

MSc Cyber Security and Digital Forensics
ACCA7027-Cloud Engineering Principles and Practice

Containerisation, Cloud Computing Concepts, Docker and Serverless Functions

Lecturer: Renuga Jayakumar



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

TABLE OF CONTENTS

1. Introduction	3
2. Cloud Technology: Virtualization and Containerization	3
2.1 The Concepts of Virtualization	3
2.2 The Concepts and Advantages of Containerization	4
2.3 Dockerfile and Image	5
2.4 Container, Volumes, and Networking	7
2.4.1 Container	7
2.4.2 Volumes	7
2.4.3 Networking	7
2.5 Publishing Images to DockerHub	9
3. Cloud Computing Concepts: Load Balancing, Scalability, and Application Gateways	9
3.1 Load Balancing in Cloud Platforms	9
3.2 Scalability in Cloud Platforms	10
3.3 Layer 7 Application Load Balancing	10
3.4 Why Load Balancing and Scaling Are Complementary	11
3.4.1 Health Probes and Auto Scale Configuration	11
3.5 Demonstrating the Combined Benefits	12
4. Forces of Cloud Growth, and the Emergence of Serverless Computing	12
4.1 The Main Driving Forces Behind Cloud Adoption	12
4.2 Serverless Functions: What They Are and How They Work	13
4.3 Comparing Serverless Functions with Containerization	14
4.3.1 Execution Model and Infrastructure	14
4.3.2 Scalability and Cost	14
4.3.3 Portability	14
4.3.4 Developer Experience and Operational Overhead	15
4.3.5 Identifying the Driving Forces	15
4.3.6 Cold Start Mitigation and Production Patterns	15
5. Conclusion	16
References	17

1. Introduction

Cloud computing is now the dominant paradigm for delivering IT services at scale and the technologies that support cloud computing are still changing very fast. The ubiquitous implementation of containerization, the coming of age of cloud-native load balancing and auto-scaling services, and the rise of serverless computing as a standard architectural design have been three of the most important changes of the last ten years.

This report critically examines these areas in detail. Part 2 explores virtualization and containerization, and practical examples of Docker are based on the work in the class. Section 3 examines the core concepts of load balancing, scalability, and application gateways across leading cloud platforms and explains why these technologies are complementary. Section 4 explores the primary impetus in cloud growth and compares critically serverless functions with containerization as delivery models. A demonstration video accompanies this report which is available at the link in Section 2.5.

By 2025, containers are in use or under consideration by more than 90 percent of organizations in one way or another,[\[1\]](#) and by 2034 the global market in serverless computing is projected to be USD 92.22 billion,[\[2\]](#) so that the question of how they operate and when each one is appropriate is no longer a specialist issue but an essential part of every cloud engineer.

2. Cloud Technology: Virtualization and Containerization

2.1 The Concepts of Virtualization

Virtualization refers to the process of abstracting physical computing resources (CPU, memory, storage and network) to enable several isolated environments to execute on the same physical machine. The concept dates back to the 1960s with the CP/CMS mainframe system at IBM, but contemporary virtualization as applied to cloud infrastructure began in earnest with the x86 hypervisor technology of VMware in the late 1990s.[\[3\]](#)

A hypervisor is a portion of software in a virtualized environment that lies between the hardware and the guest operating systems. Type 1 (bare-metal) hypervisor operating systems operate directly on the host hardware such as Microsoft Hyper-V, VMware ESXi and KVM. Type 2 hypervisors are based upon a host operating system, such as VirtualBox or VMware Workstation.[\[4\]](#) Each Virtual Machine (VM) packages a full guest operating system including its own virtualized CPU, memory, and disk, and the application, making VMs highly isolated and resource-intensive.[\[5\]](#)

The key benefit of the virtualisation to cloud providers is the ability to consolidate servers: one physical server can contain a large number of VMs that can be billed as independent entities. Amazon EC2, Google Compute Engine and Azure Virtual Machines all base their infrastructure around this, [\[3\]](#) but each VM incurs the cost of a full OS, usually occupying gigabytes of memory even when idle, and takes minutes to boot up, not seconds.[\[6\]](#)

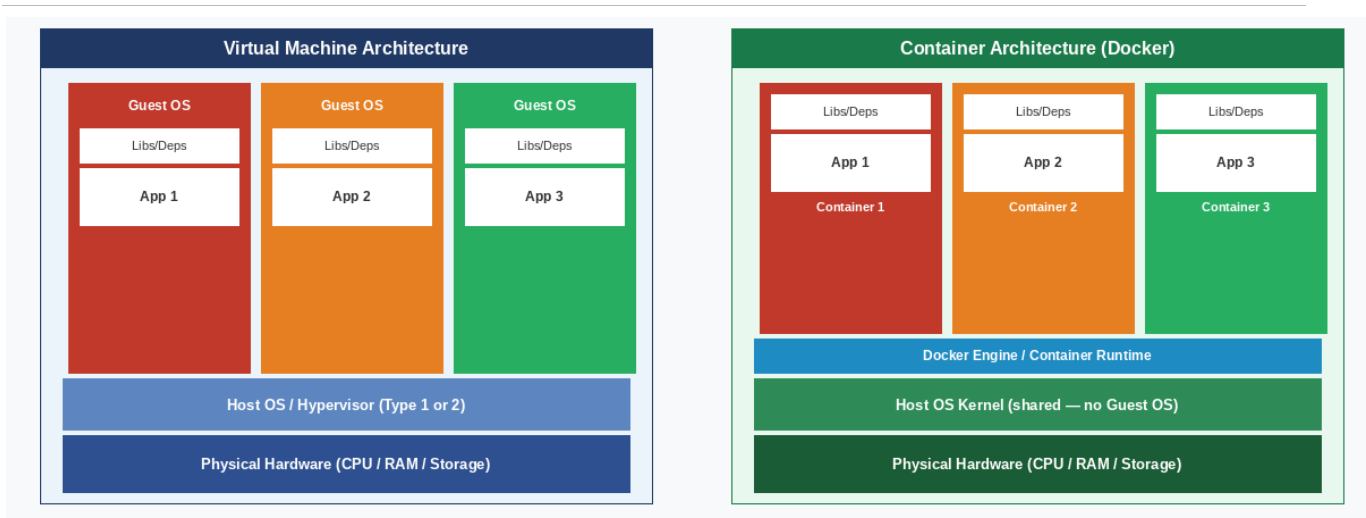


Figure 1: Architecture comparison between Virtual Machines (left) and Docker Containers (right). VMs carry a full guest OS per instance; containers share the host OS kernel, dramatically reducing overhead.

The fundamental architectural difference is shown in figure 1. A VM environment needs its own guest operating system per instance of application, and this overheads hundreds of megabytes per application instance. A study by (Aqasizade et al., 2024) had concluded that a server with 12 KVM-based VMs could support up to 89 Docker containers under the same workloads as a server with 12 VM-based deployments, and that containerized environments had resource utilization rates of up to 78% relative to 42% in VM-based deployments. [4]

2.2 The Concepts and Advantages of Containerization

Containerization takes a different approach to isolation. Containers are isolated user spaces that share the host operating system kernel and use two Linux primitives (namespaces to isolate processes and cgroups to set resource limits) to create isolated user spaces instead of virtualizing hardware and executing distinct operating systems.[4] Docker, originally published in 2013, by Solomon Hykes, created a simple toolchain to build, distribute, and run containers. [7]

Containerisation offers significant advantages in cloud engineering. Containers are started in milliseconds instead of minutes, since there is no operating system to boot into them, instead of gigabytes to boot a VM image into a production environment in the cloud. [8] They are lightweight; a typical container image is a few megabytes to a few hundred megabytes, compared to gigabytes to boot a VM image into a cloud production environment. [6] Containers are highly portable: Docker's philosophy of 'build once, run anywhere' means a container tested on a developer's laptop will behave identically in a cloud production environment.[9] A 2024 Forrester study found that enterprises adopting Docker reduced infrastructure costs by an average of 66% while increasing developer productivity by 43%.[10]

Docker provides isolation using two features of the Linux kernel. Namespaces provide distinct perspectives on the system to each container - a container has its own process tree (PID namespace), network interfaces (network namespace), hostname (UTS namespace), and filesystem mount points (mount namespace). [4] This isolation and resource management is what makes containers safe to co-exist with other containers on the same physical machine, and why cloud providers can support much more container instances per physical machine than VM instances. [6]

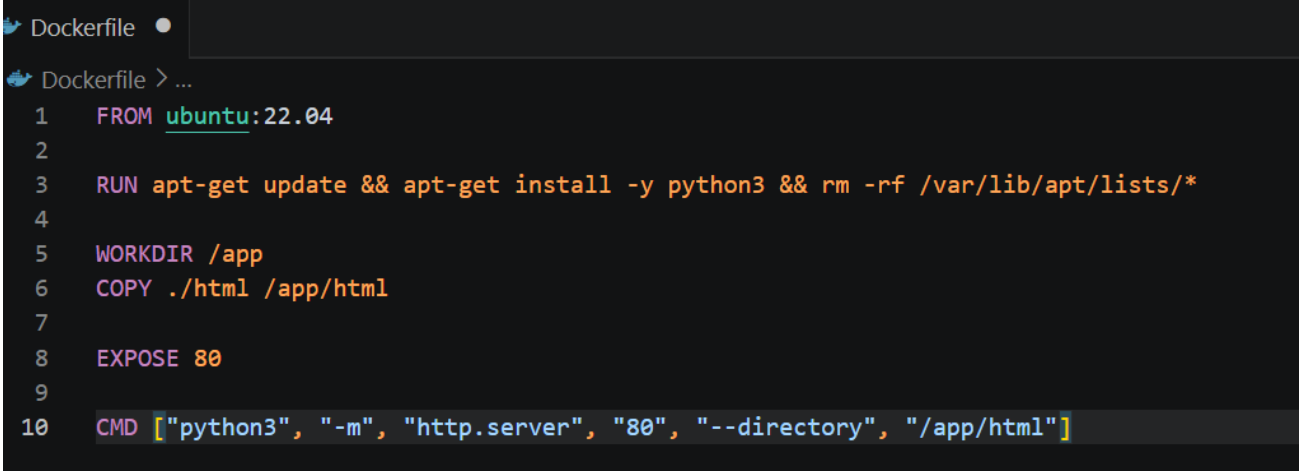
Another important design principle is the Docker image layering system. Each command in a Dockerfile forms a new immutable layer over the last one. [11] Each time a new version of the image is created, only the layers that have changed must be downloaded or stored - layers that are the same as an existing version are used out of the cache. In the case of a team that releases updates regularly, it implies that a 500MB base image can only

need a 5MB patch download to be updated to most releases, which saves a lot of time and bandwidth expense on deployment. [\[9\]](#)

Containerization is another natural fit with microservices architecture, in which complex applications are broken down into small, deployable services. Microservices can be scaled independently according to demand and updated without affecting other services and each microservice is executed in its own container, [\[1\]](#) which Gartner estimates will run over 95 percent of all new digital workloads by 2025, mostly on cloud-native, containerized platforms. [\[1\]](#)

2.3 Dockerfile and Image

A Dockerfile is a text file containing a sequence of instructions that Docker follows to build an image. Every instruction builds a new layer in the image, and Docker stores these layers to accelerate future builds. [\[11\]](#) The next Docker file makes a simple web server which serves a custom HTML page with Python built-in HTTP server module.



```
Dockerfile
1 FROM ubuntu:22.04
2
3 RUN apt-get update && apt-get install -y python3 && rm -rf /var/lib/apt/lists/*
4
5 WORKDIR /app
6 COPY ./html /app/html
7
8 EXPOSE 80
9
10 CMD ["python3", "-m", "http.server", "80", "--directory", "/app/html"]
```

The FROM instruction is the base image, in this case, Ubuntu 22.04 LTS, a popular and stable Linux distribution. [\[11\]](#) The RUN instruction is an instruction that is run during the build process; the flag of --rm is used to ensure the apt-get cache is empty to keep the final image lean. WORKDIR establishes the working directory of the further instructions. COPY copies the local html directory to the image. EXPOSE specifies that the container listens on port 80 and CMD command that is used when the container is started.

In order to create the image based on this Dockerfile, the following command is entered:

```

PS C:\Users\ASUS\Projects\my-web-server> docker build -t my-server .
[+] Building 1.3s (9/9) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile             0.0s
=> => transferring dockerfile: 265B                             0.0s
=> [internal] load metadata for docker.io/library/ubuntu:22.04 0.5s
=> [internal] load .dockerignore                               0.0s
=> => transferring context: 2B                                   0.0s
=> [1/4] FROM docker.io/library/ubuntu:22.04@sha256:eb29ed27b0821dca 0.1s
=> => resolve docker.io/library/ubuntu:22.04@sha256:eb29ed27b0821dca 0.0s
=> [internal] load build context                               0.0s
=> => transferring context: 333B                                 0.0s
=> CACHED [2/4] RUN apt-get update && apt-get install -y python3 && 0.0s
=> CACHED [3/4] WORKDIR /app                                   0.0s
=> [4/4] COPY ./html /app/html                                0.1s
=> exporting to image                                           0.4s
=> => exporting layers                                           0.1s
=> => exporting manifest sha256:a411681a90e2cb40c7c241f11d1746988d99 0.0s
=> => exporting config sha256:9e676febee21bd5e982a38e28caf2836a56c9c 0.0s
=> => exporting attestation manifest sha256:c9c35fe153d57a2ab5cab5ac 0.1s
=> => exporting manifest list sha256:db7a489214499fd36df0cece22ddca5 0.0s
=> => naming to docker.io/library/my-server:latest             0.0s
=> => unpacking to docker.io/library/my-server:latest           0.0s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/o5orv56ys5aew1etf5ae1sigs
PS C:\Users\ASUS\Projects\my-web-server>

```

The `-t` option gives the image a name and label. The final `.` indicates the current directory to the build context, which Docker uses to read the Dockerfile and any other files mentioned in COPY instructions. [\[11\]](#) An image may be listed after being built with:

```

PS C:\Users\ASUS\Projects\my-web-server> docker images

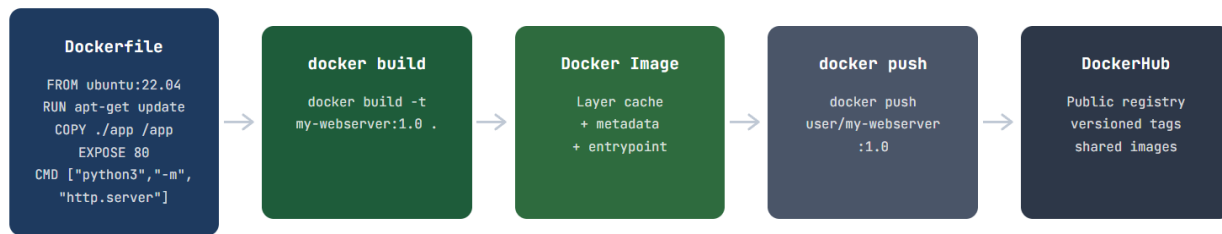
```

IMAGE	ID	DISK USAGE	CONTENT SIZE	EXTRA
alpine:latest	25109184c71b	13.1MB	3.95MB	U
docker/welcome-to-docker:latest	c4d56c24da4f	22.2MB	6.03MB	U
my-server:latest	db7a48921449	161MB	40.1MB	U
my-web-server:1.0	dcd8a69d2b3d	161MB	40.1MB	U
my-webserver:1.0	c31d4714fd65	161MB	40.1MB	U
nginx/nginx-ingress:latest	c7e16a426c93	422MB	107MB	U
nginx:latest	bc45d248c4e1	240MB	65.8MB	U

```

PS C:\Users\ASUS\Projects\my-web-server>

```



The Docker lifecycle: a Dockerfile defines the image; docker build creates it; docker push publishes to DockerHub.

Figure 2: The Docker Lifecycle

2.4 Container, Volumes, and Networking

2.4.1 Container

Once the image has been created, the container can be launched with:

```
PS C:\Users\ASUS\Projects\my-web-server> docker run -d -p 8080:80 --name my-we
bc my-server
4c4e5da5c8e9184fd1314f30f535d88d4a141a08a85506dbc6ef94cdd75bc9ca
PS C:\Users\ASUS\Projects\my-web-server>
```

The -d flag allows the container to run in detached mode (background) and -p 8080:80 binds port 80 on the host to port 80 on the container, allowing the web server to be accessed at <http://localhost:8080>. [9] --name assigns a human-readable identifier to the running container. The web server has now been started and is serving the HTML content that is bundled in the image.

2.4.2 Volumes

Any data written into a running container is lost by default when the container is stopped. Docker volumes address this by providing storage that is independent of the lifecycle of the container, and is managed by Docker on the host filesystem. [12]

```
PS C:\Users\ASUS\Projects\my-web-server> docker volume create web-content
web-content
PS C:\Users\ASUS\Projects\my-web-server>

PS C:\Users\ASUS\Projects\my-web-server> docker run -d -p 8080:80 -v web-con
tent:/app/html --name my-webserver my-server
e47a958e30332c1566ac835ecb0014dba343f99596e670302459c807774fe281
PS C:\Users\ASUS\Projects\my-web-server>
```

Now any changes to files in web-content on the host are immediately reflected inside the container at /app/html.[12] This can be especially handy in development environments where developers wish to edit HTML or configuration files without regenerating the image and in production when persistent data needs to be maintained even when the container is restarted, such as databases or logging. Multiple containers can also share volumes - e.g. a web container and a logging container can both mount the same volume.

2.4.3 Networking

Docker provides several networking modes for containers within a host.[13] Containers within a host are linked to a bridge network and may interact with one another by means of container names as DNS hostnames in case they share a user-defined network. It is easy to make a custom network:

```
PS C:\Users\ASUS\Projects\my-web-server> docker network create my-app-network
0b8178e882f48901718ef076e89bcc7176ec83713f3e3a6c120a49565d6ac1cc
PS C:\Users\ASUS\Projects\my-web-server>
```

```
PS C:\Users\ASUS\Projects\my-web-server> docker run -d --name web --network
my-app-network my-server
6bf598e8d82490340cb58cd11346b453d6b4d75d7017277534fd68c9d91a868f
PS C:\Users\ASUS\Projects\my-web-server>
```

```
PS C:\Users\ASUS\Projects\my-web-server> docker run -d --name db --network m
y-app-network postgres:15
Unable to find image 'postgres:15' locally
15: Pulling from library/postgres
a8a0725a32a6: Pull complete
f7049a93e93f: Pull complete
5435b2dcdf5c: Pull complete
af2a6e5f285c: Pull complete
af0c3ea2fe5e: Pull complete
9328c154beae: Pull complete
c45b346571fc: Pull complete
00f33d580dc3: Pull complete
5ca81a59b0a4: Pull complete
8ecdbd190da6: Pull complete
71a32c7571f4: Pull complete
2c38b75ffca2: Pull complete
5c152148754e: Pull complete
b4765ac2e7e6: Pull complete
cebcb07837b: Download complete
b9015a4f6b87: Download complete
Digest: sha256:3e43515057e113ee741fca2f621f15300be34a6d4a8dfcf2e20651288c827
2f3
Status: Downloaded newer image for postgres:15
945d17514ef2c4efa1c4ec7bb5989d885f247abf745faebe466f058b19abdcfb
PS C:\Users\ASUS\Projects\my-web-server>
```

The web container can reach the database container at hostname db without knowing its IP address.[\[13\]](#) This is how multi-container applications are usually configured - all services are in their own container, and a common user-defined network gives service discovery. This pattern is formalized in Docker Compose, which defines multi-container applications in a single YAML file.

The host networking mode provides the container with direct access to the network stack of the host (which can be used in applications sensitive to performance), whereas the none mode enables no networking at all, appropriate to batch processing containers which do not need to interact with the external world in any way.
[\[13\]](#)

2.5 Publishing Images to DockerHub

DockerHub is Docker's public registry which is a repository where images can be stored, versioned, and shared. [\[9\]](#) To publish the web server image built above:

```
PS C:\Users\ASUS\Projects\my-web-server> docker tag my-server herlock2/my-server
PS C:\Users\ASUS\Projects\my-web-server>
PS C:\Users\ASUS\Projects\my-web-server> docker push herlock2/my-server
Using default tag: latest
The push refers to repository [docker.io/herlock2/my-server]
114433e5973e: Pushed
dd277a264acc: Pushed
de47083ed7d7: Pushed
ee2aff703969: Pushed
902ef905f973: Pushed
latest: digest: sha256:db7a489214499fd36df0cece22ddca5b3af9afdd4aadd9b2d9236dfb02add664 size: 856
```

The image can be pulled by anyone who has a DockerHub account or the repository is public. This allows team members to pull and run the same environment with just one command: [\[9\]](#)

```
PS C:\Users\ASUS\Projects\my-web-server> docker pull herlock2/my-server
Using default tag: latest
latest: Pulling from herlock2/my-server
Digest: sha256:db7a489214499fd36df0cece22ddca5b3af9afdd4aadd9b2d9236dfb02add664
Status: Image is up to date for herlock2/my-server:latest
docker.io/herlock2/my-server:latest
```

DockerHub supports automated builds triggered by code commits, image vulnerability scanning, and private repositories for sensitive workloads. [\[9\]](#) In the enterprise, workloads Teams may use a private registry, like Azure Container Registry or AWS ECR, instead of DockerHub, to store images in their own infrastructure and address data governance needs.

Video demonstration: There is a video walkthrough showing the entire process, writing the Dockerfile, building the image, running the container, creating volumes, configuring a network, and pushing to DockerHub, which can be found at:

3. Cloud Computing Concepts: Load Balancing, Scalability, and Application Gateways

3.1 Load Balancing in Cloud Platforms

Load balancing is a process whereby the network traffic is distributed to a group of back-end resources in a way that does not cause a bottleneck by any specific instance. [\[14\]](#) Each of the major cloud providers, such as Microsoft Azure, Amazon Web Services (AWS) and Google Cloud Platform (GCP) provide a selection of load balancing services that are all optimized to suit a range of application architectures and traffic patterns. [\[15\]](#)

A Layer 4 load balancer works on the transport layer (TCP and UDP) and forwards the traffic according to the IP address and port number without having to examine the content of the request. Examples are Azure Load Balancer (Standard tier), AWS Network Load Balancer and pass-through Network Load Balancer of GCP. [14] Each of them automatically removes unhealthy backends with health probes and allocates traffic based on a five-tuple hash (source IP, source port, destination IP, destination port, protocol) and can be configured to run on availability zones to implement zone-redundant deployments to provide high availability across availability zones. [15]

The offerings of the cloud providers are constantly being improved; in particular, in September 2025, Microsoft decommissioned the Basic Load Balancer and unified around the Standard tier. [14] In the case of global (multi-region) traffic management, each cloud provider offers specific solutions: Azure Front Door and Traffic Manager, AWS Route 53 and CloudFront, or GCP Cloud CDN and Cloud Armor all of which provide DNS-based or anycast routing to redirect users to the nearest or healthiest regional endpoint, and have in-built WAF. [15]

3.2 Scalability in Cloud Platforms

Cloud systems offer managed auto-scaling groups, a collection of identically configured VMs that will automatically scale up or down according to demand. Azure Virtual Machine Scale Sets (VMSS), AWS Auto Scaling Groups (ASG), and GCP Managed Instance Groups (MIG) are all based on this model [16], and are automatically registered in the load balancer backend pool. The rules of the scale in these platforms may be pegged on CPU, memory, HTTP queue depth, custom metrics or a schedule. [17]

Managed Kubernetes services: Azure Kubernetes Service (AKS), Amazon EKS, and Google GKE all have pod-level scaling with the Horizontal Pod Autoscaler (HPA) and node-level scaling with the Cluster Autoscaler, and are supported by containerized workloads. [18] Likewise, PaaS solutions like Azure App Service, AWS Elastic Beanstalk, and GCP App Engine all feature automatic horizontal scaling depending on the rate of incoming requests, and do not need to be managed as a cluster.

The main difference between horizontal scaling (adding more instances - also known as scaling out) and vertical scaling (increasing the resources of an existing instance - scaling up) is significant in cloud architecture, [18] as it spreads load and failure risk across several independent instances, and is only limited by the capacity of the cloud provider and not by the physical constraints of a single machine.

3.3 Layer 7 Application Load Balancing

Application layer Layer 7 load balancers are implemented at the application layer, and they examine the contents of HTTP/HTTPS requests to make smart routing choices based on URL paths, host names, query strings, and other application-layer attributes. Initial implementations are Azure Application Gateway, AWS Application Load Balancer (ALB) and GCP Cloud Load Balancing. [16] This is essentially more competent than Layer 4 load balancing. These services have a similar set of capabilities:

URL-based routing: /api/ can be routed to one group of backends (a group of API servers) and /static/ to another group (a CDN or a server with static files) and finally each part of the application can scale and be deployed individually [16]

SSL termination: SSL termination: Layer 7 load balancers are able to decrypt inbound HTTPS traffic and redirect plain HTTP to back-end servers (optionally re-encrypting prior to the back-end). This removes cryptographic computation at the instance of the application and concentrates on certificate management. All of the mentioned Azure Application Gateway, AWS ALB, and GCP HTTPS Load Balancer support termination of SSL/TLS. [16]

the average CPU is less than 30% in 15 minutes, delete 1 instance. The asymmetry - scaling out aggressively, scaling in conservatively - is intentional: it avoids thrashing (rapid scale-out and scale-in events) when load varies. [\[19\]](#)

Health probes are also vital but often overlooked. [\[14\]](#) Cloud load balancers send periodic health probes — typically HTTP GET requests to a designated endpoint or TCP connection attempts — to each backend instance. In case an instance does not respond to a fixed number of consecutive probes, it is declared unhealthy and eliminated in the rotation. It is not sent new requests until it comes back and again passes health checks. This is automated health management; in this case, the failure of applications or deployment failures does not result in any apparent downtime to the end user provided that one healthy instance remains. [\[14\]](#)

3.5 Demonstrating the Combined Benefits

Consider an e-commerce application with a product catalogue API (containerized in Kubernetes), a payment gateway (on dedicated VMs), and a static front-end (served by a CDN and object storage). During a flash sale, traffic may spike 10–20x within minutes. [\[1\]](#)

An L7 load balancer (Azure Application Gateway, AWS ALB, or GCP HTTPS Load Balancer) sits at the entry point and routes `/api/*` to the container cluster and `/static/*` to the CDN origin. Its WAF blocks bot traffic and injection attacks, which typically surge during high-profile sale events. The load balancer itself auto-scales based on incoming connection rates [\[16\]](#)

The Horizontal Pod Autoscaler detects rising request latency per pod and deploys additional API pods within seconds. The Cluster Autoscaler adds nodes to the cluster when pod demand exceeds current node capacity. [\[18\]](#) An L4 load balancer distributes TCP traffic across the cluster nodes. When the sale ends, scale-in removes surplus nodes within minutes, and the customer pays only for what was consumed. [\[14\]](#)

This pattern — an L7 gateway for intelligent routing, WAF, and SSL; an L4 load balancer for fast transport-layer distribution and health monitoring; and a managed auto-scaling group for capacity elasticity — is cloud-provider agnostic and applies equally whether deployed on Azure, AWS, or GCP. [\[17\]\[19\]](#)

4. Forces of Cloud Growth, and the Emergence of Serverless Computing

4.1 The Main Driving Forces Behind Cloud Adoption

The combination of economic, operational and technical pressures has been the driving force behind the growth of cloud computing as a niche technology to the default infrastructure model of most organizations.

Cost transformation: the reduction of capital expenditure (CapEx) to operational expenditure (OpEx) is, perhaps, the most potent driver. The purchase and maintenance of physical servers are expensive upfront and tie up capacity to estimated demand and incurs constant costs whether used or not [\[10\]](#) whereas cloud computing turns this into a pay-as-you-use model in which organizations only pay to use the resources needed at a particular scale, at the granularity of seconds or milliseconds in serverless platforms. [\[20\]](#) This has been transformative for startups and small organizations that could previously never afford the infrastructure needed to scale globally.

Elasticity and flexibility: the traditional data centers are dimensioned to peak capacity and thus inefficient at average load. Scaling can be done quickly and automatically with cloud platforms to accommodate surges in

traffic and de-scale afterwards, [18] and this elasticity is what made cloud infrastructure essential during unpredictable demand events, such as the pandemic-induced remote working surge and major sporting events that have generated millions of simultaneous streams.

Speed of deployment: it can take days or weeks to provision a physical server. It is possible to provision cloud resources with an API call in minutes and deploy containerized applications in seconds, [3] which has become a competitive necessity. Container and cloud API-based Continuous Integration and Continuous Deployment (CI/CD) pipelines enable teams to release numerous software changes daily instead of quarterly.

Global reach: cloud vendors have data centers in dozens of locations around the globe. By deploying an application to a new geography, it is no longer necessary to have a physical presence in that geography [17] it is a configuration change that can be completed in a matter of minutes, and it allows organizations to deliver low latency services to users anywhere on the planet, with content delivery networks and edge computing.

Managed services and reduced operational burden: Cloud providers offload operational tasks such as hardware failure management, physical security, network and to a growing extent, operational management of software platforms cloud providers manage hardware failure, physical security, network infrastructure and increasingly the operational management of software platforms databases, message queues, identity services to enable engineering teams to concentrate on application logic instead of on infrastructure plumbing. [20]

4.2 Serverless Functions: What They Are and How They Work

Serverless computing and Function as a Service (FaaS) is another layer of abstraction over containers. Under a FaaS model, developers create single functions, or code segments that are implemented in response to a particular event, and send them to a cloud provider, [21] who takes care of the infrastructure, including provisioning, scaling, availability and decommissioning. Containers and servers are never accessed by developers.

In 2014, AWS Lambda was introduced, creating the FaaS category and is the market leader. [22] Azure Functions, the equivalent provided by Microsoft, integrates with the rest of the Azure ecosystem - triggers can invoke functions in response to HTTP requests, changes in Azure Blob Storage, messages on a Service Bus queue, database updates in Cosmos DB, and dozens of other event sources. [23]

The implementation model is dynamic and transient. The platform spins up an isolated execution environment (itself based on a containerized runtime) when a trigger event is fired, executes the function, and returns the result, and then either keeps the environment warm or recycles it. [21] This warm vs cold difference has a practical impact: a cold start, in which no warm environment exists and a new one needs to be initialized, adds tens to hundreds of milliseconds of additional latency based on the runtime and function size. [24]

Billing is based on a very fine granularity: AWS Lambda is charged on a per-million request basis and per GB-second of execution (memory allocated × duration in seconds). [22] With infrequently-used functions or functions with a short execution time, this is dramatically less expensive than maintaining a continuously-provisioned server or container. AWS offers 1 million free Lambda executions and 400,000 GB-s of compute time free every month. [22]

The worldwide serverless computing market is estimated to be USD 6.86 billion in 2025, and is anticipated to be USD 92.22 billion in 2034, increasing by a CAGR of 14.15%. [2] This growth is indicative of the growing role of serverless not only in simple event processing but also in AI inference pipelines, real-time analytics, mobile backends, and IoT data processing. [2]

4.3 Comparing Serverless Functions with Containerization

Both serverless functions and containers represent the same set of driving forces: cost efficiency, agility, and scalability, but they trade-off in different ways that will make them fit different situations. [25] Figure 4 presents a systematized comparison.

Dimension	Containers (Docker/K8s)	Serverless (FaaS)	Better for...
Execution model	Long-running processes	Event-driven, ephemeral	Containers
Infrastructure	Developer-managed	Fully managed by provider	Serverless
Startup time	Seconds (image pull)	Milliseconds (cold start risk)	Serverless
Scalability	Manual or Kubernetes	Automatic, near-infinite	Serverless
Cost model	Pay for reserved capacity	Pay-per-execution only	Serverless
State	Stateful possible	Stateless by default	Containers
Portability	High — any cloud/on-prem	Tied to cloud provider	Containers
Use case fit	Microservices, ML, DBs	APIs, triggers, IoT, ETL	Depends
Control level	Full control	Abstracted — less control	Containers
Long-running tasks	Excellent	Limited (max 15 min)	Containers

Figure 4: Serverless (FaaS) vs Containers — comparison across ten dimensions including execution model, cost, scalability, portability, and use case fit.

4.3.1 Execution Model and Infrastructure

Containers are long-lived processes. A container run in Docker is launched and runs until it is stopped, handling an unlimited number of requests. This makes containers highly suitable to continuous, stateful, or long-running workloads - web servers, databases, message consumers. [25]

Serverless functions, in contrast, are short lived. A function invocation is a separate entity, and the function is supposed to finish within a specified time constraint (15 minutes in AWS Lambda, 10 minutes in the Azure Functions Consumption plan). [21] This is not suitable to workloads that need a long-running computation or sustained connection, but is well suited to discrete event-driven workloads.

4.3.2 Scalability and Cost

This is where serverless functions can best benefit some workloads. [20] Containers can scale with orchestration tools such as Kubernetes which is powerful but needs configuration, monitoring, and operational expertise. Serverless functions automatically scale to thousands of concurrent instances in seconds, and are controlled by the provider without any configuration needed. [21]

Containers are structurally different as well in terms of costs. Containers are also costly to run idle. Serverless functions are free when not used. [22] When workload bursts are event-driven, and there are zero-load periods, serverless can be many times cheaper. Containers can be more economical with the per-invocation pricing of FaaS scaling to larger volumes, since high-volume, continuously loaded workloads often require large numbers of invocations. [24]

4.3.3 Portability

Containers have the property that they are portables. Local Docker images will work on any cloud provider, any on-premises Kubernetes cluster, or any Docker-compatible runtime without changes.[8]

Serverless functions do not have this guarantee of portability at all. AWS Lambda functions have AWS-specific trigger integrations (S3 events, DynamoDB streams, API Gateway), and Azure Functions have Azure-specific

bindings (Blob Storage, Service Bus, Event Grid). Although the function code can be portable in nature, event integrations and deployment infrastructure are closely tied to the provider. Moving an application based on serverless computing off of AWS to Azure is not a configuration change, but rather a major re-engineering effort. [\[25\]\[23\]](#)

4.3.4 Developer Experience and Operational Overhead

Serverless saves development teams a lot of overhead by not needing an OS to maintain, a cluster to operate, or a load balancer to set up. A function is written by developers, deployed and it executes. It is one of the main motivating factors that have led to the adoption of FaaS it enables small teams to create and run complex event-driven systems without infrastructure engineering. [\[20\]](#)

Containerized workloads at scale with Kubernetes demand knowledge of networking, storage, RBAC, image security, and cluster lifecycle management. It is not a frivolous investment, but the expanding ecosystem of managed Kubernetes services (AKS, EKS, GKE) has made it much easier. [\[5\]](#)

4.3.5 Identifying the Driving Forces

Both serverless functions and containers address the same underlying forces of cloud driving forces, albeit in different directions. Cost efficiency is served by both: AKS simplifies infrastructure use in containers, and serverless simplifies it to zero. [\[25\]\[10\]\[3\]](#)

A growing industry consensus is that the two models are not mutually exclusive, but complementary and a typical architecture pattern is to deploy stateless, event-driven functions on a serverless platform (processing asynchronous events, asynchronous notifications or data transformations) and to run stateful, continuously loaded services in containers (APIs, databases, background workers). [\[25\]\[24\]\[21\]](#)

4.3.6 Cold Start Mitigation and Production Patterns

One of the most commonly mentioned operational issues with FaaS platforms is cold start latency: Once an instance of a FaaS server has been idle long enough that the server can recycle its execution environment, subsequent invocations have to wait until a new execution environment is created, the runtime is started, and the function package is loaded. For Java or .NET runtimes with heavy initialization, this may introduce 500ms2000ms of latency to a fast function otherwise. [\[24\]\[21\]](#)

Both AWS and Azure have added features to avoid this. AWS Lambda Provisioned Concurrency maintains a fixed number of execution environments at all times, preventing cold starts on those instances at the expense of a provisioned-capacity billing model (which somewhat cancels the cost advantage of serverless on those parallel slots). Azure Functions Premium plan maintains at least one warm instance at all times and supports VNET [\[24\]\[22\]](#) integration for private network access features not available on the free Consumption plan.[\[23\]](#)

A hybrid architectural pattern is becoming more popular in production, particularly core application services, which are APIs which execute synchronous user request where latency is a concern, and which are implemented in containers (Kubernetes or Azure Container Apps) and enjoy warm, predictable execution environments. Background tasks that are event-driven, such as image processing triggered by file uploads, email notifications triggered by a user action, scheduled data export jobs, and others, are implemented as serverless functions with cold start latency acceptable and the pay-per-invocation model cost-effective in those cases. [\[25\]\[24\]\[21\]](#)

5. Conclusion

Three dimensions of modern cloud engineering, which are interrelated, have been discussed in this report. First, hardware virtualization has been transformed into containerization which has made it possible to achieve greater density, portability, and efficiency by sharing operating system kernels. Docker has also made this process easier with an easy and potent toolchain. The real-world examples (construction of a web server image, establishing persistent storage volumes, setting up Docker networks, and publishing images to Docker Hub) illustrate the entire container lifecycle in action.

Second, the Layer 7 application gateways, Layer 4 load balancers, and auto-scaling groups show how cloud-native infrastructure can be resilient, scaled, and elastic. Even though these ideas are demonstrated with help of Azure, they apply to other big cloud providers like AWS and GCP as well, where every single component enhances the others.

Third, the discussion of the similarities and differences in serverless functions and containerization points out the ways both of these models handle significant cloud computing requirements including elasticity, cost-efficiency, speed of deployment, and managed services and including various trade-offs. It is the best practice that would be dependent on the workload needs, maturity of operations, and other factors like platform dependency.

With the constant change of cloud infrastructure due to the introduction of new technologies like serverless containers, edge computing, and AI-enhanced deployment pipelines, the fundamental principles of containerization, wide-scale distribution, and event-based execution will still be crucial skills of cloud engineers in the future.

References

- [1] Virtualization.info, *State of Docker and the Container Industry in 2025*. Feb. 2025. [Online]. Available: <https://virtualization.info/2025/02/23/state-of-docker-and-the-container-industry-in-2025/>
- [2] Precedence Research, *Serverless Computing Market Size to Hit USD 92.22 Billion by 2034*. Nov. 2025. [Online]. Available: <https://www.precedenceresearch.com/serverless-computing-market>
- [3] OpenSourceForU, *Docker or Virtual Machines: What's the Right Choice?* Nov. 2025. [Online]. Available: <https://www.opensourceforu.com/2025/11/docker-or-virtual-machines-whats-the-right-choice/>
- [4] Aqasizade A. et al., *Experimental assessment of containers running on top of virtual machines*. *IET Networks*, 2024. [Online]. Available: <https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/ntw2.12138>
- [5] Veeam, *Virtualization vs Containerization: Key Differences*. Oct. 2025. [Online]. Available: <https://www.veeam.com/blog/virtualization-vs-containerization.html>
- [6] World Journal of Advanced Engineering Technology and Sciences, *Containers vs. Virtual machines: Understanding the shift to containerised cloud infrastructure*, 2025. [Online]. Available: https://journalwjaets.com/sites/default/files/fulltext_pdf/WJAETS-2025-0305.pdf
- [7] Bernstein D., *Containers and Cloud: From LXC to Docker to Kubernetes*. *IEEE Cloud Computing*, vol. 1, no. 3, pp.81–84, 2014. [Online]. Available: <https://ieeexplore.ieee.org/document/7036275>
- [8] Scale Computing, *Virtualization vs Containerization — What's the Difference?* Jan. 2025. [Online]. Available: <https://www.scalecomputing.com/resources/whats-the-difference-between-virtualization-vs-containerization>
- [9] Docker Inc., *Docker Overview — What is Docker?*. *Docker Documentation*, 2025. [Online]. Available: <https://docs.docker.com/get-started/docker-overview/>
- [10] Bitcot, *What is Docker? Key Benefits for Cloud Computing in 2025*. May 2025. [Online]. Available: <https://www.bitcot.com/docker-in-cloud-computing/>
- [11] Docker Inc., *Dockerfile Reference*. *Docker Documentation*, 2025. [Online]. Available: <https://docs.docker.com/reference/dockerfile/>
- [12] Docker Inc., *Manage Data in Docker — Volumes*. *Docker Documentation*, 2025. [Online]. Available: <https://docs.docker.com/engine/storage/volumes/>
- [13] Docker Inc., *Networking Overview — Docker*. *Docker Documentation*, 2025. [Online]. Available: <https://docs.docker.com/engine/network/>
- [14] Microsoft, *What is Azure Load Balancer?* *Microsoft Learn*, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview>
- [15] Loadbalancer.org, *What's the best cloud load balancer? AWS, Azure, GCP comparison*. Jan. 2026. [Online]. Available: <https://www.loadbalancer.org/blog/whats-the-best-cloud-load-balancer/>
- [16] Microsoft, *Azure Application Gateway — Load-Balancing Solution*. *Microsoft Azure*, 2025. [Online]. Available: <https://azure.microsoft.com/en-us/products/application-gateway>
- [17] Microsoft, *Load Balancing Options — Azure Architecture Center*. *Microsoft Learn*, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/load-balancing-overview>
- [18] FreeCodeCamp, *Load Balancing with Azure Application Gateway and Azure Load Balancer*. May 2025. [Online]. Available: <https://www.freecodecamp.org/news/load-balancing-with-azure-application-gateway-and-azure-load-balancer/>
- [19] Narayana S., *A Deep Dive Into Azure Load Balancing Options*. *DZone*, Jan. 2024. [Online]. Available: <https://dzone.com/articles/mastering-scalability-and-performance-a-deep-dive>
- [20] CloudPlexo, *Serverless Compute Comparison: AWS Lambda vs Azure Functions vs Google Cloud Functions*. Apr. 2024. [Online]. Available: <https://cloudplexo.com/blog/serverless-compute-comparison-aws-lambda-vs-azure-functions-vs-google-cloud-functions/>
- [21] Pluralsight, *Serverless Showdown: AWS Lambda vs Azure Functions vs Google Cloud Functions*. *Pluralsight*, 2025. [Online]. Available: <https://www.pluralsight.com/resources/blog/cloud/serverless-showdown-aws-lambda-vs-azure-functions-vs-google-cloud-functions>
- [22] AWS, *AWS Lambda — Serverless Compute*. *Amazon Web Services*, 2025. [Online]. Available: <https://aws.amazon.com/lambda/>
- [23] Microsoft, *Azure Functions — Serverless Compute*. *Microsoft Azure*, 2025. [Online]. Available: <https://azure.microsoft.com/en-us/products/functions>
- [24] DevelopersVoice, *Beyond the Cold Start: A 2025 Architect's Showdown of Azure Functions vs AWS Lambda*. Aug. 2025. [Online]. Available: <https://developersvoice.com/blog/cloud/azure-functions-vs-aws-lambda/>
- [25] DEV Community, *Serverless vs Containers: What's Winning in 2026?* Nov. 2025. [Online]. Available: <https://dev.to/ripenapps-technologies/serverless-vs-containers-whats-winning-in-2026-556e>

- [26] Amazon Web Services, *Application Load Balancer — Elastic Load Balancing*. AWS Documentation, 2025. [Online]. Available: <https://aws.amazon.com/elasticloadbalancing/application-load-balancer/>
- [27] Google Cloud, *Cloud Load Balancing Overview*. Google Cloud Documentation, 2025. [Online]. Available: <https://docs.cloud.google.com/load-balancing/docs/load-balancing-overview>

Docker Video Link: <https://www.youtube.com/watch?v=wOTIUFmPFq4>
