

Assignment

by Rezaul Haque

Submission date: 11-May-2026 01:11PM (UTC+0100)

Submission ID: 283061057

File name: 67170_Rezaul_Haque_Assignment_726729_1711568819.docx (954.31K)

Word count: 5437

Character count: 30536



Name: Rezaul Haque

ID: 2507164

Demonstration of Video Link: [Cloud Engineering](#)

<https://youtu.be/liQCIHzLKlg>

Table of Contents

Introduction	3
1. Docker, Containerisation and Practical Implementation.	3
1.1 The history of developing Virtualisation into Containerisation.	3
1.2 Docker Architecture: Images, Containers and the Engine.	5
1.3 Dockerfile and Image Construction.	6
1.4 hands-on Implementation: Nginx Web Server on Docker.	8
1.5 Container Management: Volumes, Networks and Inspection.	10
2. Load Balancing, Scalability and Application Gateways.	13
2.1 Load balancing in the cloud environment.	13
2.2 VM Scale Sets and Scalability.	14
2.3 Application Gateways and Layer 7 Routing.	15
2.4 Combining the Three Technologies: Netflix on AWS	15
3. Drivers and Serverless computing.	16
3.1 The Major forces that contribute to the expansion of clouds.	16
3.2 Serverless Computing and Functions as a Service.	17
3.3 Serverless vs Containerisation: A Comparative Analysis.	18
Conclusion	19
References	21

Introduction

The use of the cloud computing technology has fundamentally altered the process of building, deploying and managing software services. The heart of this change is containerisation: a technology that packages applications and their dependencies into portable, isolated units that can be run consistently across any environment. This report analyzes the Docker as the most popular containerisation platform, and a multi-page Nginx web server as an example of its practical implementation. It then examines the three fundamental cloud computing concepts of load balancing, scalability, and application gateways which are applied to real-world deployment at scale. Lastly, it explores the key drivers of cloud adoption and compares the emerging serverless paradigm with containerisation and critically evaluates the areas where each technology most effectively fits.

1. Docker, Containerisation and Practical Implementation.

1.1 The history of developing Virtualisation into Containerisation.

To see what containerisation is, one must see what its evolution of virtualisation was. Virtualisation was developed to address the inefficiency of physical implementations of servers and applications, with a single application taking up an entire machine despite how little of its resources it actually used. Virtualisation enabled the execution of multiple operating system instances at the same time on physically shared infrastructure [1].

The most popular type of virtualisation that was pre-containers was hypervisor-based. Type 1 hypervisors, also known as bare-metal hypervisors, are hypervisors that directly run on a physical computer. This technique was originally devised by IBM in the 1960s with systems like CP/CMS, and modern equivalents include VMware ESXi, Microsoft Hyper-V and Xen [1]. In comparison, type 2 hypervisors are applications running within a host operating system, providing an extra layer of abstraction and the associated performance cost. The major limitation of the two types is that each virtual machine has a complete guest operating system, usually several gigabytes in size, which has its own kernel, system libraries and processes. This overhead is quite acceptable to most workloads, but it grows more and more inefficient as services are decomposed into smaller units.

Bernstein [2] follows up the evolution of Linux Containers (LXC) to Docker to Kubernetes as a logical development in response to this overhead issue. The containerisation model, as outlined in the module lectures as OS-level virtualisation, has the guest OS completely removed. Rather, containers just share the host operating system kernel, but have isolated user-space environments. This isolation is enforced by the kernel using namespaces and control groups (cgroups) to ensure that processes, filesystems and network interfaces are separated between containers without the need to have a separate instance of the operating system in each container [2]. This renders containerisation a very attractive solution to Platform-as-a-Service (PaaS) cloud environments, where the overhead of hypervisor-based VMs would otherwise represent a significant cost burden to lightweight services.

The architectural difference has a significant practical impact. A benchmark study was conducted by Felter et al. [3] comparing the performance of KVM-based virtual machines against Docker containers using CPU, memory, storage and networking workloads. Their findings indicated that containers performed equally or better than VMs in almost all test conditions, with the most significant improvements in workloads that are I/O intensive. This finding is supported by the material covered in the module lecture materials, which identify containers to be typically measured in megabytes, start up almost immediately as compared to the boot time of a VM, and can be packed much more densely on the same physical hardware.

The following table (Table 1 below) summarises the key differences between the hypervisor-based virtual machines and the Docker containers across the dimensions that most significantly apply to the decisions on cloud deployment.

Table 1: Virtual Machines vs Docker Containers.

Feature	Virtual Machine (Hypervisor)	Docker Container
Architecture	Guest OS per instance + Hypervisor layer	Shared host OS kernel, no guest OS
Typical size	Several gigabytes	Tens to hundreds of megabytes
Startup time	Minutes (full OS boot)	Seconds or less

Performance	Near-native with hardware assist; overhead from hypervisor	Equal or better than VMs across CPU, memory, I/O
Isolation	Strong: separate kernel per VM	Process-level: shared kernel, namespace separation
Portability	Limited: tightly coupled to hypervisor and hardware	High: run on any host with compatible Docker Engine
Use case	Full OS isolation, legacy workloads	Cloud-native services, microservices, CI/CD

Sources: Felter et al. [3]; Bernstein [2]; Pahl [5]

1.2 Docker Architecture: Images, Containers and the Engine.

The most widely used containerisation platform is Docker which uses OS-level virtualisation via client-server architecture. The Docker Engine or the Docker Daemon is a set of software that sits between the host operating system and the containers it operates, performing all the container lifecycle operations such as creating, starting, stopping and removing containers [4]. The engine provides an API which the Docker command-line client communicates with, providing users with a uniform interface irrespective of the underlying operating system.

In this architecture, there are two concepts on which it is based and they are images and containers. An image is a read-only template that defines all the requirements to run an application, including the base operating system layer, the application code, its dependencies, configuration files and the command to be executed on startup. Poulton [1] refers to images as the build-time artefact and containers as the runtime instantiation of the artefact. One image can be run with several container instances, each isolated and not connected with others but with the same underlying image layers.

Images are built in layers with each layer comprising the result of a single instruction in the Dockerfile. Such a layered architecture, based on a union filesystem (OverlayFS in contemporary Docker systems) called layers, implies that layers shared across images are stored once only on disk. This fact can be practically proved with references to the picture

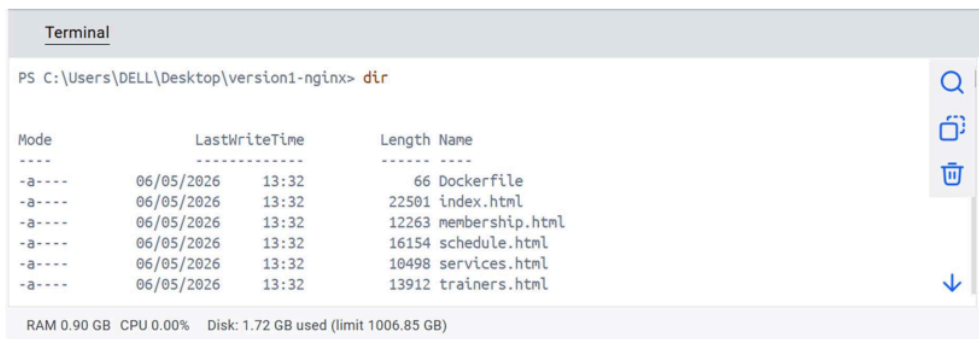
inspection which was conducted during this project. Running `docker image inspect v1-nginx` revealed that the image had eight distinct layers, each with its own SHA-256 hash, with each layer being identified by a metadata containing the base architecture as AMD64 running Linux and the version of Nginx as 1.29.8. The layered approach is not only of importance in terms of storage efficiency, but also with respect to build performance: when an image is rebuilt due to a small change, only the layers that are being changed and those that are downstream of the change require re-execution; upper layers not affected by the change are obtained in the cache.

The containers expand images with a thin layer that people can write on. Any changes that occurred during the container runtime, e.g. log files written or configuration updated, are captured in this writable layer but it is discarded when the container is removed. It is this immutability of the base image layers that gives Docker containers their reproducibility: the same image, when run on any host with a compatible Docker Engine, will yield an identical container environment [4].

1.3 Dockerfile and Image Construction.

A Dockerfile is a text file of plain text containing a sequence of instructions that are executed by Docker in sequence in order to build an image. The instructions add a new layer to the resulting image. The Dockerfile with which the Nginx web server image was built in this project was actually minimal and reflected the minimalistic nature befitting a case of a static web server, namely the Nginx image:

```
FROM nginx:latest  
  
COPY .html /usr/share/nginx/html/  
  
EXPOSE 80
```



```
Terminal
PS C:\Users\DELL\Desktop\version1-nginx> dir

Mode                LastWriteTime         Length Name
----                -
-a-----         06/05/2026    13:32             66 Dockerfile
-a-----         06/05/2026    13:32          22501 index.html
-a-----         06/05/2026    13:32          12263 membership.html
-a-----         06/05/2026    13:32          16154 schedule.html
-a-----         06/05/2026    13:32          10498 services.html
-a-----         06/05/2026    13:32          13912 trainers.html

RAM 0.90 GB  CPU 0.00%  Disk: 1.72 GB used (limit 1006.85 GB)
```

Figure 1: Dockerfile and project folder contents in Windows Explorer

The FROM instruction indicates the base image upon which to construct. Here, nginx:latest offers a ready-to-run Nginx installation with a Debian Linux foundation, which is downloaded via Docker Hub. The COPY command will move all HTML files out of the host build context, and into the Nginx default web root directory within the image. The EXPOSE instruction documents show that the container will listen on port 80, but does not automatically publish that port to the host. Port publishing is not supported at container runtime, but can be activated through the -p flag [4].

This three-instruction Dockerfile exemplifies one of the key principles of container image design: to build an image, it is possible to use a trusted, maintained base image, instead of building everything on your own. The base image of nginx:latest is maintained by the Nginx Docker team and is regularly updated regarding its security. The custom image can be configured to have a fully configured and production-grade web server and the Dockerfile only needs to express what is specific to this application.

The image was constructed by using the docker image build command, docker image build -t v1-nginx and the current directory is specified as the build context by using the dot command. The build output indicated successful completion of all seven build steps, including the pulling of the base image (Docker Hub) and the application of the COPY instruction to add the gym web site HTML files.

```
PS C:\Users\DELL\Desktop\version1-nginx> docker image build -t v1-nginx .
[+] Building 13.9s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 103B
=> [internal] load metadata for docker.io/library/nginx:latest
=> [auth] library/nginx:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load build context
=> => transferring context: 75.55kB
=> [1/2] FROM docker.io/library/nginx:latest@sha256:6e23479198b998e5e25921dff8455837c7636a67111a0
                                docker:desktop-linux 0.1s
                                                                0.0s
                                                                1.9s
                                                                0.0s
                                                                0.0s
                                                                0.0s
                                                                0.1s
                                                                0.0s
                                                                11.2s
```

Figure 2: Docker image build output confirming successful completion

1.4 hands-on Implementation: Nginx Web Server on Docker.

The application that will be deployed in this project is a five-page static web page of a fictional gym, IronForge. The site has a home page, a trainers page, a services page, a day-tab interactive class schedule page, a membership pricing page, and a frequently asked questions accordion page. The one Nginx container serves all five pages, showing how a containerised web server can serve all five pages with no extra configuration beyond the default Nginx set-up.

The container was started using the following command:

```
docker container run -d -p 8080:80 v1-nginx-v1-nginx
```

```
=> => unpacking to docker.io/library/v1-nginx:latest 0.0s
PS C:\Users\DELL\Desktop\version1-nginx> docker container run -d -p 8080:80 --name v1-nginx-container v1-n
576bb7dc9ca9be8d353b7ff49c7b24bb9be16ef5bdaaa076b41241fe7adad061
PS C:\Users\DELL\Desktop\version1-nginx>
```

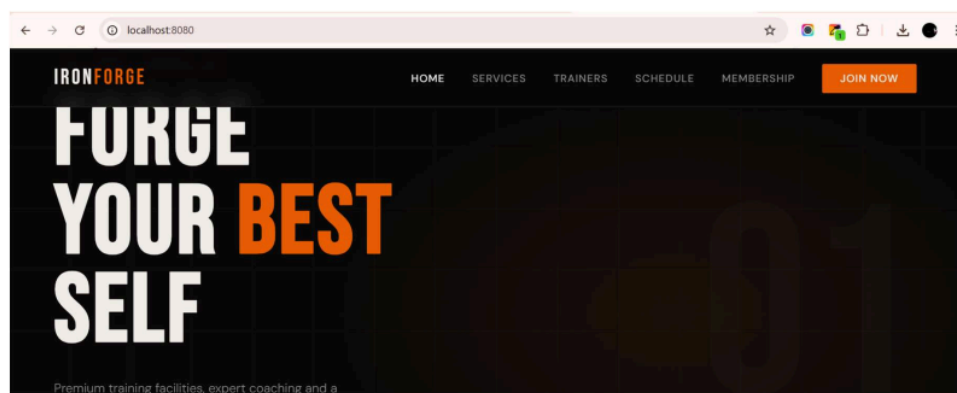


Figure 3: IronForge gym website served by Nginx container

The `-d` flag puts the container in detached mode, i.e., it will be running in the background without holding the terminal. The `-p 8080:80` flag publishes the port 80 within the container to the port 8080 on the host, and the web site will be accessible at the URL of `localhost:8080`. The `name` flag gives the container a human readable identifier which can be used later in the management processes. As the container was running, all the five pages of the IronForge site were verified as being accessible on the browser, and the Nginx access log, accessed using docker container logs `v1-nginx-container`, confirmed the presence of the HTTP 200 response to the root document as well as the browser user-agent string.

The image was then labeled and pushed to an open repository of Dockerhub with the following commands:

```
docker image tag v1-nginx rezaulhaque123/v1-nginx:latest
```

```
docker push rezaulhaque123/v1-nginx:latest
```

Terminal

```
Password:
Login Succeeded
PS C:\Users\DELL\Desktop\version1-nginx> docker image tag v1-nginx rezaulhaque123/v1-nginx:latest
>> docker push rezaulhaque123/v1-nginx:latest
The push refers to repository [docker.io/rezaulhaque123/v1-nginx]
a7dbafce205d: Pushed
4bed64fbc9b1: Pushed
85c66128325a: Layer already exists
677c63196868: Layer already exists
4677c2a9a3d4: Layer already exists

RAM 0.83 GB CPU 0.50% Disk: 1.72 GB used (limit 1006.85 GB)
```

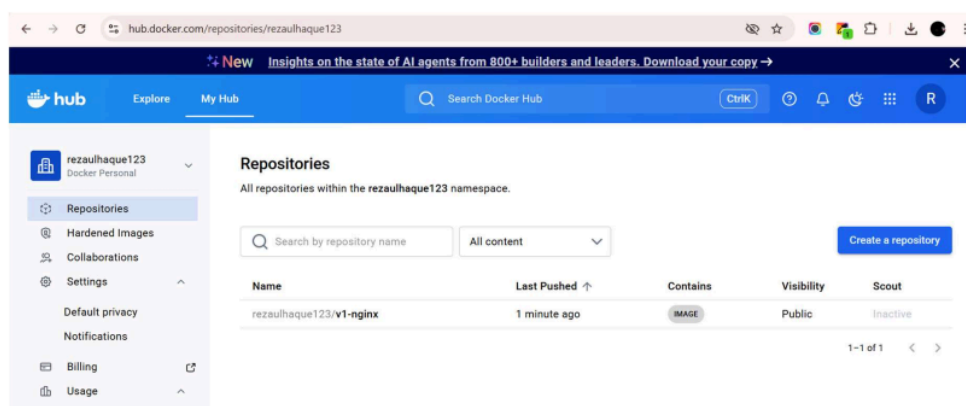


Figure 4: DockerHub repository showing pushed image rezaulhaque123/v1-nginx:latest

The tag command sets the path to the DockerHub repository to the image, and the push command uploads all the eight layers to the remote registry. This is the ship stage of the build-ship-run workflow of Docker [1]. It is now publicly available under the image rezaulhaque123/v1-nginx:latest and can be pulled and run on any host with Docker installed, without any dependency on the original build environment. This mobility is part of the determining commercial benefits of containerisation.

1.5 Container Management: Volumes, Networks and Inspection.

In addition to the core build-run-push workflow, Docker also offers a collection of management primitives that are necessary to comprehend how containers can be used in production-like environments. This project was carried out through the exercise of three of these.

Docker volume create v1-vol and then attaching to a second Nginx container instance using the -v v1-vol:/usr/share/nginx/html flag at run time created a named volume, v1-vol. A docker volume inspect v1-vol run confirmed that the volume was provisioned at /var/lib/docker/volumes/v1-vol/_data on the host filesystem with local scope. Volumes decouple persistent data of the container lifecycle: even when the container is stopped and removed, the data written to the volume is still present and can be accessed by any other container that subsequently mounts the same volume.

The custom bridge network v1-net was built with the help of docker network create v1-net. The main container was then linked to this network with the help of docker network connect v1-net v1-nginx-container, and docker network inspect v1-net, which confirmed the connection, showing the container assigned the IP address 172.20.0.0.2 within the 172.20.0.0/16 subnet. Bridge networks defined by users give automatic service discovery between containers on the same network: another container on the same network can reach it by its container name but not by its IP address.

Lastly, the docker container inspect v1-nginx-container returned all the JSON metadata of the running container, including its network configuration, port bindings, environment variables, the SHA-256 digest of the image it was created using, and its internal state confirming a running state with PID 2186. Such level of introspection is useful not only in operational troubleshooting but also in ensuring that a container is running precisely the image and configuration intended.

This project includes the original diagram of the Docker architecture as implemented in this project, as shown by the relationship between the host OS, Docker Engine, image layers and the running container.

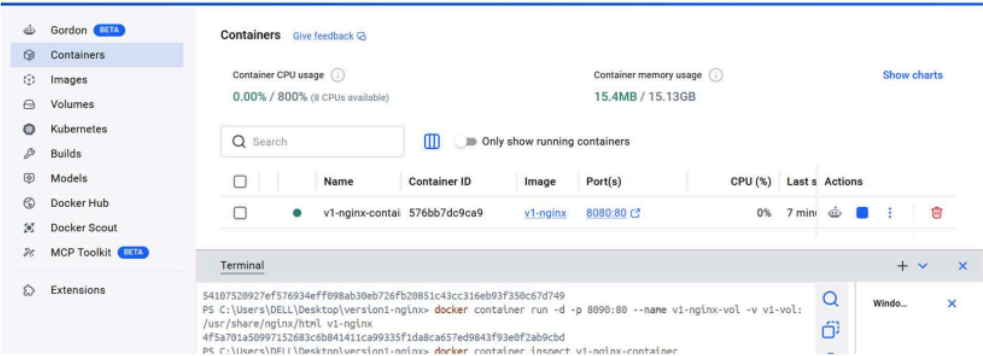


Figure 5: Docker Desktop Containers view showing v1-nginx-container running

```
41 38/0183077/132003000414118773331 10806803/00707431730012807C00
PS C:\Users\DELL\Desktop\version1-nginx> docker container inspect v1-nginx-container
>> docker container logs v1-nginx-container
>> docker image inspect v1-nginx
>> docker volume inspect v1-vol
>> docker network inspect v1-net
[
{
```

Figure 6: docker volume inspect and docker network inspect outputs

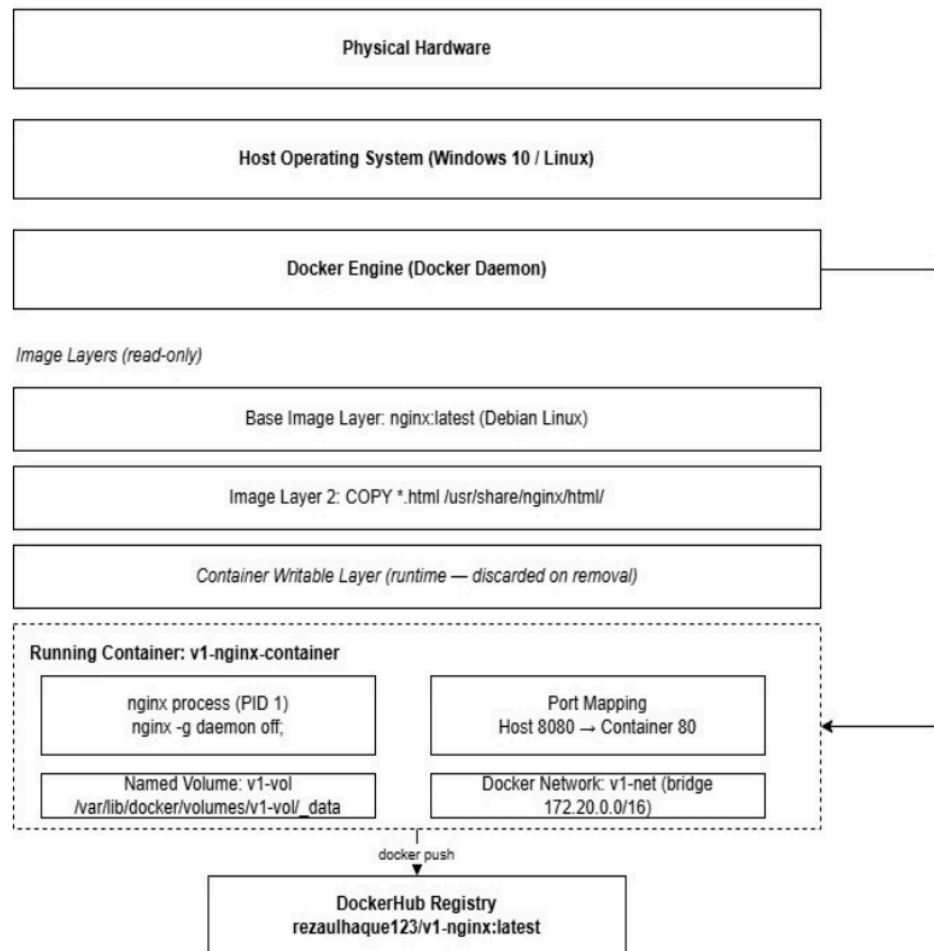


Figure 7: Docker Architecture - Host OS to Running Container.

2. Load Balancing, Scalability and Application Gateways.

2.1 Load balancing in the cloud environment.

Load balancing refers to the process of spreading the incoming network traffic across a pool of back-end resources in such a way that no single server is overworked with a disproportionate share of the workload. It is not just an optimisation of performance but an essential availability mechanism in cloud environments: When one of the backend servers fails, a properly configured load balancer notices the failure and redirects the traffic to the remaining healthy instances, ensuring that a single point of failure does not cause a service outage [6].

Azure load balancer uses the IP address and port combination to make a routing decision instead of making a routing decision based on the contents of the request itself. Its key components include the frontend IP address pool, backend address pool, load balancing rules and health probes [6]. The public facing endpoint is determined by the frontend IP address pool which defines the publicly facing endpoint to which clients can connect. The network interface cards (NICs) of the VMs that will process the traffic are contained in the backend address pool. A load balancing rule is a combination of a frontend IP and port to a backend pool and a backend port that defines how traffic is passed through the balancer. Health probes periodically submit test requests to each untagged VM; any untagged VM that fails to respond is automatically dropped out of the pool until it recovers, ensuring that traffic is only routed to running instances.

A Standard Load Balancer was provisioned in the Azure CLI and configured with three VMs in its backend pool in the Azure tutorials completed as part of this module. The load balancing rule redirected TCP packets coming on port 80 of the public frontend IP to port 80 on each backup VM. The high availability behaviour of the load balancer was observed to work in practice by deallocating individual VMs and observing that traffic continued to be served by the remaining instances. The default distribution algorithm implemented by the Azure Load Balancer is a hash-based algorithm, based on a five-tuple of the source IP, source port, destination IP, destination port and protocol. This makes packets belonging to the same TCP session always reach the same instance of the backend, avoiding any connection state problems [6].

2.2 VM Scale Sets and Scalability.

Whereas load balancing divides traffic among a fixed group of servers, scalability deals with the dynamic issue of how to match infrastructure capacity to the actions of a changing demand. Azure Virtual Machine Scale Sets (VMSS) is the ability of an organisation to automatically add or remove VM instances within a group based on specified metrics or schedule [8]. The scale set is configured to use the same base image to create each VM in the scale set, and the scale set is configured to directly integrate with Azure Load Balancer to ensure that newly provisioned VMs are automatically added to the backend pool and begins receiving traffic without manual intervention.

Scaling in a VMSS can be done in two ways: manual and automatic scaling. Manual scaling sets the number of instances in a set by directly commanding the `az vmss scale` command, as shown in the Tutorial 5 worksheet where the `az vmss scale` command was issued to scale the set out to three instances and then back to two. The automatic scaling mechanism is based on the Azure Monitor autoscale rules, which will generate scaling events depending on the real-time measurements. In the tutorial setup, a scaling out rule was defined, to scale out by two VMs when average CPU utilisation increased to more than 40% in a five-minute period and a scaling back rule was defined, to scale back by one VM when average CPU utilisation decreased to less than 30% in a five-minute period. The five-minute minimum window is a designed protection against responding to transient spikes which will otherwise incur unnecessary and expensive provisioning cycles [8].

Two types of scaling should be differentiated. Scale-out (sometimes called horizontal scaling) scales up to more instances of the same resource to handle more load; that is what VMSS does. Scale-up (also known as vertical scaling) is the process of scaling the capacity of an existing instance by moving the instance to a larger VM size. Horizontal scaling is highly desired in cloud architecture since it is possible to perform without downtime and can theoretically be extended to extremely large numbers of instances, whereas vertical scaling is limited by the largest available VM size and theoretically can be extended to very large numbers of instances. Load balancing and horizontal scaling are thus complementary to each other: the load balancer is the mechanism that provides the means of distributing the traffic across the however many instances there are, whereas the scale set is the mechanism that provides the means of changing that number dynamically in response to demand.

2.3 Application Gateways and Layer 7 Routing.

Azure Application Gateway builds on the features of a load balancer by running on Level 7, the application layer, of the OSI model. Where the Azure Load Balancer can work with raw network packets with IP addresses and ports, the Application Gateway is able to inspect the full HTTP or HTTPS request, including the URL path, headers and session cookies [7]. This ability allows a type of routing decisions that cannot be done at Layer 4.

The most important of these is URL path-based routing. The Application Gateway may be programmed with an URL path map which determines the backend pools to be used by the incoming request based on the path segment of the URL. This was done in the Tutorial 6 worksheet by configuring an Application Gateway to direct requests to /sales/ to a pool of VMs, requests to /orders/ to another pool of VMs, and all other traffic to a default pool. Each pool is a different collection of backend servers possibly optimised to the local workload it serves. The architecture helps organisations to specialize their backend infrastructure: image-serving endpoints can be deployed on storage-optimised VMs, API endpoints on compute-optimised VMs, and all this behind a single public-facing endpoint in the form of a gateway.

The internal workflow of the Application Gateway consists of four main components: a frontend IP on which the incoming connections are received, a listener on which the incoming traffic on a given port and protocol is monitored, a request routing rule that evaluates the listener traffic and applies path-based logic, and HTTP Settings, which define how the gateway opens a new TCP connection to the selected backend pool [7]. One important point in the implementation of the tutorial is that the frontend protocol and port may not match the backend protocol and port. In the tutorial, inbound HTTPS traffic on port 8080 was proxied to plain HTTP on port 80 at the backend, allowing the Application Gateway itself to handle TLS termination, and forward plaintext traffic to VMs that did not need to manage certificates independently.

2.4 Combining the Three Technologies: Netflix on AWS

The complementary nature of load balancing, scalability and application gateway routing comes into focus most clearly when it comes to looking at real world deployment at scale. The most common example of these three technologies working in a production setting as cited by Netflix. Being a streaming service with a massive demand in terms of size and extremely

dynamic in terms of the timing of the day, the day of the week and the schedule of releases of the content they provide, Netflix has a demand that is not only massive in scale, but also highly variable by both the time of day and the day of the week, as well as the timing of the release of the content they provide.

Netflix is a streaming service that runs on Amazon Web Services and uses Elastic Load Balancing (ELB) to allocate incoming streaming requests to fleets of EC2 instances. Functionally equivalent to Azure VM Scale Sets, EC2 Auto Scaling Groups automatically add more streaming server instances during peak demand periods such as evening viewing windows, and scale back in during off-peak periods to keep the cost down. Distributed on the boundary of the Netflix microservices architecture, API Gateway does the routing of incoming requests to the correct backend service based on the requested path, redirecting /streaming traffic to video delivery services and /account requests to account management services. This separation of concerns, which is made at the gateway level, enables Netflix to scale each backend service independently based on its demand profile as opposed to treating the entire application as a single unit. All three technologies coming together provide the high availability, elastic capacity and intelligent routing that a service of the scale of Netflix needs [9].

3. Drivers and Serverless computing.

3.1 The Major forces that contribute to the expansion of clouds.

The development of cloud computing has not been a factor in itself but a combination of economic, operational and technological pressures that collectively made the traditional model of owning and managing physical infrastructure an increasingly difficult aspect to justify. According to Mell and Grance [13], cloud computing can be defined as a model which enables on-demand network access to a common pool of configurable computing resources which can be provisioned and released rapidly and with minimum management effort. This definition reflects the radical change in the way organisations think about infrastructure: not as a capital asset that should be purchased and maintained, but as a utility that should be consumed on demand.

Cost is the most immediate driver that is compelling. The old on-premise infrastructure involves huge initial capital investment in servers, storage, networking equipment and the

physical facility to host them. Most enterprise server industry estimates show that most organisations are actually paying to use resources they are not using [13]. Cloud computing substitutes this with a pay as you go system where organisations are only charged on what they consume. The common infrastructure of a cloud provider realises economies of scale that no single organisation could match, and the savings are willed out in the form of competitive pricing.

In addition to cost, flexibility and agility are also very important drivers. The fact that a new virtual machine, container instance or serverless function can be provisioned within seconds, as opposed to weeks, months or even years to procure and configure physical hardware, fundamentally changes the speed with which organisations can respond to opportunity or demand. According to Erlang et al. [11], this elasticity is one of the distinguishing features of cloud computing that differentiated it in relation to the previous models of out-sourced hosting. Scalability, horizontally and vertically, enables organisations to precisely match their infrastructure footprint to the current demand. The increase in big data workloads, mobile computing and geographically dispersed infrastructure needs has further hastened the adoption of the cloud, as these applications require the type of distributed, geographically dispersed infrastructure that only major cloud providers can offer at reasonable cost.

3.2 Serverless Computing and Functions as a Service.

Serverless computing, also known as Functions as a Service (FaaS), is yet another abstraction of the containers in the cloud computing model. A serverless architecture uses individual functions that are written by the developer and uploaded to the cloud platform where they are executed in response to specified trigger events. The three most popular implementations are AWS Lambda, Microsoft Azure Functions and Google Cloud Functions. The cloud provider is fully charged with provisioning, scaling and managing the underlying execution environment, and the developer is only charged the actual time of function execution in milliseconds as opposed to the time a server or container is allocated [12].

According to Castro et al. [12], the serverless model is event driven in nature: a function may have its invocation triggered by an HTTP request, a message in a queue, the uploading of a file to object storage, or a scheduled timer. Any invocation is independent and stateless. The platform automatically scales the number of concurrent function executions to match the

incoming demand, zero invocations when there is no demand, and thousands of parallel executions when the demand peaks, without any manual configuration of scaling rules. This automatic and granular scaling is a valuable enhancement to the cost efficiency argument in favour of cloud adoption: organisations are paying nothing when their functions are not being called, which is particularly beneficial in the case of workloads with intermittent or unpredictable patterns.

3.3 Serverless vs Containerisation: A Comparative Analysis.

The introduction of serverless computing does not make containerisation obsolete, but instead places the two approaches at different ends of the control versus operational simplicity spectrum, and each better suited to workloads of a particular type.

A critical review by Hellerstein et al. [10] of first-generation serverless computing argues that although the autoscaling and billing model is truly innovative, current implementations introduce constraints that make them unsuitable to a broad range of applications. The greatest of these is the stateless execution model. Since each function invocation is independently executed and the execution environment can be destroyed between invocations, serverless functions cannot maintain in-memory state between invocations. Applications with persistent connections, a state of a session or communication between concurrent instances have real architectural issues in a serverless model. A web app that delivers a gym membership site like the one deployed in this project, which needs a consistent routing and predictable response time, is not a good fit to use serverless execution. Running the same application as a containerised instance of Nginx as shown in Section 1, provides complete control of the runtime environment, has deterministic behaviour and has no cold start latency.

Containers, in contrast, are long-running processes that retain their state and configuration as long as they are running. They provide predictable and consistent performance since the execution environment is not altered between the requests. They also offer significantly more flexibility in what can be deployed: any application that can be packaged into a container image, regardless of language, runtime or dependency needs are all capable of being

containerised whereas serverless platforms place language and runtime constraints and have execution time limits precluding long running processes.

Operational overhead is the trade-off. Containers involve the developer managing the Dockerfile, the image build process, the container runtime configuration, volumes, networks and the orchestration platform in case of running at scale. None of this is needed in serverless functions: an author writes a function, deploys it, and the platform manages everything else. When dealing with simple, stateless, event-driven workloads, such as image resizing, email notifications or webhook processing, serverless is obviously the less complex and more cost-efficient option. Containers are still the more suitable tool to use in complex, stateful applications that need fine-grained control over the execution environment.

Practically, more advanced cloud architectures are often a mix of both: containers are used to run the core application services, and serverless functions are used to run asynchronous, event-driven tasks at the edges of the system. The best architectural choices have realised this complementarity instead of viewing the two approaches as mutually exclusive [10].

Conclusion

This report has discussed the concept of containerisation as a useful technology as well as an evolutionary stage in the history of virtualisation. By deploying a multi-page Nginx web server using Docker, it has demonstrated the entire process of containerisation of applications running on Nginx starting with the Dockerfile creation, the image construction, the execution of the container, the management of the volume and management of the network, and publication of the image to the registry on DockerHub. The comparison against the hypervisor-based virtualisation reveals that containers provide an attractive combination of performance, portability and resource efficiency that makes them the right choice to use in the modern cloud-native web services.

The discussion on the load balancing, scaling and application gateway capabilities of Azure illustrates how the three technologies work together as an integrated system, each of which addresses a different aspect of the production-scale cloud deployment. Load balancing ensures availability and equal distribution of traffic; VM Scale Sets are elastic horizontal scaling; and the Application Gateway is intelligent Layer 7 routing that enables bespoke backend

architectures. Collectively, as the case of Netflix shows, they constitute the backbone of infrastructure of services running at global scale.

The analysis of serverless computing shows that it is a strong yet contextually oriented solution. Its advantages of cost efficiency and operational simplicity are real but the stateless model of execution and the absence of runtime control makes it unsuitable to the class of persistent, stateful web services that containers are managing with confidence. The best cloud architectures are based on both paradigms, with each being applied where its strengths are most likely to be best matched to the workload at hand.

References

- [1] N. Poulton, Docker Deep Dive, 2023 ed. Self-published, 2023. ISBN 978-1521822807.
- [2] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014. doi: 10.1109/MCC.2014.51.
- [3] W. Felter, A. Ferreira, R. Rajamony and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in Proc. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 2015, pp. 171-172. doi: 10.1109/ISPASS.2015.7095802.
- [4] Docker Inc., "Dockerfile reference," Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/reference/dockerfile/> [Accessed: 6 May 2026].
- [5] C. Pahl, "Containerisation and the PaaS Cloud," IEEE Cloud Computing, vol. 2, no. 3, pp. 24-31, 2015. ISSN 2325-6095.
- [6] Microsoft, "What is Azure Load Balancer?," Microsoft Azure Documentation, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview> [Accessed: 6 May 2026].
- [7] Microsoft, "What is Azure Application Gateway?," Microsoft Azure Documentation, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/overview> [Accessed: 6 May 2026].
- [8] Microsoft, "What are Virtual Machine Scale Sets?," Microsoft Azure Documentation, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/overview> [Accessed: 6 May 2026].
- [9] A. Choudhary et al., "Issues and Challenges of Load Balancing Techniques in Cloud Computing: A Survey," ACM Computing Surveys, vol. 51, no. 6, pp. 1-35, 2019. doi: 10.1145/3281010.
- [10] J. M. Hellerstein et al., "Serverless Computing: One Step Forward, Two Steps Back," in Proc. 9th Biennial Conference on Innovative Data Systems Research (CIDR 2019), Asilomar, CA, USA, Jan. 2019. [Online]. Available: <http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf> [Accessed: 6 May 2026].

- [11] T. Erl, R. Puttini and Z. Mahmood, Cloud Computing: Concepts, Technology and Architecture. Upper Saddle River, NJ: Prentice Hall, 2013. ISBN 978-0133387520.
- [12] P. Castro, V. Ishakian, V. Muthusamy and A. Slominski, "The Rise of Serverless Computing," Communications of the ACM, vol. 62, no. 12, pp. 44-54, Dec. 2019. doi: 10.1145/3368454.
- [13] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Special Publication 800-145, 2011. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf> [Accessed: 6 May 2026].

Assignment

GRADEMARK REPORT

FINAL GRADE

75 / 100

GENERAL COMMENTS

This is a well-structured and technically strong submission that covers all required areas with consistent depth. The Docker section is well executed - the Dockerfile is shown and each instruction explained clearly, the build-ship-run workflow is followed through to a DockerHub push, and the volume and network demonstrations include specific inspected outputs (mount points, subnet, container IP, PID) that demonstrate good engagement. The use of a five-page gym website as the practical application gives the work a coherent thread. The discussion of image layering, OverlayFS, and the union filesystem goes beyond the surface level, and the VM versus container comparison table is a useful addition. Academic referencing is strong, drawing on several well known sources alongside official Docker documentation, and citations are applied consistently throughout.

The cloud concepts section is detailed, covering the five-tuple hash distribution algorithm in Azure Load Balancer, autoscale thresholds and the rationale for the five-minute cooldown window in VMSS, and TLS termination and URL path-based routing in Application Gateway, all grounded in tutorial work. The Netflix case study is a reasonable way to tie all three together, though the citation used to support it does not obviously correspond to Netflix specifically. The serverless versus containerisation comparison is evaluative, not just descriptive, engaging critically with constraints on first-generation serverless and making a defensible argument for hybrid architectures. The driving forces section is competent, though it stays at a relatively high level and could engage more critically with the literature beyond cost and agility. The Dockerfile itself is minimal (three instructions), which is appropriate for the use case but leaves little room to demonstrate awareness of more advanced image design considerations such as layer ordering for cache efficiency or non-root user configuration.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22
