or applications can run side by side on a single physical machine instead of relying on multiple separate servers [3]. As a result, organisations can reduce costs, save energy, and simplify management.

Another important principle is isolation. Every VM runs independently, which means that if one VM crashes, becomes overloaded, or faces a security issue, it does not affect the others. This increases security and stability, especially in cloud environments where many customers may share the same physical infrastructure [4].

Virtualisation also enables flexibility and mobility. A VM can be moved from one physical server to another without shutting it down, using a technique known as live migration. This allows maintenance to take place without interrupting service and helps balance workloads across different servers to improve performance [5].

A further benefit is the ability to allocate resources dynamically. Administrators can adjust CPU, memory, or storage for each VM depending on demand, helping to optimise performance.

Virtualisation also supports running different operating systems on the same hardware, which makes testing, development, and multi-platform applications easier.

Overall, virtualisation makes computing environments more efficient, secure, and cost-effective. By creating separate virtual systems on shared hardware, it allows organisations to run multiple applications at once, reduce downtime, and improve system performance while making better use of physical resources.

3. Principles & Benefits of Containerisation

Containerisation bundles an application with everything required for it to run—its libraries, configuration files, and other dependencies—into a single self-contained unit known as a container [6]. As a result, the application runs consistently on any system that supports the container runtime, whether that is a developer’s laptop, a test server, or a cloud platform. This consistency reduces issues where software functions correctly on one machine but fails on another due to missing packages or mismatched tool versions [7].

Although each container operates in its own isolated environment, they all rely on the host machine’s operating system kernel. Because they do not need a separate OS for every instance, containers are more lightweight and typically start up more quickly than virtual machines [6]. This leads to better utilisation of CPU and memory resources, allowing many containers to run on hardware that might otherwise support only a small number of virtual machines [8].

Using containers simplifies and stabilises both development and deployment. Developers can create an image once and reuse the same container across multiple stages—development, testing, and production—without altering the underlying environment [6]. This model aligns closely with modern DevOps workflows, including continuous integration and continuous delivery (CI/CD), where applications must be updated and released regularly in a predictable and repeatable manner [7].

3.1 Benefits of Containerisation

Containerisation has several clear benefits:

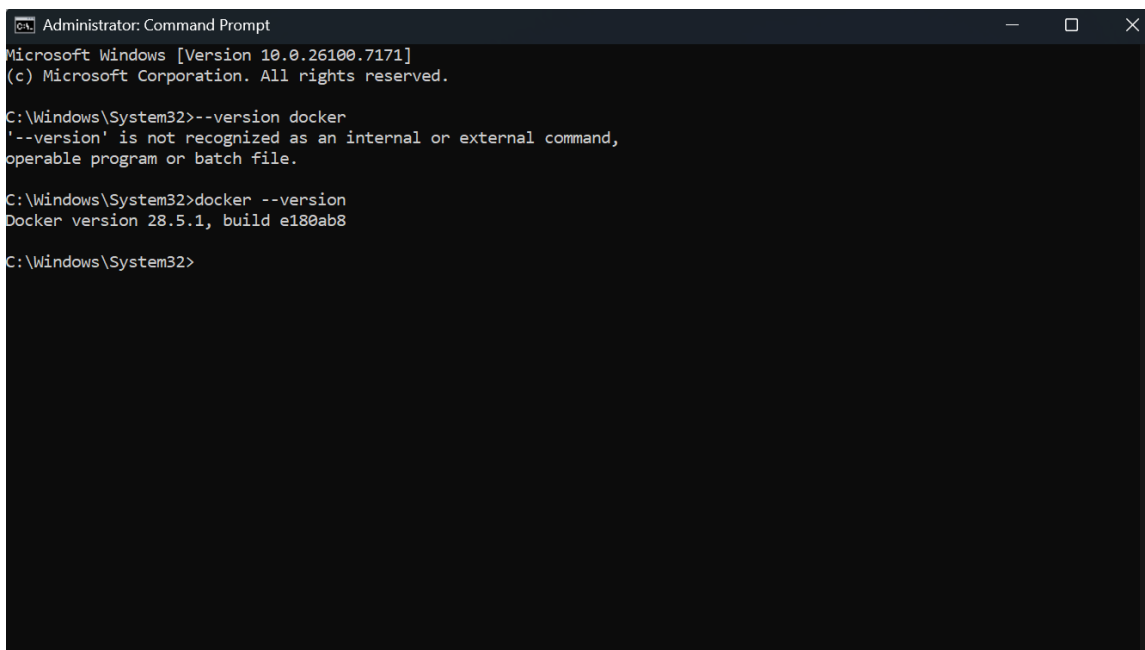
- **Lightweight:** Containers use fewer resources and start very quickly.
- **Consistent:** They run the same way on every machine, avoiding setup problems.
- **Portable:** Containers can be moved easily between laptops, servers, or cloud platforms.
- **Easy to scale:** Create or remove containers within seconds depending on demand.
- **Fast deployment:** Updating applications becomes quicker and simpler.
- **Better resource use:** More applications can run on the same hardware, reducing cost.

3.2 Portability in Docker

Docker is popular because it makes applications truly portable [8]. Instead of installing software separately on every machine, developers package the application and everything it needs—libraries, runtime, configuration files, and other dependencies—into a Docker image. When a container is started from this image, it runs in the same way on any system that supports Docker, whether it is Windows, Linux, macOS, or a cloud platform such as AWS or Azure [8][9]. Which significantly cuts down on situations where an application runs correctly on one system but fails on another, where code on different systems it have different behaviour due to missing or conflicting dependencies.

A Docker image is built in layers, which represent different stages such as the base operating system image, installed packages, and the application code itself [8]. These layers can be reused across multiple images, which saves storage space and speeds up distribution. Once an image is created, it can be stored in a container registry such as Docker Hub or a private registry, and then pulled and run on any compatible host. This supports a consistent workflow from development to testing and production, as the same container can be used in every stage [9].

This model fits very well with modern DevOps practices and continuous integration and continuous delivery (CI/CD). Build pipelines can automatically create new Docker images when the code changes, run tests inside containers, and then deploy updated containers to servers or Kubernetes clusters [9][10]. Because the environment inside the container does not change, teams can be more confident that what was tested is exactly what is deployed. Overall, Docker's approach to packaging and portability makes development, testing, and deployment smoother, more predictable, and easier to automate [8][10].



```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.26100.7171]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>--version docker
'--version' is not recognized as an internal or external command,
operable program or batch file.

C:\Windows\System32>docker --version
Docker version 28.5.1, build e180ab8

C:\Windows\System32>
```

Figure 1. Docker Version

3.3 Docker Images and Publishing Images

A Docker image is a kind of blueprint or template used to create containers. It contains everything needed for an application to run, such as the application code, system tools, libraries, and settings. When start a container, Docker uses this image as the starting point.

Developers usually create a Docker image by writing a file called a Dockerfile. The Dockerfile includes a list of instructions, such as which base image to start from, which packages to install, which files to copy, and which command to run when the container starts. Docker reads this file and then builds the image step by step.

Once the image has been built, it can be stored on the local machine or pushed to a remote registry. A common public registry is Docker Hub, but companies often use private registries as well. By uploading an image to Docker Hub, developers make it easy for others to download and run the same application on their own systems. This makes sharing, deploying, and updating applications much simpler and more consistent across different environments.

Example image steps:

1. Build image: **docker build -t waleed2171/docker-web-demi:v1**

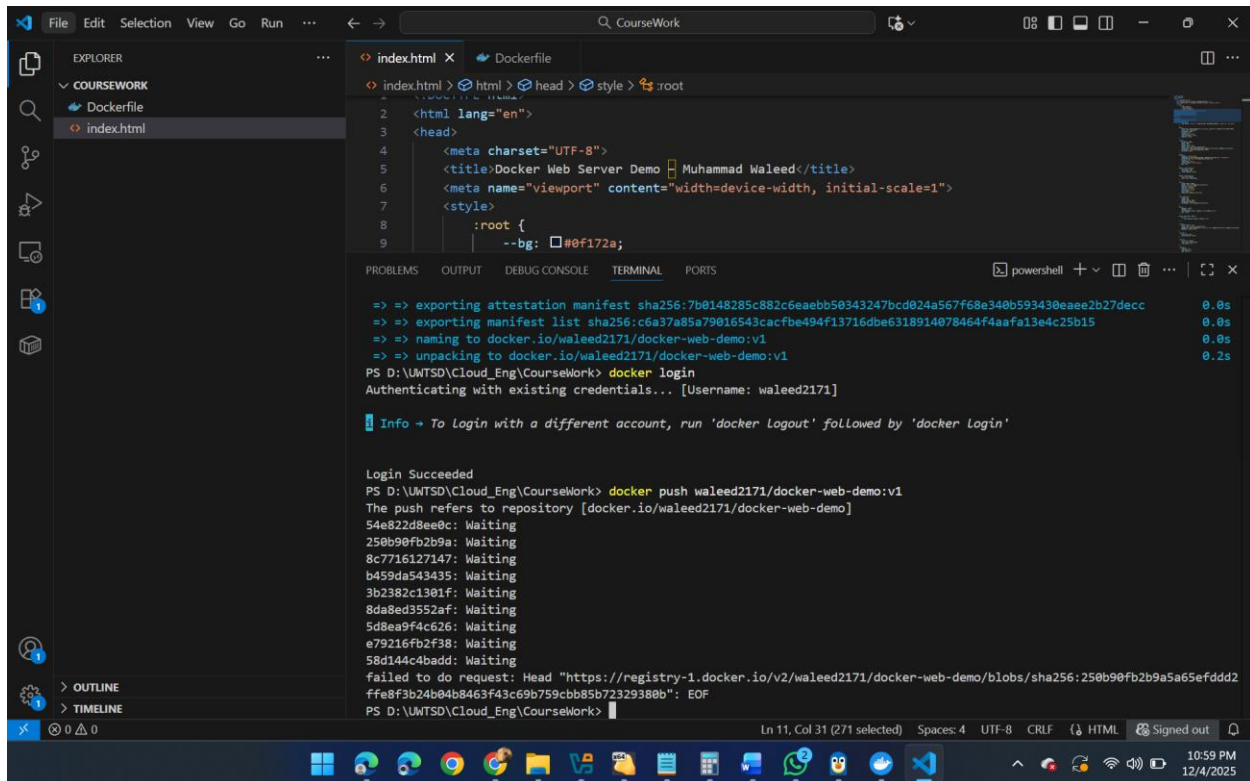
```

PS D:\UMTSD\Cloud_Eng\CourseWork> docker build -t waleed2171/docker-web-demi:v1 .
[+] Building 2.2s (7/7) FINISHED
=> [internal] load build definition from Dockerfile                                0.1s
=> => transferring dockerfile: 117B                                              0.1s
=> [internal] load metadata for docker.io/library/nginx:latest                  0.2s
=> [internal] load .dockerignore                                                 0.1s
=> => transferring context: 2B                                                  0.0s
=> [internal] load build context                                                0.1s
=> => transferring context: 11.96kB                                             0.0s
=> [1/2] FROM docker.io/library/nginx:latest@sha256:3b7732505933ca591ce4a6d860cb713ad96a3176b82f7979a8dfa9973486a0 0.4s
=> => resolve docker.io/library/nginx:latest@sha256:3b7732505933ca591ce4a6d860cb713ad96a3176b82f7979a8dfa9973486a0 0.1s
=> [2/2] COPY index.html /usr/share/nginx/html/index.html                      0.2s
=> => exporting to image                                                         0.7s
=> => exporting layers                                                         0.3s
=> => exporting manifest sha256:e8f74d5f46bab406af78991f1a206838584e5a7ec1467767655a2a00151ffb2b 0.0s
=> => exporting config sha256:3fd08c7278a6ac6e72f0b5b317ca7752d5c5de6851f02e6729a0197f7d104fd2 0.0s
=> => exporting attestation manifest sha256:7b0148285c882c6eae5b50343247bcd024a567f68e340b593430eae2b27decc 0.0s
=> => exporting manifest list sha256:c6a37a85a79016543cacfb494f13716db6318914078464f4aafa13e4c25b15 0.0s
=> => naming to docker.io/waleed2171/docker-web-demi:v1                       0.0s
=> => unpacking to docker.io/waleed2171/docker-web-demi:v1                    0.2s
PS D:\UMTSD\Cloud_Eng\CourseWork>

```

Figure 2: Docker Build image

2. Push to DockerHub: `docker push waleed2171/docker-web-demo:v1`



```

index.html x Dockerfile
index.html > html > head > style > root
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <title>Docker Web Server Demo | Muhammad Waleed</title>
6 <meta name="viewport" content="width=device-width, initial-scale=1">
7 <style>
8 :root {
9   --bg: #0f172a;

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + - - - - - x

=> => exporting attestation manifest sha256:7b0148285c882c6eae5b50343247bcd024a567f68e340b593430eae2b27decc 0.0s
=> => exporting manifest list sha256:c6a37a85a79016543cacfb494f13716db6318914078464f4aafa13e4c25b15 0.0s
=> => naming to docker.io/waleed2171/docker-web-demo:v1 0.0s
=> => unpacking to docker.io/waleed2171/docker-web-demo:v1 0.2s
PS D:\UWTSD\Cloud_Eng\CourseWork> docker login
Authenticating with existing credentials... [Username: waleed2171]

Info - To Login with a different account, run 'docker logout' followed by 'docker login'

Login Succeeded
PS D:\UWTSD\Cloud_Eng\CourseWork> docker push waleed2171/docker-web-demo:v1
The push refers to repository [docker.io/waleed2171/docker-web-demo]
54e822d8ee0c: Waiting
250b90fb2b9a: Waiting
8c7716127147: Waiting
b459da543435: Waiting
3b2382c1301f: Waiting
8da8ed3552af: Waiting
5d8ea9f4c626: Waiting
e79216fb2f38: Waiting
58d144c4badd: Waiting
failed to do request: Head "https://registry-1.docker.io/v2/waleed2171/docker-web-demo/blobs/sha256:250b90fb2b9a5a65efddd2
ffe8f3b24b04b8463f43c69b759cbb85b72329380b": EOF
PS D:\UWTSD\Cloud_Eng\CourseWork>
  
```

Figure 3: Docker Push image

3.4 Docker Volumes

Docker volumes are used to store data outside the container's own file system. This is important because anything saved only inside a container is temporary and will be lost if the container is stopped, deleted, or recreated. By using a volume, the data is kept in a separate location on the host machine, so it remains available even when containers change.

Volumes are particularly valuable for storing data that must persist beyond the life of a container, such as databases, log files, configuration files, or user-uploaded content. For instance, a database container can save its data in a volume so that the records remain intact even if the container is updated, recreated, or replaced.

Docker provides straightforward tools for creating and managing volumes. A named volume can be created using a command such as:

docker volume create mydata

```
PS D:\UWTSD\Cloud_Eng\CourseWork> docker volume ls
DRIVER      VOLUME NAME
local       mydata
PS D:\UWTSD\Cloud_Eng\CourseWork> docker inspect mydata
[
  {
    "CreatedAt": "2025-12-04T23:23:11Z",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/mydata/_data",
    "Name": "mydata",
    "Options": null,
    "Scope": "local"
  }
]
```

Figure 4. Docker Volume Creation

Then attach this volume to a container: **docker run -d -p 8080:80 -v mydata:/usr/share/nginx/html --name web-volume-demo nginx:latest**

```
PS D:\UWTSD\Cloud_Eng\CourseWork> docker run -d -p 8080:80 -v mydata:/usr/share/nginx/html --name web-volume-demo nginx:latest
64d5ab5c016889b0b53a83877ba22970fc6e0b8c326357b80e91e5f8db112af6
docker: Error response from daemon: failed to set up container networking: driver failed programming external connectivity on endpoint web-volume-demo (276749cb215aa7f8e3fa54ba56edd2ee1e99e6155b741828764e74bb1da36017): Bind for 0.0.0.0:8080 failed: port is already allocated

Run 'docker run --help' for more information
PS D:\UWTSD\Cloud_Eng\CourseWork>
```

Figure 5. Volume Attach to a Container

3.5 Docker Networking

Docker networking allows containers to talk to each other and to the outside world in a controlled way. When containers run, they frequently need to interact with databases, web services, or other application components. Docker's networking capabilities enable this communication. By default, Docker sets up a bridge network, and any container started without specifying a custom network is typically connected to this default bridge. Containers within the same bridge network can communicate using IP addresses and, in certain configurations, by referring to each other by container name. Ports can also be exposed from a container to the host system using the `-p` option, for example:

```
docker run -d -p 8080:80 --name docker-web-demo waleed2171/docker-web-demo:v1
```

```
PS D:\UWTSD\Cloud_Eng\CourseWork> docker run -d -p 8080:80 --name docker-web-demo waleed2171/docker-web-demo:v1
821a6d31e0f25af57f515624219431248ff67b84babf8502aa275f0a613bc1b3
PS D:\UWTSD\Cloud_Eng\CourseWork>
```

Figure 6. Docker Network

Docker also supports host networks, where the container shares the host's network stack. In this mode, the container does not get its own separate IP address; instead, it uses the host's IP and ports directly. This can be useful for performance in some cases but offers less isolation.

In addition, Create custom networks. Custom bridge networks are commonly used because they let containers communicate using container names instead of IP addresses, which makes setups easier to manage. For example:

```
docker network create mynetwork
docker run -d --network mynetwork -p 8080:80 --name web waleed2171/docker-web-demo:v1
docker run -d --network mynetwork --name db -e
MYSQL_ROOT_PASSWORD=Password123 mysql:latest
docker network ls
```

```
PS D:\UWTSD\Cloud_Eng\CourseWork> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
104f7c814eae        bridge              bridge               local
199ba8f79779        host                host                 local
684500d927b2        mynetwork           bridge               local
cf7c07e1299e        none                null                 local
PS D:\UWTSD\Cloud_Eng\CourseWork> docker ps
CONTAINER ID        IMAGE               NAMES                COMMAND              CREATED             STATUS              PORTS
ee898c60b0bb       mysql:latest       db                   "docker-entrypoint.s..." 20 seconds ago     Up 18 seconds      3306/tcp,
33060/tcp
50bcfade80c4       waleed2171/docker-web-demo:v1 "/docker-entrypoint..." About a minute ago   Up About a minute   0.0.0.0:8080->80/tcp, [::]:8080->80/tcp web
PS D:\UWTSD\Cloud_Eng\CourseWork>
```

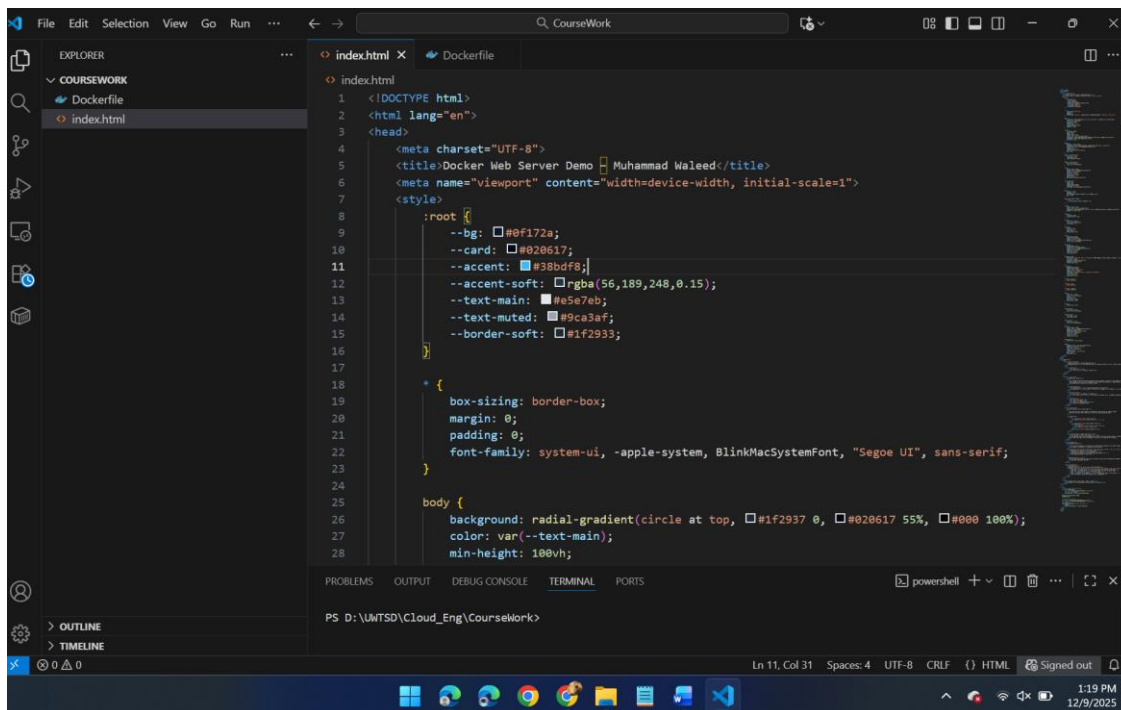
Figure 7. Docker Custom Network

4. Combined Example: Building, Running & Publishing a Web Server

This example shows the full process of creating a small web server using Docker. It includes building the image, running the container, checking the web page, and then publishing the image to DockerHub. This demonstrates how containerisation makes it easy to package and share applications.

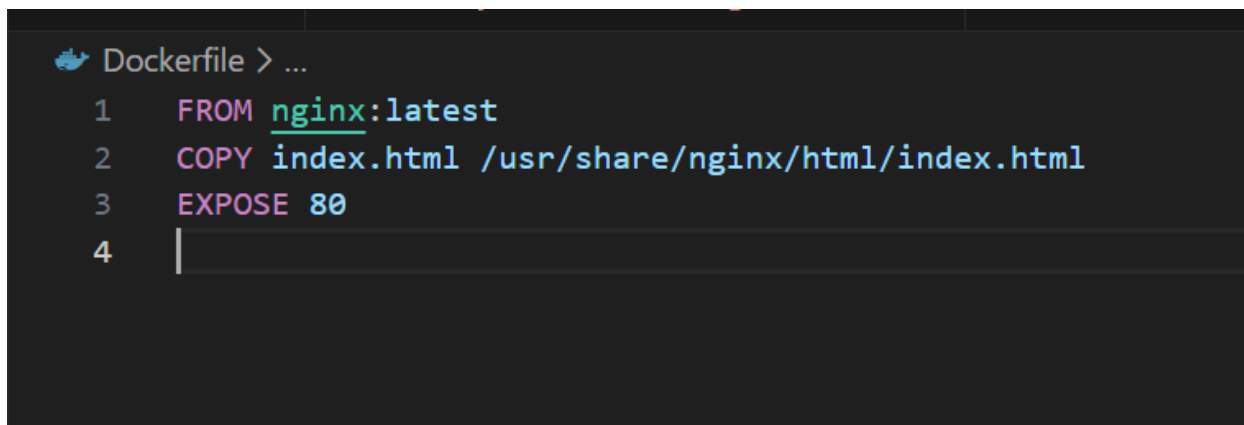
Step 1: Create a simple HTML file

Create an index.html file with a short message such as:



Step 2: Write a Dockerfile

The Dockerfile tells Docker how to build the image:



Step 3: Build the Docker image

Command: `docker build -t waleed2171/docker-web-demo:v1 .`

Docker then produces an image that includes both the Nginx web server and the provided web page.

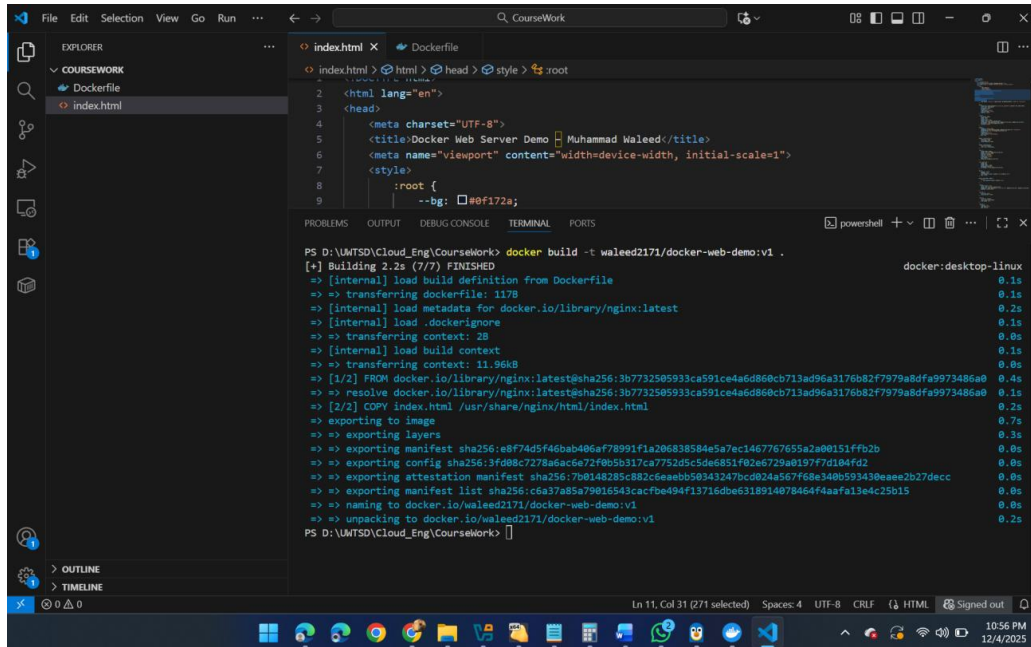


Figure 8. Image Build

Step 4: Run the container

Launch the web server container by mapping 80 port inside the container to port :8080 on the host side/machine:

Command: **docker run -d -p 8080:80 --name docker-web-demo waleed2171/docker-web-demo:v1**

This runs the web server in the background.

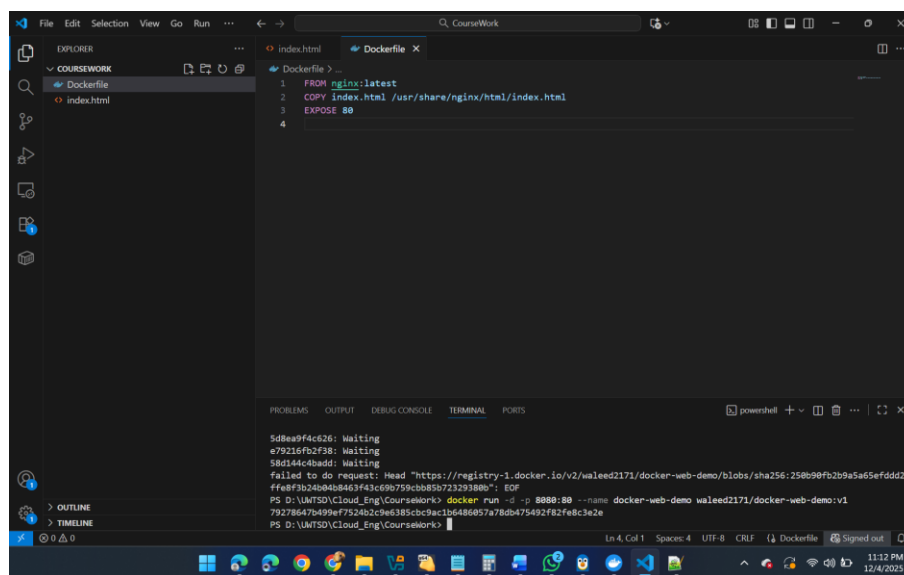


Figure 9. Run Container

Step 5: Open the web page

Open a browser and go to: `http://localhost:8080`

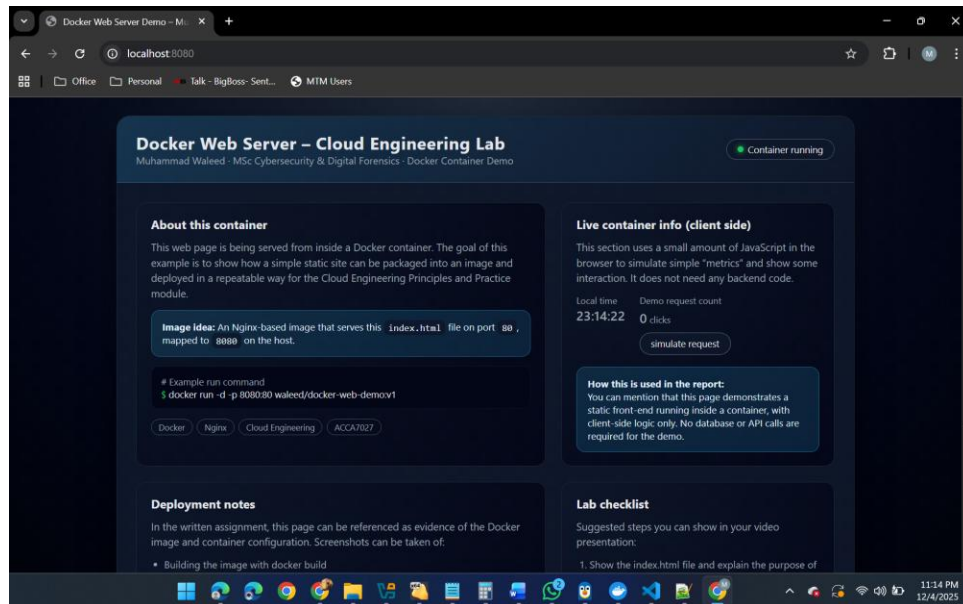


Figure 10. Docker Web Demo

Step 6: Publish the image to DockerHub

First log in: **docker login**

Then push image: **docker push waleed2171/docker-web-demo:v1**

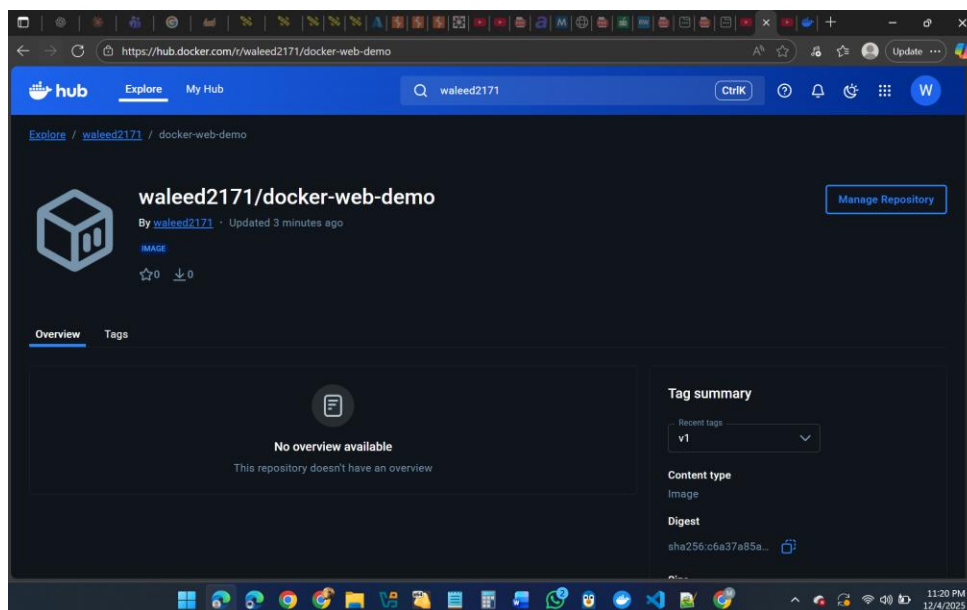


Figure 11. Docker Hub

5. Video Presentation

https://www.youtube.com/watch?v=Lxsl6bgTY_s

6. Cloud Provider Approaches to Major Cloud Concepts

Cloud platforms such as AWS, Microsoft Azure, and Google Cloud use a variety of built-in services to help manage traffic, scale applications, and protect web systems. These tools allow applications to handle large numbers of users, stay available during busy periods, and defend against security threats. By offering features such as load balancing, automatic scaling, and advanced security filtering, these platforms make it much easier to run reliable online services without needing complex hardware on-site. Overall, these cloud services work together to improve performance, increase stability, and ensure that applications always remain secure and accessible at all times [11].

6.1 Load Balancing

Load balancing spreads incoming traffic across multiple servers so that no single server becomes overloaded. This helps applications stay available and responsive during busy periods and reduces the risk of one machine becoming a bottleneck or point of failure [12]. Modern load balancers can also use different algorithms, such as round-robin, least connections, or based on server health and response time, to decide where to send each request [12].

AWS uses Elastic Load Balancing (ELB) to offer multiple load-balancing options for different scenarios [13]. The Application Load Balancer (ALB) operates at the application layer (Layer 7), handling HTTP and HTTPS requests. It enables capabilities such as routing based on URL paths, directing traffic by hostname, and working closely with containers and microservices. The Network Load Balancer (NLB) functions at the transport layer (Layer 4) and is optimised for extremely high throughput and very low latency, which makes it ideal for TCP/UDP workloads and sudden surges in connections [13].

In Microsoft **Azure**, two main services provide similar functionality: Azure Load Balancer and Azure Application Gateway [14]. Azure Load Balancer is primarily a Layer 4 service that spreads traffic across virtual machines in the same region and supports both inbound and outbound flows. Azure Application Gateway runs at Layer 7, where it can analyse HTTP(S) requests, apply URL-based routing rules, and integrate with a Web Application Firewall (WAF) to enhance security. It also offers SSL termination so encrypted traffic can be decrypted and inspected before being forwarded to backend servers [14].

Google Cloud provides Cloud Load Balancing, which is a fully managed, software-defined service that can distribute traffic across instances in multiple regions [15]. It supports both global HTTP(S) load balancing for web traffic and regional load balancing for other protocols. One key advantage is that it offers a single anycast IP address that can automatically route users to the

closest healthy backend, helping to reduce latency and improve user experience across different geographic locations [15].

All of these services also perform regular health checks on backend instances to see if they are responding correctly [13][15]. If a server fails a health check, it is automatically removed from the pool of available backends, so no new traffic is sent to it. When the server recovers and passes health checks again, it can be added back into the rotation. This automatic monitoring and failover behaviour improves the stability and reliability of applications without requiring manual intervention.

6.2 Scalability

Scalability allows applications to increase or decrease their resources depending on demand. When more users access the system or when workloads grow, extra computing power, memory, or instances can be added. When demand drops, these resources can be reduced again so that the system does not waste capacity or money. In cloud environments, this process is often automated, so the platform can react to changes in traffic without someone needing to adjust things manually [16].

Cloud platforms support this through features like automatic scaling rules and policies. For example, the system can be configured to add more instances when CPU usage goes above a certain level, or remove instances when it stays low for a period of time. This helps the application perform well during busy times while staying cost-effective during quieter periods. Scalability is therefore a key advantage of cloud computing, as it allows applications to grow or shrink smoothly as needs change, rather than being limited by fixed physical hardware [16].

- **AWS Auto Scaling Groups** automatically add or remove EC2 instances based on CPU or traffic levels [17].
- **Azure Virtual Machine Scale Sets (VMSS)** adjust the number of VMs according to rules set by the user [18].
- **Google Cloud Managed Instance Groups** work in a similar way to maintain performance during peak usage [19].

Auto-scaling helps reduce costs by only using the resources needed at any given moment.

6.3 Application Gateways

Application gateways work at the application layer (Layer 7), which means they can look at the actual content of web requests rather than just IP addresses and ports [20]. Because of this, they can make smarter decisions about where to send traffic, such as routing users based on the URL path, hostname, or specific HTTP headers. For example, one application gateway can send requests for /api to a backend API service and requests for /shop to a different web service.

They also provide important security features. One common function is SSL termination, where the gateway handles the process of decrypting HTTPS traffic. This takes the heavy encryption work away from the backend servers, making them faster and easier to manage. Another key feature is the Web Application Firewall (WAF), which inspects requests for signs of attacks such as SQL injection, cross-site scripting, or other common web vulnerabilities [20]. Suspicious or malicious requests can be blocked before they reach the application.

By combining intelligent routing and security filtering, application gateways help to protect web applications, simplify certificate management, and improve overall performance. They act as a smart entry point into the system, ensuring that only safe and correctly handled traffic reaches the backend services [20].

- **AWS Application Load Balancer** supports path-based routing and integrates with AWS security services [13].
- **Azure Application Gateway** includes a built-in WAF that helps block attacks such as SQL injection and cross-site scripting [14].
- **Google Cloud HTTP(S) Load Balancer** provides similar application-level routing and security features [21].

Gateways help ensure that traffic is routed safely and efficiently to the correct service.

7. Why Load Balancing & Scaling Are Complementary

Load balancing and scaling work together because they both help an application stay stable and responsive when the number of users changes. Load balancing spreads incoming traffic across several servers so that no single machine becomes overloaded. This keeps the system running smoothly and reduces the risk of slowdowns or crashes.

Scaling supports this by automatically increasing or decreasing the number of servers based on demand. For example, when traffic rises, autoscaling services such as Azure Autoscale [22] and Google Cloud Autoscaler [23] can add more instances to handle the extra load. When demand drops, they reduce the number of servers to save cost.

However, scaling alone is not enough. Without a load balancer, new servers would not receive traffic properly. Likewise, a load balancer cannot keep the system stable if there are not enough servers running. As explained by Snyder [24], the two features work best when used together because scaling provides the resources, while load balancing ensures traffic is shared evenly across them.

Together, they allow applications to remain fast, reliable, and cost-efficient, even during sudden spikes in usage [22][24].

8. Benefits of Combining Load Balancing, Scaling & Gateways

Using load balancing, scaling, and application gateways together creates a strong and reliable cloud environment. Each technology has its own role, but when they work together, they provide much better performance, stability, and security than when used on their own.

Load balancing spreads incoming traffic across several servers, helping prevent overload and keeping the system responsive [11]. If one server fails, the load balancer can redirect users to a healthy server, allowing the application to continue running smoothly.

Scaling supports this by adding or removing servers based on demand. When user activity increases, more servers can be created automatically, and when activity drops, extra servers are removed to save cost [12], [13]. This ensures the load balancer always has enough resources to distribute traffic effectively.

Application gateways add an additional layer of control and security. They filter and manage incoming requests, block unsafe traffic, handle SSL certificates, and route users to the correct part of the application [14]. This improves both security and overall performance.

When all three elements are combined, the system becomes highly efficient and resilient. Load balancing manages traffic, scaling provides the necessary computing power, and the application gateway protects and directs incoming requests. Together, they create a cloud environment that can adapt to sudden increases in demand, stay secure, and deliver a reliable user experience.

9. Driving Forces Behind Cloud Expansion

Cloud computing has grown very quickly over the last decade, and several key factors are driving this expansion. One of the most important is cost efficiency. Instead of buying and maintaining their own servers, organisations can rent computing resources from cloud providers and only pay for what they use. This shifts spending from large upfront capital costs (CAPEX) to flexible operational costs (OPEX), which is easier to manage and scale [29][30].

Another major driver is elasticity and scalability. Cloud platforms can quickly add or remove resources in response to changing demand. When more users access an application, the system can automatically scale out to handle the load. When usage drops, resources can be scaled back to save money. This level of elasticity is very hard to achieve with traditional on-premise infrastructure [29][30].

Global reach and low latency are also important. Large cloud providers such as AWS, Azure, and Google Cloud operate data centres around the world. Applications can be deployed closer to users in different regions, which helps reduce latency and improve user experience. For many businesses, this makes it possible to serve a global audience without building their own international infrastructure [30].

Cloud platforms also offer many managed services, such as databases, message queues, analytics tools, and AI services. These reduce the amount of time teams spend on infrastructure and allow them to focus more on application features and business value. Built-in capabilities for backup, disaster recovery, monitoring, and security further encourage organisations to move workloads into the cloud [29][30].

Overall, the combination of lower cost, elasticity, global availability, and rich managed services has made cloud computing an attractive choice for both small and large organisations, and continues to drive its rapid growth [29][30].

10. Serverless vs Containerisation

10.1 Serverless Functions

Serverless computing is a cloud model that allows code to run without the need to manage servers. Services such as AWS Lambda and Azure Functions enable users to upload small units of code, known as functions, which are executed automatically by the cloud provider in response to events such as HTTP requests, queue messages, file uploads, or scheduled triggers [31][32].

In the serverless model:

- Virtual machines and containers are not managed directly by the user.
- The platform automatically adjusts its capacity by scaling up or down based on incoming request volume.
- Costs are typically based on actual execution time and number of requests, rather than on maintaining idle server resources.

Serverless functions are typically short-lived, stateless, and well suited for tasks such as APIs, background jobs, event processing, and glue code between services [31][32].

10.2 Comparison: Serverless vs Containerisation

Both serverless and containerisation are used to run applications in the cloud, but they follow different models and are useful in different situations.

With containerisation (such as Docker containers deployed on Kubernetes clusters or virtual machines) an application and all of its dependencies are packaged into a container image. This approach provides greater control over the runtime environment, operating system libraries, networking configuration, and long-running processes. Containers are well suited for microservices, web applications, and workloads that require custom runtimes or continuous operation. However, the responsibility of deciding how many containers to run, how to scale them, and how to keep them up to date still remains with the user or the platform.

In contrast, serverless computing abstracts most infrastructure management. The cloud provider handles resource provisioning, scaling, and fault tolerance, while the user supplies only the functions to be executed [31][32]. This reduces operational overhead but offers less control over the execution environment. Serverless functions typically have limitations related to execution time, memory capacity, supported runtimes, and may experience “cold starts” when they are invoked infrequently.

From a cost perspective, serverless architectures can be highly efficient for event-driven or highly variable workloads because charges are incurred only when functions run. Containers may be more economical for consistent, long-running workloads where resource consumption is predictable and always-on services are required.

In practice, many modern systems combine both models. Core or persistent services often run in containers, while serverless functions are used for background operations, scheduled tasks, or event-driven processing. The decision between serverless and containerisation depends on factors such as desired level of control, performance requirements, workload characteristics, and the technical expertise available within the organisation [30][32].

References

- [1] R. Buyya, C. Vecchiola, and S. Selvi, *Mastering Cloud Computing: Foundations and Applications Programming*. New York, NY, USA: McGraw-Hill, 2013.
- [2] B. Snyder, “Cloud load balancing and scalability: Principles and practices,” *IEEE Internet Computing*, vol. 18, no. 4, pp. 61–65, 2014.
- [3] P. Barham et al., “Xen and the art of virtualization,” *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.
- [4] M. Rosenblum and T. Garfinkel, “Virtual machine monitors: Current technology and future trends,” *IEEE Computer*, vol. 38, no. 5, pp. 39–47, 2005.
- [5] C. Clark et al., “Live migration of virtual machines,” in *Proc. 2nd Symp. Networked Systems Design and Implementation (NSDI)*, 2005, pp. 273–286.
- [6] M. Boettiger, “An introduction to Docker for reproducible research,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.
- [7] F. Pahl, “Containerization and the PaaS cloud,” *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.
- [8] A. Eftimie and E. Borcoci, “Containerization using Docker technology,” in *Proc. Twentieth Int. Conf. on Networks (ICN)*, 2021, pp. 41–44.
- [8] M. Boettiger, “An introduction to Docker for reproducible research,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.
- [9] F. Pahl, “Containerization and the PaaS cloud,” *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.
- [10] A. Eftimie and E. Borcoci, “Containerization using Docker technology,” in *Proc. Twentieth Int. Conf. on Networks (ICN)*, 2021, pp. 41–44.
- [11] R. Buyya, C. Vecchiola, and S. Selvi, *Mastering Cloud Computing: Foundations and Applications Programming*. New York, NY, USA: McGraw-Hill, 2013.
- [12] H. Simonyan, “The essential guide to load balancing,” 2022. [Online]. Available: <https://hayksimonyan.substack.com/p/the-essential-guide-to-load-balancing>
- [13] Amazon Web Services, “Elastic Load Balancing,” AWS Documentation, 2024. [Online]. Available: <https://docs.aws.amazon.com/elasticloadbalancing/>
- [14] Microsoft, “Azure Load Balancer overview,” Microsoft Azure Documentation, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview/>
- [15] Google Cloud, “Cloud Load Balancing overview,” Google Cloud Documentation, 2023. [Online]. Available: <https://cloud.google.com/load-balancing/docs/overview>

- [16] P. Varghese and R. Buyya, “Next generation cloud computing: New trends and research directions,” *Future Generation Computer Systems*, 2018. [Online]. Available: <https://arxiv.org/abs/1707.07452>
- [17] Amazon Web Services, “Amazon EC2 Auto Scaling,” AWS Documentation, 2024. [Online]. Available: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html>
- [18] Microsoft, “Virtual Machine Scale Sets,” Microsoft Azure Documentation, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/overview>
- [19] Google Cloud, “Managed Instance Groups,” Google Cloud Documentation, 2023. [Online]. Available: <https://cloud.google.com/compute/docs/instance-groups/managed-instance-groups>
- [20] Microsoft, “Azure Application Gateway,” Microsoft Azure Documentation, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/overview>
- [21] Google Cloud, “HTTP(S) Load Balancing,” Google Cloud Documentation, 2023. [Online]. Available: <https://cloud.google.com/load-balancing/docs/https/>
- [22] Microsoft, “Autoscale overview,” Microsoft Azure Documentation, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview>
- [23] Google Cloud, “Autoscaling,” Google Cloud Documentation, 2023. [Online]. Available: <https://cloud.google.com/compute/docs/autoscaler>
- [24] B. Snyder, “Cloud load balancing and scalability: Principles and practices,” *IEEE Internet Computing*, vol. 18, no. 4, pp. 61–65, 2014.
- [25] Amazon Web Services, “Elastic Load Balancing,” AWS Documentation, 2024. [Online]. Available: <https://docs.aws.amazon.com/elasticloadbalancing/>
- [26] Microsoft, “Autoscale overview,” Microsoft Azure Documentation, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview>
- [27] Google Cloud, “HTTP(S) Load Balancing,” Google Cloud Documentation, 2023. [Online]. Available: <https://cloud.google.com/load-balancing/docs/https/>
- [28] Microsoft, “Azure Application Gateway,” Microsoft Azure Documentation, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/overview>
- [29] M. Armbrust *et al.*, “A view of cloud computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010. [Communications of the ACM](#)
- [30] B. Varghese and R. Buyya, “Next generation cloud computing: New trends and research directions,” *Future Generation Computer Systems*, vol. 79, pp. 849–861, 2018. [arXiv](#)

[31] Amazon Web Services, “AWS Lambda Documentation,” 2024. [Online]. Available: <https://docs.aws.amazon.com/lambda/> [AWS Documentation+1](#)

[32] Microsoft, “Azure Functions Documentation,” 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-functions/>