



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

**Cloud Engineering: Containerisation, Docker
and Scalable Cloud Architectures**

Name: Dipkumar Hemantkumar patel

Student Id: 2535265

Course: MSc Cybersecurity and Digital Forensics

Module: Cloud Engineering Principles and Practice

University: University of Wales, Trinity Saint David

Table of Contents

1. Introduction	3
2. Containerisation and Docker in the Cloud.....	3
2.1 Principles of Virtualisation.....	3
2.2 Principles and Benefits of Containerisation.....	5
2.3 Building a Docker Image (Dockerfile)	7
2.4 Images and Publishing to Docker Hub	8
2.5 Container Volumes	9
2.6 Container Networking.....	10
2.7 End-to-End Example: Web Server in a Container	12
3. Core Cloud Computing Concepts	13
3.1 Load Balancing	13
3.2 Scalability (Auto-scaling)	13
3.3 Application Gateways	15
3.4 Why Load Balancing and Scaling Are Complementary.....	16
3.5 Combined Benefits to the Customer.....	16
4. Driving Forces of Cloud Expansion and Serverless	17
4.1 Main Driving Forces	17
4.2 Serverless / Functions as a Service (FaaS)	18
4.3 Serverless vs Containers	18
5. Video Demonstration	20
6. Conclusion	20
7. References	21

1. Introduction

Cloud computing has revolutionized how contemporary software gets designed, packaged, deployed and consumed. Various enabling technologies and services support this transition. For example, hardware virtualisation, containerisation and managed cloud services contribute to resource efficiency, elasticity, traffic distribution and security. This report looks into these technologies and their relationship in today's cloud environments, and especially at Docker, which is a very popular containerisation platform [1][2].

The report adheres to evaluation requirements and extends discussion of the basic principles of virtualisation and containerisation to their practical implementation using Docker. Section 2 explores virtualisation and containerisation first, before then demonstrating how to construct a Dockerfile, publish images to Docker Hub, setting container volumes and networking and finally an end-to-end web-server deployment. The third section broadens the discussion from individual containerised applications to enterprise-level cloud environments by evaluating load balancing, scalability and application gateways using Microsoft Azure as the reference cloud provider. Section 4 looks at the broader drivers of cloud adoption, including cost, flexibility and elasticity, before analysing serverless functions (FaaS) and comparing them to container deployment. Section 5 presents the video demonstration, while Section 6 concludes the report.

The report uses numeric in-text references as per the assessment requirements, and the complete reference list is provided in Section 7.

2. Containerisation and Docker in the Cloud

According to research conducted, the containerisation technique is a pivotal part of numerous cloud-native programs since it is a very portable, reproducible and resource efficient way of packaging and deploying software [1][3]. When there is a requirement to develop, test and deploy the application consistently in different computing environments, its value becomes apparent. Nonetheless, containerisation does not just substitute for virtualisation; the two approaches provide different degrees of isolation, resource efficiency and operational flexibility.

This evolution sets up the technical relationship between the virtual machines and containers for the Docker examples.

2.1 Principles of Virtualisation

Virtualisation is a mechanism for abstracting physical computing resources, including CPU, memory, storage and networking, into multiple logical instances that can run systems and workloads independently while sharing the underlying hardware [4]. A hypervisor, for example VMware ESXi, Microsoft Hyper-V or KVM, manages the physical resources underneath and provides isolation between VMs.

Hypervisors are commonly classified into two types [4]:

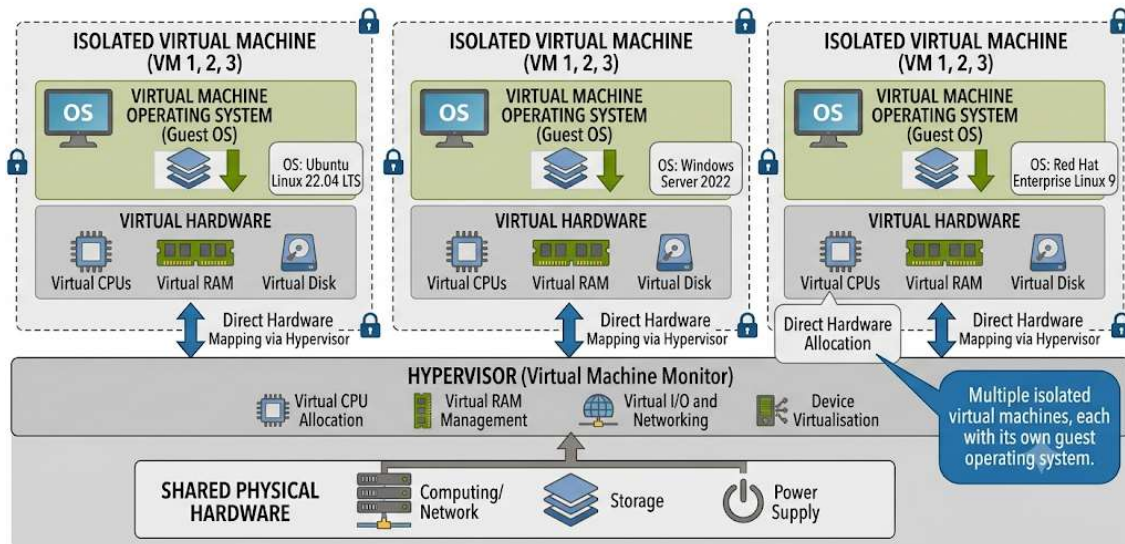
- **Type 1 (bare-metal) hypervisors** run directly on physical hardware and manage the allocation of hardware resources to virtual machines. Examples include VMware ESXi, Microsoft Hyper-V and Xen. They are widely used in server and cloud environments because they provide efficient resource management and strong workload isolation.
- **Type 2 (hosted) hypervisors** run as applications on a host operating system, such as VirtualBox and VMware Workstation. They are more commonly used for desktop, testing and development environments.

One enabler for several foundational cloud capabilities like multi-tenancy, elasticity and hardware consolidation is Virtualisation. [2][4] The capabilities of multi-tenancy and elasticity enable multiple isolated workloads to share the same physical infrastructure while allowing the provisioning and removal of VMs according to demand. Utilization can also be improved through hardware consolidation by allowing multiple workloads to run on physical servers instead of requiring each application to have dedicated hardware. As can be seen in Figure 1, the hypervisor abstracts the hardware and allows several guest operating systems to run.

Still, because of the guest operating system required with each VM, virtualisation comes with resource overhead. This increases resource requirements and can result in slower startup times and lower workload density than containers. Containerisation allows cloud-native applications that need fast provisioning, high workload density and frequent scaling to use resources more efficiently. Because VMs provide stronger isolation and can support legacy operating systems, they may be more suitable than containers for workloads with these requirements.

Consequently, virtualisation and containerisation should not be viewed as competing technologies in which one universally replaces the other. The appropriate approach depends on workload requirements, isolation needs, operating-system dependencies and operational objectives. This limitation of VM-based deployment provides the rationale for examining containerisation, which can achieve greater workload density by sharing the host operating-system kernel rather than deploying a complete guest operating system for each workload.

Figure 1. Virtualisation: a hypervisor exposes shared physical hardware as multiple isolated virtual machines, each with its own guest operating system.



2.2 Principles and Benefits of Containerisation

A container packages an application together with its libraries, runtime and configuration into a portable application environment that shares the host operating system kernel [1][3]. Unlike a VM, a container does not include its own guest OS; instead, it uses Linux kernel features such as namespaces for isolation and control groups for resource management [3]. This architecture reduces the overhead associated with running a separate guest operating system for each workload and makes containers particularly suitable for applications requiring rapid deployment and efficient resource utilisation.

The key benefits of containerisation in a cloud context are:

- **Portability** — A container image delivers a consistent application environment from development to CI and production platforms, with variation only due to underlying architecture and runtime configuration [1].
- **Lightweight footprint** — Containers need relatively fewer resources than VMs. Containers open and start faster because containers do not boot a guest operating system [3].
- **Density** — The overhead of containers is less than VMs which allows more application workloads to run on the same host as compared to traditional VMs. This can improve resource utilization and reduces costs [2].
- **Reproducibility** — Images can come from declarative Dockerfiles with properly controlled image versions and dependencies. In this way, they support consistent application environments [1].

- **Microservices alignment** — Containers can package services that are deployable independently of one another. As a result, one can update a component and scale it without redeploying the entire application [5].

Table 1. Comparison of virtual machines and containers.

Dimension	Virtual Machines	Containers
Isolation boundary	Hypervisor/VM boundary	Host OS kernel
Guest OS	Full OS per VM	Shares host kernel
Start-up	Typically slower	Typically faster
Image/footprint	Generally larger	Generally smaller
Host density	Generally lower	Generally higher
Best suited to	Strong isolation, legacy OS workloads	Microservices, CI/CD and cloud-native applications

Although containerisation has many benefits, such as portability, scalability and resource efficiency, it does not replace virtual machines. Because containers share the kernel with the host operating system, their isolation boundary is generally weaker than VMs. This makes them less suitable for workloads that require isolation or a different operating system. Further, managing a large number of containers often requires orchestration platforms like Kubernetes that add additional operational complexity. Notwithstanding these shortcomings, the characteristically lightweight nature of containerisation, coupled with its speed of deployment as well as ease of integration with DevOps practices, can outweigh these disadvantages, making it particularly relevant for many of today’s cloud-native applications and microservices. Thus, organisations should choose containers or VMs depending on the application needs, security, and operational goals instead of assuming one is better than the other.

These characteristics explain why Docker behaves this way. The subsequent sections demonstrate how containerisation principles are implemented through Dockerfile-based image construction, image publication to Docker Hub, persistent volumes and container networking.

2.3 Building a Docker Image (Dockerfile)

A Dockerfile is a declarative text file containing instructions that Docker uses to construct a container image [1]. Docker processes these instructions as a sequence of build steps and can use build caching to avoid unnecessarily repeating unchanged steps during subsequent builds. Listing 1 shows the Dockerfile used in the class exercise to package a static HTML page behind an NGINX web server using Alpine Linux.

```
1 FROM nginx:alpine
2
3 LABEL maintainer="Dipkumar Patel"
4
5 COPY ./site/ /usr/share/nginx/html/
6
7 EXPOSE 80
8
9 CMD ["nginx", "-g", "daemon off;"]
```

Listing 1. A minimal Dockerfile that serves a static HTML site using NGINX on Alpine Linux.

```
PS C:\Users\dp454\Cloud Engineering Assignment> docker build -t cloud-assignment:1.0 .
[+] Building 4.5s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 188B
=> [internal] load metadata for docker.io/library/nginx:alpine
=> [auth] library/nginx:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 220B
=> [1/2] FROM docker.io/library/nginx:alpine@sha256:4a73073bd557c65b759505da037898b61f1be6cbcc3c2c3aeac22d2a470c1752
=> => resolve docker.io/library/nginx:alpine@sha256:4a73073bd557c65b759505da037898b61f1be6cbcc3c2c3aeac22d2a470c1752
=> => sha256:d0008c891db48b5f526d914bce9e8d889fe1a9d1f08291ae03fe97f871726f38 1.21kB / 1.21kB
=> => sha256:390dc935348d8070e695fbaae2a4bb114fb9e69c59f628e7576036ee9d5244c9 1.40kB / 1.40kB
=> => sha256:46519e7231d2eb5604df229beb44d59719a489eaa7aca52982535a010b07a9ed 20.31MB / 20.31MB
=> => sha256:46f977ee452f4399c208714afa034868d6056864f8a0cf3c643ab143dd802c80 404B / 404B
=> => sha256:62bec68d7c31c4c8a19d812d84da5f7748e54690c037979945b6c5b6c924b142 957B / 957B
=> => sha256:1223f016b4e4a2c21f7c49d4837fbfd47a9da6436b511690ca1e582fc2810d59 627B / 627B
=> => sha256:3cd534fe98c64d68a1f4f1c83abb8d5cba7ecfd7be88e592389929d12e6253da 1.89MB / 1.89MB
=> => sha256:55afa1ecc21d2bb5e5045f32dafee56272ffdb89860bac26f6c32123439af26a4 3.85MB / 3.85MB
=> => extracting sha256:55afa1ecc21d2bb5e5045f32dafee56272ffdb89860bac26f6c32123439af26a4 0.4s
=> => extracting sha256:3cd534fe98c64d68a1f4f1c83abb8d5cba7ecfd7be88e592389929d12e6253da 0.2s
=> => extracting sha256:55afa1ecc21d2bb5e5045f32dafee56272ffdb89860bac26f6c32123439af26a4 0.4s
=> => extracting sha256:3cd534fe98c64d68a1f4f1c83abb8d5cba7ecfd7be88e592389929d12e6253da 0.2s
=> => extracting sha256:55afa1ecc21d2bb5e5045f32dafee56272ffdb89860bac26f6c32123439af26a4 0.4s
=> => extracting sha256:3cd534fe98c64d68a1f4f1c83abb8d5cba7ecfd7be88e592389929d12e6253da 0.2s
=> => extracting sha256:62bec68d7c31c4c8a19d812d84da5f7748e54690c037979945b6c5b6c924b142 0.0s
=> => extracting sha256:46f977ee452f4399c208714afa034868d6056864f8a0cf3c643ab143dd802c80 0.0s
=> => extracting sha256:d0008c891db48b5f526d914bce9e8d889fe1a9d1f08291ae03fe97f871726f38 0.0s
=> => extracting sha256:390dc935348d8070e695fbaae2a4bb114fb9e69c59f628e7576036ee9d5244c9 0.0s
=> => extracting sha256:46519e7231d2eb5604df229beb44d59719a489eaa7aca52982535a010b07a9ed 0.5s
=> [2/2] COPY ./site/ /usr/share/nginx/html/
=> => exporting image 0.3s
=> => exporting layers 0.1s
=> => exporting manifest sha256:7433ff56a5f0ddb11d6a1f3d11be07453ed49e5d8f634a1abc89e60dc68aa8b7 0.0s
=> => exporting config sha256:3db42b597051ee888c49aa41f7f44f6eef60f99537ed4e3f7be0a6a1af943c14 0.0s
=> => exporting attestation manifest sha256:cad9d2c2ed387379d25032751b2963c287f977161a2852620fee3ff7bcf93b28 0.0s
=> => exporting manifest list sha256:a78d31b7468568bbdc2df6949d23973f9949214435e9279ff55cb96e3e24241 0.0s
=> => naming to docker.io/library/cloud-assignment:1.0 0.0s
=> => unpacking to docker.io/library/cloud-assignment:1.0 0.0s
```

```
View build details: docker_desktop://dashboard/build/desktop-linux/desktop-linux/2skxa4zs4q76w0s4c02v96vrg
PS C:\Users\dp454\Cloud Engineering Assignment> docker run -d --name web -p 8080:80 cloud-assignment:1.0
```

The image is then built and run locally using the commands shown below. The `-p 8080:80` option maps port 80 inside the container to port 8080 on the host, allowing the web application to be accessed through `localhost:8080`. This demonstrates the separation between the application running inside the container and the host port used to access the network service [2].

Since the application and the runtime components it requires are all included within the image, the same image and run configuration can usually reproduce the web server on compatible Docker hosts. This shows the portability in section 2.2 and provides an instance of how container images can reduce inconsistencies during deployment due to environment issues.

The Dockerfile also offers a reliable and repeatable approach to defining how the application image will be created. Representing the application environment through version-controlled configuration supports don't hard-code secrets reliable deployment and is also compliant with Infrastructure as Code practices. However, Dockerfiles need careful design so that the images do not add unnecessary weight, packages and vulnerabilities to the software. When images are large, storage and transfer requirements will increase. Similarly, when unnecessary packages are included, the attack surface increases. Known vulnerabilities may also come from outdated dependencies. As a result, Dockerfiles should always use a lightweight base image, minimize unnecessary components, avoid hard-coding secrets, and keep dependencies up to date regularly [1]. The application of these principles will enhance the safety and sustainability of the containerized application development processes.

2.4 Images and Publishing to Docker Hub

Docker Hub is the most commonly used container image registry. It provides both private and public repositories for storing and distributing the Docker images [1]. When an image has been created locally, it can be tagged for a target repository, followed by authentication and a push to Docker Hub. As evident from the screenshot provided, the image layers were successfully uploaded and the push yielded a SHA-256 digest in a practical demonstration.

```
PS C:\Users\dp454\Cloud Engineering Assignment> docker login
Authenticating with existing credentials... [Username: dippatel1955]
```

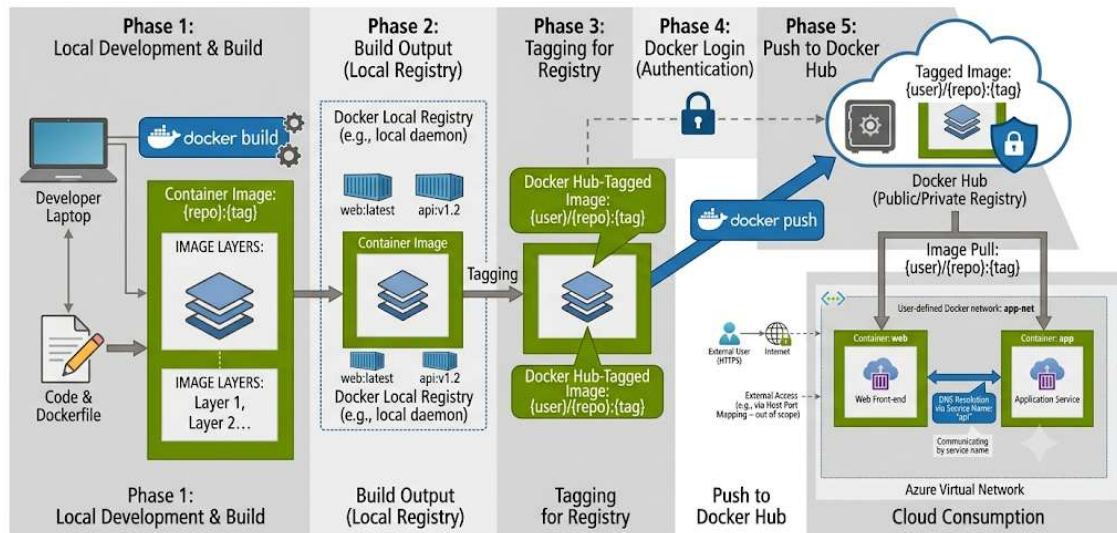
 Info → To login with a different account, run 'docker logout' followed by 'docker login'

```
Login Succeeded
PS C:\Users\dp454\Cloud Engineering Assignment> docker tag cloud-assignment:1.0 dippatel1955/cloud-assignment:1.0
PS C:\Users\dp454\Cloud Engineering Assignment> docker push dippatel1955/cloud-assignment:1.0
The push refers to repository [docker.io/dippatel1955/cloud-assignment]
62bec68d7c31: Pushed
55afa1ecc21d: Pushed
390dc935348d: Pushed
1223f016b4e4: Pushed
46f977ee452f: Pushed
3cd534fe98c6: Pushed
46519e7231d2: Pushed
52c7d63d3eed: Pushed
d0008c891db4: Pushed
6d0ba1db7ad3: Pushed
1.0: digest: sha256:a78d31b7468568bbdcb2df6949d23973f9949214435e9279ff55cb96e3e24241 size: 856
```

Publishing the image takes the Dockerfile and image-building process described in Section 2.3 from local development to image distribution. When an image is stored in a registry, compatible Docker hosts, cloud services, and container platforms, such as Azure Container Instances, Amazon ECS and Kubernetes, can retrieve it. This helps in ensuring a uniform deployment workflow where identical application images can be promoted through different

environments: development, testing and production [5].

Figure 2. Workflow for building locally with docker build, tagging, and pushing to Docker Hub for cloud consumption



However, Docker tags should not be treated as immutable identifiers. In the traditional sense. A tag like 1.0 gives a nice human-readable reference and can be re-assigned, on the other hand image digest is an identifier of the image content itself. Accordingly, the use of controlled image variants, as well as digest-based references where appropriate, can enhance deployment reproducibility and reduce the risk of unintentional execution of differing image contents across different environments.

Docker Hub also introduces operational issues and security problems. In cases of incorrect image publishing, application components may be exposed due to public repositories. Compromised user credentials can allow unauthorized image modification. Images should be stored in appropriate public or private repositories, access passwords should be secured, and images should be subject to periodic scanning and update to clean vulnerable dependencies. Consequently, Docker Hub is an effective tool for container image distribution, but reliable cloud deployment requires correct implementation of version management, access management and image security practices.

2.5 Container Volumes

When you remove a container, it's designed to be ephemeral, meaning that its writable layer gets destroyed. Any data stored only in that writable layer gets lost[1]. Docker volumes, managed outside the container's writable layer, are the ideal persistent storage mechanism for stateful workloads like databases or content management systems.

A named volume is created and used as follows:

```

PS C:\Users\dp454\Cloud Engineering Assignment> docker volume create cloud-data
cloud-data
PS C:\Users\dp454\Cloud Engineering Assignment> docker volume ls
DRIVER      VOLUME NAME
local       cloud-data
PS C:\Users\dp454\Cloud Engineering Assignment> docker run -d --name web-volume -p 80
81:80 -v cloud-data:/usr/share/nginx/html cloud-assignment:1.0
878c2d3d877b3f5d4de65f6c20d2a5d1ee62f981b356dbb0fad38225e7d3e02c

```

In the practical exercise, we created a named volume called cloud-data using docker volume create and mounted it to the NGINX web directory using -v cloud-data:/usr/share/nginx/html. The image shows the creation and validation of the volume before it is mounted to the web container. This prevents the stored data from becoming ephemeral like the lifecycle of the container itself. Thus, when a container is removed and replaced, data can still be available.

As a result, volumes exhibit persistence as their main advantage. An alternative container can mount the volume with the same name, enabling application upgrades and container replacement without the need for persistent data to be stored in the container itself. Persistent information can be shared among appropriate containers and can therefore form part of a broader backup and recovery strategy.

However, resilience imposes new operational requirements. Volumes still need proper backup procedures, access permissions, capacity management and performance considerations. In most cloud deployments, persistent container storage is backed by managed storage services such as Azure Managed Disks or Amazon Elastic Block Store (EBS), depending on the architecture and availability requirements [1]. Therefore, volumes solve the problem of container-lifecycle persistence but do not by themselves guarantee backup, high availability or disaster recovery.

2.6 Container Networking

Docker provides several default network modes, including bridge, host and none, and supports user-defined networks for more controlled container communication [1]. A user-defined bridge network provides DNS-based service discovery, allowing containers connected to the same network to communicate using service names rather than hard-coded IP addresses. This is particularly useful for multi-container and microservice applications.

Terminal

> docker ps

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
64ad5651d970	nginx:alpine	"sh -c 'sed -i 's/li...' "	6 minutes ago	Up 6 minutes	80/tcp	api
2057ad0c7e3c	cloud-assignment:1.0	"/docker-entrypoint..."	12 days ago	Up About a minute	0.0.0.0:8082->80/tcp, [::]:8082->80/tcp	web-network

```

PS C:\Users\dp454\Cloud Engineering Assignment> docker run --rm --network cloud-network curlimages/curl http://api:3000
% Total    % Received % Xferd  Average Speed   Time    Time     Time
           Dload  Upload   Total     Spent    Left     Speed
100    896  100    896    0     0  217.9k      0     0      0
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, nginx is successfully installed and working.
Further configuration is required for the web server, reverse proxy,
API gateway, load balancer, content cache, or other features.</p>

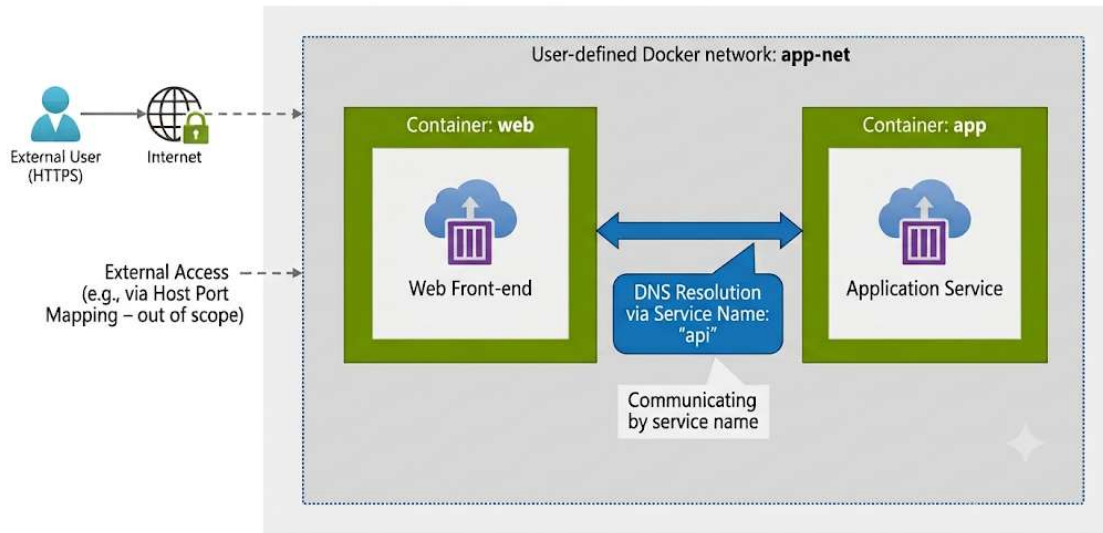
```

```
Terminal
API gateway, load balancer, content cache, or other features.</p>
<p>For online documentation and support please refer to
<a href="https://nginx.org/">nginx.org</a>.<br/>
To engage with the community please visit
<a href="https://community.nginx.org/">community.nginx.org</a>.<br/>
For enterprise grade support, professional services, additional
security features and capabilities please refer to
<a href="https://f5.com/nginx">f5.com/nginx</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
100 896 100 896 0 0 208.7k 0 0
PS C:\Users\dp454\Cloud Engineering Assignment> docker network inspect cloud-network
[
  {
    "Name": "cloud-network",
    "Id": "475f64041a752a2c1082173ad40c80ea7c1282e96e3786c7d86f8cb254fea072",
    "Created": "2026-08-03T14:21:07.143977399Z",
    "Scope": "local",
    "Driver": "bridge",
    "DriverInfo": {
      "Driver": "bridge",
      "EnableIPv4": true,
      "EnableIPv6": false,
      "IPAM": {
        "Driver": "default",
        "Options": {},
        "Config": [
          {
            "Subnet": "172.21.0.0/16",
            "Gateway": "172.21.0.1"
          }
        ]
      },
      "Internal": false,
      "Attachable": false,
      "Ingress": false,
      "ConfigFrom": {
        "Network": ""
      },
      "ConfigOnly": false,
      "Options": {
        "com.docker.network.enable_ipv4": "true",
        "com.docker.network.enable_ipv6": "false"
      },
      "Labels": {},
      "Containers": {
        "2057ad0c7e3c53f92f95a6b8d07f6d2ff653e0051c17915763addf23fc38c141": {
          "Subnets": {
            "172.21.0.0/16": {
              "IPsInUse": 5,
              "DynamicIPsAvailable": 65531
            }
          }
        }
      }
    }
  ]
}
PS C:\Users\dp454\Cloud Engineering Assignment>
```

In the practical exercise, the web-network and api containers were connected to the user-defined cloud-network. The network configuration confirmed that both containers were attached to the same Docker bridge network. Communication was then verified using the service name api rather than its dynamically assigned IP address. The request to `http://api:3000` successfully returned the NGINX response, demonstrating DNS-based service

discovery and network communication between containers.

Figure 3. Two containers (web and api) attached to a user-defined bridge network `app-net`, communicating by service name



The conceptual example in Figure 3 illustrates how a web service can communicate with an API service using a service name rather than a hard-coded IP address. This method is particularly helpful in microservice architectures because the IP addresses of containers can change when the containers are recreated, while communication via service names is more stable. The same principle underpins technologies at a higher level, particularly Docker Compose and orchestration platforms [5].

Increased networking flexibility brings certain operational and security challenges. Increased numbers of containers and services lead to complex network configuration whereas, unnecessary connectivity or exposed ports increase the attack surface of the application. As a result, appropriate network segmentation should be implemented, and only the services and ports necessary for legitimate traffic should be exposed in production deployments.

2.7 End-to-End Example: Web Server in a Container

Combining the previous subsections, the class exercise produced a complete end-to-end workflow. A static HTML page was placed in the `./site/` directory, packaged using the Dockerfile in Listing 1, and built into the image `cloud-assignment:1.0`. The image was then tagged for the Docker Hub repository `dippatel1955/cloud-assignment:1.0` and pushed to the registry. The published image was subsequently run with host port 8080 mapped to container port 80, as shown in Figure 6.

```
PS C:\Users\dp454\Cloud Engineering Assignment> docker run -d -p 8080:80 dippatel1955
/ccloud-assignment:1.0
a41e1328895a1ee9aa1174860e31cdad995ba50744854f07d1f47fb88d862407
PS C:\Users\dp454\Cloud Engineering Assignment>
```

Figure 6. Running the published Docker Hub image with host port 8080 mapped to container port 80.

The command demonstrates how the published container image can be retrieved and executed using a consistent image reference. This end-to-end workflow integrates the principles examined in Sections 2.3–2.6, including declarative image construction, container execution, port mapping, image distribution and networking. It therefore demonstrates how containerisation can provide a repeatable approach to application deployment.

However, Docker alone does not provide all the capabilities required for large-scale production environments. As application demand increases, orchestration platforms such as Kubernetes can provide automated scheduling, scaling, service discovery and recovery. Therefore, this practical exercise provides a foundation for understanding how individual containerised workloads can be incorporated into larger cloud-native architectures. The accompanying video in Section 5 demonstrates the practical workflow.

While the example demonstrates how Docker can package and deploy an application consistently, production services exposed to variable demand require additional mechanisms for distributing traffic, increasing capacity and controlling application-layer access. These requirements are examined in Section 3.

3. Core Cloud Computing Concepts

Modern cloud applications must handle fluctuating numbers of users while maintaining low latency and high availability. Load balancing, scalability and application gateways provide complementary capabilities for distributing traffic, adjusting application capacity and controlling application-layer requests. The analysis employs Microsoft Azure as a reference cloud provider in order to demonstrate these concepts on a public e-commerce platform comprised of multiple containerized microservices [5][6].

3.1 Load Balancing

A load balancer distributes network traffic across multiple backend instances, preventing excessive demand from being concentrated on a single instance. It improves availability and resource utilisation [6]. Azure offers a variety of load balancers for handling different types of traffic. The Azure Load Balancer is a Layer 4 load balancer that distributes TCP and UDP traffic while the Azure Application Gateway provides regional Layer-7 HTTP/S load balancing and Azure Front Door provides global Layer-7 application delivery and traffic management [6][7].

In the e-commerce example, Azure Load Balancer sits in front of a backend pool containing multiple application instances. When a customer's request arrives, the service distributes connections to healthy backend instances according to its configured distribution mode. Azure Load Balancer uses five-tuple hash-based distribution by default, while source-IP-based session persistence can be configured when application requirements justify it. Health probes monitor backend instances, and new traffic is directed only to instances considered healthy.

This provides several customer-facing benefits. By partitioning connections across many instances, bottlenecks can be alleviated and response times can be improved. Also, health-

based routing will increase availability by preventing new connections from being sent to unhealthy backends. Load balancing also provides the foundation for horizontal scaling because additional healthy instances can be added to the backend pool as application capacity increases [6][7].

However, load balancing does not create additional processing capacity by itself. If the backend pool is already operating at capacity, distributing traffic across those instances cannot resolve the underlying resource constraint. Distribution can also become uneven when session persistence is used or when traffic patterns provide limited variation for hashing. Therefore, load balancing should be combined with appropriate health-probe configuration, monitoring and auto-scaling rather than treated as a standalone solution for high-demand applications [7].

3.2 Scalability and Auto-scaling

Scalability is the ability of a system to handle increased load by adding resources. The cloud supports two main approaches [2]:

- **Vertical scaling (scaling up)** — increasing the resources of a single instance, such as CPU and RAM. This approach is relatively simple but is constrained by the maximum capacity of the instance and may require a restart.
- **Horizontal scaling (scaling out)** — adding more instances behind a load balancer. It can provide substantially greater capacity and fault tolerance than scaling a single instance and is widely used in cloud-native architectures [5].

Azure implements horizontal scaling primarily through Virtual Machine Scale Sets (VMSS) and Azure Kubernetes Service (AKS) cluster autoscalers [7]. An auto-scaling rule is defined in terms of a performance metric, such as an average CPU-utilisation threshold of 70%, and an associated scaling action, such as adding two instances up to a configured maximum. After certain metrics fall below a specified lower limit, instances can be deleted, which will use fewer resources and may lower costs. Scaling rules can be based on a schedule. For instance, additional instances can be provisioned before periods of anticipated demand [7].

In the e-commerce example, the marketing campaign produces a tenfold spike in traffic. The auto-scaling function reacts by bringing additional instances of the storefront online and the load balancer will distribute incoming requests across these extra healthy instances. Reducing the number of instances once the demand reduces will help save resources and operating cost [2][7].

This demonstrates the elasticity advantage of cloud infrastructure compared with fixed-capacity deployments.

Auto-scaling can largely help in enhancing operational efficiency and cost-effectiveness. Making scaling decisions based on the wrong performance metrics can lead to over-provisioning which adds to the operating cost and vice versa. If there is under-provisioning, this can cause performance degradation or a complete failure of services experiencing an

increase in traffic. In addition, rapidly scaling distributed applications requires careful management of application state, database performance and network capacity. Therefore, successful cloud architectures combine auto-scaling with effective monitoring, load balancing and performance optimisation to achieve both cost efficiency and service reliability.

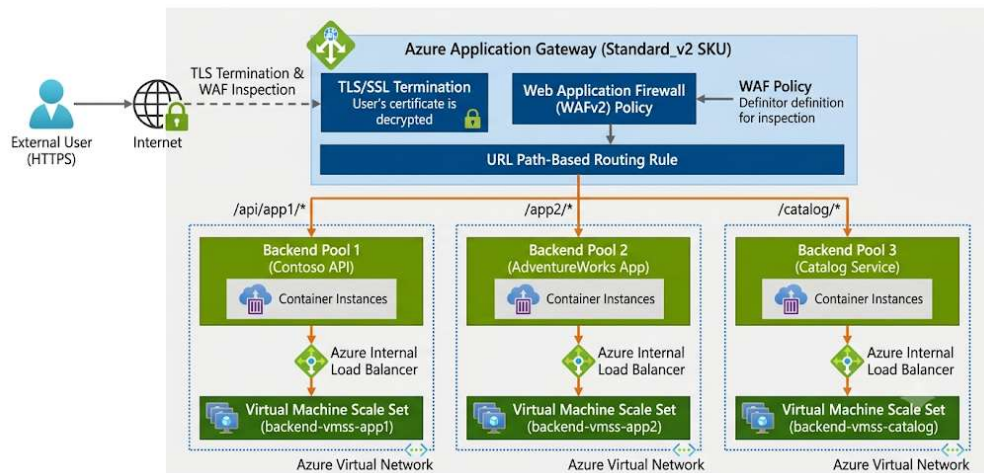
3.3 Application Gateways

An application gateway is a Layer-7 reverse proxy that provides intelligent traffic management on top of basic load balancing [7]. Azure Application Gateway provides URL-based and host-based routing, SSL/TLS termination, cookie-based session affinity, an integrated Web Application Firewall (WAF) for protection against common web application attacks, and end-to-end TLS to backend services [7]. Because it operates at the HTTP layer, it can inspect request information such as headers, paths and hostnames and route traffic according to application-level rules.

In the e-commerce example, requests to `/api/*` can be routed to a backend pool of API containers, requests to `/images/*` to a static-content pool, and requests to the root path to the storefront pool. TLS can be terminated at the gateway, reducing the need for individual backend containers to manage external certificates. The WAF can detect and block malicious HTTP requests, including common SQL-injection and cross-site-scripting attacks, before they reach the application [7]. Cookie-based session affinity can also maintain a user's session on the same backend instance where this is required by the application's design.

Although application gateways can significantly improve security and traffic management, they also introduce additional cost and architectural complexity. Incorrect routing rules, poorly configured security policies or SSL/TLS certificates can negatively affect application availability and user experience. Furthermore, inspecting Layer-7 traffic introduces additional processing compared with basic Layer-4 load balancing. Therefore, application gateways can be an important component of cloud-native architectures where Layer-7 routing and application-level security are required, but their deployment should be justified by the application's traffic-management and security requirements.

Figure 4. Azure Application Gateway terminating TLS, applying WAF rules and routing by URL path to multiple containerised backend pools, each fronted by an Azure Load Balancer and backed by a VM Scale Set



3.4 Why Load Balancing and Scaling Are Complementary

Load balancing and scaling address complementary aspects of the same capacity problem and are most effective when used together [2][6][7]. Scaling without load balancing means that additional instances may be provisioned without receiving an appropriate share of incoming traffic, leaving existing instances as potential bottlenecks. Conversely, load balancing without scaling distributes traffic across a fixed backend pool, but a sufficiently large demand surge can still overwhelm the available capacity.

Combined, they form a coordinated process: the load balancer distributes traffic and uses health information to determine suitable backends, while the autoscaler monitors configured utilisation or performance metrics and adjusts the size of the backend pool. Traffic through the load-balancing layer can be received by healthy instances newly provisioned. This interaction creates a basis for elastic and fault-tolerant cloud services.

Cloud elasticity is illustrated by the complementary relationship between load balancing and auto-scaling. Organisations can optimise application performance by reacting to workload demands and distributing requests among resources to avoid any unnecessary capacity of infrastructure. Nonetheless, effective monitoring, sensible scaling thresholds, and well-designed policy interventions are needed. Poorly configured scaling thresholds can lead to failed scaling actions, overprovisioning or unnecessary repeated scaling. All these can negatively impact the reliability and cost efficiency of an application. Therefore, load balancing and auto-scaling should be designed as complementary components rather than as separate services.

3.5 Combined Benefits of Load Balancing, Scaling and Application Gateways

The combination of load balancing, auto-scaling and application gateways enhances customer experience and operational management [6][7]:

- **Performance** — The application gateway uses application-level rules to route requests. The load balancer redirects traffic onto healthy backend instances. And the auto-scaling can add capacity according to demand. When grouped together, these services help maintain application responsiveness during traffic spikes.
- **Availability** — health probes and multiple backend instances reduce the risk of a single instance failure causing service disruption. Unhealthy backends can be excluded from traffic distribution, improving service continuity.
- **Security** — A Web Application Firewall (WAF), TLS termination and private backend pools can centralise certificate and application-layer security while limiting attacks on the external facing surfaces.
- **Cost efficiency** — Auto-scaling is the process that would adjust the resource capacity according to particular workload requirements, which has the potential to reduce the cost of preserving peak capacity that remains unused [2].

- **Operational agility** — The routing and deployment configurations can be modified to incorporate additional microservice back-ends to enhance the application capacity and traffic-management policies depending on the business needs.

As a unified architecture rather than as stand-alone services, these technologies illustrate how modern cloud platforms can deliver scale, security, resilience and operational efficiency. Nonetheless, these benefits are dependent on correct configuration and monitoring and coordination of the components. If scaling thresholds are not good or use incorrect routing rules or if security policies are misconfigured, then it can reduce performance, increase outlay or affect availability. The main benefit is not merely three managed services but a coordinated design around the application's workload, availability and security requirements.

4. Driving Forces of Cloud Expansion and Serverless

4.1 Main Driving Forces of Cloud Adoption

The expansion of cloud computing over the past decade has been driven by a combination of economic, operational and strategic factors [2][8]:

- **Cost model** — Cloud computing can help shift organisations away from large upfront capital expenditure (CapEx) on data centres and hardware. Instead, cloud expenditure... When capacity and costs are more closely aligned with demand, it is less likely that poorly controlled consumption will cause unexpectedly large costs [2].
- **Elasticity and flexibility** — Cloud capacity can often be provisioned or removed rapidly. This makes it possible for businesses to respond to demand spikes, seasonal peaks and product launches without the procurement lead times [2][7].
- **Global reach** — Major cloud service providers have regions and availability zones across the geography where applications can be deployed in proximity to end-users to reduce latency and support data-residency requirements [7].
- **Time-to-market** — Cloud platforms provide managed infrastructure, development services and deployment tools that can reduce the time required to develop, test and release new applications [8].
- **Reliability and resilience** — Redundant storage and backup services, availability zones, and other features can offer levels of resilience that may be very costly and operationally complex to replicate with wholly on-premises infrastructure [2].
- **Security and compliance** — Major cloud providers offer extensive physical-security controls, encryption capabilities, identity services and compliance programmes, including support for standards and regulatory requirements such as ISO 27001, SOC 2, GDPR and PCI-DSS [8]. However, organisations still have responsibility for configuring and managing their workloads.
- **Innovation velocity** — Providing serverless computing, AI/ML platforms, edge computing and other services, cloud platforms are making it easy for organisations to

adopt new technologies without having to develop all supporting infrastructure themselves [8].

Even though all these factors are accelerating the pace of cloud adoption across industries, organisations should also be aware of hurdles like vendor lock-in, data sovereignty, regulatory compliance, and unpredictable operational costs stemming from usage-based pricing. Successful cloud adoption therefore calls for effective governance, costing and workload assessment to establish whether expected business benefits outweigh the associated risks and costs. Cloud computing should not be regarded as superior to on-premises infrastructure. Instead, enterprise-grade cloud solutions can deliver flexibility, scalability, and managed capabilities that suit an organisation's technical and business requirements.

4.2 Serverless Functions (Functions as a Service)

Serverless or Functions as a Service (FaaS) is a cloud execution model whereby the provider executes short-lived functions in reaction to an event and allocates computing resources on-demand while employing usage-based pricing, where pricing is usually based on parameters like the count of invocations, the duration of execution, and allocated resources [9]. AWS Lambda, Azure Functions and Google Cloud Functions are common examples of FaaS platforms [9][10].

In a serverless deployment, the developer writes the function logic and declares its trigger, such as an HTTP request, queue message, file upload or scheduled event. The cloud platform manages much of the underlying provisioning, patching, scaling and infrastructure operation. When no function invocations occur, the function compute capacity can scale to zero, avoiding execution charges for that function, although associated services may still incur costs [9].

Serverless is particularly suitable for event-driven and intermittent workloads, such as image-thumbnail generation following a file upload, webhook handlers, scheduled report jobs, lightweight APIs and integration logic connecting managed cloud services. It is less suitable for long-running computations, latency-sensitive workloads where cold-start delays are unacceptable, or applications requiring specialised runtimes or persistent in-process state [9][10].

While serverless computing can significantly reduce the infrastructure provisioning and management burden, it is not suitable for every application. Cold-start latency, execution limits and limited runtime customisation can affect performance for long-running or latency-sensitive workloads. Overdependence on provider-specific services can lead to vendor lock-in, creating hurdles for migration between cloud platforms. Therefore, organisations should assess workload duration, invocation patterns, latency requirements, runtime requirements and portability before adopting serverless computing.

4.3 Serverless vs Containers: Compare and Contrast

Both containers and serverless functions are responses to the same driving forces (cost, agility, elasticity), but they make different trade-offs.

Table 2. Containers compared with serverless functions

Dimension	Containers (e.g., Docker on AKS/ECS)	Serverless (e.g., Lambda, Azure Functions)
Unit of deployment	Container image	Function/code package with an event trigger
Scaling	Horizontal scaling through an orchestrator or managed platform; speed depends on configuration and infrastructure	Automatic event/request-driven scaling, with scale-to-zero supported by some platforms
Billing	Usage-based container/compute resources; depends on hosting platform	Typically based on invocations and execution duration, with additional service charges possible
Cold start	Usually no per-request cold start once running, although new containers require startup time	Possible when a new execution environment must be initialised; impact varies by runtime and platform
Runtime control	High — supports custom images, libraries and binaries	More limited — constrained by the supported runtime and platform environment
Execution duration	Suitable for long-running services; limits depend on the hosting platform	Platform-specific execution limits; for example, AWS Lambda has a maximum invocation duration
Portability	High — container images can generally run across compatible container environments	Lower — applications may depend on provider-specific triggers, APIs and services
Operational responsibility	Greater customer control and responsibility for images, runtime and deployment configuration	Provider manages underlying execution infrastructure; customer remains responsible for code, configuration, permissions and application security
Best suited to	Long-running microservices, custom runtimes and applications requiring greater runtime control	Event-driven, intermittent or unpredictable workloads where operational simplicity is important

The comparison shows that neither one technology is inherently better; rather, both respond to different operational needs. Serverless computing is most useful for workloads that are event-driven and intermittent where operational simplicity and scaling elasticity are priorities. Containers generally provide greater portability, runtime flexibility and operational control than serverless functions. As a result, they are better suited for long-running services, microservices, and workloads that need customized runtimes. As a result, modern cloud architectures can utilise serverless functions for lightweight event processing and containerised services for core business applications. This enables organisations to choose the deployment model based on workload characteristics rather than labelling either technology as superior.

5. Video Demonstration

The practical Docker workflow described in Section 2 is illustrated in a short video that lasts 15 minutes. It will demonstrate building an image from a Dockerfile, running the container, creating and using a persistent volume, creating and using a user-defined docker network, and publishing an image to Docker Hub.

The video demonstrates concrete proof supporting the implementation discussed in Section 2.

Video link: https://youtu.be/_RVZHQpAxe4

6. Conclusion

This report examines three interdependent components of contemporary cloud engineering. Virtualisation provides the foundation for efficient resource utilisation, while containerisation, through Docker, enables portable and consistent application deployment. Cloud services such as load balancing, auto-scaling and application gateways support traffic distribution, capacity adjustment, availability and application security.

The report also examines the main drivers of cloud adoption, including cost efficiency, elasticity, global availability, managed services and technological innovation. Serverless computing can reduce the infrastructure management burden and is particularly suitable for event-driven and intermittent workloads. Containerisation, on the other hand, provides application- and service-level control, portability and flexibility for long-running and customised workloads. Therefore, there is no universally best deployment model. Organisations should select technologies based on their workload characteristics, performance requirements, operational responsibilities, security needs and business objectives. Successful cloud engineering relies not on a single technology, but on selecting and combining appropriate services to achieve the right balance of scalability, resilience, security, cost efficiency and operational control.

7. References

- [1] Docker, Inc., "Docker Documentation", 2026. [Online]. Available: <https://docs.docker.com/> [Accessed: 30 June 2026].
- [2] T. Erl, R. Puttini and Z. Mahmood, Cloud Computing: Concepts, Technology & Architecture, 2nd ed. Pearson, 2021.
- [3] E. Mouat, Using Docker: Developing and Deploying Software with Containers, 3rd ed. O'Reilly Media, 2023.
- [4] Microsoft, "Hyper-V Documentation", Microsoft Learn, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/> [Accessed: 30 June 2026].
- [5] S. Newman, Building Microservices, 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.
- [6] Microsoft, "What is Azure Load Balancer?", Microsoft Learn, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview> [Accessed: 30 June 2026].
- [7] Microsoft, "Azure Application Gateway documentation", Microsoft Learn, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/> [Accessed: 30 June 2026].
- [8] M. Armbrust et al., "A View of Cloud Computing", Communications of the ACM, vol. 53, no. 4, pp. 50–58, 2010.
- [9] Amazon Web Services, "AWS Lambda Developer Guide", 2026. [Online]. Available: <https://docs.aws.amazon.com/lambda/> [Accessed: 30 June 2026].
- [10] Microsoft, "Azure Functions documentation", Microsoft Learn, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-functions/> [Accessed: 30 June 2026].