Engineer

Docker

a way to ship a program with everything it needs

What it is

Docker is a toolset for packing a program, its libraries, its language runtime and its files into one artefact called an image, and then running that image as an isolated process called a container. The image is built once from a recipe, the Dockerfile, and the same image then starts on a laptop, on a build server and in production. That is the whole promise in one sentence: the thing that was tested is, byte for byte, the thing that runs.

Four nouns carry most of the vocabulary. A Dockerfile is a text file of instructions such as `FROM`, `COPY` and `RUN`. An image is a read-only template built from those instructions, stored as a stack of layers, one per instruction, so that a rebuild after a small change only redoes the layers that changed. A container is a running instance of an image with a thin writable layer on top. A registry, such as Docker Hub or a cloud provider's own, is where images are pushed and pulled.

Under the command line sits a client and a server. The `docker` command is a client; it sends requests over a REST API, usually through a Unix socket, to a long-running daemon called `dockerd`. The daemon owns images, containers, networks and volumes. For the container lifecycle itself it hands work down to containerd, and containerd by default uses runc, the small program that asks the Linux kernel for namespaces and control groups and starts the process inside them. A container is therefore an ordinary process on the host kernel with a restricted view of the system. It is not a virtual machine, and on macOS or Windows Docker Desktop runs a small Linux virtual machine to have a Linux kernel to share.

The image format outgrew the company that made it. In June 2015 Docker, CoreOS and others launched the Open Container Initiative, Docker donated its image format and runc to it, and the resulting specifications are why an image

built with Docker runs unchanged under Podman, containerd or a managed cloud service.

Use the concept in a decision

Put “What is Docker?” to work in a real decision. We can help relate the concept to the system you are building or buying. Packaged to behave identically on a laptop, on a test server and in production, and to scale up when a busy week arrives.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Ran 20 Docker apps on Google Cloud and AWS — 20 containerised applications deployed and watched

Ran 20 Docker apps on Google Cloud and AWS Get in touch →

Why it matters

The useful question is never “Docker or not” in the abstract. It is whether a given workload earns an image, or whether a plain service on the host would do the same job with fewer moving parts.

An image earns its place when the program drags an environment behind it. A Python or R application that needs one exact set of packages, a legacy service pinned to an old runtime, two programs that want conflicting versions of the same library: each of these is painful to reproduce by hand and trivial to reproduce from an image. Rolling back means starting the previous image. A new developer runs one command instead of following a page of setup notes. Docker vs a virtual machine comes down to the same trade from the other side: containers start in moments and share one kernel, so dozens fit on a host that would hold a handful of virtual machines, at the price of weaker isolation.

The cost is real infrastructure. By default the daemon runs as root and manages networking; Docker also supports rootless mode. There is a registry to run or pay for, base images that collect security fixes and need rebuilding to receive them, and a storage layer whose rules differ from the host’s. A single static binary with no dependencies gains very little from all that, and a sandboxed systemd unit runs it just as well.

The difference between Docker and Podman matters here too. Both run the same images and accept most of the same commands. Docker uses a daemon, privileged by default; Podman starts containers directly as children of the

command or of systemd. For a laptop the difference is small. For a server whose firewall and process supervision someone has already designed, it can decide the question.

What it looks like in practice

Docker appears across this site's work, and one of the projects is a deliberate decision to leave it out.

Between 2022 and 2024, twenty containerised applications ran across Google Cloud and AWS, among them analytics applications written in R, where reproducing the exact package set is the hard part. The twenty were deployed and maintained with consistent configuration and monitoring, which kept them stable and reproducible and made problems faster to diagnose across the clouds.

At larger scale, a fleet of a hundred Docker applications sat on Ubuntu hosts behind a mix of Apache and Nginx, and each needed a valid TLS certificate. An ACME-based workflow issued, renewed and deployed those certificates automatically, so all hundred kept valid certificates and the manual certificate work disappeared.

The third case points the other way. On a host whose firewall rules come from a hardening role, the container plane went onto Podman and Quadlet instead of Docker, because Docker's default networking rewrites the kernel's packet-filter rules and publishes container ports above the firewall's own. The reason is written down with the choice, so the next person to touch the host knows the default was weighed rather than forgotten.

Where it goes wrong

Docker makes the first container easy. The trouble arrives later, usually in one of these four forms.

A port the firewall blocks answers from the internet. By default a published port binds to every address on the host, and the packet-filter rules Docker writes send that traffic to the container before a tool such as ufw ever sees it. The firewall reports the port as closed while it is open. When only a local reverse proxy needs the container, publish to the loopback address:

```
# reachable from anywhere, whatever ufw reports
docker run -d -p 8080:80 nginx:1.27

# reachable only from the host itself
docker run -d -p 127.0.0.1:8080:80 nginx:1.27
```

Data vanishes with the container. Anything written to the container's own writable layer is deleted when the container is removed, which is exactly what an upgrade does. Databases, uploads and anything else that must survive go on a named volume or a bind mount, and that volume then needs a backup of its own.

A rebuild produces something different from last week. A tag such as `python:3` or `latest` is a moving pointer, so the same Dockerfile can build a different image on a different day. Pin base images to a specific version, or to a digest when the build must be exactly repeatable, and rebuild on purpose when the base image publishes security fixes.

A credential ships inside the image. Build arguments and environment variables persist in the final image, and a file copied in one layer stays in that layer even if a later instruction deletes it. Anyone who can pull the image can read it. Pass secrets to the build through a secret mount, which exposes them to a single `RUN` step without writing them into any layer.

Further reading

- Docker Docs — the official manuals, guides and reference for the engine, the command line and Compose.
- Docker overview — the client, the daemon, images and containers explained by the project itself.
- Port publishing and mapping — why published ports are open to the outside by default, and how to bind them to localhost.
- Packet filtering and firewalls — how Docker writes iptables or nftables rules and where ufw stops seeing the traffic.
- Build secrets — passing credentials to a build without leaving them in the image.
- OCI Image Format Specification — the open standard that lets one image run under Docker, Podman and containerd alike.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is Docker?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/docker/