

UNIVERSITY OF WALES TRINITY SAINT DAVID
London Campus

Module: Cloud Engineering Principles (CENG7023)
Assessment: Task 1 – Technical Report on Virtualisation and Containerisation

TASK 2: Cloud Load Balancing, Scalability, and App Gateways

Student Name: Rishabh Choudhary
Student ID: 2434332
Program: MSc Cyber Security & Digital Forensics
Module Leader: Renuga Jayakumar
Submission Date: 18 December 2025

A video demonstration supporting Task 1 has been uploaded to YouTube

Video link: <https://youtu.be/21oeM7H5ZGY>

Abstract

Virtualisation allows efficient allocation and utilization of resources provided by the compute hardware. Traditional virtual machine-based virtualisation causes certain restrictions regarding performance and scalability, which are tied to running full operating systems. Containerisation has proved to be a promising solution to overcome such restrictions, which are based on operating system levels.

In this case, the principles of virtualisation are explored and the containerisation solution to the classical problems associated with virtualisation are evaluated. As part of the practical implementation, a web app developed on Docker is used to illustrate the creation and distribution of a container, long-lasting storage using volumes, and communication between various containers using virtual networks. These elements are used to pave the way to explore further into Task 2.

Introduction	1
1. Cloud Engineering Concepts and Practical Implementation	2
1.1 Principles of Virtualisation	2
1.1.1 Role of Hypervisors in Virtualisation.....	3
1.1.2 Types of Hypervisors	3
1.1.2 Types of Virtualisation	4
1.2 Challenges of Traditional Virtualisation	5
1.3 Principles and Benefits of Containerisation	6
1.3.1 Docker as a Containerisation Platform	6
1.4 Docker Images and Publishing to DockerHub	7
1.5 Limitations of Ephemeral Containers	9
1.6 Docker Volumes	9
1.6.1 Procedure: Docker Volume Creation and Usage	10
Transition from Storage to Network Virtualisation	12
1.7 Container Networking.....	12
1.7.1 Procedure: Docker Network Creation and Usage.....	13
1.8 Summary and Conclusion.....	16
TASK 2: Cloud Load Balancing, Scalability, and App Gateways	17
2.1 – Cloud Provider’s Implementation.....	17
2.2 - Why Load Balancing & Scaling Are Complementary	20
2.3 - Benefits of Combining All Three Technologies	21
2.4 – Driving Forces Behind the Adoption of Cloud Computing.....	25
2.5 Serverless Functions vs Containerisation.....	26
2.5.1 Serverless (AWS Lambda, Azure Functions)	27
2.5.2 Containers (Docker, Kubernetes)	27
References	1

Introduction

Virtualisation is an underlying concept in the computing that enables multiple isolated environments to run simultaneously on the same physical resource through abstraction. This concept brings various benefits such as improvement in the utilization of resources, isolation, and flexibility, which are essential for modern cloud engineering [1].

Nonetheless, the traditional virtual machine-based virtualisation incurs overhead as it requires complete guest operating systems, which translates to high resource utilization and longer deployment times. These limitations make it less efficient for an environment where rapid app deployment and scalability are required.

Containerisation provides a solution for such problems, as it provides lightweight application isolation on the operating system-level. This makes it possible for various applications to run inside one common kernel but still be isolated. Docker is one such popular platform that provides an implementation for containerisation based on the standardization of the packaging, running, and distribution of the applications.

This report investigates the concept of virtualisation and critically evaluates containerisation as an efficient method for dealing with the challenges that come with the use of virtualisation. This report provides an overview of the use of Docker for the development of a web app, which provides insights into the process of Docker image building, sharing, and the use of volumes for persistence, as well as leveraging virtual networks for communication between containers. This report serves as the basis for Task 2.

1. Cloud Engineering Concepts and Practical Implementation

1.1 Principles of Virtualisation

Some of the defining principles of virtualisation include abstraction, isolation, and resource pooling. Here, abstraction hides the intricacies of hardware infrastructure and provides standardized virtual hardware resources for the operating systems of different virtual machines. Isolation prevents tasks executed in virtual machines from affecting each other. In this principle of virtualisation, resources are dynamically allocated and deallocated on the demands of applications.

Virtualisation is used to transform physical infrastructure into virtualised environment which reduces the overheads to maintain the hardware. Virtualisation can be achieved by creating Virtual Machines (VMs) [1].

A layer of indirection can help solve many problems including efficiency, security, high availability, elasticity, fault containment, mobility, and scalability [2].

In cloud engineering, the use of virtualisation enables cloud service providers to achieve optimal hardware utilization and provide users with flexibility in accessing computing power when needed. Yet the application of such concepts will be made possible by certain software elements that control the operation of virtualised environments. **They are brought in through the use of hypervisors.**

1.1.1 Role of Hypervisors in Virtualisation

Hypervisors are very important in hardware-based virtualisation and work as the control entity between hardware and virtual machines. In this case, a hypervisor is known to oversee both creation and execution of virtual machines and control the isolation between them. In addition, it controls the sharing of computing resources like CPU, memory, storage, and network connectivity between operating systems.

The hypervisor is a program or a combination of software, hardware or firmware that creates and executes various Virtual machines. The computer running hypervisor is called the host machine.

1.1.2 Types of Hypervisors

Type 1 hypervisors, also called "bare-metal" hypervisors, run directly on the hardware of a server, without an underlying host operating system. This makes them faster, more secure, and ideal for such environments as cloud platforms and enterprise data centers. Examples include VMware ESXi, Microsoft Hyper-V, and Xen [2].

Type 2 hypervisors reside on top of an existing host operating system and run like regular software. Easy to install and use, they are popular for development, testing, and personal virtual machines; examples include VirtualBox and VMware Workstation [2], [3].

In the context of the implemented environment investigated in this report, it is identified that Type 2 hypervisor architecture is utilized for the support of virtualisation. Based on the hosted hypervisor, it is possible to create Linux virtual machines on Windows host environments for further application deployment and testing.

The virtual machine created on this hypervisor platform forms the basis for operating system-level virtualisation and container-based applications that are included in the final product. This has very practical applications in cloud computing, as many cloud virtualisation systems are integrated together for scalable applications.

1.1.2 Types of Virtualisation

Hardware Virtualisation

This means many virtual machines can run on one physical server, each acting like its own computer. Hardware level virtualisation provides computer hardware as an execution environment on which guest operating systems can be executed.

A virtual machine created by hardware virtualisation acts like a real computer with an operating system. The underlying hardware resources can separate various software runs on virtual machines. For example, if a host machine has an operating system like Microsoft Windows, then a guest machine may have Ubuntu Linux as an operating system. It can also run Ubuntu-based software on it. Hardware assisted virtualisation, para-virtualisation, and full-virtualisation are different types of hardware virtualisation techniques [3].

The principles of virtualisation allow one physical computer to behave like many separate machines through shared hardware, isolation, resource pooling, and abstraction.

1.2 Challenges of Traditional Virtualisation

Though cloud computing owes its existence to virtualisation and is ever reliant on it, the classic use of virtual machines poses some inherent difficulties when it comes to hosting modern applications. In this case, virtual machines contain an entire operating system and applications, and this causes the operating systems and applications to be replicated in the case of multiple workloads.

Even provisioning and scaling virtual machines have some shortcomings in environments that demand rapid elasticity. Generally, virtual machines entail higher boot times and preprovisioning of resources, making them less optimal for dynamically intensive applications like microservices and CD pipelines. Even further, administration of many virtual machines incurs higher complexities in the operating system patching process and inconsistencies in development, testing, and production environments pertaining to the dependencies of applications. Altogether, such shortcomings limit cloud-native application environments in terms of achieving higher agility and scalability [4].

Containerisation emerges as a response to the structure inefficiencies of hardware-based virtualisation, shifting isolation from the hardware layer to the operating system layer. In containerisation, instead of building an entire machine, isolation is achieved at the process level that shares the same operating system kernel on the host operating system. This makes it much less expensive compared to virtualisation.

1.3 Principles and Benefits of Containerisation

Containerisation is a technology of virtualisation of applications, which results in significant enhancement in the management of cloud applications . The Containers encapsulate an application and the associated dependencies inside its own environment. It allows them to execute in isolated fashion while utilizing the same resources, including the Operating System. It enables a rapid, light weight application deployment because the resources are utilized in feasible fashion by not wasting on running separate operating system [5]. Containerisation offers numerous benefits compared to traditional virtual machines because it runs applications in isolated, light environments that share an operating system. Without having a full guest OS, they start up faster and use less system resources, thus being more efficient and scalable for cloud environments. In this regard, containerisation helps in achieving more consistency in environments used for the deployment of applications [5], [6].

1.3.1 Docker as a Containerisation Platform

Docker represents a popular container infrastructure solution that provides a pragmatic realization of the containerisation paradigm. It helps applications be bundled together with their dependencies in portable, immutable forms called container images.

One of the main strengths of Docker is its deployment predictability. This is because applications packaged as containers behave in the same manner within the development, testing, and production environments. Another strength of Docker is its efficiency in terms of resource usage.

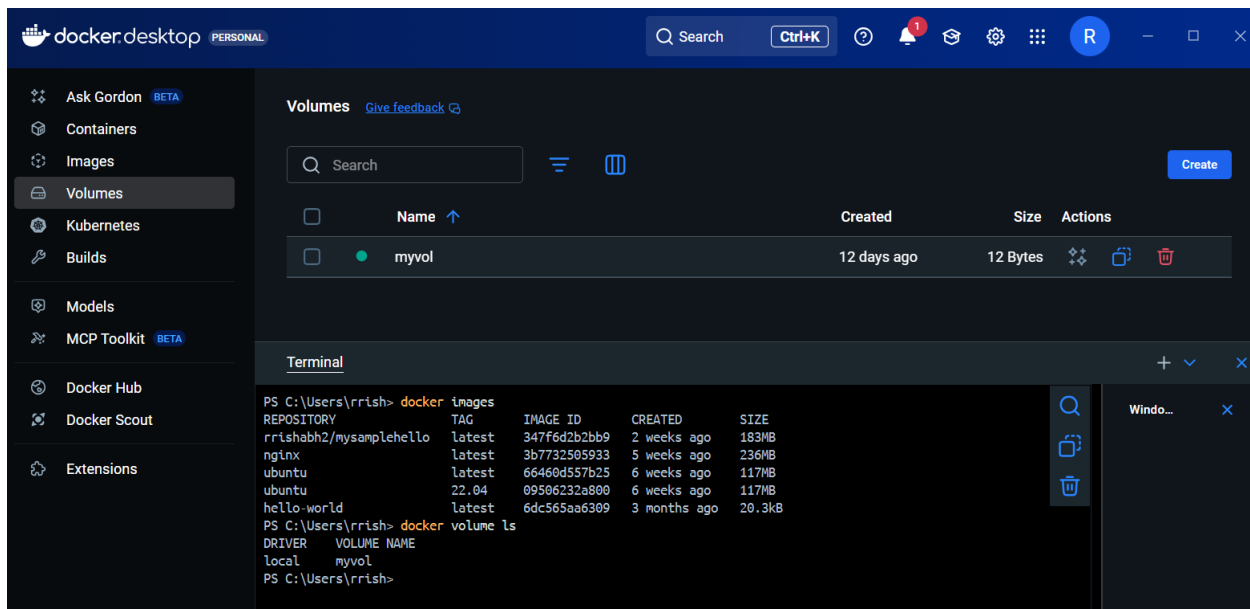


Figure 1: Verification of locally built Docker images

1.4 Docker Images and Publishing to DockerHub

Docker images serve as read-only templates for creating containers. Every image has everything necessary to run an application, including the base operating system, application files, libraries, and dependencies. Docker uses a layered architecture for images, with each instruction in the Dockerfile creating a new layer of the image, which increases reusability and reduces storage needs [6].

One can pull images from public registries like Docker Hub or create them locally with the help of a Dockerfile. Once created, an image can be tagged, versioned, shared, and deployed on any host running Docker. That makes

Docker images highly portable and consistent across development, testing, and cloud environments [6].

To demonstrate application deployment using Docker, a Dockerfile was used to define the runtime environment and required application files for a containerised web application.

The Dockerfile describes the base image as well as the necessary application files that must be incorporated into the image. An example Configuration can be seen as follows:

Listing 1: Dockerfile used to build the NGINX web application image

```
FROM nginx:latest
```

```
COPY index.html /usr/share/nginx/html/index.html
```

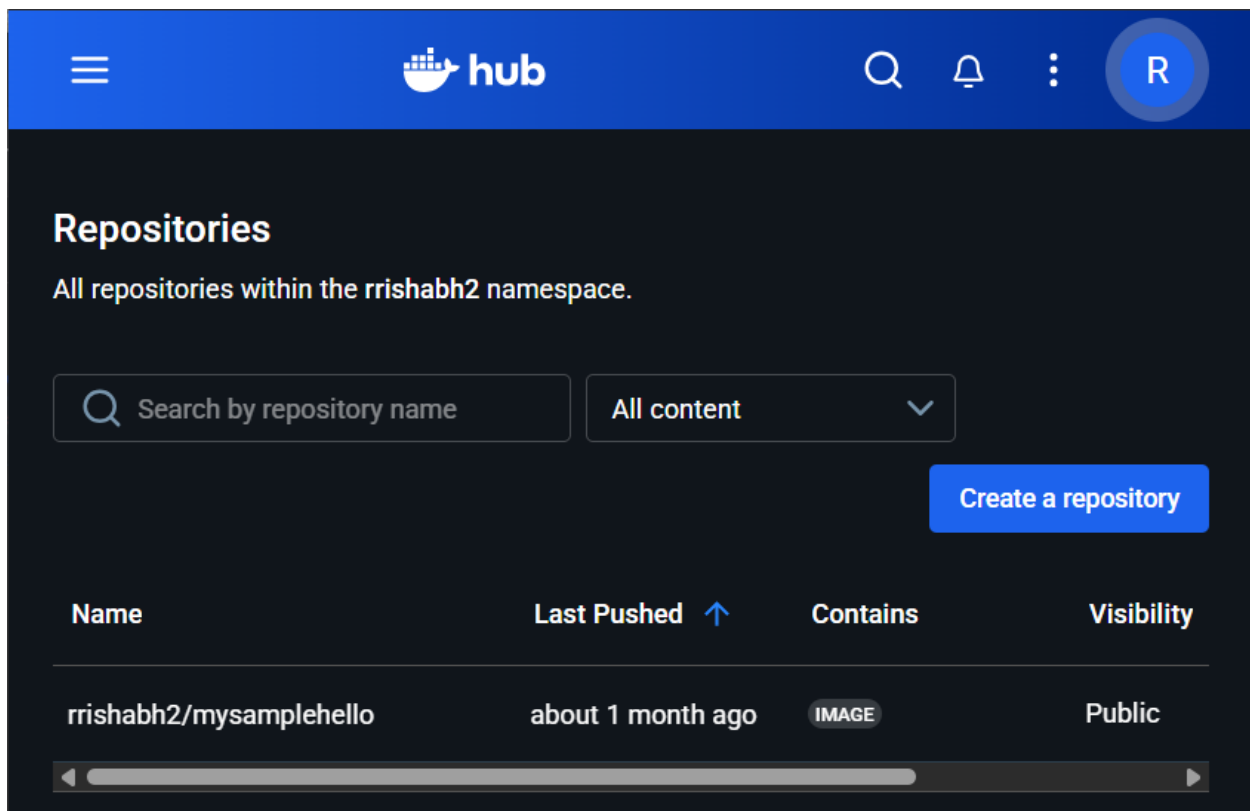


Figure 2: Published Docker Image repository on DockerHub

1.5 Limitations of Ephemeral Containers

Containerised applications are designed to be lightweight and disposable. By default, data written to a container's writable layer is lost when the container is stopped or removed. While this behavior is acceptable for stateless services, it presents significant limitations for stateful applications that require persistent storage, such as web servers storing Configuration data, logs, or user-generated content. In cloud engineering environments, the inability to retain data beyond the lifecycle of individual containers undermines reliability, scalability, and fault tolerance. As containers may be recreated frequently due to scaling operations, failures, or updates, a mechanism is required to complete application data from container instances [7].

Docker Volumes as a Storage Virtualisation Solution. Docker volumes address the persistence challenge by providing a storage abstraction that exists independently of individual containers. Volumes are managed by the container platform and mapped into containers at runtime, enabling data to persist even when containers are stopped, replaced, or redeployed.

1.6 Docker Volumes

Docker volumes are a way to store data outside of the container filesystem so that it persists even after containers are removed or recreated. Unlike the internal container layer—which is lost when the container is deleted—volumes allow applications to persist in logs, configuration files, or databases across multiple runs. This becomes essential in stateful applications, data-driven services, and real-world deployments where temporary storage is not enough [8].

Volumes are directly managed by Docker, so they are easy to back up, share between containers, and migrate across environments. They also provide better performance and reliability as compared to storing data inside the container's writable layer.

1.6.1 Procedure: Docker Volume Creation and Usage

To enable persistent storage for containerised applications, Docker volumes were used to implement storage virtualisation. Docker volumes provide a platform-managed storage mechanism that exists independently of container lifecycles.

1st Step - Volume Creation

A named Docker volume has been created by running the following command:

```
docker volume create myvol
```

This command tells Docker to assign a logical storage resource, which is handled by the Docker engine.

2. Verification of Volume

The successful creation of the volume was proved by listing all available Docker volumes:

```
docker volume ls
```

The output provided validation on the creation of the new volume and served as confirmation that the new storage resource has been successfully provisioned. The new volume is further mounted into a running container using the binding concept between the volume and a directory inside the container filesystem. The validation of the new volume creation serves to illustrate the concept of Docker volumes and how applications are decoupled

from container runtime, which is essential in determining data persistence after the container is terminated.

Step 3 – Validation Persistence

- Data written to the volume on the mounted directory persisted beyond the lifetime of the container. When a new container instance is spun up with the volume mounted, data written before the container instance existed was accessible, showing evidence of persistent resources beyond individual container instances. This is how volumes, supported by Docker, implement the concept of storage virtualisation. This concept involves decoupling data and compute resources. Below are screenshot proving the creation, mounting, and persistence of data on volumes.

```
rrishabh@rishabh:~$ docker volume create myvol
myvol
rrishabh@rishabh:~$ docker volume ls
DRIVER      VOLUME NAME
local       myvol
rrishabh@rishabh:~$ docker run -it -v myvol:/data ubuntu bash
root@fbf3a6a7b47e:/# echo "hello volume" > /data/test.txt
ls /data
test.txt
root@fbf3a6a7b47e:/# exit
exit
rrishabh@rishabh:~$ docker run -it -v myvol:/data ubuntu bash
ls /data
root@bf04fcb834a8:/#
```

Figure 3: Persistent data retained across container lifecycle using Docker volume

Transition from Storage to Network Virtualisation

While Docker volumes address data persistence and storage of independence within containerised applications, cloud-based systems rarely operate in isolation. Containerised services often need to communicate with other services, clients, and infrastructure components. Ensuring reliable and secure communication between isolated containers introduces additional networking challenges. To address these requirements, container networking is used to provide virtualised communication pathways between containers.

1.7 Container Networking

The container networking enables the ability to communicate with other containers, a host system, and external networks. Each container will be assigned to a virtual network interface and an IP address to make network isolation or connectivity of applications possible depending on the deployment needs [9].

Docker networking provides a virtualised networking layer that allows containers to communicate securely while maintaining logical isolation. Through user-defined networks, Docker enables containers to discover and communicate with each other using container names rather than fixed IP addresses. This approach abstracts underlying network infrastructure and reflects the principles of network virtualisation used in cloud platforms.

It is a flexible networking model that allows different containers to collaborate as microservices, remaining isolated and secure [9].

1.7.1 Procedure: Docker Network Creation and Usage

As a practical example of container networking, a Docker bridge network has been created by the user to facilitate the controlled interaction of the containers.

Step 1 – Network Creation

A custom Docker network has been created using the following command:

```
docker network create mynetwork
```

This resulted in a bridge network defined by the user, enabling name-based communications between the connected containers.

Step 2 – Network Verification

The existence of the network was verified using:

```
docker network ls
```

The presence of ‘mynetwork’ in the output is an indication of a successfully created network.

Step 3 – Attaching Containers to the Network

Containers were launched and attached to the custom network at runtime:

```
docker run -d --name web1 --network mynetwork nginx  
docker run -it --name client --network mynetwork alpine sh
```

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

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

PS C:\Users\rrish> docker network create mynetwork
Error response from daemon: network with name mynetwork already exists
PS C:\Users\rrish> docker network ls
NETWORK ID        NAME                DRIVER              SCOPE
75168594c9ba      \                   bridge              local
0d135e5103ed      bridge             bridge              local
3e04dca61d32      host               host                local
82c5b25ddb8a      mycustomnet        bridge              local
e4004d6e9a8e      mynetwork          bridge              local
c82625eb69bb      none               null                local
PS C:\Users\rrish>
```

Figure 4: User-defined Docker bridge network

Step 4 – Network Communication Validation

Connectivity between the containers is tested by forming network requests between the containers using their respective names. Connectivity between the containers is essential to guarantee correct Docker networking. These steps prove the relevance of network virtualisation when it comes to containerized networking, where communications are mutually isolated yet fully operational. The screenshots provided prove the creation, connectivity, and successful communications to and from the containers.

```
Terminal
c82625eb69bb  none      null      local
PS C:\Users\rrish> docker run -d --name web1 --network mynetwork nginx
57801a043dd46085116b1d2dbaace39bdaf9a6aac43e51608df6046e025d6791
PS C:\Users\rrish> docker run -it --name client --network mynetwork alpine sh
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
2d35ebdb57d9: Pull complete
Digest: sha256:4b7ce07002c69e8f3d704a9c5d6fd3053be500b7f1c69fc0d80990c2ad8dd412
Status: Downloaded newer image for alpine:latest
/ # ping web1
PING web1 (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.848 ms
64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.144 ms
64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.153 ms
64 bytes from 172.18.0.2: seq=3 ttl=64 time=0.185 ms
64 bytes from 172.18.0.2: seq=4 ttl=64 time=0.120 ms
64 bytes from 172.18.0.2: seq=5 ttl=64 time=0.145 ms

RAM 1.00 GB CPU 4.72% Disk: 1.98 GB used (limit 1006.85 GB)
```

Figure 5: Inter-container communication

The following Figure demonstrates the successful deployment of the Containerised application, integrating container images, persistent storage, and networking.

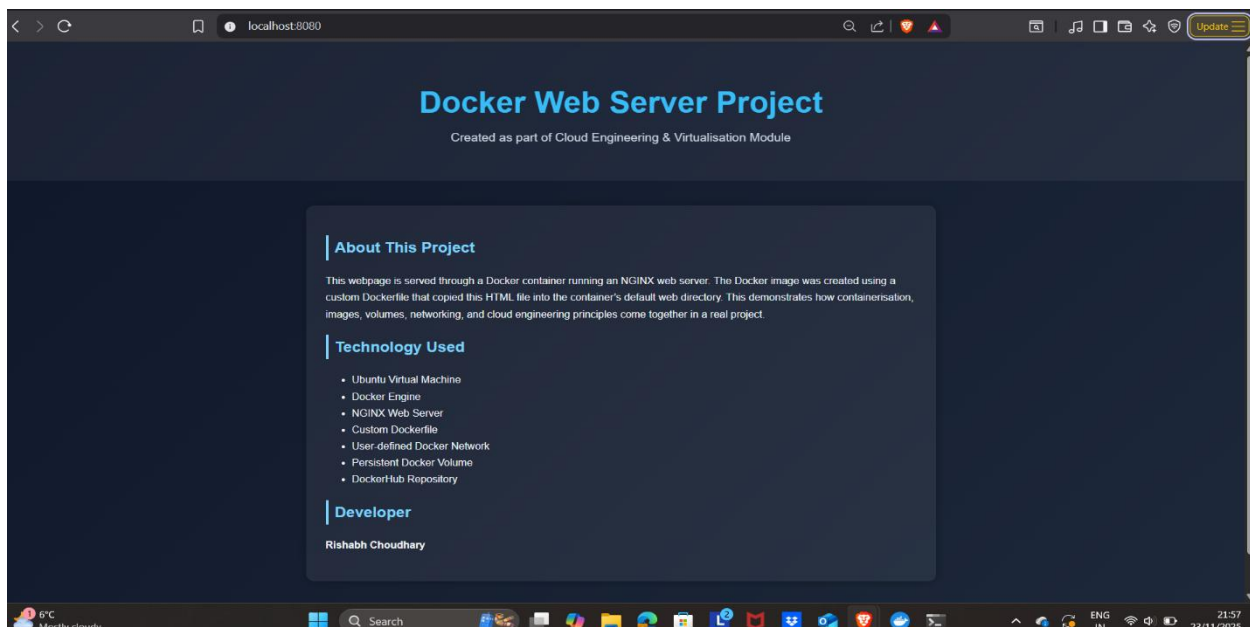


Figure 6: NGINX web app successfully deployed using docker container

1.8 Summary and Conclusion

This report examines the concept of virtualisation and highlights the limitations faced by the conventional methods of virtual machine-based systems, specifically with regards to performance and efficiency. The efficiency of containerisation is also assessed and identified to address such limitations by promoting light operating system-level containment. Docker is discussed and defined as a platform to achieve containerisation.

To further demonstrate the concept and implementation of a containerized web app, it is important to understand the different elements, such as the creation and distribution of a Docker image, the long-lasting storage of data using volumes, and communication between various containers using virtual networks. The implementations discussed demonstrate the importance of containerized infrastructures to improve the efficiency and scalability of applications today, and the understanding and implementation discussed will enable further applications and assessments to take place, which are covered under Task 2.

TASK 2: Cloud Load Balancing, Scalability, and App Gateways

2.1 – Cloud Provider’s Implementation

Cloud applications depend on three core mechanisms: **load balancing** to distribute requests, **auto-scaling** to adjust capacity and **application gateways** to manage traffic at the application layer. Each cloud provider offers these functions, but their design choices reflect different priorities. The table below summarises the principal services.

Provider	Load Balancing Services	Auto Scaling / Gateway Services
Amazon Web Services (AWS)	The Elastic Load Balancers include Application Load Balancers (ALB), which handle HTTP/2 and gRPC; Network Load Balancers (NLB), which support high-throughput TCP/UDP; and Gateway Load Balancers (GLB), which assist in connecting third-party appliances	Auto Scaling Groups (ASGs) do horizontal scaling of EC2 instances according to rule-based, step, tracking, or predictive scaling, and container platforms (EKS, Fargate) support cluster and pod autoscalers. API Gateway/ALB offers layer 7 routing, TLS termination, and integration with WAF, while CloudFront functions as a global gateway for static and dynamic content.
Microsoft Azure	Azure Load Balancer (layer-4) and Application Gateway (layer-7) load and direct TCP/UDP and HTTP/S traffic;	Various services autoscale VMs or apps by metrics, schedule, or forecast; Azure Kubernetes Service (AKS) autoscales clusters and horizontal pods. Azure App

Provider	Load Balancing Services	Auto Scaling / Gateway Services
	global acceleration with Azure Front Door.	Gateways (WAF and path-based routing), Azure API Management (API lifecycle), and Front Door (global TLS termination and routing).
Google Cloud Platform (GCP)	Cloud Load Balancing offers global HTTP(S), SSL proxy, TCP/UDP, and internal anycast load balancing capabilities.	Managed Instance Groups (MIGs) scale Compute Engine instances based on CPU, request, or custom metrics, Google Kubernetes Engine (GKE) supports cluster and pod autoscalers, while Cloud Functions and Cloud Run scale to zero. API Gateways, Cloud Endpoints, and Apigee X handle APIs; the global HTTP(S) load balancer is also a layer 7 gateway with URL-based routing and WAF capabilities.

AWS led the way with elastic infrastructure, providing a mature set of load balancing solutions. The Application Load Balancer (ALB) offers host-based and path-based routing, is tightly coupled with Web Application Firewall (WAF), a managed service offered by AWS, and automatically includes instances launched from Auto Scaling groups. The Network Load Balancer (NLB) is a low-latency networking solution, while the Gateway Load Balancer (GLB) is a new set of third-party devices. Auto Scaling is triggered by metrics

that are monitored via CloudWatch, which use thresholds, schedules, or machine learning predictions. In the AWS environment, Application Gateway is delivered via API Gateway and ALB, with support for REST, WebSocket, gRPC, routing, validation, and authentication.

Azure focuses on enterprise-level integration. The layer-4 Azure Load Balancer is used for high-throughput TCP/UDP traffic, while the Application Gateway is layer 7 with SSL offloading, cookie-based sessions, and WAF. The Azure Front Door Service provides worldwide routing, leveraging Microsoft's WAN to route customers close to the nearest available region. The Service VM Scale Sets support scaling of VMs, while autoscaling for web and API apps is supported by App Service, with AKS offering AKS autoscalers. The Azure Application Gateway, along with API Management, provides centralized authentication, rate limiting, and developer portals.

The service uses the planet-wide Google network, with Cloud Load Balancing a software-defined, anycast-based service that supports worldwide HTTP(S) load balancing with failover capabilities across regions. TCP/UDP, SSL proxy, and Target Proxies support network-focused applications, with internal load balancing that keeps traffic within a Virtual Private Cloud. Managed Instance Groups dynamically scales Compute Engine instances either reactively or predictively, with Google Kubernetes Engine (GKE) autoscalers scaling clusters and pods, while Cloud Functions and Cloud Run support scaling to zero. The Google Cloud Platform supports API management via API Gateway & Apigee X, with an HTTP(S) LoadBalancer also acting as an application gateway.

Such components interact with each other, with load balancers facilitating even distribution, autoscalers managing capacity based on demand, and application gateways ensuring secured routing. Referring to scalability,

Jamshidi et al. describe how the autoscaling controllers buy and release resources in the cloud to suit the fluctuating workloads [10]. Bernstein explains that cloud infrastructure relies on resource pooling, together with a high degree of elasticity, for scalability of cloud services [11].

2.2 - Why Load Balancing & Scaling Are Complementary

Both load balancing and scaling are solutions that address different issues that arise in a queuing problem. In a queuing system, the waiting time rises substantially close to 100% utilization of the server. The load balancing helps to balance the requests on the servers such that a server is never overwhelmed, hence shortening the waiting response time. The auto-scaling takes care of adding more servers when the aggregate demand surpasses the availability of the existing scaled set of servers. In the case that there is no scaling, a load balancing tool is useful because it spreads the load on the overloaded servers; on the other hand, when there is no balancing, the scaled solution may not be fully utilized.

Autoscaling controllers dynamically procure and release virtual machines based on variations in workload [10]. This is a fundamental characteristic of cloud economies, which enable scalability. Bernstein states that cloud service vendors aggregate resources, which can then be allocated and released automatically by the system [11]. Load balancing helps optimize the use of such resources. It avoids a situation where certain instances are idle while others are overloaded, thus increasing the likelihood of SLA violations. An article from a 2023 IEEE Access survey explains that an imbalanced load affects scalability efficiency [12].

On the economic side, the effect of combining load balancing with auto-scaling means that there is minimal wastage on over-provisioning as well as under-provisioning. Over-provisioning is wastage that occurs when an organization spends money on idle instances, while under-provisioning can result in a penalty on performance as well as a loss of revenue.

In terms of performance, synergy improves availability, as load balancers use mechanisms such as instance bypass, stickiness, and content-based routing, which are supplemented by the autoscaler replacing failed instances. The application gateway provides additional synergy by enforcing routes that dynamically change with scaling actions on the application backends. The overall result is a system that has a responsive design with attributes such as low latency, high throughput, and high availability when it comes to fault tolerance.

2.3 - Benefits of Combining All Three Technologies

The combined effect of application gateways, load balancing, and auto-scaling provides a robust, secure, and high-performance cloud service infrastructure. Contrary to being used separately, these technologies together complement each other; they tackle different layers of traffic management and

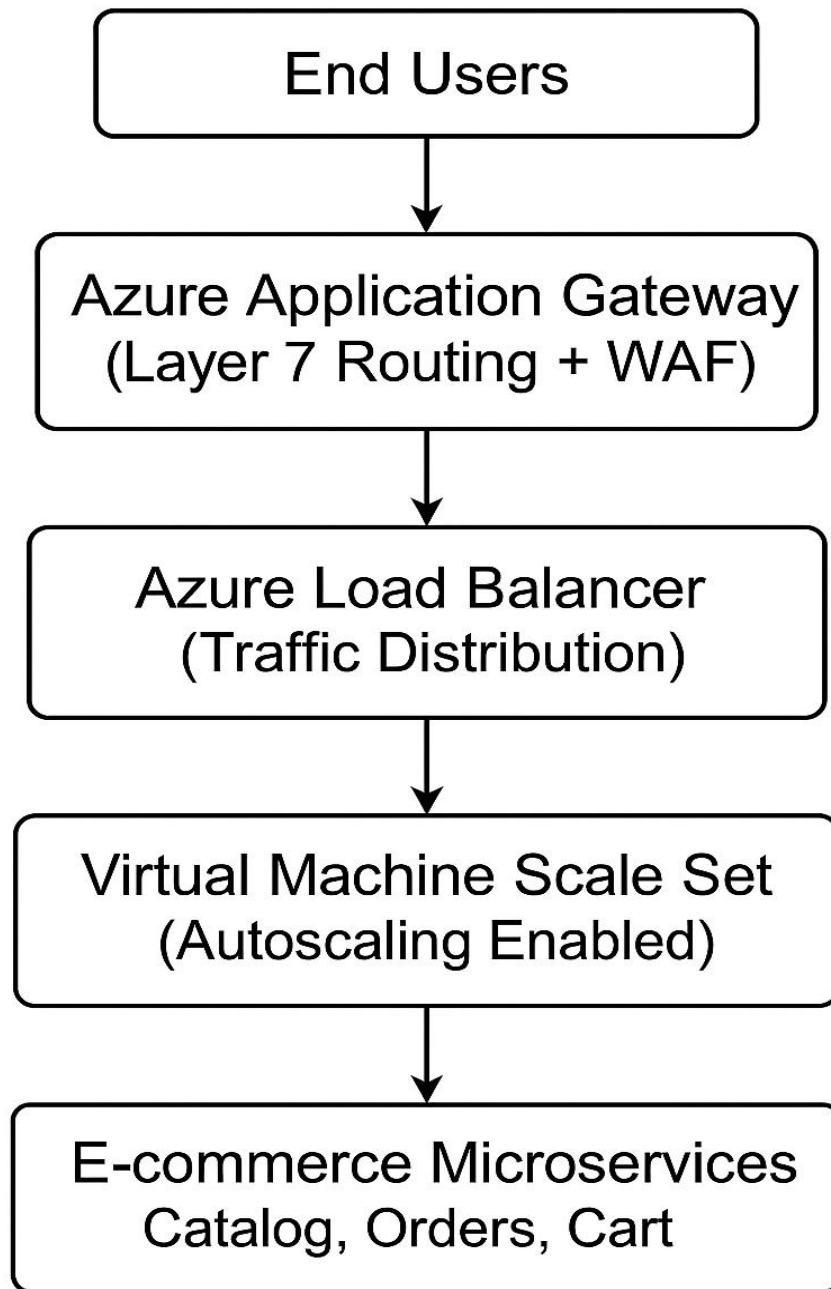


Figure 7: High-level cloud architecture for an e-commerce

As illustrated in **Figure 7**, client requests are first handled by an application gateway, which operates at the application layer (Layer 7) to perform intelligent routing, SSL termination, and web application firewall (WAF) functions. This ensures that only validated and correctly routed traffic is

forwarded to backend services. The load balancer then distributes incoming requests evenly across multiple application instances, preventing overload on any single resource and improving overall system reliability.

Auto-scaling enhances this architecture by dynamically adjusting the number of backend instances based on real-time demand. During peak usage, additional instances are automatically provisioned to maintain performance, while excess resources are removed during low-demand periods to optimise cost efficiency. This elasticity ensures consistent service availability without manual intervention.

In a real-world scenario such as an e-commerce platform, this integrated approach allows secure user access, uninterrupted service during traffic surges (e.g., seasonal sales), and cost-effective resource utilisation. Azure portal evidence shown in **Figures 8** demonstrates the practical implementation of these technologies working together to deliver a scalable, secure, and fault-tolerant cloud service service availability.

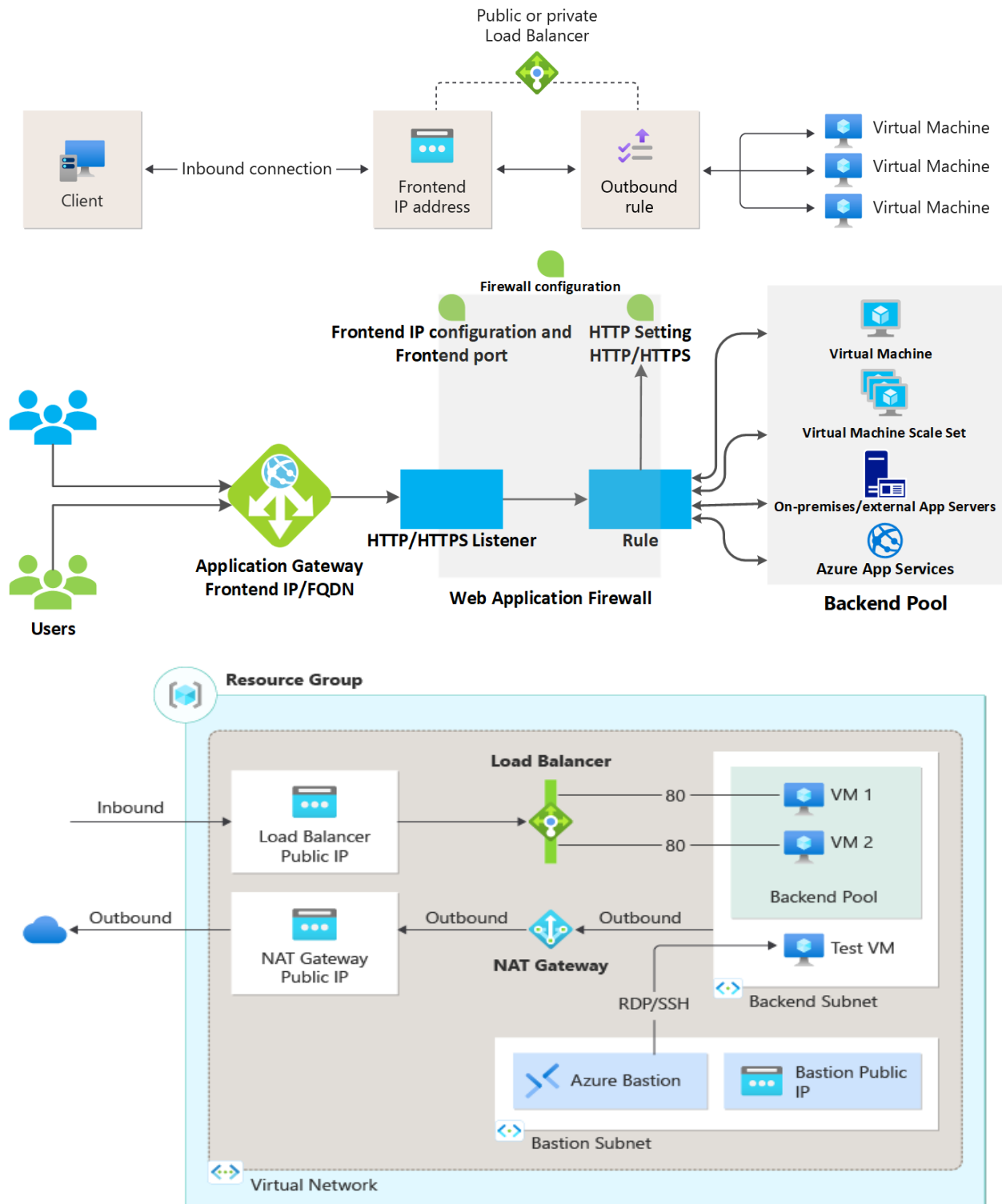


Figure 8: Azure portal evidence showing application gateway, load balancer, and autoscaling configuration

2.4 – Driving Forces Behind the Adoption of Cloud Computing

Cloud adoption is triggered by a mixture of economic factors and enabling technologies.

Cost Optimization & Pay as You Go Model: On-premises infrastructure requires capital outlay in terms of equipment, software, and real estate. Cloud infrastructure technologies transform capital expenses into operating costs. Bernstein identifies that multi-tenancy, which is adopted by cloud infrastructure, allows customers to pay only for what they use [11]. Autoscaling further increases this benefit by releasing back resources that are not in use, with load balancing that makes use of the assigned resources.

Elasticity and flexibility: Elasticity is the characteristic that helps systems scale quickly with demand, which is useful when handling unpredictable workloads. An autoscaling controller can dynamically acquire and release resources for quick scalability [10]. The application gateways support flexible decoupling of routes from back-end services, which helps implement microservices that can be deployed and updated without disrupting a service.

Global presence: The service providers have data centers on different continents, and with the use of anycast load balancers, customers are directed to the closest location, thus minimizing latency. The importance of keeping a balance when distributing loads to servers is evident in that, it is essential for scalability, which is accomplished using global load balancers [12].

Agility & Lower CapEx Costs: The shift from a CapEx to an OpEx business model helps businesses match expenses with revenues, making it easier for businesses to test new services with reduced risks of significant capital expenses. Cloud service providers remain innovators, providing services such as machine learning, which can be easily added to existing applications.

Security and Compliance: The managed load balancers and application gateways come with Web Application Firewalls (WAF), protection against DDoS, and auto-patching. The inherent security mechanisms, in conjunction with the providers' compliance statements, make it easier for businesses to comply with regulations compared to on-premises configurations. Additionally, auto-scaling helps minimize the attack surface by scaling back the unused instances.

2.5 Serverless Functions vs Containerisation

Serverless computing (Function as a Service) removes the need for infrastructure management, as the developer only must implement a function that reacts to an event, with the infrastructure, scaling, and cost handled by the cloud service provider, as with AWS Lambda, Azure Functions, and Google Cloud Functions. On the other hand, containerization is a technique that packages an application and all its dependencies inside a lightweight container that shares a kernel with other containers. Docker is a champion of this technique, with Kubernetes handling the orchestration of a set of containers.

2.5.1 Serverless (AWS Lambda, Azure Functions)

Functions run on demand because of triggers (HTTP requests, message queues, file uploads). They horizontally scale to handle concurrent requests, with scaling to zero when idle, providing fine-level pricing on a pay-per-use basis. Functions, though, are constrained in terms of runtime as well as memory, hence not useful for tasks that are long-running/resource-intensive. The cold-start problem may pose usability issues, though these are resolved with offered concurrency and upgraded plans. The degree of control offered to developers is limited, with runtime environments/networking specified by the service, though debugging support for distributed functions is limited too. The paradigm supports event-programming use cases such as image processing, real-time transformation, and APIs.

2.5.2 Containers (Docker, Kubernetes)

Containers package applications with libraries, which run on shared hosts. Compared to full virtualization, because containers use the shared host OS, they use fewer resources. The size of a containerized environment is dramatically smaller compared to a hypervisor-managed environment, which can boot quickly[11]. With containers, lightweight virtualization, resource optimization, and CD pipelines are now available for more efficient application development [13]. The Kubernetes environment manages a set of containers efficiently, providing declarative programming, rolling updates, self-healing, as well as horizontal and cluster autoscaling. Containers are best used for stateful applications, long-running microservices, and for scenarios that need different runtime or networking configurations. They cannot scale to zero; hence, basic infrastructure costs are still incurred even when unused.

Comparison and Critical Evaluation

Scalability and performance: Functions are self-healing, with scaling based on burst size, eliminating capacity forecasting. The cold start problem, as well as a runtime limitation, are the major trade-offs. Although containers are close to native performance with support for long-running applications, scaling is via setting up orchestration; recent research on autoscalers brings together reactivity and prediction [14].

Cost structure: The cost structure of serverless computing is fine-grained, which can be very economical for spiky applications. For steady, high-throughput applications, it may be more economical to use containers, which include the cost of a cluster but enable more fine-grained resource utilization.

Maintenance & control: The serverless paradigm hides infrastructure-level tasks, providing quick development but with little possibility of modification. The container solution draws on a deeper stack, imposing a requirement for image upkeep, security updates, and orchestration management. This leads to increased flexibility, with the container running on any supported runtime environment, even on-premises solutions.

Use cases: The use cases for which containers are apt include microservices, web applications, databases, as well as services with low latencies or specialty runtimes. Most applications are hybrids, combining serverless computing with containers. It is also necessary to keep in mind that a balanced solution considers trade-offs. Rule-based autoscaling, although accurate, is a challenge when dealing with uncertainties. Serverless computing handles this problem by delegating this responsibility to a cloud service provider but may be vulnerable to vendor lock-in effects [10]. The need for more infrastructure has been offset by portability and control offered by container platforms. Recent literature on collaborative autoscaling frameworks indicates that a proactive-reactive hybrid can reduce SLA violations [14]

References

- [1] F. Douglass and O. Krieger, "Virtualization," *IEEE Internet Comput.* vol. 17, no. 2, pp.6–9, 2013, doi: 10.1109/MIC.2013.42.
- [2] P. Barham, et al., "Xen and the art of virtualization," vol. 37, no. 5, pp.164–177, 2003, doi: 10.1145/945445.945462.
- [3] R. Uhlig, et al., "Intel virtualization technology," vol. 38, no. 5, pp.48–56, 2005, doi: 10.1109/MC.2005.163.
- [4] T. P. Morgan, "Virtualization in cloud computing," vol. 12, no. 5, pp.14–17, 2010, doi: 10.1109/MITP.2010.86.
- [5] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," no. 239, pp.1–12, 2014, Available at: <https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment>.
- [6] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," vol. 1, no. 3, pp.81–84, 2014, doi: 10.1109/MCC.2014.51.
- [7] W. Felter, et al., "An updated performance comparison of virtual machines and Linux containers", pp. 171–172. [Online].doi: 10.1109/ISPASS.2015.7095802.
- [8] J. Anderson, "Understanding Docker container storage: Union file systems and volumes," vol. 2016, no. 278, pp.1–10, 2016,
- [9] R. Morabito, "Virtualization on IoT edge devices with container technologies: A performance evaluation," vol. 6, pp.544–558, 2018, doi: 10.1109/ACCESS.2017.2778504.
- [10] P. Jamshidi, C. Pahl and N. C. Mendonça, "Managing Uncertainty in Autonomic Cloud Elasticity Controllers," vol. 3, no. 3, pp.50–60, 2016, doi: 10.1109/MCC.2016.57.

- [11] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," vol. 1, no. 3, pp.81–84, 2014, doi: 10.1109/MCC.2014.51.
- [12] Y. Lohumi, D. Gangodkar and P. Srivastava, "Load Balancing in Cloud Environment: A State-of-the-Art Review," vol. 11, pp.1–20, 2023, doi: 10.1109/ACCESS.2023.3291642.
- [13] C. Pahl, A. Brogi, J. Soldani and P. Jamshidi, "Cloud Container Technologies: A State-of-the-Art Review," vol. 7, no. 3, pp.677–692, 2019, doi: 10.1109/TCC.2017.2702586.
- [14] G. Quattrocchi, E. Incerto and C. Pinciroli, "Autoscaling Solutions for Cloud Applications under Dynamic Workloads," 2024,

Video link: <https://youtu.be/21oeM7H5ZGY>

