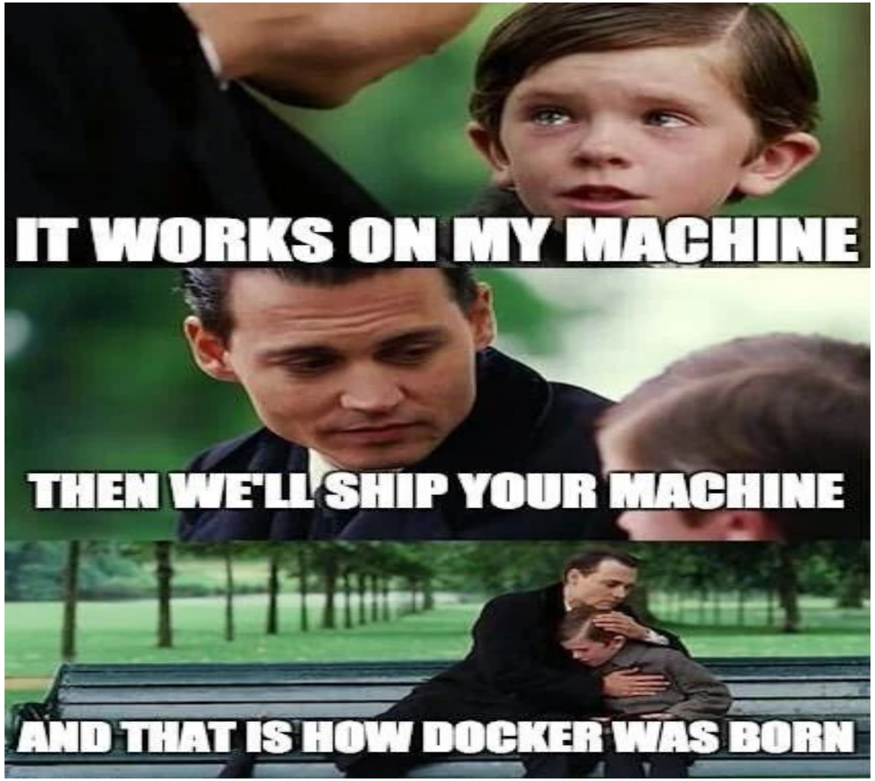
Docker Containers



What is a container?

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application (https://www.docker.com/resources/what-container/).

History of virtualization

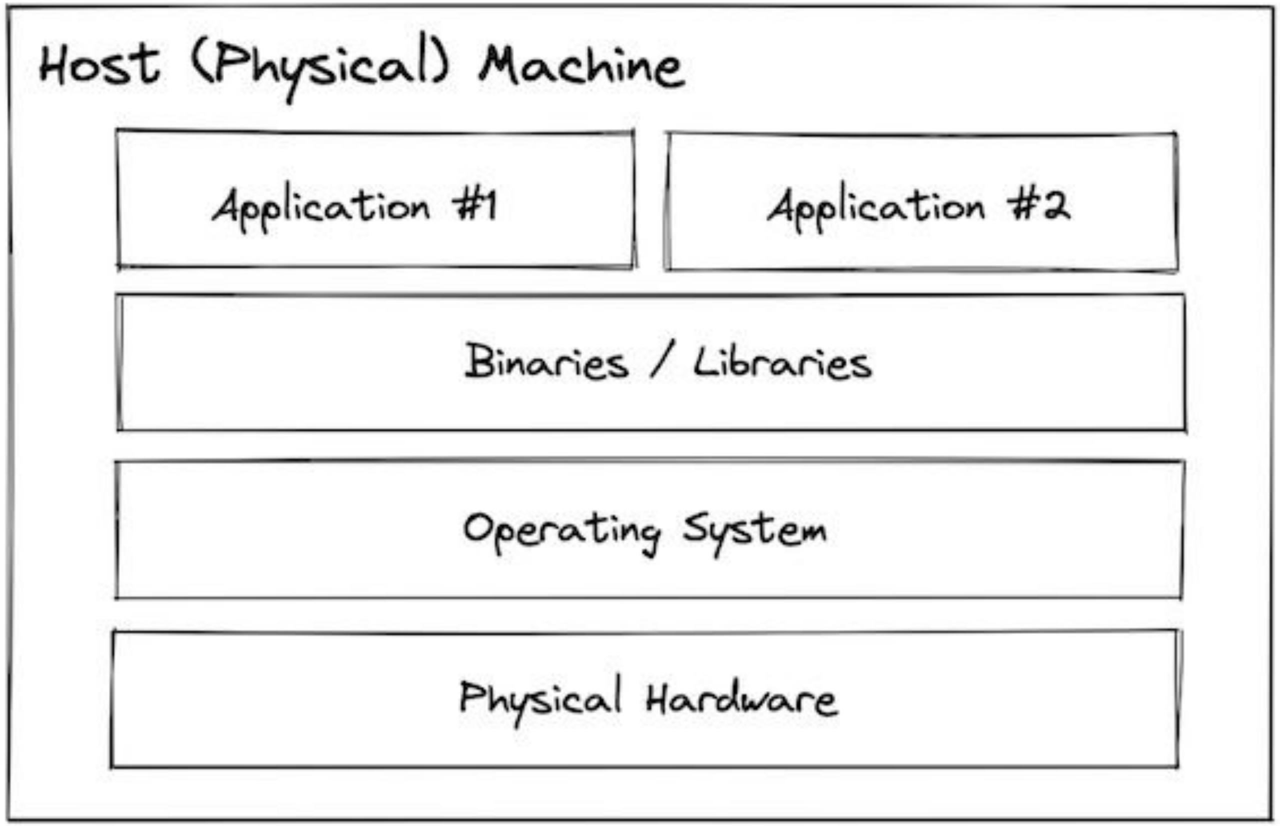
Bare Metal

Before virtualization was invented, all programs ran directly on the host system. The terminology many people use for this is "bare metal". While that sounds fancy and scary, you are almost certainly familiar with running on bare metal because that is what you do whenever you install a program onto your laptop/desktop computer.

With a bare metal system, the operating system, binaries/libraries, and applications are installed and run directly onto the physical hardware.

This is simple to understand and direct access to the hardware can be useful for specific configuration, but can lead to:

- Hellish dependency conflicts
- Low utilization efficiency
- Large blast radius
- Slow start up & shut down speed (minutes)
- Very slow provisioning & decommissioning (hours to days)

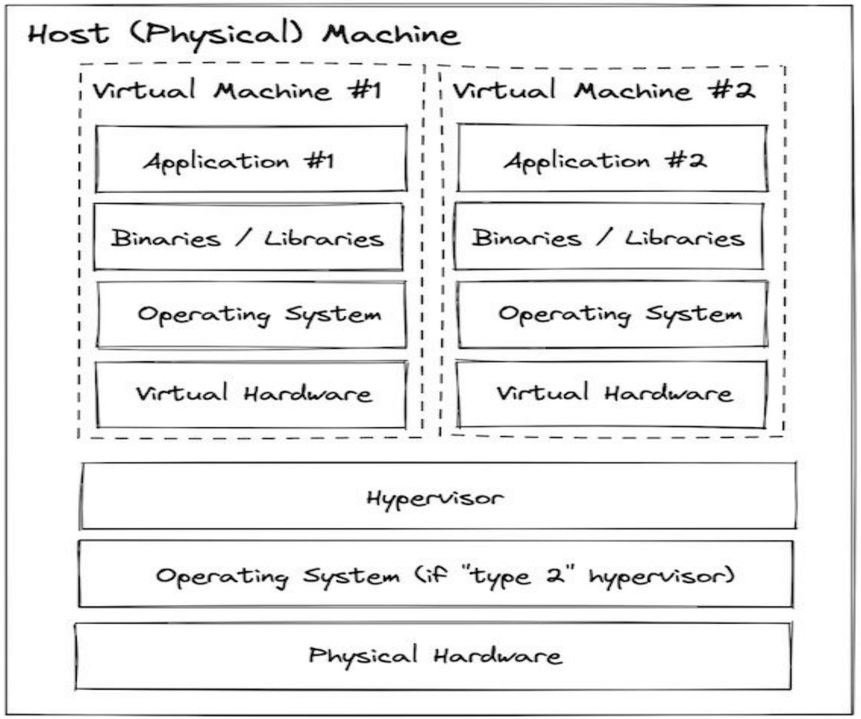

Host (Physical) Machine
Application #1
Application #2
Binaries / Libraries
Operating System
Physical Hardware

Virtual Machines

Virtual machines use a system called a "hypervisor" that can carve up the host resources into multiple isolated virtual hardware configuration which you can then treat as their own systems (each with an OS, binaries/libraries, and applications).

This helps improve upon some of the challenges presented by bare metal:

- No dependency conflicts
- Better utilization efficiency
- Small blast radius
- Faster startup and shutdown (minutes)
- Faster provisioning & decommissioning (minutes)

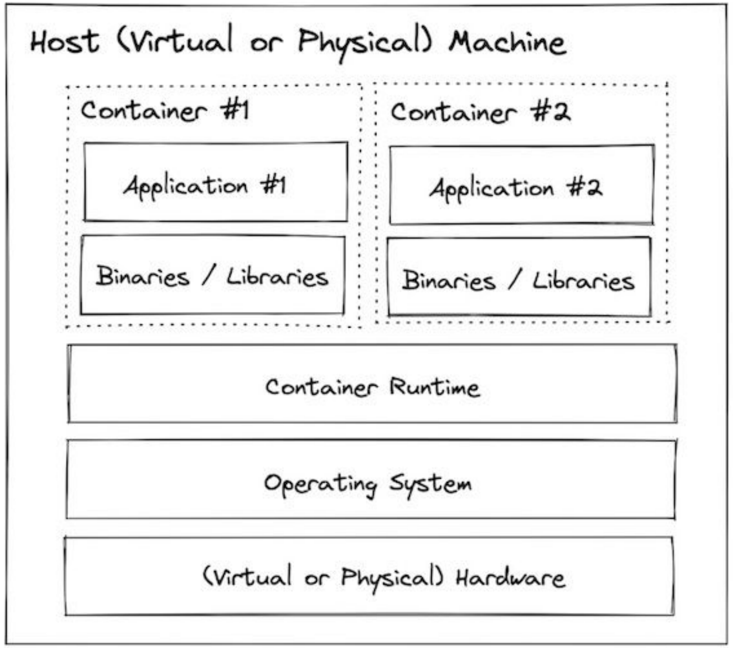


Containers

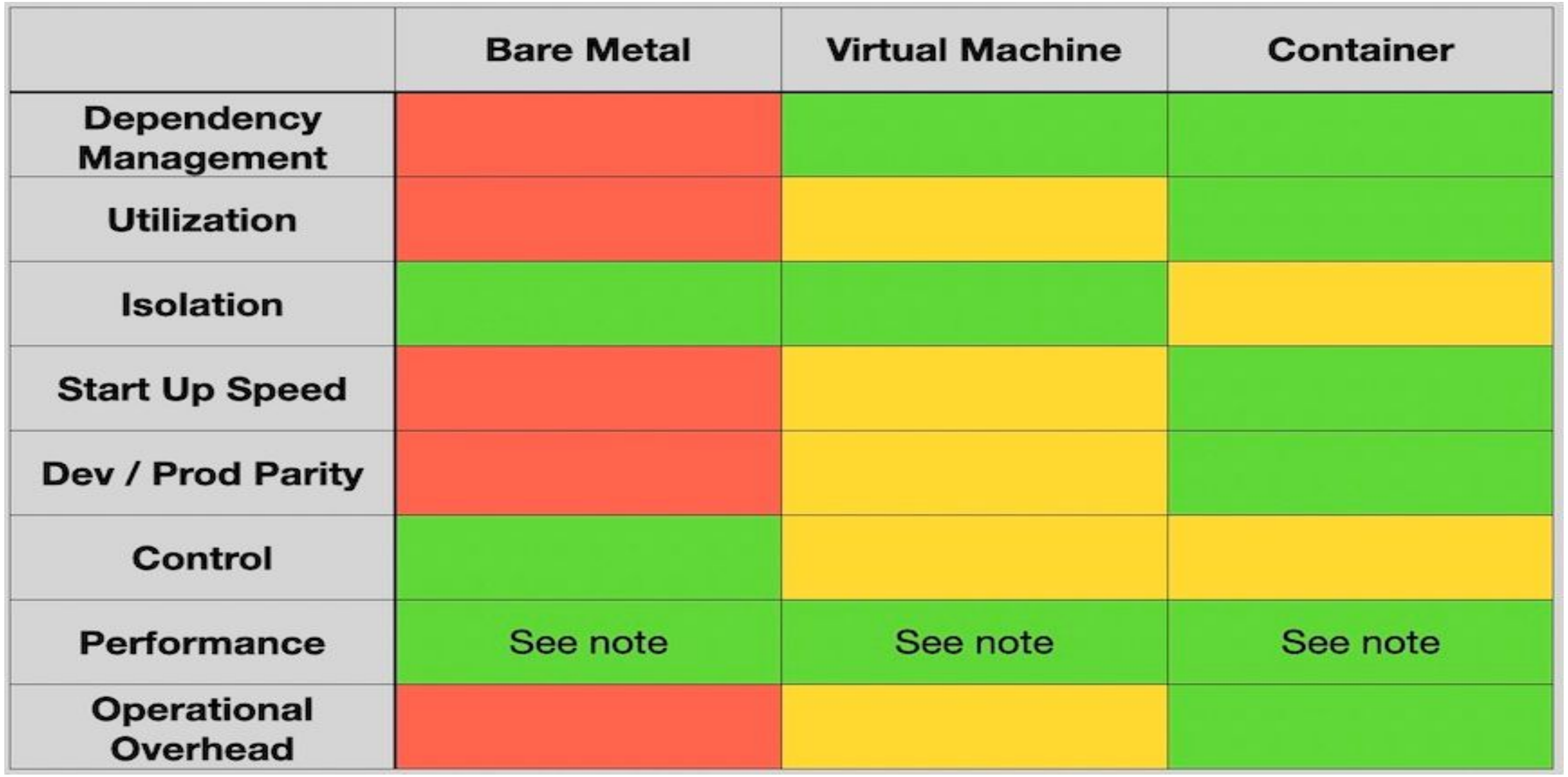
Containers are similar to virtual machines in that they provide an isolated environment for installing and configuring binaries/libraries, but rather than virtualizing at the hardware layer containers use native linux features (cgroups + namespaces) to provide that isolation while still sharing the same kernel.

This approach results in containers being more "lightweight" than virtual machines, but not providing the save level of isolation:

- No dependency conflicts
- Even better utilization efficiency
- Small blast radius
- Even faster startup and shutdown (seconds)
- Even faster provisioning & decommissioning (seconds)
- Lightweight enough to use in development!



Tradeoffs



	Bare Metal	Virtual Machine	Container
Dependency Management	🟥	🟩	🟩
Utilization	🟥	🟨	🟩
Isolation	🟩	🟩	🟨
Start Up Speed	🟥	🟨	🟩
Dev / Prod Parity	🟥	🟨	🟩
Control	🟩	🟨	🟨
Performance	🟩 See note	🟩 See note	🟩 See note
Operational Overhead	🟥	🟨	🟩

Technology Overview

Linux Building Blocks

Containers leverage linux kernel features cgroups and namespaces to provide resource constraints and application isolation respectively. They also use an union filesystem that enables images to be built upon common layers, making building and sharing images fast and efficient.

Note: Docker did not invent containers. For example, LXC containers (https://linuxcontainers.org/) was implemented in 2008, five years before Docker launched. That being said, Docker made huge strides in developer experience, which helped container technologies gain mass adoption and remains one the most popular containerization platforms.

Cgroups

Cgroups are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored.

Namespaces

A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes.

Union filesystems

A union filesystem allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.

Contents of directories which have the same path within the merged branches will be seen together in a single merged directory, within the new, virtual filesystem.

This approach allows for efficient use of space because common layers can be shared. For example, if multiple containers from the same image are created on a single host, the container runtime only has to allocate a thin overlay specific to each container, while the underlying image layers can be shared. More detail on understanding the implications of these filesystem on data persistence can be found in 04-using-3rd-party-containers.

Docker Application Architecture

It is useful to break down the various components within the Docker ecosystem. The first distinction to make is between "Docker Desktop" and "Docker Engine".

Docker Desktop is an application you install on development systems that provides:

- A client application:
  - Docker CLI (command line interface for interacting with Docker)
  - GUI for configuring various system settings
  - Credential helpers for accessing registries
  - Extensions (3rd party plugins)
- A Linux virtual machine containing:
  - Docker daemon (dockerd), exposing the Docker API
  - (Optional) Kubernetes cluster

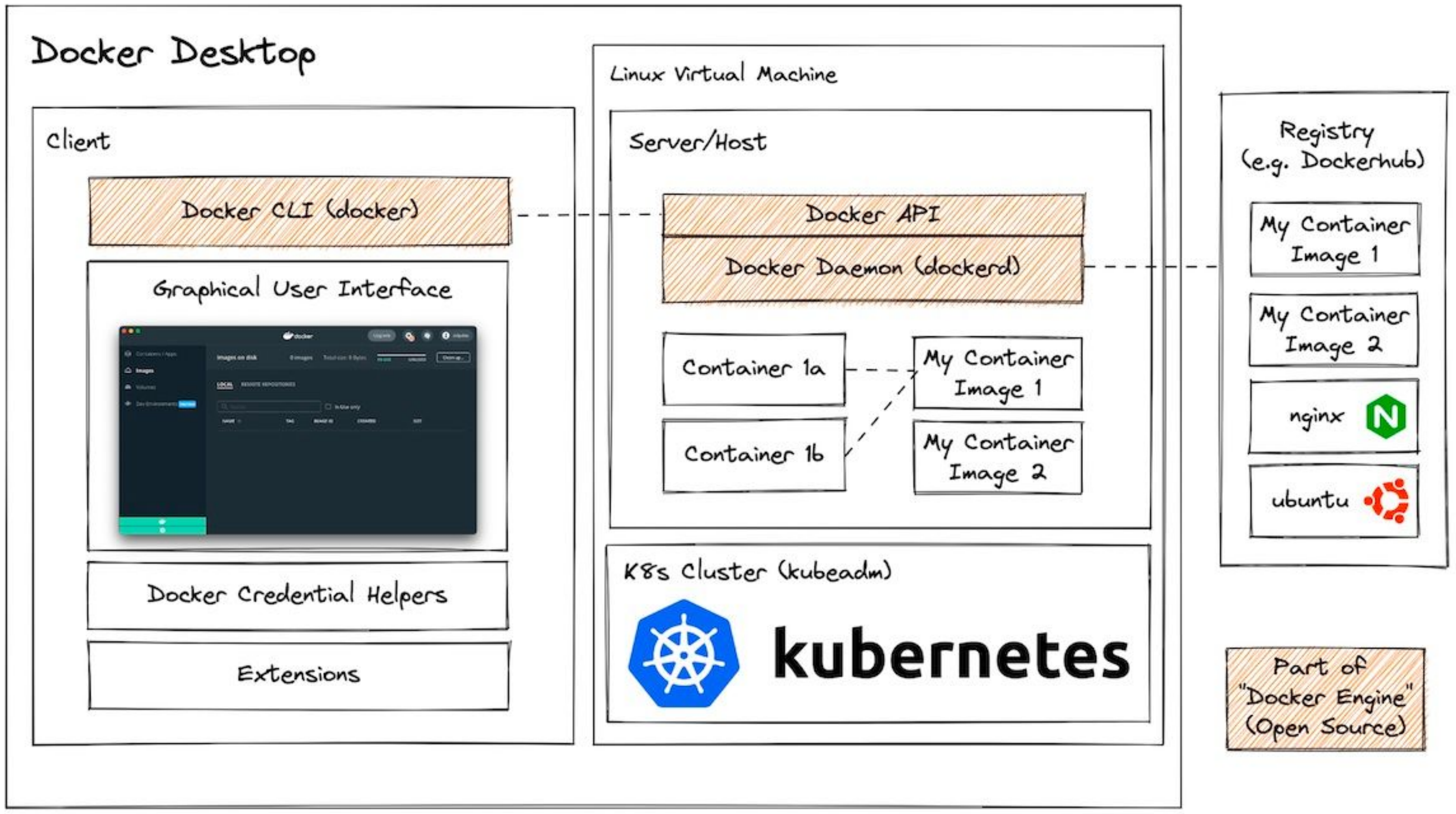
Docker Desktop is free for personal use, but requires a subscription for certain commercial use cases.

Docker Engine refers to a subset of those component which are free and open source and can be installed only on Linux. Specifically Docker Engine includes:

- Docker CLI
- Docker daemon (dockerd), exposing the Docker API

Docker Engine can build container images, run containers from them, and generally do most things that Docker Desktop but is Linux only and doesn't provide all of the developer experience polish that Docker Desktop provides.

Container image registries are not part of Docker itself, but because they are the primary mechanism for storing and sharing container images it is worth including it here. Docker runs a registry named DockerHub, but there are many other registries as well. More info on these can be found in `07-container-registries`.


Docker Desktop
Client
Docker CLI (docker)
Graphical User Interface
Docker Credential Helpers
Extensions
Linux Virtual Machine
Server/Host
Docker API
Docker Daemon (dockerd)
Container 1a
Container 1b
My Container Image 1
My Container Image 2
K8s Cluster (kubeadm)
kubernetes
Registry (e.g. Dockerhub)
My Container Image 1
My Container Image 2
nginx
ubuntu
Part of "Docker Engine" (Open Source)

Installation and Set Up

Docker Desktop: https://docs.docker.com/get-docker/

Docker Engine: https://get.docker.com/

Running Your First Containers

Hello World:

```
docker run docker/whalesay cowsay "Hey Team! 👋"
```

Run Postgres:

```
docker run \
  --env POSTGRES_PASSWORD=foobarbaz \
  --publish 5432:5432 \
   postgres:15.1-alpine
```

General Process

Dockerfiles generally have steps that are similar to those you would use to get your application running on a server.

1. Start with an Operating System
2. Install the language runtime
3. Install any application dependencies
4. Set up the execution environment
5. Run the application

Note: *We can often jump right to #3 by choosing a base image that has the OS and language runtime preinstalled.*

Writing Good Dockerfiles:

Here are some of the techniques demonstrated in the Dockerfiles within this repo:

1. **Pinning a specific base image:** By specifying an image tag, you can avoid nasty surprises where the base image
2. **Choosing a smaller base image:** There are often a variety of base images we can choose from. Choosing a smaller base image will usually reduce the size of your final image.
3. **Choosing a more secure base image:** Like image size, we should consider the number of vulnerabilities in our base images and the attack surface area. Chaingaurd publishes a number of hardened images (https://www.chainguard.dev/chainguard-images).
4. **Specifying a working directory:** Many languages have a convention for how/where applications should be installed. Adhering to that convention will make it easier for developers to work with the container.
5. **Consider layer cache to improve build times:** By undersanding the layered nature of container filesytems and choosing when to copy particular files we can make better use of the Docker caching system.
6. **Use COPY —link where appropriate:** The `--link` option was added to the `COPY` command in march 2022. It allows you to improve cache behavior in certain situations by copying files into an independent image layer not dependent on its predecessors.
7. **Use a non-root user within the container:** While containers can utilize a user namespace to differentiate between root inside the container and root on the host, this feature won't always be leveraged and by using a non-root user we improve the default safety of the container. When using Docker Desktop, the Virtual Machine it runs provides an isolation boundary between containers and the host, but if running Docker Engine it is useful to use a user namespace to ensure container isolation (more info here: https://docs.docker.com/engine/security/userns-remap/). This page also provides a good description for why to avoid running as root: https://cloud.google.com/architecture/best-practices-for-operating-containers#avoid_running_as_root.
8. **Specify the environment correctly:** Only install production dependencies for a production image, and specify any necessary environment variables to configure the language runtime accordingly.
9. **Avoid assumptions:** Using commands like `EXPOSE <PORT>` make it clear to users how the image is intended to be used and avoids the need for them to make assumptions.
10. **Use multi-stage builds where sensible:** For some situations, multi-stage builds can vastly reduce the size of the final image and improve build times. Learn about and use multi-stage builds where appropriate.

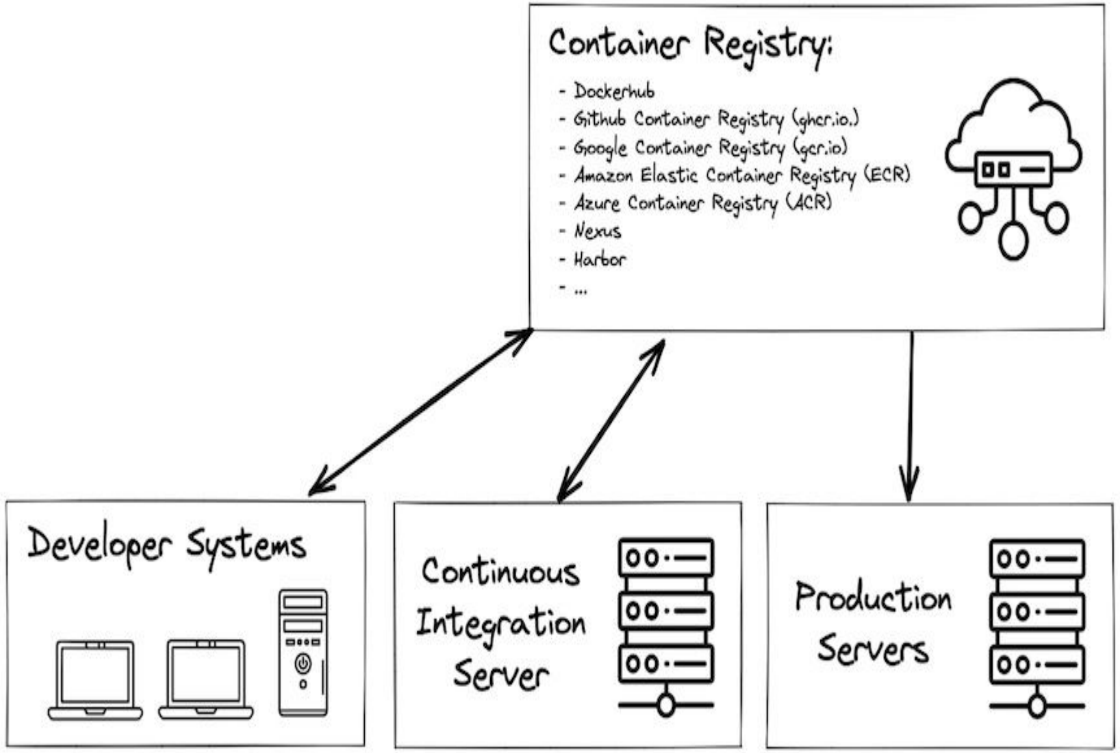
In general, these techniques impact some combination of (1) build speed, (2) image security, and (3) developer clarity. The following summarizes these impacts:

- Pin specific versions [🔒 👁️]
  - Base images (either major+minor OR SHA256 hash) [🔒 👁️]
  - System Dependencies [🔒 👁️]
  - Application Dependencies [🔒 👁️]
- Use small + secure base images [🔒 🏎️]
- Protect the layer cache [🏎️ 👁️]
  - Order commands by frequency of change [🏎️]
  - COPY dependency requirements file → install deps → copy remaining source code [🏎️]
  - Use cache mounts [🏎️]
  - Use COPY --link [🏎️]
  - Combine steps that are always linked (use heredocs to improve tidiness) [🏎️ 👁️]
- Be explicit [🔒 👁️]
  - Set working directory with WORKDIR [👁️]
  - Indicate standard port with EXPOSE [👁️]
  - Set default environment variables with ENV [🔒 👁️]
- Avoid unnecessary files [🔒 🏎️ 👁️]
  - Use .dockerignore [🔒 🏎️ 👁️]
  - COPY specific files [🔒 🏎️ 👁️]
- Use non-root USER [🔒]
- Install only production dependencies [🔒 🏎️ 👁️]
- Avoid leaking sensitive information [🔒]
- Leverage multi-stage builds [🔒 🏎️]

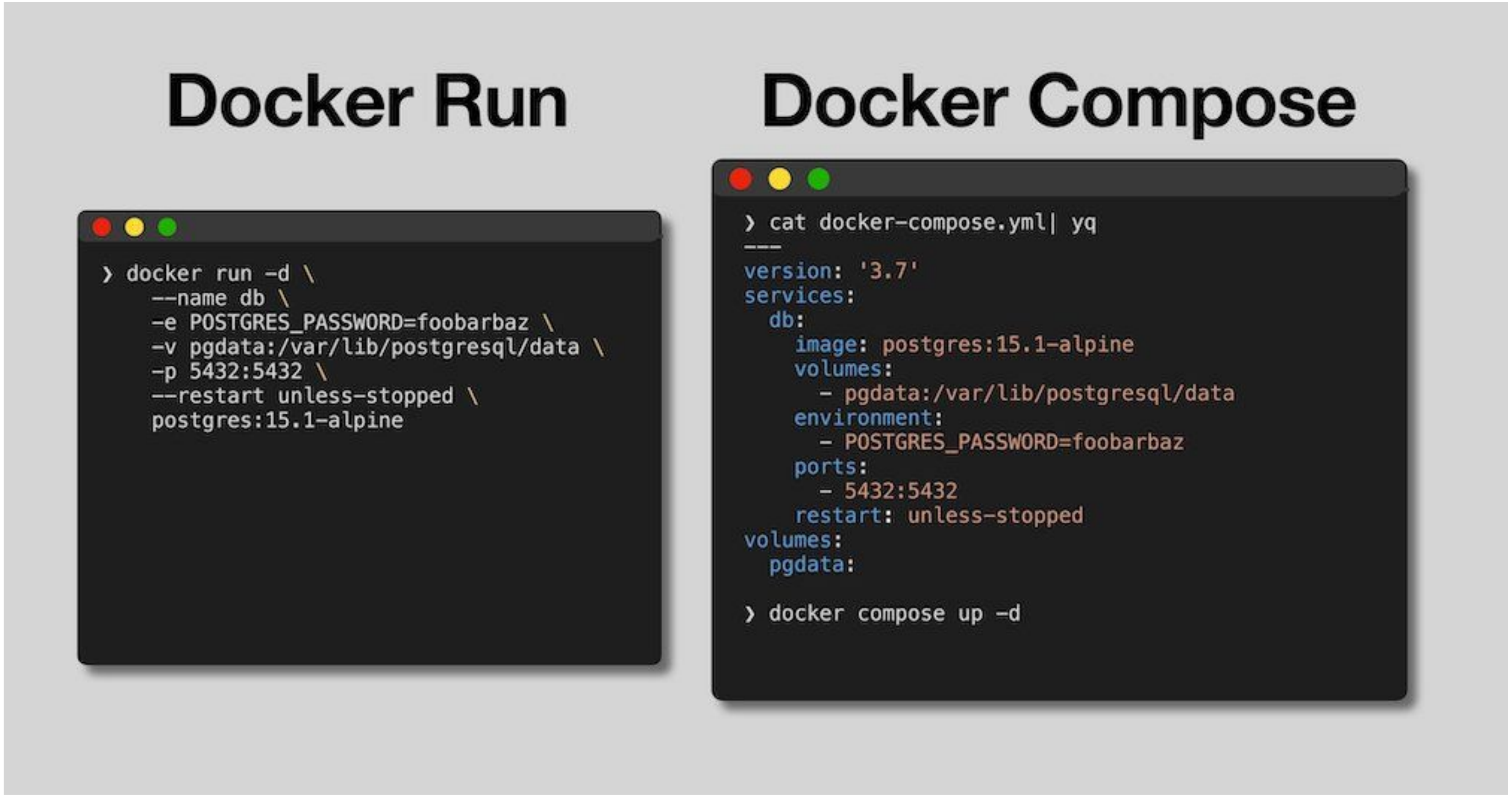
Container Registries

Examples of popular container registries include:

- Dockerhub
- Github Container Registry (ghcr.io.)
- Gitlab Container Registry
- Google Container Registry (gcr.io)
- Amazon Elastic Container Registry (ECR)
- Azure Container Registry (ACR)
- jFrog Container Registry
- Nexus
- Harbor


Container Registry:
- Dockerhub
- Github Container Registry (ghcr.io.)
- Google Container Registry (gcr.io)
- Amazon Elastic Container Registry (ECR)
- Azure Container Registry (ACR)
- Nexus
- Harbor
- ...
Developer Systems
Continuous Integration Server
Production Servers

Running Containers (with Docker)



```
❯ docker run -d \
    --name db \
    -e POSTGRES_PASSWORD=foobarbaz \
    -v pgdata:/var/lib/postgresql/data \
    -p 5432:5432 \
    --restart unless-stopped \
    postgres:15.1-alpine
```

```
❯ cat docker-compose.yml| yq
---
version: '3.7'
services:
  db:
    image: postgres:15.1-alpine
    volumes:
      - pgdata:/var/lib/postgresql/data
    environment:
      - POSTGRES_PASSWORD=foobarbaz
    ports:
      - 5432:5432
    restart: unless-stopped
volumes:
  pgdata:

❯ docker compose up -d
```

Important Configuration Options

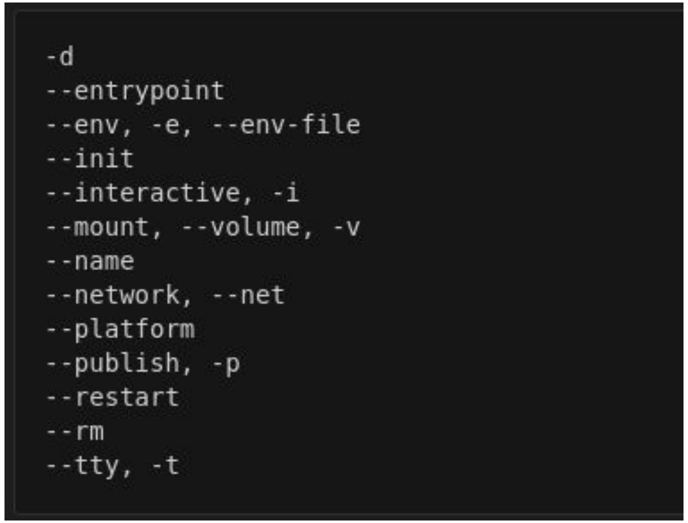
The example shows many configuration options, but does not cover them all.

Documentation: https://docs.docker.com/engine/reference/run/.

All of the command line flags/options can also be specified them within a compose file: https://docs.docker.com/compose/compose-file/

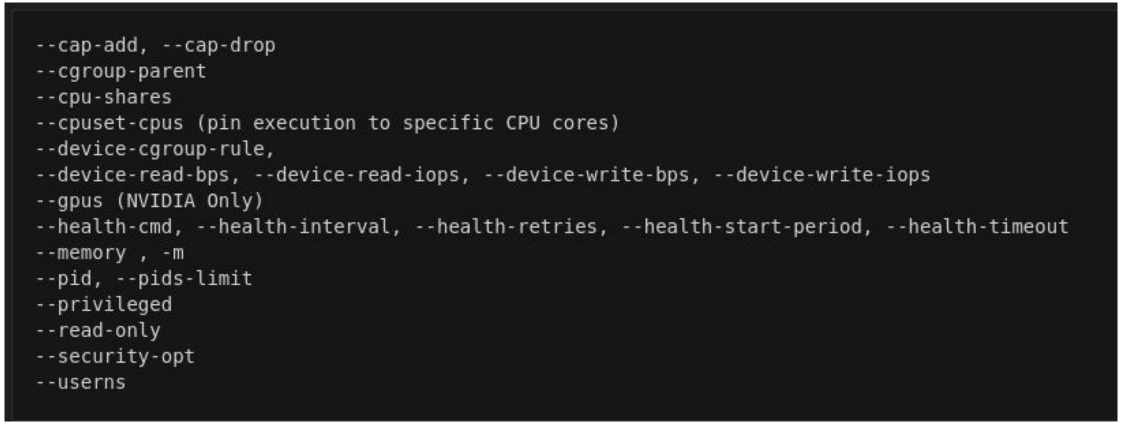
Here are a set of options everyone should know:

Frequently used



```
-d
--entrypoint
--env, -e, --env-file
--init
--interactive, -i
--mount, --volume, -v
--name
--network, --net
--platform
--publish, -p
--restart
--rm
--tty, -t
```

Less frequently used



```
--cap-add, --cap-drop
--cgroup-parent
--cpu-shares
--cpuset-cpus (pin execution to specific CPU cores)
--device-cgroup-rule,
--device-read-bps, --device-read-iops, --device-write-bps, --device-write-iops
--gpus (NVIDIA Only)
--health-cmd, --health-interval, --health-retries, --health-start-period, --health-timeout
--memory , -m
--pid, --pids-limit
--privileged
--read-only
--security-opt
--userns
```

Container Security

Image Security

“What vulnerabilities exist in your image that an attacker could exploit?”

- Keep attack surface area as small as possible:
  - Use minimal base images (multi-stage builds are a key enabler)
  - Don’t install things you don’t need (don’t install dev deps)
- Scan images!
- Use users with minimal permissions
- Keep sensitive info out of images
- Sign and verify images
- Use fixed image tags, either:
  - Pin major.minor (allows patch fixes to be integrated)
  - Pin specific image hash

Runtime Security

If an attacker successfully compromises a container, what can they do? How difficult will it be to move laterally?

Individual containers:

- Use read only filesystem if writes are not needed
- --cap-drop=all, then --cap-add anything you need
- Limit cpu and memory --cpus=“0.5” --memory 1024m
- Use --security-opt
  - seccomp profiles (https://docs.docker.com/engine/security/seccomp/)
  - apparmor profiles (https://docs.docker.com/engine/security/apparmor/)

Interacting with Containers and Other Docker Objects

Familiarize yourself with the docker command line!

You should:

1. Use the documentation here: https://docs.docker.com/engine/reference/commandline/cli/
2. Use the `--help` flag (e.g. `docker build --help`) to get more info about each command.
3.

```
docker image COMMAND:

  build       Build an image from a Dockerfile (`docker build` is the same as `docker image build`)
  history     Show the history of an image
  import      Import the contents from a tarball to create a filesystem image
  inspect     Display detailed information on one or more images
  load        Load an image from a tar archive or STDIN
  ls          List images
  prune       Remove unused images
  pull        Pull an image or a repository from a registry
  push        Push an image or a repository to a registry
  rm          Remove one or more images
  save        Save one or more images to a tar archive (streamed to STDOUT by default)
  tag         Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
```

Scanning Images

Not a `docker image` subcommand, but still something you do with images:

```
docker scan IMAGE
```

Signing Images

Another protection against software supply chain attacks is the ability to uniquely sign specific image tags to ensure an image was created by the entity who signed it.

```
docker trust sign IMAGE:TAG
docker trust inspect --pretty IMAGE:TAG
```

Volumes

`docker volume COMMAND`:

```
create      Create a volume
 inspect     Display detailed information on one or more volumes
 ls          List volumes
 prune       Remove all unused local volumes
   rm          Remove one or more volumes
```

Containers

```
attach      Attach local standard input, output, and error streams to a running container

commit      Create a new image from a container's changes

cp          Copy files/folders between a container and the local filesystem

create      Create a new container

diff        Inspect changes to files or directories on a container's filesystem

exec        Run a command in a running container

export      Export a container's filesystem as a tar archive

inspect     Display detailed information on one or more containers

kill        Kill one or more running containers

logs        Fetch the logs of a container

ls          List containers

pause       Pause all processes within one or more containers

port        List port mappings or a specific mapping for the container

prune       Remove all stopped containers
```

Containers

```
    rename      Rename a container
    restart     Restart one or more containers
    rm          Remove one or more containers
    run         Run a command in a new container
    start       Start one or more stopped containers
    stats       Display a live stream of container(s) resource usage statistics
    stop        Stop one or more running containers
    top         Display the running processes of a container
    unpause     Unpause all processes within one or more containers
    update      Update configuration of one or more containers
    wait        Block until one or more containers stop, then print their exit codes
```

Networks

```
docker network COMMAND:

  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks
```