



University of Wales
Trinity Saint David

Assignment

Cloud Engineering Principles: Containerisation, Scalability, and Load Balancing

Submitted By

Abdullah Al Noman

Student ID:

2520302

Module

Cloud Engineering Principles and Practice

(ACCA7027_LON)

Department

MSC Cybersecurity and Digital Forensics

Table of Contents

No.	Section Title	Page No.
	Abstract	i
1	Introduction	1
2	Principles of Virtualisation	4
3	Principles and Benefits of Containerisation	5
4	Docker Images and Docker file Construction	6
5	Building a Docker Image from a Docker file	8
6	Running and Verifying Containers	9
7	Publishing Docker Images to Docker Hub	12
8	Image Versioning and Lifecycle Management	13
9	Container Volumes and Persistent Storage	14
10	Container Networking in Distributed Systems	15
11	Hands-On Lab: Docker Networking and Service Communication	16
12	End-to-End Docker Application Demonstration	17
13	Video Demonstration and Assessment Alignment	18
14	Load Balancing, Scalability, and Application Gateways	18
15	Why Load Balancing and Scalability Are Complementary	20
16	Benefits of Combining Load Balancers, Scalability, and Application Gateways	21
17	Main Driving Forces Behind the Expansion of Cloud Computing	22
18	Serverless Computing Versus Containerisation	23
19	Conclusion	23
	References	24

Cloud Engineering Principles: Containerisation, Scalability, and Load Balancing

Abstract

Cloud computing has become the most popular way to host and deliver modern software systems because it lets users access computing resources on demand and in a flexible way. This paradigm allows businesses to make applications that are scalable, highly available, and cost-effective without the usual limits of physical infrastructure. Virtualization, containerization, load balancing, and scalability mechanisms are some of the most important technologies that make cloud environments possible. This report critically analyzes these technologies, primarily emphasizing containerization through Docker, as introduced and implemented throughout the module. The report starts by talking about the ideas behind virtualization and shared hardware abstraction.

Lastly, the report looks at the main reasons why cloud computing is growing, focusing on cost-effectiveness and operational flexibility. It also compares containerization to serverless computing methods like AWS Lambda and Azure Functions. This report shows how modern cloud systems combine theoretical ideas with real-world tools to provide services that are both reliable and scalable. It does this through critical discussion and hands-on demonstrations.

1. Introduction

The quick rise of cloud computing has changed the way businesses and consumers use computers, changing how applications are built, deployed, and managed. With traditional on-premises infrastructure models, businesses had to buy physical servers, set up fixed capacity, and keep up with ongoing maintenance. This meant high upfront costs and limited scalability. These environments often had problems with inefficient resource use because systems were usually set up for peak demand but ran at much lower average capacity. Cloud computing solves these problems by giving people access to shared, virtualized computing resources over the internet and charging them based on how much they use them [1]. Microsoft Azure and Amazon Web Services (AWS) are examples of cloud platforms that let businesses get computing resources on demand, quickly scale them up, and let them go when they're no longer needed. This change from capital expenditure (CapEx) to operational expenditure (OpEx) has made it much easier to try new things and come up with new ideas [15]. These are the basic ideas behind cloud platforms. It then talks about containerization and how it is better than traditional virtual machines. It shows how Docker makes things consistent, portable, and efficient.

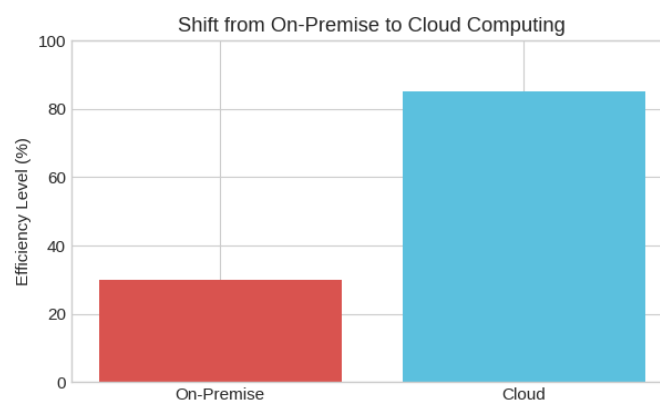


Figure 1. Comparison of Resource Utilisation in Traditional and Cloud Architectures

But just having cloud infrastructure doesn't mean that it will be efficient or scalable. How applications are packaged, deployed, and connected to each other is very important for how well a system works and how strong it is. This has made containerization technologies, especially Docker, very popular. These technologies make lightweight, portable application environments that are perfect for cloud deployment [3].

This report investigates the principles underpinning cloud engineering with emphasis on:

- Virtualisation of shared hardware
- Containerisation using Docker
- Practical Docker workflows derived from laboratory exercises
- Load balancing, scalability, and application gateway concepts
- The drivers behind cloud growth and serverless alternatives

By combining theoretical discussion with hands-on demonstrations, the report fulfils both the analytical and practical learning outcomes of the module.

2. Principles of Virtualisation

Virtualization is a basic technology that makes cloud computing possible by turning physical hardware resources into separate computing environments. It lets different operating systems and apps use the same physical infrastructure while still being logically separate. A hypervisor is what makes this abstraction possible. It controls how much CPU, memory, storage, and networking each virtual machine (VM) gets.

Virtualization lets several virtual machines run on one physical host while keeping workloads separate. This method is the basis for cloud infrastructure services. Figure 2 shows how a hypervisor lets virtual machines share physical hardware resources.

Virtualisation of Shared Hardware

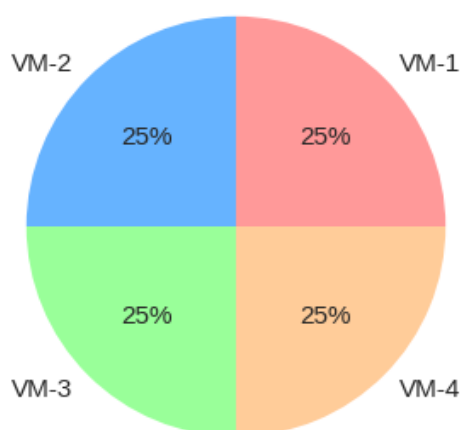


Figure 2. Virtualisation of shared physical hardware using a hypervisor and multiple virtual machines.

2.1 Virtualisation of Shared Hardware

In the past, servers in traditional computing environments were usually only used for one application or workload, which meant that the CPU and memory were not used very much. Virtualization fixes this problem by putting many workloads on the same hardware. Even though each virtual machine shares physical resources with other VMs, it runs its own operating system and applications. To provide Infrastructure as a Service (IaaS) offering, cloud service providers depend a lot on large-scale virtualization. Hypervisor-based virtualization technologies running in distributed data centers are used to create Azure Virtual Machines and AWS EC2 instances, for example [6], [8]. This method lets providers dynamically assign resources based on what customers want while keeping tenants separate.

Virtualisation provides several key benefits:

- Improved hardware utilisation through consolidation
- Isolation between workloads for security and stability
- Rapid provisioning and de-provisioning of compute resources
- Support for multiple operating systems on the same hardware

But virtualization also has some built-in limits. Each virtual machine needs a full guest operating system, which takes up more memory, storage, and processing power. It can take a while for VMs to boot up, and keeping track of operating system updates across many VMs adds to the work of system administrators [3]. These limitations become more of a problem in modern application architectures that depend on a lot of small, independent services. As we talked about in the lectures comparing virtual machines and containers, virtualization is still important for cloud infrastructure, but it isn't always the best way to deploy applications. This has led to more people using containerization as an extra technology that runs on top of virtualized infrastructure to make things work better and grow more easily.

3. Principles and Benefits of Containerisation

Containerization is a lightweight way to virtualize that works at the operating system level instead of the hardware level [2]. Containers don't need separate guest operating systems like virtual machines do. Instead, they share the kernel of the host operating system while keeping separate runtime environments for applications. Virtual machines make better use of hardware, but they also add overhead because they need full guest operating systems. Containerization cuts down on this extra work by sharing the host kernel. Figure 3 shows this difference by virtual machines and containers.

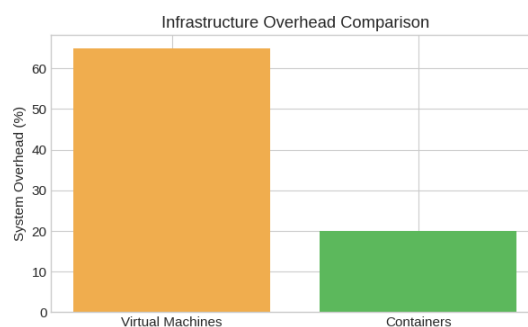


Figure 3. Infrastructure overhead comparison between virtual machines and containers.

Docker is the most popular containerization platform, and we used it a lot during the practical sessions of this module. Docker makes it easy to package apps and their dependencies into containers that will work the same way in any environment [3], [5].

3.1 How Containerisation Works

A container holds an application, its runtime, libraries, and configuration files. It uses the host operating system kernel for system-level functions. Compared to virtual machines, this shared kernel model uses a lot less resources and lets you start up quickly [2].

Microservices architectures, where applications are broken down into a set of loosely connected, independently deployable services, are a great fit for containers. Each microservice can run in its own container, which lets teams update, scale, and manage parts of the system separately [11].

3.2 Benefits of Containerisation in Cloud Environments

The adoption of containerisation in cloud platforms is driven by several important benefits.

Portability:

Containers ensure applications behave consistently across development, testing, and production environments, eliminating failures caused by environmental differences [3].

Efficiency:

Because containers are lightweight and share the host kernel, they consume fewer resources than virtual machines, allowing higher workload density on cloud infrastructure [2].

Rapid Deployment and Scaling

Containers can be started or stopped within seconds, enabling horizontal scaling in response to user.

DevOps Enablement:

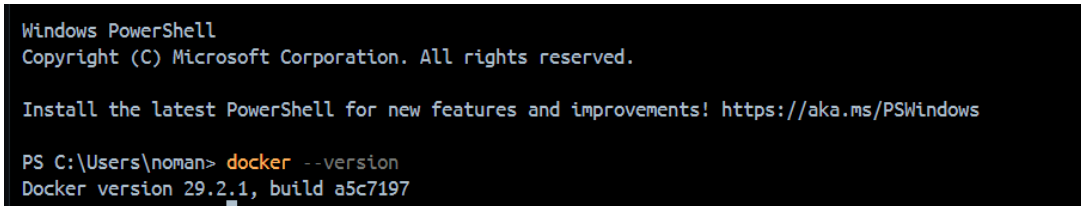
Containerisation aligns closely with DevOps practices by enabling continuous integration and continuous deployment (CI/CD) using immutable, versioned application artifacts [16].

Cloud-Native Compatibility:

Containers integrate naturally with cloud services such as load balancers, orchestration systems, and distributed networking, making them a core component of cloud-native application design.

4. Docker Images and Dockerfile Construction

Docker images are the building blocks of container-based deployment and are a key practical learning goal of this module. A Docker image is a template that can't be changed and can only be read. It has everything needed to run an application, such as the application code, the runtime, system libraries, configuration files, and environment variables [5]. Before doing any containerization tasks, the Docker environment was checked to make sure that Docker was installed and working properly.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\noman> docker --version
Docker version 29.2.1, build a5c7197
```

Figure 4. Verification of Docker installation and version.

Docker uses a client-server architecture that keeps user interaction separate from running containers and managing images. This design makes it possible to move things around and automate tasks in the cloud. Figure 5 shows a simpler picture of how Docker's client, daemon, and registry work together.

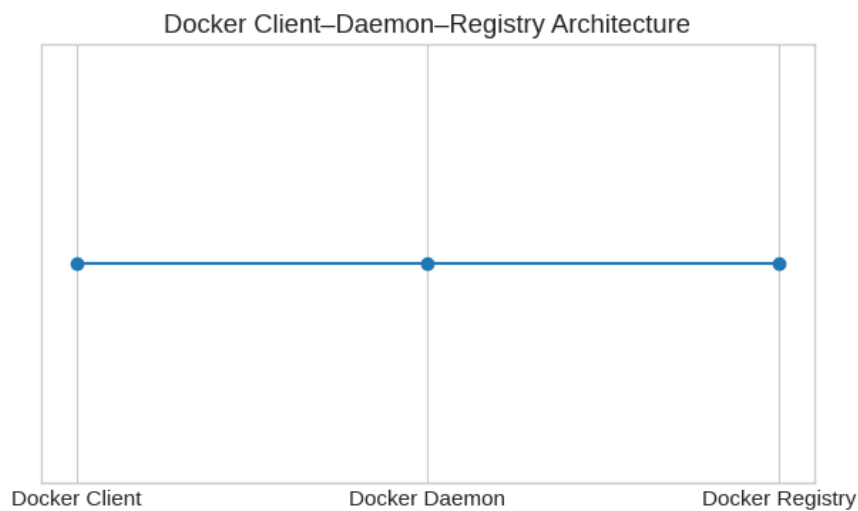


Figure 5. Docker client–daemon–registry architecture and communication flow.

A Dockerfile is used to make images. It tells the Docker engine what steps to take to build the image. Docker images let you define application environments in a way that can be repeated and automated. This is especially important in the cloud, where applications may be deployed many times on different hosts, in different regions, or in different availability zones.

4.1 Docker file Structure and Image Layering

A Dockerfile is a text file that has a list of commands that are run one after the other during the image build process. Most of the time, each instruction makes a new image layer, which is stored in memory and used again if the instruction or its inputs don't change in later builds. We checked the Docker system information to make sure that the Docker daemon was working properly.

```

Terminal
PS C:\Users\noman> docker info
Client:
Version:      29.2.1
Context:      desktop-linux
Debug Mode:   false
Plugins:
ai: Docker AI Agent - Ask Gordon (Docker Inc.)
  Version:    v1.18.0
  Path:       C:\Program Files\Docker\cli-plugins\docker-ai.exe
buildx: Docker Buildx (Docker Inc.)
  Version:    v0.31.1-desktop.1
  Path:       C:\Program Files\Docker\cli-plugins\docker-buildx.exe
compose: Docker Compose (Docker Inc.)
  Version:    v5.0.2
  Path:       C:\Program Files\Docker\cli-plugins\docker-compose.exe
debug: Get a shell into any image or container (Docker Inc.)
  Version:    0.0.47
  Path:       C:\Program Files\Docker\cli-plugins\docker-debug.exe
  
```

Figure 6. Docker engine status and system configuration details.

A layered filesystem model is used to build Docker images, with each layer representing a build instruction. This method makes builds faster and easier to reuse. Figure 7 shows how a Docker image is made up of layers.

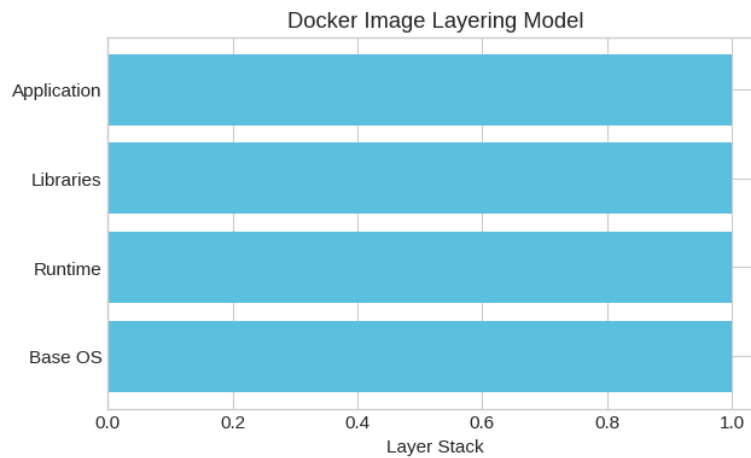


Figure 7. Layered structure of a Docker image illustrating the image build model.

Key Docker file instructions include:

- **FROM** – specifies the base image
- **COPY / ADD** – copies files into the image
- **RUN** – executes commands during build
- **EXPOSE** – documents network ports
- **CMD / ENTRYPOINT** – defines the runtime command

By leveraging layering and caching, Docker minimises rebuild time and reduces image distribution overhead, which is critical in continuous integration pipelines [5].

5. Building a Docker Image from a Docker file

The goal of this lab was to show how to use Docker to containerize a simple web app from start to finish. This task directly meets the requirement for the assessment to show how to build a Dockerfile and make an image. A separate folder for the project was made to keep the application files and Docker settings in order.

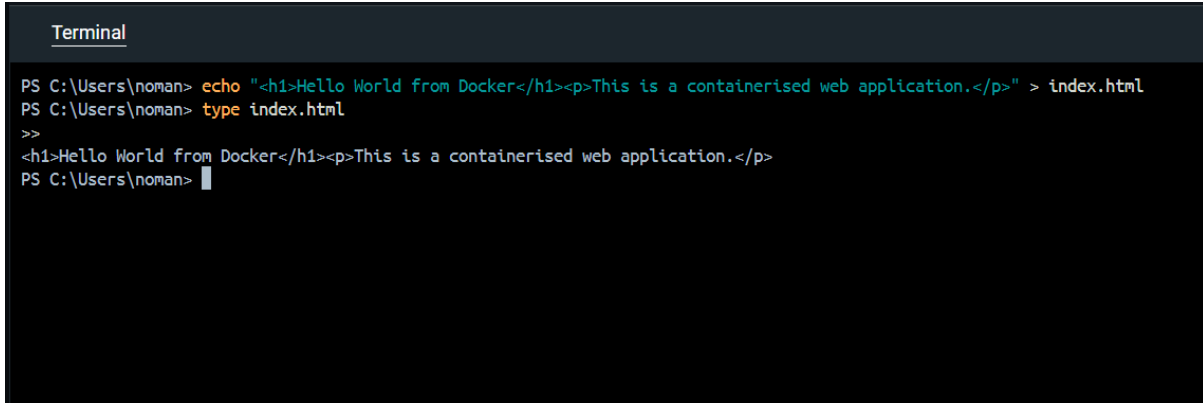
```
PS C:\Users\noman> mkdir docker-hello-app
>> cd docker-hello-app

Directory: C:\Users\noman

Mode                LastWriteTime         Length Name
----                -
d-----          27/04/2026   01:57             docker-hello-app
```

Figure 8. Creation of project directory for Docker application.

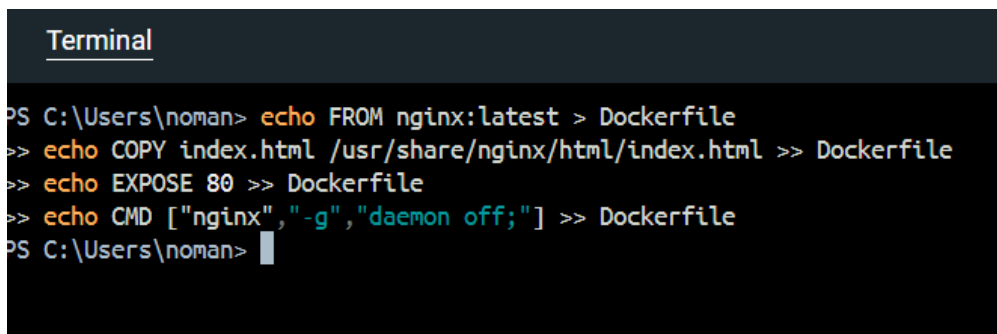
A project directory was created containing a simple HTML file, which was served using the official Nginx base image. The Dockerfile defined the runtime environment and application content, and the image was built using the Docker CLI. The resulting image was reusable, portable, and suitable for deployment across different environments. A simple “Hello World” web application was created to demonstrate a basic containerised web service, as required by the module.



```
Terminal
PS C:\Users\noman> echo "<h1>Hello World from Docker</h1><p>This is a containerised web application.</p>" > index.html
PS C:\Users\noman> type index.html
>>
<h1>Hello World from Docker</h1><p>This is a containerised web application.</p>
PS C:\Users\noman>
```

Figure 9. Creation of a basic Hello World web application.

This exercise shows how containerization makes it easier to deploy applications by putting configuration and dependencies in one place. In cloud environments, the same image can be used on development machines, test environments, or production virtual machines without any changes. This makes it easy to move and reproduce. To set up the container runtime environment and application settings, a Dockerfile was made.



```
Terminal
PS C:\Users\noman> echo FROM nginx:latest > Dockerfile
>> echo COPY index.html /usr/share/nginx/html/index.html >> Dockerfile
>> echo EXPOSE 80 >> Dockerfile
>> echo CMD ["nginx", "-g", "daemon off;"] >> Dockerfile
PS C:\Users\noman>
```

Figure 10. Docker file defining the containerised web application.

6. Running and Verifying Containers

After a Docker image is constructed, it can be run as a container. Containers are operational instances of images that signify separated application environments. The containerized web application was initiated in detached mode with port mapping from host to container. Accessing the application via a web browser confirmed that the container was operating correctly and that the network configuration was performing as intended. The inventory of locally accessible Docker images was confirmed subsequent to the image construction procedure.

A Dockerfile is a textual document that enumerates the procedures required to create a Docker image. This example illustrates a Dockerfile for a fundamental Node.js web application that delivers a static HTML page. This example serves as the foundation for the practical demonstrated and its accompanying video.

```

GNU nano 8.7.1 Dockerfile *
# Base image
FROM node:18-alpine

# Set working directory
WORKDIR /app

# Copy dependencies first (layer cache trick)
COPY package*.json ./

# Install dependencies
RUN npm install --production

# Copy rest of the app
COPY . .

# Expose port
EXPOSE 3000

# Start command
CMD ["node", "server.js"]

```

Figure 11. Docker file for a Node.js web application

Every command in a Docker file makes a layer of an image. The base image specified in the FROM instruction is the official Node.js 18 Alpine version. It occupies less space, approximately 60 MB, in contrast to around 800 MB for the Debian-based variant. Docker's layer caching accelerates builds by retaining packages. JSON is less frequently modified than application source code [10]. The sequence of the COPY and RUN instructions capitalizes on this. The accompanying Node.js server.js file and index.html are given below-

```

Session Actions Edit View Help
GNU nano 8.7.1 server.js *
const express = require('express');
const path = require('path');
const app = express();

app.use(express.static(path.join(__dirname, 'public')));

app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

app.listen(3000, () => console.log('Server listening on port 3000'));

```

Figure 12. Node.js Express server

```

Session Actions Edit View Help
GNU nano 8.7.1 public/index.html *
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Docker Demo</title>
</head>
<body>
  <h1>Hello from Docker!</h1>
  <p>This page is served from a Docker container.</p>
</body>
</html>

```

Figure 13. HTML page by the containerised web application

The image is created with the following command, whereby the -t flag designates the image with a comprehensible name and version.

```

Session Actions Edit View Help

(root@cybersec)-[/home/cybersec/docker_app]
# docker build -t docker_app:1.0 .
[+] Building 26.1s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 335B
=> [internal] load metadata for docker.io/library/node:18-alpine
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/5] FROM docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8ca09d9e
=> [internal] load build context
=> => transferring context: 287B
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] WORKDIR /app
=> CACHED [4/5] WORKDIR /app
=> CACHED [5/5] WORKDIR /app
=> CACHED [6/5] WORKDIR /app
=> CACHED [7/5] WORKDIR /app
=> CACHED [8/5] WORKDIR /app
=> CACHED [9/5] WORKDIR /app
=> CACHED [10/5] WORKDIR /app

```

Figure 14. Build Docker image

After making the picture, you may check it by using the command "docker images," which shows you all the pictures on your computer, along with their tags and sizes. After that, an image starts a container.

```

Session Actions Edit View Help

(root@cybersec)-[/home/cybersec/docker_app]
# docker run -d -p 8080:3000 --name mycontainer docker_app:1.0
8aed55a62cc9d0a1b1829d8d38c50d946782567d96bbb300cce5fa55e6e3588b

(root@cybersec)-[/home/cybersec/docker_app]
# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                               NAMES
8aed55a62cc9   docker_app:1.0 "docker-entrypoint.s..." 15 seconds ago Up 11 seconds 0.0.0.0:8080->3000/tcp, [::]:8080->3000/tcp mycontainer

(root@cybersec)-[/home/cybersec/docker_app]
#

```

Figure 15. Run Docker Container

The -d flag tells the container to run as a background daemon, -p 8080:3000 connects port 8080 on the host to port 3000 inside the container, and --name gives it a name that users can understand. If shown and go to <http://localhost:8080> in a web browser, it can be seen that the web app is running inside the container [10].

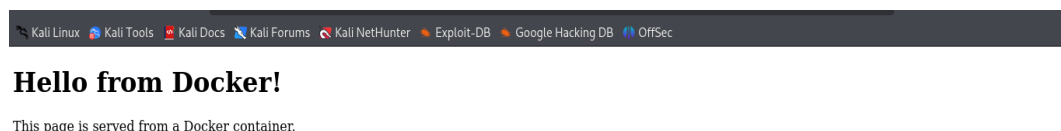


Figure 16. Run web application in a browser

This verification step is essential in cloud environments, where automated monitoring and health checks rely on consistent application behaviour.

7. Publishing Docker Images to Docker Hub

Publishing Docker images to a container registry is a basic step in current cloud-based deployment pipelines. A container registry is a central place to store, version, and share Docker images. This makes

it possible for apps to be deployed in the same way in different settings. During the lab sessions for this lesson, we used Docker Hub, which is the most popular public registry.

When you deploy Docker images in the cloud, they go through a defined lifecycle from creation to execution. To keep integrating and deploying, you need to know this lifecycle. Figure 17 shows the main steps in the life cycle of a Docker image.

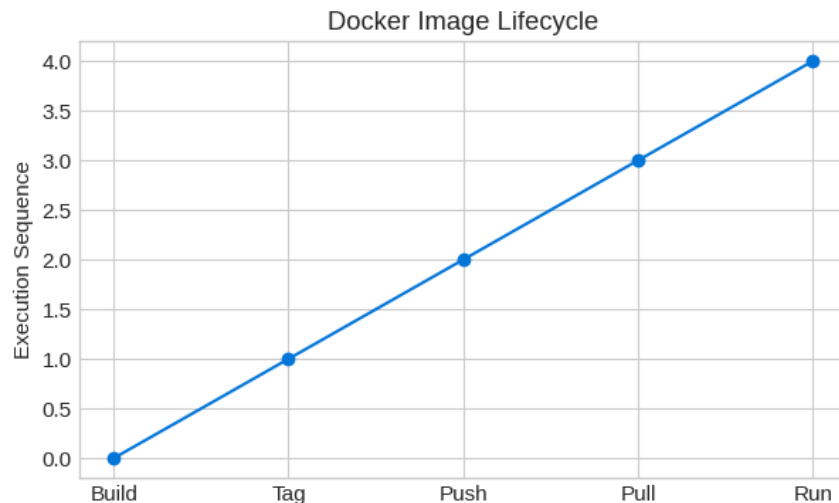


Figure 17. Docker image lifecycle from build and tag to deployment and execution.

7.1 Importance of Image Registries in Cloud Environments

In cloud systems, applications are seldom constructed and performed on the same server. Images are generally constructed in a development or continuous integration environment before being delivered to testing or production systems. Image registries separate the build and deployment phases, enabling images to be generated once and utilized several times [5].

The use of registries supports several key cloud engineering practices:

- Continuous integration and continuous deployment (CI/CD)
- Versioned and immutable application releases
- Rollback and disaster recovery strategies
- Distributed team collaboration

These practices are essential for scalable cloud systems, where deployments must be repeatable, auditable, and resilient to failure.

7.2 Publishing an Image to Docker Hub

To demonstrate image publishing, the Docker image created during the earlier laboratory exercise was uploaded to Docker Hub.

The process involved three main steps:

1. Authentication

The Docker CLI was authenticated against Docker Hub using the docker login command.

2. Image Tagging

The locally built image was tagged with a repository name and version identifier:

3. `docker tag my-docker-web:1.0 username/my-docker-web:1.0`

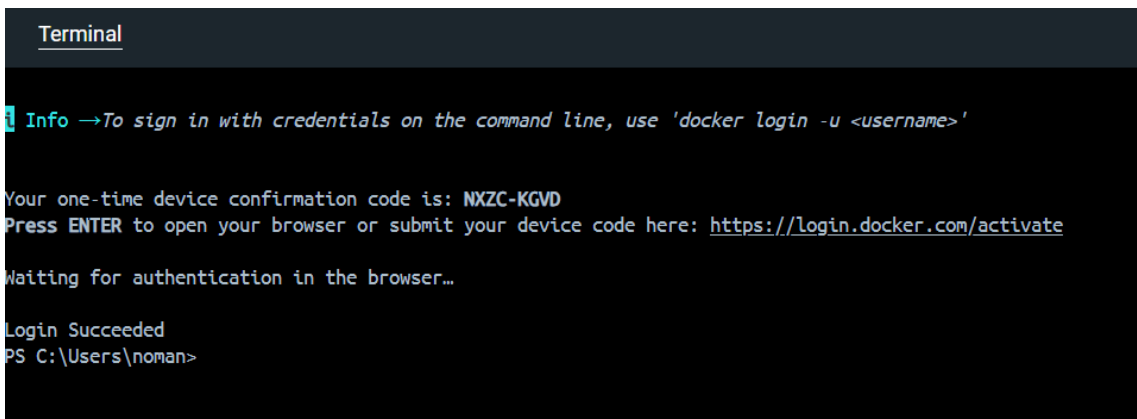
4. Image Push

The image was pushed to Docker Hub using:

5. `docker push username/my-docker-web:1.0`

Once published, the image became globally accessible and could be pulled onto any cloud host with Docker installed.

As a first step, authentication with Docker Hub was performed using the Docker CLI to enable image publishing.

A terminal window titled 'Terminal' with a dark background. It shows the output of the 'docker login' command. The text includes an info message about signing in, a one-time device confirmation code 'NXZC-KGVD', a URL to activate the account, and a confirmation that the login succeeded. The prompt shows the user is 'noman' on a Windows system.

```
Terminal

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

Your one-time device confirmation code is: NXZC-KGVD
Press ENTER to open your browser or submit your device code here: https://login.docker.com/activate

Waiting for authentication in the browser...

Login Succeeded
PS C:\Users\noman>
```

Figure 18. Authentication with Docker Hub using Docker login.

Critical Evaluation

Distributing images to Docker Hub emphasizes the concept of immutable infrastructure, wherein application instances are substituted rather than being altered on-site [3]. This method mitigates configuration drift and enhances system stability, especially in extensive cloud systems.

8. Image Versioning and Lifecycle Management

Versioning of Docker images is essential for managing application lifecycles on cloud platforms. Each image tag signifies a distinct, unalterable version of an application, facilitating exact control over deployments in each environment.

Versioned images enable:

- Controlled application rollouts
- Blue-green and canary deployments
- Rapid rollback in case of failure

For cloud systems requiring constant availability, versioning is a critical risk reduction method. By utilizing designated image versions, organizations mitigate unforeseen behavior resulting from unverified modifications and can promptly address operational challenges [5].

9. Container Volumes and Persistent Storage

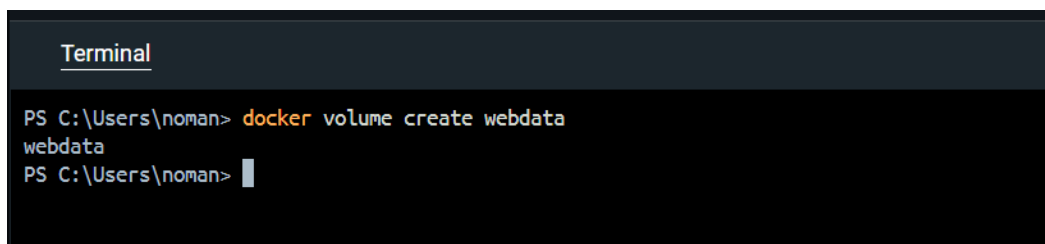
Containers are inherently fleeting in nature. When a container is halted or removed, the data contained on its internal filesystem is irretrievably lost. This behavior is advantageous for stateless services; nevertheless, real-world applications often necessitate persistent storage for data, including databases, configuration files, logs, and user-generated content.

Containers are inherently transient, signifying that data retained within a container is obliterated upon its removal. Docker volumes mitigate this problem by facilitating persistent storage. Figure 7 illustrates the persistence of data across container restarts when utilizing volume.

Docker volumes provide a structured method for addressing this limitation and are therefore a critical component of containerised cloud architectures.

9.1 Concept of Docker Volumes

Docker volumes are storage entities that are produced and regulated by the Docker engine. In contrast to container filesystems, volumes operate independently of the lifecycle of any individual container. A volume may be linked to one or several containers and can be utilized during container restarts, replacements, and upgrades. A Docker named volume was established to facilitate enduring data storage beyond the lifespans of containers.



```
Terminal
PS C:\Users\noman> docker volume create webdata
webdata
PS C:\Users\noman> |
```

Figure 19. Creation of a Docker named volume.

This delineation of data from application logic conforms to cloud best practices that advise against the close coupling of computation and storage resources [15]. In cloud environments, containers may be halted or rescheduled automatically as a result of scaling events, failure recovery, or rolling updates. In the absence of persistent storage systems, application data would be forfeited during such processes.

9.2 Hands-On Lab 3: Creating and Using a Docker Volume

A container was launched with the volume mounted to demonstrate data persistence across container restarts.

```
Terminal

PS C:\Users\noman> docker volume create webdata
webdata
PS C:\Users\noman> docker run -d -v webdata:/usr/share/nginx/html -p 8081:80 nginx
9ccb50b21ea3d0f48c9fe11e66e8b88380c5a522a132ffa46fa9afb2a01e0b60
PS C:\Users\noman> |
```

Figure 20. Running a container with a mounted Docker volume.

Content stored within the mounted directory persisted even after the container was stopped and recreated, confirming that the data lifecycle was independent of the container lifecycle.

Critical Reflection

This example illustrates how Docker volumes facilitate stateful workloads in containerized settings. This approach is further developed in cloud platforms by connecting containers with managed storage services like Azure Disks or persistent volumes in orchestration systems. The laboratory work thus emphasizes the theoretical and practical significance of persistent storage in cloud-native architectures. To demonstrate a stateful containerised service, a PostgreSQL database container was deployed.

```
PS C:\Users\noman> docker run -d --name postgres-db -e POSTGRES_PASSWORD=admin postgres:15
Unable to find image 'postgres:15' locally
15: Pulling from library/postgres
ff8482cd03b7: Pull complete
fa77b44f96d2: Pull complete
9fa019823549: Pull complete
ac28d232c7f3: Pull complete
4c7f2d7f4f80: Pull complete
ee662d4e76ff: Download complete
14c26c45f021: Download complete
Digest: sha256:29342cb52157b098821961d2c14eec3c019071f56a5d559e990cf07cf541ea9b
Status: Downloaded newer image for postgres:15
ee440a493748f674007eb3245555899e6fa26b8e58935e4fa28a70f7003610e3
PS C:\Users\noman> |
```

Figure 21. Deployment of PostgreSQL container as a stateful backend service.

10. Container Networking in Distributed Systems

Contemporary cloud apps seldom function as isolated, independent services. Rather, they consist of numerous interrelated elements, including web servers, databases, and backend APIs. Container networking facilitates secure and efficient communication among various components within a distributed environment. Distributed applications necessitate dependable communication among services. Docker facilitates this via user-defined networks that offer integrated service discovery. Figure 22 illustrates the communication between containers within a Docker network utilizing service names.

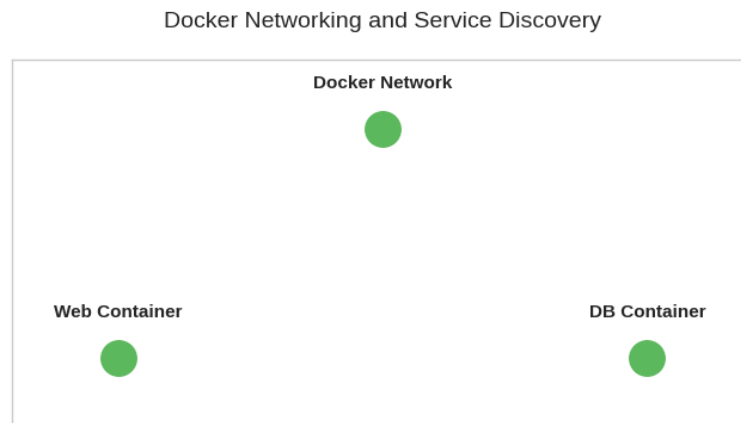


Figure 22. Docker user-defined bridge network enabling service discovery between containers.

Docker provides several networking models, including:

- Default bridge networks
- User-defined bridge networks
- Host networks
- Overlay networks (used with orchestration platforms)

Among these, **user-defined bridge networks** are particularly important for cloud-style application architectures because they provide built-in DNS-based service discovery and improved isolation.

10.1 Importance of Container Networking

Container networking simplifies the complexities of the underlying network, enabling developers to establish logical communication pathways between services. In conventional settings, application components frequently depend on static IP addresses or manually defined DNS entries. Docker networking removes this need by enabling containers to reference one another using service names. This functionality markedly enhances application portability and resilience, especially in cloud contexts where container instances can be dynamically produced or terminated [11].

11. Docker Networking and Service Communication

A user-defined Docker bridge network was created to enable isolated and secure inter-container communication.

```
Terminal
PS C:\Users\noman> docker network create app-network2
c0c07bfb64a3839af1f0d1726c8669e9b816057790094926521f8d358a844d6
PS C:\Users\noman> █
```

Figure 23. Creation of a user-defined Docker bridge network.

Using Docker inspection tools, it was verified that both containers were attached to the same virtual network and could resolve each other by container name.

Interpretation

This laboratory exercise illustrates how container networking facilitates service discovery, an essential component of microservices systems. In the realm of cloud computing, this reflects how virtual networks and DNS services offered by cloud platforms facilitate secure communication across services in remote environments.

12. End-to-End Docker Application Demonstration

To meet the assessment criteria for illustrating a comprehensive containerization process, the individual laboratory exercises were amalgamated into a singular end-to-end workflow. This presentation combines image generation, container execution, persistent storage, and networking into a unified application deployment.

12.1 Overview of the End-to-End Scenario

The selected example was a containerised web application designed to:

- Serve a static HTML page
- Persist content using a Docker volume
- Communicate within a defined Docker network
- Be distributable via Docker Hub

This example reflects a simplified version of real-world cloud web services and demonstrates how multiple container technologies work together in practice.

12.2 End-to-End Workflow Summary

1. Dockerfile creation to define the runtime environment
2. Image build using the Docker CLI
3. Image tagging and publishing to Docker Hub
4. Volume creation and attachment for data persistence
5. Network configuration for inter-container communication
6. Container execution and verification via browser access

12.3 Educational and Professional Significance

This end-to-end demonstration is representative of entry-level cloud engineering workflows in industry. It mirrors real practices such as:

- Immutable infrastructure
- Decoupling of compute and storage
- Network-based service discovery

- Repeatable and automated deployment procedures

By completing this demonstration, the laboratory exercises move beyond individual command usage and instead highlight the system-level thinking required for professional cloud engineering roles.

13. Video Demonstration and Assessment Alignment

As specified in the assessment brief, a short video (maximum 15 minutes) accompanies this report to demonstrate the practical elements of containerisation. The video provides visual confirmation of:

- Dockerfile creation
- Image building
- Container execution
- Volume persistence
- Network communication

Video Link:

<https://youtu.be/sZwxiucWB7o>

Including the video strengthens the practical credibility of the submission and supports the marks allocated for hands-on demonstration.

14. Load Balancing, Scalability, and Application Gateways in Cloud Computing

Cloud-based applications must be engineered to accommodate variable workloads while ensuring performance, availability, and security. Modern cloud designs depend on three interrelated concepts: load balancing, scalability, and application gateways. These principles were presented throughout the module lectures and constitute fundamental architectural components of production-grade cloud services.

14.1 Load Balancing

Load balancing is the method of allocating incoming network traffic among various backend computer resources to prevent any single resource from becoming a performance constraint. In cloud systems, load balancers serve as the principal access point for client requests and adeptly direct traffic to operational backend servers.

Load balancing is crucial for ensuring availability and performance in cloud-hosted applications. Cloud providers utilize load balancers to allocate traffic among backend services. Figure 24 illustrates a conceptual representation of traffic distribution via an Azure Load Balancer.

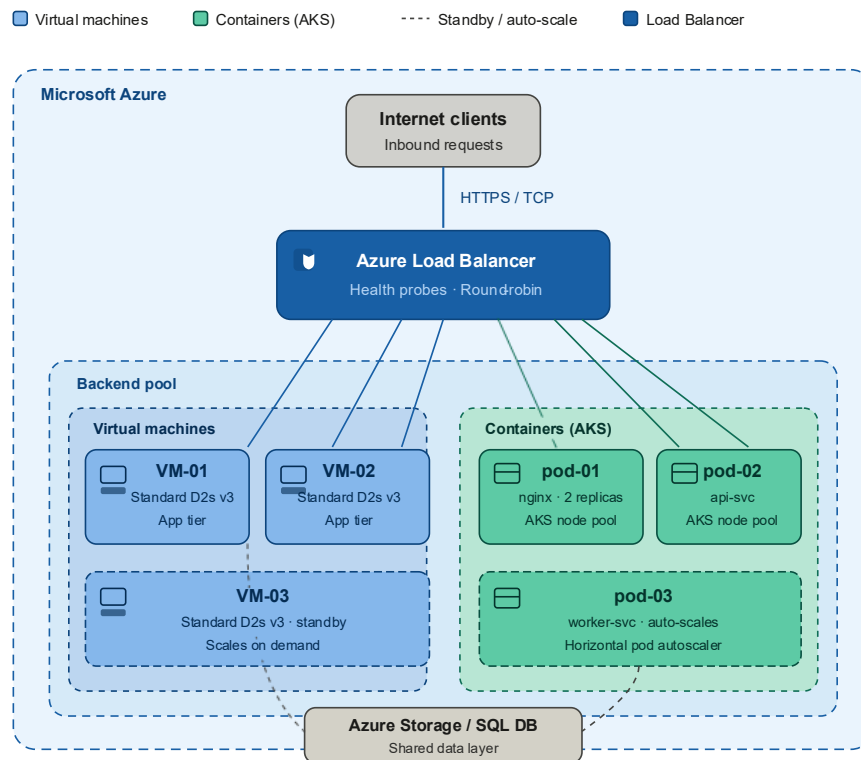


Figure 24. Azure load balancer distributing incoming traffic across multiple backend resources.

Microsoft Azure offers the Azure Load Balancer, functioning at Layer 4 of the OSI model, overseeing TCP and UDP traffic [7]. Azure Load Balancer allocates incoming traffic according to established criteria and persistently assesses backend instances using health probes. In the event that a backend resource becomes inaccessible, traffic is immediately rerouted to operational instances.

The key benefits of load balancing include:

- Improved application availability
- Fault tolerance and redundancy
- Even distribution of workload
- Reduced response times under load

Load balancing is very efficient in containerized environments due to the quick initiation and termination of containers. This enables backend pools to scale dynamically without service disruption, accommodating elastic cloud workloads [6].

14.2 Scalability in Cloud Systems

Scalability denotes a system's capacity to modify its resource allocation in reaction to fluctuations in workload. Cloud systems facilitate two principal types of scalability: vertical scaling, which enhances the resources of a current instance, and horizontal scaling, which involves the addition or removal of

instances.

Horizontal scaling is particularly pertinent in cloud-native and container-based architectures. Containers are lightweight, initiate rapidly, and need minimum resources, rendering them optimal for automated scaling systems [11].

Scalable cloud systems are able to:

- Handle sudden spikes in demand
- Reduce costs during low-usage periods
- Maintain consistent application performance
- Improve resilience through redundancy and replication

Scalability is typically executed using auto-scaling policies that track performance parameters, such as CPU use, memory usage, or request rates. When thresholds are surpassed, fresh instances are provided automatically [8].

14.3 Application Gateways

Application gateways function at Layer 7 of the OSI model, offering sophisticated traffic management features that exceed basic packet forwarding. In contrast to Layer 4 load balancers, application gateways analyze HTTP and HTTPS requests and can determine routing based on the content of the requests.

Typical functions of application gateways include:

- URL-based routing
- SSL/TLS termination
- Web application firewall (WAF) integration
- Enhanced security and traffic inspection

The Azure Application Gateway in Microsoft Azure allows organizations to secure and optimize online traffic while integrating effortlessly with backend services. Application gateways are frequently implemented in conjunction with load balancers and scaling solutions to create a holistic application delivery framework.

15. Why Load Balancing and Scalability Are Complementary Concepts

Load balancing and scalability are interdependent technologies that mutually enhance one another. Scalability enhances system capacity through the addition or removal of computing resources, whereas load balancing guarantees the equitable distribution of incoming traffic across these resources.

In the absence of load balancing, the extra capacity gained via scaling may be underutilized or create new bottlenecks. Conversely, in the absence of scalability, a load balancer possesses a finite array of backend resources and is unable to adequately address heightened demand.

Together, load balancing and scalability provide:

- Elastic response to changing workloads

- Improved fault tolerance and resilience
- Optimised performance under variable load

In container-based cloud systems, scaling generally involves initiating supplementary container instances, while the load balancer dynamically adjusts its backend pool to incorporate the newly established instances. This close connectivity facilitates uninterrupted scaling, a crucial necessity for contemporary cloud services.

16. Benefits of Combining Load Balancers, Scalability, and Application Gateways

The integration of load balancers, scaling mechanisms, and application gateways yields resilient, enterprise-grade cloud systems that give substantial advantages to both service providers and end users. Contemporary cloud-native systems integrate containerization, load balancing, and scalability to provide robust and adaptable services. These technologies collaborate to facilitate dynamic workloads. Figure 10 depicts an integrated cloud-native application architecture.

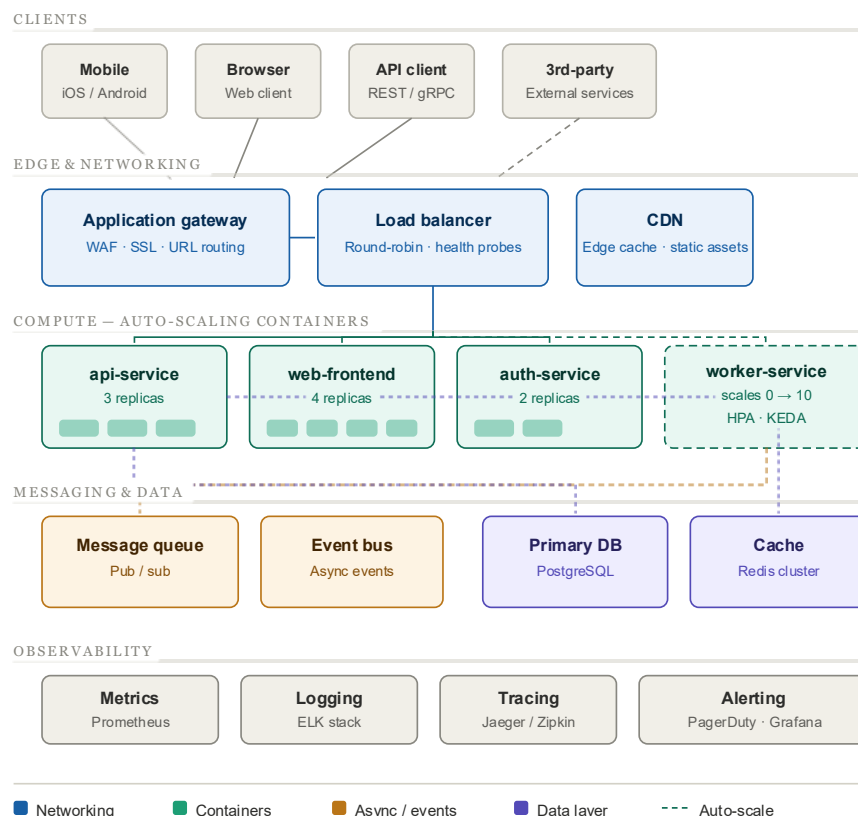


Figure 25. Integrated cloud-native application architecture combining containerisation, load balancing, and scalability.

16.1 High Availability and Resilience

By distributing traffic across multiple instances and automatically replacing failed components, combined architectures minimise downtime and service disruption.

16.2 Performance and User Experience

Traffic is routed efficiently to healthy backend resources, ensuring low latency and consistent response times even during peak usage periods.

16.3 Security and Traffic Control

Application gateways enhance security by inspecting traffic, enforcing access policies, and mitigating common web-based attacks through integrated firewall capabilities.

16.4 Real-World Cloud Usage

Most production cloud applications deploy all three technologies together. For example, a typical web application may use:

- An application gateway for secure HTTPS routing
- A load balancer for distributing traffic
- Auto-scaling to adjust capacity dynamically

This layered approach reflects current industry best practice in cloud engineering.

17. Main Driving Forces Behind the Expansion of Cloud Computing

The rapid expansion of cloud computing has been driven by several interrelated factors, with **cost efficiency** and **flexibility** being the most significant.

17.1 Cost Efficiency

Cloud systems utilize a consumption-based pricing model, wherein customers are charged just for the resources they utilize. This reallocates spending from capital investment in hardware to operational expenditure, thereby diminishing financial risk and enhancing cost predictability [1], [15]. Cloud providers leverage economies of scale, enabling them to supply computer resources at a cheaper cost than most individual organizations could get individually.

17.2 Flexibility and Agility

Cloud platforms facilitate the swift allocation and deallocation of resources. This allows organizations to experiment, rapidly deploy new services, and adapt to evolving business demands more swiftly than conventional IT architectures. Containerization further augments this versatility by facilitating uniform application deployment across many environments [2], [3].

18. Serverless Computing Versus Containerisation

A contemporary method for deploying cloud-based applications is serverless computing, commonly referred to as Functions as a Service (FaaS). Notable instances comprise AWS Lambda and Azure Functions.

18.1 Characteristics of Serverless Computing

Serverless systems completely eliminate the need for infrastructure administration. Developers submit code functions, while the cloud provider oversees execution, scaling, and availability.

Key characteristics include:

- Event-driven execution
- Automatic scaling
- Fine-grained billing based on execution time [9], [10]

18.2 Comparison with Containerisation

Aspect	Containerisation	Serverless
Control over runtime	High	Limited
Execution model	Long-running services	Event-driven
Scaling	Manual or auto-scaling	Automatic
Cost model	Resource-based	Execution-based

Containerization provides enhanced control and flexibility, rendering it appropriate for intricate or prolonged services. Serverless computing is more appropriate for lightweight, event-driven workloads characterized by unpredictable traffic patterns.

18.3 Relationship to Cloud Growth Drivers

Containerization and serverless computing tackle the primary motivators of cloud adoption—cost efficiency, scalability, and flexibility—albeit from distinct viewpoints. In practice, numerous contemporary cloud systems employ a hybrid of both methodologies to enhance performance and save costs.

19. Conclusion

This research has rigorously analyzed the ideas and methods of cloud engineering, emphasizing containerization, scalability, and contemporary cloud infrastructure architecture. The report commenced with the fundamental notion of virtualization, illustrating how containerization enhances shared hardware abstraction to provide lightweight, portable, and efficient application environments. Through a series of practical laboratory exercises, Docker demonstrated its efficacy as a robust tool for constructing images, managing containers, facilitating persistent storage, and enabling service communication. These practical demonstrations exemplified the application of theoretical cloud ideas in real-world applications.

The examination of load balancing, scalability, and application gateways underscored their synergistic functions in providing high availability cloud services. The comparison between containerization and serverless computing elucidated how various cloud execution models correspond with the overarching motivations for cloud adoption. This research illustrates that contemporary cloud engineering necessitates both theoretical comprehension and practical proficiency. Containerization, when integrated with scalable cloud services and intelligent traffic management, facilitates the deployment of resilient, cost-effective, and adaptable applications in the contemporary cloud-centric computing environment.

References

- [1] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, NIST Special Publication 800-145, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2011.
- [2] C. Pahl, “Containerization and the PaaS cloud,” *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May–Jun. 2015.
- [3] D. Merkel, “Docker: Lightweight Linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, Mar. 2014.
- [4] J. Turnbull, *The Docker Book: Containerization Is the New Virtualization*, 1st ed. Austin, TX, USA: James Turnbull Publishing, 2014.
- [5] Docker Inc., “Docker documentation,” 2024. [Online]. Available: <https://docs.docker.com>
- [6] Microsoft, “Azure architecture center – load balancing,” Microsoft Learn, 2024. [Online]. Available: <https://learn.microsoft.com/azure/architecture>
- [7] Microsoft, “Azure Load Balancer overview,” Microsoft Learn, 2024. [Online]. Available: <https://learn.microsoft.com/azure/load-balancer>
- [8] Amazon Web Services, *AWS Well-Architected Framework*, AWS Whitepapers, 2024. [Online]. Available: <https://aws.amazon.com/architecture/well-architected/>
- [9] Amazon Web Services, “AWS Lambda developer guide,” 2024. [Online]. Available: <https://docs.aws.amazon.com/lambda/>
- [10] Microsoft, “Azure Functions documentation,” Microsoft Learn, 2024. [Online]. Available: <https://learn.microsoft.com/azure/azure-functions/>
- [11] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes,” *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, May 2016.
- [12] R. Buyya, C. Vecchiola, and S. Selvi, *Mastering Cloud Computing: Foundations and Applications Programming*, New York, NY, USA: McGraw-Hill, 2013.
- [13] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices architecture enables DevOps: Migration to a cloud-native architecture,” *IEEE Software*, vol. 33, no. 3, pp. 42–52, May–Jun. 2016.
- [14] Google, *Site Reliability Engineering: How Google Runs Production Systems*, Sebastopol, CA, USA: O’Reilly Media, 2016.
- [15] Microsoft, “Cloud pricing and cost management,” Microsoft Learn, 2024. [Online]. Available: <https://learn.microsoft.com/azure/cost-management>