



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

**Containerisation, Cloud Scalability and Serverless
Computing:**

**A Critical Evaluation of Docker and Contemporary Cloud
Engineering Practice**

Module: Cloud Engineering Principles and Practice

Module Code: ACCA7027

Programme: MSc Cybersecurity and Digital Forensics

Student Name: YASH KACHHADIYA

Student ID Number: 2543707

Lecturer: Renuga Jayakumar

Table of Contents

- 1 Introduction
- 2 Docker, containerization and the cloud
 - 2.1 The Principles of Virtualisation
 - 2.1.1 Virtualization of Shared Hardware
 - 2.2 The Principles and Benefits of Containerisation
 - 2.2.1 Building an Image from a Dockerfile
 - 2.3 Images and Publishing Images
 - 2.3.1 Pushing an Image to Docker Hub
 - 2.4 Container Volumes
 - 2.5 Networking for Containers
 - 2.6 End-to-End Demo: Publishing of Web Server
- 3 Basic Cloud Computing Concepts
 - 3.1 Load Balancing
 - 3.2 Scalability
 - 3.3 Application Gateways
 - 3.4 Why Load Balancing and Scaling Are Complementary
 - 3.5 Advantages of the Three Technologies Integration
- 4 Drivers of Cloud Growth & Serverless Computing
 - 4.1 The Main Driving Forces: Cost and Flexibility
 - 4.2 What is a serverless function?
 - 4.3 Serverless vs. Containerization
- 5 Video Demonstration
- 6 Abstract
- 7 Bibliography

1. Introduction

Cloud computing has emerged as the de-facto approach to provide computer resources, changing the way businesses build, deploy and run software. Enterprises today consume computing, storage and networking as on-demand services that are invoiced according to utilization rather than owning and managing physical infrastructure [1]. This change has been driven mostly by two families of technology. The first is containerisation (e.g. Docker) and the second is the elastic service architecture that allow applications to reliably and economically fulfill variable demand (load balancing, scalability and application gateways) [2].

This study critically reviews containerisation and its significance in the cloud, including practical examples generated in class using Docker. It then considers three important cloud computing principles load balancing, scalability and application gateways – in the context of a major cloud provider's implementation, before evaluating the main reasons driving cloud adoption and contrasting serverless services with containerisation. The report is structured following the marking scheme: Section 2 addresses Docker and containerisation; Section 3 addresses the key cloud principles; Section 4 addresses the driving forces and serverless computing; Section 5 links to the associated video presentation; and Section 6 ends. In-text citations are in numeric form and a numbered reference list is given in Section 7.

2. Docker, Containerization and the Cloud

Docker is an open-source framework that consolidates an application together with its dependencies, libraries, and configurations into a singular portable entity known as a container[3]. A container encompasses all the necessary components for a program to operate, ensuring uniform behaviour across a developer's laptop, a testing server, and a production cloud setting, hence eradicating environment-specific configuration discrepancies that could lead to deployment failures. Docker was not the first container technology Linux namespaces and cgroups had existed for years but it was the first to package these primitives behind an accessible tooling and image format, which drove rapid adoption and led to the containerisation of the format as the Open Container Initiative (OCI) standard, now supported across all major cloud platforms [6]. This standardisation is precisely why containerisation matters to cloud engineering: it decouples an application from any single provider's infrastructure. This section builds on the underlying concept of virtualisation through to a full end-to-end demonstration of creating, running and publishing a containerised web server.

2.1 The Principles of Virtualisation

Virtualisation is the primitive abstraction on which modern cloud computing is built. It uses software (a hypervisor) to split one physical server into numerous isolated virtual machines (VMs), each running its own guest operating system as if it were installed on dedicated hardware [1][4]. There are two types of hypervisor: Type 1 (bare-metal) hypervisors, such as VMware ESXi and Microsoft Hyper-V, run directly on the physical hardware with no intervening host operating system, giving them lower overhead and better performance isolation, which is why they are the standard choice in cloud data centres. Type 2 (hosted) hypervisors, such as Oracle VirtualBox, instead run as an application on top of a conventional host operating system and are typically used for local development and testing rather than production workloads [4].

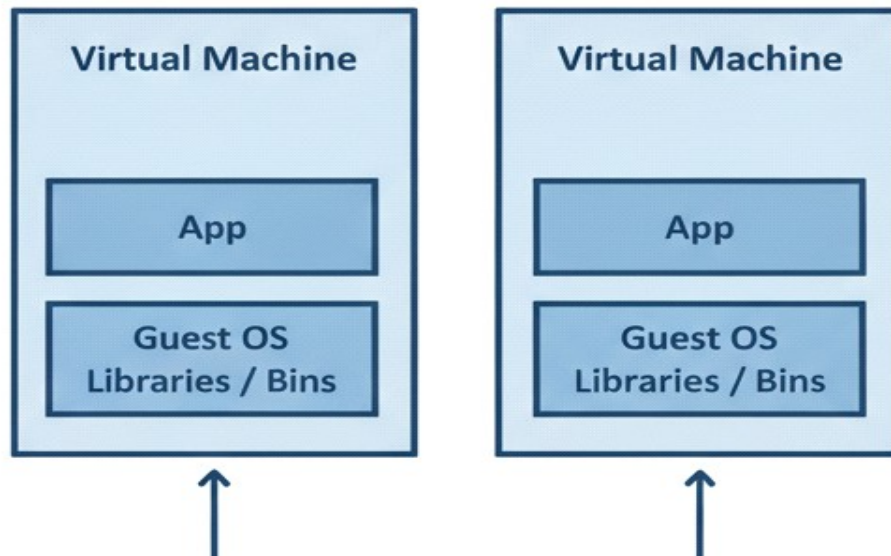
Virtualisation has strategic importance because it decouples a workload from the physical machine it runs on. Workloads can be provisioned, migrated, cloned or destroyed within minutes, and a single powerful physical server can be partitioned to support many tenants simultaneously. This directly enables the resource pooling and rapid elasticity that the widely-cited NIST definition identifies as essential characteristics of cloud computing, alongside on-demand self-service and measured service [1]. Critically, however, each VM still carries a full guest operating system, which consumes significant memory, storage and boot time a shortcoming that containerisation was later designed to overcome.

2.1.1 Virtualisation of Shared Hardware

Figure 1 shows the way that a hypervisor creates a virtualised shared physical hardware. The resources of a single host (CPU, memory, storage and network interfaces) are pooled and provided to several guest VMs. During class we showed this by running many segregated virtual machines on one host, each oblivious of the others, illustrating how a single hardware investment may be safely shared between distinct workloads.

```
PS C:\Users\Yash> docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED    STATUS    PORTS
7bc182ca0732   nginx     "/docker-entrypoint..." 7 days ago Up 19 seconds 80/tcp
3096bf7feb03   ubuntu    "/bin/bash"              3 weeks ago Up 21 seconds 0.0.0.0:82->80/tcp, [::]:82->80/tcp
89d8a39d6b37   nginx     "/docker-entrypoint..." 4 weeks ago Up 23 seconds 0.0.0.0:80->80/tcp, [::]:80->80/tcp
PS C:\Users\Yash>
```

Virtualisation of Share



2.2 The Principles and Benefits of Containerisation

Containerisation is a lightweight virtualisation technology that operates at the operating-system level rather than the hardware level. Rather than virtualising the host machine's kernel and hardware in full, Linux containers rely on two kernel features namespaces, which isolate a process's view of resources such as the file system, network stack and process IDs, and control groups (cgroups), which limit and account for the CPU, memory and I/O each container may consume [5]. Together these mechanisms isolate the user space, file system and processes of each application [3][5] while all containers on a host continue to share a single underlying kernel. Because no guest operating system is duplicated, containers are typically megabytes rather than gigabytes in size, start in seconds or less, and allow far higher workload density on the same hardware than an equivalent set of VMs [5].

The benefits of this approach are broadly four-fold. First, portability: the classic "it works on my machine" problem is largely solved, as the container packages all dependencies together with the application [3]. Second, efficiency: the shared-kernel model has little overhead, improving hardware utilisation and reducing infrastructure cost. Third, consistency across the software lifecycle, enabling reliable continuous integration and continuous delivery (CI/CD) pipelines. Fourth, elasticity at scale: identical containers can be replicated rapidly to meet demand and orchestrated automatically by systems such as Kubernetes [6]. This efficiency is not without cost, however because containers share a kernel, they provide weaker isolation than a VM, which is a security consideration relevant to multi-tenant cloud environments. Table 1 compares virtual machines and containers directly.

Table 1. Comparison of virtual machines and containers.

Characteristic	Virtual Machine	Container
Abstraction level	Hardware (hypervisor)	Operating system kernel
Guest OS	Full OS per VM	Shares host kernel
Typical size	Gigabytes	Megabytes
Start-up time	Minutes	Seconds or less
Isolation	Strong (full OS)	Process-level
Density per host	Lower	Much higher

2.2.1 Constructing a Docker file to Build an Image

A Docker file is a script of plain instructions that Docker reads sequentially to assemble an image automatically and reproducibly [3]. Each command adds a layer (read-only). Docker caches layers so that unaltered stages do not have to be recreated. The following Docker file, created in class, builds a package of a small static web page to be served by the lightweight Nginx web server.

```
PS C:\Users\yash\Cloud Engineering Assignment> cat Dockerfile
FROM nginx:alpine

LABEL maintainer="yash"

COPY ./site/ /usr/share/nginx/html/

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
PS C:\Users\yash\Cloud Engineering Assignment> █
```

FROM picks a small, official base image. COPY adds our local HTML page to the directory Nginx serves. EXPOSE tells what port to listen on. CMD defines the process to execute when the container is launched. The image is then created and labeled in a single command, as illustrated in Figure 2.

```
docker build -t mywebserver:1.0 .
```

```
> docker build -t mywebserver:1.0 .
[+] Building 1.3s (7/7) FINISHED
=> [internal] load build definition from Dockerfile                                0.0s
=> => transferring context: 2B                                                    0.0s
=> [internal] load build context                                                  0.0s
=> => transferring context: 61B                                                    0.0s
=> [1/2] FROM docker.io/library/nginx:alpine@sha256:4a73073bd557c65b759505da037898b61f1be6cbcc3c2c3 0.1s
=> => resolve docker.io/library/nginx:alpine@sha256:4a73073bd557c65b759505da037898b61f1be6cbcc3c2c3 0.1s
=> CACHED [2/2] COPY ./site/ /usr/share/nginx/html/                             0.0s
=> exporting to image                                                            0.3s
=> => exporting layers                                                            0.0s
=> => exporting manifest sha256:5a7859e2e52f640462b2276c8b2e8aa275e334761574b2b05b5b0cd25f22806e 0.0s
=> => exporting config sha256:7a50780981719a847f7eba873f36f88d5545859a1a61847e2363e28f91c57e55 0.0s
=> => exporting attestation manifest sha256:5cb031281baff0603ba783a9ba4f77802e95116685cd3057338eb9e 0.1s
=> => exporting manifest list sha256:678d3922d47da1ff61b818499e77a79ad743909e39f8867f01f7d65d86cce9 0.0s
=> => naming to docker.io/library/mywebserver:1.0                               0.0s
=> => unpacking to docker.io/library/mywebserver:1.0                             0.0s
```

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/ue5xs468mb4ens5zzl5kyv9r4
PS C:\Users\yash\Cloud Engineering Assignment>

Figure 2. Building a Docker image from the Docker file, showing each instruction creating a cached layer.

2.3 Images and Publishing Images

A Docker image is an immutable, read-only template built from a Dockerfile, and a container is a running instance of that image [3]. Images consist of layers stacked on top of one another, and because each layer is content-addressed, Docker stores any given layer only once and allows multiple images to reference the same underlying layer rather than duplicating it. This means images built from a common base such as the same Nginx base image used across several projects share those foundation layers, which saves storage space on disk and reduces the amount of data that must be transferred when pushing or pulling an image. After an image has been built and tested locally, it can be published to a registry so that other developers, or a production cloud platform, can pull and run it directly. The default public registry is Docker Hub, although cloud providers also offer private alternatives such as Amazon Elastic Container Registry and Azure Container Registry for organisations that do not want images to be publicly accessible [6].

2.3.1 Pushing an Image to Docker Hub

Publishing to Docker Hub entails authentication, labeling the image with your account namespace and pushing it. This was taught in class and illustrated in Figure 3. Tagging connects the local image with a repository on the user's Docker Hub account.

```
docker login
```

```
docker tag mywebserver:1.0 yashpatel201220/mywebserver:1.0
```

```
docker push myusername/mywebserver:1.0
```

[Insert screenshot here]

Figure 3. Pushing the tagged image to Docker Hub, with each layer uploaded and the repository made available for others to pull.

The image is portable enough that, once published, it can be pulled onto any Docker-enabled host with a single command, which is the portability that makes containers essential to cloud deployment.

```
docker pull myusername/mywebserver:1.0
```

2.4 Container Volumes

Containers are ephemeral by design: when a container is removed, any data written to its writable layer is lost along with it. This behaviour is appropriate for stateless application logic but unsuitable for stateful workloads such as databases, where data must persist independently of any single container's lifecycle. Docker volumes solve this by providing persistent storage that is managed directly by the Docker daemon and stored on the host filesystem, outside the writable layer of any individual container [3][5]. A volume can be mounted into a container at a specified path so that data survives container restarts, image upgrades and even container removal, and because a volume exists independently of any one container the same volume can also be mounted into multiple containers simultaneously, allowing them to share data directly, for example a set of worker containers all reading from and writing to a common data directory. This differs from a bind mount, which maps a specific host directory into a container; volumes are the preferred mechanism in most cases because they are fully managed by Docker and are portable across host operating systems. Figure 4 shows the creation of a volume and its attachment to a container.

```
docker volume create mydata
```

```
docker run -d --name web -v mydata:/usr/share/nginx/html mywebserver:1.0
```

[Insert screenshot here]

Figure 4. Creating a Docker volume and mounting it into a running container so that data persists beyond the container lifecycle.

2.5 Container Networking

Container networking governs how containers communicate with each other and with the outside world. Docker provides several network drivers, including bridge (the default, for containers on a single host), host (which removes network isolation entirely), none, and overlay (for multi-host communication in a cluster). The default bridge network allows communication between containers on the same host but requires containers to address each

other by IP, since it does not provide automatic DNS resolution. User-defined bridge networks improve on this by adding automatic service discovery: Docker runs an internal DNS resolver so containers can refer to one another by container name rather than a fixed IP address, which is more resilient, since a container's IP address can change if it is restarted or rescheduled [3]. This is particularly important for multi-container systems, such as the web server built in this report's end-to-end demonstration (Section 2.6), which could reference a separate database container by name for example db:5432 without needing to track its IP address directly. Figure 5 illustrates the creation of a user-defined network and the attachment of a container to it.

```
docker network create appnet
```

```
docker run -d --name web --network appnet -p 8080:80 mywebserver:1.0
```

[Insert screenshot here]

Figure 5. Creating a user-defined Docker network and attaching a container, enabling name-based service discovery between containers.

2.6 End-to-End Demonstration: Publishing a Web Server

Combining the previous demonstrations shows the overall containerisation workflow end to end. In class, we wrote a static HTML page, packaged it with the Docker file from Section 2.2.1, built it into an image, ran it locally as a container, verified it in a browser, and published the finished image to Docker Hub. The full sequence was:

1. Create the HTML page and the Docker file.
2. Build the image: `docker build -t mywebserver:1.0`.
3. Run the container: `docker run -d -p 8080:80 mywebserver:1.0`.
4. Verify the page loads at `http://localhost:8080` in a web browser.
5. Tag and push the image to Docker Hub so it can be reused elsewhere.

This single workflow exercises image building, container execution and image publication together, and could be extended with the volume and network steps demonstrated separately in Sections 2.4 and 2.5 to represent a fully production-ready deployment. It highlights why Docker is so significant in cloud engineering: an application validated locally can be deployed unchanged to any cloud platform, then scaled and coordinated to meet demand [6]. This combination of reliability and portability is precisely why containers have become the de facto packaging format for cloud-native applications.

3. Core Cloud Computing Concepts

Load balancing, scalability and application gateways are three interdependent mechanisms that together allow a cloud service to remain resilient and responsive under variable demand. Each addresses a different part of the same underlying problem: scalability determines how much capacity exists, load balancing determines how traffic is spread across that capacity, and an application gateway governs how traffic is inspected and routed before it ever reaches a backend instance. The sections below describe how a major cloud provider, Microsoft Azure, implements each concept, with Amazon Web Services (AWS) referenced for meaningful comparison, before examining how the three concepts reinforce one another in practice [2][7].

3.1 Load Balancing

Load balancing is the process of distributing incoming network traffic among multiple backend servers so that no single instance is overloaded, improving both responsiveness and availability [7]. Azure Load Balancer operates at Layer 4 (the transport layer), distributing TCP and UDP traffic using a hashing algorithm combined with continuous health probes. Because it makes routing decisions without inspecting application-layer content, it introduces minimal latency, which makes it well suited to high-throughput workloads such as gaming backends or database clusters, in contrast to the content-aware, application-layer routing provided by Azure Application Gateway (discussed in Section 3.3). If a backend instance fails a health probe, the load balancer immediately stops directing traffic to it, providing automatic fault tolerance without operator intervention. The analogous AWS service is the Elastic Load Balancing family, specifically the Network Load Balancer, which likewise operates at Layer 4 and is designed for the same class of latency-sensitive, high-throughput traffic, whereas AWS's Application Load Balancer serves the Layer 7 use case instead [2]. An important setup step that exemplifies the concept in practice is the health probe: the operator defines the endpoint, protocol and polling interval that the load balancer uses to evaluate whether an instance is healthy, directly determining how quickly a failing instance is removed from rotation.

3.2 Scalability

Scalability is the ability of a system to handle increasing load by adding resources. There are two types of scaling: vertical scaling (scaling up), in which an existing instance is given more CPU, memory or storage, and horizontal scaling (scaling out), in which additional instances are added alongside existing ones [1][7]. Vertical scaling is simple but limited it is bounded by the largest instance size a provider offers, and typically requires downtime to resize. Cloud platforms therefore favour horizontal scaling, since it removes any single point of failure or

hardware ceiling and allows capacity to be added or removed incrementally without interrupting the running service. Azure implements this through Virtual Machine Scale Sets combined with auto scale rules that automatically add or remove instances based on a defined criterion, such as CPU utilisation or a fixed schedule. For example, a rule might add an instance when average CPU exceeds 70% for five minutes, and remove one when it falls below 30% meaning capacity is continually adjusted to track actual demand, and cost tracks capacity in turn, so the organisation pays only for resources it is genuinely using. The equivalent AWS service is Auto Scaling Groups, which apply the same demand-driven scaling policy model [2].

3.3 Application Gateways

An application gateway is a Layer 7 (application-level) load balancer that makes routing decisions based on the content of HTTP/HTTPS requests for example the URL path, host header, or other request headers rather than on raw transport-layer traffic alone [7]. The Azure Application Gateway can route requests for /images to one backend pool and /api to another (path-based routing), terminate SSL/TLS to offload encryption from backend servers, and, through its integrated Web Application Firewall (WAF), inspect traffic for common attacks such as SQL injection and cross-site scripting before any request reaches an application server. This positions the gateway as the first point of content inspection in the request path, ahead of the Layer 4 load balancing and autoscaling discussed in Sections 3.1 and 3.2. This capability is directly relevant to a cybersecurity and digital forensics context, since it provides a centralised, hardened and auditable entry point into the application rather than exposing backend servers directly to untrusted traffic. The equivalent AWS service is the Application Load Balancer, which similarly supports path- and header-based routing and integrates with AWS WAF for the same class of request-level threat inspection [2].

3.4 Why Load Balancing and Scaling Are Complementary

Load balancing and scaling are complementary because each solves a problem the other cannot solve alone. Scaling out creates additional instances, but those instances are of no benefit unless traffic can be routed to them this is the load balancer's role. Conversely, a load balancer can only distribute traffic across the instances that already exist; it cannot create new capacity when the existing pool is insufficient to meet demand [7]. In practice the two work as a single feedback loop: an auto scale rule detects rising load and provisions new instances, the load balancer's health probes detect that a new instance has become healthy, and traffic is then routed to it all without manual intervention. When demand falls, the reverse occurs: instances are removed, and the load balancer simply stops routing to them. The result is a system that elastically expands and contracts while ensuring every request is still served throughout, which is the practical, mechanical realisation of the elasticity that the cloud computing model promises in principle [1].

3.5 Benefits of Combining the Three Technologies

Combining load balancing, scaling and an application gateway creates a service that is simultaneously secure, resilient and cost-efficient. Consider a seasonal sale at an e-commerce site at its peak. The application gateway terminates SSL/TLS, screens incoming requests for malicious patterns via its Web Application Firewall, and routes each request to the correct backend pool based on its path. Autoscaling detects the resulting CPU spike and provisions additional instances to absorb the extra load. The load balancer then distributes traffic evenly across all healthy instances, while automatically redirecting traffic away from any instance that fails a health check [2][7]. From the customer's perspective the result is a fast, seamless and secure experience; from the business's perspective, the architecture avoids both the lost revenue of an outage during peak demand and the wasted cost of permanently over-provisioning infrastructure for a peak that occurs only occasionally. This is the core economic trade-off the three technologies jointly resolve: without them, an organisation must either risk under-provisioning (poor performance or outages at peak) or accept the sunk cost of over-provisioning (idle capacity for most of the year). Together, they transform a rigid, fixed-capacity deployment into an adaptive service that scales securely to meet actual demand.

4. Driving Forces Behind Cloud Expansion and Serverless Computing

4.1 The Main Driving Forces: Cost and Flexibility

The growth of cloud computing is driven by a number of reinforcing forces, the most important of which are cost and flexibility [1][8]. In terms of cost, the cloud replaces large upfront capital expenditure (CapEx) on hardware with pay-as-you-go operating expenditure (OpEx); organisations pay only for what they consume and avoid the sunk cost of over-provisioning for peak demand [8]. Hyperscale providers can typically deliver compute resources more cheaply than most organisations could achieve independently, owing to economies of scale in hardware procurement, power and staffing. This shift is not without trade-offs, however: OpEx-based billing can be harder to forecast than a fixed capital purchase, and reliance on a single provider's proprietary services can introduce vendor lock-in, a cost that is easy to overlook when evaluating cloud adoption purely on price.

Beyond cost, the cloud provides flexibility through fast provisioning, near-infinite elastic capacity, and global reach, enabling organisations to deploy close to their users and to experiment cheaply, since failed experiments can be discontinued with no stranded physical assets [1]. Additional benefits include a reduced maintenance burden, built-in resilience and disaster recovery capabilities, and access to advanced managed services such as analytics and machine learning platforms that would be prohibitively expensive to build and maintain in-house. These same forces, particularly the drive toward minimising operational overhead and paying only for actual usage, are taken to their logical conclusion in the serverless computing model discussed next.

4.2 Serverless Functions Explained

Serverless computing, also known as Functions as a Service (FaaS), is a computing paradigm where the cloud provider autonomously allocates and oversees the foundational servers, while the developer contributes just the application code in the form of distinct functions [9]. Servers continue to exist regardless of the nomenclature; the designation merely indicates that the creator no longer supplies, updates, or oversees them. Services such as AWS Lambda and Azure Functions execute a function in response to an event an HTTP request, a file upload, a message arriving in a queue, or a scheduled timer and scale from zero to thousands of concurrent executions and back again largely automatically, though the first invocation after a period of inactivity can incur a brief "cold start" delay as the provider initialises a new execution environment [9][10]. Billing is calculated per invocation and per unit of execution time, so when no events occur, nothing executes and nothing is charged. AWS Lambda and Azure Functions differ mainly in integration depth with their respective ecosystems Lambda is tightly bound to AWS services such as S3 and API Gateway, while Azure Functions integrates similarly with Azure Blob Storage and Event Grid but both express the same underlying model. This event-driven, scale-to-zero paradigm is a direct expression of the cost and flexibility pressures discussed in Section 4.1, and suits intermittent, bursty or unpredictable workloads particularly well.

4.3 Serverless versus Containerisation

Serverless functions and containerisation are complementary cloud-native techniques that express the same underlying driving forces cost efficiency and flexibility in different ways, and the choice between them involves clear trade-offs [9][10]. Containers package a whole application and all its dependencies into a portable unit that the operator still deploys, scales and manages; serverless instead abstracts the runtime away entirely, so the developer is responsible only for application code. The comparison is summarised in Table 2.

Table 2. Comparison of containerisation and serverless functions.

Factor	Containerisation (Docker)	Serverless (FaaS)
Unit of deployment	Whole application/service	Individual function
Management burden	Operator manages scaling/orchestration	Provider manages everything
Scaling	Manual or orchestrated (e.g. Kubernetes)	Automatic, scale to zero
Cost model	Pay for running instances	Pay per invocation/time
Portability	High (runs on any host)	Lower (provider lock-in)
Best suited to	Long-running, complex, portable apps	Short, event-driven, bursty tasks

In terms of driving forces, serverless maximises cost efficiency for intermittent workloads through scale-to-zero, pay-per-invocation billing, and maximises flexibility by removing infrastructure management entirely; however, it can suffer cold-start latency on the first request after idling, and typically increases dependence on a single provider's platform [10]. Containers offer greater mobility between cloud providers, avoid this lock-in, and give the operator more control over the runtime and more predictable performance for long-running or resource-intensive workloads, at the cost of the operator taking on responsibility for scaling and orchestration [9]. In practice the two approaches are frequently blended rather than treated as a binary choice for example, a containerised e-commerce application handling core order processing might offload an episodic task such as thumbnail generation or invoice PDF creation to a serverless function, allowing the organisation to capture the cost and flexibility advantages of each approach exactly where it best applies.

5. Video Demonstration

This report is complemented by a supporting video (max. 15 minutes) that demonstrates the Docker examples detailed in Section 2, such as producing an image from a Docker file, starting a container, creating a volume and a network, and publishing the image to Docker Hub, with a verbal explanation of each step. The video is uploaded as “Unlisted” and can be viewed at the below link.

Video link: [<https://youtu.be/T0--kgXQInw>]

6. Conclusion

This report has examined the key role that containerisation plays in the cloud, situating it within the broader set of concepts that make cloud services robust and economically viable. It first considered virtualisation the abstraction of physical hardware into isolated virtual machines and showed how containerisation takes this same underlying idea and distils it down to the operating-system level, producing lightweight, portable and efficient application packaging by comparison. Practical demonstrations with Docker covered the entire container lifecycle: writing a Docker file, building and running an image, persisting data using volumes, connecting containers via a user-defined network, and publishing the finished image to Docker Hub for reuse on any Docker-enabled host.

The report then examined load balancing, scalability and application gateways. It found that load balancing and scaling are complementary scaling creates capacity, while load balancing directs traffic to it and that combining both with an application gateway produces a service that is simultaneously secure, resilient and cost-aligned to actual demand. It further concluded that cost and flexibility remain the principal drivers of cloud adoption, and that serverless functions and containerisation, rather than competing directly, are complementary tools whose trade-offs portability and operator control versus fully managed, scale to zero economics should guide the choice per workload rather than dictate a single organisation-

wide standard. Collectively, these technologies represent the core engineering principles that enable businesses to build, deploy and operate modern cloud services reliably and at scale, and their effective combination, rather than any one technology in isolation, is what distinguishes mature cloud engineering practice.

7. 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, 2011.
- [2] Amazon Web Services, “Overview of Amazon Web Services,” AWS Whitepaper, 2023. [Online]. Available: <https://docs.aws.amazon.com/whitepapers/latest/aws-overview/>
- [3] Docker Inc., “Docker overview,” Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/get-started/overview/>
- [4] VMware, “What is a Hypervisor?” VMware Glossary, 2023. [Online]. Available: <https://www.vmware.com/topics/glossary/content/hypervisor.html>
- [5] S. Newman, Building Microservices, 2nd ed. Sebastopol, CA: O’Reilly Media, 2021.
- [6] The Kubernetes Authors, “Kubernetes Documentation: Concepts,” 2024. [Online]. Available: <https://kubernetes.io/docs/concepts/>
- [7] Microsoft, “Load-balancing options – Azure Architecture Center,” Microsoft Learn, 2024. [Online]. Available: <https://learn.microsoft.com/azure/architecture/guide/technology-choices/load-balancing-overview>
- [8] M. Armbrust et al., “A view of cloud computing,” Communications of the ACM, vol. 53, no. 4, pp. 50–58, 2010.
- [9] Cloud Native Computing Foundation, “CNCF Serverless Whitepaper,” 2023. [Online]. Available: <https://github.com/cncf/wg-serverless>
- [10] Amazon Web Services, “AWS Lambda Developer Guide,” 2024. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>