



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

Module Code: ACCA7027

Module Title: Cloud Engineering Principles and Practice.

**Assignment Title: Cloud Computing Architectures Containerization, Scalability,
and Managed Services.**

Student Name: Md Arafat Rahman

Student ID: 2437469

Module Tutor: Renuga Jayakumar

Submission Date: 28/04/2026

Table of contents

1. Introduction	2
2. Cloud Technology - Docker and Containerization	2
2.1 The Principles of Virtualization.....	3
2.2 The Principles and Benefits of Containerization.....	4
2.3 Images and Publishing Images.....	5
2.4 Container Volumes	9
2.5 Container Networking.....	10
2.6 Complete Working Example - Web Server Demonstration.....	12
3. Cloud Computing Concepts	15
3.1 Load Balancing	16
3.2 Scalability and Auto Scaling	16
3.3 Application Gateways.....	17
3.4 Why Load Balancing and Scaling are Complementary	17
3.5 Benefits of Combining All Three Technologies	18
4. Driving Forces of Cloud Expansion and Serverless Computing	19
4.1 Main Driving Forces Leading to Cloud Expansion.....	19
4.2 Serverless Computing Azure Functions	20
4.3 Serverless vs Containerization Compare and Contrast	20
5. Conclusion	22
References.....	23

1. Introduction

Cloud computing has changed the way organizations create, deploy and administrate their digital infrastructure, forever. Instead of investing huge sums in on-premises hardware, now companies can rent scalable computing resources over the Internet on a pay as go basis. This transition has hastened innovation, lowered the cost of operations and allowed organizations to react quickly to rapidly changing market demands. Central to modern cloud computing is a tapestry of enabling technologies, one of the most impactful containerizations, particularly Docker [1].

Docker became the standard for running applications in portable and isolated containers. Containers on the other hand, share their OS kernel with the host which makes them way less resource-heavy and faster to start in contrast to traditional virtual machines (VMs). This high level of efficiency, coupled with Docker's ability to standardize application behavior across disparate environments has made it one of the cornerstones of cloud-native application development [2].

In this report, we made an analysis of cloud computing principles and practice based on Docker and containerization, Azure cloud services and serverless computing. The report is structured as follows; Technical principles of virtualization, Containerization, Docker images, volumes, networking and a fully available working example will all be covered in section two. In the third section, we will investigate important cloud computing concepts such as load balancing and scalability along with application gateways in Microsoft Azure. Section 4 looks at the reasons why clouds are growing up and a brief discussion of serverless computing (using Azure Functions, with containerization, as a case study). The conclusion, which is found in Section 5, summarizes the key findings.

In this report, we detail several illustrative example setups using Microsoft Azure as the primary cloud platform, due to the breadth of managed services offered with enterprise-grade service level agreements (SLAs) up until October 2023 that can be helpful. The practical demonstration was done locally in the local docker desktop and then commercial deployment using Azure Container Instances and Docker Hub.

2. Cloud Technology - Docker and Containerization

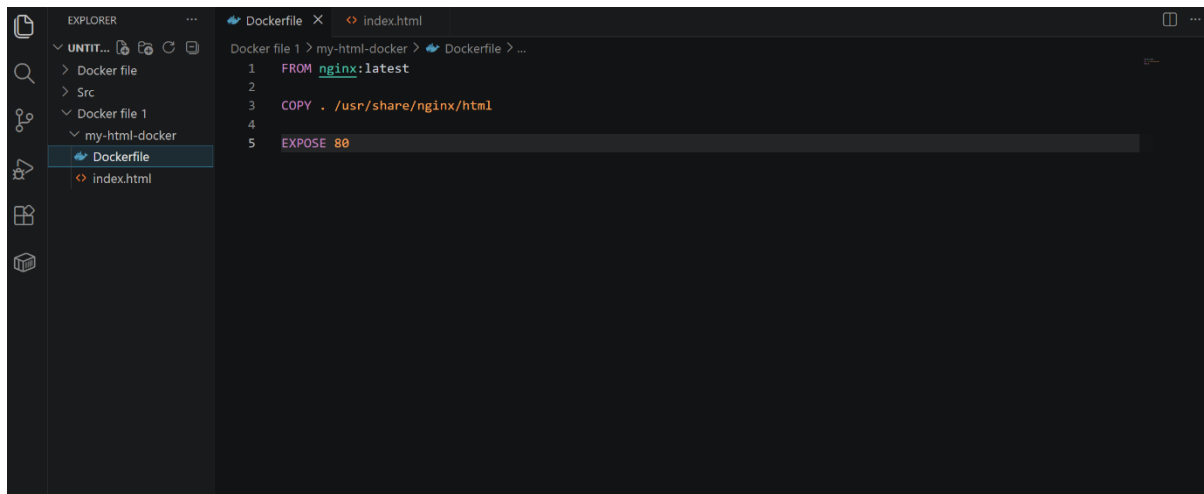
Virtualization and Docker containerization This section takes a tour of the technologies that form the basis for modern cloud deployments. This forms the basic infrastructure of how cloud providers like Microsoft Azure efficiently deliver large scale, isolated computing environments.

2.1 The Principles of Virtualization

The act of virtualization means the creation of a software-based (i.e. Virtual), representation physical computing resources such as servers, storage and networking. Virtualizations abstract hardware to allow multiple independent VMs to all live on a single physical machine even though they may run its own OS and applications as well as have its own resources. This maximizes hardware usage and allows a better chargeback model in data centers.

A hypervisor (also called a Virtual Machine Monitor or VMM) is the software layer responsible for enabling virtualization. There are two types of hypervisors: Type 1 or bare-metal hypervisors run directly on the physical hardware without an underlying operating system. They outperform others by a wide margin and are widely deployed in enterprise-level and cloud environments. Microsoft Hyper-V, VMware and Xen are examples. As for type 2 (hosted) hypervisors, they are built on top of standard host operating systems, meaning these types of hypervisors are more suitable for development and testing environments. Examples are Oracle VirtualBox and VMware Workstation [3].

On the other hand, Azure uses Type 1 hypervisor technology at tremendous scale as its underlying infrastructure for offering basic Infrastructure-as-a-Service (IaaS) services. Its virtualization layer is customized version of Hyper-V, meaning that Azure can slice up the physical servers in its data centers into isolated little islands of performance called virtual machines that are able to quickly provision on demand. Every Azure VM gets a set number of vCPUs, RAM, and disks, the hypervisor strongly isolates VMs running on separate physical hosts [4].



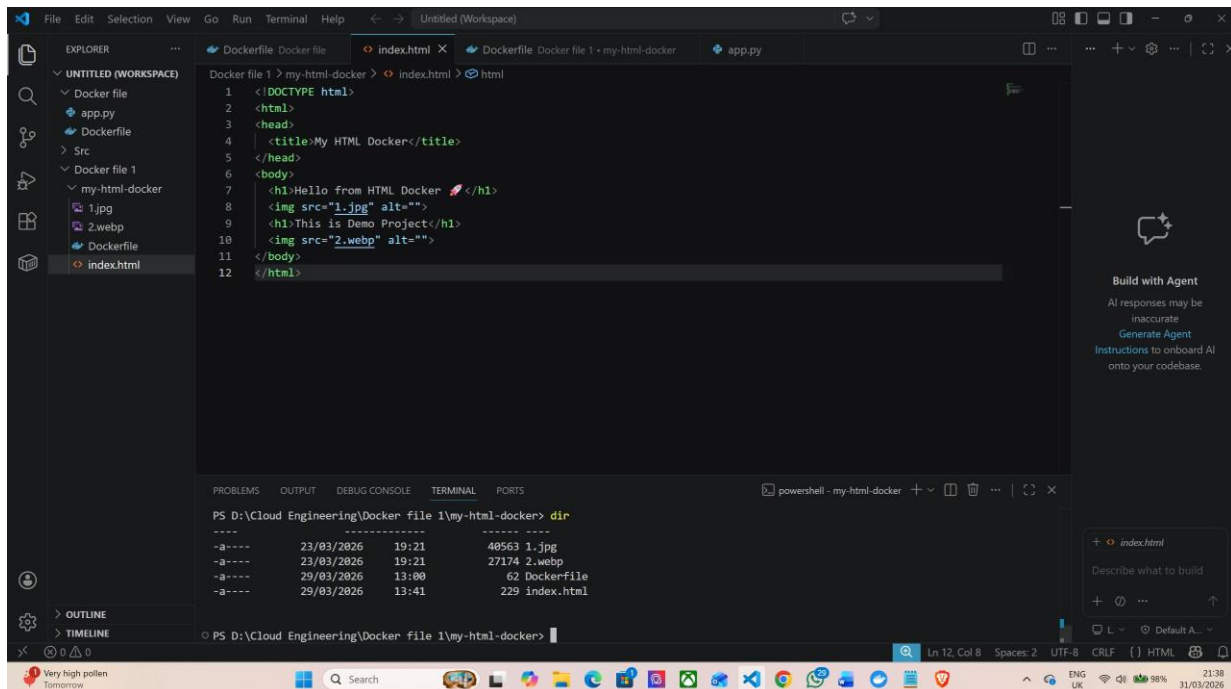
Virtualization architecture showing Type 1 and Type 2 hypervisors and VM isolation

2.2 The Principles and Benefits of Containerization

Whereas virtualization abstracts the hardware, containerization runs at the OS level. A container is essentially an executable package that includes everything needed to run and a piece of software, including the code, runtime, system tools, libraries and configuration. Differently from VMs, containers do not ship with a full guest OS: they share the host OS kernel along with isolating processes and filesystems.

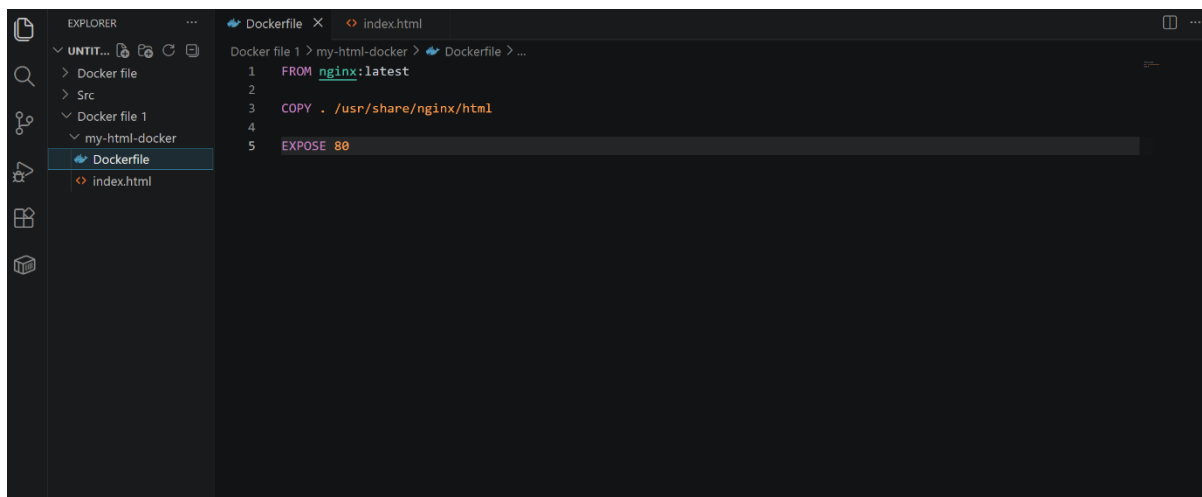
Docker is one of the most widely used containerization platforms. An architect of its architecture has some important components. Docker Engine The underlying runtime for building and running containers A Docker Image is a read-only template with application code and dependencies that are built in the form of layers based using a docker file. A Docker Container is the running instance of a docker image. Docker Hub is a public registry where images are stored and shared. A Docker file is a simple text file containing instructions to build a docker image.

Containerization provides multiple clear advantages over traditional virtualization. Portability means a containerized application can run the same in the developer laptop, test server or any production cloud. By including everything in the container, consistency solves the issue of it works on my machine. Containers are lightweight thanks to sharing the host OS kernel and the fast startup (in milliseconds). Containers (unlike VMs) are resource efficient, enabling many more containers to run on a host than can with VMs. Isolation guarantees that all containers run in their own user space to prevent conflicting dependencies. Finally, containers seamlessly fit into CI/CD pipelines and DevOps processes [5].



2.3 Images and Publishing Images

Docker image is a layered, read-only filesystem that is used as the template for a container. A Docker file is a set of instructions that define what goes into the image. A Docker file builds images from top to bottom in layers. The better the instruction, the smaller, more efficient, and quicker an image becomes (fewer parts to download). This architecture is powerful because it caches layers and reuses them if you modify the application code, Docker only rebuilds that layer [1]. Here is an example of a simple web server image Docker file based on the official nginx base image:



The FROM command here indicates what the base image is, here docker hub nginx: latest. COPY copies an arbitrary html file to the nginx web root folder EXPOSE tells that the container is listening port 80. The CMD: command means which will run after you are running an image as a container. Here, we write the docker file and publish it to Docker Hub via these commands:

Create Docker image and take a Tag:

```
PS D:\Cloud Engineering\docker file 1\my-html-docker> docker build -t my-html-docker .
[+] Building 2.3s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 99B
=> [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: 12.41kB
=> CACHED [1/2] FROM docker.io/library/nginx:latest@sha256:7f0adca1fc6c29c8dc49a2e90037a10ba20dc266baaed0988e9fb4d0d8b85ba0
=> => resolve docker.io/library/nginx:latest@sha256:7f0adca1fc6c29c8dc49a2e90037a10ba20dc266baaed0988e9fb4d0d8b85ba0
=> [2/2] COPY . /usr/share/nginx/html
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:cd02cf27719c93e3b71d50f92345372e07453db79c21ce27745e7ceaa484e4e8
=> => exporting config sha256:610afaebbaa821b7a099e9776173f10b87d939d22d6c6c24f6342a916eb44a51
=> => exporting attestation manifest sha256:b93e9fe7b32a4cd2416e61fcd485e1acba54f53bb0a2ffc3a5bb39e43bc99db8
=> => exporting manifest list sha256:5ff4a0f7396e12c7c96d2b40940d7fd9a717804945ec62dbace6953121727bfe
=> => naming to docker.io/library/my-html-docker:latest
=> => unpacking to docker.io/library/my-html-docker:latest
PS D:\Cloud Engineering\docker file 1\my-html-docker>
```

It uses a command to build Docker image with digger file. Build the image with output as follows: pulling base nginx image to creating next layers and exporting. It will tag the image correctly as my-html-docker so you can use it later.

```
PS D:\Cloud Engineering\docker file 1\my-html-docker> docker images
```

IMAGE	ID	DISK USAGE	CONTENT SIZE	EXTRA
arafatcse/my-html-app:latest	25383c03f722	237MB	63MB	U
arafatcse/my-python-image:latest	f6ba37a07c5a	1.58GB	406MB	U
docker/welcome-to-docker:latest	c4d56c24da4f	22.2MB	6.03MB	U
hello-world:latest	85404b3c5395	25.9kB	9.52kB	U
my-html-app:latest	efe175b78f83	237MB	63MB	U
my-html-docker:latest	5ff4a0f7396e	237MB	63MB	U
my-html-image:latest	b87b6f5c9a4e3	237MB	63MB	U
my-python-image:latest	f6ba37a07c5a	1.58GB	406MB	U

```
PS D:\Cloud Engineering\docker file 1\my-html-docker>
```

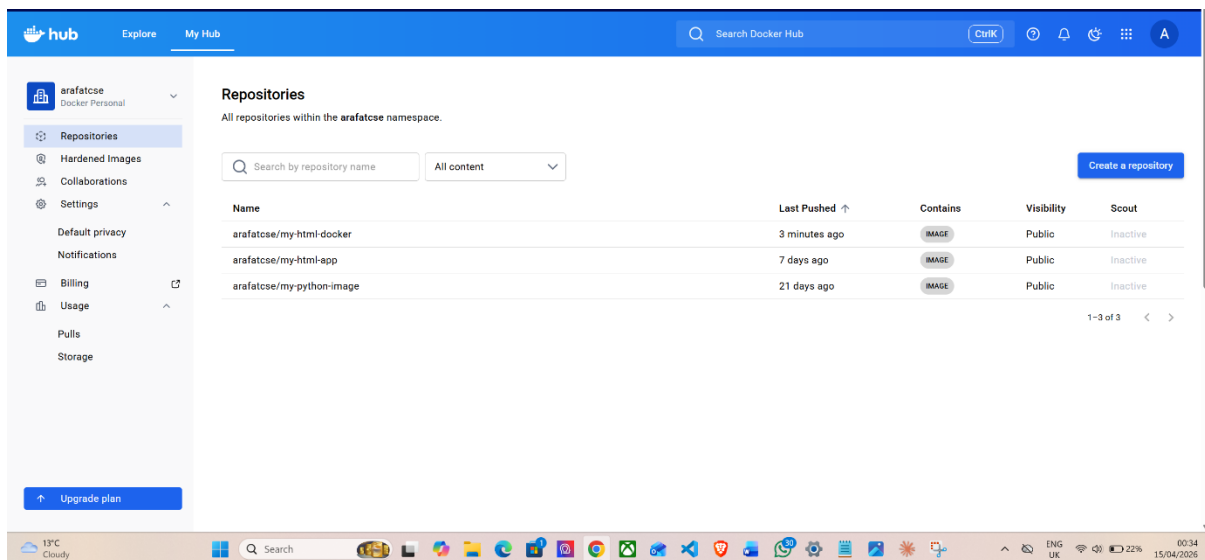
displays all the images in your system. The second image shows several images one of which is my-html-docker that was created in the previous step. They are used to create containers or environments.

Authenticate with Docker Hub:

```
PS D:\Cloud Engineering\docker file 1\my-html-docker> docker tag my-html-docker arafatcse/my-html-docker:v1.0
PS D:\Cloud Engineering\docker file 1\my-html-docker> docker push arafatcse/my-html-docker:v1.0
The push refers to repository [docker.io/arafatcse/my-html-docker]
d2f9f5382d48: Pushed
ac077f1c36c4: Pushed
5435b2dcdf5c: Mounted from arafatcse/my-html-app
054715a6bffa: Mounted from arafatcse/my-html-app
84e114c2bb36: Mounted from arafatcse/my-html-app
4a038fd18db1: Mounted from arafatcse/my-html-app
88d1d984b765: Mounted from arafatcse/my-html-app
7b5d674621c2: Mounted from arafatcse/my-html-app
448ea5cac5d5: Mounted from arafatcse/my-html-app
v1.0: digest: sha256:5ff4a0f7396e12c7c96d2b40940d7fd9a717804945ec62dbace6953121727bfe size: 856
PS D:\Cloud Engineering\docker file 1\my-html-docker>
```

We use docker tag command to tag my-html-docker image with version v1. 0). Finally, the docker push command transfers the tagged image to Docker Hub(arafatcse/my-html-docker) so other users or different parts of your environment can see it.

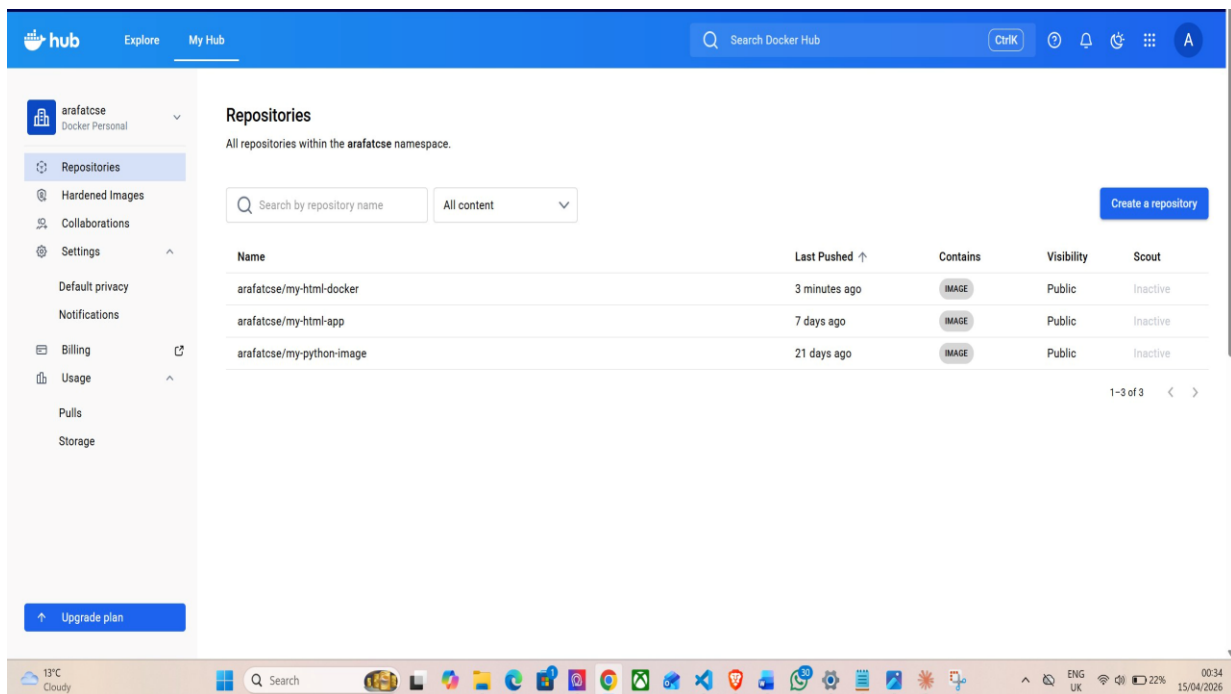
Push the image to the Docker Hub registry:



The whole team/CI/CD pipelines etc need to publish the images in Docker Hub (or in a private registry e.g. Azure Container Registry) One more thing: A docker image can be saved and run anywhere a docker is integrated, to standardize environments as much as possible [1].

```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker build -t my-html-docker .
[+] Building 2.3s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 99B
=> [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: 12.41kB
=> CACHED [1/2] FROM docker.io/library/nginx:latest@sha256:7f0adca1fc6c29c8dc49a2e90037a10ba20dc266baaed0988e9fb4d0d8b85ba0
=> => resolve docker.io/library/nginx:latest@sha256:7f0adca1fc6c29c8dc49a2e90037a10ba20dc266baaed0988e9fb4d0d8b85ba0
=> [2/2] COPY . /usr/share/nginx/html
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:cd02cf27719c93e3b71d50f92345372e07453db79c21ce27745e7ceaa484e4e8
=> => exporting config sha256:610afaebbaa821b7a099e9776173f10b87d939d22d6c6c24f6342a916eb44a51
=> => exporting attestation manifest sha256:b93e9fe7b32a4cd2416e61fcd485e1acba54f53bb0a2ffc3a5bb39e43bc99db8
=> => exporting manifest list sha256:5ff4a0f7396e12c7c96d2b40940d7fd9a717804945ec62dbace6953121727bfe
=> => naming to docker.io/library/my-html-docker:latest
=> => unpacking to docker.io/library/my-html-docker:latest
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

Terminal output docker build -t my-html-docker. completing 8/8 build steps successfully.



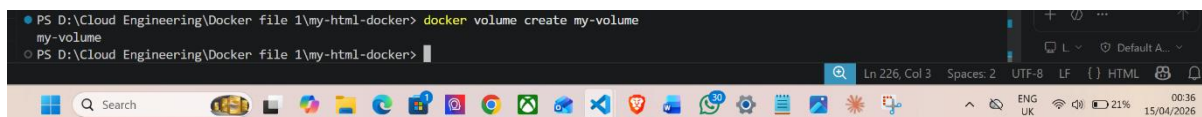
Docker Hub repository showing arafatcse/my-html-docker successfully published as a public image.

2.4 Container Volumes

Containers are ephemeral, anything written in the container files would be lost on the stopping of a container or deleting. This is a disadvantage for stateful applications, such as databases, content management systems or any application you want to keep data alive after the container lifecycle. Docker addresses this through volumes.

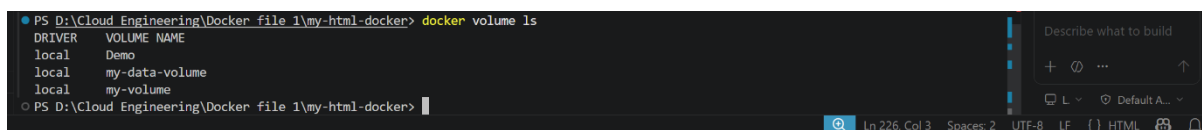
Docker Volume is a storage structure managed by docker that happens even outside the container management boundary. Named volumes are different from bind mounts (which take a directory on the host and mount it inside the container), since volumes are managed by Docker, which stores them in a subdirectory of the data directory on the host. They're also easy to back up and better suited for more devices. The most significant difference is that bind mounts are dependent on the host filesystem path while named volumes are decoupled; managed by Docker [1].

The following commands demonstrate creating and using a named volume:



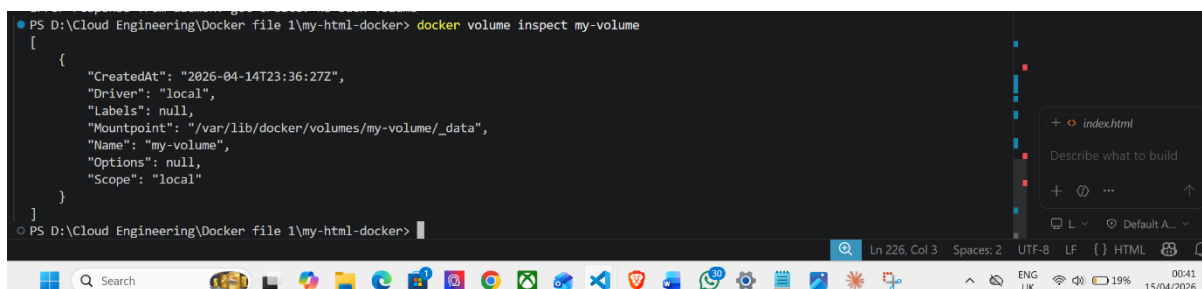
```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker volume create my-volume
my-volume
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

This creates a persistent Docker volume named my-volume, storing data outside the container.



```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker volume ls
DRIVER      VOLUME NAME
local       Demo
local       my-data-volume
local       my-volume
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

Lists all Docker volumes, including my-volume, allowing you to track volumes on your system.



```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker volume inspect my-volume
[
  {
    "CreatedAt": "2026-04-14T23:36:27Z",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/my-volume/_data",
    "Name": "my-volume",
    "Options": null,
    "Scope": "local"
  }
]
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

Displays detailed info about the volume, such as its storage path on the host and associated settings.

To allow containers sharing data together, many containers can mount same named volume (a classic use of sidecars, log aggregates or shared config) Azure File Shares is a type of persistent cloud-managed storage that can be mounted as volumes, allowing containers to restart and not lose any data in an Azure File Share [6], which is very useful when using Azure Container Instances or deploying an application to Azure Kubernetes Service (AKS).

```
PS D:\Cloud Engineering\docker file 1\my-html-docker> docker volume create my-volume
my-volume
PS D:\Cloud Engineering\docker file 1\my-html-docker>
```

Docker volume create — named volume 'my-volume' created successfully

2.5 Container Networking

Docker provides a rich ecosystem that offers an extensible networking model between containers and across the network. Since the aim is to deploy a multi-container application (for example a web app and database running in their own containers), understanding Docker networking is critical.

Docker supports several network drivers. Bridge Network By default, containers can communicate with each other among a bridge network that we created, it means this provides automatic DNS resolution for container names to be used as hostnames. For Host network, it eliminates network separation between the container and the Docker host and is a good choice for high performance applications. If you are using Docker Swarm, any node can communicate with other containers in the Overlay network. None: Stops all networking for a container

Custom bridge networks (avoiding default bridges as much as possible): this provides a lot of advantages on isolated communication (assumed containers need to talk), full automated resolution for container name one can reach them just using the name and some level of fine-grain control over which container can communicate [1]. The following commands demonstrate creating a custom network and running two containers on it:

Create a custom bridge network:

```
PS D:\Cloud Engineering\docker file 1\my-html-docker> docker network create my-app-network
d414a7a96e09f7c55417e97a84e4d456823d7579eda62fff0618a815dcccde442
PS D:\Cloud Engineering\docker file 1\my-html-docker>
```

This creates a custom network named my-app-network, allowing containers to communicate privately.

Start a web container on the custom network:

```
PS D:\Cloud Engineering\docker file 1\my-html-docker> docker network create my-app-network
d414a7a96e09f7c55417e97a84e4d456823d7579eda62fff0618a815dcccde442
PS D:\Cloud Engineering\docker file 1\my-html-docker>
```

Once the custom network is created, containers can be run and attached to it. For example, in your images, a MySQL container (db.) is created and attached to this network:

Start a database container on the same network

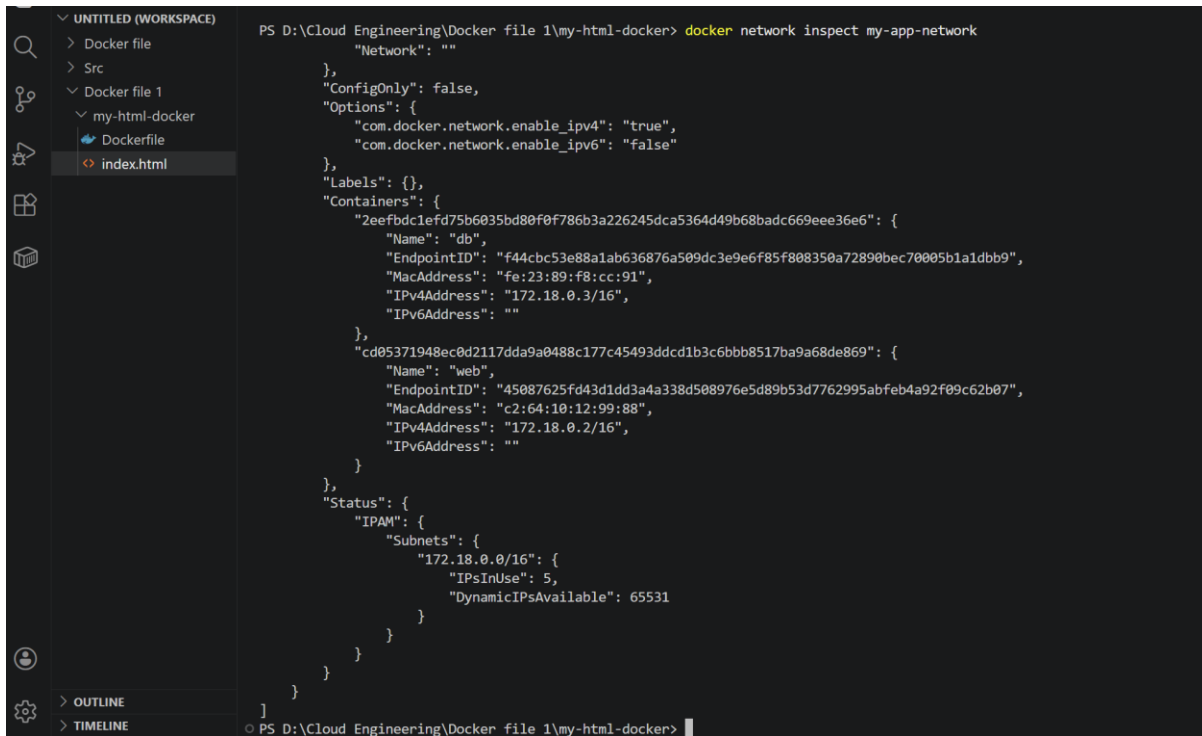
```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker run -d --network my-app-network --name db -e MYSQL_ROOT_PASSWORD=secret mysql:8.0
Unable to find image 'mysql:8.0' locally
8.0: Pulling from library/mysql
656ad78a0e00: Pull complete
79d0deb4205f: Pull complete
2bc0ec069c9c: Pull complete
3fccd87f358c: Pull complete
4641ac6bb997: Pull complete
b061cabb08c3: Pull complete
0c9513d9d126: Pull complete
4abc1d5ebae4: Pull complete
2d5cfd83619e: Pull complete
083ea9601bec: Pull complete
93b0c0ef01e4: Pull complete
75f86fb85099: Download complete
96fdc86d1376: Download complete
Digest: sha256:529b4afa138686871cf698617ef6e347a857179fa2c58e6811b6c20862a1cd6a
Status: Downloaded newer image for mysql:8.0
2eefbdc1efd75b6035bd80f0f786b3a226245dca5364d49b68badc669eee36e6
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

the process of running a MySQL container on a custom Docker network, pulling the MySQL 8.0 image and setting the root password.

List of all networks

```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
bb01064a8268        bridge              bridge              local
929e95db54f2        host                host                local
d4147a96e09         my-app-network      bridge              local
95dfe0237935        none                null                local
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

With this configuration the alias on 'web' can access to 'db' over its hostname db since they both created under the same custom network. This is like production microservices architectures, where services discover each other by name instead of IP address. There are two layers of isolation and service discovery in Azure, which is enabled by creating an Azure Virtual Network (V Net) or an Azure Container Apps environment [4].



```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker network inspect my-app-network
[{"Network": "",
  "ConfigOnly": false,
  "Options": {
    "com.docker.network.enable_ipv4": "true",
    "com.docker.network.enable_ipv6": "false"
  },
  "Labels": {},
  "Containers": {
    "2eefbdc1efd75b6035bd80f0f786b3a226245dca5364d49b68badc669eee36e6": {
      "Name": "db",
      "EndpointID": "f44cbc53e88a1ab636876a509dc3e9e6f85f808350a72890bec70005b1a1ddb9",
      "MacAddress": "fe:23:89:f8:cc:91",
      "IPv4Address": "172.18.0.3/16",
      "IPv6Address": ""
    },
    "cd05371948ec0d2117dda9a0488c177c45493ddcd1b3c6bbb8517ba9a68de869": {
      "Name": "web",
      "EndpointID": "45087625fd43d1dd3a4a338d508976e5d89b53d7762995abfeb4a92f09c62b07",
      "MacAddress": "c2:64:10:12:99:88",
      "IPv4Address": "172.18.0.2/16",
      "IPv6Address": ""
    }
  },
  "Status": {
    "IPAM": {
      "Subnets": {
        "172.18.0.0/16": {
          "IPsInUse": 5,
          "DynamicIPsAvailable": 65531
        }
      }
    }
  }
}]
```

Docker network inspects showing web and db containers connected to my-app-network with separate IPs

2.6 Complete Working Example - Web Server Demonstration

In this section, we will provide a fully end-to-end example of Docker in action: building an image (Docker file), running a Docker webserver, and getting the image online to Docker Hub. This example is practical demonstration of all the concepts from previous sections.

The case is to run a customized HTML page by deploying it in nginx web server using Docker. The crafted workflow goes like the following: Custom index First. An empty HTML file is created containing basic HTML. Next up, we write a Docker file that will build an image based on nginx: latest that serves this HTML file. It builds the image locally and tests it by running the container and opening it in a web browser. And lastly, the image is pushed to Docker Hub for others to pull and deploy later.

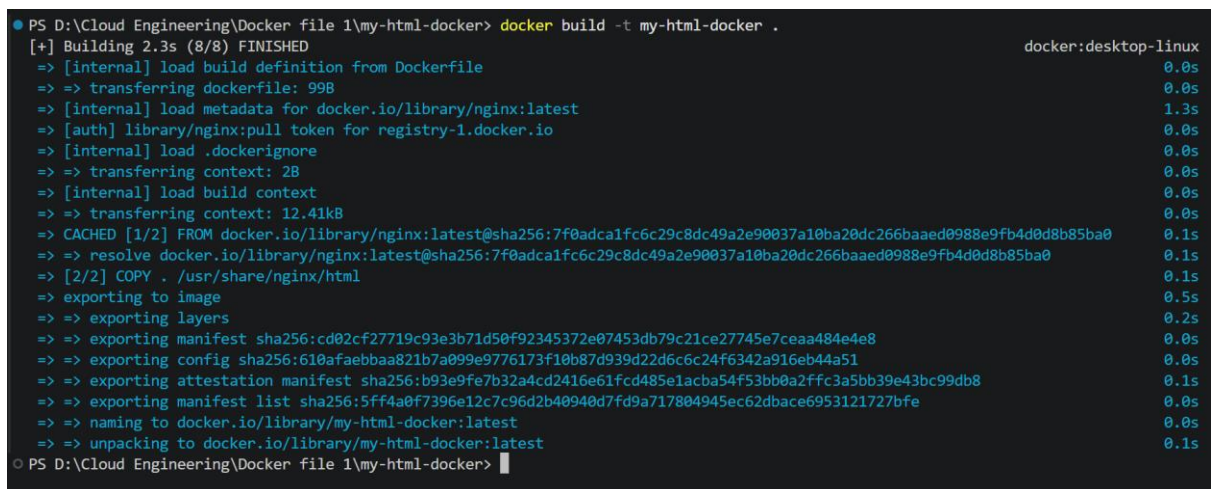
Step 1: Create the custom HTML page (index.html)



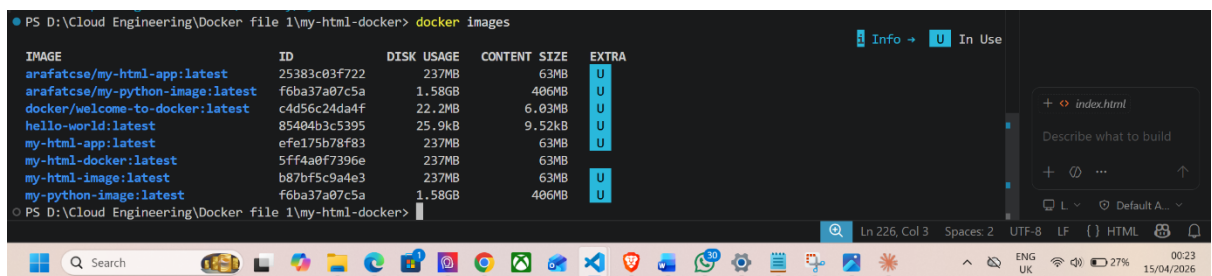
```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8" />
5   <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
6   <title>Docker Project - Md Arafat Rahman</title>
7   <link href="https://fonts.googleapis.com/css2?family=Space+Mono:ital,wght@0,400;0,700;1,400&family=Syne:wght@400;400" rel="stylesheet">
8 </head>
```

Step 2: Build, run, and test the container

Build the image:



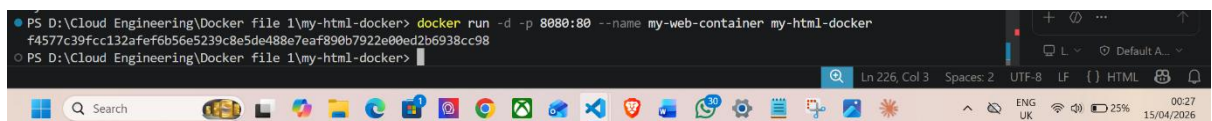
```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker build -t my-html-docker .
[+] Building 2.3s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] load metadata for docker.io/library/nginx:latest
=> [auth] library/nginx:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> [internal] load build context
=> [internal] transferring context: 28
=> [internal] load build context
=> [internal] transferring context: 12.41kB
=> CACHED [1/2] FROM docker.io/library/nginx:latest@sha256:7f0adca1fc6c29c8dc49a2e90037a10ba20dc266baaed0988e9fb4d0d8b85ba0
=> resolve docker.io/library/nginx:latest@sha256:7f0adca1fc6c29c8dc49a2e90037a10ba20dc266baaed0988e9fb4d0d8b85ba0
=> [2/2] COPY . /usr/share/nginx/html
=> exporting to image
=> exporting layers
=> exporting manifest sha256:cd02cf27719c93e3b71d50f92345372e07453db79c21ce27745e7ceaa484e4e8
=> exporting config sha256:610afaebbaa821b7a099e9776173f10b87d939d22d6c6c24f6342a916eb44a51
=> exporting attestation manifest sha256:b93e9fe7b32a4cd2416e61fcd485e1acba54f53bb0a2ffc3a5bb39e43bc99db8
=> exporting manifest list sha256:5ff4a0f7396e12c7c96d2b40940d7fd9a717804945ec62dbace6953121727bfe
=> naming to docker.io/library/my-html-docker:latest
=> unpacking to docker.io/library/my-html-docker:latest
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```



```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker images
```

IMAGE	ID	DISK USAGE	CONTENT SIZE	EXTRA
arafatcse/my-html-app:latest	25383c03f722	237MB	63MB	U
arafatcse/my-python-image:latest	f6ba37a07c5a	1.58GB	406MB	U
docker/welcome-to-docker:latest	c4d56c24da4f	22.2MB	6.03MB	U
hello-world:latest	85404b3c5395	25.9kB	9.52kB	U
my-html-app:latest	efe175b78f83	237MB	63MB	U
my-html-docker:latest	5ff4a0f7396e	237MB	63MB	U
my-html-image:latest	b87bf5c9a4e3	237MB	63MB	U
my-python-image:latest	f6ba37a07c5a	1.58GB	406MB	U

Run the container, mapping port 8080 on the host to port 80 in the container:



```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker run -d -p 8880:80 --name my-web-container my-html-docker
f4577c39fcc132afef6b56e5239c8e5de488e7eaf890b7922e00ed2b6938cc98
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker tag my-html-docker arafatcse/my-html-docker:v1.0
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker push arafatcse/my-html-docker:v1.0
The push refers to repository [docker.io/arafatcse/my-html-docker]
d2f9f5382d48: Pushed
ac077f1c36c4: Pushed
5435b2dcdf5c: Mounted from arafatcse/my-html-app
054715a6bffa: Mounted from arafatcse/my-html-app
84e114c2bb36: Mounted from arafatcse/my-html-app
4a038fd18db1: Mounted from arafatcse/my-html-app
88d1d984b765: Mounted from arafatcse/my-html-app
7b5d674621c2: Mounted from arafatcse/my-html-app
448ea5cac5d5: Mounted from arafatcse/my-html-app
v1.0: digest: sha256:5ff4a0f7396e12c7c96d2b40940d7fd9a717804945ec62dbace6953121727bfe size: 856
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

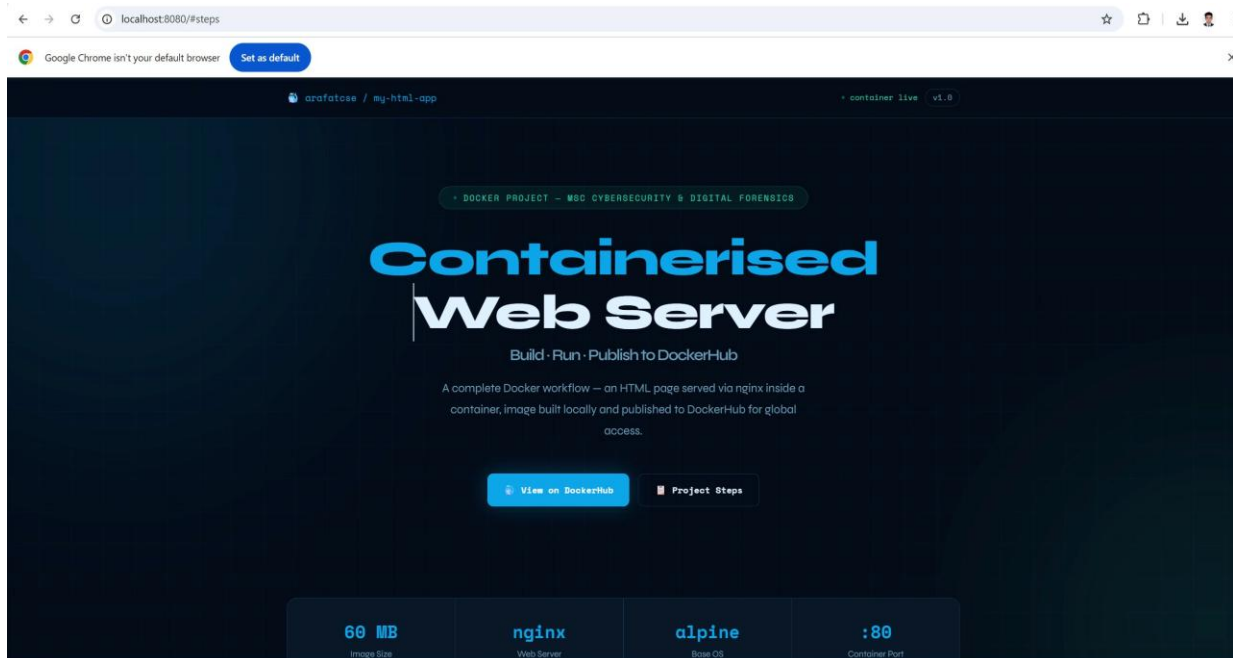
Verify the container is running:

```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
f4577c39fcc1	my-html-docker	"/docker-entrypoint..."	59 seconds ago	Up 58 seconds	0.0.0.0:8080->80/tcp, [::]:8080->80/tcp	my-web-container

```
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

The -p 8080:80 flag maps port 8080 on the host to port 80 inside the container, so going to <http://localhost:8080> in the browser will show our custom HTML page served using nginx. Running docker PS shows the container is running and its port mapping.



Containerized Web Server lives at localhost:8080 HTML page served by nginx inside Docker container.

```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
f4577c39fcc1   my-html-docker "/docker-entrypoint..." 59 seconds ago Up 58 seconds 0.0.0.0:8080->80/tcp, [::]:8080->80/tcp
web-container

PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

Docker PS output confirming my- web-container running with port mapping
0.0.0.0:8080->TCP

Step 3: Push the image to Docker Hub

Docker login:

```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker login
Authenticating with existing credentials... [Username: arafatcse]

Info -> To Login with a different account, run 'docker logout' followed by 'docker login'

Login Succeeded
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

Docker push your-username/demo-webserver: v1

```
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker tag my-html-docker arafatcse/my-html-docker:v1.0
PS D:\Cloud Engineering\Docker file 1\my-html-docker> docker push arafatcse/my-html-docker:v1.0
The push refers to repository [docker.io/arafatcse/my-html-docker]
d2f9f5382d48: Pushed
ac077f1c36c4: Pushed
5435b2dcdf5c: Mounted from arafatcse/my-html-app
054715a6bffa: Mounted from arafatcse/my-html-app
84e114c2bb36: Mounted from arafatcse/my-html-app
4a038fd18db1: Mounted from arafatcse/my-html-app
88d1d984b765: Mounted from arafatcse/my-html-app
7b5d674621c2: Mounted from arafatcse/my-html-app
448ea5cac5d5: Mounted from arafatcse/my-html-app
v1.0: digest: sha256:5ff4a0f7396e12c7c96d2b40940d7fd9a717804945ec62dbace6953121727bfe size: 856
PS D:\Cloud Engineering\Docker file 1\my-html-docker>
```

The image is publicly available once uploaded to Docker Hub and can be pulled and run on any docker machine, even Azure Container Instances, with just a command. This serves as proof of concept regarding the portability and re-usability of Docker images.

3. Cloud Computing Concepts

Here are three of the fundamental concepts found in cloud computing you need to be aware of when trying to build resilient, scalable, and performant cloud architectures: load balancing, scalability and application gateways. These concepts are universal across all major cloud providers, including Microsoft Azure, Amazon Web Services (AWS), Google Cloud Platform (GCP) and others. Each provider offers managed implementations of these capabilities, often with platform-specific names and features but built on the same underlying principles.

3.1 Load Balancing

Load Balancing is a performance optimization technique where incoming network traffic is directed to multiple backend servers so that any one server on which the load has been balanced does not get overwhelmed, increasing availability, reliability, and throughput. All public cloud providers offer managed implementations: Azure Load Balancer, AWS Elastic Load Balancer (ELB) and GCP Cloud Load Balancing.

Cloud load balancers can operate either at Layer 4 (Transport Layer) where it routes TCP/UDP traffic to backends based on IP and port, or at Layer 7 (Application Layer) where HTTP content is routed. Azure Load Balancer and AWS Network Load Balancer (NLB) are Layer 4-based solutions which support deployment modes of types: public (internet-facing) [3], or internal (private network) [4] GCP's TCP/UDP Load Balancing offers similar functionality, too.

Load balancers are built on the same common components across providers, namely Frontend IP (client entry point), Backend Pool (server instances), Health Probes (to flag and remove unhealthy instances), and Routing Rules (to specify how to distribute traffic). All the concepts outlined above are implemented in Azure, AWS and GCP [4].

For a medium-size target example, one typical distribution mechanism is hash of source/destination IP, port and protocol-based algorithm that will guarantee all the packets coming from the same client flow reach to the same backend instance (session stickiness) without needing any application-level cookies. This five-tuple hash method is used by default on Azure Load Balancer, AWS NLB and GCP regional load balancers [4].

3.2 Scalability and Auto Scaling

Scalability refers to a system's ability to handle increasing workload by adding resources. There are two fundamental approaches. Vertical scaling (scale-up) involves increasing the resources of an existing VM for example, upgrading from a 2-vCPU VM to an 8-vCPU VM. This approach is simple but has physical limits and typically requires downtime. Horizontal scaling (scale-out) involves adding more instances of a VM or service to distribute the load. This approach has far greater potential scale, supports zero-downtime scaling, and aligns naturally with cloud pricing models.

Cloud providers offer managed auto scaling services that automatically adjust the number of running instances based on defined rules or observed metrics. On Azure, this is provided by Azure Auto scale working in conjunction with Virtual Machine Scale Sets, a group of identically configured VMs managed as a single unit. AWS offers Auto Scaling

Groups (ASGs) with a similar model, while GCP provides Managed Instance Groups (MIGs) with built-in autoscope support. All these services monitor metrics such as average CPU utilization, memory usage, network throughput, or custom application metrics, and trigger scale-out (add instances) or scale-in (remove instances) actions when thresholds are crossed [7], [8].

For example, an auto scale rule might specify: If average CPU utilization exceeds 70% for 5 minutes, add 2 VM instances. If average CPU drops below 30% in 10 minutes, remove 1 VM instance. Minimum and maximum instance counts prevent runaway scaling or complete scale-in during low periods. Cooldown periods prevent thrashing by enforcing a waiting period between scaling events.

Cloud auto scaling services also typically support schedule-based scaling in addition to metric-based scaling, allowing organizations to pre-emptively scale out before predicted high-traffic periods (such as a lunchtime peak or a planned product launch) and scale in afterwards to minimize costs. Azure Auto scale, AWS Auto Scaling, and GCP Autoscope all offer this capability [7].

3.3 Application Gateways

While Layer 4 load balancers route traffic based on IP addresses and ports, Application Gateways (Layer 7 load balancers) function at the Application Layer of the OSI model. The big distinguishing factor is that a Layer 7 gateway can inspect HTTP/HTTPS traffic, and route based on URL path, hostname, request headers and cookies to make for much more intelligent traffic management. Each of the major cloud providers has a Layer 7 solution as well: Azure Application Gateway, AWS Application Load Balancer (ALB), GCP Cloud Load Balancing with routing rules for HTTP(S).

Application gateways are similar in many features across the cloud providers. The second is URL based routing that sends traffic to different backend pools depending on the path of the URL for example, requests for /Api go to an API pool and /images goes to a pool backed by storage. Under SSL/TLS Termination, the computationally heavy work to decrypt traffic is handled by the gateway rather than backend servers. WAF Web Applications Native protection against common vulnerabilities (SQL Injection, Cross-site scripting (XSS) based on OWASP rule sets, integrated WAF Keep your app safe from zero-day WAF threats. Sticky Sessions (based on cookie) allows to keep a user session routed to the same backend. These features are available in Azure Application Gateway [9], AWS ALB, GCP.

They're suited for relatively small apps, as well as large enterprise deployments, since application gateways support autoscaling and can dynamically increase or decrease capacity with traffic load. Cloud providers seamlessly connect their gateways to in-house

monitoring platforms Azure Monitor, AWS CloudWatch and GCP Cloud Monitoring [9] providing access logs, performance metrics and firewall logs for security auditing.

3.4 Why Load Balancing and Scaling are Complementary

Load balancing and autoscaling, both of which aim to treat variations in workload more efficiently (and reliably), simply address different parts of the same larger challenge; they are only at their most powerful when utilized together. Load balancing routes traffic to the backend instances that are online now, such that all of them get their fair share of requests. If all instances are under heavy load though, load balancing by itself is no solution and the system needs bigger hardware specs.

Autoscaling solves this by automatically adjusting the number of running instances. When an autoscope (such as Azure Auto scale, AWS Auto Scaling, or GCP Autoscope) provisions new server instances due to high load, the cloud load balancer automatically discovers and begins routing traffic to them because they are part of the same backend pool. Likewise, when the autoscope removes instances during low-traffic periods, the load balancer ceases sending traffic to those instances before they are terminated (respecting health probe failures and connection draining).

For instance, think of an e-commerce web application that bears load during a flash sale. Incoming requests are split across the five server instances present now, using the load balancer. When CPU usage exceeds the auto scaling threshold for a certain period, three more instances are provisioned by the cloud autoscope. These new instances are automatically added to the backend pool, distributing the load across 8 servers. Once the sale is over, and after normal traffic returns, the autoscope removes the extra instances, and the load balancer resumes routing traffic across just those original five. This yields a system that is performant at peak load and cost effective during normal traffic.

3.5 Benefits of Combining All Three Technologies

Combining an application gateway, a load balancer and an autoscope whether on Azure, AWS, GCP or another cloud creates a multi-layered, redundant architecture that addresses availability, security, performance and cost all at once.

Imagine an enterprise e-commerce platform running during a peak sales week. An application gateway (such as Azure Application Gateway or AWS ALB) provides a central entry point with SSL termination, a Web Application Firewall (WAF) to protect against malicious traffic, and URL-based routing to direct API calls to an API server farm and image requests to a storage-backed pool. A Layer 4 load balancer then distributes traffic across all server instances in the application tier, ensuring no single instance becomes a bottleneck. As traffic continues to build, the cloud autoscope monitors the instance group and automatically provisions additional servers to maintain response times.

The benefits of this layered approach are: high availability by removing single points of failure at each tier, performance through intelligent Layer 7 routing and even traffic distribution, security through WAF filtering at the application gateway tier, cost optimization with pay for what you use autoscaling, and operational simplicity through the managed services offered by any major cloud provider.

4. Driving Forces of Cloud Expansion and Serverless Computing

4.1 Main Driving Forces Leading to Cloud Expansion

In under two decades, cloud computing has matured from a niche idea to the prevailing paradigm for enterprise IT. This astounding growth is driven by a few overlapping motivations.

One of the most exciting drivers has been cost efficiency. On-premises IT traditionally involves stables of Capex in servers, storage, networking hardware and physical data center facilities that will steadily depreciate and go out of fashion. This is transformed into Operational Expenditure through Cloud computing offering more predictable, pay as you go model where organizations only pay for what they use. This significantly reduces the entry barrier for start-up and enables enterprises to pay for stranded capacity costs. Gartner found that when organizations shift the right workloads to cloud platforms, they realize 15-30% cost savings [10].

Another great driver goes towards flexibility and scalability. Traditional infrastructure planning involves estimating your capacity needs months or years ahead of time. If it is under-provisioned, performance will degrade; if over-provisioned, capital too is wasted. Cloud platforms enable organizations to flexibly adjust resource availability within minutes, responding based on real-world demand. This elasticity is especially beneficial for businesses with fluctuating workloads, seasonally cyclical demands or fast growth patterns.

Cloud adoption has been propelled at an accelerated rate due to the trends of remote work and global collaboration especially after COVID-19. It enables distributed teams to work together in real-time by hosting applications and data in the cloud accessible from any place with an internet connection. This availability was an essential enabler for organizations that had to adapt immediately to working from home in 2020 [3].

One more most important driver is disaster recovery and business continuity capabilities. Cloud providers run multiple data centers in varied locations with redundancy in place enabling organizations to create strong backup and disaster recovery strategies that are economically unviable on premises. For instance, Azure provides geo-redundant storage, site recovery services to migrate VMs between regions as well as paired regions that enable automatic failover in case of a regional outage.

The final aspect that has been revolutionary is the speed of innovation. Cloud platforms allow immediate access to hundreds of managed services and machine learning, databases, messaging queues, IoT platforms, analytics tools etc. organizations no longer must build the necessary infrastructure and operate it themselves. This is a game changer for cloud-native organizations with respect to the time-to-market; it must be ideal to production cycle manifold [11].

4.2 Serverless Computing Azure Functions

The term serverless is somewhat ironic, given that serverless computing isn't without servers at all. Note that even servers exist in the data centers of the cloud provider; serverless means developers have no responsibilities for managing, provisioning, patching and scaling servers. All infrastructure management is automatically performed by the cloud provider.

The main pattern for serverless is Function-as-a-Service (Faas). In Faas, developers create small pluggable pieces of application logic (functions) that are executed in response to events on demand. Instead of having a continually running application server, the platform allocates computing resources on-and-off as needed by instance when a function gets called and releases that data once done. The event-driven execution model means that charges are incurred only when there is actual execution (in milliseconds and number of invocations [12]).

Microsoft Azure has its Faas offer called Azure Functions. It supports programming languages such as C#, JavaScript, Python, Java, and PowerShell. Functions can respond to several things: HTTP requests (serverless API endpoint), timer schedule (scheduled task), messages on Azure Service Bus or Storage Queues, blob storage events and database changes in Azure Cosmos DB. Bindings enable functions to declaratively connect with other Azure services for input or output while eliminating boilerplate connection code [13].

You have the two hosting plans available for you in Azure Functions. With a Consumption Plan, you get real serverless characteristics functions are invoked in response to events on demand, the platform automatically scales down to zero when not needed and you only pay for individual executions, with plenty of free points as well. The Dedicated (App Service) Plan always-on VMs that allows running the functions and is used for predictable, high-volume workloads or long-running functions [13].

4.3 Serverless vs Containerization Compare and Contrast

Serverless computing and containerization are both great strides in the evolution of cloud-native application development, but when it comes to strengths and weaknesses not to mention use cases, they are entirely different. For cloud engineers, knowing when to use what kind of technology is a key to success.

Summary of serverless (Azure Functions) versus containerization (Docker/Azure Container Instances) in table below:

Feature	Serverless (Azure Functions)	Containerisation (Docker)
Startup Time	Milliseconds (cold start possible)	Seconds (image pull + start)
Scaling	Automatic, per-request, scales to zero	Manual or auto scale rules required
Cost Model	Pay per execution (Ms billing)	Pay for running container time
State	Stateless by design	Can be stateful with volumes
Complexity	Low — not infra management	Medium — Docker file, orchestration
Control	Limited (managed by provider)	High (full environment control)
Ideal Use Case	Event-driven, short tasks, APIs	Long-running apps, microservices
Azure Example	Azure Functions	Azure Container Instances / AKS

Comparison of Serverless (Azure Functions) and Containerization (Docker)

Using serverless is ideal for short-lived, event-driven workloads. For example, image processing post upload to blob storage, email notifications in response to a database change, HTTP API requests with unpredictable traffic patterns and periodically run of data transformation jobs. The main benefits are no infrastructure management, scale down to zero (no cost when idle) and quick development of granular functions.

By contrast, containerization is great for long-lived services, complicated microservices architectures, and applications that have demands on runtime precision. For all containerized applications, e.g. those running in Azure Kubernetes Service (AKS), the dependencies, configuration, and runtime environment can be explicitly defined, tested against a specific version of ACR before going live. App containers are also more portable between clouds, as the same Docker image can work on Azure, AWS, Google Cloud or in-house [14], [11].

The two technologies solve the cloud adoption motivators discussed in Section 4.1, but differently. With granular billing and zero idle costs, serverless is the most cost-efficient computing model while containers give maximum portability and control over environment. For some types of use cases, serverless speeds up development and deployment (Ex: event-driven microservices), for others, containers are essential to handle complex, stateful microservice. In practice, almost any modern cloud-native architecture uses both: serverless functions for event-driven edge cases and background jobs paired with containers hosting the core application services.

5. Conclusion

Using Log Analytics and Azure Monitoring to build the above cloud infrastructure behind your applications, using a log approach for monitoring purposes; all of this by utilizing important principles and technologies relevant to today engineering cloud infrastructure on Docker Containerization, Microsoft Azure services, serverless computing respectively. They help show that this is easily transformative in its own right as illustrated by docker containerization: pot innovation wraps applications (and their requisite dependencies) into independent and portable units or containers which make it easier to isolate one part of the system from another which further prevents environment differences between dev/stage/prod environments within the CI/CD pipeline through immediate consistency of cross-environment behavior/response.

A breakdown of Azure cloud services for instance, but not limited to Azure Load Balancer, Azure Auto scale and now the new Azure Application Gateway shows how many technologies you can blend (creating high availability, performance and cost effectiveness within your architecture). A few of these are: load balancers which ensure traffic is balanced between backend instances, autoscores that automatically adjust efficiency according to demand and application gateways at the entry point of our architecture offering intelligent Layer 7 routing and security features. Both make for a perfect starting platform for enterprise-grade cloud deployments.

How cloud growth driving force analysis helps: Value performance, elasticity willpower, mobility, catastrophe restoration and speed of innovation lead the occasions deciding whether they moved to the cloud. Azure Functions provides serverless computing as a solution to many of these drivers with its full abstraction infrastructure layer and millisecond-billed execution delivered via an event-driven programming model that flows and auto-scales based on demand simply drops to zero. Hence the basis of competition that you witness from a Serverless vs Containerization paradigm perspective is around the pattern your application supports vs your actual needs for scaling to meet load requirements therefore more complementary than competitive.

Looking ahead Towards the future, Expect that. Container + serverless converged even faster (e.g. Azure Containers Apps & WASM (Web Assembly) RUNTIME); With edge computing and AI-driven infra management that automates many of the optimization decisions engineers make today on the cloud, Cloud-native patterns will finally reach the network edge. The principles of abstraction, isolation, scale and event-driven architecture first laid out in this report will continue to form the bedrock cloud engineering for some time to come.

References

- [1] Docker Inc "Docker Documentation" Docker Docs Accessed: Mar 2026[Online] Available: <https://docs.docker.com>
- [2] J. Turnbull, The Docker Book: Containerization is the New Virtualization Self-published, 2014 [Online] Available: <https://dockerbook.com>
- [3] P. Mell and T. Grance, "The NIST Definition of Cloud Computing" Nat. Inst Stand Technol Gaithersburg, MD, USA, NIST Spec Publ. 800-145, Sep 2011 [Online] Available: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>
- [4] Microsoft, "Azure Load Balancer Documentation" Microsoft Azure Accessed: Mar 2026. [Online] Available: <https://docs.microsoft.com/en-us/azure/load-balancer/>
- [5] A. Mouat, Using Docker: Developing and Deploying Software with Containers Sebastopol, CA: O'Reilly Media, 2016.
- [6] Microsoft, "Azure Container Instances Documentation" Microsoft Azure Accessed: Mar 2026 [Online]. Available: <https://docs.microsoft.com/en-us/azure/container-instances/>
- [7] Microsoft, "Autoscale Overview" Microsoft Azure Monitor Accessed: Mar 2026[Online]. Available: <https://docs.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview>
- [8] Microsoft, "Azure Virtual Machine Scale Sets Documentation" Microsoft Azure Accessed: Mar 2026 [Online] Available: <https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/>
- [9] Microsoft, "What is Azure Application Gateway" Microsoft Azure Accessed: Mar 2026 [Online] Available: <https://docs.microsoft.com/en-us/azure/application-gateway/>
- [10] Gartner, "Forecast: Public Cloud Services Worldwide 2021-2027" Gartner Inc Stamford CT, USA Rep 2023.
- [11] N. Kratzke and P. C. Quint, "Understanding cloud-native applications after 10 years of cloud computing" J. Syst. Soft vol 126, pp1-16, Apr 2017.
- [12] M. Roberts, "Serverless Architectures" martinowler.com Accessed: Mar 2026 [Online]. Available: <https://martinowler.com/articles/serverless.html>

[13] Microsoft, "Azure Functions Overview" Microsoft Azure Accessed: Mar 2026 [Online]. Available: <https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview>

[14] Microsoft, "What is Azure Kubernetes Service (AKS)" Microsoft Azure Accessed: Mar 2026 [Online] Available: <https://docs.microsoft.com/en-us/azure/aks/>

<https://www.youtube.com/watch?v=gUyTjETeoFM>