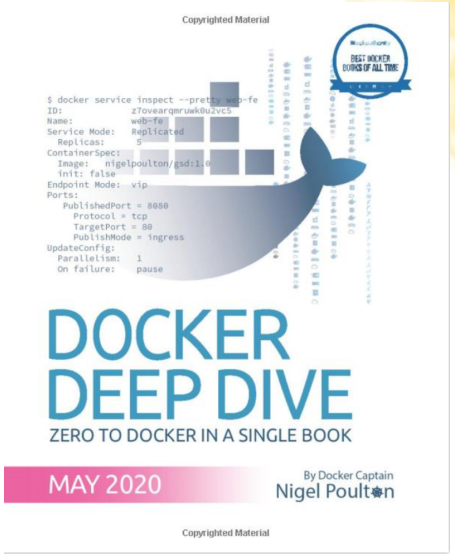
UMD DATA605 - Big Data Systems

Lesson 3.1: DevOps

Instructor: Dr. GP Saggese, gsaggese@umd.edu

Docker - Resources

- We will use Docker during the class project and most tutorials
- Concepts in the slides
- Class tutorials:
  - tutorial_docker
  - tutorial_docker_compose



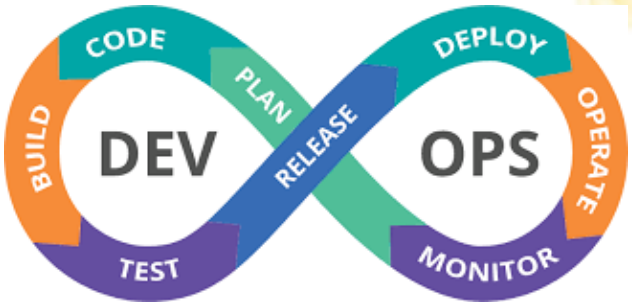
- Web resources:
  - Docker Tutorial for beginners
  - https://labs.play-with-docker.com/
  - https://training.play-with-docker.com
  - A Beginner-Friendly Introduction to Containers, VMs and Docker
  - Official Docker Getting Started Tutorial
- Mastery:
  - Poulton, Docker Deep Dive: Zero to Docker in a single book, 2020

Application Deployment

- For (almost all) Internet companies, **the application is the business**
  - If the application breaks, the business stops
  - E.g., Amazon, Google, Facebook, online banks, travel sites, ...
- **Problem**
  - Release, deploy, manage, monitor applications
- **Solutions**
  - Before 2000s: "bare-metal era"
  - 2000s-2010s: "virtual machine era"
  - After ~2013: "container era"

DevOps

- **DevOps** = practices combining:
  - Software development (*dev*)
  - IT operations (*ops*)
- **Containers revolutionized DevOps**
  - Enable independence between app development and IT ops
    - One team creates an app
    - Another team deploys and manages apps
  - Foster collaboration and innovation
    - IT: "It doesn't work!"
    - Devs: "What? It works for me"



- Plan
- Code
- Build
- Test
- Release
- Deploy
- Operate
- Monitor

Run on bare metal

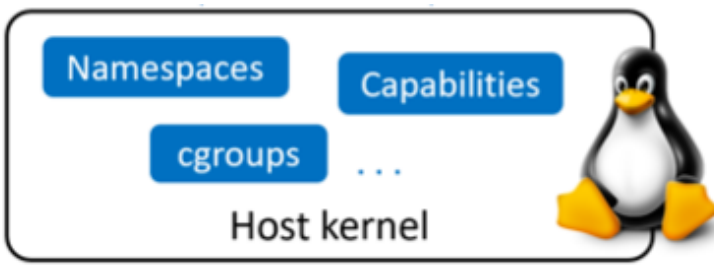
- $<$ 2000s
  - Running few applications per server (no virtualization)
- **Pros**
  - No virtualization overhead
- **Cons**
  - Insecure due to no application separation
  - Expensive
- **Expensive / low efficiency**
  - New server for each application
  - Hard to spec machines $\rightarrow$ buy "big and fast servers"
    - Overpowered servers at 5-10% capacity
    - Significant spending during 2000 DotCom boom on machines and networks
- Resurfaced in 2020 with different Cloud Computing use cases
- **Winners**: Cisco, Sun, Microsoft

Virtual Machine Era

- **Circa 2000-2010: Virtual Machine**
  - Run multiple OS copies on the same hardware
- **Pros**
  - Run multiple apps safely on one server
  - Use existing servers with spare capacity
- **Cons**
  - Each VM needs an OS (wastes CPU, RAM, disk)
  - Buy OS license
  - Monitor and patch each OS
  - Slow boot times
- **Winners**: VMWare, RedHat, Citrix

Containers Era

- **Circa 2013: Docker becomes ubiquitous**
- **Docker**
  - Didn't invent containers
  - Made containers simple and mainstream
- Linux supported containers
  - Kernel namespaces
  - Control groups
  - Union filesystems
- **Pros**
  - Containers are fast and portable
  - Don't require full OS
  - All run on a single host
  - Reduce OS licensing cost
  - Reduce OS patching and maintenance
- **Cons**
  - CPU overhead
  - Toolchain to learn / use
- **Winners**: AWS, Microsoft Azure, Google (not Docker Inc.)

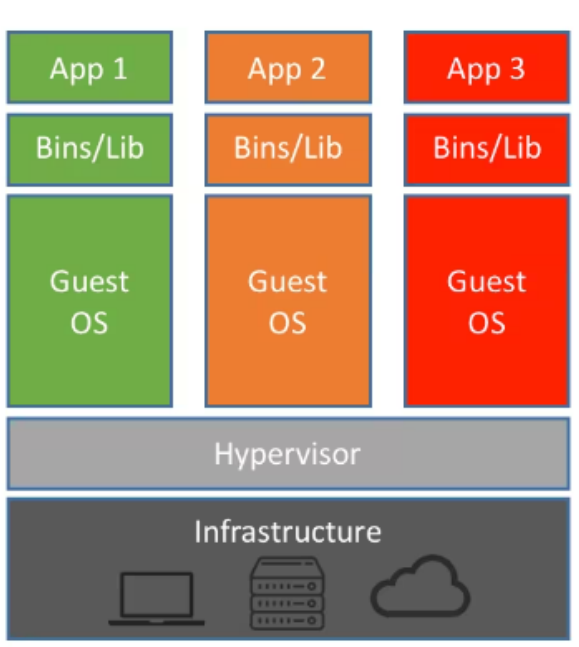


Serverless Computing

- **Containers run in an OS, OS runs on a host**
  - **Where is the host running?**
    - Local (your laptop)
    - On premise (your computer in a rack)
    - Cloud instance (e.g., AWS EC2)
  - **How is the host running?**
    - On bare-metal server
    - On a virtual machine
    - On a virtual machine running a virtual machine
- **Serverless computing**
  - Application runs without concern for *"how"* or *"where"*
  - E.g., AWS Lambda

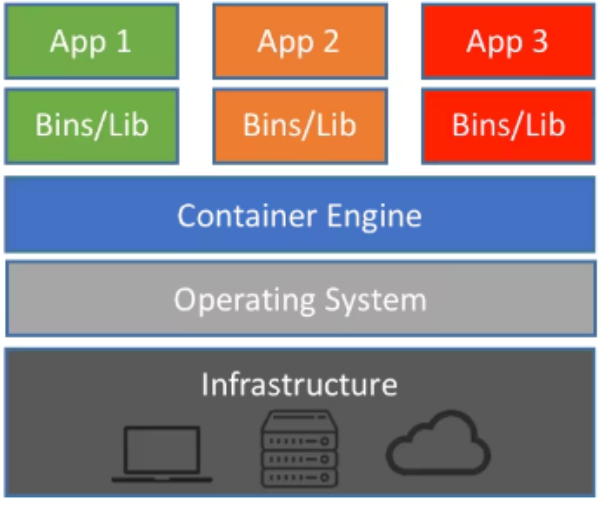
HW vs OS Virtualization

- **Hypervisor performs HW virtualization**
  - Carves physical hardware into VMs
  - Allocates CPUs, RAM, storage to a VM
  - Like having multiple computers
- "Virtual machine tax"
  - Running 3 apps requires 3 VMs
    - Each VM takes time to start
    - Consumes CPU, RAM, storage
    - Needs OS license
    - Requires admins, patching
  - You just want to run 3 apps!



Machine Virtualization

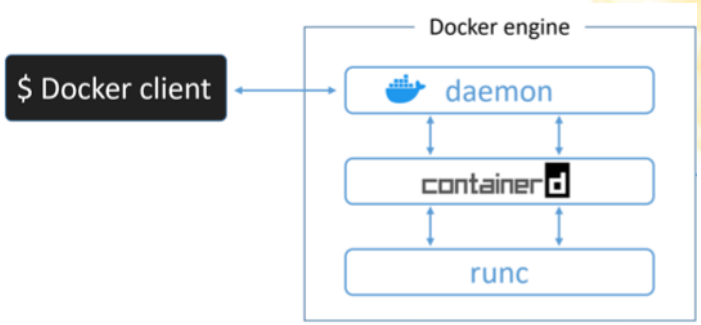
- **Containers perform OS virtualization**
  - It's like having multiple OSes

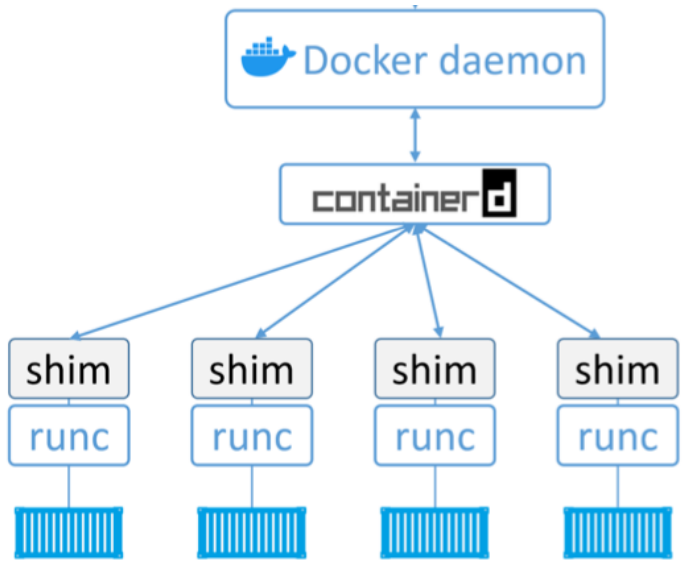


Containers

Docker: Client-Server

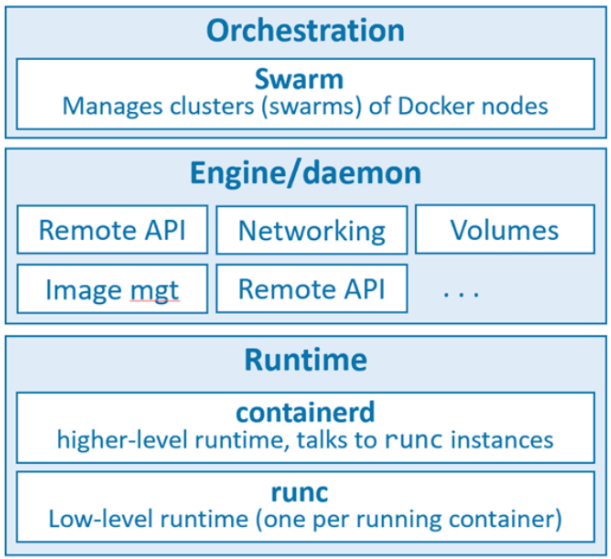
- Docker relies on a **client-server architecture**
- **Docker client**
  - Command line interface
  - Communicate with server through IPC socket
    - E.g., `/var/run/docker.sock` or IP port
- **Docker engine**
  - Run and manage containers
  - Modular, built from OCI-compliant sub-systems
    - E.g., docker daemon, `containerd`, `runc`, plug-ins for networking and storage





Docker Architecture

- **Docker run-time**
  - `runc`: start and stop containers
  - `containerd`
    - Pull images
    - Create containers, volumes, network interfaces
- **Docker engine**
  - `dockerd`
    - Expose remote API
    - Manage images, volumes, networks
- **Docker orchestration**
  - `docker swarm`
  - Manage clusters of nodes
  - Replaced by Kubernetes
- **Open Container Initiative (OCI)**
  - Standardize low-level components of container infrastructure
  - E.g., image format, run-time API
  - "Death" of Docker



Docker Container

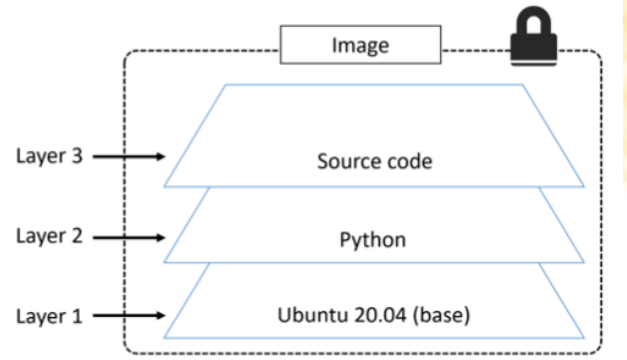
- **Docker Container**
  - Unit of computation
  - Lightweight, stand-alone, executable software package
- **Includes everything needed to run**
  - Code
  - Runtime/system libraries
  - Settings
- **Run-time object**
  - Docker images are build-time objects
  - Like program running (container) vs program code (image)

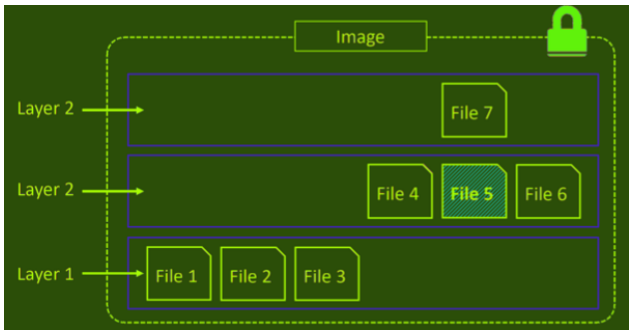
Docker Image

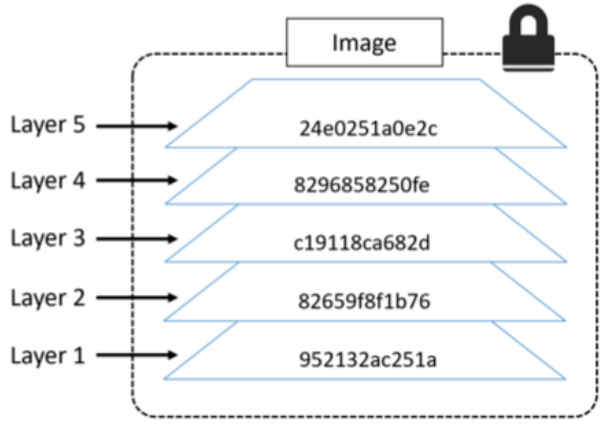
- **Docker Image**
  - Unit of deployment
- **Contains everything needed to run an app**
  - Application code
  - Application dependencies
  - Minimal OS support
  - File system
- Users can
  - Build images from Dockerfiles
  - Pull pre-built images from a registry
- Multiple layers stacked
  - Typically few 100s MBs

Docker Image Layers

- **Docker image** is a configuration file listing layers and metadata
  - Composed of read-only layers
  - Each layer is independent
  - Each layer comprises many files
- **Docker driver**
  - Stacks layers as a unified filesystem
  - Implements copy-on-write behavior
  - Files from top layers can obscure files from bottom layers
- **Layer hash**
  - Each layer has a hash based on content
  - Layers are pulled and pushed compressed
- **Image hash**
  - Each image has a hash
  - Hash is a function of the config file and layers
  - Image changes generate a new hash





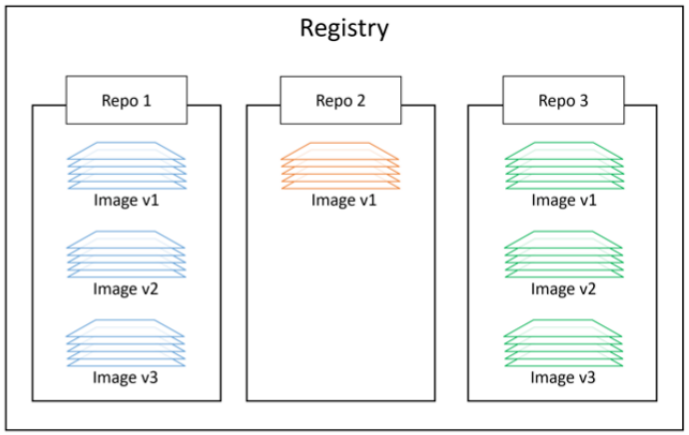


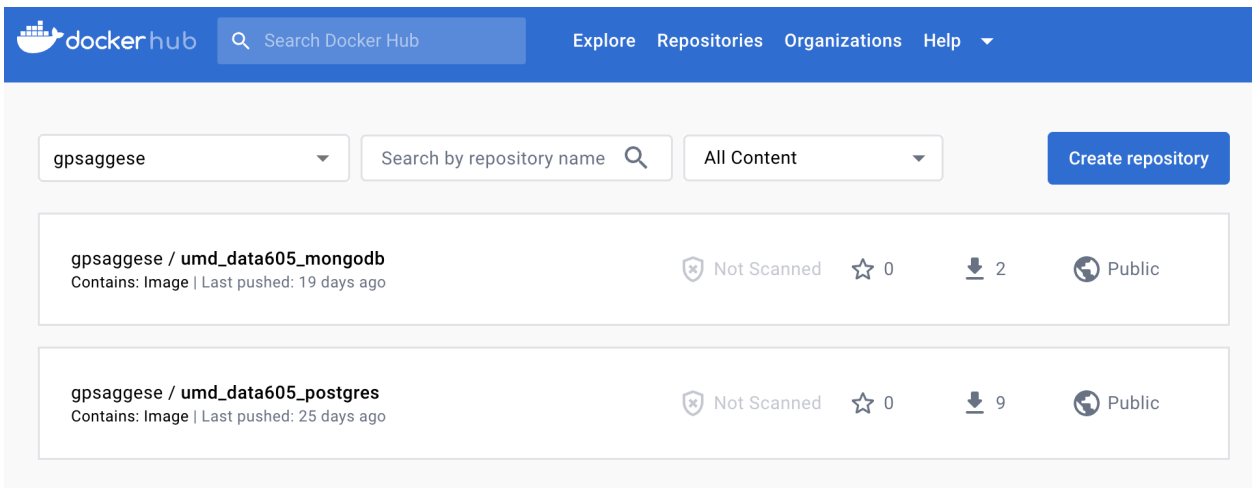
Docker: Container Data

- A container has access to different data
- **Container storage**
  - Copy-on-write layer in the image
  - Ephemeral (temporary data)
  - Data persists until the container is killed
    - Stopping or pausing a container doesn't lose data
  - Containers are immutable
    - Avoid writing persistent data into containers
- **Bind-mount a local dir**
  - Mount a local dir to a dir inside a container
- **Docker volumes**
  - Volumes exist separately from the container
    - E.g., store Postgres DB content
  - State is permanent across container invocations
  - Shareable across containers

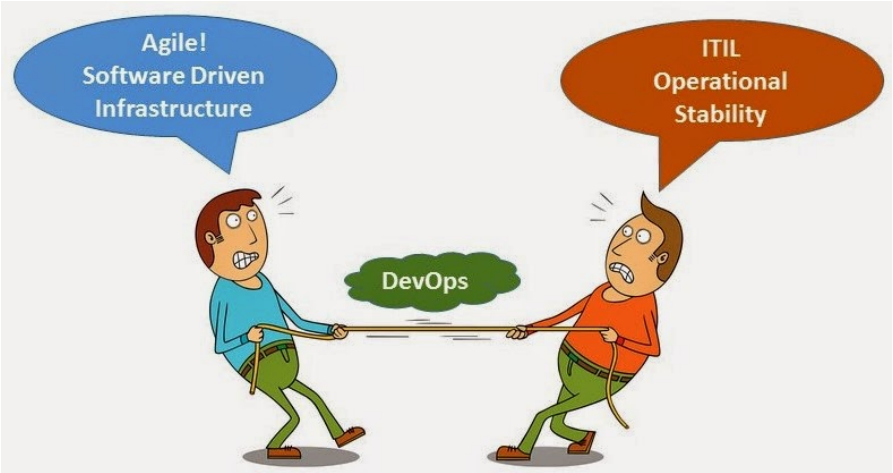
Docker Repos

- **Docker Repo (Registry)**
  - E.g., DockerHub, AWS ECR
  - Store Docker images
    - `<registry>/<repo>:<tag>`
    - E.g., `docker.io/alpine:latest`
  - Some repos are vetted by Docker
  - Unofficial repos shouldn't be trusted
  - E.g., https://hub.docker.com/





Devops = Devs + Ops



- **Devs**
- Implement app
  - Python, virtual env
- Containerize app
  - Create Dockerfile
  - Instructions to build image
- Build image
- Run app as container
- Test locally

- **Ops**
- Download container images
  - Filesystem, application, dependencies
- Start / destroy containers
- Reproduce issues easily
  - "Here is the log"
  - Run command line
  - Deploy on test system and debug