Containers

Different types of virtualization

Computing resources can be virtualized.

- **Virtual Machine (VM)**: is an emulation of a computer system. A piece of software “pretends” to be hardware
- OS-level virtualization (a.k.a. **Container**): is an isolated user-space instance within an OS
- **Sandbox**: is a security mechanism for separating running program

Definition

> **OS-level virtualization** is an operating system paradigm in which the **kernel** allows the existence of multiple isolated **user space** instances. Such instances, called sometimes ***containers***, may look like real computers from the point of view of programs running in them.

A computer program running on an ordinary operating system can see all resources (connected devices, files and folders, network shares, CPU power, quantifiable hardware capabilities) of that computer. However, programs running inside of a container can only see the container's contents and devices assigned to the container.

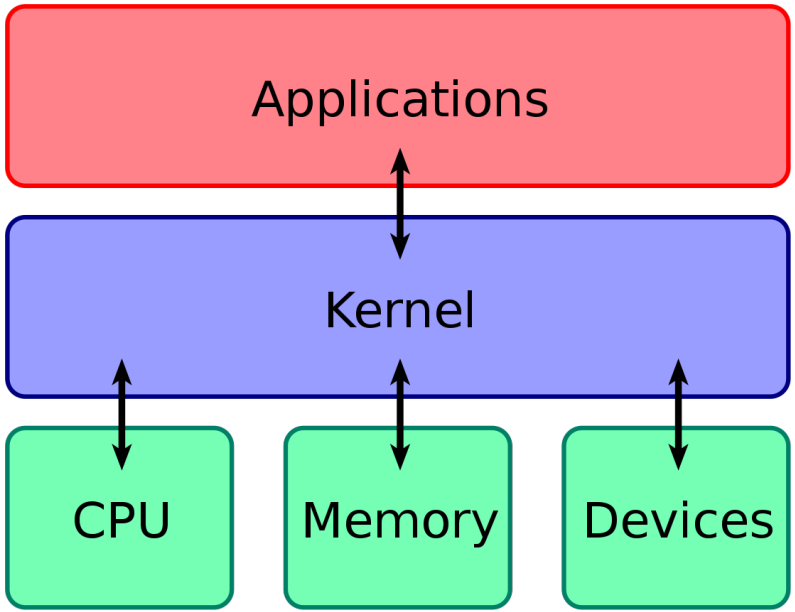
source: https://en.wikipedia.org/wiki/OS-level_virtualization

Containers or…

Such instances are called:

- **containers** (LXC, Solaris containers, Docker),
- **Zones** (Solaris containers),
- **virtual private servers** (OpenVZ),
- **partitions**,
- **virtual environments** (VEs),
- **virtual kernels** (DragonFly BSD),
- **jails** (FreeBSD jail or chroot jail)

What is the *kernel*?



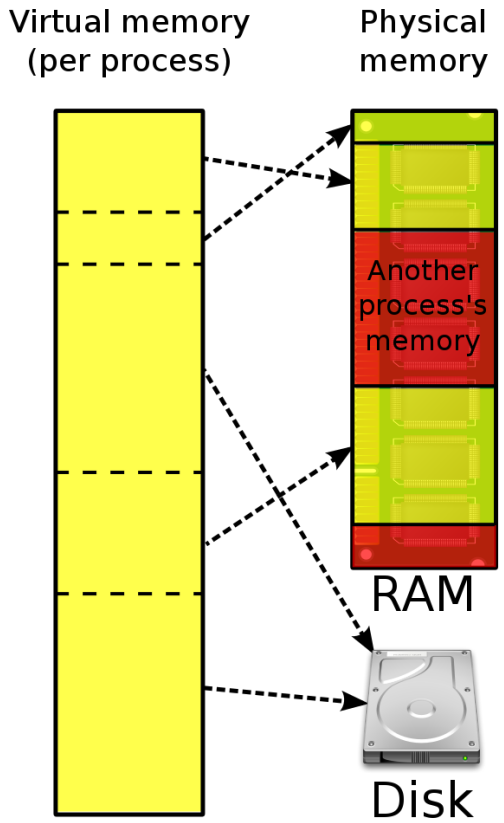
The **kernel** is a computer program at the core of a computer's operating system with complete control over everything in the system. It is an integral part of any operating system. It is the portion of the operating system code that is always resident in memory. It facilitates interactions between hardware and software components. On most systems, it is one of the first programs loaded on startup (after the bootloader). It handles the rest of startup as well as input/output (I/O) requests from software, translating them into data-processing instructions for the central processing unit. It handles memory and peripherals like keyboards, monitors, printers, and speakers.

source: https://en.wikipedia.org/wiki/Kernel_(operating_system)

What is the *user space*?

A modern computer operating system usually segregates virtual memory into **kernel space** and **user space**. Primarily, this separation serves to provide memory protection and hardware protection from malicious or errant software behavior.

Kernel space is strictly reserved for running a privileged operating system kernel, kernel extensions, and most device drivers. In contrast, **user space is the memory area where application software and some drivers execute**.



source: https://en.wikipedia.org/wiki/User_space

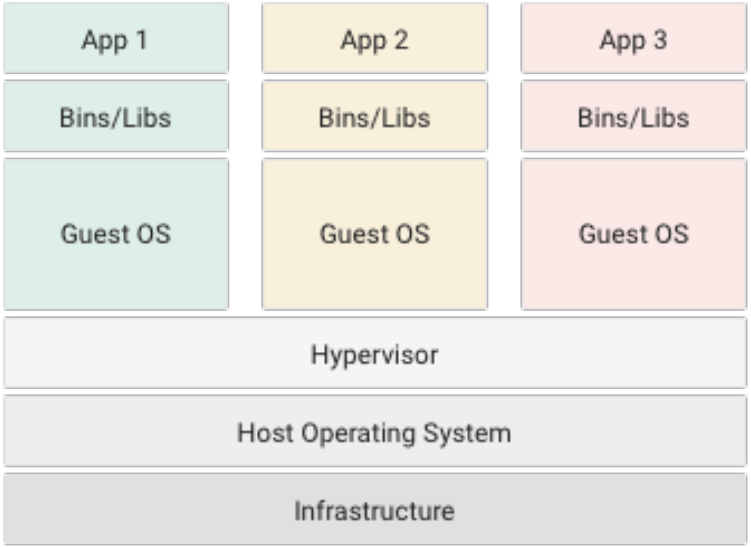
Containers

Containers offer a **logical packaging** mechanism in which **applications can be abstracted from the environment** in which they actually run.

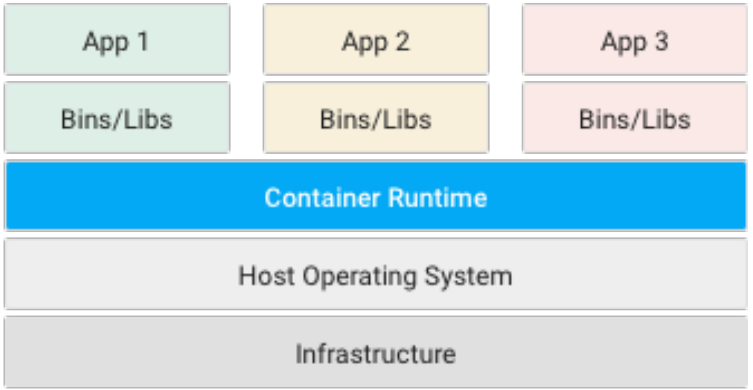
This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's personal laptop.

Containerization provides a clean separation of concerns, as **developers** focus on their application logic and dependencies, while **IT operations** teams can focus on deployment and management without bothering with application details such as specific software versions and configurations specific to the app.

Containers vs. Virtual Machines



Virtual Machines



Containers

Instead of virtualizing the hardware stack as with the virtual machines approach, containers virtualize at the operating system level, with multiple containers running atop the OS kernel directly. This means that containers are far more lightweight: they share the OS kernel, start much faster, and use a fraction of the memory compared to booting an entire OS.

The absence of the guest OS is why containers are so lightweight and, thus, fast and portable.

Ship with **efficiency**

Using **standard** tools

Delivering **custom** goods

Benefits of containers

The primary advantage of containers, especially compared to a VM, is providing a level of abstraction that makes them lightweight and portable.

- **Lightweight:** Containers share the machine OS kernel, eliminating the need for a full OS instance per application and making container files small and easy on resources. Their smaller size, especially compared to virtual machines, means they can spin up quickly and better support cloud-native applications that scale horizontally.
- **Portable and platform independent:** Containers carry all their dependencies with them, meaning that software can be written once and then run without needing to be re-configured across laptops, cloud, and on-premises computing environments.
- **Supports modern development and architecture:** Due to a combination of their deployment portability/consistency across platforms and their small size, containers are an ideal fit for modern development and application patterns—such as DevOps, serverless, and microservices—that are built are regular code deployments in small increments.
- **Improves utilization**: Like VMs before them, containers enable developers and operators to improve CPU and memory utilization of physical machines. Where containers go even further is that because they also enable microservice architectures, application components can be deployed and scaled more granularly, an attractive alternative to having to scale up an entire monolithic application because a single component is struggling with load.

source: https://www.ibm.com/cloud/learn/containers

Benefits of containers

- **Efficiency and security:** Processes share OS resources, but remain segregated
- Quick and easy to create, delete, start, stop, download, and share
- **Can be treated as unchangeable:** every container created from a single image will be the same. Container images are versions.
  You should not patch (update/fix) a container, instead patch the image when it needs to be updated, and then create a new container with the new version.
  When you don't need it anymore, a container can be easily deleted or replaced.

source: https://www.ibm.com/cloud/learn/containers

Containerization

Containerization involves **packaging up software code and all its dependencies** so that it can run uniformly and consistently on any infrastructure.

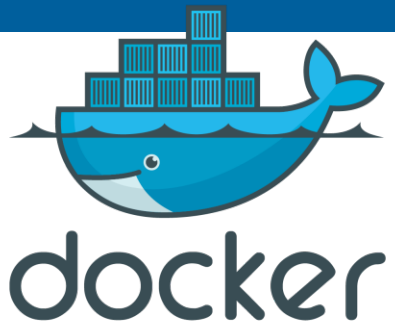
Containerization allows developers to create and deploy applications faster and more securely. With traditional methods, code is developed in a specific computing environment which, when transferred to a new location, often results in bugs and errors. For example, when a developer transfers code from a desktop computer to a VM or from a Linux to a Windows operating system. Containerization eliminates this problem by bundling the application code together with the related configuration files, libraries, and dependencies required for it to run. This single package of software or “container” is abstracted away from the host operating system, and hence, it stands alone and becomes **portable**: able to run across any platform or cloud, free of issues.

Microservices

Microservices (or microservices architecture) are a cloud native architectural approach in which a single application is composed of many loosely coupled and independently deployable smaller components, or services. These services typically

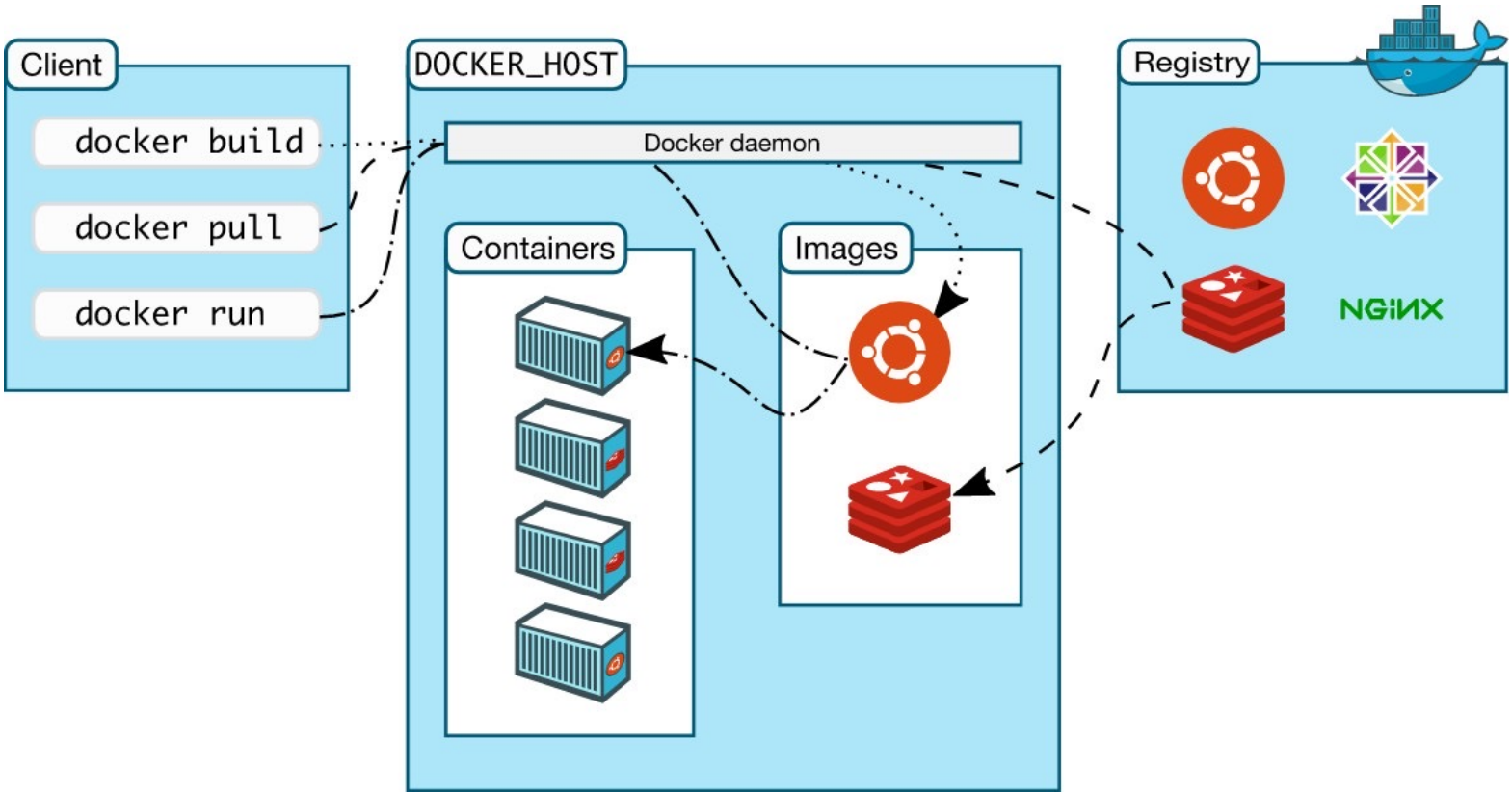
- have their own stack, inclusive of the database and data model;
- communicate with one another over a combination of REST APIs, event streaming, and message brokers; and
- are organized by business capability, with the line separating services often referred to as a bounded context.

Docker



- Docker is the most popular **open source containerization platform**. Docker enables developers to package applications into containers
- While developers can create containers without Docker, Docker makes it easier, simpler, and safer to build, deploy, and manage containers. It's essentially a toolkit that enables developers to build, deploy, run, update, and stop containers using simple commands and work-saving automation.
- Docker also refers to *Docker, Inc.*, the company that sells the commercial version of Docker, and to the *Docker open source project*, to which Docker Inc. and many other organizations and individuals contribute.

Docker architecture



Docker components

- The **Docker daemon** (*dockerd*) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
- The **Docker client** (docker) is the primary way that many Docker users interact with Docker. When you use commands such as <docker run>, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.
- A **Docker registry** stores Docker images. **Docker Hub** is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. When you use the <docker pull> or <docker run> commands, the required images are pulled from your configured registry. When you use the <docker push> command, your image is pushed to your configured registry.
- **Docker objects**
  When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects.

source: https://docs.docker.com/get-started/overview/

Docker objects

- **Images**

An image is a **read-only template with instructions for creating a Docker container**. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a **Dockerfile** with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

- **Containers**

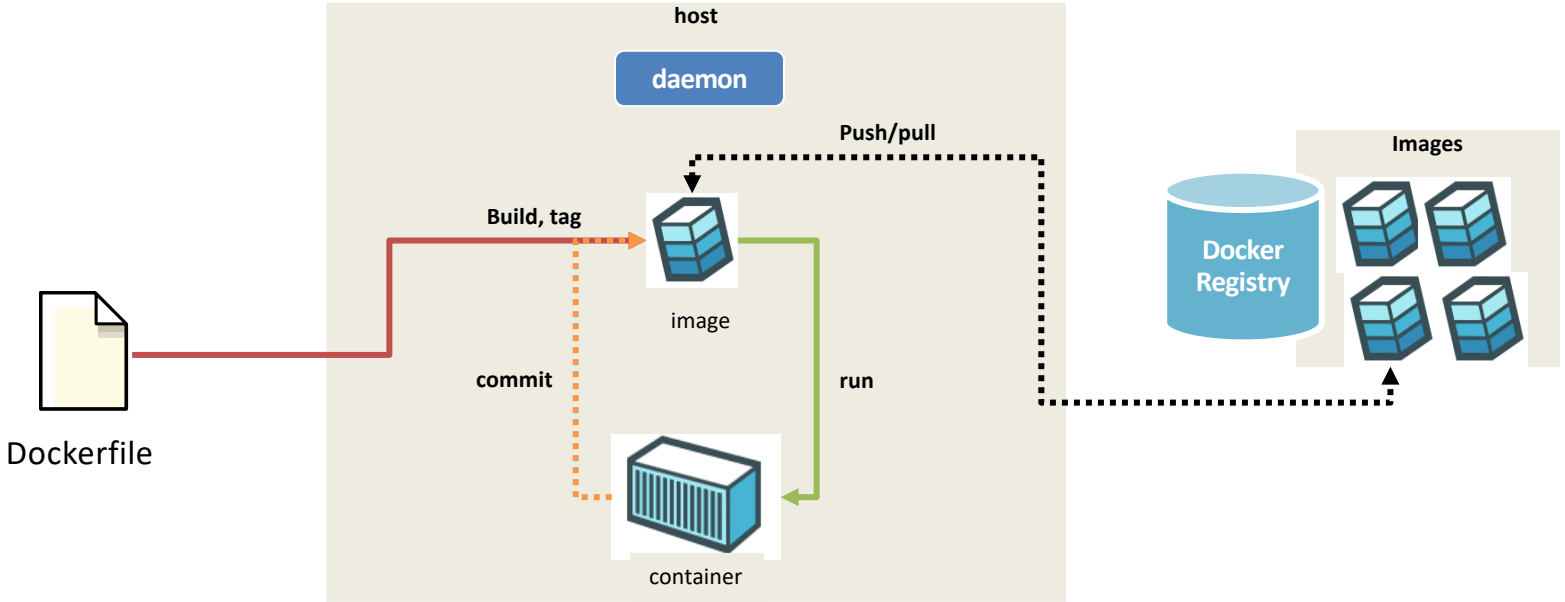
A container is a **runnable instance of an image**. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

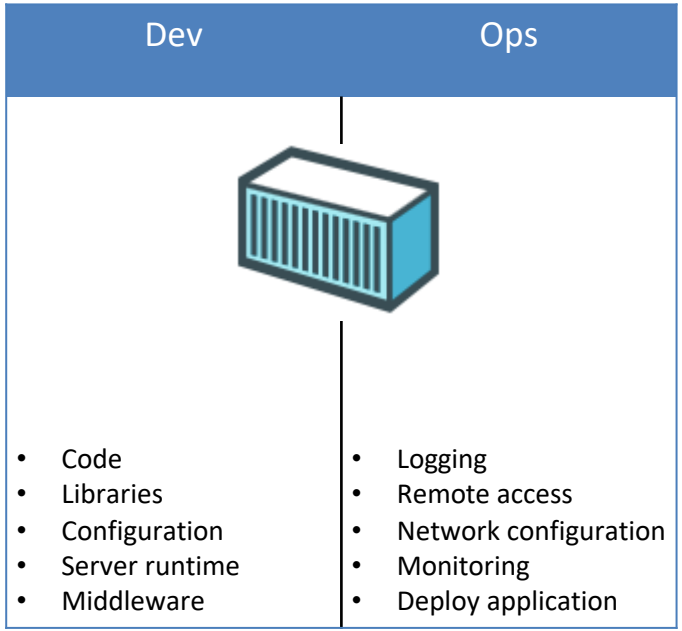
A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

source: https://docs.docker.com/get-started/overview/

A typical workflow has you create images, pull them from the repository, build, and run as containers.
Changes to a container may be committed to create a new image. (See docker commit command.)
The more typical and more controlled and documented approach to making image changes is to change the Dockerfile and build a new image.

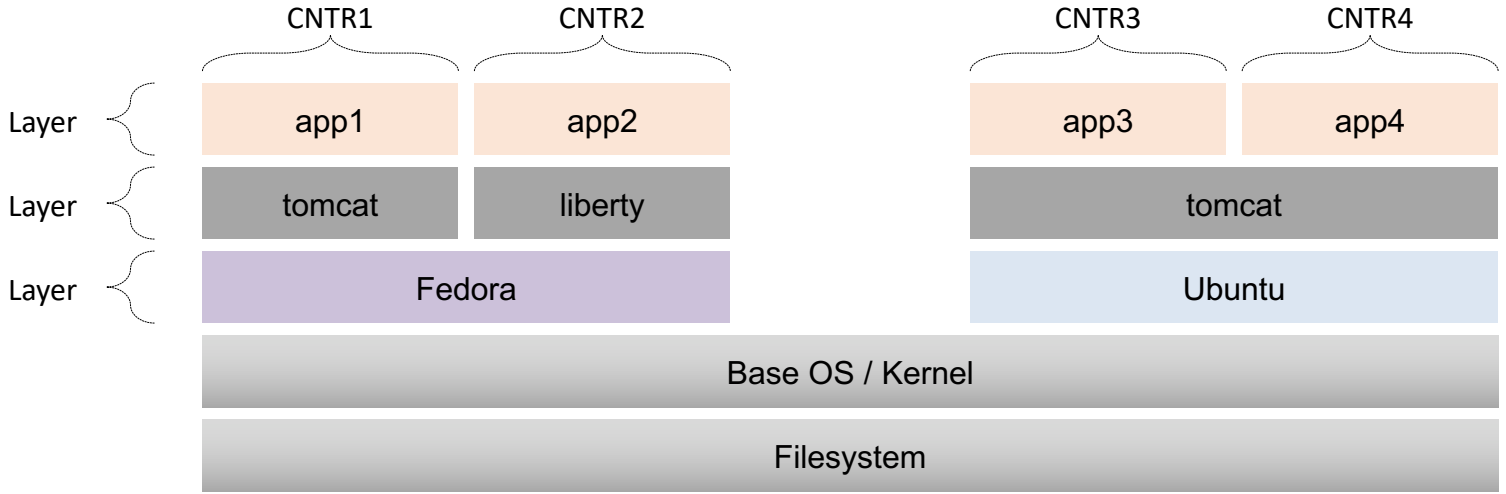


Development vs Operations



- Separation of concerns
- A container separates and bridges the Dev and Ops in DevOps
- Dev focuses on the application environment
- Ops focuses on the deployment environment

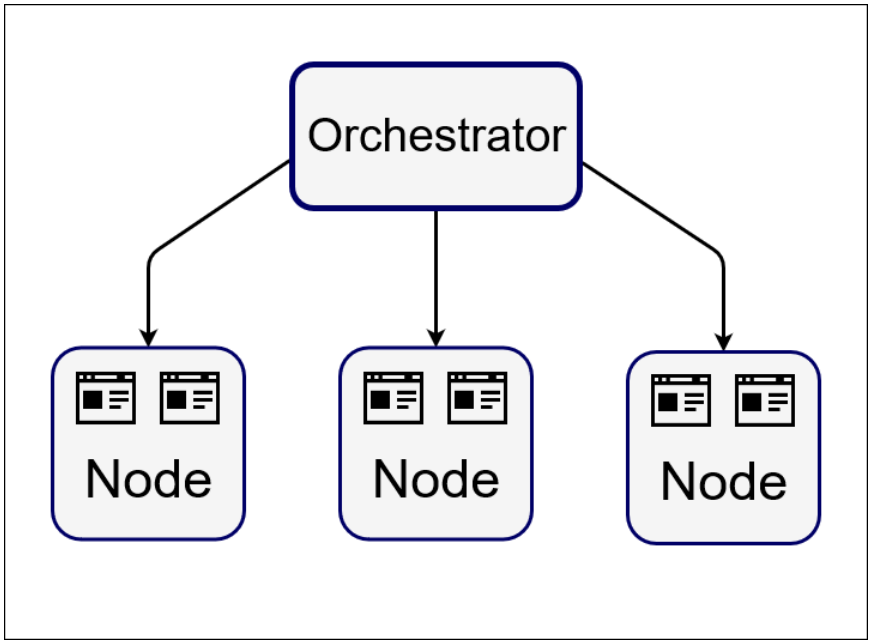
- Docker uses a copy-on-write (union) filesystem
- New files (& edits) are only visible to current/above layers



- Layers allow for reuse
  - More containers per host
  - Faster start-up/download time – base layers are "cached"
- Images
  - Tarball of layers (each layer is a tarball)

Container orchestration

Container orchestration automates the deployment, management, scaling, and networking of containers.



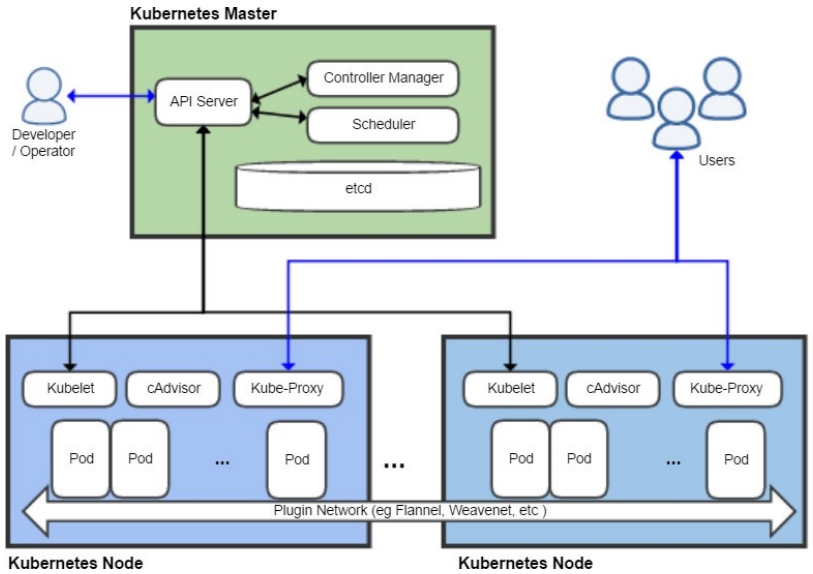
Container orchestration

Container **orchestration** is all about managing the lifecycles of containers, especially in large, dynamic environments. Software teams use container orchestration to control and automate many tasks:

- Provisioning and deployment of containers
- Redundancy and availability of containers
- Scaling up or removing containers to spread application load evenly across host infrastructure
- Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
- Allocation of resources between containers
- External exposure of services running in a container with the outside world
- Load balancing of service discovery between containers
- Health monitoring of containers and hosts
- Configuration of an application in relation to the containers running it

Kubernetes (K8s)

Kubernetes (**K8s**) is the main open-source system for automating deployment, scaling, and management of containerized applications.
It groups containers that make up an application into logical units for easy management and discovery.



Reading list

- D. Bernstein, Containers and Cloud: From LXC to Docker to Kubernetes, *IEEE Cloud Computing* (2014)

(document available on Blackboard)

Resources

- https://cloud.google.com/containers
- https://www.ibm.com/cloud/learn/containers
- https://www.docker.com/resources/what-container
- https://docs.docker.com/
- Containerization Explained [Video by IBM Cloud]
- Container Orchestration Explained [Video by IBM Cloud]

Questions, comments?