

# **Cloud Engineering Principles and Practices**

## ***A Technical Report on Containerisation, Cloud Scalability, and Serverless Computing***

**Student Name:** Patel Diyaben Pradipbhai

**Student Number:** 2523935

**Module:** Cloud Engineering Principles and practices

**Submission Date:** 28<sup>th</sup> April 2026

**Video Link:** [https://youtu.be/Gxqe9JQvV3A?si=s\\_EcnjtASkh8xsnt](https://youtu.be/Gxqe9JQvV3A?si=s_EcnjtASkh8xsnt)

## Table of Content

|                                                                  |           |
|------------------------------------------------------------------|-----------|
| <b>1. Introduction .....</b>                                     | <b>4</b>  |
| <b>2. Containerization and Docker .....</b>                      | <b>4</b>  |
| 2.1 Principles of Virtualization .....                           | 4         |
| 2.2 Limitations of Virtualization.....                           | 5         |
| 2.3 Benefits of Containerization.....                            | 6         |
| 2.4 Technical Implementation Evidence .....                      | 6         |
| 2.4.1 Docker file Construction and Image Build.....              | 7         |
| 2.4.2 Building and Publishing Images.....                        | 7         |
| 2.4.3 Container Volumes .....                                    | 9         |
| 2.4.4 Container Networking.....                                  | 11        |
| 2.4.5 End-to-End Demonstration Summary .....                     | 12        |
| 2.5 Video Demonstration Link .....                               | 13        |
| <b>3. Core Cloud Computing Concepts .....</b>                    | <b>13</b> |
| 3.1 Load Balancing.....                                          | 13        |
| 3.1.1 Routing Algorithms.....                                    | 14        |
| 3.1.2 Layer 4 vs Layer 7 Load Balancing .....                    | 15        |
| 3.1.3 Health Checking and Self-Healing Infrastructure .....      | 15        |
| 3.1.4 AWS Implementation — Elastic Load Balancing.....           | 15        |
| 3.2 Scalability.....                                             | 16        |
| 3.2.1 Vertical Scaling (Scale Up/Down) .....                     | 16        |
| 3.2.2 Horizontal Scaling (Scale Out/In).....                     | 16        |
| 3.2.3 Autoscaling and Predictive Scaling .....                   | 16        |
| 3.2.4 AWS Implementation — EC2 Auto Scaling.....                 | 17        |
| 3.3 Application Gateways.....                                    | 17        |
| 3.4 Why Load Balancing and Scaling are Complementary .....       | 17        |
| 3.5 Benefits of Combining All Three Technologies .....           | 18        |
| <b>4. Driving Forces Behind Cloud Expansion .....</b>            | <b>19</b> |
| 4.1 Economic and Operational Drivers.....                        | 19        |
| 4.2 Serverless Computing — Functions as a Service.....           | 19        |
| 4.3 Serverless vs Containerization — A Technical Comparison..... | 20        |

**5. Conclusion.....21**

**6. References ..... 23**

## 1. Introduction

Cloud computing has changed how I think about IT infrastructure. It is no longer just a way to rent remote servers, but a full engineering discipline that covers automation, software-defined networking, security, and scalable deployment. Organizations that used to spend months provisioning hardware can now deploy globally in minutes, paying only for what they consume [\[1\]](#).

However, this shift brings real complexity. One of the most impactful developments in cloud engineering is containerization. Rather than bundling an application with a full virtual operating system, containers package only the application code and its runtime dependencies. This makes deployments consistent, portable, and fast – solving the classic problem of an application behaving differently in development versus production [\[22\]](#).

This report covers three interconnected areas. First, I evaluated the transition from traditional virtualization to Docker-based containerization, supported by practical lab evidence. Second, I analyzed the three core cloud concepts; load balancing, scalability, and application gateways, using AWS as the primary provider example. Third, I examined the business and technical forces driving cloud adoption, comparing serverless functions with containerization as competing and complementary deployment strategies.

## 2. Containerization and Docker

### 2.1 Principles of Virtualization

Virtualization transformed computing economics by letting a single physical machine host multiple isolated workload simultaneously. The concept began with IBM mainframe research in the 1960s, but it became mainstream when x86 Virtualization arrived in the late 1990s, making it practical to run multiple environments on commodity hardware [\[3\]](#).

At the core of any virtualized system is the hypervisor, also called a Virtual Machine Monitor (VMM). It sits between the hardware and the guest operating systems, intercepting privileged instructions and presenting each virtual machine (VM) with the illusion of a dedicated physical computer.

Type 1 (Bare-Metal) Hypervisors – Examples include VMware ESXi and Microsoft Hyper-V. These run directly on physical hardware, giving them low overhead, high performance, and strong isolation [\[4\]](#).

Type 2 (Hosted) Hypervisors – Examples include Oracle VirtualBox and VMware Workstation. They run as applications on a host OS, which introduces an extra layer of

overhead. I use Type 2 hypervisors for personal testing and development, but they are unsuitable for heavy production workloads.

Virtualization abstracts hardware across three dimensions: compute (using Intel VT-x or AMD-V to create isolated execution environments [\[3\]](#)), memory (using Extended Page Tables to translate guest addresses), and storage/networking.

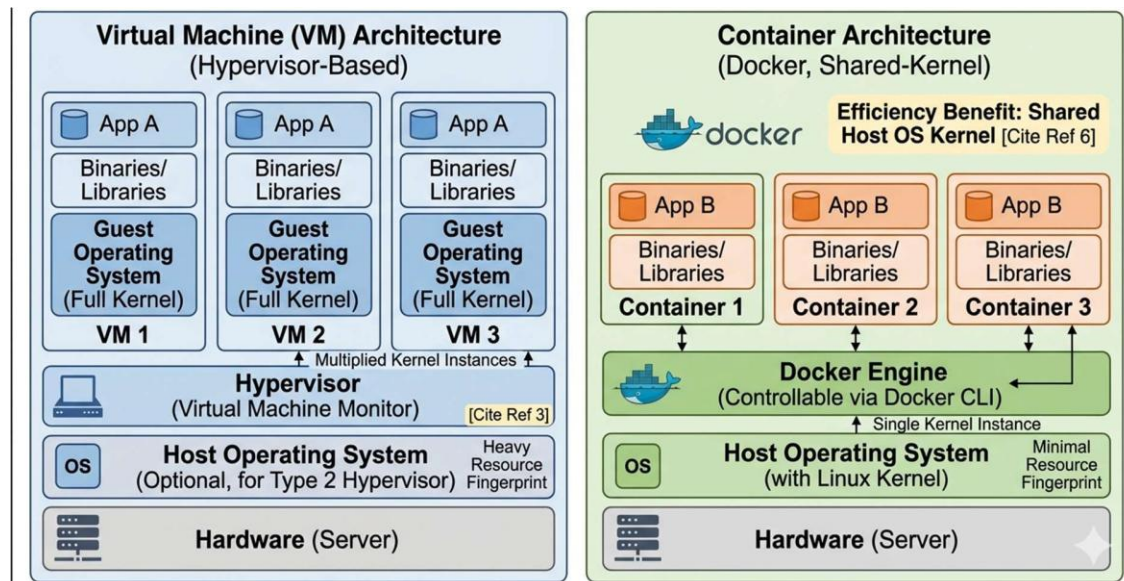
## 2.2 Limitations of Virtualization

While VMs were a major breakthrough, working with Docker in this module made the weight of the VM model immediately obvious.

- **The Guest OS Tax.** Every VM requires a full, independent operating system. Even a minimal Linux installation consumes 512 MB to 2 GB of RAM before the application code even starts [\[5\]](#).
- **Performance Latency.** A network packet generated by a guest OS must traverse multiple layers before reaching the physical hardware, increasing latency [\[5\]](#).
- **Slow Boot Times.** A VM must replicate a full hardware boot sequence, which takes significantly longer compared to container startup.
- **Management Sprawl and Security Risk.** Each VM requires independent updates and maintenance, increasing complexity and potential security vulnerabilities.
- **Density Limits and Infrastructure Cost.** Because each VM carries a full OS, fewer VMs fit on each physical host, increasing infrastructure cost [\[5\]](#).

As shown in Figure 1, the virtual machine model requires a full guest operating system for each instance, whereas container-based architecture shares a single host kernel, significantly improving efficiency and reducing overhead resource.

**Figure 1: VM vs Container Architecture – Illustrating the guest OS overhead of the VM model compared to the shared-kernel approach used by Docker [3], [5], [6]**



## 2.3 Benefits of Containerization

Containerization addresses the limitations of Virtualization by sharing the host operating system kernel while isolating applications using namespaces and control groups (cgroups) [6].

This approach significantly improves resource efficiency. Containers have a much smaller footprint compared to VMs, allowing a single host to run many more instances simultaneously. Additionally, containers start almost instantly, enabling rapid scaling in response to demand.

Another major advantage is portability. A Docker image contains all required dependencies, ensuring consistent behavior across development, testing, and production environments [22]. This eliminates compatibility issues and simplifies deployment processes.

Containerization also supports modern DevOps practices, including continuous integration and continuous deployment (CI/CD), by enabling reproducible and automated workflows.

What struck me most practically during this module is how containerisation shifts the conversation away from infrastructure entirely. Before working with Docker, I was thinking about which OS to install, which dependencies to configure, and whether the environment on my laptop would match the server. After building even a simple nginx container, that mental overhead disappeared. The environment is defined once in the Dockerfile and that is the end of the conversation. That shift in thinking — from managing environments to just writing application code — is arguably the most important cultural change containerisation brings to a

development team, not just a technical one.

## **2.4 Technical Implementation Evidence**

The following sections document the practical Docker implementation completed as part of this module.

### 2.4.1 Dockerfile Construction and Image Build

The Dockerfile defines how a container image is built. As shown in Figure 2, it specifies the base image, application files, and exposed ports.

**Figure 2: Dockerfile structure and content**

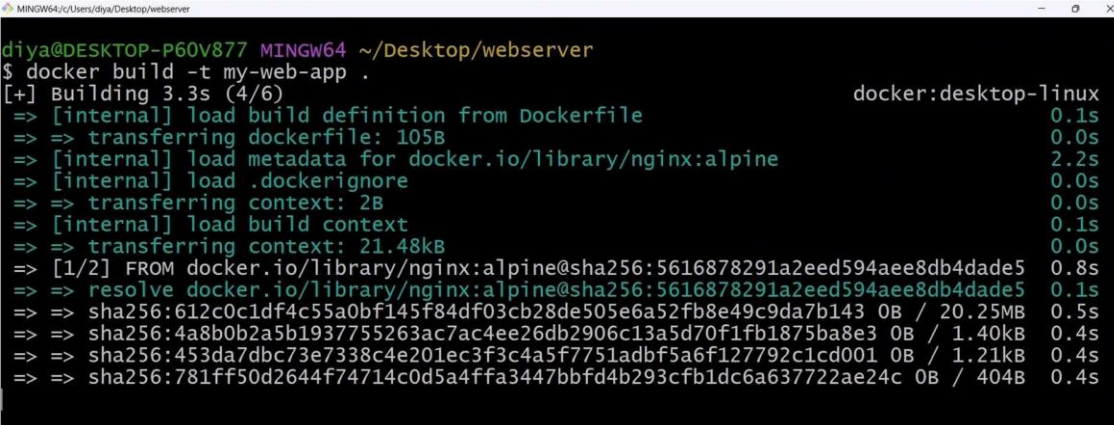
```
FROM nginx:alpine

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

EXPOSE 80
```

Figure 3 demonstrates the Docker build process and successful image creation.

**Figure 3: Docker build command output showing layer processing and successful image creation**



```
diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker build -t my-web-app .
[+] Building 3.3s (4/6)                                docker:desktop-linux
=> [internal] load build definition from Dockerfile    0.1s
=> => transferring dockerfile: 105B                    0.0s
=> [internal] load metadata for docker.io/nginx:alpine 2.2s
=> [internal] load .dockerignore                       0.0s
=> => transferring context: 2B                          0.0s
=> [internal] load build context                       0.1s
=> => transferring context: 21.48kB                     0.0s
=> [1/2] FROM docker.io/library/nginx:alpine@sha256:5616878291a2eed594aee8db4dade5 0.8s
=> => resolve docker.io/library/nginx:alpine@sha256:5616878291a2eed594aee8db4dade5 0.1s
=> => sha256:612c0c1df4c55a0bf145f84df03cb28de505e6a52fb8e49c9da7b143 0B / 20.25MB 0.5s
=> => sha256:4a8b0b2a5b1937755263ac7ac4ee26db2906c13a5d70f1fb1875ba8e3 0B / 1.40kB 0.4s
=> => sha256:453da7dbc73e7338c4e201ec3f3c4a5f7751adb5a6f127792c1cd001 0B / 1.21kB 0.4s
=> => sha256:781ff50d2644f74714c0d5a4ffa3447bbfd4b293cfb1dc6a637722ae24c 0B / 404B 0.4s
```

### 2.4.2 Building and Publishing Images

After building the Docker image locally, it can be tagged and pushed to Docker Hub so that it can be accessed and deployed from any environment [\[6\]](#).

The process begins with authentication, followed by tagging the image and pushing it to the repository.

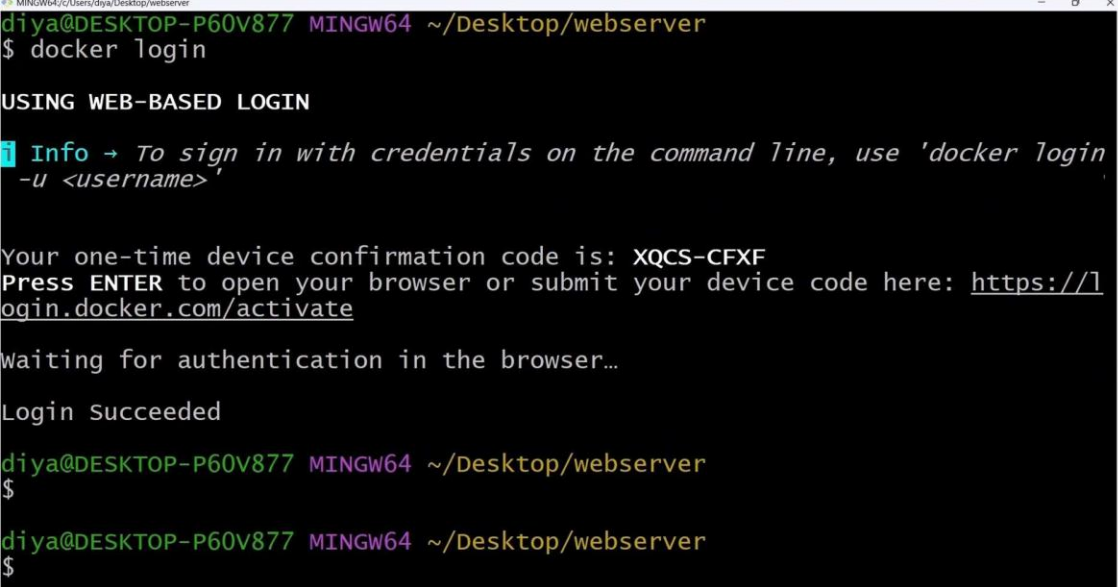
As shown in Figures 4 to 7, the process is carried out using the following commands:

```
docker login

docker tag my-web-app:1.0 <yourusername>/my-web-app:1.0

docker push <yourusername>/my-web-app:1.0
```

**Figure 4: Docker login command output showing successful authentication**



```
diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker login

USING WEB-BASED LOGIN

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

Your one-time device confirmation code is: XQCS-CFXF
Press ENTER to open your browser or submit your device code here: https://l
ogin.docker.com/activate

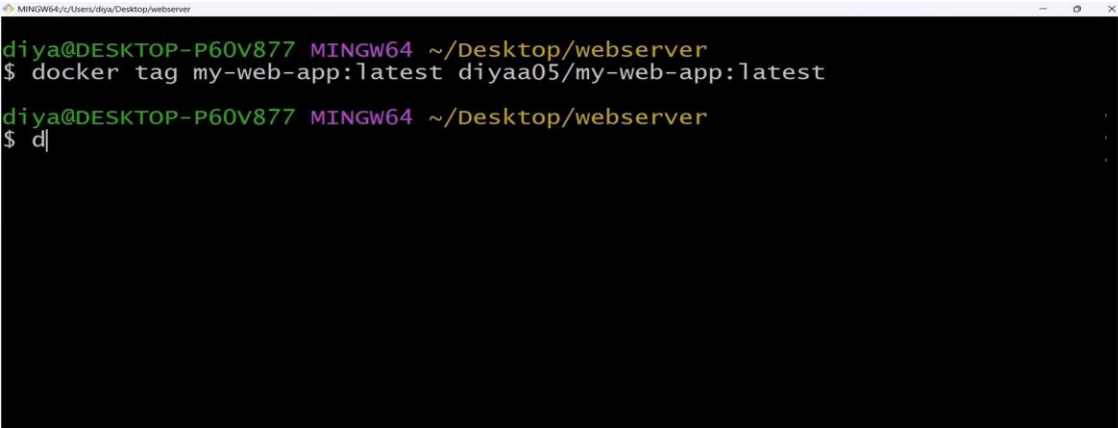
Waiting for authentication in the browser...

Login Succeeded

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$

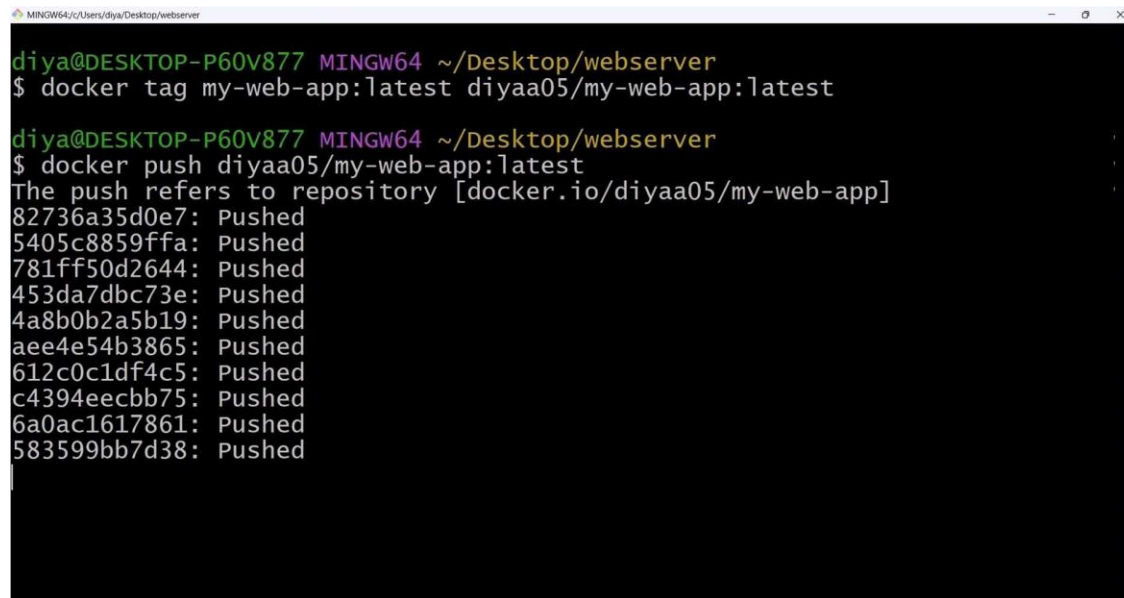
diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$
```

**Figure 5: Docker tag command applying repository naming convention**



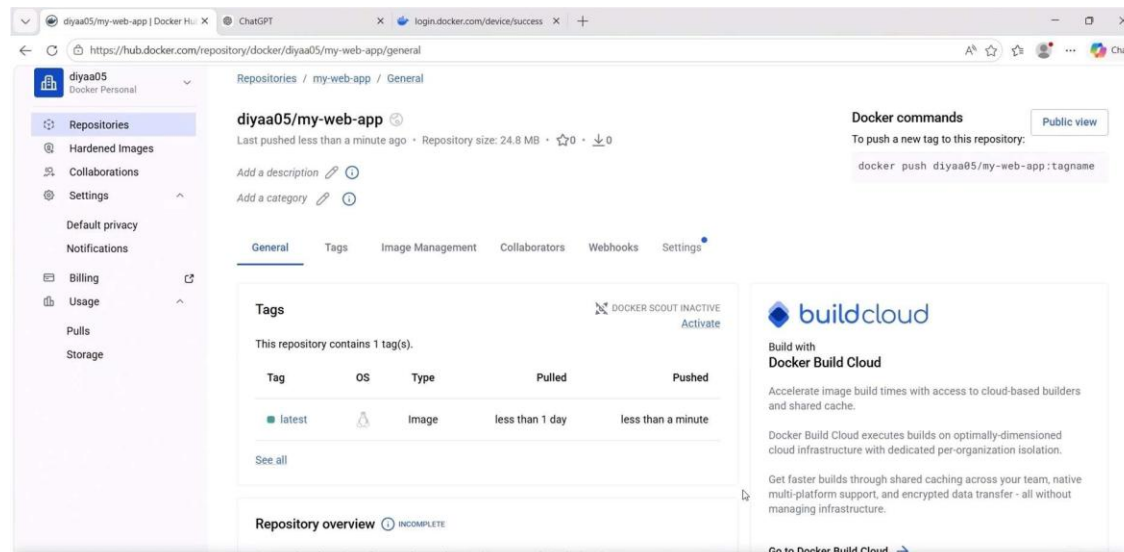
```
diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker tag my-web-app:latest diyaa05/my-web-app:latest

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ d|
```

**Figure 6: Docker push command output showing successful image layer upload**A terminal window with a black background and green and white text. The prompt is 'diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver'. The user enters '\$ docker tag my-web-app:latest diyaa05/my-web-app:latest'. Then they enter '\$ docker push diyaa05/my-web-app:latest'. The output shows 'The push refers to repository [docker.io/diyaa05/my-web-app]' followed by a list of 11 layers being pushed, each with a hash and the word 'Pushed'.

```
diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker tag my-web-app:latest diyaa05/my-web-app:latest

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker push diyaa05/my-web-app:latest
The push refers to repository [docker.io/diyaa05/my-web-app]
82736a35d0e7: Pushed
5405c8859ffa: Pushed
781ff50d2644: Pushed
453da7dbc73e: Pushed
4a8b0b2a5b19: Pushed
aee4e54b3865: Pushed
612c0c1df4c5: Pushed
c4394eecbb75: Pushed
6a0ac1617861: Pushed
583599bb7d38: Pushed
```

**Figure 7: Docker Hub repository showing uploaded image and tag details**

### 2.4.3 Container Volumes

Container filesystems are ephemeral by design. Any data written inside a running container is lost when the container is removed. For applications that need to persist state, Docker volumes provide storage that exists outside the container lifecycle [9].

Volumes allow data to remain intact even when containers are deleted and recreated, making them essential for production environments where persistence is required.

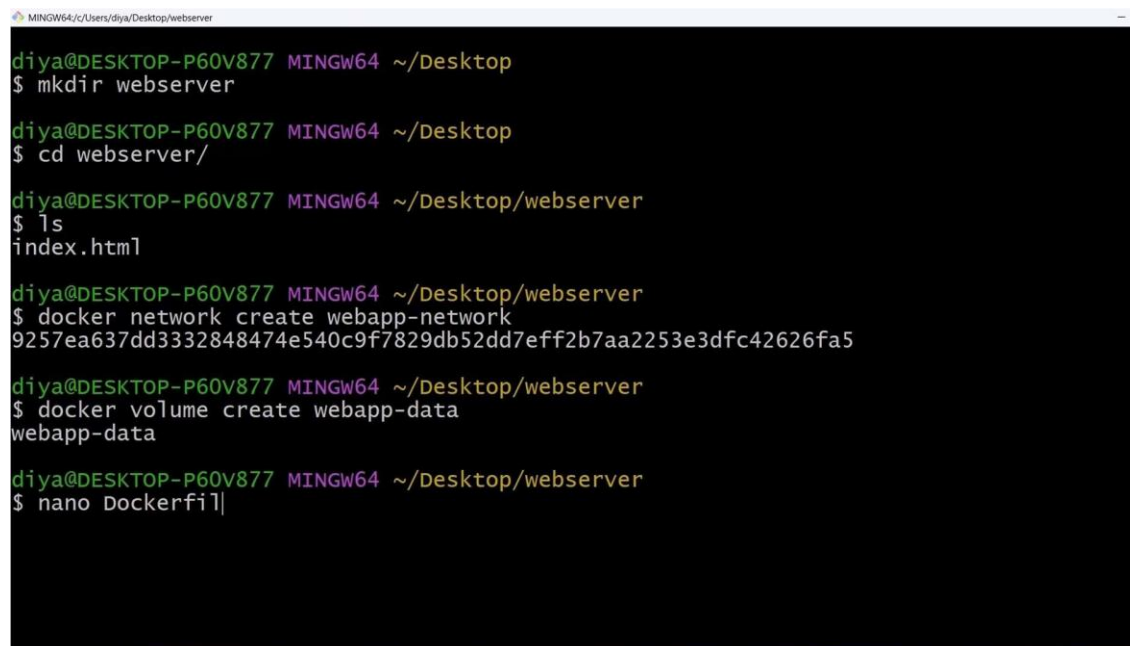
The following command was used to create a Docker volume:

```
docker volume create webdata

docker run -d --name my-web-app \
  --mount source=webdata,target=/usr/share/nginx/html \
  -p 8080:80 my-web-app:1.0
```

Figure 8 shows the creation and inspection of the Docker volume.

**Figure 8: Docker volume creation and inspection output**



```
MINGW64/c/Users/diya/Desktop/webserver
diya@DESKTOP-P60V877 MINGW64 ~/Desktop
$ mkdir webserver

diya@DESKTOP-P60V877 MINGW64 ~/Desktop
$ cd webserver/

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ ls
index.html

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker network create webapp-network
9257ea637dd3332848474e540c9f7829db52dd7eff2b7aa2253e3dfc42626fa5

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker volume create webapp-data
webapp-data

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ nano Dockerfil|
```

**Figure 9: Docker run command using the -v flag to mount the named volume and attach the custom network**

```

=> => extracting sha256:aee4e54b3865ee4d98545b6b49f9e8ab3b9b6ed114a2eafedc7a22e984 0.0s
=> => extracting sha256:781ff50d2644f74714c0d5a4ffa3447bbfd4b293cfeb1dc6a637722ae24 0.0s
=> => extracting sha256:453da7dbc73e7338c4e201ec3f3c4a5f7751adbf5a6f127792c1cd001a 0.0s
=> => extracting sha256:4a8b0b2a5b1937755263ac7ac4ee26db2906c13a5d70f1fb1875ba8e37 0.0s
=> => extracting sha256:612c0c1df4c55a0bf145f84df03cb28de505e6a52fb8e49c9da7b143fc 1.3s
=> [2/2] COPY index.html /usr/share/nginx/html/index.html 0.1s
=> exporting to image 0.5s
=> => exporting layers 0.2s
=> => exporting manifest sha256:f4d5aba8bd7216e4383a90d06e3fb25350dec7f8e8b8518464 0.0s
=> => exporting config sha256:52258caf26b12f825a9e9bcf86085eccc96f28065ce5ceed4cc8 0.0s
=> => exporting attestation manifest sha256:cfc51746d409ce1442798a2b5a9b717e4df24b 0.1s
=> => exporting manifest list sha256:5529f28aa4a9bf6e073252ac9b83d8b0b3eeec0d6bda1 0.0s
=> => naming to docker.io/library/my-web-app:latest 0.0s
=> => unpacking to docker.io/library/my-web-app:latest 0.1s

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker images



| IMAGE             | ID           | DISK USAGE | CONTENT SIZE | EXTRA | In Use |
|-------------------|--------------|------------|--------------|-------|--------|
| my-web-app:latest | 5529f28aa4a9 | 92.6MB     | 26MB         |       |        |



diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker run -d --name web-server --network webapp-network -v webapp-data:/usr/share/nginx/html -p 8080:80 my-web-app:latest
77c653dd3b0bf25f675fe217af6a7301fe752bd744a5a5ece9e4c201358f74ad

```

## 2.4.4 Container Networking

Docker networking enables communication between containers within isolated environments

[\[10\]](#).

By default, Docker uses a bridge network, but custom networks provide better control and allow containers to communicate using names instead of IP addresses.

The following command was used to create a custom network:

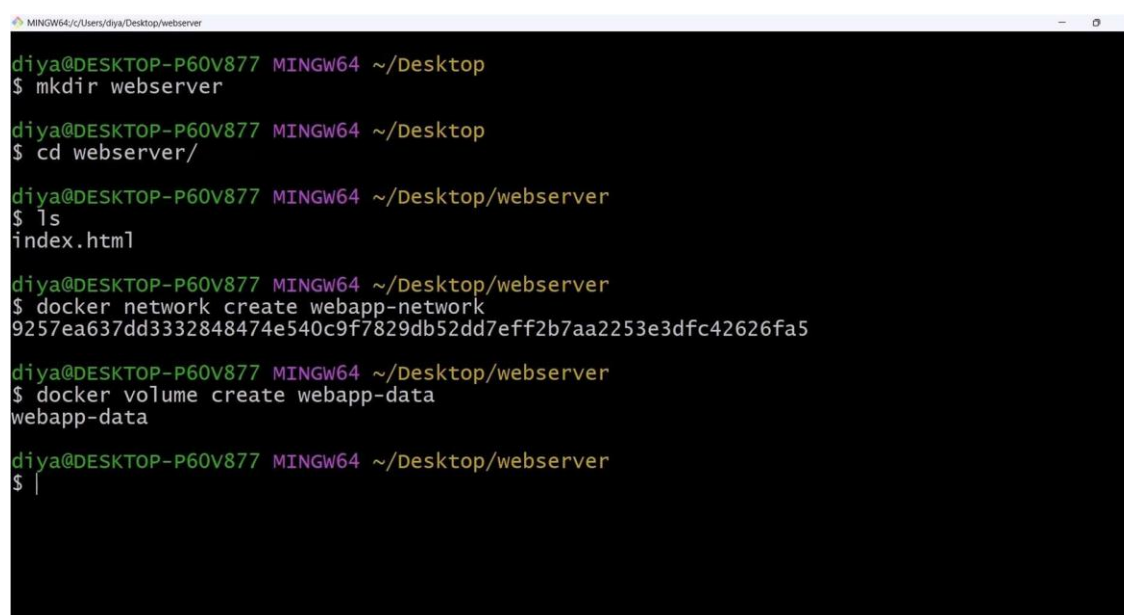
```

docker network create webapp-network

docker run -d --name my-web-app \
  --network webapp-network \
  -p 8080:80 my-web-app:1.0

```

Figure 10 shows the creation of the custom Docker network.

**Figure 10: Docker network creation command output**


```

MINGW64~/Users/diya/Desktop/webserver
diya@DESKTOP-P60V877 MINGW64 ~/Desktop
$ mkdir webserver

diya@DESKTOP-P60V877 MINGW64 ~/Desktop
$ cd webserver/

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ ls
index.html

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker network create webapp-network
9257ea637dd3332848474e540c9f7829db52dd7eff2b7aa2253e3dfc42626fa5

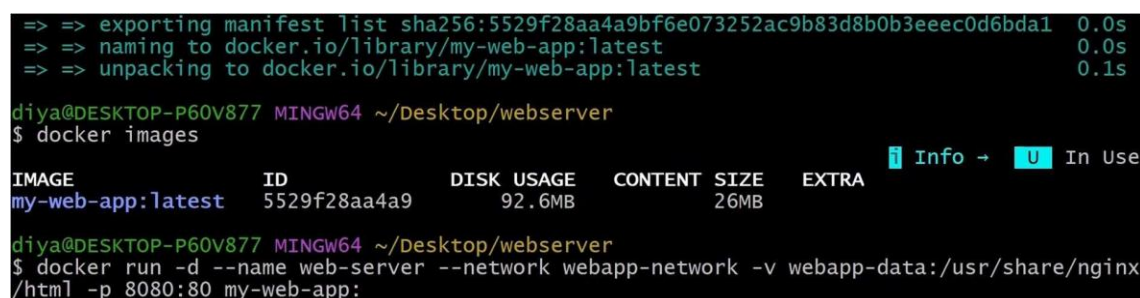
diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker volume create webapp-data
webapp-data

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ |

```

The network can then be inspected to view connected containers and configuration details.

Figure 11 shows the inspection of the Docker network.

**Figure 11: Docker network inspection showing connected containers and subnet configuration**


```

=> => exporting manifest list sha256:5529f28aa4a9bf6e073252ac9b83d8b0b3eeec0d6bda1 0.0s
=> => naming to docker.io/library/my-web-app:latest 0.0s
=> => unpacking to docker.io/library/my-web-app:latest 0.1s

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker images

IMAGE          ID                DISK USAGE  CONTENT SIZE  EXTRA
my-web-app:latest 5529f28aa4a9      92.6MB      26MB

diya@DESKTOP-P60V877 MINGW64 ~/Desktop/webserver
$ docker run -d --name web-server --network webapp-network -v webapp-data:/usr/share/nginx/html -p 8080:80 my-web-app:

```

### 2.4.5 End-to-End Demonstration Summary

Taken together, these steps demonstrate the complete lifecycle of a containerised application. A Dockerfile defines the environment, the image is built using the docker build command, and a container is run with volume and network configurations applied.

The image is then tagged and pushed to Docker Hub, making it accessible for deployment across different environments. This workflow reflects how modern cloud-native applications are developed, deployed, and managed.

## 2.5 Video Demonstration Link

The practical implementation demonstrated in this report is supported by a video walkthrough showing the full Docker workflow, including image creation, container execution, volume usage, networking, and publishing to Docker Hub.

Video Link: [https://youtu.be/Gxqe9JQvV3A?si=s\\_EcnjtASkh8xsnt](https://youtu.be/Gxqe9JQvV3A?si=s_EcnjtASkh8xsnt)

The video is divided into three phases that mirror a real deployment pipeline. Phase 1 covers Dockerfile construction and the build process, showing how each layer is cached and how the final image ID is produced. Phase 2 demonstrates volume creation and network configuration, explaining why persistent storage and named networks matter in multi-container systems. Phase 3 covers Docker Hub authentication, image tagging, and the push operation, ending with verification of the hosted repository.

## 3. Core Cloud Computing Concepts

To demonstrate load balancing, scalability, and application gateways in a realistic context, I use a containerized online retail application throughout this section. The application consists of a public web frontend, an API tier handling product and order services, and backend data services — a common microservices pattern that exercises all three concepts clearly.

### 3.1 Load Balancing

In any cloud architecture handling real traffic, a load balancer is non-negotiable. Its job is to distribute incoming requests across a pool of backend instances so that no single server becomes a bottleneck while others sit idle. Beyond raw traffic distribution, a modern load balancer provides health monitoring, automatic failover, and session management [\[15\]](#).

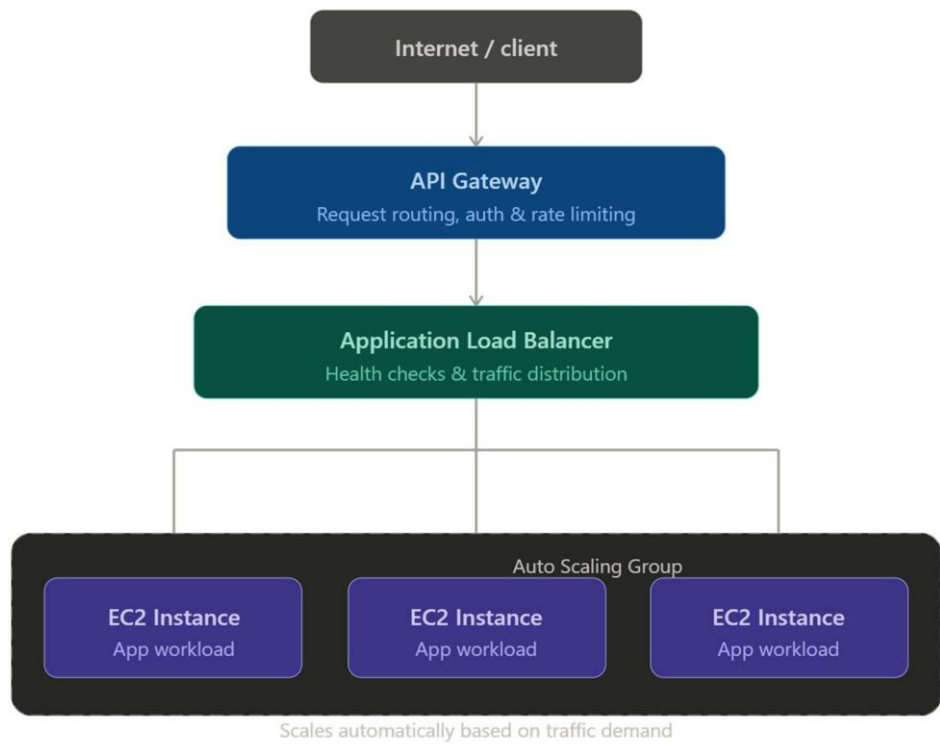


Figure D2 — API Gateway, Application Load Balancer and Auto Scaling Group

3.1.1 Routing Algorithms

The algorithm a load balancer uses to select a target server directly affects both performance and fairness. The six main strategies, in order of increasing sophistication, are:

Table 1: Load Balancing Algorithm Comparison [\[15\]](#)

| Algorithm            | How It Works                                          | Best Use Case                            |
|----------------------|-------------------------------------------------------|------------------------------------------|
| Round Robin          | Cycles through servers in fixed rotation              | Identical servers, uniform request times |
| Weighted Round Robin | Assigns proportional weights based on server capacity | Heterogeneous fleets                     |
| Least Connections    | Routes to server with fewest active connections       | Variable request processing times        |

|                            |                                                           |                                          |
|----------------------------|-----------------------------------------------------------|------------------------------------------|
| Weighted Least Connections | Combines capacity weights with real-time connection count | Most production workloads                |
| IP Hash                    | Maps client IP to a fixed server using a hash function    | Session persistence without shared cache |
| Least Response Time        | Directs traffic to server with lowest measured latency    | Latency-sensitive applications           |

### 3.1.2 Layer 4 vs Layer 7 Load Balancing

Layer 4 (Transport Layer) load balancers operate on IP addresses and TCP/UDP port numbers without inspecting packet content. This makes them extremely fast with minimal overhead — suitable for non-HTTP protocols and scenarios where raw throughput is the priority.

Layer 7 (Application Layer) load balancers inspect HTTP headers, URLs, and cookies to make intelligent routing decisions. This enables *content-based routing*: in my retail application, requests to `/api/*` route to application servers, while `/static/*` requests go directly to a CDN origin. This is essential for microservices architectures where different URL paths correspond to entirely different backend services.

### 3.1.3 Health Checking and Self-Healing Infrastructure

A load balancer continuously probes its backend pool with health checks — ranging from simple TCP connection tests to HTTP requests that verify the application returns a valid response body. When a backend fails its health check, the load balancer removes it from rotation and redistributes its traffic to healthy instances automatically, without operator intervention. When the server recovers, it is returned to the pool. This pattern, combined with autoscaling, produces infrastructure that heals itself [\[12\]](#) [\[13\]](#).

### 3.1.4 AWS Implementation — Elastic Load Balancing

AWS implements load balancing through Elastic Load Balancing (ELB), which provides three specialized services: the Application Load Balancer (ALB) for Layer 7 routing and microservices; the Network Load Balancer (NLB) for ultra-low-latency Layer 4 traffic; and the Gateway Load Balancer (GWLB) for routing through third-party security appliances. In my

retail example, ALB serves as the primary entry point, routing /products requests to the product service and automatically stopping traffic to any unhealthy instance [\[12\]](#).

### 3.2 Scalability

Scalability is the ability of a system to increase or decrease capacity in response to demand without requiring changes to application logic [\[17\]](#). Cloud platforms achieve this through two distinct but complementary strategies.

#### 3.2.1 Vertical Scaling (Scale Up/Down)

Vertical scaling increases the capacity of an existing instance by upgrading its CPU, memory, or storage. It requires no application changes, making it operationally simple. The drawbacks are significant though: there is a hard ceiling at the maximum hardware specification, scaling up typically requires downtime, and a single large instance is a single point of failure.

#### 3.2.2 Horizontal Scaling (Scale Out/In)

Horizontal scaling adds more instances of the same size. This is the foundational pattern of cloud-native architecture because it offers a virtually unlimited capacity ceiling, cost efficiency (small standard instances are cheaper per unit of compute than large, specialised ones), fault tolerance (one failed instance in a pool of ten reduces capacity by 10%, not 100%), and geographic distribution across multiple regions.

#### 3.2.3 Autoscaling and Predictive Scaling

Manual scaling is too slow for real-world traffic patterns. Autoscaling systems monitor metrics such as CPU utilization, request queue depth, and memory pressure, and automatically add or remove instances when thresholds are crossed. A rule such as "if average CPU exceeds 70% for two consecutive minutes, launch two additional instances" allows the system to absorb traffic spikes without any operator involvement [\[13\]](#).

Advanced platforms extend this with predictive scaling, using historical traffic patterns and machine learning to provision capacity before demand actually arrives. If traffic spikes every Friday at 6 PM, AWS predicts this and scales up at 5:45 PM — proactively rather than reactively.

### 3.2.4 AWS Implementation — EC2 Auto Scaling

AWS implements this through Amazon EC2 Auto Scaling, which manages *Auto Scaling Groups* — logical collections of instances with minimum, desired, and maximum capacity counts. Scaling policies include target tracking (keep CPU at 50%), scheduled scaling (pre-scale for a known event), and the predictive scaling described above. AWS automatically replaces any unhealthy instance and balances capacity across Availability Zones for resilience.

### 3.3 Application Gateways

While a load balancer distributes traffic across identical backend instances, an API Gateway manages the complexity of routing requests across architecturally distinct services in a microservices system [\[16\]](#). Without a gateway, every client would need to know the network address of every service and handle the complexity of authentication, rate limiting, and versioning themselves.

An API Gateway takes over five core responsibilities: request routing (matching URLs to the correct backend service), protocol translation (e.g., converting an external HTTP request into an internal gRPC call), security (centralizing authentication so individual services do not each implement it), rate limiting (preventing abuse by throttling high-volume clients), and observability (providing a single point to collect metrics and traces for all API traffic) [\[14\]](#).

AWS implements this through Amazon API Gateway, which supports REST, HTTP, and WebSocket APIs. It integrates natively with Lambda, EC2, and private load balancers, enforces JWT and IAM-based authorization, and generates detailed metrics in CloudWatch. In my retail example, API Gateway exposes `/login`, `/catalogue`, and `/checkout` endpoints while enforcing rate limits and forwarding each request to the correct backend service.

What I find interesting about API Gateways is that they make a microservices architecture feel manageable from the outside even when it is genuinely complex on the inside. A customer using my retail application has no idea that their single checkout request is being authenticated by the gateway, routed to an order service, which then talks to a payments service and an inventory service behind the scenes. From their perspective it is one seamless interaction. The gateway is what creates that illusion of simplicity, and I think that is an underappreciated part of what makes the pattern so powerful in practice.

### 3.4 Why Load Balancing and Scaling are Complementary

Load balancing and scaling are architecturally interdependent. Each delivers only part of its value without the other, and neither is a substitute for the other.

Scaling without load balancing: If autoscaling provisions five new instances in response to a

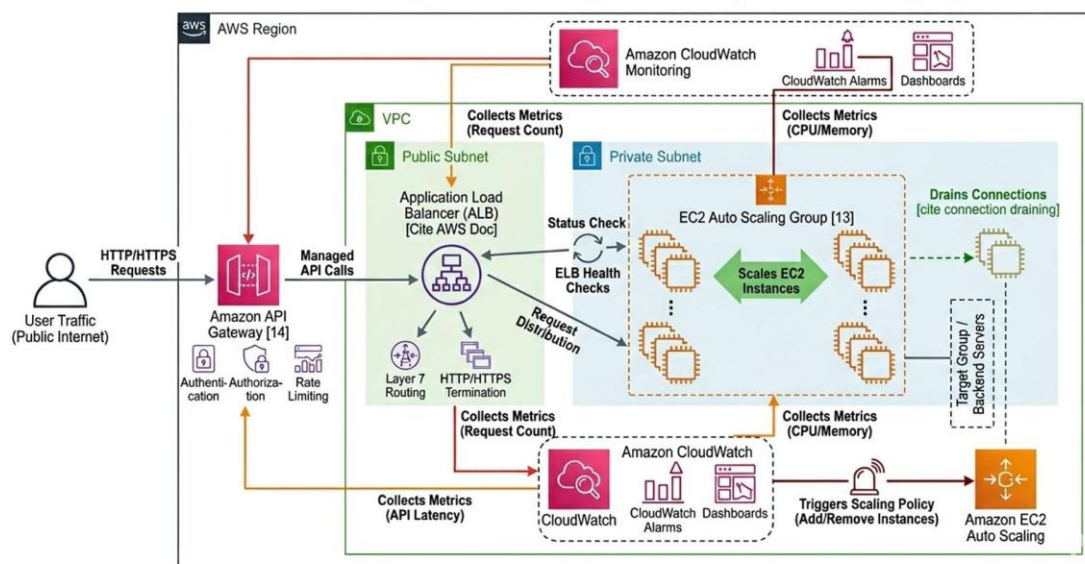
traffic spike, but there is no load balancer to direct traffic to those instances, they sit idle while the original instance continues to absorb every request. Capacity exists, but it cannot be reached.

**Load balancing without scaling:** A load balancer can distribute traffic perfectly across three instances, but when all three hit 100% CPU, the load balancer has nowhere to send additional requests. The system degrades regardless of how intelligent the routing algorithm is.

When integrated correctly, they form a closed feedback loop: a traffic spike triggers the autoscaling system, which provisions new instances, which pass health checks and register with the load balancer, which immediately starts distributing traffic to them. When the spike subsides, autoscaling scales in, and the load balancer gracefully drains connections from the instances being removed — preventing in-flight requests from being abruptly terminated.

**Figure 12: AWS cloud architecture showing the relationship between API Gateway, Application Load Balancer, EC2 Auto Scaling Group, and CloudWatch monitoring [12], [13] [14]**

**AWS Cloud Architecture: API Gateway, Load Balancing, Auto Scaling, and Monitoring Figure D2**



### 3.5 Benefits of Combining All Three Technologies

Using API Gateway, ALB, and EC2 Auto Scaling together creates a capability that none of them provides alone: a self-regulating, self-healing service that adapts to demand without human intervention.

For the customer, this means lower latency (fewer overloaded servers), greater resilience (failed instances are automatically replaced), stronger security (centralized authentication at the API layer), and a consistent experience even during traffic spikes. For the engineering team, it reduces operational burden dramatically — the system manages its own capacity, the load balancer manages its own backend pool, and the gateway manages its own security and routing policies. The team focuses on application code, not infrastructure management [12] [13] [14].

## 4. Driving Forces Behind Cloud Expansion

### 4.1 Economic and Operational Drivers

The shift to cloud computing is driven by both economic and operational forces that reinforce each other. The most fundamental is the move from capital expenditure (CapEx) to operational expenditure (OpEx). Instead of buying enough hardware to handle peak demand and leaving it underutilized for the other 364 days of the year, organizations pay for capacity as they consume it [\[18\]](#).

This economic shift has three practical consequences. First, it removes the barrier to entry for smaller organizations — a startup can access the same infrastructure as a Fortune 500 company on a per-hour basis. Second, it eliminates the risk of over-provisioning: there is no longer a business case to buy servers for a Black Friday traffic spike if the cloud can auto-scale to meet it and scale back down immediately after. Third, it enables rapid experimentation: if an idea fails, I delete the instances and pay only for the hours they ran.

Beyond cost, cloud adoption is driven by agility and global reach. Provisioning an environment now takes minutes rather than months of hardware procurement. Deploying to a data centre in Singapore or São Paulo is a configuration change, not a capital project. This speed of delivery has made DevOps and continuous deployment practices not just possible but standard.

### 4.2 Serverless Computing — Functions as a Service

A newer deployment model has emerged within cloud computing: serverless computing, also known as Functions as a Service (FaaS). In this model, I write a discrete function — a piece of code that performs a specific task — and the cloud provider handles everything else: provisioning, scaling, fault tolerance, and the OS runtime itself.

AWS Lambda is the most widely adopted FaaS platform [\[19\]](#). It executes functions in response to events — an HTTP request, an S3 file upload, a message arriving on an SQS queue — and scales from zero to thousands of concurrent executions automatically. Critically, it scales back to zero when idle, meaning I pay nothing when the function is not running.

The serverless model has two key technical characteristics that distinguish it from containers. First, billing is per invocation: I pay for execution time measured in milliseconds, not for allocated compute-time whether or not the function is actually doing work. For spiky or intermittent workloads, this can be dramatically cheaper than running persistent containers. Second, cold starts are a real consideration: if a function has not been invoked recently, the

provider must spin up a new micro-VM to host it, adding 100 ms to 1 second of latency to the first request [\[20\]](#) [\[21\]](#).

### 4.3 Serverless vs. Containerization — A Technical Comparison

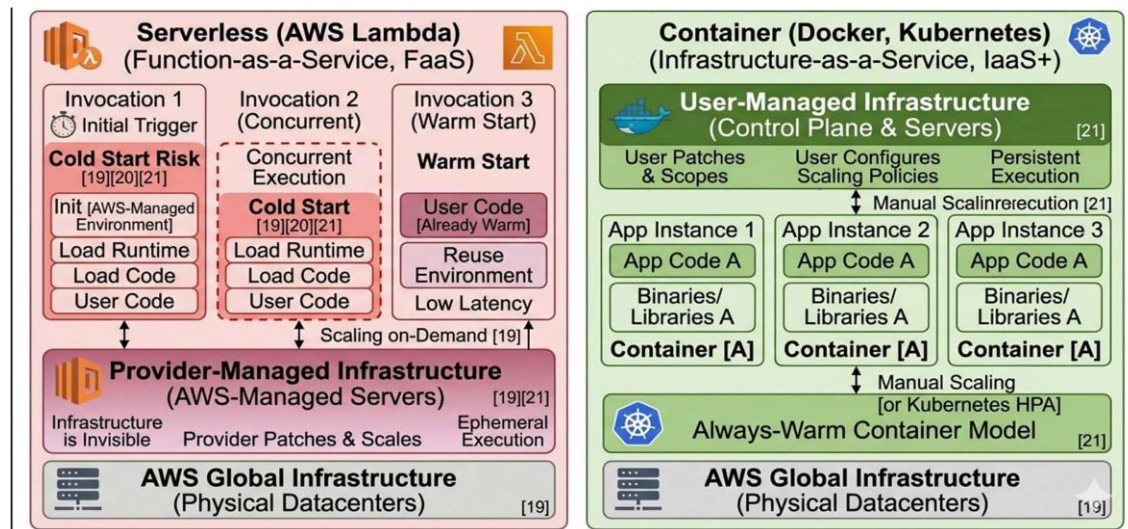
Both serverless and containerization address the same fundamental cloud adoption drivers — cost efficiency, agility, and faster delivery — but they optimize for fundamentally different constraints.

**Table 2: Containerisation vs. Serverless — Technical Comparison** [\[6\]](#) [\[19\]](#) [\[20\]](#) [\[21\]](#)

| Criterion           | Containerisation (Docker)                                            | Serverless Functions (Lambda)                                          |
|---------------------|----------------------------------------------------------------------|------------------------------------------------------------------------|
| Operational Control | Full control: I manage the Dockerfile, scaling rules, and networking | Provider manages runtime, OS, and scaling — I only write function code |
| Billing Model       | Pay for allocated compute-time while containers are running          | Pay per invocation and milliseconds of execution — \$0 when idle       |
| Startup Time        | Seconds (container process starts in running kernel)                 | Milliseconds to ~1s; cold starts can add latency after idle periods    |
| Runtime Duration    | Unlimited — containers run as long as required                       | Bounded — 15-minute maximum on AWS Lambda                              |
| Portability         | High — same image runs on any container runtime                      | Lower — design often tied to provider-specific event triggers          |
| Stateful Services   | Supported with volumes and external storage                          | Stateless by design; external storage required for all persistence     |

|          |                                                                |                                                                     |
|----------|----------------------------------------------------------------|---------------------------------------------------------------------|
| Best Fit | Long-running APIs, web servers, microservices, custom runtimes | Event processing, image resizing, scheduled tasks, lightweight APIs |
|----------|----------------------------------------------------------------|---------------------------------------------------------------------|

**Figure 13: Serverless (AWS Lambda) vs Container (Docker) execution models — illustrating the cold start risk and provider-managed infrastructure of FaaS compared to the always-warm container model [19][20] [21]**



In practice, the decision is not binary. Most organizations adopt a hybrid model: containers for core application services that require consistent, long-running behavior and portability, and serverless functions for asynchronous or event-triggered background tasks such as sending emails, processing uploaded images, or running scheduled reports. Kubernetes now supports serverless-style workloads through tools like Knative, and AWS Lambda can deploy functions packaged as container images — the two models are converging rather than competing.

## 5. Conclusion

This report has argued that cloud engineering is built on a set of interconnected architectural principles, not simply on renting remote infrastructure. The transition from traditional Virtualization to Docker-based containerization was the most significant shift in my practical understanding during this module. VMs were a breakthrough, but their guest OS overhead, slow boot times, and management complexity make them poorly suited to the fine-grained, rapidly changing workloads that cloud-native architectures demand.

The Docker implementation I completed — building a Dockerfile, managing volumes and custom networks, and publishing to Docker Hub — demonstrates how containerisation solves these problems in practice. The same image I built on my laptop is the artefact that runs in production. That consistency is what makes continuous deployment reliable.

Load balancing, autoscaling, and API gateways are not independent features; they form a single self-regulating system. An API Gateway governs access and routes traffic to the right backend; an ALB distributes that traffic across healthy instances; and Auto Scaling adjusts the size of that instance pool in real time. Together, they produce a service that is resilient, cost-efficient, and capable of handling unpredictable demand without operator intervention.

Finally, the growth of serverless computing does not replace containerization — it complements it. Containers provide the control, portability, and long-running capability that core application services need. Serverless functions provide the operational simplicity and per-invocation economics that event-driven background tasks benefit from. The cloud engineer's skill is knowing which tool fits the specific problem.

## 6. References

- [1] M. Armbrust et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: <https://dl.acm.org/doi/10.1145/1721654.1721672>
- [2] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014. [Online]. Available: <https://ieeexplore.ieee.org/document/7036275>
- [3] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, May 2005. [Online]. Available: <https://ieeexplore.ieee.org/document/1430627>
- [4] P. Barham et al., "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: <https://dl.acm.org/doi/10.1145/1165389.945462>
- [5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proc. IEEE ISPASS*, Philadelphia, PA, USA, Mar. 2015. [Online]. Available: [An updated performance comparison of virtual machines and Linux containers for ISPASS 2015 - IBM Research](#)
- [6] Docker Inc., "Docker overview," Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/get-started/docker-overview/>
- [7] Docker Inc., "Dockerfile reference," Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/reference/dockerfile/>
- [8] Docker Inc., "Building best practices," Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/build/building/best-practices/>
- [9] Docker Inc., "Volumes," Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/engine/storage/volumes/>
- [10] Docker Inc., "Networking overview," Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/engine/network/>
- [11] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May 2015. [Online]. Available: <https://www.researchgate.net/publication/281432589> [Containerisation and the PaaS Cloud](#)
- [12] Amazon Web Services, "Application Load Balancer," AWS Documentation, 2024. [Online]. Available: <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html>

- [13] Amazon Web Services, "What is Amazon EC2 Auto Scaling?" AWS Documentation, 2024. [Online]. Available: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html>
- [14] Amazon Web Services, "Amazon API Gateway features," AWS, 2024. [Online]. Available: <https://aws.amazon.com/api-gateway/features/>
- [15] NGINX, "Choosing a load balancing method," NGINX Blog, 2019. [Online]. Available: <https://www.nginx.com/blog/choosing-a-load-balancing-method/>
- [16] C. Richardson, "API Gateway pattern," Microservices.io, 2018. [Online]. Available: <https://microservices.io/patterns/apigateway.html>
- [17] 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. [Online]. Available: <https://doi.org/10.1016/j.future.2008.12.001>
- [18] J. R. Stormont and M. Fuller, *Cloud FinOps: Collaborative, Real-Time Cloud Financial Management*. Sebastopol, CA, USA: O'Reilly Media, 2022. [Book]
- [19] Amazon Web Services, "AWS Lambda features," AWS, 2024. [Online]. Available: <https://aws.amazon.com/lambda/features/>
- [20] I. Baldini et al., "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, Singapore: Springer, 2017, pp. 1–20. [Online]. Available: [https://link.springer.com/chapter/10.1007/978-3-319-54645-2\\_1](https://link.springer.com/chapter/10.1007/978-3-319-54645-2_1)
- [21] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, Feb. 2019. [Online]. Available: <https://arxiv.org/abs/1902.03383>
- [22] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, no. 239, Mar. 2014. [Online]. Available: <https://lwn.net/Articles/529999>
- [23] J. Lewis and M. Fowler, "Microservices: A definition of this new architectural term," *martinfowler.com*, 2014. [Online]. Available: <https://martinfowler.com/articles/microservices.html>
- [24] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[25] National Institute of Standards and Technology (NIST), Guide to Security for Full Virtualization Technologies, NIST Special Publication 800-125, Jan. 2011. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-125.pdf>

[26] Some figures included in this report were generated using AI-based tools (Google Gemini) for illustrative purposes.