



Prifysgol Cymru

Y Drindod Dewi Sant

University of Wales

Trinity Saint David

Submitted by

Shamsuddin Mahmud (2432880)

Supervisor: Renuga Jayakumar

Course: Cloud Engineering Principles and Practice

Project submitted as part of the requirements for the
award of MSc

Cloud Computing with Docker: Virtualisation, Containerisation, and Scalable Deployment Practices

Table of Contents

Abstract.....	3
1. Principles of Virtualisation.....	3
2. Principles of Containerisation.....	3
3. Dockerising a Web Server: Images, Containers, Volumes, and Networking	4
3.1 Dockerfile Construction and Image Build.....	5
3.2 Docker Networking (Creating and Using a Docker Network).....	7
3.3 Docker Volumes (Creating and Mounting a Volume)	8
3.4 Running the Container with Port Mapping, Network, and Volume	9
3.5 Video Demonstration.....	11
4. Load Balancing, Scalability, and Application Gateways in Cloud Computing.....	11
4.1 Load Balancing.....	11
4.2 Scalability and Autoscaling.....	12
4.3 Application Gateways	12
5. Why Load Balancing and Scaling Work Together.....	12
6. Benefits of Combining Load Balancing, Autoscaling, and Gateways	13
7. Serverless Computing vs. Containerisation.....	14
8. Driving Forces Behind Cloud Adoption.....	15
References	16

Abstract

Cloud computing has revolutionised the development and deployment of web applications, delivering computation as a utility. Public cloud Infrastructure-as-a-Service (IaaS) is based on virtualisation, and containerisation provides a lightweight alternative for running portable workloads that are quick to deploy, in line with DevOps practices. The concept of virtualisation, including containerisation with Docker images and volumes, as well as Docker networking, which all underpin the process of building containers, is outlined through an applied exercise in Dockerisation. Then it goes on to explore how the top cloud providers (AWS and Azure) provide features like load balancing, autoscaling, and application gateways for high availability, performance, and security of cloud-hosted applications.

1. Principles of Virtualisation

Virtualisation abstracts physical hardware into multiple isolated execution environments, known as virtual machines (VMs), on a single host. A software layer called a hypervisor mediates access to the underlying CPU, memory, storage, and network, enabling multiple operating system instances to run concurrently on one physical server[1]. This consolidation and multi-tenancy are central to cloud economics, allowing providers to improve hardware utilisation and offer on-demand infrastructure [1], [2]. Two common hypervisor types are Type 1 (bare-metal), which runs directly on hardware for optimal performance and isolation, and Type 2 (hosted), which runs on a host OS (often used for local development and testing)[1][2].

Example – virtualisation of shared hardware: A single physical server can host multiple VMs, each with its own guest OS, libraries, and application stack. The hypervisor schedules virtual CPU time for each VM, enforces memory isolation, and virtualises I/O devices, allowing different teams or tenants to share the hardware safely [3][4]. This approach increases utilisation and supports rapid provisioning. Deploying a new VM is faster and more cost-efficient than procuring and installing a new physical server [2], [3]. Virtualisation's key strengths are compatibility and isolation. However, VMs are relatively heavy: each includes a full guest OS, has longer boot times, and typically achieves lower application density per host compared to container-based approaches [4].

2. Principles of Containerisation

Containerisation is a lightweight form of virtualisation that isolates applications at the process level while sharing the host system's kernel. On Linux, this is implemented via namespaces and cgroups[5]. Unlike a VM, a container does not require a whole guest OS; instead, all containers on a host share the host's operating system kernel.

Modern container tooling, pioneered by Docker, standardises an image format and provides convenient runtime and registry services for distributing container images [5]. Docker containers are created from images that package the application and its dependencies, but rely on the host OS kernel, making them much lighter than VMs. Key benefits of containerisation include portability, fast startup, and high density [6]. These attributes support modern practices, including continuous

integration/continuous delivery (CI/CD) and microservices architectures, which are often orchestrated by platforms such as Kubernetes [6].

However, containerisation is not a complete replacement for VMs in all cases. Containers are a much lighter-weight alternative to VMs, but they do share the host kernel. If there were a vulnerability at the kernel level, it would impact all containers, meaning "isolation" is also weaker than that of VMs in some ways. Furthermore, orchestration and external tools for network/storage/security need to be put in place before large-scale container deployments achieve parity with VM environments [4][6]. In practice, organisations often use both VMs and containers: for example, VMs provide strong isolation for multi-tenant scenarios, while containers provide efficiency and agility for application deployment.

3. Dockerising a Web Server: Images, Containers, Volumes, and Networking

To illustrate containerisation in practice, this section walks through a class practical that containerises a simple static web page using Docker and Nginx. The project consists of a single-page HTML file (index.html) that will be served by an Nginx web server inside a Docker container. We demonstrate the creation of a Dockerfile, building a Docker image, and utilising Docker volumes and networks for persistence and connectivity.

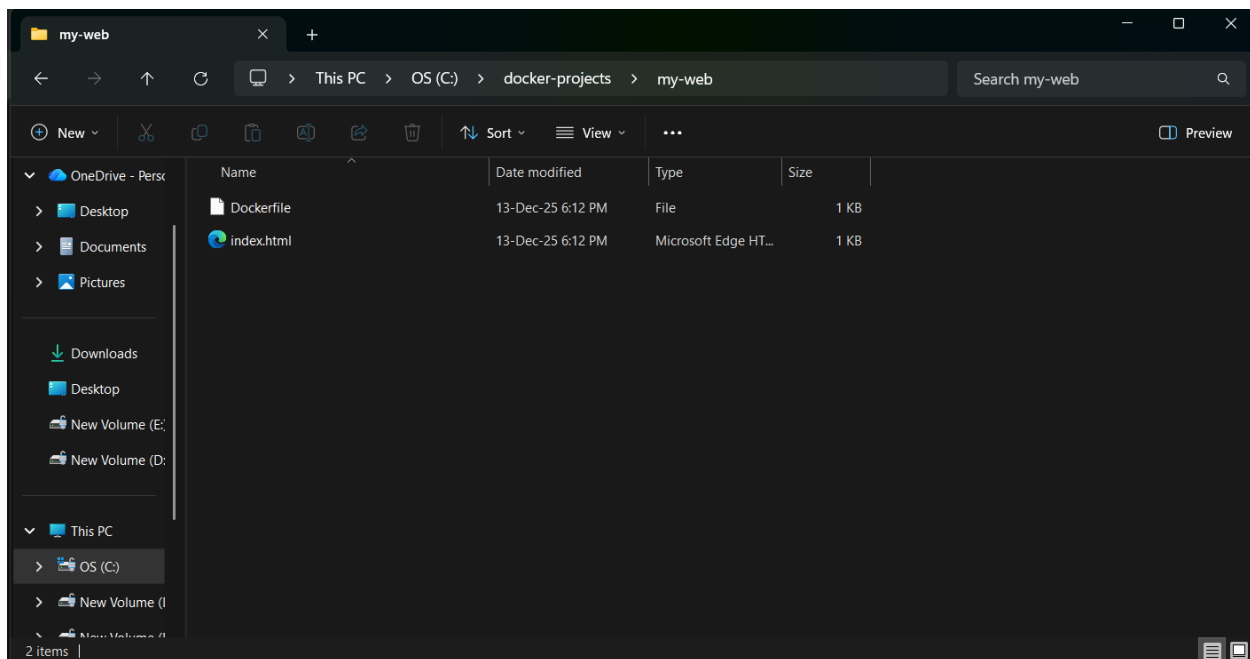


Figure 1. Project folder structure for the Docker web server project.

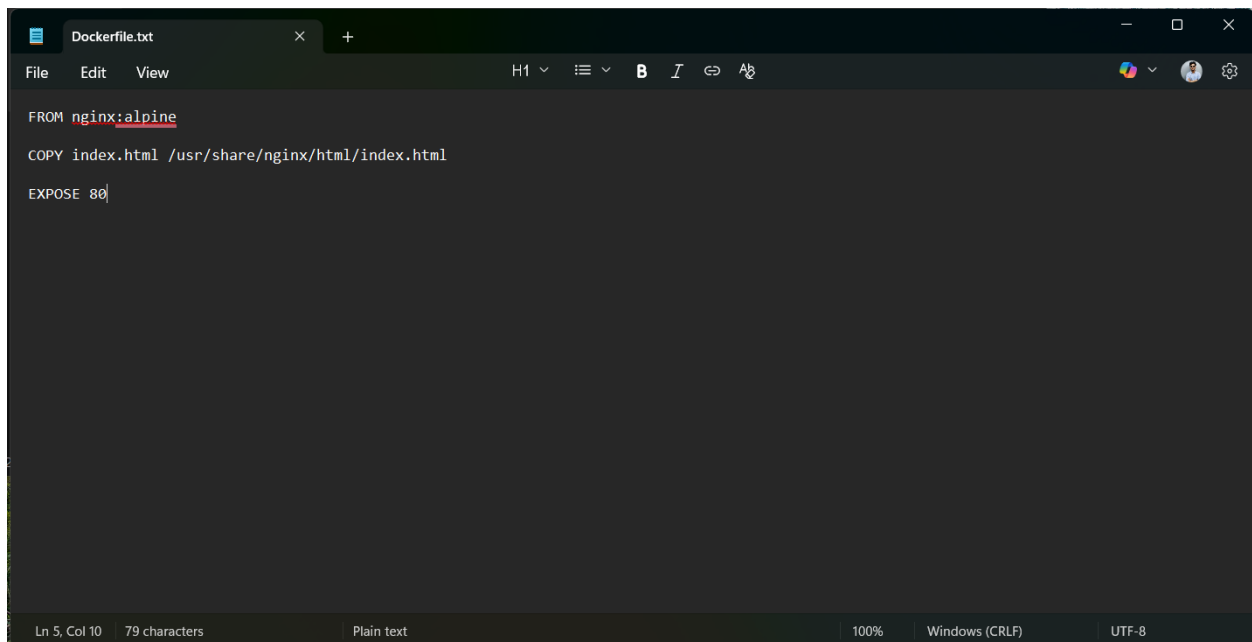
The project directory (Figure 1) contains a Dockerfile and the static HTML content to be served. Keeping these files together ensures the Docker build context includes all necessary components (the server configuration and web page).

3.1 Dockerfile Construction and Image Build

A Dockerfile is a text file that contains instructions for assembling a Docker image. In this case, a minimal Dockerfile is used to create an Nginx-based web server container. The Dockerfile starts from an official Nginx base image and copies the website's HTML file into Nginx's default document root:

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

This Dockerfile utilises the lightweight Alpine-based Nginx image to minimise image size and reduce the attack surface [7]. It then adds the index.html file to the appropriate directory in the image so that Nginx will serve it, and it declares that the container will listen on port 80.

A screenshot of a code editor window titled 'Dockerfile.txt'. The editor has a dark theme and shows the following text: 'FROM nginx:alpine', 'COPY index.html /usr/share/nginx/html/index.html', and 'EXPOSE 80'. The status bar at the bottom indicates 'Ln 5, Col 10', '79 characters', 'Plain text', '100%', 'Windows (CRLF)', and 'UTF-8'.

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

Figure 2. Dockerfile contents as shown in the code editor.

The Dockerfile (Figure 2) specifies all instructions needed to build the container image. Using a small base image and including only the necessary file (the HTML page) keeps the image efficient [5][7].

Before building the image, the working directory is set to the project folder containing the Dockerfile. This ensures Docker has access to the index.html during the build. In a terminal, one would navigate (with cd) into the project directory:

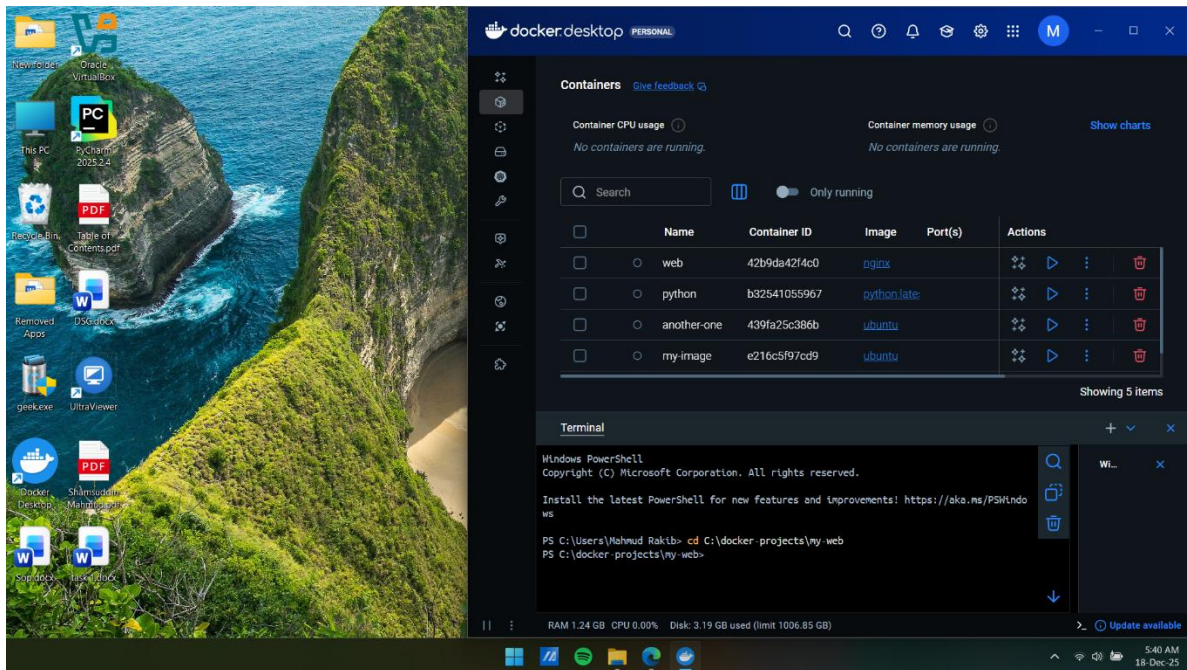


Figure 3. Terminal navigation to the project directory before building the image.

Figure 3 shows moving into the project folder in a terminal. This step is required so that the Docker build command can find the Dockerfile and context.

With the Dockerfile ready, an image can be built using the Docker CLI. The build command is:
`docker build -t mahmud1821740043/my-web:1.0 .`

This command reads the Dockerfile in the current directory (.) and creates a new image tagged `my-web:1.0` (under a Docker Hub username namespace for later publishing). The output of the build process shows Docker pulling the base image and executing the instructions to produce each layer of the new image.

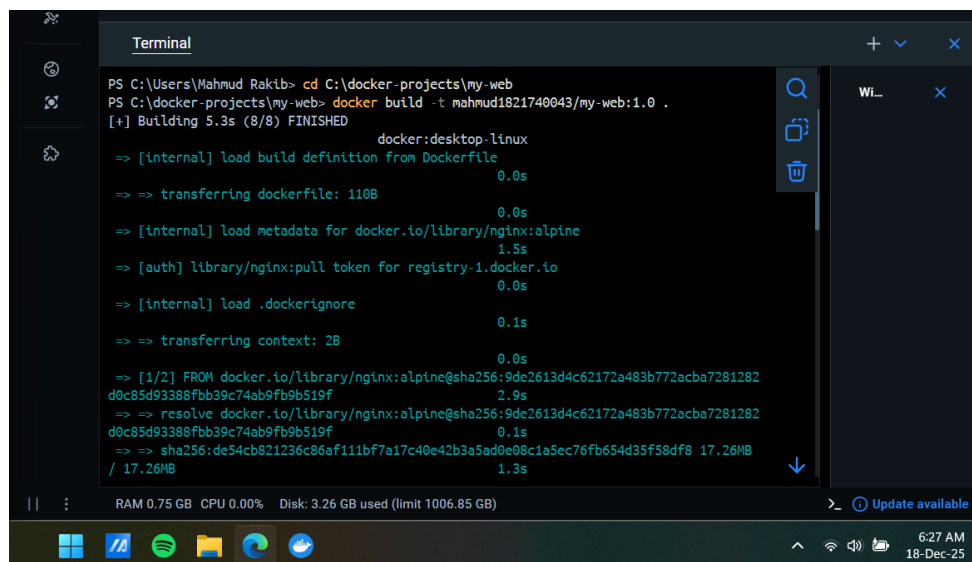
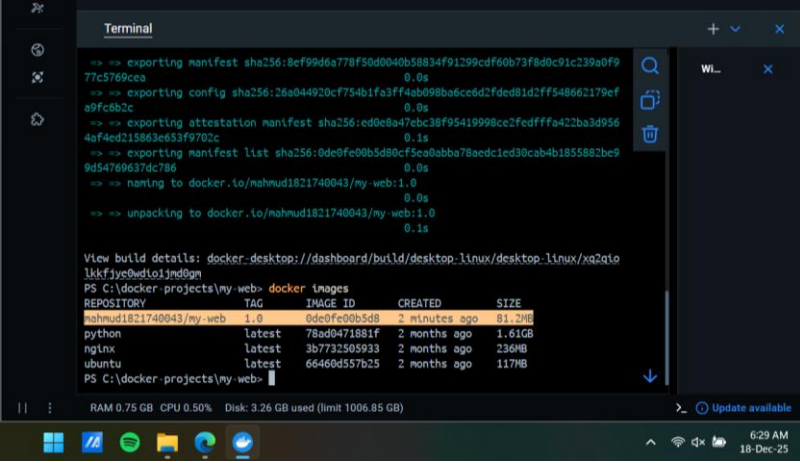


Figure 4. Docker build output, creating the Docker image layer by layer.

As shown in Figure 4, the Docker build command processes the Dockerfile and displays the build steps. Each instruction creates a new image layer [8]. Upon completion, the new image is stored on the local Docker engine.

After a successful build, the new image can be verified by listing the local images:



```

=> exporting manifest sha256:8ef99d6a778f30d0040b58834f91299cdf60b73f8d0c91c239a0f9
77c5769cca 0.0s
=> exporting config sha256:26a044920cf754b1fa3ffa0980a6ce6d2fde081d2ff548662179ef
a9fc5b2c 0.0s
=> exporting attestation manifest sha256:ed0e8a47cbc38f95419998ce2fedffa422ba3d956
4af4ed215863e653f9702c 0.1s
=> exporting manifest list sha256:0dc0fe0b5d80cf5ea8abba78aedc1ed30cab4b1855882be9
9d54769637dc786 0.0s
=> naming to docker.io/mahmud1821740043/my-web:1.0 0.0s
=> unpacking to docker.io/mahmud1821740043/my-web:1.0 0.1s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/xq2qio
lkkfyje0wdiojnd0gm
PS C:\docker-projects\my-web> docker images
REPOSITORY          TAG          IMAGE ID       CREATED        SIZE
mahmud1821740043/my-web 1.0         0dc0fe0b5d80  2 minutes ago  81.2MB
python              latest      78a60471891f  2 months ago  1.61GB
nginx               latest      3b7732568933  2 months ago  236MB
ubuntu              latest      66460d557b25  2 months ago  117MB
PS C:\docker-projects\my-web>

```

Figure 5. Listing Docker images after the build, including the newly created image.

Figure 5 displays the result of running Docker images, where the image appears in the list alongside its size. This confirms that the Docker image has been built and is now available locally.

3.2 Docker Networking (Creating and Using a Docker Network)

Before running the container, a custom Docker network can be established to demonstrate Docker's networking capabilities. By default, containers launched in Docker share a standard bridge network; however, user-defined bridge networks enable containers to communicate by name, offering improved isolation from containers of other projects [9]. In a multi-container application, a dedicated network is useful for inter-container communication.

A Docker network is created with the command:

```
docker network create my-web-net
```



```
0.0s
=> => unpacking to docker.io/mahmud1821740043/my-web:1.0
0.1s

View build details: docker\_desktop://dashboard/build/desktop-linux/desktop-linux/xq2qio
lkkfjye0wdiojnd0gm
PS C:\docker-projects\my-web> docker images
REPOSITORY          TAG         IMAGE ID      CREATED        SIZE
mahmud1821740043/my-web  1.0        0de0fe00b5d8  2 minutes ago  81.2MB
python               latest     78ad0471881f  2 months ago  1.61GB
nginx                 latest     3b7732505933  2 months ago  236MB
ubuntu                latest     66460d557b25  2 months ago  117MB
PS C:\docker-projects\my-web> docker network create my-web-net
Error response from daemon: network with name my-web-net already exists
PS C:\docker-projects\my-web> docker network ls
NETWORK ID          NAME           DRIVER        SCOPE
e69998f7549d        bridge        bridge        local
ea689ca39922        host          host          local
0d644877bfe9        my-network    bridge        local
16b184b9158c        my-web-net    bridge        local
faa8d01ee4df        none         null          local
PS C:\docker-projects\my-web>
```

RAM 0.72 GB CPU 0.33% Disk: 3.26 GB used (limit 1006.85 GB) Update available

Figure 6. Docker network creation – defining a custom bridge network named my-web-net.

Figure 6 shows the execution of the network creation command. The new network, my-web-net, will isolate our container’s traffic from the default network and will allow multiple containers to communicate via this network by referring to each other’s container names.

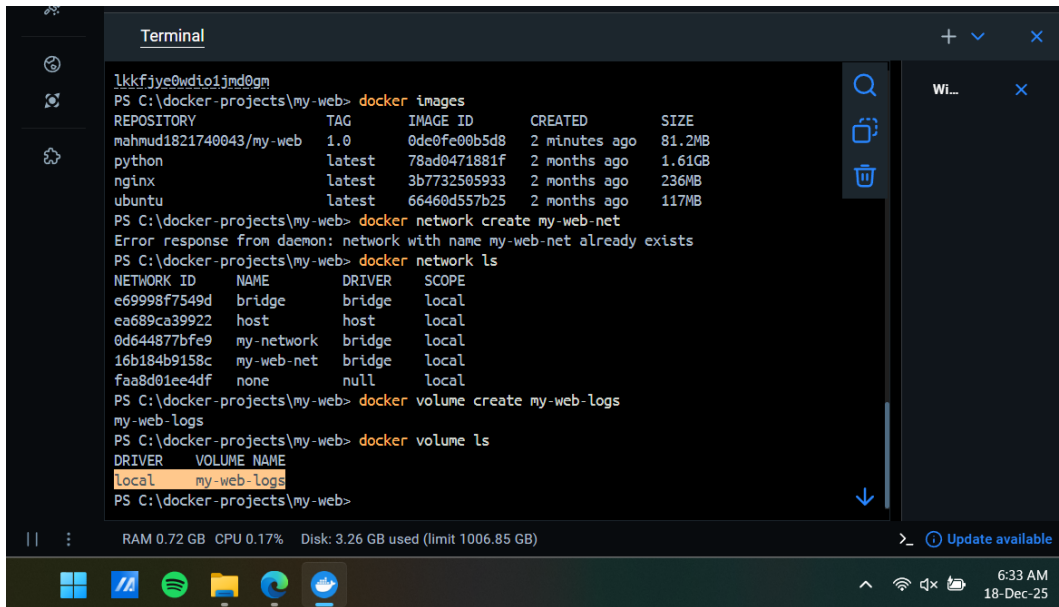
3.3 Docker Volumes (Creating and Mounting a Volume)

Containers are designed to be ephemeral; any data written to a container’s file system is lost when the container is removed or rebuilt. Docker volumes address this by providing persistent storage outside the container’s writable layer. A volume is a managed storage area that can be mounted into one or more containers, and it persists even if the container is deleted[10][11].

To create a Docker volume, the command used is:

```
docker volume create my-web-logs
```

This defines a named volume called html. The volume can then be mounted into a container’s filesystem space at a specified path. In our Nginx example, we can mount the volume at /usr/share/nginx/html, which is the location of the website files inside the container. By doing so, the container will serve content from the volume, and any changes to the volume will be reflected, even across container restarts.



```
lkkfjye0wdiojnd0gn
PS C:\docker-projects\my-web> docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
mahmud1821740043/my-web 1.0            0de0fe00b5d8   2 minutes ago  81.2MB
python              latest         78ad0471881f   2 months ago   1.61GB
nginx               latest        3b7732505933   2 months ago   236MB
ubuntu              latest        66460d557b25   2 months ago   117MB
PS C:\docker-projects\my-web> docker network create my-web-net
Error response from daemon: network with name my-web-net already exists
PS C:\docker-projects\my-web> docker network ls
NETWORK ID          NAME             DRIVER          SCOPE
e69998f7549d        bridge           bridge          local
ea689ca39922        host             host            local
0d644877bfe9        my-network       bridge          local
16b184b9158c        my-web-net       bridge          local
faa8d01ee4df        none            null            local
PS C:\docker-projects\my-web> docker volume create my-web-logs
my-web-logs
PS C:\docker-projects\my-web> docker volume ls
DRIVER              VOLUME NAME
local               my-web-logs
PS C:\docker-projects\my-web>
```

Figure 7. Docker volume creation and listing

Figure 7 demonstrates how to create a volume and then use the Docker volume ls command to list Docker volumes. The new html volume is now available for use. Storing the web page in a volume ensures that the content can remain independent of the container's lifecycle and can be updated without rebuilding the image [12]. This is beneficial during development (where code changes frequently) and in any scenario involving persistent data (such as user uploads or database files), instance [13][14].

3.4 Running the Container with Port Mapping, Network, and Volume

With the image built (section 3.1) and both a custom network (section 3.2) and volume (section 3.3) prepared, we now **run the Docker container** using all these components. The run command combines several options:

```
docker run -d --name my-web-container --network my-web-net -p 8080:80 -v my-web-logs:/var/log/nginx mahmud1821740043/my-web:1.0
```

In this command, -d starts the container in detached mode and --name my-web-container assigns a name to it. The -p 8080:80 option maps port 80 inside the container to port 8080 on the host machine, making the web page accessible via http://localhost:8080 on the host. The --network my-web-net flag attaches the container to the previously created custom network. The -v my-web-logs:/var/log/nginx option mounts the volume into the container's /usr/share/nginx directory (overriding the index.html that was included in the image). This means the container will serve whatever content is in the volume.

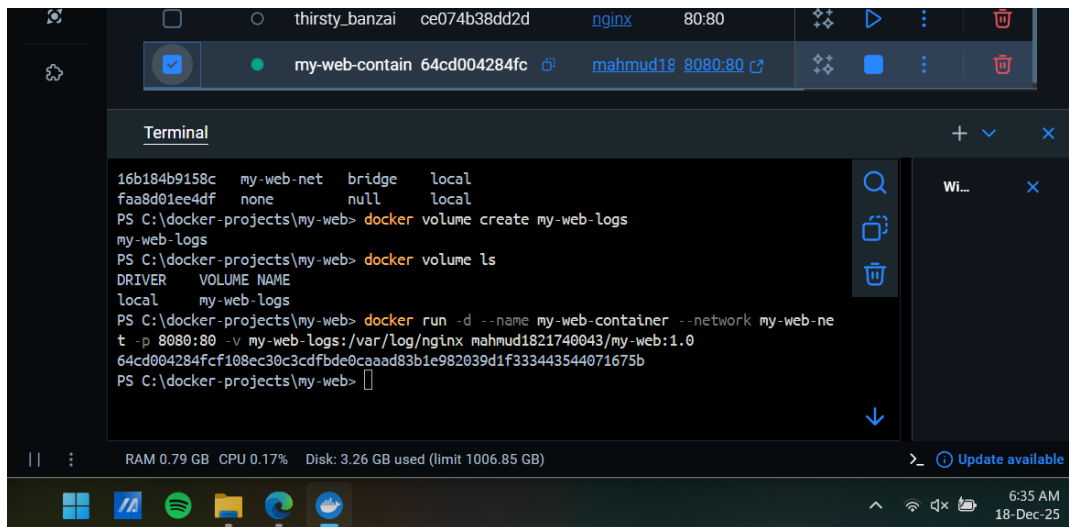


Figure 8. Running the container with port mapping, a custom network, and a mounted volume.

Once the container is running, we can verify the web server is functioning by accessing it from a browser. Since the container's port 80 is published to port 8080 on the host, the URL to test is

<http://localhost:8080>.

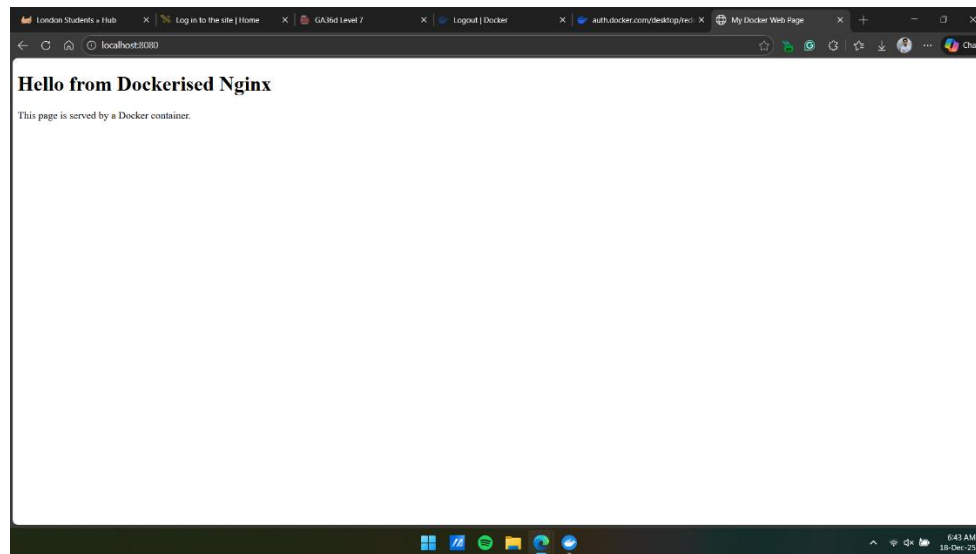


Figure 9. Browser output from the running container – the static homepage is served successfully.

Figure 9 shows the web page being served from the Docker container, confirming that the Dockerised Nginx server is working. The content displayed in the browser indicates that the entire pipeline, which builds the image, runs the container with the correct port, network, and volume

configuration, has been successful. This provides functional proof that containerisation of the application was successful.

3.5 Video Demonstration

YouTube: <https://youtu.be/4PULNJDHjD4?si=dZ6sYFdRuBoN9bix>

4. Load Balancing, Scalability, and Application Gateways in Cloud Computing

To understand how containerised applications (or any cloud-hosted services) are delivered at scale, it is essential to examine three core cloud concepts: load balancing, scalability (including autoscaling), and application gateways. In a typical production scenario, such as a customer-facing e-commerce web application, users access a frontend or API that is served by multiple backend instances for resilience and performance. Traffic management and security at the system's entry point are handled by load balancers and gateways, while scalability mechanisms ensure the system can handle varying levels of demand. These technologies work in concert to maintain availability, throughput, and security[16].

4.1 Load Balancing

Load balancing refers to the process of distributing incoming network traffic among two or more servers. In the cloud, a load balancer serves as a single point of contact for clients, distributing requests to the actual workhorses (backend servers). This increases service availability: if one of the instances fails, another can start fulfilling requests and prevent a single server from becoming a performance bottleneck.

Load balancing is commonly available as a managed service from cloud providers. For instance, Amazon Web Services (AWS) provides the service Elastic Load Balancing, including two types: the Application Load Balancer (ALB), which deals with the HTTP/HTTPS (Layer 7 Layer 4) traffic, and the more performant TCP/UDP Network Load Balancer cases (HTTP/HTTPS). ALB can learn routes based on HTTP attributes (e.g., URL path, host header), allowing for more advanced use cases, such as routing traffic to multiple microservices based on the request path. 1.9 What is the difference between a Layer 4 and a Layer 7 load balancer? Microsoft Azure offers services such as Azure Load Balancer for Layer 4 load balancing and Application Gateway for Layer 7 load balancing, which includes web application firewall (WAF) functionality. In both cases, in AWS/Azure, the Load Balancer uses health checks (also known as heartbeat requests) to determine the status of backend instances; traffic is automatically redirected from any instance it detects as unhealthy. This guarantees high availability, as it sends users only to healthy application instances.

4.2 Scalability and Autoscaling

Scalability is the system's ability to handle increasing load by adjusting its resources. In cloud environments, **horizontal scaling** is more commonly used than vertical scaling (adjusting the size of an individual instance) because it tends to improve resilience and can be automated. Autoscaling refers to the automatic adjustment of the number of running instances based on demand, according to predefined policies or real-time metrics. For example, an autoscaling policy might specify adding a new server instance when CPU usage remains above 70% for several minutes or removing instances when demand decreases.

On AWS, this functionality is provided by **Auto Scaling Groups** for virtual machines (EC2 instances) and similarly for container services. An Auto Scaling Group monitors metrics and can launch or terminate EC2 instances to maintain the target performance. These groups integrate with load balancers, so that as new instances launch, they are automatically registered to the load balancer's target group, and as instances are terminated, they are deregistered. Azure's equivalent is Virtual Machine Scale Sets, which also integrate with Azure load balancers and can automatically scale out/in VMs based on metrics or schedules [12]. The net effect is a system that can dynamically adjust its capacity to match the current workload, maintaining performance during traffic spikes and controlling costs during periods of low traffic.

4.3 Application Gateways

An application gateway operates at Layer 7 and provides advanced request routing and policy enforcement at the edge of an application. While a load balancer may simply distribute traffic, an application gateway is more content-aware: it can terminate TLS (HTTPS) connections, inspect and route HTTP requests based on properties such as the URL path or headers, and enforce security rules. This makes application gateways especially valuable for microservices architectures and API-based services, where each request may need to be routed to a different service and where centralised SSL/TLS handling and threat protection are desired.

In Azure, the managed Azure Application Gateway provides these Layer 7 capabilities, including WAF integration for protecting against common web vulnerabilities [10]. In AWS, the Application Load Balancer covers many gateway-like features and can be combined with AWS WAF for similar protections. Additionally, AWS offers API Gateway for RESTful and WebSocket APIs, which serves as a specialised gateway for microservice APIs, handling authentication, request/response transformation, and throttling [13]. In summary, application gateways add an intelligent control point that goes beyond simple load distribution to also address security and routing concerns at the application protocol level.

5. Why Load Balancing and Scaling Work Together

Load balancing and autoscaling are complementary mechanisms in cloud architecture. Autoscaling ensures that sufficient resources (instances) are available to handle the current load. However, without a load balancer, clients would not automatically be aware of new instances or be able to distribute traffic to them. Conversely, a **load balancer** can spread traffic among multiple servers;

however, if all those servers are at capacity, the system will still be overloaded unless new capacity is added. When combined, the two create a powerful elasticity system: the load balancer continuously directs incoming requests to whatever instances are currently healthy and available, while the autoscaling system adjusts the number of those instances behind the load balancer in response to demand[16].

In practice, the integration works as follows: when the load increases, autoscaling launches new instances, which are automatically registered with the load balancer and begin receiving a share of incoming requests. The load balancer's health checks ensure only healthy instances get traffic. When the load decreases, autoscaling can terminate excess instances; the load balancer will stop sending traffic to those once they are marked for removal. This coordination ensures high availability, performance, and cost efficiency [8][11].

6. Benefits of Combining Load Balancing, Autoscaling, and Gateways

Utilising load balancing, autoscaling, and application gateways in conjunction yields significant benefits for real-world deployments. For an e-commerce application scenario, the combination provides:

- **High Availability and Resilience:** With multiple instances across load balancer target groups and autoscaling to replace unhealthy ones, the application can tolerate instance failures without downtime. Health probes and failover mechanisms reduce single points of failure.
- **Performance and Scalability:** Load balancers spread out user traffic, preventing overload on any single instance, while autoscaling adds capacity during peak periods to maintain responsive performance. This ensures a good user experience even under heavy load.
- **Security Enhancement:** Application gateways with an integrated WAF can block common web attacks (such as SQL injection or cross-site scripting from the OWASP Top 10 [14]) before they reach the application servers. They also centralise TLS certificate management and can enforce secure protocols and authentication at the edge.
- **Operational Efficiency and Cost Control:** Autoscaling aligns resource usage with demand, enabling a pay-as-you-go model that avoids paying for idle servers [2][11]. Combined with the above technologies, this means the application can scale out when needed but also automatically scale in to reduce costs, all while users remain unaware of these behind-the-scenes changes.

In summary, the trio of load balancer, autoscaling, and application gateway is a common architectural pattern in cloud deployments that maximises uptime, optimises performance, fortifies security, and controls costs.

7. Serverless Computing vs. Containerisation

An emerging trend in cloud computing is **serverless computing**, often delivered as Function-as-a-Service (FaaS) platforms, such as **AWS Lambda** and **Azure Functions**. Serverless platforms allow developers to deploy code functions that are executed on demand in response to events (such as HTTP requests, message queue events, or file uploads) without managing any server infrastructure. The cloud provider automatically provisions runtime environments, scales the functions up and down, and charges only for actual execution time (typically metered in milliseconds).

When comparing serverless functions to containerisation, several trade-offs emerge:

- **Cost and Efficiency:** Serverless can be more cost-effective for **bursty** or infrequent workloads because there are no charges when the code is not running. Containers, on the other hand, typically involve having at least one instance running to serve requests, which incurs cost even during idle times. For services that constantly run with a stable load, containers may be more economical. However, for spiky or low-volume services, serverless often wins on cost.
- **Scalability and Agility:** Serverless platforms handle scaling each function invocation transparently, can scale out horizontally almost without limit, and the platform manages concurrency. This eliminates a significant amount of operational complexity. Container-based services typically require an orchestrator or manual scaling policies to scale, which involves more effort to set up and manage. However, containers can be orchestrated to scale very effectively, too, just with more configuration.
- **Startup Latency:** Containers are designed for long-running services, and once started, they provide consistent low-latency responses. Serverless functions may experience a **"cold start"** delay when a function is invoked after a period of inactivity, as the platform must spin up a new runtime environment for it. This can introduce latency for the first request, which might be a drawback for certain real-time applications.
- **Control and Flexibility:** With containers, developers have more control over the runtime environment, choosing specific OS libraries, custom configurations, and ensuring compatibility. Serverless functions run in a provider-managed environment with certain limitations. If an application requires custom software that isn't supported in a serverless runtime, a container might be necessary. Containers are also typically better suited for long-running processes, whereas serverless functions have time limits.

In practice, many organisations use a hybrid approach: core services and long-running applications may run in containers or VMs, while event-driven or intermittent tasks run as serverless functions. For example, a web application's primary backend could be containerised, but tasks like image processing, sending emails on a trigger, or CRON-like scheduled jobs could be offloaded to serverless functions for efficiency. These can be fronted by an API gateway which routes API calls either to containerised microservices or to serverless functions as appropriate [13][15][16]. This hybrid model combines the reliability and consistency of containerised services with the agility and cost savings of serverless for certain workloads.

8. Driving Forces Behind Cloud Adoption

Finally, it is worth noting the major drivers that have led enterprises to adopt cloud computing (and by extension, containerisation and serverless technologies on the cloud). Several key motivations are commonly cited [2][6]:

- **Cost Efficiency:** Cloud computing shifts IT spending from capital expenditure to operational expenditure. By leveraging elasticity, organisations can **right-size** their infrastructure and only pay for what they use. This is enhanced by autoscaling and serverless paradigms, which further optimise resource utilisation, eliminating expense on idle capacity. The overall result is often a lower cost of ownership, especially for variable workloads.
- **Agility and Time-to-Market:** The cloud's on-demand nature and automation capabilities enable much faster development and deployment cycles. Teams can provision infrastructure in minutes and easily incorporate new technologies, which accelerates innovation. Containerisation specifically contributes to agility by making deployment environments consistent across development, testing, and production, thereby reducing the infamous "it works on my machine" problems.
- **Flexibility and Innovation:** Cloud providers offer a rich ecosystem of services beyond basic VMs and containers, such as machine learning APIs, IoT platforms, and global content delivery networks, which companies can tap into without large upfront investment. This fosters innovation, as developers can readily experiment and integrate advanced capabilities. The rise of serverless is part of this story, as it allows developers to focus purely on code and business logic, while the platform handles scaling and infrastructure concerns.
- **Security and Compliance:** Major cloud vendors invest heavily in security measures and certifications. They provide built-in security features that can strengthen an organisation's security posture. While security in the cloud operates on a shared responsibility model, many organisations find that using cloud services enables them to improve their security and compliance more easily than on-premises, thanks to the tools and best practices provided by cloud platforms. For instance, using an application gateway with a WAF or container image scanning services can significantly reduce risk with relatively little effort.

In summary, containerization and Docker have become crucial within cloud computing, not only for the technical efficiencies they provide, but also because they align with broader drivers of cloud adoption. They enable cost savings through better resource utilisation, improve agility by simplifying deployments, and integrate well with emerging cloud services. Docker containers can be run on any major cloud (often managed by services like AWS ECS/Fargate, Azure Container Instances, Google Cloud Run, or Kubernetes clusters), making them a fundamental unit of modern cloud-native application design.

References

- [1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” NIST SP 800-145, 2011. [Online]. Available: nvlpubs.nist.gov. Accessed: 18 Dec. 2025.
- [2] L. M. Vaquero *et al.*, “A Break in the Clouds: Towards a Cloud Definition,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, 2009.
- [3] R. Goldberg, “Survey of Virtual Machine Research,” *Computer*, vol. 7, no. 6, pp. 34–45, Jun. 1974.
- [4] D. Bernstein, “Containers and Cloud: From LXC to Docker to Kubernetes,” *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, 2014.
- [5] Docker, “Docker Documentation – *Volumes and Networking*,” 2025. [Online]. Available: docs.docker.com. Accessed: 18 Dec. 2025.
- [6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, Omega, and Kubernetes,” *Commun. ACM*, vol. 59, no. 5, pp. 50–57, 2016.
- [7] NGINX, “NGINX Documentation – Serving Static Content,” 2025. [Online]. Available: nginx.org. Accessed: 18 Dec. 2025.
- [8] Amazon Web Services, “Elastic Load Balancing – Developer Guide,” 2025. [Online]. Available: docs.aws.amazon.com. Accessed: 18 Dec. 2025.
- [9] Microsoft, “Azure Load Balancer Documentation,” 2025. [Online]. Available: learn.microsoft.com. Accessed: 18 Dec. 2025.
- [10] Microsoft, “Azure Application Gateway Documentation,” 2025. [Online]. Available: learn.microsoft.com. Accessed: 18 Dec. 2025.
- [11] Amazon Web Services, “Amazon EC2 Auto Scaling – User Guide,” 2025. [Online]. Available: docs.aws.amazon.com. Accessed: 18 Dec. 2025.
- [12] Microsoft, “Virtual Machine Scale Sets Documentation,” 2025. [Online]. Available: learn.microsoft.com. Accessed: 18 Dec. 2025.
- [13] Amazon Web Services, “Amazon API Gateway Developer Guide,” 2025. [Online]. Available: docs.aws.amazon.com. Accessed: 18 Dec. 2025.
- [14] OWASP, “OWASP Top Ten Project – 2021 Edition,” 2023. [Online]. Available: owasp.org. Accessed: 18 Dec. 2025.
- [15] Amazon Web Services, “AWS Lambda – Developer Guide,” 2025. [Online]. Available: docs.aws.amazon.com. Accessed: 18 Dec. 2025.
- [16] Microsoft, “Azure Functions Documentation,” 2025. [Online]. Available: learn.microsoft.com. Accessed: 18 Dec. 2025.