

ACCA7027 - ZE 1908686 - Assignment 1

by Zack Evans

Submission date: 10-May-2026 05:30PM (UTC+0100)

Submission ID: 282946725

File name: 18253_Zack_Evans_ACCA7027_-_ZE_1908686_-_Assignment_1_726729_126931761.docx (1.13M)

Word count: 4264

Character count: 24928



ACCA7027 - Cloud Engineering Principles and Practice

Assignment Video - <https://youtu.be/VAkEeyz0iM4>

ACCA7027 - Cloud Engineering Principles and Practice	1
1. Virtualisation and Containerisation in Cloud Computing	4
1.1 Introduction	4
1.2 Principles of Virtualisation	4
1.2.1 Type 1 & 2 Virtualisation	5
Type 1 (Bare-Metal)	6
Type 2 (Hosted)	7
1.2.2 Example: Setting up a virtualised cloud instance	7
1.3 Principles and Benefits of Containerisation	10
1.3.1 Example: Demonstrating the Construction of a Dockerfile	11
1.4 Images and Publishing Images	12
1.4.1 Example: Building and Pushing an Image	14
1.5 Container Volumes	15
1.5.1 Example: Creating a Docker Volume	15
1.6 Container Networking	16
1.6.1 Example: Creating and Using a Docker Network	17
2. Main Cloud Concepts and Cloud Computing Expansion	19
2.1 Cloud Computing Concepts in Web Applications	19
2.1.1 Load Balancing	19
2.1.2 Scalability	20
Vertical Scaling	20
Horizontal Scaling	20
Scale Sets / Autoscaling	20
2.1.3 Application Gateways	21
2.2 Interplay Between Load Balancing and Scalability	22
2.3 Benefits of Combining All Three Technologies	23
2.4 Driving Forces Behind Cloud Computing Expansion and the Role of Serverless vs. Containerisation	24
2.4.1 The key driving forces	24

Cost efficiency	24
Scalability	24
2.4.2 Serverless Computing	24
Serverless Functions	25
Comparing against Containerisation.....	25
Flexibility	25
Scalability.....	26
Operational Efficiency	26
3. References	27

1. Virtualisation and Containerisation in Cloud Computing

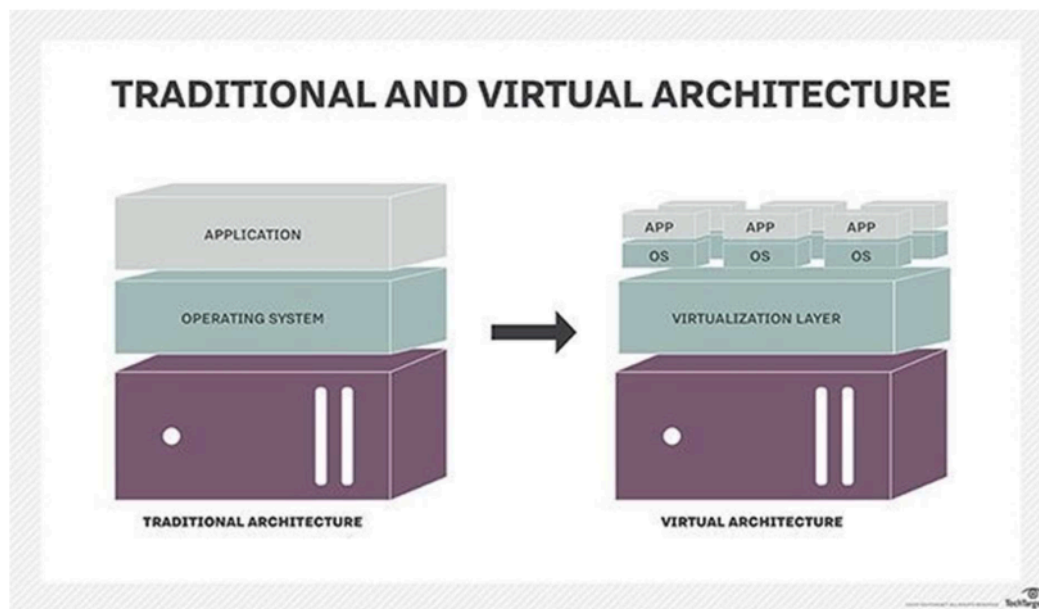
1.1 Introduction

The concept of cloud computing has transformed how modern applications are developed, maintained and deployed. The main driving factor behind this is the move from monolithic applications to microservices and containerisation [1]. Containerisation technologies like Docker and Podman enable applications to be scalable as well as portable across different environments [2]. In this section of the report, principles like virtualisation and containerisation will be explored, and why they are important when it comes to the cloud and how we can implement this in a real world scenario when it comes to web applications.

1.2 Principles of Virtualisation

Virtualisation refers to the process of creating a software simulated representation of a physical computer otherwise known as a virtual machine or VMs [3]. To implement virtualisation, we have to move away from traditional systems that use the system of **machine > operating system > application**.

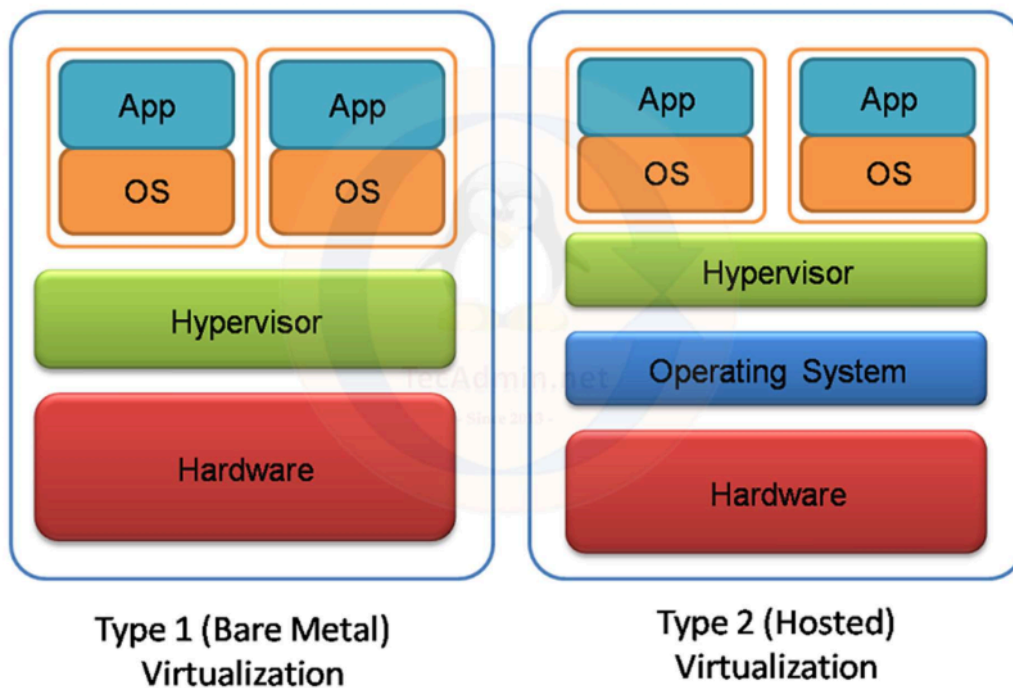
To do this, another layer is added just after the hardware layer, called the virtualisation layer. This allows the creation of several VMs within a single machine.



(<https://stl.tech/wp-content/uploads/2022/01/virtualization2.jpg>)

1.2.1 Type 1 & 2 Virtualisation

When it comes to virtualisation, you can opt for two types. Type 1 also known as bare-metal, due to it being installed directly onto the machine. While Type 2 virtualisation is installed after the operating system layer. Both have their own advantages and disadvantages [4].



<https://tecadmin.net/wp-content/uploads/2023/09/type-1-vs-type-2-virtualization.png>

Type 1 (Bare-Metal)

This can run directly on the physical hardware without any preexisting host operating system. This allows the layer to work directly between the hardware and VMs.

Such examples that are widely used:

- VMware ESXi
- Microsoft Hyper-V
- Proxmox VE (KVM)
- Linux KVM

Due to the architecture of the Type 1 hypervisor. This allows the layer to have direct access to the hardware, leading to high performance and efficiency due to minimal overhead compared to Type 2 hypervisors that rely on a preexisting operating system. Additionally, it also helps to provide a strong isolation between virtual machines, ensuring that each VM runs independently and securely without interacting with each other. Making it an ideal choice for cloud environments. However, there are some drawbacks when it comes down

to this hypervisor. If you're setting such a system up locally instead of relying on preexisting hosting. The setup can be rather complex, and requires dedicated hardware.

Type 2 (Hosted)

These types of hypervisors can run on top of an existing operating system such as Windows, macOS or Linux. It acts like any other application that is installed on the system, meaning it relies on the host OS to access hardware.

Such examples that are widely used:

- VirtualBox
- VMware Workstation
- Parallels Desktop
- QEMU

Due to it being installed on top of an preexisting OS. It makes installing and setting up more accessible and simplistic compared to Type 1 virtualisation software. This makes it ideal for development environments. It also doesn't require dedicated hardware, helping to keep costs down. However, performance is generally lower due to the additional overhead and there can be increased security risks since the overall system security is tied to the host OS.

1.2.2 Example: Setting up a virtualised cloud instance

For the web application, an azure cloud instance is created to host the docker container containing the web application. To do this:

- A resource group will need to be created
- Creating a virtual machine
- Accessing the virtual machine through SSH

First a resource group will need to be created. To do this, it can be done either by the web UI or azure console. For this tutorial, this will be done via the web UI.

Create a resource group ...

Basics

Tags

Review + create

Resource group - A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. [Learn more](#)

Subscription * ⓘ

Azure for Students

Resource group name * ⓘ

acca7027-assignment

Region * ⓘ

(Europe) France Central

Under the report, the resource group called “acca7027-assignment” was created with a simple name to easily identify where we will be creating the instances.

This then allows Azure users to create new instances under the given resource group. With this example, an instance that has 2vCPUs and 1GB RAM can be created.

Create a virtual machine

...



[Help me choose the right VM](#)

✓ Validation passed

Basics

Subscription	Azure for Students
Resource group	acca7027-assignment
Virtual machine name	assignment-1
Region	France Central
Availability options	Availability zone
Zone options	Self-selected zone
Availability zone	2
Security type	Trusted launch virtual machines
Enable secure boot	Yes
Enable vTPM	Yes
Integrity monitoring	No
Image	Ubuntu Server 24.04 LTS - Gen2
VM architecture	x64
Size	Standard B2ats v2 (2 vcpus, 1 GiB memory)
Enable Hibernation	No
Authentication type	SSH public key
Username	zack
SSH Key format	Ed25519
Public inbound ports	SSH
Azure Spot	No

```
[zack@fedora ~]$ ssh -i .ssh/azure zack@20.199.73.170
Welcome to Ubuntu 24.04.4 LTS (GNU/Linux 6.17.0-1013-azure x86_64)

* Documentation:  https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:       https://ubuntu.com/pro

System information as of Sun May 10 11:54:13 UTC 2026

System load:  0.75          Processes:           140
Usage of /:   5.4% of 28.02GB Users logged in:       0
Memory usage: 35%          IPv4 address for eth0: 10.0.0.4
Swap usage:   0%

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

Last login: Sun May 10 11:54:15 2026 from 82.9.73.114
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

zack@assignment-1:~$
```

1.3 Principles and Benefits of Containerisation

Containerisation is a different approach when it comes to isolating computing resources. It is done by grabbing an application with its runtime, dependencies and required system libraries and then packaged into an image / container. Containers are designed to run consistently across different environments such as local development machines to cloud computing. A widely used platform for containerisation is Docker [5].

A key principle that is behind containerisation is the operating system sharing. Unlike traditional virtualisation where each VM has its own OS, containers share the host OS. This leads to a reduction in overhead as well as making them start up quicker.

Another principle is isolation. Containers use Linux kernel features such as namespaces to create separate environments. This ensures that the processes, networks and file systems

within one container cannot affect another. In addition, control groups (cgroups) manage and limit the resources (CPU, memory, and storage) that each container can use, preventing any single container from consuming excessive resources [6].

In practice, containerisation follows a simple lifecycle:

- Build: A configuration file defines the application and its environment.
- Ship: The application and dependencies are packaged into an image.
- Run: The image is executed as a container by a runtime environment.

There are many advantages in using containerisation:

- Portability: This ensures that containers behave the same regardless of the instance they are deployed on.
- Efficient: Multiple containers can run on a single instance, allowing the use of shared resources.
- Speed: Due to Docker using the host OS. Containers can boot up quickly.
- Support Modern Practices: With the way that Docker is designed. It helps with the development and implementation of microservices.
- Scaling: With Kubernetes (k8s). Docker can scale to demand across several instances, allowing services to respond to traffic surges.

1.3.1 Example: Demonstrating the Construction of a Dockerfile

A Dockerfile is a text based script used to define how a container image is built. It specifies the base image, application code, dependencies, and commands required to run the application.

Below is a simple example of a Dockerfile for deploying a basic NGINX web server:

```
FROM almalinux:9

RUN dnf -y update \
    && dnf -y install nginx \
    && dnf clean all \
    && rm -rf /var/cache/dnf

WORKDIR /usr/share/nginx/html
```

```
COPY index.html ./index.html

EXPOSE 80

CMD ["/usr/sbin/nginx", "-g", "daemon off;"]
```

Explanation:

- **FROM** defines the base image (alma linux).
- **RUN** installs dependencies.
- **WORKDIR** sets the working directory inside the container.
- **COPY** transfers application files into the container.
- **CMD** specifies the command that runs when the container starts.

This Dockerfile demonstrates how an application can be packaged into a portable image. Once built, the image can be deployed and run consistently across any environment that supports container runtimes such as Docker.

```
docker build -t nginx-almalinux:latest . && docker run --rm -p 80:80  
nginx-almalinux:latest
```

This command allows the image to be built and deploys the image into a container to allow access to the NGINX server.

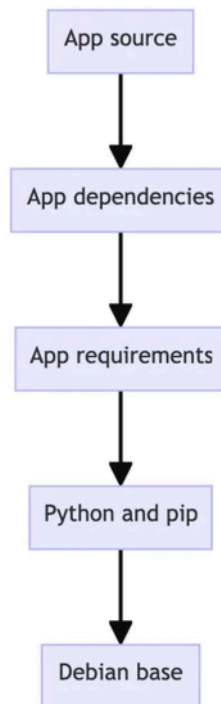
1.4 Images and Publishing Images

A Docker image is a template used to create containers. Once built, images can be stored and shared via repositories such as Docker Hub [6]. Containers are also built using layered images. These work by layering different images on top of one another that perform their individual tasks such as:

1. The first layer adds basic commands and a package manager, such as apt.
2. The second layer installs a Python runtime and pip for dependency management.
3. The third layer copies in an application's specific requirements.txt file.
4. The fourth layer installs that application's specific dependencies.
5. The fifth layer copies in the actual source code of the application.

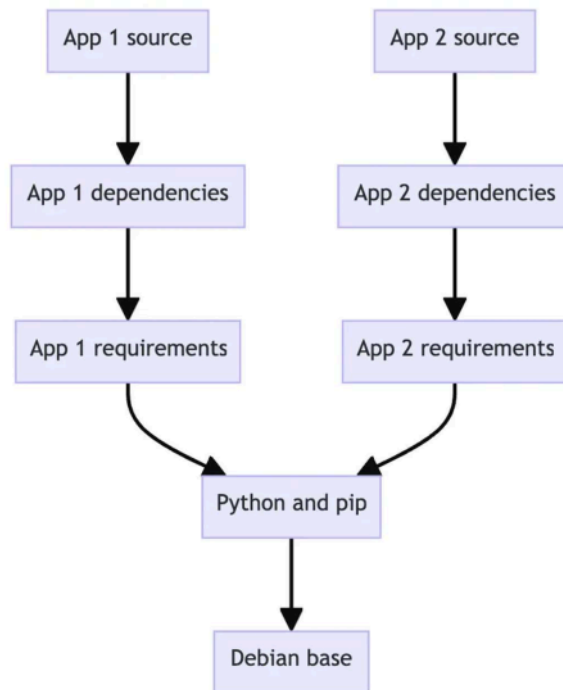
This example might look like:

This example might look like:



(docker)

“This is beneficial because it allows layers to be reused between images. For example, imagine you wanted to create another Python application. Due to layering, you can leverage the same Python base. This will make builds faster and reduce the amount of storage and bandwidth required to distribute the images.” [7] The image layering might look similar to the following:



Layers let you extend images of others by reusing their base layers, allowing you to add only the data that your application needs.

1.4.1 Example: Building and Pushing an Image

First, the docker image needs to be built.

```
docker build -t nginx-almalinux:latest .
```

After it has been built, authentication between Docker Hub and local device needs to be authenticated. To do this:

```
docker login
```

Once authenticated, the image needs to be tagged to identify it.

```
docker tag nginx-almalinux:latest <username>/nginx-almalinux:latest
```

Once applied with the correct username, the image can then be uploaded to Docker Hub.

```
docker push ziggyyy/nginx-almalinux:latest
```

Benefits:

- Easy sharing and collaboration
- Version control of application environments
- Deployment consistency

1.5 Container Volumes

Volumes are persistent data stores for containers. They ensure that data is retained even after containers are stopped, restarted or even deleted. This makes them essential for applications that require reliable data management beyond the lifecycle of the single container / version [8].

Docker volumes are widely used in scenarios such as:

- Database storage: Ensuring that database data is preserved across container restarts or updates.
- Log persistence: Retaining application logs for monitoring, debugging, or auditing purposes.
- Shared data between containers: Allowing multiple containers to access and modify the same data volume.

By having the volumes being decoupled from containers, this enhances the reliability, portability and scalability in containerised applications.

1.5.1 Example: Creating a Docker Volume

Inside the official NGINX image, the default site files live at: `/usr/share/nginx/html`

That's the path that the volume will be attached to.

Create a named volume:

```
docker volume create nginx-data
```

List volumes:

```
docker volume ls
```

Inspect it:

```
docker volume inspect nginx-data
```

Create an nginx container and attach created volume:

```
docker run --name my-nginx -p 80:80 -v nginx-  
data:/usr/share/nginx/html nginx
```

Inspect container to verify that volume is attached:

```
docker inspect my-nginx
```

Now, the default html can be changed and saved:

```
docker exec -it my-nginx sh  
echo "<h1>Hello from Docker Volume</h1>" >  
/usr/share/nginx/html/index.html  
exit
```

Refreshing the page should show the changed html.

Now to prove the data survives container deletion. To remove the container:

```
docker rm -f my-nginx
```

Now start a new NGINX container using the same volume:

```
docker run --name my-nginx -p 80:80 -v nginx-  
data:/usr/share/nginx/html nginx
```

1.6 Container Networking

The in-built networking tools that are provided by Docker enable containers to have their own isolated network to communicate with each other. The networking tools allow the deployment of distributed systems such as microservices and managing communication between them.

Docker supports multiple network drivers to accommodate different use cases [14]:

- Bridge (default): Suitable for standalone containers on a single host. It provides automatic DNS resolution and internal isolation.
- Host: Removes network isolation by allowing containers to share the host's network stack, offering improved performance but reduced security.
- Overlay: Enables communication across multiple Docker hosts, commonly used in clustered environments such as Docker Swarm.
- Ipvlan: This gives users total control over both IPv4 and IPv6 addressing. The VLAN driver builds on top of that in giving operators complete control of layer 2 VLAN tagging and even Ipvlan L3 routing for users interested in underlay network integration.

- Macvlan: Assigns containers their own IP addresses on the physical network, making them appear as separate devices.

Docker networking offers several key advantages:

- Service isolation: Containers can be grouped into separate networks to segment application components and limit exposure.
- Secure communication: Network boundaries and configurations help control traffic flow and reduce the attack surface.
- Scalability: Networks make it easier to scale services horizontally while maintaining consistent communication patterns.
- Support for microservices architecture: Facilitates interaction between independently deployed services in a modular system.

1.6.1 Example: Creating and Using a Docker Network

To create a network:

```
docker network create my-nginx-network
```

Check if it exists:

```
docker network ls
```

Inspect details:

```
docker network inspect my-nginx-network
```

Create nginx server with new network:

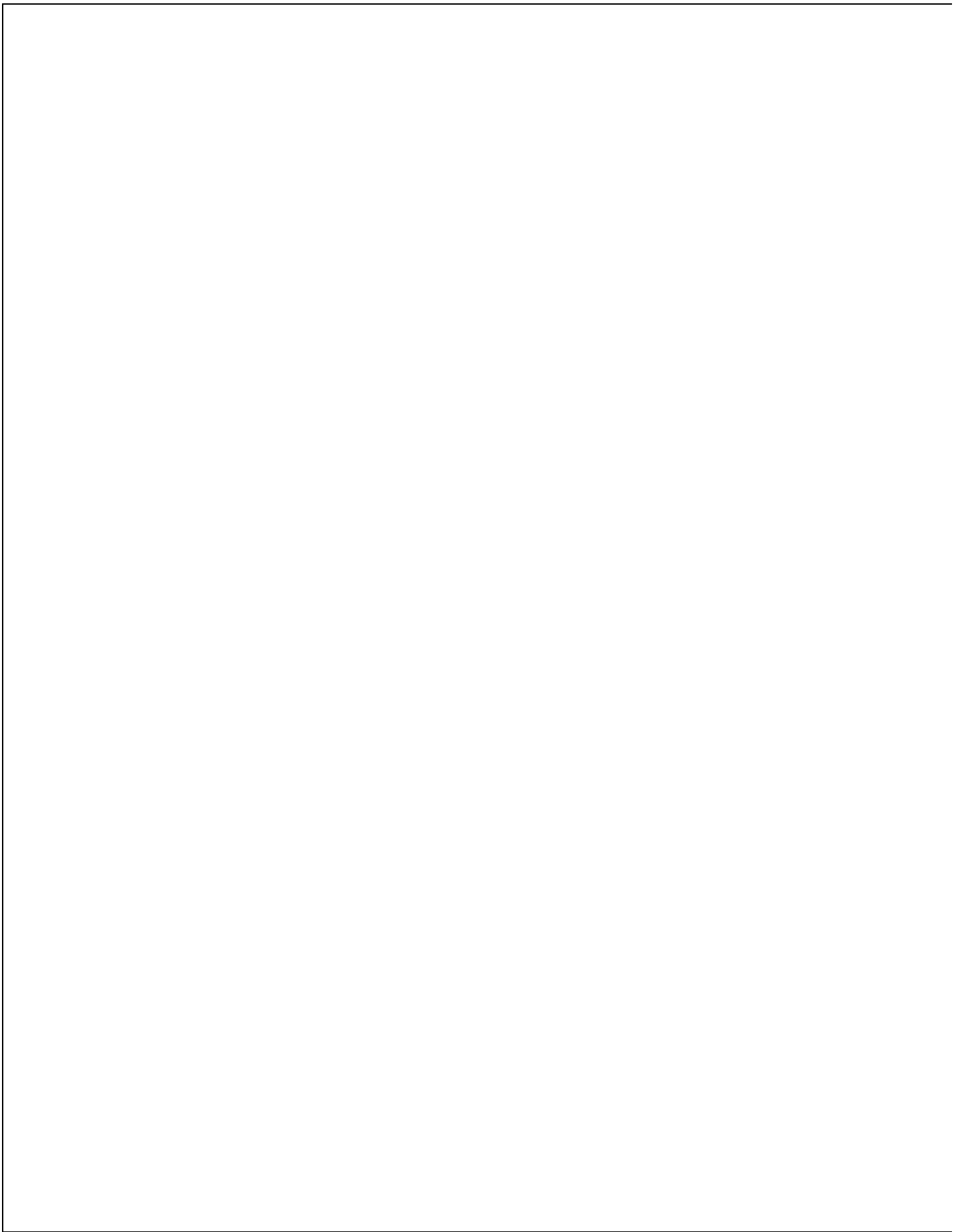
```
docker run --name my-nginx --network my-nginx-network -p 80:80 nginx
```

To verify if network is working, another container will be created and attached to the existing network:

```
docker run --rm -it --network my-nginx-network almalinux sh
```

And by running the curl command, the expected nginx output should be displayed:

```
curl http://my-nginx
```

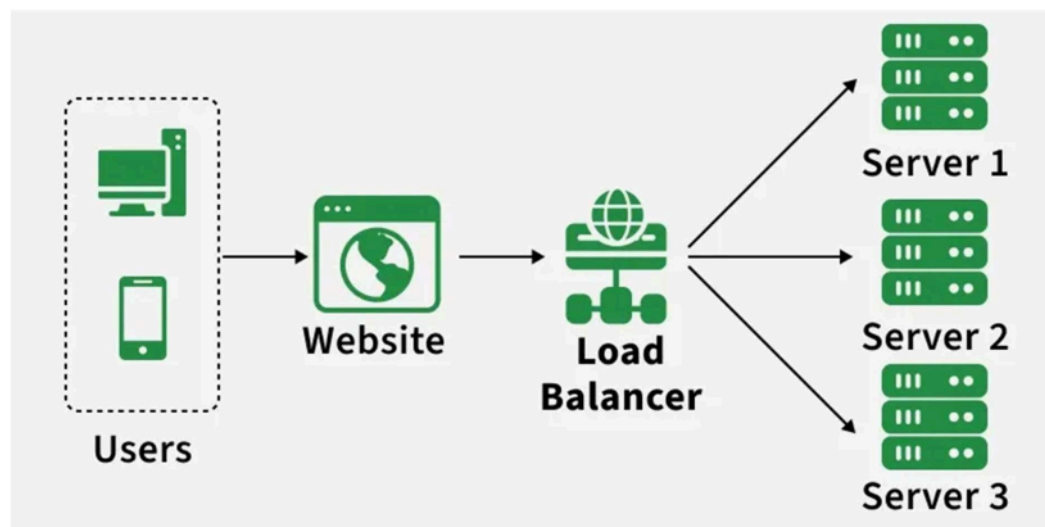


2. Main Cloud Concepts and Cloud Computing Expansion

2.1 Cloud Computing Concepts in Web Applications

2.1.1 Load Balancing

Load balancing allows distributing workloads between two or more devices. In the world wide web, load balancing is often deployed to distribute web traffic among several servers. This helps to reduce strain on a singular server and makes the overall process more efficient. Load balancing is essential when it comes to high traffic web applications [9].



(GeeksForGeeks)

Azure provides several load balancing solutions. But in this report, only the main default Load Balancer that Azure offers will be looked at. Azure Load Balancer is a Layer 4 (transport layer) service that distributes incoming network traffic across multiple virtual machines or instances within a virtual network. Its goal is to enable high availability and reliability when it comes to distributing traffic to avoid overloading a single instance.

The implementation that Azure uses for its configuration is:

- Frontend IP: the public facing IP which allows web users to access the load balancer. One or more can be assigned to a load balancer.
- Backend pools: Assigning pools enables the load balancer to distribute incoming traffic between the allocated instances.

- Inbound Rules: Allows administrators to assign specific rules to ports. Specifying which pools are allowed to receive traffic from ports, and how session persistence is configured.
- Outbound Rules: Allows the instances to interact with the internet if needed. Otherwise, this is automatically generated with the inbound rules.
- Health Probe: This enables the load balancer to distribute traffic to healthy instances. So if latency goes up within one instance, other instances are then distributed to.

2.1.2 Scalability

In cloud computing, scalability is essential when it comes to handling increasing workloads efficiently [10]. This can be done in two ways. Either by adding more resources to an existing server (Vertical Scaling) or by creating more servers (Horizontal Scaling).

Azure provides both vertical and horizontal scaling options to allow their customers to adapt to everchanging work loads.

Vertical Scaling

Azure allows hosting administrators to increase or decrease the resources allocated to a virtual machine. This can be done by changing the instance type (vCPUs and memory) to match the workload requirements.

Horizontal Scaling

Once a load balancer is created, Horizontal scaling can be implemented in Azure. This is done by creating or deleting instances allocated to a load balancer, allowing traffic to be distributed between them. For instance, Azure Virtual Machine Scale Sets automatically adjust the number of VM instances based on performance metrics or a defined schedule.

Scale Sets / Autoscaling

The autoscale feature monitors metrics like CPU usage, memory or even custom metrics. These can trigger scaling actions automatically. Either increasing or decreasing the amount of instances within the scale set.

An example of a autoscale within a scale set:

- Instance range of 1 - 3
- To create a new instance when the Average CPU usage is above 70%
- To delete an instance when the Average CPU usage is below 20 %

Microsoft Azure Search resources, services, and docs (G+/) Copilot pdavis@contoso.com DEFAULT DIRECTORY

Home > Monitor | Autoscale >

Autoscale setting

VMSS-001 (Virtual machine scale set)

Save Discard Refresh Logs Feedback

specific instance count, or via a custom Autoscale policy that scales based on metric(s) thresholds, or schedule instance count which scales during designated time windows. Autoscale enables your resource to be performant and cost effective by adding and removing instances based on demand. [Learn more about Azure Autoscale](#) or [view the how-to video](#).

Choose how to scale your resource

Manual scale
 Maintain a fixed instance count

Custom autoscale
 Scale on any schedule, based on any metrics

Custom autoscale

Autoscale setting name * VMSS-001-Autoscale-441

Resource group rg-001

Predictive autoscale Mode Disabled Pre-launch setup of instances (minutes)

☒ Enable Forecast only or Predictive autoscale. [Learn more about Predictive autoscale](#)

Default* Auto created default scale condition

Delete warning *The very last or default recurrence rule cannot be deleted. Instead, you can disable autoscale to turn off autoscale.*

Scale mode ☒ Scale based on a metric ☐ Scale to a specific instance count

Rules

Scale out	Scale in
When VMSS-001 (Average) Percentage CPU > 70 Increase count by 1	When VMSS-001 (Average) Percentage CPU < 20 Decrease count by 1

+ Add a rule

Instance limits Minimum * 1 Maximum * 3 Default * 1

Schedule This scale condition is executed when none of the other scale condition(s) match

+ Add a scale condition

(Microsoft)

With this setup, a web application can scale with user traffic without downtime or manual intervention.

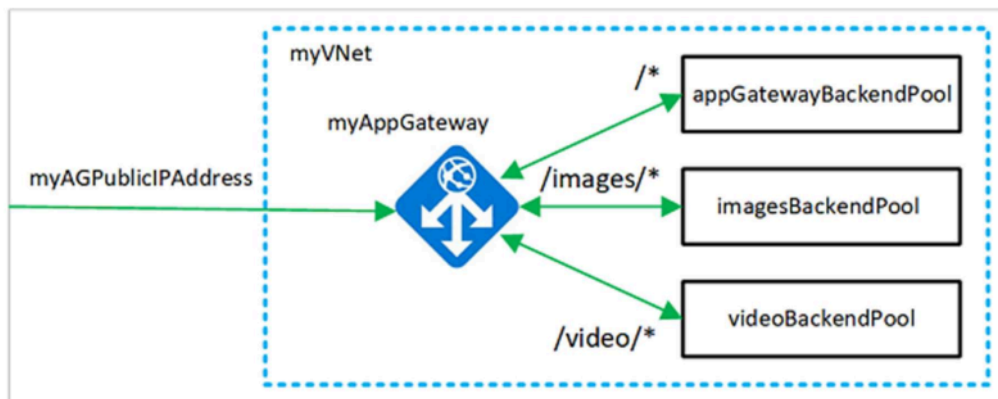
2.1.3 Application Gateways

Application gateways differ from traditional load balancers. Instead of managing traffic at the protocol level (Layer 4), Application Gateways manage traffic at the application layer (Layer 7). This allows the gateways to have more indepth configurations such as directing requests based on URLs, hostnames, headers or cookies. It often also provides security features like SSL/TLS termination and protection against common web attacks.

Azure's implementation is a managed service that combines smart traffic routing, security and scalability for web applications. It has all the configuration settings that a normal load balancer would have, but also includes additional features:

- Azure allows traffic to be distributed on an application-level. Meaning that path-based routing (e.g. splitting /api and /images between different backend pools / instances).
- Web Application Firewall (WAF): protects the backend web applications from common exploits like SQL injection and cross-site scripting (XSS). WAF uses managed rule sets based on the OWASP core rule set or custom rules created by users to inspect incoming HTTP requests.
- SSL/TLS Termination: Decrypts traffic at the gateway, reducing the load on backend servers and allowing for centralized certificate management.
- Session Affinity: Supports cookie sessions, ensuring a website user sticks to the same instance.
- RL-based and Host-based Routing: Allows hosting multiple websites or web applications under one domain (e.g., website.com/direct vs website.com/api).

For a web application, services such as the application, image CDN and video CDN could be split into their own microservices and put into their own backend pools.



(Microsoft)

2.2 Interplay Between Load Balancing and Scalability

Load balancing and scaling complement each other when put together. Scaling is all about increasing or decreasing capacity. By either vertical or horizontal scaling more traffic can

be handled by a service such as a web application. This is specifically true if horizontal scaling takes place. Load balancing is about distributing traffic. It efficiently distributes traffic across available instances so no single instance is overwhelmed [11].

Scaling without load balancing is inefficient. If you add more instances but do not distribute traffic properly, some instances may sit idle while others are overwhelmed by traffic. Load balancing without scaling is limited, distributing traffic evenly across limited resources will still result in overwhelmed instances if there is not enough resources.

Together, scaling provides the capacity where demand rises and falls. While load balancing ensures that traffic is spread evenly against the capacity.

For a typical web application, traffic normally comes in as user web requests such as HTTP/HTTPS. When those web requests are sent to a load balancer or even an application balancer, the requests can be distributed between the allocated instances.

During normal traffic, 2 instances holding the web application would handle traffic. But in the event of a sudden spike in traffic, auto scaling can detect the increase in resource usage and create additional instances 2 -> 10. This allows the load balancer to detect the new instances and start to distribute the sudden traffic surge to healthy instances [12].

With this, no instances are overwhelmed, response times are stable, and the web application stays available at surge times.

2.3 Benefits of Combining All Three Technologies

When combined all together, they can create a robust cloud application:

- **High Availability:** Even if one instance goes down, with load balancing other instances can still handle the incoming traffic.
- **Cost Effective:** Resources would scale up only when needed.
- **Security:** Application gateways having inbuilt security helping to protect against threats.
- **Seamless Experience:** Users would not notice when the web application receives a spike in traffic. As automatic scaling and distribution would handle the increase in load.

In a given web application that has unpredictable traffic demand, this results in a system that is reliable, secure and performant.

2.4 Driving Forces Behind Cloud Computing Expansion and the Role of Serverless vs. Containerisation

Cloud computing has become a foundational part of modern IT infrastructure. Its growth has been achieved by the appealing nature of its economic, technical and operational factors that startups and enterprises like. Among its growth, serverless computing such as Functions as a Service (FaaS) has emerged as a compelling option when it comes to cloud computing.

2.4.1 The key driving forces

Cost efficiency

One of the major shifts in the industry is the move from investing in physical servers and hubs, to paying for cloud computing like Virtual Private Servers (VPS), Virtual Dedicated Servers (VDS) and Bare Metal Machines. Due to this, businesses only need to pay for what they use, leading to reduced upfront costs and minimised financial risks. Cloud providers handle hardware maintenance, updates, and fault tolerance. Built-in redundancy and disaster recovery capabilities improve system reliability without requiring hiring the personnel with expertise.

Scalability

Hosting providers enable near instant scaling of instances. This allows customers to automatically scale up during surging traffic and scale down when the demand drops. Hosting providers also allow customers to buy resources in different regions of the world, allowing for the expansion of reach for their applications.

2.4.2 Serverless Computing

Serverless computing is a product which allows developers to write and deploy their applications without managing any underlying infrastructure. Hosting providers automatically handle the resources, scaling and running of the serverless computing.

Available serverless computing that is offered by hosting providers:

- AWS Lambda
- Azure Functions
- Google Cloud Functions

Serverless Functions

As described by cloudflare “Function-as-a-Service (FaaS) is a serverless way to execute modular pieces of code on the edge. FaaS lets developers write and update a piece of code on the fly, which can then be executed in response to an event, such as a user clicking on an element in a web application. This makes it easy to scale code and is a cost-efficient way to implement microservices.” [13].

With this, developers can spend more time on actually developing the application instead of worrying about hosting and deploying. This can typically lead to a faster development cycle.

Comparing against Containerisation

When it comes to infrastructure management. Serverless Computing like FaaS does not require any management except for deploying the code. On the other hand, user management is needed when containers are created.

Included in the management, FaaS requires no input when it comes to scaling. Due to its fully managed nature, when traffic surges. The functions automatically scale to the demand. Whereas, when it comes to scaling up or down containers there are automated services like Kubernetes (k8s) or policy-based scale sets provided by a hosting provider. But these require configuration to be done beforehand.

For cost comparison, when it comes down to FaaS you only pay for each execution of the deployed code. When it comes to containers, you need to pay for resources to host containers on.

FaaS also includes some setbacks. Having a hosting provider manage the infrastructure makes it troublesome when it comes to understanding the whole system. Also “It can be very difficult to incorporate FaaS code into a local testing environment, making thorough testing of an application a more intensive task.” [13].

Cost

- Serverless aligns strongly with cost efficiency due to granular billing.
- Containers may incur higher baseline costs due to always-on resources.

Flexibility

- Containers provide greater flexibility in runtime configuration.
- Serverless offers flexibility in development speed and deployment simplicity.

Scalability

- Serverless excels with automatic scaling and zero idle cost.
- Containers scale effectively but require orchestration tools and planning.

Operational Efficiency

- Serverless reduces operational burden significantly.
- Containers still require DevOps expertise but offer more control.

3. References

- [1] N. Kratzke, "A Brief History of Cloud Application Architectures," *Applied Sciences*, vol. 8, no. 8, p. 1368, Aug. 2018, doi: <https://doi.org/10.3390/app8081368>.
- [2] C. Kan, "DoCloud: An elastic cloud platform for Web applications based on Docker," 2022 24th International Conference on Advanced Communication Technology (ICACT), pp. 478–483, Jan. 2016, doi: <https://doi.org/10.1109/icact.2016.7423440>.
- [3] IBM, "Virtualization," *Ibm.com*, Apr. 09, 2025.
<https://www.ibm.com/think/topics/virtualization>
- [4] RedHat, "What is a hypervisor?," www.redhat.com, Jan. 03, 2023.
<https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor>
- [5] Claus Pahl, "Containerisation and the PaaS Cloud," *ResearchGate*, Jun. 30, 2015.
https://www.researchgate.net/publication/281432589_Containerisation_and_the_PaaS_Cloud
- [6] S. Singh and N. Singh, "Containers Docker: Emerging roles future of Cloud technology," *IEEE Xplore*, Jul. 01, 2016.
<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7912109>
- [7] "Understanding the image layers," *Docker Documentation*, 2024.
<https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/>
- [8] M. Mohamed, R. Engel, A. Warke, S. Berman, and H. Ludwig, "Extensible persistence as a service for containers," *Future Generation Computer Systems*, vol. 97, pp. 10–20, Aug. 2019, doi: <https://doi.org/10.1016/j.future.2018.12.015>.
- [9] P. U, M. N, A. J, S. R, and R. P, "Load Balancing Policies of Web Servers: Research Analysis, Classification and Perspectives," *ICST Transactions on Scalable Information Systems*, vol. 6, no. 21, p. 159098, Jun. 2019, doi: <https://doi.org/10.4108/eai.10-6-2019.159098>.
- [10] J. Krzywda, A. Ali-Eldin, T. E. Carlson, P.-O. Östberg, and E. Elmroth, "Power-performance tradeoffs in data center servers: DVFS, CPU pinning, horizontal, and vertical scaling," *Future Generation Computer Systems*, vol. 81, pp. 114–128, Apr. 2018, doi: <https://doi.org/10.1016/j.future.2017.10.044>.

[11] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic load balancing on Web-server systems," IEEE Internet Computing, vol. 3, no. 3, pp. 28–39, 1999, doi:

<https://doi.org/10.1109/4236.769420>.

[12] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment," 2009 IEEE International Conference on e-Business Engineering, 2009, doi: <https://doi.org/10.1109/icebe.2009.45>.

[13] "What is Function-as-a-Service (FaaS)? | Cloudflare," Cloudflare. Available: <https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/>

[14] "Network drivers," Docker Documentation, 2024.

<https://docs.docker.com/engine/network/drivers/>

FINAL GRADE

64/100

GENERAL COMMENTS

This is a solid and well-structured submission. The Docker section is the strongest part of the work; the Dockerfile example is practical and correctly constructed, the coverage of volumes and networking is accurate and well-illustrated with appropriate commands, and the image layering explanation is clear. The decision to use an Azure VM as the hosting environment for the container is a reasonable real-world choice and adds some practical depth. The cloud concepts section covers load balancing, scalability, and application gateways competently, with good attention to Azure-specific implementation detail (frontend IPs, backend pools, WAF, path-based routing). The explanation of why load balancing and scaling are complementary is well argued. References are numerous, relevant, and correctly formatted, drawing on a good mix of academic and technical sources.

The main weaknesses are in breadth and depth rather than accuracy. The driving forces section is thin, reducing to essentially just cost and scalability with limited critical analysis, and the serverless vs containerisation comparison, while structurally sound, stays mostly descriptive without significant evaluating statements. The serverless discussion relies heavily on a single Cloudflare source and does not engage with cold start latency, vendor lock-in, or state management as distinguishing factors, all of which I'd hope to see. The video is good, clarifying the implementation.

-TB

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

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27
