

ACCA7027 – CLOUD ENGINEERING PRINCIPLES AND PRACTICE

The Principles and Benefits of Containerisation: Demonstration Dockerfile construction and Image Building

Deep Patel (2504308)

UNIVERSITY OF WALES TRINITY SAINT DAVID - Wales Institute of Science Arts – School
of Applied Computing

Table of Contents

1. Introduction	3
2. What is Containerisation?	4
2.1 From Physical Servers to Containers: The Evolution.....	4
2.2 The working of Docker Containerisation.....	4
2.3 Container Layers and the Union Filesystem	5
3. Principles of Containerisation.....	6
3.1 Immutability.....	6
3.2 Isolation	6
3.3 Portability.....	6
3.4 Ephemerality.....	6
3.5 Layered Build System.....	7
4. Benefits of Containerisation	8
5. Constructing a Dockerfile to build an Image.....	9
5.1 Dockerfile Syntax and Instructions.....	9
5.2 Class Exercise: Nginx Web Server Dockerfile (L10).....	10
5.3 Step-by-Step Walkthrough of Each Instruction	11
5.3.1 FROM Ubuntu: 24.04	11
5.3.2 RUN apt-get update	11
5.3.3 RUN apt-get install -y nginx.....	11
5.3.4 COPY ./nginx.conf /etc/nginx/nginx.conf	12
5.3.5 COPY ./www-data /usr/share/nginx/html.....	12
5.3.6 EXPOSE 80	12
5.3.7 CMD ["nginx"].....	12
5.4 Class Exercise: Node.js Application Dockerfile (L10)	13
5.4.1 WORKDIR /usr/src/app	14
5.4.2 Two-stage COPY Pattern (The Most Important Optimisation).....	14
5.5 CMD vs ENTRYPOINT: A Critical Distinction	14
5.6 The .dockerignore File	15
5.7 Construction of the Image: Commands and Process.....	16
5.8 Running a Container based on the Image.	17
5.9 Managing Images: Listing, Removing, and Pruning.....	18
6. Video demonstration section	20

7. Conclusion	21
References.....	22

List of Figures

Figure 1 Conceptual representation of Docker image layers for the Nginx Dockerfile	5
Figure 2 Nginx Dockerfile from L10 class exercise (original form).....	11
Figure 3 Node.js application Dockerfile from L10 class exercise (Slides 25–32)	14
Figure 4 dockerignore file with patterns from L10 and common additions.....	16
Figure 5 Building and verifying a Docker image (L10 class exercise commands)....	17
Figure 6 Running the Nginx container and verifying web server functionality	18
Figure 7 Docker image lifecycle management commands from L10.....	19

List of Tables

Table 1 Evolution of application deployment models from bare-metal to containerisation	4
Table 2 Key benefits of containerisation with Docker	8
Table 3 Complete reference for Dockerfile instructions covered in L10 class exercises	9
Table 4 CMD vs ENTRYPOINT: key differences and usage guidance	14

1. Introduction

The fast rate of development of cloud computing has completely transformed the way software applications are developed, packaged and deployed. The introduction of containerisation - a lightweight concept of application packaging - has become one of the most important technological changes in the recent years [1]. Containerisation and Docker in particular have come to be considered the bedrock of the modern DevOps practice allowing teams to build applications once and execute them anywhere: on the laptop of a developer, or on a testing server, or on a globally distributed cloud platform [2].

The report provides a critical and practically based analysis of the principles and benefits of containerisation with a particular focus on demonstrating how the Dockerfiles are built to create the container images. The real-life examples are directly based on the class work, which is presented in the lecture material of the ACCA7027 (L10 -Container Images: Building and Running) lecture course [3], complemented with academic and industry references.

The report begins by contextualising containerisation to the greater development of software deployment, starting with bare-metal servers and then moving on to virtualisation and finally to containers. It then proceeds in a systematic analysis of the main principles underlying the containerisation technology, evaluates its key benefits and provides a detailed, annotated walkthrough of a real construction of a Dockerfile entry, including the example of the Nginx web server and the example of Node.js applications used in the lecture and explaining what each line is actually doing, why the line is written the way it is written and how the line is related to the underlying container architecture.

2. What is Containerisation?

2.1 From Physical Servers to Containers: The Evolution

To have the full picture of the process of containerisation, it would be good to have a picture of the models of deployment that preceded it. The successive generations were sired to correct the inefficiencies of the previous one [4].

Table 1 Evolution of application deployment models from bare-metal to containerisation

Era	Model and Key Characteristics
1990s–2000s	Physical (Bare-Metal) Servers: One application per server. Secure and isolated, but massively inefficient — servers averaged 5–15% CPU utilisation. Provisioning took weeks [4].
2000s–2010s	Virtual Machines (Hypervisors): Multiple VMs on one physical host, each with its own full OS. Dramatically improved utilisation, but each VM carries a full OS copy (GBs of overhead), boots slowly (minutes), and the hypervisor adds a performance layer [5].
2013–present	Containers: Share the host OS kernel; bundle only the application and its dependencies. Start in milliseconds, occupy megabytes, and provide process-level isolation with negligible overhead [2].

Containers Containerisation became accessible and viable to mainstream software development courtesy of the technology that became known as Docker, released by Docker Inc. in March 2013. Prior to Docker, container-like systems were present in Linux (LXC, chroot jails), but were complicated to use. Docker simplified this complexity under a simple and developer friendly CLI and the Dockerfile format [2].

2.2 The working of Docker Containerisation.

A Docker container is a running process of a Docker image. The analogy between the two is a class and an object in object-oriented programming: the image is the immutable template (the class), and the container is the live and running instance (the object) [3]. Docker achieves container isolation using two Linux kernel features [6]:

Namespace: Isolate system resources - each container is given its view of the process tree (PID namespace), network interfaces (NET namespace), filesystem mounts (MNT namespace), hostname (UTS namespace) and inter-process communication (IPC namespace). This means that any processes in a container cannot see or interfere with processes in other containers or on the host.

Control Groups (cgroups): Control resource limits - CPU time, memory, disk I/O and network bandwidth can each be limited per-container. That will prevent the occurrence of a single runaway container starving the host or any other containers of resources.

Notably, there is no copy of the OS kernel in containers - they share the kernel of the host. That is what makes them so light when compared to VMs, but also means that all the containers on a host must be running on the same architecture as the OS kernel [6].

2.3 Container Layers and the Union Filesystem

The layered image system, which is a Union Filesystem (like OverlayFS on a modern Linux system), is one of the most architecturally significant features of Docker. Every command in a Dockerfile forms a new, read-only layers of filesystem. They are stacked atop one another and Docker exposes them to the container as a single unified filesystem [7].

Conceptual: Docker image layer stack for the Nginx example

Layer 5 (top) ← CMD ["nginx", "-g", "daemon off;"]

Layer 4 ← EXPOSE 80

Layer 3 ← COPY ./www-data /usr/share/nginx/html

Layer 2 ← COPY ./nginx.conf /etc/nginx/nginx.conf

Layer 1 ← RUN apt-get update && apt-get install -y nginx

Layer 0 (base) ← FROM ubuntu:24.04

Each layer stores only the DIFF from the layer below (delta)

Layers are cached: if a layer has not changed, Docker reuses the cache

Multiple images can SHARE base layers, saving disk space

Figure 1 Conceptual representation of Docker image layers for the Nginx Dockerfile

The implications of this stratum of approach are significant in practice. When you rebuild by just replacing the application code, Docker will reuse the cache of the OS and dependency layers - only recreating the changed layers. With a large application and a large number of dependencies, this can decrease rebuild time by minutes to seconds [7].

When a container is being run, Docker will add a single additional writable layer (the container layer) atop the read-only image layers. Everything that is done inside a running container (files created, logs written, databases updated, etc.) is written to this writable layer. This layer is lost when the container is removed (except data was written to a volume). The image layers in the background are not touched and can be shared by the required amount of containers [7].

3. Principles of Containerisation

There are a few basic design principles that drive containerisation over and above the past deployment paradigms. These principles are critical to understanding in designing efficient container-based architectures [8].

3.1 Immutability

A Docker image is read-only, once created, it cannot be changed. When a bug is patched, or a dependency is updated, a new version of the image is built; the previous one is not overwritten. This idea is revolutionary to software operations: instead of patching running servers (which introduces configuration drift, where the server in production is no longer the one as documented), containers are simply replaced with new ones built around a known, tested image [8].

3.2 Isolation

Each container runs in its own isolated environment - its own file system, network stack, and process tree and user space - with the courtesy of Linux namespaces. Containers on the same host do not know anything about each other unless it is explicitly connected together using Docker networking. This isolation provides security boundaries between applications and has been eliminated: Container A can then run Python 3.8 on the same host as Container B can run Python 3.12, and they do not conflict [9].

3.3 Portability

A Docker image is the sum of all that an application needs to be executable: its code, runtime, system libraries, environment variables and configuration files. This renders it absolutely portable. A program developed on a Windows or macOS workstation will run the same on a Linux CI/CD server or a cloud Kubernetes cluster. It will mean that this notorious it works on my machine issue that software teams have battled against in decades will be eliminated [2].

In the L10 lecture material, the central philosophy of Docker is Build, Ship and Run - developers build the image of the container as a part of their development cycle, ship it via a registry (Docker Hub) and run it in any environment and are confident that it will act in the same manner [3].

3.4 Ephemerality

Containers are designed ephemeral -short-lived and disposable. Containers are expected to be deployed, operated and destroyed with no penalty compared to traditional servers which are highly maintained and never restarted. Any continuing data should be packed in volumes not a component of the container. Elastic scaling can also be applied, i.e. spinning up 50 additional containers to handle a traffic spike,

and then destroying them when the spike was over is a common pattern of operation [10].

3.5 Layered Build System

The layered file system, as discussed in Section 2.3, is more than an implementation detail, but a concept. It promotes developers to order Dockerfiles in such a way that the most stable, seldom-changing layers (base OS, system dependencies) are listed first, and the most frequently-changing layers (application code) are listed last. This maximises the use of the cache and reduces the build times a key principle in fast CI/CD pipelines [7].

4. Benefits of Containerisation

The principles described above translate into concrete, measurable benefits that explain why containerisation has achieved near-universal adoption in modern software engineering [1][2][9]:

Table 2 Key benefits of containerisation with Docker

Benefit	Explanation and Evidence
Speed and Agility	Containers start in milliseconds versus minutes for VMs. A development team can run hundreds of containers on a standard laptop. This speed transforms workflows: developers can spin up local replicas of complex multi-service architectures instantly [2].
Consistency Across Environments	The same image runs identically in development, testing, staging, and production. This eliminates entire categories of environment-related bugs — a leading source of deployment failures in traditional operations [8].
Resource Efficiency	Containers share the host OS kernel and have no hypervisor overhead. A server that might run 10–20 VMs can run hundreds of containers simultaneously, yielding dramatically better hardware utilisation and reduced cloud costs [5].
DevOps and CI/CD Integration	Dockerfiles are plain-text files committed to version control alongside application code. As the L10 lecture notes, 'the work for setting up the infrastructure is now the responsibility of the developer' — and it can be handed to QA alongside the code [3]. This integration is the foundation of modern CI/CD pipelines.
Micro services Architecture	Containers are the natural unit of deployment for micro services. Each service — user authentication, payment processing, product catalogue — runs in its own container with its own dependencies, independently deployable and scalable [10].
Scalability	Container orchestration platforms (Kubernetes, Docker Swarm) can automatically scale services up or down in response to demand, adding or removing container instances in seconds. This is the foundation of cloud-native scalability [10].
Simplified Management Dependency	All application dependencies are packaged inside the image. There is no need to install runtimes or libraries on the host server, and no risk of version conflicts between applications sharing a host [9].
Reproducibility and Auditability	Because images are immutable and built from a versioned Dockerfile, every version of an application that was ever deployed can be precisely reproduced. This is invaluable for debugging production issues and regulatory compliance [8].

5. Constructing a Dockerfile to build an Image

The basic artefact of Docker containerisation is a Dockerfile. It is a plain-text file, named Dockerfile with no file extension, containing a sequence of instructions to be followed by the Docker Engine to build an image [3][11]. Each command forms a novel immutable layer on the resulting image. Due to the nature of each instruction, syntax, and behaviour, it is necessary to understand the purpose, syntax, and behaviour of each individual instruction in order to build efficient, correct container images.

5.1 Dockerfile Syntax and Instructions

The Docker Engine executes the instructions in Dockerfile sequentially, top to bottom. Every instruction is structured in the following way: INSTRUCTION arguments. By convention, instructions are case-insensitive, but are written in UPPERCASE to distinctly differentiate between instructions and arguments [11]. Figure 4 will serve as a full reference to the instructions to be utilized in class:

Table 3 Complete reference for Dockerfile instructions covered in L10 class exercises

Instruction	Purpose, Syntax, and Notes
FROM	Mandatory first instruction. Specifies the base image from which to build. Images are pulled from Docker Hub if not available locally. Syntax: FROM <image>[:<tag>] Example: FROM ubuntu:24.04
RUN	Executes a command in a new layer on top of the current image during the build process. The result is committed as a new image layer. Syntax: RUN <command> Example: RUN apt-get update && apt-get install -y nginx Multiple commands can be chained with && \ to avoid creating unnecessary layers [11].
COPY	Copies files or directories from the build context (the host machine) into the image filesystem. The src path is relative to the build context; dest is an absolute container path or relative to WORKDIR. Syntax: COPY <src> <dest> Preferred over ADD for local file copying [3].
ADD	Functionally similar to COPY, but also supports remote URL sources and automatic extraction of compressed archives (.tar files). Best reserved for tar extraction; COPY should be used for all other file copying [3][11].
EXPOSE	Documents the port number(s) the container listens on at runtime. Does NOT publish the port to the host — that requires the -p flag in docker container run. Syntax: EXPOSE <port> [<port>...] Example: EXPOSE 80

CMD	Sets the default command executed when a container starts. Only the last CMD instruction takes effect. Can be overridden entirely by providing a command in docker container run. Syntax (JSON array, preferred): CMD ["executable", "param1"] Syntax (shell form): CMD command param1 [11]
ENTRYPOINT	Defines the executable that always runs when the container starts, regardless of what arguments are passed at runtime. Unlike CMD, it cannot be overridden — only appended to. Use for the container's primary process [11].
WORKDIR	Sets the working directory for all subsequent RUN, CMD, ENTRYPOINT, COPY, and ADD instructions. Creates the directory automatically if it does not exist. Syntax: WORKDIR /path/to/directory [3]
ENV	Sets environment variables inside the container. Variables set with ENV are available both during the build process and at container runtime. Syntax: ENV <key>=<value> Example: ENV NODE_ENV=production [3]

5.2 Class Exercise: Nginx Web Server Dockerfile (L10)

The Dockerfile shown below is directly taken within the class exercise in Lecture 10 (L10 - Container Images) [3]. It creates a bespoke image of Nginx web server (based on Ubuntu 24.04) and installs Nginx, adds configuration and web content, and sets up the container to serve HTTP traffic on port 80.

dockerfile ← my-nginx/dockerfile (L10 class exercise)

```
FROM ubuntu:24.04
```

```
RUN apt-get update
```

```
RUN apt-get install -y nginx
```

```
COPY ./nginx.conf /etc/nginx/nginx.conf
```

```
COPY ./www-data /usr/share/nginx/html
```

```
EXPOSE 80
```

CMD ["nginx"]

Figure 2 Nginx Dockerfile from L10 class exercise (original form)

5.3 Step-by-Step Walkthrough of Each Instruction

The following is a step-by-step analysis of each instruction in the L10 Nginx Dockerfile explaining what each instruction does and why it is written that way and how it contributes to the final container image. This walkthrough has a direct connection with the exercise of the lecture L10 entitled what is this file doing? (Slides 17–24) [3].

5.3.1 FROM Ubuntu: 24.04

This is the image that is pulled by default and defines the base image ubuntu: 24.04 pulls the official Ubuntu 24.04 LTS (Long-Term Support) image on Docker Hub. Each and every further instruction is added on top of this base layer. Ubuntu 24.04 offers a well-known, tested Linux platform with access to the APT package manager, which is needed in the next step to install nginx [11].

The tag 24.04 is extremely important: without a specific tag, Docker will default to: latest that may pull in a different version of Ubuntu on different machines, which violates the principle of reproducibility. Always fix base images to particular versions [8].

5.3.2 RUN apt-get update

This command runs the apt-get update command in a temporary container constructed upon the previous layer, and then commits the result as a new layer. apt-get update refreshes the package index - the local database of available packages and their versions provided by the package repositories of Ubuntu. The next apt-get install command would then attempt to install nginx based on a potentially out-of-date package list, which may fail or install an older version [11].

The RUN command forms a new layer of the image. This is the best practice in production Dockerfiles to chain related commands with the help of && to minimise the number of layers and resulting image size. The reason as to why the L10 exercise uses two distinct RUN instructions to make it clear is quite acceptable.

5.3.3 RUN apt-get install -y nginx

This command mounts the Nginx web server package into the container image. The -y flag (or -yes) will automatically respond with yes to any query that requires a yes or no answer during installation, which is necessary in a non-interactive build environment - otherwise the build would hang awaiting some form of keyboard input that can never arrive [11].

Once this instruction is run, the Nginx binary, default configuration files, and all its dependencies are available in the image. Nginx is not running yet, it is a build step, not a run step.

5.3.4 COPY ./nginx.conf /etc/nginx/nginx.conf

This command is a copy of a custom Nginx configuration file in the build context (the directory on a host machine where the dockerfile resides, here my-nginx/) into the container image at the path /etc/nginx/nginx.conf - the standard location where Nginx looks to find its configuration [3].

This is an overlay of the default Nginx setup with a customised one such that the developer can have an accurate control of the behaviour of Nginx: what sites to serve, performance settings, logging configuration etc. The relative path of the build context to the build context directory is the src path of the build context.

5.3.5 COPY ./www-data /usr/share/nginx/html

This command copies the entire contents of the www-data directory in the build context into /usr/share/nginx/html within the image - the default root directory which Nginx serves web files. The custom HTML, CSS, JavaScript and image files to the site are packed into the container image [3].

This step is the gist of what makes this a customised image of a web server, in contrast to a generic Nginx image: the content of the web site is baked into the image itself, making the container a fully self-sufficient deployable system, with no external file system dependencies

5.3.6 EXPOSE 80

EXPOSE 80 files that are monitored by the application running within this container on TCP port 80 (the default HTTP port). It should be noted - as L10 lecture draws attention to - that EXPOSE does not really publish the port to the host machine. It is utilized both to communicate intent to other developers and tools reading the Dockerfile, and to provide container-to-container communication within a Docker network [3].

To actually make the port accessible in the host machine or external network, the -p flag will be required when running the container: docker container run -p 8080:80 my-nginx maps host port 8080 to container port 80.

5.3.7 CMD ["nginx"]

CMD sets the default command which runs when a container is launched off this image. The JSON array form that is the direct executable of the nginx executable (exec form) without the use of a shell is the CMD forms nginx. This is the most desirable form since it allows signals (such as SIGTERM to gracefully shut down a process) to be passed on directly to nginx process [11].

Since the L10 lecture notes on Slide 24 [3], since we are using CMD, the command can be overridden by simply stating a different command in the docker container run

command - e.g., `docker container run my-nginx /bin/bash` would start a shell instead of Nginx. The lecture notes that this might not be desirable and that `ENTRYPOINT` could have been the better choice of a production picture since it makes the key purpose of the container fixed.

5.4 Class Exercise: Node.js Application Dockerfile (L10)

The second sample of Dockerfile to containerise Node.js web application, which was given in the L10 lecture (Slides 2532) [3], is a more sophisticated pattern. It presents the npm package manager `WORKDIR` as well as a two-stage `COPY` pattern, the basis of efficient Node.js container builds:

dockerfile ← Node.js application (L10, Slides 25–32)

```
FROM node:carbon

# Set the working directory inside the container
WORKDIR /usr/src/app

# Copy package.json (and package-lock.json if it exists) first
# The wildcard package*.json matches both files
COPY package*.json .

# Install Node.js dependencies
RUN npm install
# OR: RUN npm install --only=production

# Copy the rest of the application source code
COPY . .

# Expose the application port
EXPOSE 8000

# Start the application
```


 CMD ["npm", "start"]

Figure 3 Node.js application Dockerfile from L10 class exercise (Slides 25–32)

This Dockerfile introduces two important concepts beyond the Nginx example:

5.4.1 WORKDIR /usr/src/app

WORKDIR The working directory of the next instructions is `/usr/src/app`. In case this directory is not present within the image, Docker automatically creates this directory. Any relative path in later **COPY** and **RUN** commands is determined relative to this directory. It is highly recommended that **WORKDIR** is used instead of **RUN mkdir && cd /path** patterns, which are weak and do not survive across build steps [3][11].

5.4.2 Two-stage COPY Pattern (The Most Important Optimisation)

The architecturally noteworthy feature of this Dockerfile is the explicit segregation of **COPY** operations:

COPY package*.json . - will only copy the package manifest files (`package.json` and optionally `package-lock.json`) first, before any application code is copied.

RUN npm install — Runs all dependencies in `package.json`.

copy . . — copies the contents of the whole application source code to the container.

The slowest step of a Node.js build is typically `npm install` (installing all Node.js dependencies) which can take several minutes, and requires downloading hundreds of dependencies. By placing it after copying only `package.json` (which rarely changes), this layer is cached and reused on every rebuild as long as `package.json` has not changed. `npm install` will only re-run `package.json` when it has been changed [7].

When **COPY. .** is used, such that `npm install` runs afterwards, any modification of any source file, even a one-line bug fix, will cause the `npm install` cache layer to become invalid and will re-download all dependencies each and every time a single build run occurs. This is not at all done with two stage pattern.

5.5 CMD vs ENTRYPOINT: A Critical Distinction

Since both **CMD** and **ENTRYPOINT** determine what is run each time a container is started, the difference between the two is often misconstrued. The main differences are summarized, depending on the L10 lecture content [3] and the Docker training content [11]:

Table 4 CMD vs ENTRYPOINT: key differences and usage guidance

Aspect	CMD vs ENTRYPOINT Behaviour
--------	-----------------------------

Override Behaviour	CMD: The entire command can be replaced by passing a command to docker container run. Example: <code>docker run my-app /bin/bash</code> starts bash instead of the CMD command. ENTRYPOINT: Cannot be overridden by a normal docker run argument. Only the <code>--entrypoint</code> flag can override it.
As a Default	CMD: Acts as the default command or default arguments. If no command is given at runtime, CMD runs. ENTRYPOINT: Always runs. Arguments passed at runtime are appended to the ENTRYPOINT command.
Combined Usage	Best practice: ENTRYPOINT ["nginx"] defines the process; CMD ["-g", "daemon off;"] provides default flags. Runtime users can change the flags (CMD) but not the process (ENTRYPOINT).
Syntax	Both support JSON array (exec) form: CMD ["executable", "param1"] — preferred, runs without a shell, passes signals correctly. Shell form: CMD command param1 — runs via <code>/bin/sh -c</code> , which can cause signal handling issues [11].
Multiple Declarations	Only the last CMD or ENTRYPOINT instruction in a Dockerfile takes effect. Earlier ones are silently ignored [3].

5.6 The .dockerignore File

The dockerignore file, which is mentioned in L10, Slide 34 [3] is a plain-text file that is placed in the same directory as the Dockerfile. It defines file and directory patterns that must not be included in the build context passed to the Docker Engine when docker image build is called.

.dockerignore ← example file from L10

```
# Ignore all .txt files
*.txt

# Ignore all files EXCEPT .jar and .war files
* *
!*.jar
!*.war

# Ignore files and directories starting with 'temp'
```

```
temp*

# Ignore 'temp' directories two levels deep from root
**/temp*

# Common Node.js / Python additions
node_modules/
__pycache__/.git/
*.log
```

Figure 4 dockerignore file with patterns from L10 and common additions

The .dockerignore file is notable in two aspects. First, it enhances the performance of builds: Docker copies the whole directory of the build context to the Docker daemon before a build is run; including large directories (such as node_modules/ or .git/) in the build context will dramatically reduce the amount of data transferred. Second, it enhances security by ensuring that sensitive files (credentials, private keys, local configuration files) do not accidentally get copied into the image as a result of a broad COPY . . command [11].

5.7 Construction of the Image: Commands and Process.

Having written the Dockerfile, the image is created with the help of the docker image build command. As pointed out in L10, Slide 37 [3], there are a number of equivalent forms:

Terminal ← Building a Docker image (L10 exercise commands)

```
# Step 1: Create and navigate to the project directory

mkdir my-nginx
cd my-nginx

# Step 2: Create the dockerfile (no file extension) in this directory

# Add the nginx.conf and www-data/ files here too
```

```
# Step 3: Build the image

# These three commands are equivalent (L10, Slide 37):

docker image build .

docker image build --tag my-nginx:latest .

docker image build -t my-nginx .


# The trailing dot (.) = build context = current directory

# --tag / -t assigns a name and optional tag (version) to the image


# Step 4: Verify the image was built

docker image ls


# Expected output:

# REPOSITORY TAG IMAGE ID CREATED SIZE
# my-nginx latest a1b2c3d4e5f6 3 seconds ago 189MB
```

Figure 5 Building and verifying a Docker image (L10 class exercise commands)

As docker image build processes are executed, Docker processes the Dockerfile line by line. In each instruction it examines the existence of a layer in the cache (dependent on the instruction and the files used). In the case of a cache hit, Docker will report a cache hit and will not re-run. Otherwise it executes the instruction, builds a new layer and caches it to be used in future builds. The build output gives a clear perspective of the layering construction process [7].

5.8 Running a Container based on the Image.

After constructing the image, docker container run is used to create containers based on the image. The following commands illustrate how to start the Nginx web server and ensure that it is serving the custom HTML page:

```
Terminal ← Running the Nginx container and verifying the web server

# Run the container in detached mode (-d = background)

# Map host port 8080 to container port 80 (-p host:container)

# Name the container for easy reference (--name)
```

```
docker container run -d -p 8080:80 --name my-nginx-server my-nginx:latest

# List running containers

docker container ls

# Expected output:

# CONTAINER ID   IMAGE          COMMAND        CREATED        STATUS        PORTS
# 9f8e7d6c5b4a   my-nginx:latest "nginx"        5 seconds ago Up 4s         0.0.0.0:8080->80/tcp

# Open a browser and navigate to: http://localhost:8080

# The custom HTML page from www-data/index.html should appear

# View container logs

docker container logs my-nginx-server

# Stop and remove the container

docker container stop my-nginx-server

docker container rm my-nginx-server
```

Figure 6 Running the Nginx container and verifying web server functionality

5.9 Managing Images: Listing, Removing, and Pruning

The L10 lecture (Slides 39–41) [3] covers image lifecycle management. The following commands are the core of day-to-day Docker image management:

Terminal ← Image management commands (L10, Slides 39–41)

```
# List all local images

docker image ls

# Remove a specific image by name

docker image rm my-nginx
```

```
# Remove multiple images
docker image rm test1 test2:latest

# Force removal (even if a container is using the image)
docker image rm -f my-nginx

# Shorthand (older syntax, still valid)
docker rmi my-nginx

# Remove ALL unused images (none with a running container)
docker image prune

# Force prune without confirmation prompt
docker image prune -f

# NOTE from L10: A container using the image must be
# stopped and removed BEFORE the image can be removed.
docker container stop my-nginx-server
docker container rm my-nginx-server
docker image rm my-nginx
```

Figure 7 Docker image lifecycle management commands from L10

6. Video demonstration section

Video title: Demonstrating Dockerfile and Image building

Video link: <https://youtu.be/pw5GnTRQlyI?si=8wHdqyxf0DRrWZP1>

7. Conclusion

The report has presented a detailed analysis of the concepts and advantages of containerisation with a practical emphasis on creating Dockerfiles to build Docker images. The analysis started with contextualising containerisation in the lineage of bare-metal servers, through virtual machines, and demonstrating how containers address inefficiencies of each of the preceding systems.

Five fundamental tenets were discovered to be the intellectual foundations of containerisation: immutability, which ensures consistency and auditability; isolation, which removes dependency conflicts; portability, which realizes the promise of building once and running anywhere; ephemerality, which enables elastic scaling; and the layered build system, which makes rebuilds fast and CI/CD pipelines viable. All these principles can be used to elucidate the radical effect that containerisation has on the processes of software development and operations.

The realistic Dockerfile tutorial was inspired by the L10 lecture activities, walking through the Nginx web server and Node.js applications Dockerfiles step by step. Among the important lessons was the difference between CMD and ENTRYPOINT - where CMD gives an overridable default and ENTRYpoint defines an immutable process - and the critical two stage COPY pattern in the Node.js Dockerfile, which uses the layer caching capabilities of Docker to avoid the need to reinstall dependencies with every code change.

These principles and practices combined, lead to the fact that Docker has become the de facto standard of application packaging. Containerisation is not just an alternative to virtual machines that is more efficient; it fundamentally changes the developer-operations relationship, making infrastructure a versioned, testable, reproducible artefact instead of a fragile, hand-maintained server. In the case of cloud engineering, it is not an incremental improvement, but a paradigm shift.

References

- [1] N. Sultan, "Cloud computing: A democratizing force?," *International Journal of Information Management*, vol. 33, no. 5, pp. 810–815, 2013.
<https://doi.org/10.1016/j.ijinfomgt.2013.05.010>
- [2] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, Mar. 2014.
<https://dl.acm.org/doi/10.5555/2600239.2600241>
- [3] M. Huntly, "Cloud Technologies – Container Images: Building and Running," Lecture Notes L10, ACCA7027 Cloud Engineering Principles and Practice, University of Wales Trinity Saint David, 2025.
- [4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.
<https://doi.org/10.1016/j.future.2008.12.001>
- [5] M. Armbrust et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.
<https://doi.org/10.1145/1721654.1721672>
- [6] C. Anderson, "Docker [Software engineering]," *IEEE Software*, vol. 32, no. 3, pp. 102–c3, May–Jun. 2015.
<https://doi.org/10.1109/MS.2015.62>
- [7] Docker Inc., "Building best practices," Docker Docs. [Online]. Available: <https://docs.docker.com/build/building/best-practices/>. [Accessed: 11-May-2026].
- [8] S. P. Kane and K. Matthias, *Docker: Up and Running: Shipping Reliable containers in production*, 3rd ed. Sebastopol, CA: O'Reilly Media, 2023.
- [9] B. Burns, J. Beda, K. Hightower and L. Evenson, *Kubernetes: Up and Running: Dive into the Future of Infrastructure*, 3rd ed. Sebastopol, CA: O'Reilly Media, 2022.
- [10] G. Kim, J. Humble, P. Debois, J. Willis and N. Forsgren, *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*, 2nd ed. Portland, OR: IT Revolution, 2021.
- [11] Docker Inc., "Dockerfile Reference," Docker Docs. [Online]. Available: <https://docs.docker.com/reference/dockerfile/>. [Accessed: 11-May-2026].