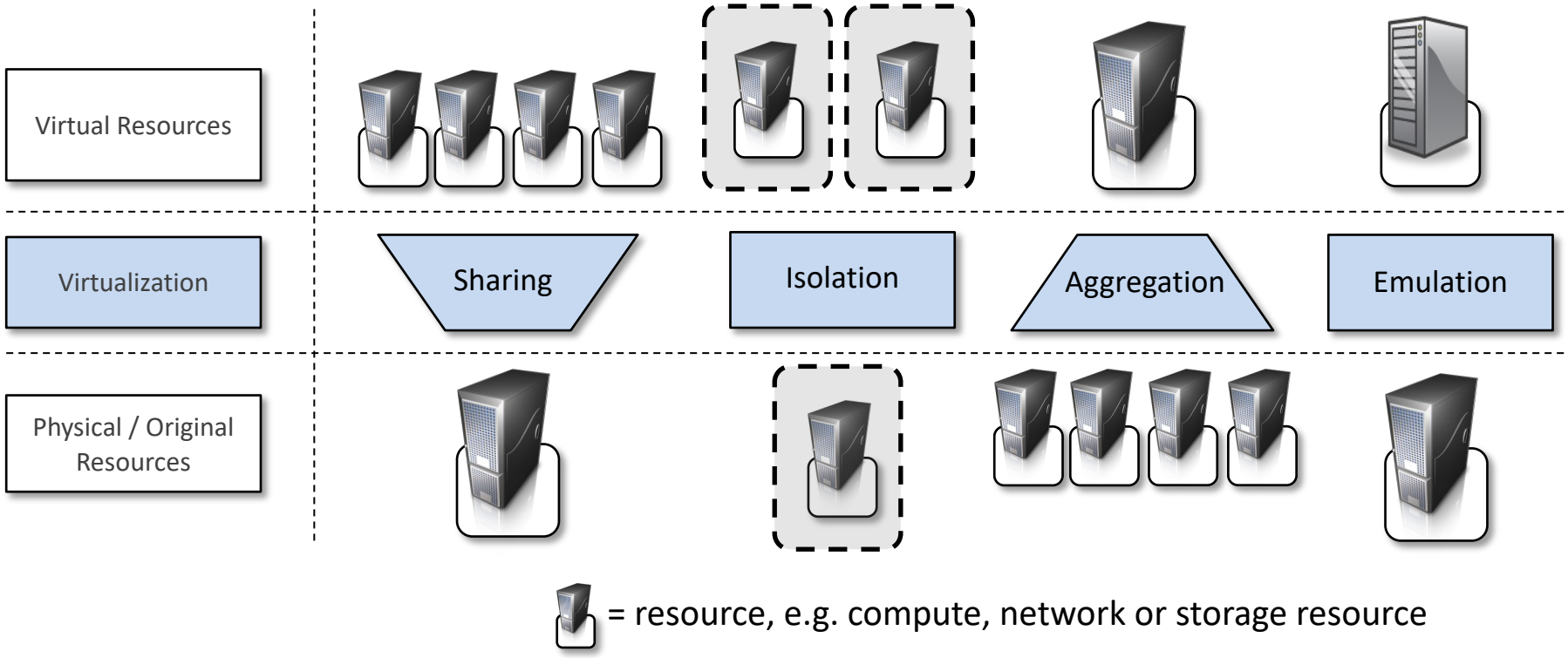
Container Virtualization with Docker

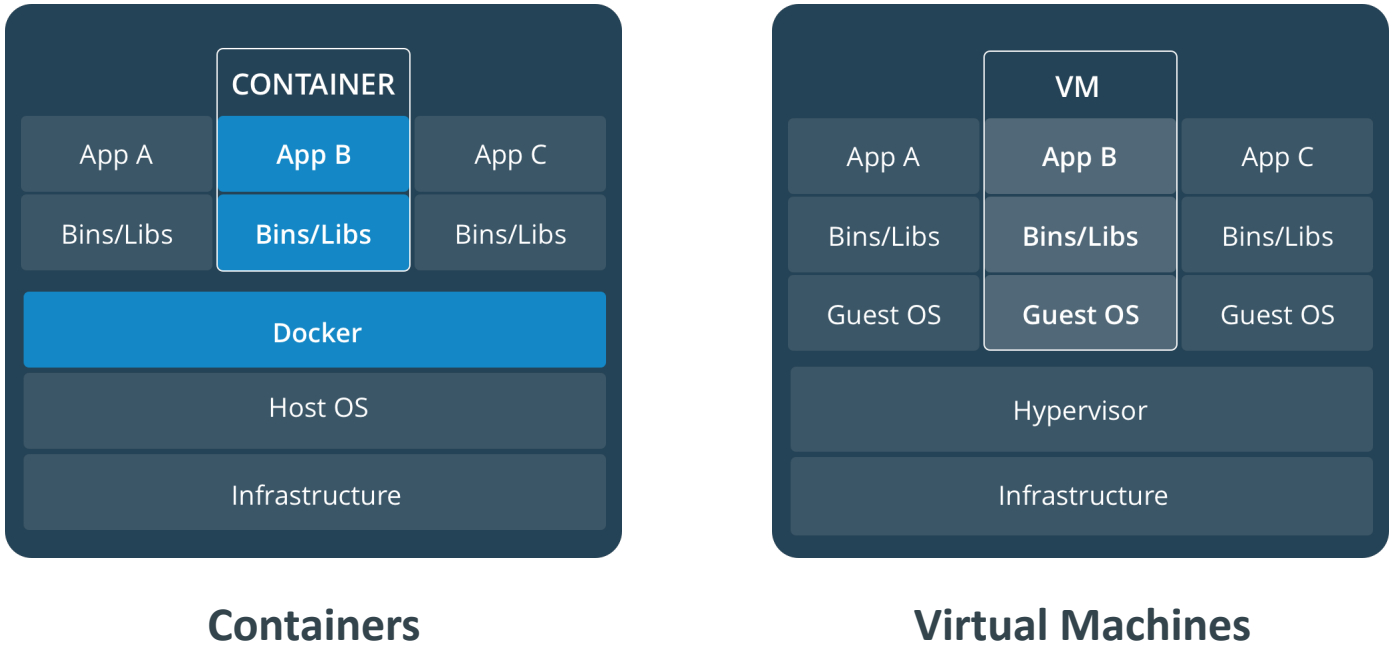
Virtualization



- **Container virtualization** mainly addresses sharing and isolation of compute resources (execution environments).

Figure from: Rajkumar Buyya, Christian Vecchiola and S. Thamarai Selvi: Mastering Cloud Computing, 2013. (modified)

Container Virtualization vs. Virtual Machines



Containers | **Virtual Machines**

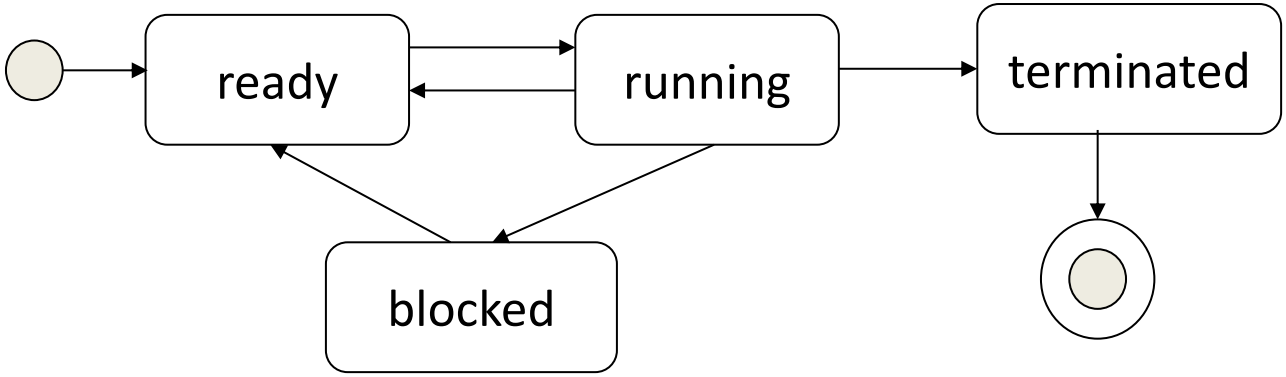
- **Containers** provide a lightweight execution environment that one can quickly spin up to run one (a few) applications.
- **Virtual machines** host an entire operating system to run an ecosystem of applications incompatible with the underlying environment.

Recap: Processes in Operating Systems

- A process represents a program which is executed under control of the operating system.
- Processes can create other processes leading to a process hierarchy.
  - Parent-child relationship of processes
- The operating system allocates resources (CPU time, memory, files, etc.) for processes.
- Common operating system features like multi-processing and virtual memory provide basic virtualization capabilities (sharing, isolation) on the process level.

Recap: Processes in Operating Systems

- A process encompasses one or more threads of control (“threads”).
- The following state-diagram depicts the basic lifecycle of a thread .



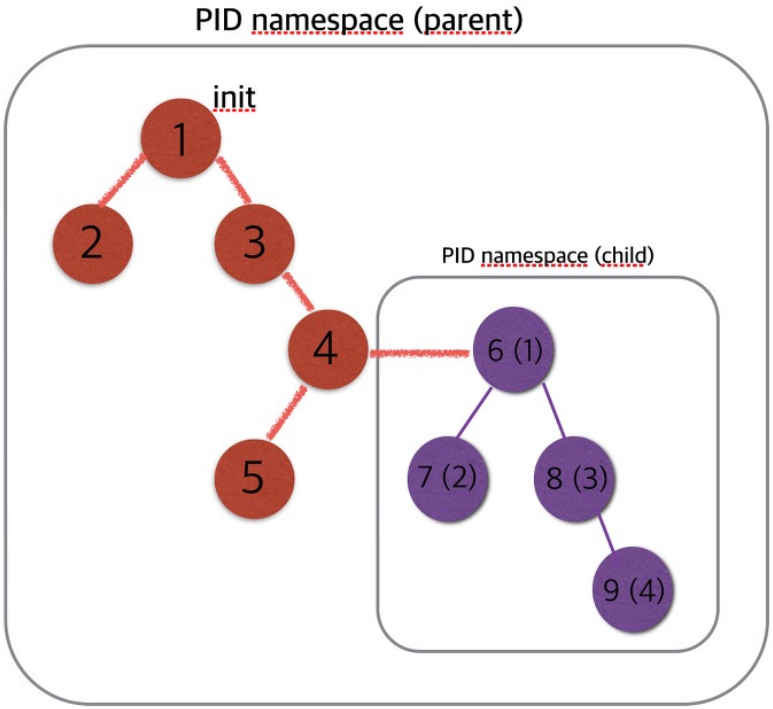
Containers

- **Containers** are well established concepts in modern operating systems to **partition** and **isolate resources** between processes.
  - Examples of container implementations
    - BSD Unix Jails
    - Solaris Zones
    - Linux Containers (LXC)
- In **Linux**, container implementations basically make use of two kernel features:
  - **kernel namespaces:** Isolation between processes
  - **control groups:** Resource restrictions for processes

Linux Kernel Namespaces

- A namespace encapsulates global resources such that it appears to processes within that namespace that they have their own isolated instances of the resources.
- Different classes of namespaces for different types of resources:
  - **PID-NS:** Isolates process IDs
  - **User-NS:** Isolates User and group IDs
  - **Mount-NS:** Isolates mountpoints (file system subtrees)
  - **UTS-NS:** Isolates hostname and NIS domain name (system ident.)
  - **Network-NS:** Isolates Network devices, ports, etc. (network stack)
  - **IPC-NS:** Isolates System V IPC, POSIX message queues
  - **cgroup-NS:** Isolates cgroup root directory

Example: PID Namespace



- The process that creates a PID namespace remains in the parent namespace, but makes its child the root of a new process tree in a separate PID namespace.
- Processes within the new namespace have 2 PIDs: One for the new namespace and one for the global namespace.
- Processes within the new namespace can not “see” the processes in the parent namespace but the parent process namespace can “see” the processes in the child namespace.

Linux Control Groups (cgroups)

- Control groups (cgroups) is a Linux kernel feature which allows to measure and limit the resource usage of a group of processes.
  - Resources quotas for CPU time, memory, network, and disk IO can be set.
  - Resource accounting may be used for billing purposes
- The processes of a cgroup are all bound to the same set of limits.
- Cgroups can be organized hierarchically, i.e. a cgroup inherits limits from its parent cgroup.

Docker

- Docker is based on the Linux technologies discussed above.
  - A native Microsoft Windows version of Docker is also available, but not widely adopted.
- Docker offers an entire ecosystem for container management.
- Docker Containers are a highly standardized environment for shipping and executing (server-)software.
- Docker enables new software engineering approaches:
  - DevOps: Combines software development (Dev) and IT operations (Ops) to shorten the systems development life cycle
  - Continuous Delivery (CI): Aims at building, testing, and releasing software with greater speed and frequency.

Open Container Initiative

- Open Container Initiative, OCI (www.opencontainers.org)
  - Linux Foundation Project
  - Promotes a set of common, minimal, open standards and specifications around container technology.
    - Runtime Specification
    - Image Specification
    - Distribution Specification
- OCI compliant Docker alternatives:
  - Podman (Red Hat): https://podman.io
    - Container management system
  - Buildah: https://buildah.io
    - Container image builder
  - containerd, runC
    - Container runtimes

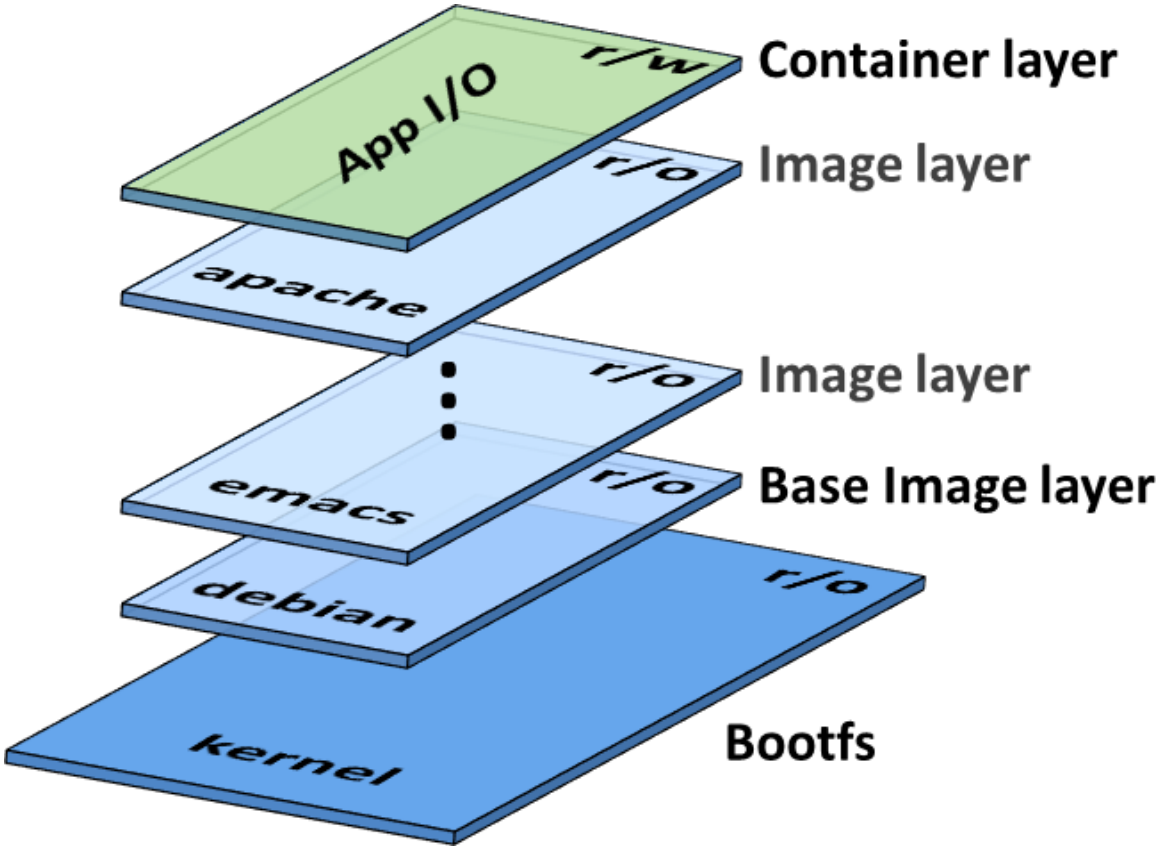
Docker Image

- A Docker image is a read-only template used to build containers.
  - Every Docker container is derived from an image.
  - Images are mainly used to provide a (root) filesystem for Docker containers.
  - Images also contain metadata that describe the container's capabilities and needs.
- The name of an image has three parts: source/imagename:tag
  - source: organization or person who constructed the image
  - imagename: name of the image
  - Tag: used to distinguish between different versions of an image
    - The tag *latest* refers to the latest version

Docker Image

- Images are employed to store and ship applications.
  - All specific dependencies of an application (binaries, libraries) are bundled in an image.
  - Thus, containers can be easily moved between development, test, and production environments.

Anatomy of a Docker Image



Anatomy of a Docker Image

- Images consists of filesystem layers:
  - The lowest layer is the bootfs filesystem of the underlaying Docker host, containing e.g. the kernel.
  - The next layer is called base image layer which is typically a (minimal) Linux filesystem.
  - On top of the base layer application specific layers are stacked.
  - The topmost layer is the container layer which is the only one with write access.
- Each container has its own writable container layer.
  - Changes are stored in this container layer (copy on write).
  - Multiple containers can share the same underlying image
    - Minimizes size of each container.

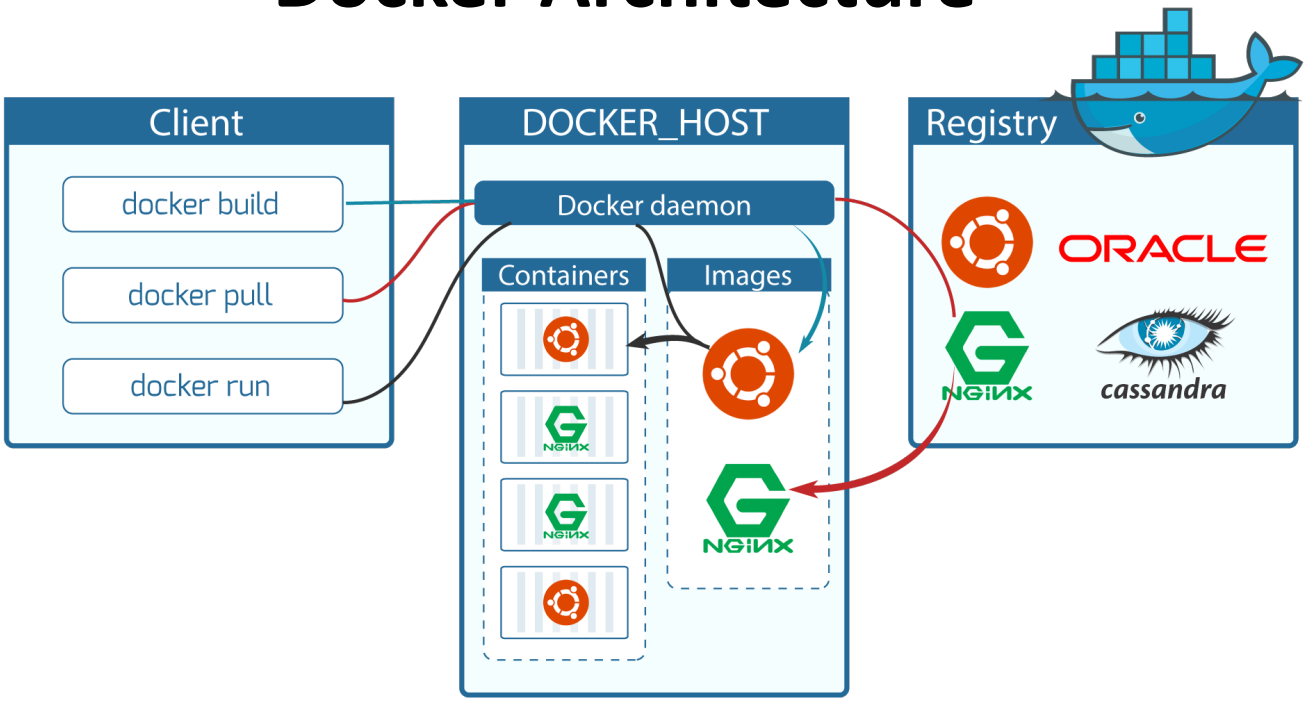
Anatomy of a Docker Image

- Technically, filesystem implementations with overlay features are used, e.g. overlay(2)fs, aufs, btrfs, zfs

Docker Container

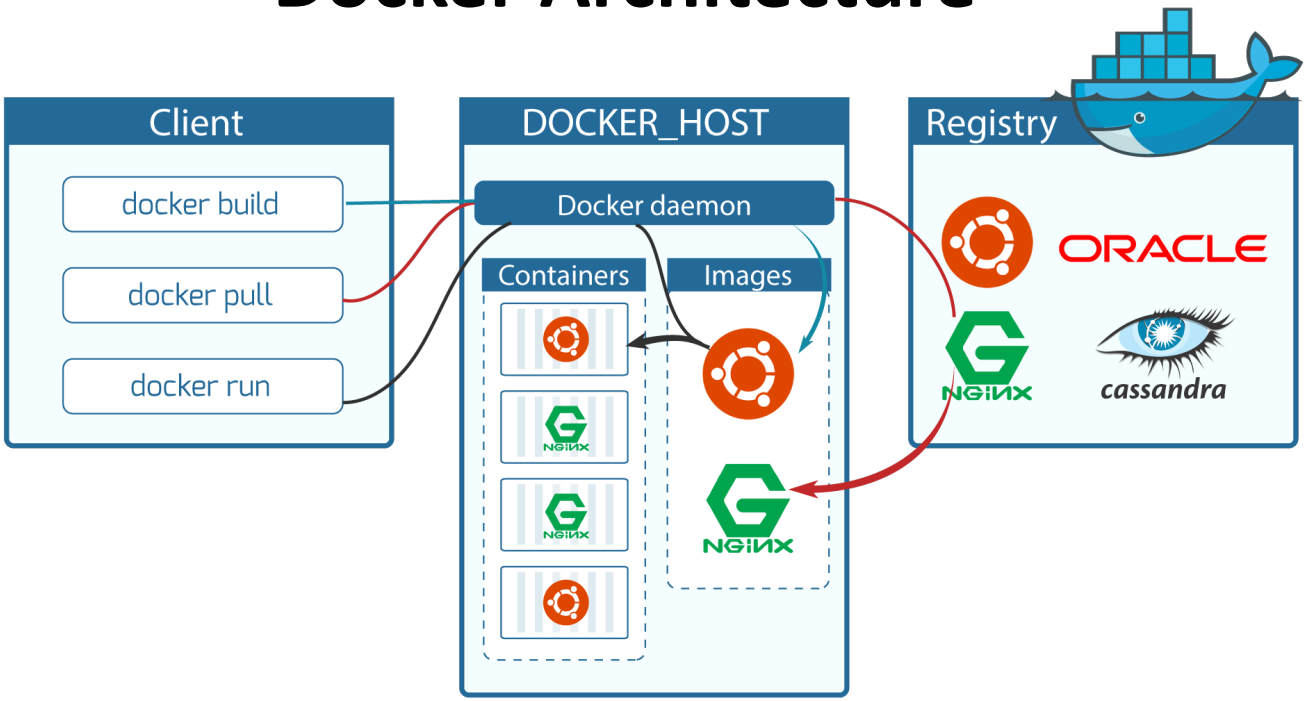
- A Docker container is basically defined by an image and additional configuration options, e.g. network connections and storage options.
- Containers only have access to resources that are defined in the image, unless additional access is defined when building the image into a container.
- You can also create a new image based on the current state of a container.

Docker Architecture



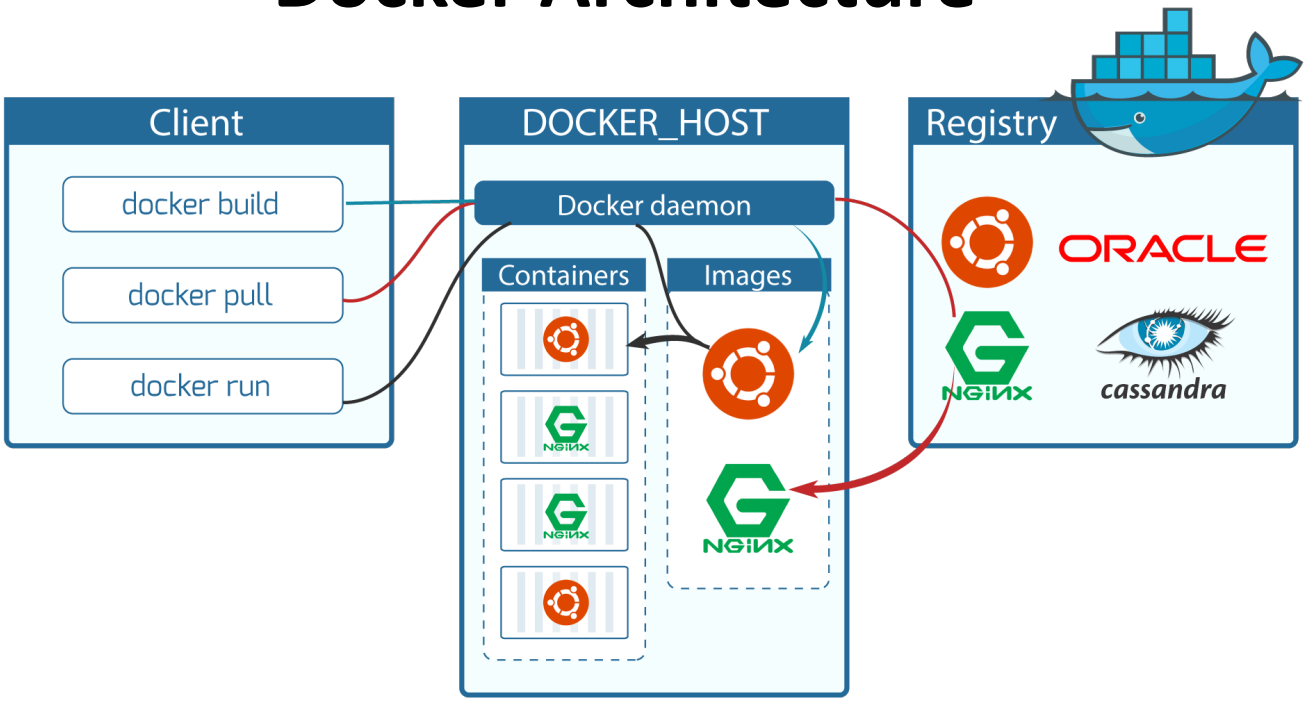
- **Docker clients:** Enable users to interact with Docker via a command line interface or graphical user interface.
  - Commands for building/downloading images and for managing containers.
- A Docker client can reside on the same host as the daemon or connects to a daemon on a remote host.

Docker Architecture



- **Docker Daemon:** A persistent background process that manages Docker images, containers, networks, and storage volumes on a Docker Host.
  - Exposes a REST API to applications / clients

Docker Architecture



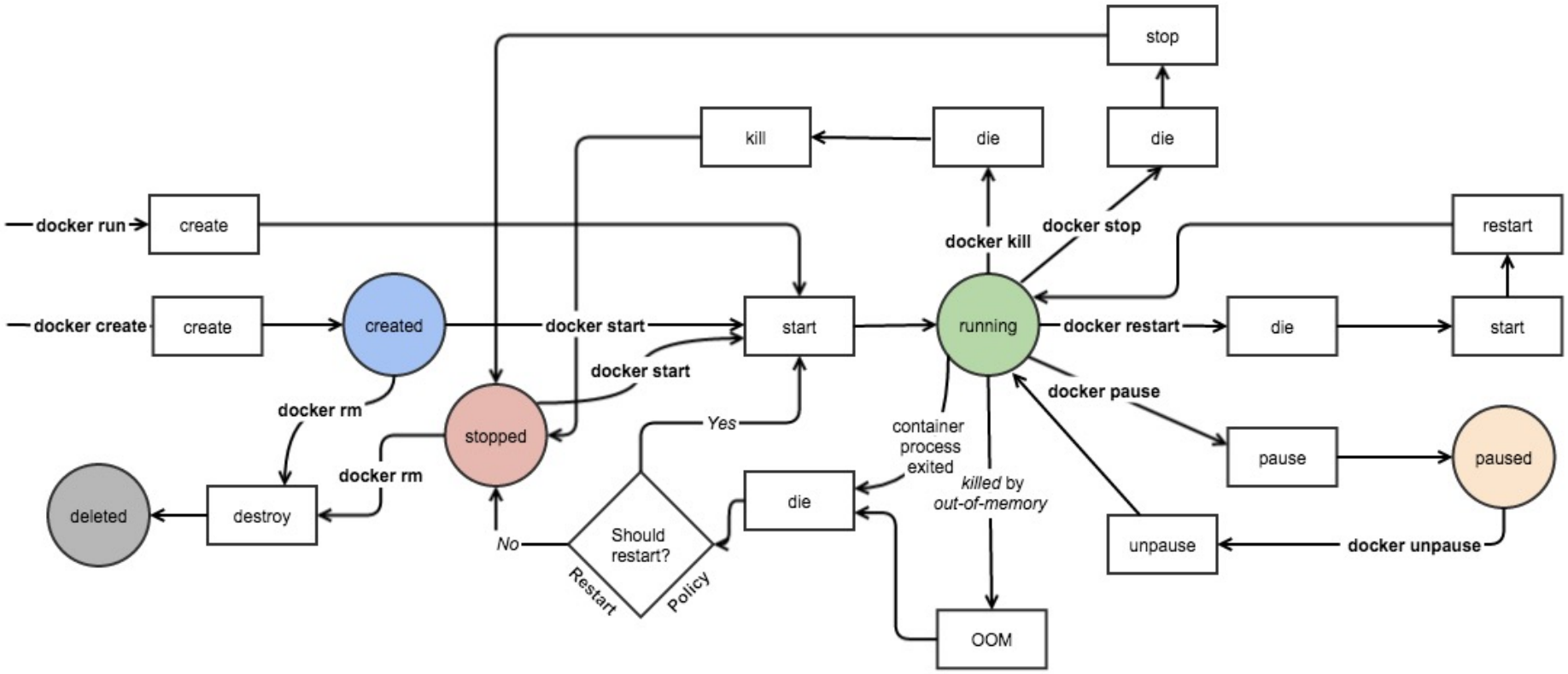
- **Registry:** Maintains a repository of Docker images that can be uploaded and downloaded by docker hosts.
  - Public Docker Hub or private registries can be used.
  - Registry enables sharing of docker images.

Basic Docker Commands

docker image pull	Pull an image from a registry
docker image ls	List local images
docker image rm	Remove one or more images
docker image build	Build an image from a Dockerfile
docker run	Run a command in a new container
docker stop	Stop one or more running containers
docker start	Start one or more stopped containers
docker top	Display the running processes of a container
docker rm	Remove one or more containers
docker kill	Kill one or more running containers
docker volume	Manage Docker volumes
docker network	Manage Docker networks
docker search	Search the Docker Hub for images

For more commands and a detailed description see:
https://docs.docker.com/engine/reference/commandline/docker/

Lifecycle of a Docker Container



Creating a Docker Image

- A Dockerfile is used to specify the content of a new image
  - Typically, the new image adds functionality to an existing image.
- Example Dockerfile:

```
FROM ubuntu:18.04
RUN apt-get update && \
    apt-get install -y nano && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
CMD ["/bin/bash"]
```

- Build image: `docker image build -t ubuntu-nano-cbdt00 .`
- For more information on the syntax of Dockerfiles see: https://docs.docker.com/engine/reference/builder/