



CLOUD COMPUTING AND CONTAINERISATION

Docker, Virtualisation, Cloud Concepts, and Serverless Functions

Student Name: Prem Raj Janga

Student ID: 2505006

Module Name: Cloud Engineering Principles and Practice

Module Code: ACCA7027

Programme: Computing Portfolio

Table of Contents

Table of Contents.....	2
Introduction.....	4
1. Cloud Technology: Docker and Containerisation.....	4
1.1 Principles of Virtualisation	4
1.2 Principles and Benefits of Containerisation.....	6
1.2.1 Key Benefits of Containerisation.....	6
1.2.2 Constructing a Dockerfile to Build an Image	7
1.3 Images and Publishing Images.....	8
1.3.1 Docker Image Layering	8
1.3.2 Pushing an Image to Docker Hub	9
1.4 Container Volumes	11
1.4.1 The Need for Persistent Storage	11
1.4.2 Creating and Using a Docker Volume	11
1.5 Container Networking.....	12
1.5.1 Docker Networking Fundamentals	12
1.5.2 Creating and Using a Docker Network.....	12
1.6 End-to-End Practical Demonstration	13
2. Cloud Computing Concepts.....	15
2.1 Load Balancing	15
2.1.1 Concept Overview	15
2.1.2 Azure's Approach: Azure Load Balancer and Azure Application Gateway.....	15
2.2 Scalability	16
2.2.1 Concept Overview	16
2.2.2 Azure's Approach: Azure Virtual Machine Scale Sets and Azure Autoscale.....	16
2.3 Application Gateways.....	17
2.3.1 Concept Overview	17
2.3.2 Azure's Approach: Azure Application Gateway	17
2.4 Why Load Balancing and Scaling are Complementary	18
2.5 Combined Benefits: Load Balancing, Scaling, and Application Gateways.....	19
3. Driving Forces Behind Cloud Computing Expansion	20
3.1 Cost Efficiency.....	20
3.2 Flexibility and Agility.....	20
3.3 Scalability and Elasticity.....	21
3.4 Reliability and Disaster Recovery	21
3.5 Security and Compliance	22

3.6 Serverless Functions vs Containerisation: Comparative Analysis.....	23
4. Conclusion	25
References.....	26

[VIDEO LINK: <https://youtu.be/26eEO8fOjYY>]

Introduction

Cloud Computing has benefited from the introduction of container technologies, which are a game-changer in terms of how to deploy Applications efficiently and Portably while ensuring that they have the potential to scale when needed [2]. The purpose of this paper is to provide an overview of Virtualization and Containerization Concepts, particularly about Docker, as well as Provide Examples of their use in a Classroom Environment. In addition, using a typical web Application scenario, key Cloud concepts such as Load Balancing, scalability, and Application Gateways are examined, and a variety of factors that drive Cloud adoption, including Serverless Computing such as AWS Lambda and Azure Functions, are considered in the analysis, and the technologies are critically evaluated [8]. While traditional virtualization abstracts Hardware resources, containerization offers a leaner Alternative, operating at the operating system level [9]. This transformation accelerates the deployment cycle in Cloud environments and improves resource utilization. The report also discusses the interplay of Load Balancing and Autoscaling and compares Serverless Functions and container solutions in terms of cost and flexibility. As shown in the following sections, these technologies comprehensively address key challenges of modern Cloud engineering, such as portability, efficiency, and low operational overhead [3]. The entire process is demonstrated using a real-world web server deployment. The Discussion will Center Around the types of real-world applications that can be built utilizing services provided by AWS, Azure, and other Cloud Computing Service Providers.

1. Cloud Technology: Docker and Containerisation

1.1 Principles of Virtualisation

Virtualization is one of the key pillars of Cloud Computing today [8]. Essentially, virtualization is the technology that allows physical resources (hardware, storage, and networking) to be represented virtually in Software. By abstracting physical infrastructure, many separate and unique environments can run at the same time on a single physical server, allowing for greater efficiency and lower operating costs [2]. The Hypervisor, or Virtual Machine Monitor (VMM), is at the center of the virtualization process, enabling the creation, management, and access to Virtual Machines (VMs) and the physical server(s).

There are primarily two kinds of hypervisors:

- **Type 1 (Bare-Metal) Hypervisors:** These hypervisors run directly on the physical hardware of the Host and therefore offer users the highest level of performance. Examples include VMware ESXi, Microsoft Hyper-V, and KVM.
- **Type 2 (Hosted) Hypervisors:** These hypervisors run as a program within a traditional operating system and are typically slower than Bare Metal hypervisors. Examples include VMware Workstation and Oracle VirtualBox.

A virtual machine consists of a complete operating system (OS), applications, and libraries. Each VM is isolated from the others; therefore, the virtual environment has great security. However, VMs require a lot of resources because the OS components are duplicated in every instance [2]. For example, users of the IaaS (Infrastructure as a Service) model, use Infrastructure Providers such as Amazon EC2 and Microsoft Azure (and their associated service offerings) to provision VMs from a common physical host, while continuing to maintain complete management over the VM's OS and benefit from resource pooling. These limitations have resulted in the use of Containers for virtualization; in the Container Model, the isolation occurs at the OS-level vs. the hardware-level, therefore minimizing overhead associated with this resource-intensive solution. As illustrated in Figure 1.1, the architectural difference between a virtual machine and a container is clearly evident: VMs each carry a full guest OS stack, whereas containers share the host OS kernel, resulting in significantly reduced resource overhead [9].

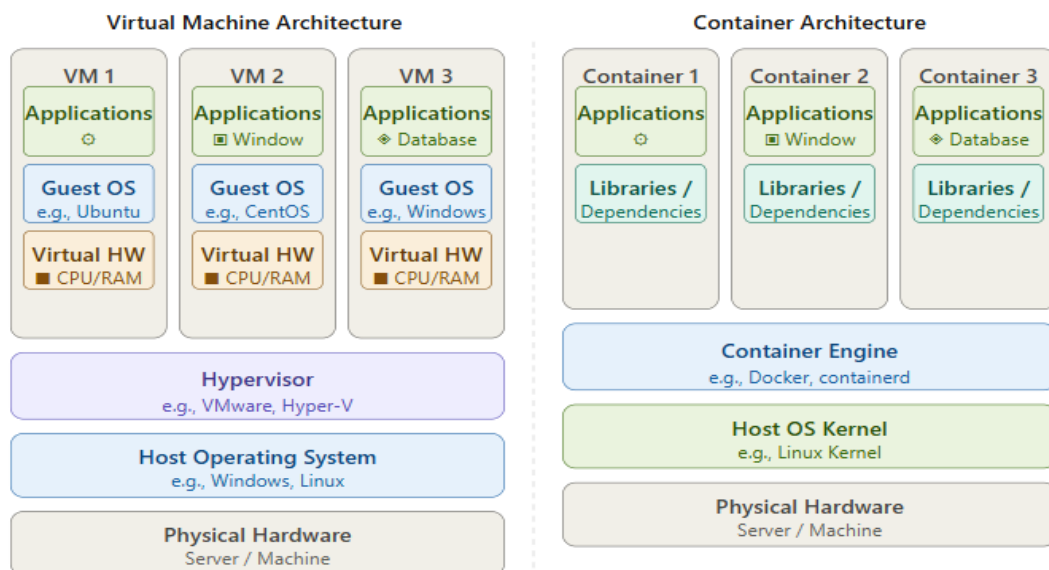


Figure 1.1: Comparison of Virtual Machine architecture (left) and Container architecture (right), illustrating how containers share the host OS kernel.

1.2 Principles and Benefits of Containerisation

Containerization is a resource-saving virtualization technology at the operating system level that bundles applications and their dependencies in self-contained units, so-called "containers", in contrast to traditional virtual machines. Containers do not contain a complete operating system, but share the Host Kernel and achieve at the same time an isolated user environment through features such as Linux Kernel Namespaces and Cgroups (Docker Inc, 2024) Docker has become the De facto Standard in the field of containerization technology Docker was announced in 2013 by Solomon Hykes of dotCloud and has significantly promoted the spread of Container technology through a standardized Image Format, a Build Toolchain and an execution environment (Merkel, 2014) The main components of Docker are:

- Docker Engine: The heart of Docker, responsible for creating and running containers.
- Docker Image: An immutable, hierarchical filesystem Snapshot, similar to the "blueprint" of a container
- Docker Container: the execution instance that actually runs the image
- Dockerfile: a declarative text file that describes the steps to create a Docker image
- Docker Hub: a cloud-based Registry for storing and distributing Docker images
- Docker Compose: an orchestration tool that defines and runs multiple Container applications using a YAML file

1.2.1 Key Benefits of Containerisation

Compared to traditional virtualization technologies, containerization offers the following advantages:

- Flexibility for use anywhere: Containers encapsulate all dependencies, ensuring that applications work identically in development, test, and production environments. The classic Problem "it works on my computer, but not in production" is a thing of the past
- Lean and responsive: Containers start in milliseconds and require significantly fewer resources than virtual machines, as each instance does not require the entire operating system.
- Scalable and adaptable: Orchestration platforms such as Kubernetes allow horizontal scaling through the creation of new Container instances on demand (Burns et al, 2022).
- Unshakable stability: Images are immutable. Updates are made by creating new Image versions, which enables reliable Rollback and version control
- Seamless Integration: Containers integrate seamlessly into CI/CD Pipelines, enabling automated testing and deployments

- Compact and efficient: A single physical Server can host significantly more Container instances than virtual machines, reducing infrastructure costs

1.2.2 Constructing a Dockerfile to Build an Image

A Dockerfile is like a recipe. It is a text file that describes the steps to create a Docker Image [5]. Each statement creates a new layer within the Image, and Docker caches those layers to speed up subsequent Builds [9]. The following Dockerfile, created as part of a tutorial, is used to serve a static HTML web page with an nginx web server. As shown in Figure 1.2, the Dockerfile specifies the base image, copies the HTML file into the container, and exposes the necessary port.

```
FROM nginx:latest
WORKDIR /usr/share/nginx/html
RUN rm -rf ./*
COPY index.html .
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Figure 1.2: Dockerfile for a custom Nginx web server serving a static HTML page.

The associated file `index.html` contains a custom HTML page created during the course. To create the Image, use the following command: The `-t` Option assigns a name and Tag to the Image, and the dot (`.`) specifies the working directory for the Build.

```
root@ip-172-31-5-170:~/my-docker-project# docker build -t my-webserver:1.0 .
[+] Building 6.9s (9/9) FINISHED
```

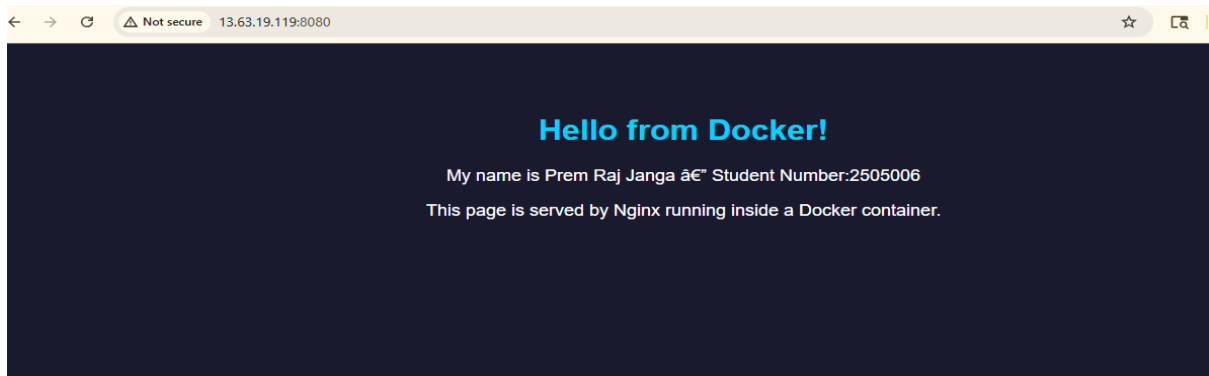
Once the Image is complete, the container can be started using the `docker run` command. This configuration assigns Port 8080 of the container to serve the HTML page. As demonstrated in

Figure 1.3, the Docker run command launches the Nginx Container with port forwarding (-p 8080:80), mapping the host machine's port 8080 to the container's internal port 80 [5].

```
root@ip-172-31-5-170:~/my-docker-project# docker run -d -p 8080:80 --name my-container my-webserver:1.0
2ca2f810e0ff25b7109b1977968d0aec8262794c6c15fa1d6843db46eb1b590a
```

Figure 1.3: Docker run command launching the Nginx container with port mapping -p 8080:80.

The -d Parameter starts the Container in the background, and the -p Parameter connects the Host Port 8080 to the Container Port 8080. When you call `http://localhost:8080` in your Browser, your custom HTML page is displayed. This completes the Build-to-Run Workflow.



1.3 Images and Publishing Images

1.3.1 Docker Image Layering

Docker Images are structured like transparent slides stacked on top of each other. Each slide is read-only and stores a specific system state. When creating a container from an Image, Docker adds a thin, writable layer to it. Any changes that occur at runtime are logged in this layer. This layered architecture offers two major advantages: slides that are common to different Images are saved only once on disk, which saves space, and unchanged processes are automatically reused when the Image is created, avoiding unnecessary reprocessing.

To view the hierarchical structure of the Image, you can use the following command:

```

root@ip-172-31-5-170:~/my-docker-project# # List all locally stored images
docker images

IMAGE                ID                DISK USAGE    CONTENT SIZE    EXTRA
my-webserver:1.0     0f3e7c8d58e2     237MB         63MB            U
root@ip-172-31-5-170:~/my-docker-project# # View detailed metadata about an image
docker inspect my-webserver:1.0

```

```

root@ip-172-31-5-170:~/my-docker-project# # View the layer history of an image
docker history my-webserver:1.0
IMAGE                CREATED          CREATED BY          SIZE              COMMENT
0f3e7c8d58e2        About an hour ago  CMD ["nginx" "-g" "daemon off;"]  0B                buildkit.dockerfile.v0

```

1.3.2 Pushing an Image to Docker Hub

Docker Hub is Docker's official public Registry and hosts over 100,000 Container Images. Publishing your custom Image to Docker Hub makes it available globally, greatly simplifying Team collaboration and deployment to Cloud environments. Publishing is as follows:

Step 1-Authenticating to Docker Hub: Docker login:

```

root@ip-172-31-5-170:~/my-docker-project# docker login -u premrajjanga
Info → A Personal Access Token (PAT) can be used instead.
       To create a PAT, visit https://app.docker.com/settings
Password:

```

Step 2-Tag the Image with your Docker Hub username and Repository Name. The tag Format is:

```

root@ip-172-31-5-170:~/my-docker-project# docker tag my-webserver:1.0 premrajjanga/my-webserver:1.0

```

Step 3 – Push the tagged Image to Docker Hub. As illustrated in Figure 1.4, the complete three-step workflow (login, tag, and push) enables global distribution of the custom Docker image [5].

```
root@ip-172-31-5-170:~/my-docker-project# docker push premrajjanga/my-webserver:1.0
The push refers to repository [docker.io/premrajjanga/my-webserver]
5435b2dcdf5c: Pushed
```

Figure 1.4: Docker Hub publish workflow: login, tag, and push commands.

After publishing, the Image is available globally. Any user or Cloud service can use the image by referencing it from Docker Hub [5].

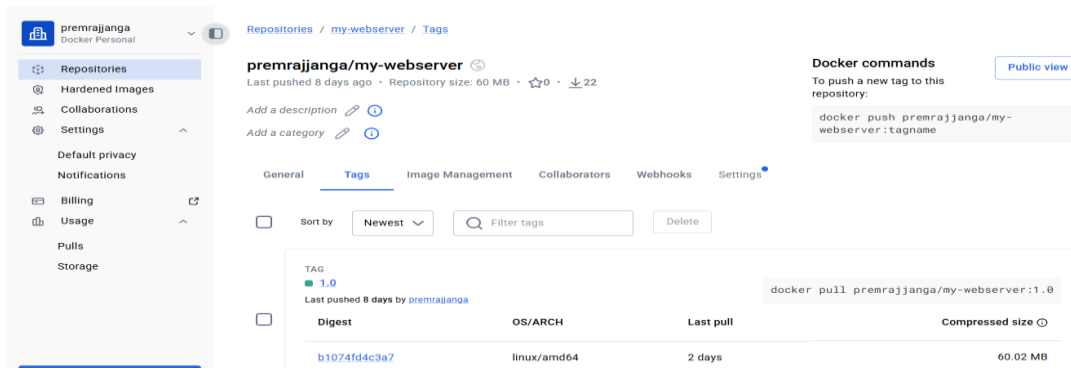
```
root@ip-172-31-5-170:~/my-docker-project# docker run -d -p 8080:80 premrajjanga/my-webserver:1.0
```

For example, Azure Container Instances (ACI) allow you to deploy a Docker Hub Image with just one Azure CLI command. This is a great example of how publishing enables seamless Cloud deployment. It includes a complete Pipeline from development to public distribution and forms the foundation of modern DevOps and CI/CD practices (Microsoft, 2024a).

The screenshot displays the Docker Hub interface for the repository `premrjajanga/my-webserver`. The left sidebar contains navigation links: **Repositories**, **Hardened Images**, **Collaborations**, **Settings**, **Default privacy**, **Notifications**, **Billing**, **Usage**, **Pulls**, and **Storage**. The main content area shows the repository details, including the last push time (7 days ago) and repository size (60 MB). A **Docker commands** section provides the command to push a new tag: `docker push premrajjanga/my-webserver:tagname`. Below this, a **Tags** section shows a table of image pulls.

Tag	OS	Type	Pulled	Pushed
1.0	linux	Image	less than 1 day	7 days

The interface also includes a **buildcloud** advertisement and a **Upgrade plan** button in the sidebar.



1.4 Container Volumes

1.4.1 The Need for Persistent Storage

By default, all data written to a container's filesystem is temporary; Their lifetime is limited to the Container. If the Container is deleted, all associated data is lost. This property, while useful for immutability and clean deployments, is a Problem for stateful applications that need to store data beyond the lifetime of the Container, such as databases. Docker meets this challenge with Volumes (Docker Inc, 2024).

1.4.2 Creating and Using a Docker Volume

Docker Volumes are used to persistently store data generated and used by Docker containers [5]. Because they are completely managed by the Docker Daemon and not the host system's filesystem, data portability and isolation are guaranteed. As shown in Figure 1.5, the following commands demonstrate how Volumes are created, mounted to a container, and verified for data persistence [5].

```

root@ip-172-31-5-170:~/my-docker-project# # Create a named volume
docker volume create my-persistent-volume

# List all volumes
docker volume ls

# Inspect volume metadata and mount point
docker volume inspect my-persistent-volume

# Mount the volume into a container at /data
docker run -d -v my-persistent-volume:/data --name data-container nginx:latest

# Write data into the volume from within the container
docker exec data-container sh -c "echo 'Persisted content' > /data/test.txt"

# Remove the container
docker rm -f data-container

# Start a new container with the same volume - data persists
docker run -it -v my-persistent-volume:/data busybox cat /data/test.txt

```

Figure 1.5: Docker volume creation, container mounting, and persistence verification.

As mentioned, the data stored in the first Container under /data remains accessible even if a second Container mounts the same Volume. This is particularly important for using databases such as MySQL and PostgreSQL in containers, as this preserves the data directory even after restarts or Upgrades of the containers. In addition, Volume drivers can be used to support network-attached Storage in Cloud environments such as Azure File Storage and Amazon EFS (Microsoft, 2024b).

1.5 Container Networking

1.5.1 Docker Networking Fundamentals

Container networks allow communication between containers, and between containers and external systems [5]. Docker offers several default network drivers:

- Bridge (default): Creates an independent virtual network on the Host Containers, and on the same Bridge, they can communicate using their Container names.
- Host: Unisolates the network so that containers can share the Host's network Stack.
- None: Disables all networking features and provides full Isolation
- Overlay: Creates a network between multiple Hosts for Docker Swarm and Kubernetes clusters.
- macvlan: Assigns MAC addresses to containers so that they appear as physical devices on the network.

1.5.2 Creating and Using a Docker Network

For communication between containers, creating a custom Bridge network is recommended [5]. This allows automatic DNS resolution based on the Container name. As demonstrated in

Figure 1.6, the following commands illustrate how a custom Docker network is created and how containers communicate with each other using DNS-based service discovery [5].

```
root@ip-172-31-5-170:~/my-docker-project# # Create a user-defined bridge network
docker network create my-app-network

# List all networks
docker network ls

# Inspect the network
docker network inspect my-app-network

# Run an Nginx web server on the custom network
docker run -d --name webserver --network my-app-network nginx:latest

# Run a BusyBox utility container on the same network
docker run -it --rm --network my-app-network busybox wget -O- http://webserver

# Connect an existing container to the network
docker network connect my-app-network existing-container

# Disconnect and remove the network
docker network disconnect my-app-network webserver
docker network rm my-app-network
```

Figure 1.6: Docker network creation and container communication using DNS-based service discovery.

In the above example, the busybox container can automatically resolve the name "webserver" to an IP address. This is because both containers belong to the same custom Bridge network. This DNS-based service discovery is a fundamental pattern in microservices architecture and allows communication between services without fixed IP addresses. In production environments, this concept is extended by CoreDNS and Kubernetes services, thus enabling the management of complex networks between Pods (Burns et al, 2022).

1.6 End-to-End Practical Demonstration

The following example illustrates the complete Workflow covered in this section. As shown in Figure 1.7, the integrated workflow encompasses image building from a Dockerfile, running a container with data volumes and a custom network, and publishing the Image to Docker Hub [5].

```
root@ip-172-31-5-170:~/my-docker-project# # Step 1: Build the custom Nginx image
docker build -t yourdockerhubusername/class-webserver:v1 .

# Step 2: Create persistent volume and custom network
docker volume create webserver-logs
docker network create webserver-net

# Step 3: Run the container with volume, network, and port mapping
docker run -d \
  --name class-webserver \
  --network webserver-net \
  -v webserver-logs:/var/log/nginx \
  -p 8080:80 \
  yourdockerhubusername/class-webserver:v1

# Step 4: Verify the web server is responding
curl http://localhost:8080

# Step 5: Push image to Docker Hub
docker login
docker push yourdockerhubusername/class-webserver:v1

# Step 6: Clean up
docker stop class-webserver && docker rm class-webserver
docker volume rm webserver-logs
docker network rm webserver-net
```

Figure 1.7: Complete integrated Docker workflow: build, volume, network, run, push, and cleanup.

This Demonstration was conducted as part of the course. It shows how the basic Docker components (Image, Volume, and network) work together to form a production-ready deployment unit. Nginx Logs are stored in a Volume, Containers communicate over an isolated network, and the Image is published to Docker Hub. This represents the entire lifecycle of a containerized application

A short video Demonstration with audio commentary that summarizes this Workflow (Image creation, Container execution, and publishing to Docker Hub) in about 15 minutes is available at the following Link:

[VIDEO LINK: <https://youtu.be/26eEO8fOjYY>]

2. Cloud Computing Concepts

This Discussion presents an overview of three key concepts in the Cloud Computing model: Load Balancing, Scalability, and Application Gateways. The focus is on Microsoft Azure.

A web-based e-commerce application is used throughout the Discussion as a working example of how these concepts relate to real-world applications.

2.1 Load Balancing

2.1.1 Concept Overview

In computing, Load Balancing is a mechanism used to spread the processing of incoming network traffic across a number of different backend servers dynamically. This helps increase the Availability, Responsiveness, and Fault Tolerance of an application. In Cloud Computing environments, Load Balancers are used extensively to route and distribute millions of requests per second across multiple dynamically provisioned infrastructures (Armbrust et al., 2010).

2.1.2 Azure's Approach: Azure Load Balancer and Azure Application Gateway

Azure offers two main types of Load Balancing services Azure Load Balancer operates on the fourth layer (transport layer) of the OSI model and distributes TCP and UDP traffic It supports both public configurations for the Internet and private configurations for internal networks an example is an E-Commerce Website A public Azure Load Balancer can distribute HTTP/S traffic to a Backend Pool of Azure VMs and container instances on which the application is running To permanently assign connections to Backend instances.

Azure uses a Hash algorithm with five elements (source IP, source port, destination IP, destination port, and protocol). For persistent Sessions (Sticky Sessions), the Hash can be reduced to two elements (source IP and destination IP). This keeps the Sessions on the same Server (Microsoft, 2024c).

Health checks are a key feature of Azure Load Balancer [12]. The Load Balancer regularly sends audit requests to the Backend instances and automatically removes instances that fail the health check from the Pool, creating a System in which the infrastructure can recover on its own without manual intervention. As depicted in Figure 2.1, the Azure Load Balancer distributes incoming traffic across a backend pool of virtual machines, with health probes ensuring that only healthy instances receive requests [12].

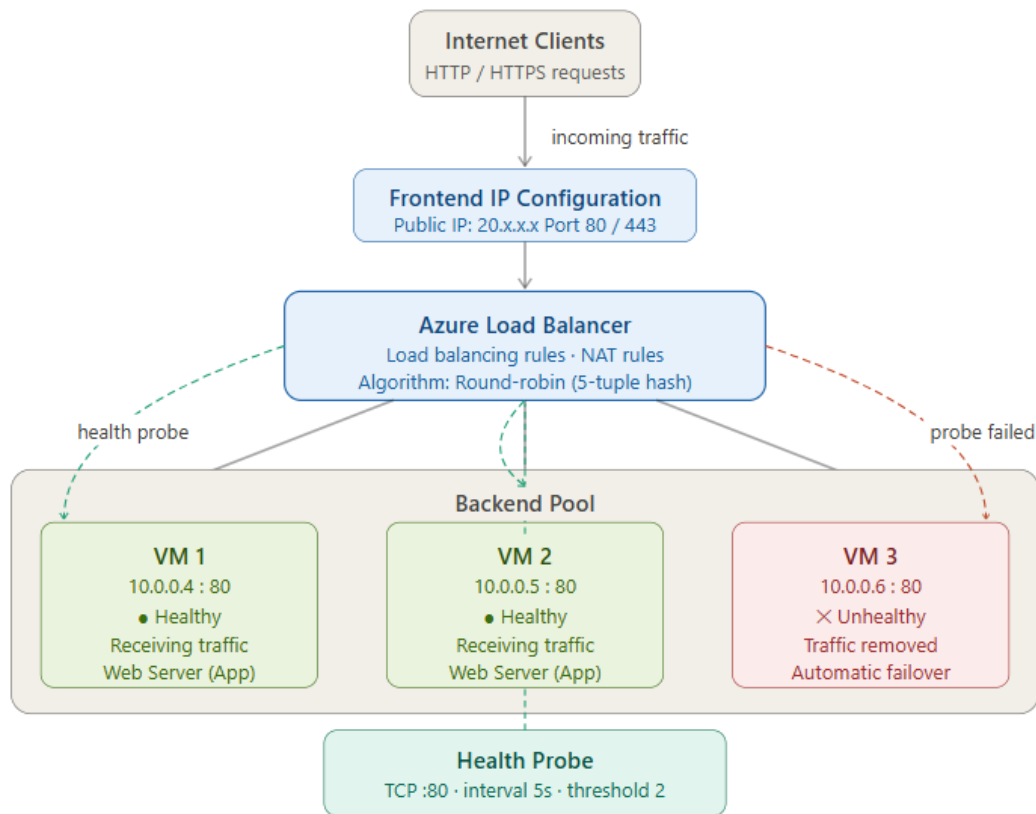


Figure 2.1: Azure Load Balancer distributing traffic across a backend pool with health probes for automatic failover.

2.2 Scalability

2.2.1 Concept Overview

Scalability refers to a System's ability to handle larger workloads by adding resources. This property, which Mel and Grance of the US National Institute of Standards and Technology (NIST) identified in 2011, is an essential feature of Cloud Computing. Cloud scaling can be roughly divided into two methods:

- **Vertical Scaling (Scale Up):** This method improves the performance of existing servers (by increasing CPU and memory). There are physical hardware constraints, and in some cases, the System must be shut down.
- **Horizontal Scaling (Scale Out):** This method distributes the load by increasing the number of servers. There is no theoretical upper limit, and expansion is possible without System disruption. Therefore, this method is preferred for cloud-native approaches.

2.2.2 Azure's Approach: Azure Virtual Machine Scale Sets and Azure Autoscale

Azure VM scaling groups and App Service auto-scaling enable automatic Horizontal scaling. Scaling rules can be configured based on metrics (such as scaling up if CPU utilization exceeds

70 % for more than 5 minutes) or schedules (such as scaling up at 8:00 AM and down at 10:00 PM to match business hours) For example, an Online store can reduce costs by automatically increasing the number of Container instances during peak times such as Black Friday and reducing off-peak times.

In addition, scaling with Azure Container Apps and Azure Kubernetes Service (AKS) extends to containerized Workloads. The Kubernetes Horizontal Pod Autoscaler (HPA) monitors custom metrics and automatically adjusts the number of replicas, while KEDA (Kubernetes Event-Driven Autoscaling) enables Event-Driven scaling based on message queue length (Microsoft, 2024d; Burns et al, 2022).

2.3 Application Gateways

2.3.1 Concept Overview

An Application Gateway is a Load Balancer that operates at layer 7 (the Application layer). It acts as an intelligent Traffic Controller and distributes Traffic based on HTTP/S characteristics such as URL paths, Host headers, and Cookies. While a typical Load Balancer distributes Traffic based solely on IP addresses and port numbers, an Application Gateway understands the "language" of applications. This enables more sophisticated and advanced Traffic control, including URL-based Routing, SSL/TLS termination, and Web Application Firewall (WAF) functionality.

2.3.2 Azure's Approach: Azure Application Gateway

Azure Application Gateway is a fully managed Layer 7 load-balancing service. In scenarios such as E-Commerce Websites, it enables intelligent Traffic distribution based on URL paths. Requests to/api/are routed to the API Backend servers, and requests to/static/are routed to the CDN-integrated storage Pool. This granular Routing ensures loose coupling of each Application Layer, thus enabling their independent scaling (Microsoft, 2024e).

In addition, SSL Offloading handles the complex SSL/TLS Handshake, thus reducing the Load on the Backend Pool before routing the Traffic [14]. Integrated Web Application Firewall (WAF) protects against OWASP Top 10 threats such as SQL Injection and Cross-Site Scripting. This is an essential feature for E-Commerce applications that process payment information [15]. As illustrated in Figure 2.2, the Azure Application Gateway provides a unified entry point with URL-based routing, SSL termination, and WAF protection for a multi-tier e-commerce architecture [14].

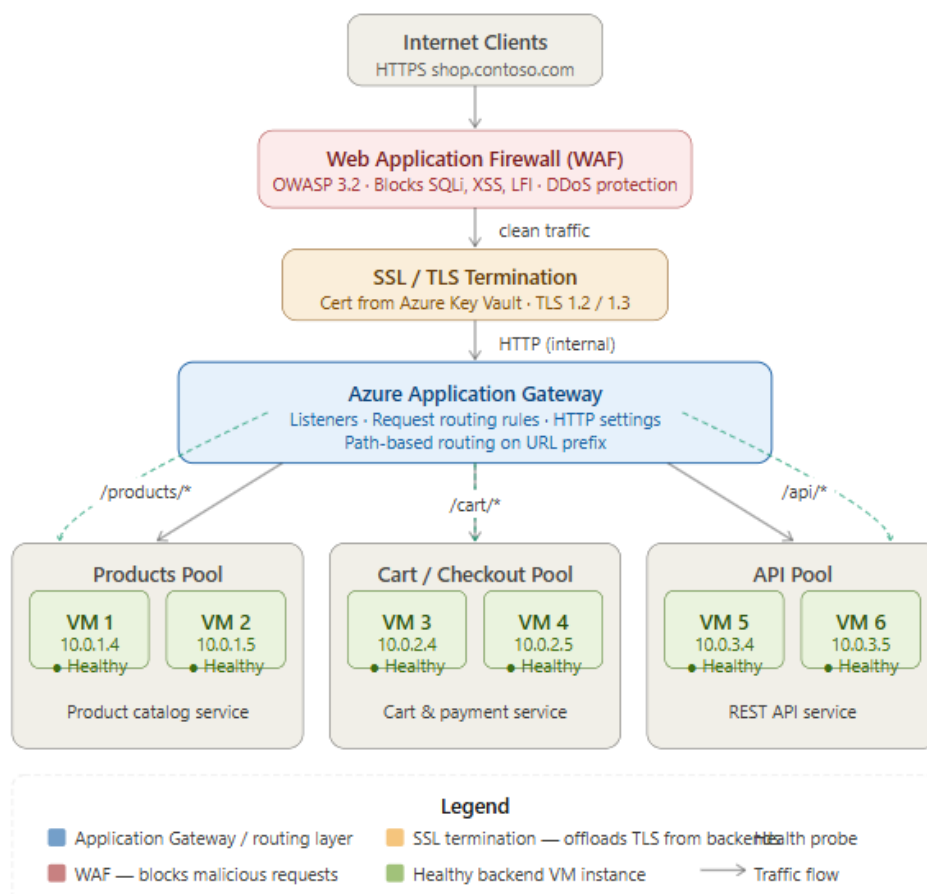


Figure 2.2: Azure Application Gateway providing URL-based routing, SSL termination, and WAF protection for an e-commerce application.

2.4 Why Load Balancing and Scaling are Complementary

Load balancing and scaling are inextricably linked. Scaling alone prevents access to newly added instances, and Load balancing alone overloads existing instances as demand increases. Both work together to form a fail-safe System. The Autoscaler adjusts the number of instances (scaling), and the Load Balancer distributes traffic to the existing instances (Load balancing). Once the Autoscaler prepares a new instance, the Load Balancer automatically adds it to the Backend Pool after a successful health check.

For example, imagine a limited-time sale in an online store. Traffic suddenly increases tenfold. If the Autoscaler starts ten additional Container instances within a few minutes, the Load Balancer immediately distributes this increase to all eleven instances, thus ensuring a response time of less than one second. Without Load balancing, all traffic, even with ten additional instances, would be concentrated on a single instance. Conversely, without scaling, the Load Balancer would have to distribute traffic to a Pool of instances that are not sufficiently loaded.

This type of partnership relationship is the basis for the resilience of the Cloud (Armbrust et al, 2010).

2.5 Combined Benefits: Load Balancing, Scaling, and Application Gateways

When these three technologies work together, a comprehensive, robust, and cost-effective service architecture is created. Using the example of an E-Commerce Website, this integrated architecture can be shown to offer the following benefits:

- **High Availability:** Application gateways and Load balancers always route traffic to error-free instances, eliminating Single Points of Failure
- **Performance:** SSL Offloading at the Gateway level reduces latency, and Load Balancing ensures even resource usage across all Backend instances
- **Cost Efficiency:** Automatic scaling adjusts compute resources to actual needs, unneeded instances are automatically released, and usage-based billing is optimized
- **Security:** The Application Gateway's Web Application Firewall (WAF) blocks malicious traffic before it reaches the Backend, and thus significantly reduces the attack surface
- **Intelligent Routing:** URL-based Routing allows splitting into Microservices so that each Service can be updated, scaled and replaced independently

This Design embodies the Cloud-native philosophy of Fowler and Lewis (2014) It provides a group of small, independently deployable services, controlled by an intelligent Routing layer and each scales autonomously. Users are guaranteed a consistent experience, regardless of fluctuations in traffic. We provide a fast, always available, secure and reliable service-as natural as breathing.

3. Driving Forces Behind Cloud Computing Expansion

The rapid adoption of Cloud Computing over the past 20 years is the result of an interplay of economic, technological and organizational factors. Here we analyze the key drivers, examine Serverless capabilities from each perspective and compare them to Container technology.

3.1 Cost Efficiency

The most common Motivation for Cloud adoption is, of course, economics. Traditional On-Premise data centers required significant investments (CapEx) for Hardware, cooling, physical security and dedicated operating personnel. Cloud Computing transforms this into an operating cost (OpEx) model where companies only pay for the resources they actually use. As described by Armbrust et al (2010), this eliminates the need for excessive investments to prepare for peak loads, previously described as the "Illusion of unlimited, "Compute resources available at all times,"

containerization boosts this cost efficiency by dramatically increasing compute density. Even a single Server that can Host only about 10 virtual machines can run hundreds of Docker containers because they share the Host operating system kernel and the startup effort is almost negligible. Companies like Spotify and Netflix report significant cost reductions in their infrastructure by migrating from VM-based to container-based deployments.

Serverless features offer even greater cost efficiency. AWS Lambda and Azure Functions calculate costs based on the number of calls and execution time (in milliseconds). A key difference to containers is that, that unused resources are not charged at all, even when no requests are processed For infrequent Workloads, such as image processing after file Uploads or nightly reporting, serverless solutions can reduce costs by more than 90 % compared to permanently active containers, but for high-throughput and continuous Workloads, containers can be more economical as they avoid Overhead per call and spread out the fixed costs.

3.2 Flexibility and Agility

Cloud Computing is like a faithful Butler, deploying diverse compute, storage, and network resources on demand. Companies can experiment, iterate, and deploy quickly, gaining a decisive competitive advantage in terms of speed to market in the age of digital Transformation (Bhatt et al, 2022)

Containerization technology significantly accelerates the software development cycle by packaging applications and all their dependencies in a portable image. Developers can build

the application once and deploy anywhere, from a developer's laptop to a Kubernetes Cluster in the Cloud. Docker's Integration with CI/CD Tools such as GitHub Actions, Jenkins, and Azure DevOps enables deployment pipelines, From Code Commit to production release in less than 10 minutes.

serverless features provide maximum deployment agility Features can be written in minutes, deployed with a single CLI command, and are immediately available worldwide Serverless features eliminate the need for infrastructure provisioning and management Cold start latency (the delay in first initializing a function instance) is continuously reduced by vendors through mechanisms such as AWS Lambda 's SnapStart and preheated instances in Azure Premium plans (Amazon Web Services, 2024) However, serverless features have limitations in execution time (up to 15 minutes in AWS Lambda), storage requirements, and deployment package size, limiting their use for complex, long-running Workloads In such cases, containers provide more flexibility .

3.3 Scalability and Elasticity

The ability to scale resources on demand and instantly is a core feature of the Cloud and a major driver of its adoption [2]. While procuring additional servers in traditional infrastructure can take weeks, Cloud infrastructure is up and running in seconds.

As explained in Chapter 2, scaling Kubernetes-based containers horizontally using the Horizontal Pod Autoscaler, however, requires scaling from zero to one (a cold start) in Kubernetes, which requires fetching Images, Pod scheduling, and Container launch, which can take 30-60 seconds. This is a limitation for applications that need to handle sudden spikes in load without lead time.

Serverless features are distinguished by their instant resiliency. Cloud providers provide a Pool of function execution environments that can scale from zero to thousands of concurrent calls in milliseconds with near-zero latency (Amazon Web Services, 2024). For this reason, Serverless is the preferred choice for event-driven architectures such as IoT data collection and API Backends that need to handle unpredictable load peaks. In return, Serverless lacks the control and predictability that container-based deployments provide. Container-based systems allow developers to customize resource allocations, configure custom networks, and run background processes.

3.4 Reliability and Disaster Recovery

High availability and Disaster Recovery are essential for modern applications [7]. Therefore, Cloud providers invest heavily in globally connected infrastructures to ensure robust Service Level Agreements (SLAs). Azure, for example, guarantees 99.99 % availability for its core services [12]. Cloud architectures with geographic redundancy across multiple availability zones and regions are resistant to hardware failures, natural disasters, and network failures.

Container technology increases reliability through a declarative deployment model. Kubernetes monitors continuously running Pods, and failed containers are automatically restarted without human intervention and moved to failure-free nodes. Immutable Images eliminate configuration differences between environments.

Fixing factors that the serverless capabilities that led to outages in production environments in traditional deployments benefit from the built-in Reliability of Cloud providers. Functions are automatically replicated across multiple availability zones, and the provider transparently handles all Disaster Recovery Management. For many organizations, this shift of operational responsibility is the main reason for adopting Cloud solutions. It allows even small development teams to operate highly available services without dedicated Site Reliability Engineers (SREs) (Hendrickson et al, 2016).

3.5 Security and Compliance

Contrary to initial skepticism about Cloud security, the security capabilities offered by Cloud providers today often exceed the capabilities of enterprises to implement them themselves [8]. These include physical data center security, Hardware security modules, identity and access Management, Encryption of data at rest and in transit, and Compliance certifications to Standards such as ISO 27001, SOC 2, PCI DSS, and GDPR.

Container security is an ever-evolving field Docker provides Image scanning Tools such as Docker Scout to detect vulnerabilities at the Image level Kubernetes enforces role-based access control, network policies, and Pod security Standards to limit the impact of compromised containers However, due to the shared Kernel architecture, Kernel vulnerabilities can affect all containers on the host-a risk that does not exist with full VM Isolation.

Serverless features provide a smaller attack surface than containers, as they eliminate the need for OS and runtime management. Security patches are deployed transparently by the service provider. such as Injection attacks (manipulated functional parameters), insecure

deserialization, and misconfigured IAM permissions. Careful attention to security design at the functional level is therefore essential (OWASP, 2023).

3.6 Serverless Functions vs Containerisation: Comparative Analysis

Serverless functions as a Service (FaaS) and containers represent different approaches to delivering cloud-native applications [1]. The choice between the two depends on the characteristics of the workload, the maturity level of the operating structure, and cost sensitivity [7]. Table 3.1 compares the characteristics of the two approaches from important perspectives.

Dimension	Containers (Docker/Kubernetes)	Serverless Functions (FaaS)
Startup Time	Seconds to minutes (cold start)	Milliseconds (warm pool)
Billing Model	Per second of running time	Per invocation + millisecond execution
Execution Limit	Unlimited (long-running)	Max 15 Min (AWS Lambda)
State Management	In-memory, volumes, external DB	Stateless; external storage required
Control & Flexibility	Full OS/runtime control	Limited; provider-managed runtime
Operational Overhead	Medium (Kubernetes management)	Very low (fully managed)
Scalability	Horizontal; HPA-managed	Automatic, instant, to thousands
Cost (High Throughput)	More economical	Higher due to per-invocation cost
Cost (Low Frequency)	Inefficient (idle costs)	Very economical (pay-per-use)
Portability	High (run anywhere with Docker)	Low (provider lock-in)
Best Use Case	Microservices, web apps, ML	Event-driven, IoT, ETL jobs

Table 3.1: Comparative analysis of Containers versus Serverless Functions across key dimensions.

Recently, hybrid architectures used by companies such as Amazon and Microsoft have gained increasing prominence. This approach cleverly combines two different concepts: stateful services (such as databases and APIs) are containerized using Kubernetes, while event-driven tasks (such as image scaling, notifications, and data transformation) leverage Serverless capabilities. By combining the strengths of both technologies, their respective limitations are balanced by both containerization and Serverless capabilities, but in different ways and complement each other. Containerization provides controllability, portability, and sustained efficiency, while Serverless architectures provide instant scalability. The development of

Cloud Computing is not characterized by the choice of one or the other technology, but by their intelligent combination.

4. Conclusion

This report highlights the pivotal role of Container technology, especially Docker, in modern Cloud Computing [9]. The shift from Hardware virtualization to container-based deployments represents a significant advance in efficiency, portability and developer experience [3]. Using practical examples such as Docker file creation, image publishing, Volume Management, and Container networking, the report vividly demonstrates Docker's comprehensive Workflow and its practical applications in real-world Cloud environments.

By exploring Cloud Computing concepts, especially the implementation of Load Balancing, scalability, and Application Gateways in Azure, the report explains how these three technologies work together to provide highly available, cost-effective, and secure services. This complementary relationship forms the foundation of cloud-native architectures: Load Balancing distributes traffic, and Application Gateways do intelligent Routing and security.

An examination of the drivers of Cloud Expansion revealed interesting differences: while Container technology provides portability, control, and efficiency for persistent Workloads, Serverless capabilities enable unprecedented scalability and cost efficiency for event-driven, sporadic tasks. The availability of these two paradigms and the industry's ability to intelligently combine them suggests that Cloud Computing has reached a mature stage. Enterprises of all sizes are developing robust, scalable, and globally accessible services.

As Cloud Computing evolves, new Trends such as WebAssembly-based functionality (WASM), secure Computing, and Edge Computing will continue to expand the opportunities described in this report. The fundamental principles of abstraction, however, fault tolerance, and usage-based billing will undoubtedly remain central to Cloud architecture for the foreseeable future.

References

- [1] Amazon Web Services, "AWS Lambda Developer Guide," 2024. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>. [Accessed: 10 Apr. 2026].
- [2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.
- [3] P. Bhatt, S. Chakraborty, and D. Bhatt, "Container technologies: Docker and Kubernetes architecture and deployment scenarios," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 1, pp. 112–121, 2022.
- [4] B. Burns, J. Beda, K. Hightower, and C. McLuckie, *Kubernetes: Up and Running*, 3rd ed. Sebastopol, CA: O'Reilly Media, 2022.
- [5] Docker Inc., "Docker Documentation," 2024. [Online]. Available: <https://docs.docker.com>. [Accessed: 10 Apr. 2026].
- [6] M. Fowler and J. Lewis, "Microservices," 2014. [Online]. Available: <https://martinfowler.com/articles/microservices.html>. [Accessed: 10 Apr. 2026].
- [7] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with OpenLambda," in *Proc. 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, USA, 2016. USENIX Association.
- [8] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Gaithersburg, MD, NIST Special Publication 800-145, 2011.
- [9] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.
- [10] Microsoft, "Deploy a container instance in Azure using the Azure CLI," 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/container-instances/>. [Accessed: 10 Apr. 2026].
- [11] Microsoft, "Use Azure Files with Azure Kubernetes Service," 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/aks/azure-files-volume>. [Accessed: 10 Apr. 2026].
- [12] Microsoft, "Azure Load Balancer documentation," 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/load-balancer/>. [Accessed: 10 Apr. 2026].
- [13] Microsoft, "Kubernetes Event-Driven Autoscaling (KEDA)," 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/aks/keda-about>. [Accessed: 10 Apr. 2026].
- [14] Microsoft, "Azure Application Gateway documentation," 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/>. [Accessed: 10 Apr. 2026].
- [15] OWASP, "OWASP Serverless Top 10," 2023. [Online]. Available: <https://owasp.org/www-project-serverless-top-10/>. [Accessed: 10 Apr. 2026]