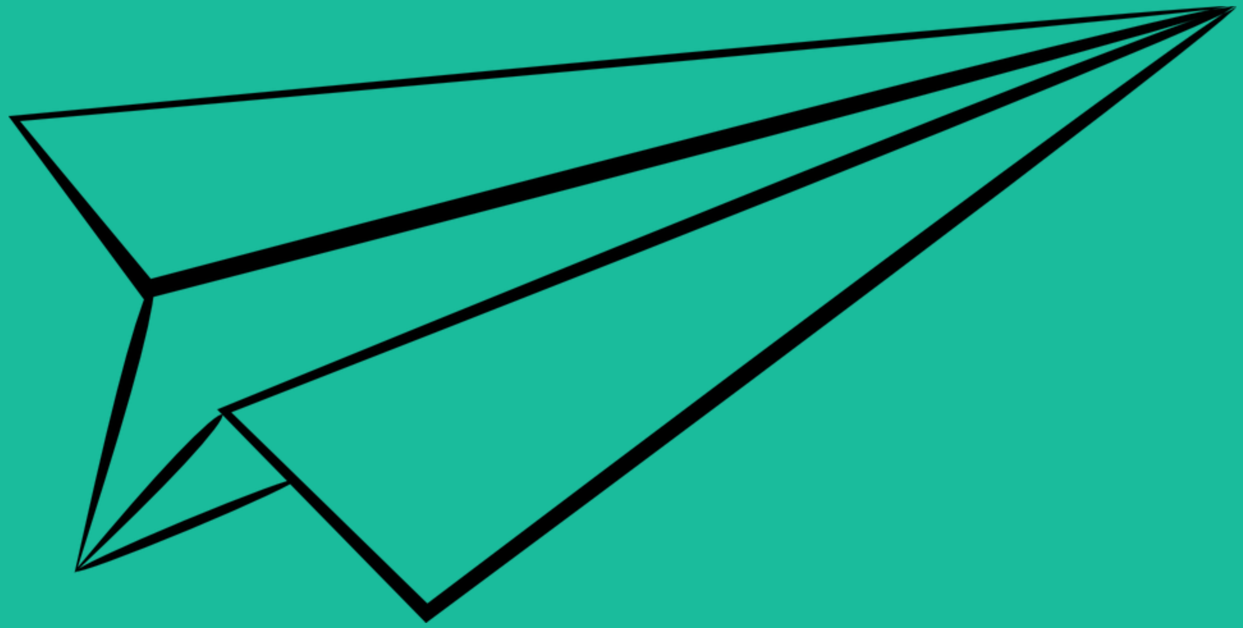


Chapter 7. Virtualization & Containers

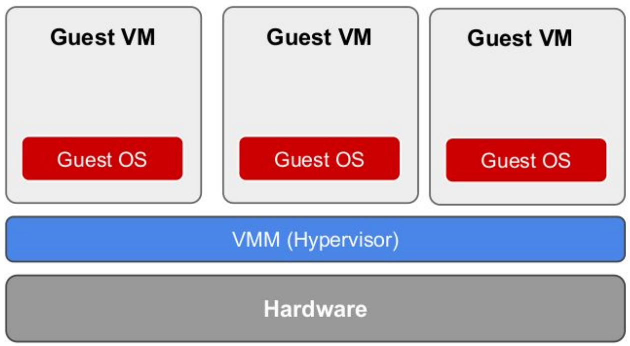
Bilkent University | CS443 | 2021, Spring | Dr. Orçun Dayıbaş

Introduction

- **Virtualization**
  - Technology that transforms hardware into software upon which other software runs
- **Types of virtualizations**
  - Hardware/Platform virtualization
    - Hardware virtualization is the virtualization of computers as **complete hardware platforms**, certain logical abstractions of their componentry, or only the functionality required to run various operating systems
    - Ex: Oracle VirtualBox, VMware workstation, Hyper-V, ESX, etc.
  - Software virtualization
    - Virtualization is provided by the host OS
    - OS kernel performs all the functionalities of a fully virtualized hypervisor by allowing existence of multiple user space instances (containers)
    - Ex: Docker, LXD, RKT, etc.

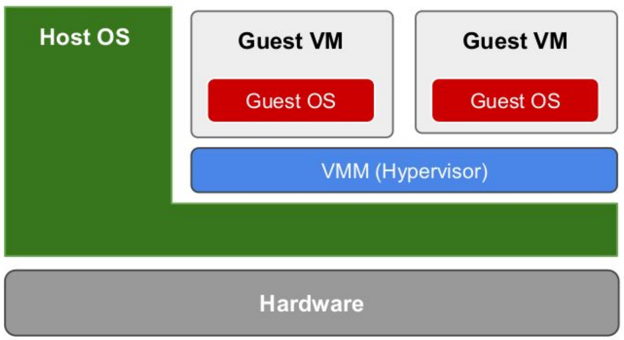
Hardware Virtualization

- **Architecture**



Bare metal architecture

- Xen, VMware ESX server, Hyper-V
- Mostly for server, but not limited
- VMM by default
- OS-independent VMM



Hosted architecture.

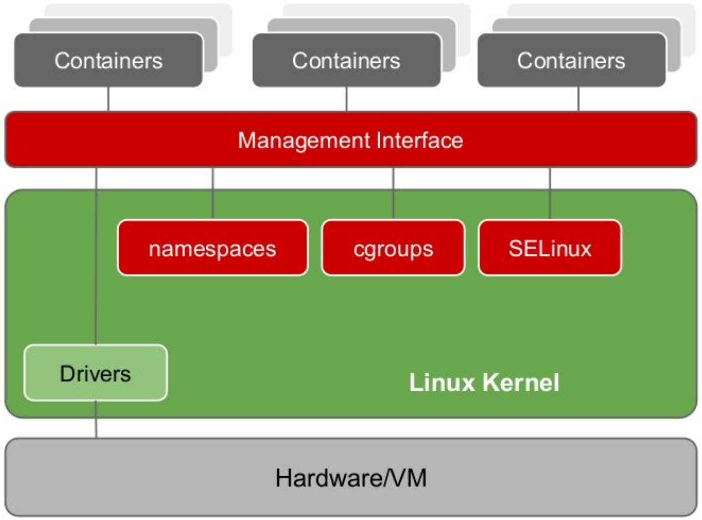
- VMware Workstation, VirtualBox
- Mostly for client devices, but not limited
- VMM on demand
- OS-dependent VMM

(source)

- **Features**
  - Each virtual machine (VM) has **its own kernel**
  - **Hypervisor manages these VMs** in host operating system by allocating hardware resources to them thereby allowing you to have several VMs all working optimally on **a single piece of computer hardware.**

Software Virtualization

- **Architecture**



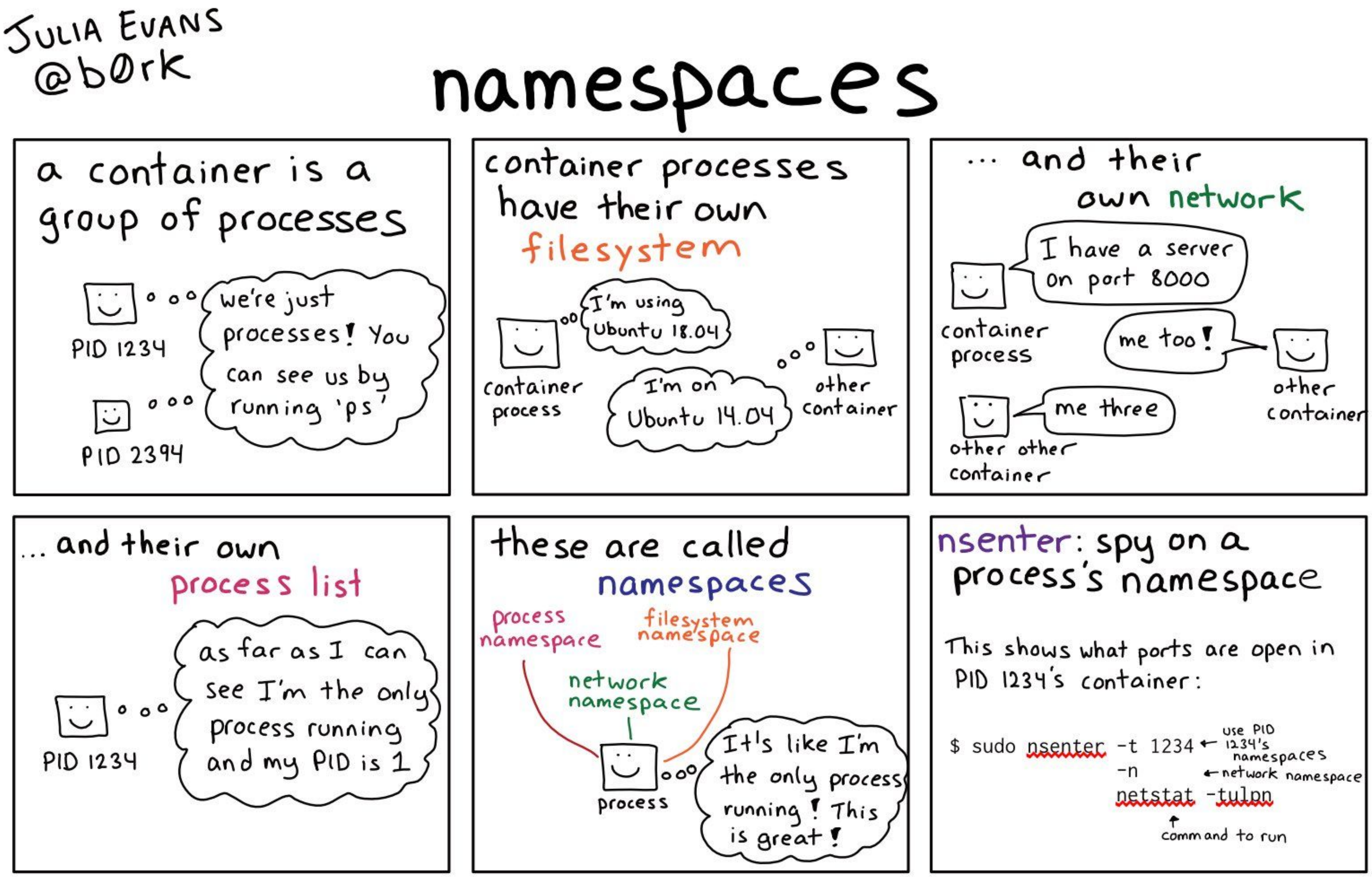
- **Features**
  - Lightweight OS-level virtualization, **No external hypervisor**
  - Illusion of running multiple OSs on a single machine sharing same host kernel
  - Lots of different implementations (e.g. LXC, Docker, OpenVZ)
    - https://en.wikipedia.org/wiki/OS-level_virtualization#IMPLEMENTATIONS

Software Virtualization

- **Linux Containers**
  - Containers are not a new technology: the earliest iterations of containers have been around in open source Linux code **for decades**
  - “A container” is a group of processes (isolated from other processes)
    - They can have their own users, network namespace, file system, process IDs, mem/CPU limits
  - Kernel features that isolates containers
    - **namespaces:** isolates process trees, networking, user IDs, file system
    - **cgroups:** allows limitation and prioritization of resources (CPU, memory, etc.)
    - **Security-Enhanced (SE) Linux:** provides secure separation of containers by applying SELinux policies and labels. It integrates with virtual devices by using the sVirt technology.
  - There are many ways to run linux containers and containers can be set up different isolations
    - You can write your own bash script (Ex: Containers from scratch)

Software Virtualization

- **Linux Containers**



Software Virtualization

- **Linux Containers**

JULIA EVANS
@b0rk

containers aren't magic

These 15 lines of bash will start a container running the fish shell. Try it!
(download this script at bit.ly/containers-arent-magic)

```
wget bit.ly/fish-container -O fish.tar        # 1. download the image
mkdir container-root; cd container-root       #
tar -xf ../fish.tar                           # 2. unpack image into a directory
cgroup_id="cgroup_$(shuf -i 1000-2000 -n 1)" # 3. generate random cgroup name
cgcreate -g "cpu,cpuacct,memory:$cgroup_id"  # 4. make a cgroup &
cgset -r cpu.shares=512 "$cgroup_id"          #    set CPU/memory limits
cgset -r memory.limit_in_bytes=1000000000 \   #
      "$cgroup_id"                            #
cgexec -g "cpu,cpuacct,memory:$cgroup_id" \  # 5. use the cgroup
    unshare -fmuipn --mount-proc \            # 6. make + use some namespaces
    chroot "$PWD" \                           # 7. change root directory
    /bin/sh -c "                              #
        /bin/mount -t proc proc /proc &&      # 8. use the right /proc
        hostname container-fun-times &&       # 9. change the hostname
        /usr/bin/fish"                        # 10. finally, start fish!
```

Software Virtualization

- **Container vs. Virtual Machine**
  - Lightweight & fast
    - Consumes much less memory (provides density, theo. limit; 6K instances/host)
    - Faster startup & shutdown (since the kernel and HW resources shared)
  - System-wide updates are easier (changes are visible to all)
  - Less secure

VMs	Containers
Heavyweight	Lightweight
Limited performance	Native performance
Each VM runs in its own OS	All containers share the host OS
Hardware-level virtualization	OS virtualization
Startup time in minutes	Startup time in milliseconds
Allocates required memory	Requires less memory space
Fully isolated and hence more secure	Process-level isolation, possibly less secure

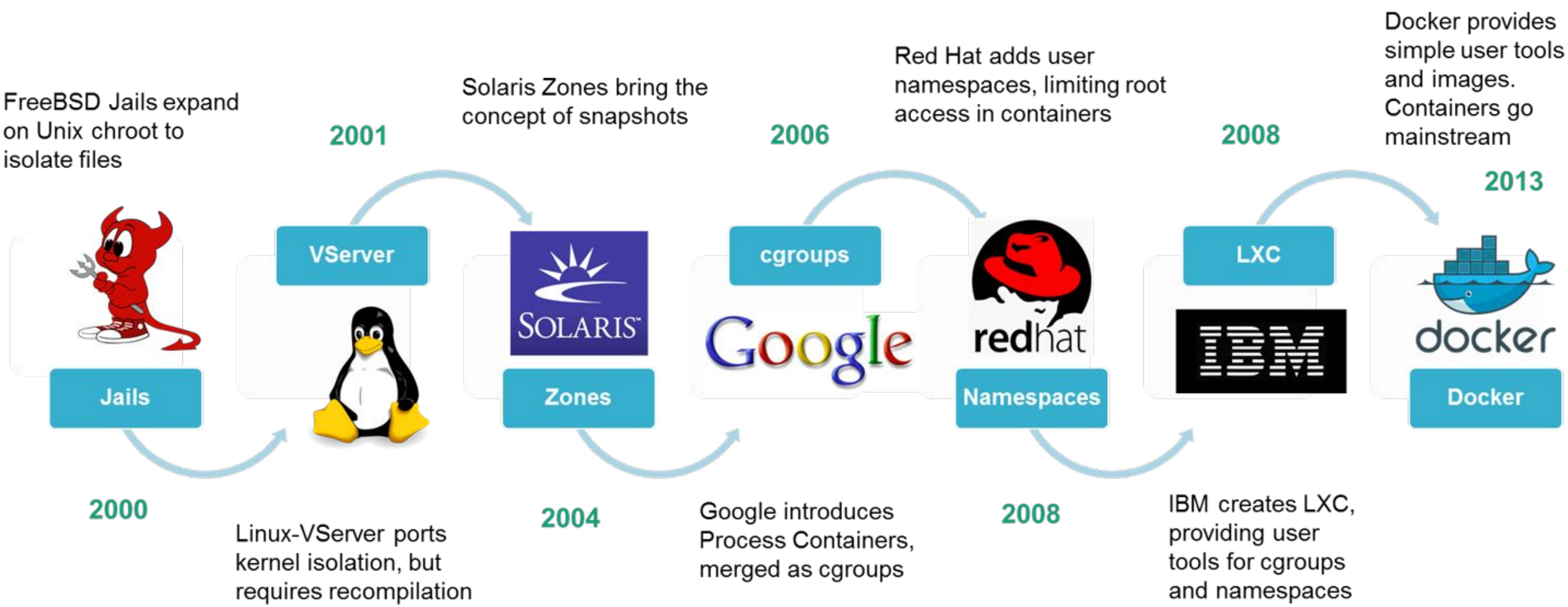
Software Virtualization

- **Will containers kill VMs?**
    - It is not possible to run a container with **a guest OS that differs from the host OS** because of the shared kernel
    - Users with **heterogeneous environments** that include **multiple OSs** and **different security controls** will likely still use a VM-focused architecture

"ship-shipping ship, shipping shipping ships"

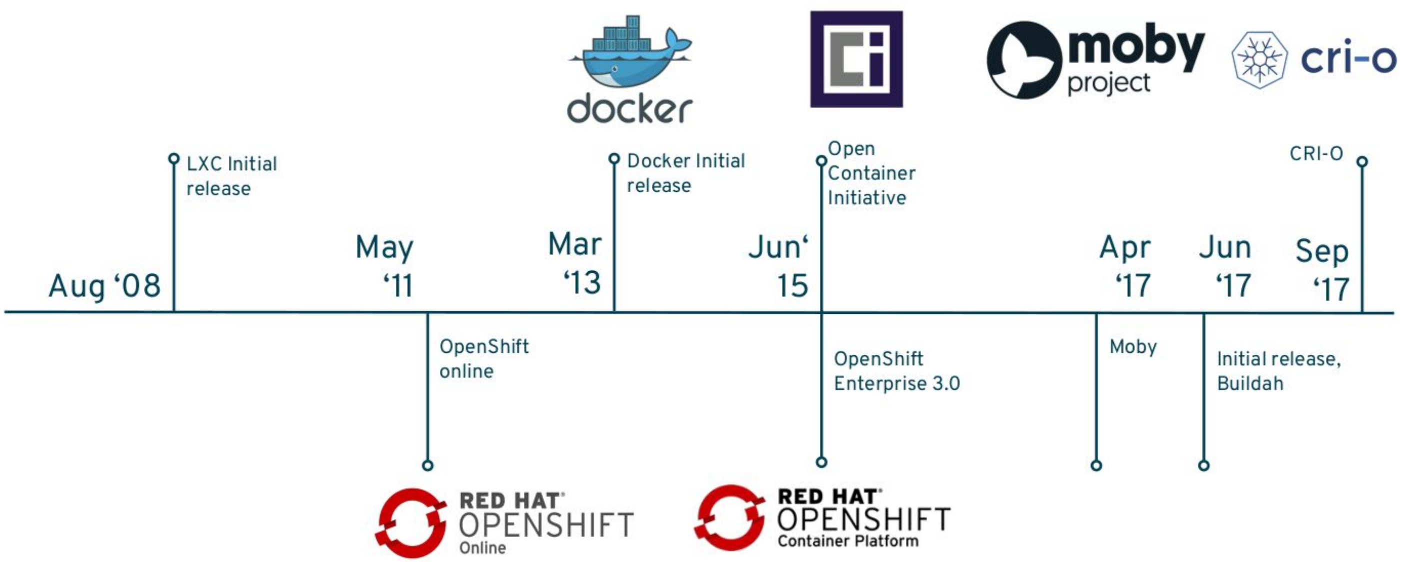
Docker

- **What is Docker?**
  - Software containerization platform
  - An extension of LXC's capabilities
    - LXC is a userspace interface for the Linux kernel containment features



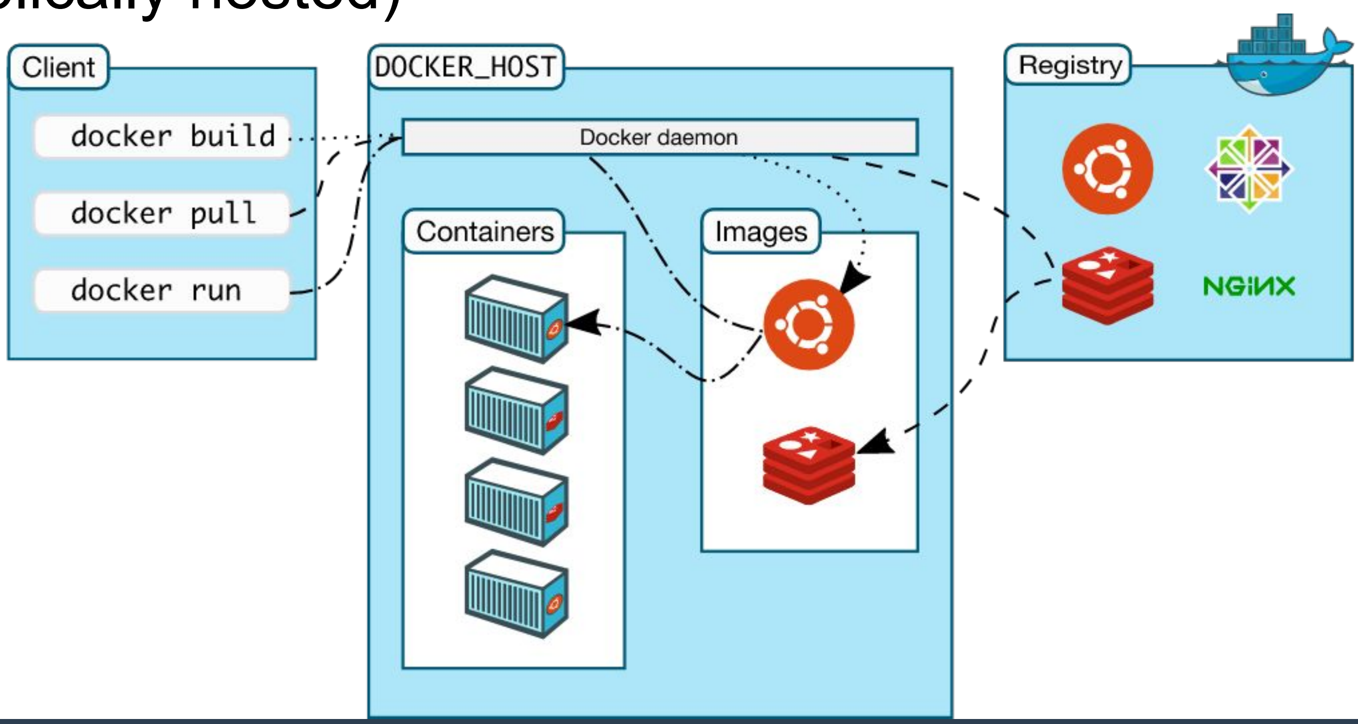
Docker

- **What is Docker?**
  - Huge business success, current valuation: ~$1 billion
  - Standardization effort started in 2015 (OCI)
  - Docker launched “enterprise edition” in 2017
  - Current products: desktop, hub, etc.



Docker

- **Client-Server Architecture**
  - Client - Communicates with docker host, gives it instructions to build, run and distribute your applications
  - Host - Communicates with clients via [RESTful API]
  - Registry - A library of docker images (can be locally or publically hosted)



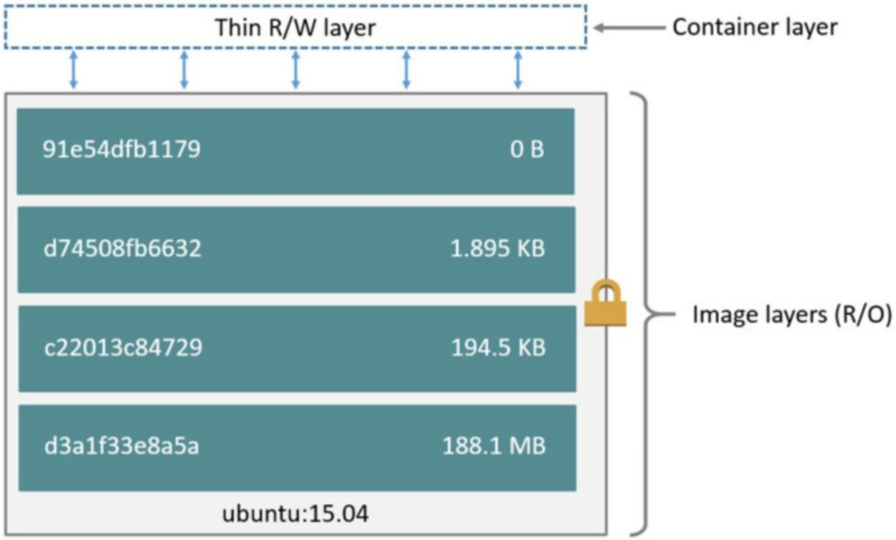
Docker

- **Images & layers**
  - Images are immutable snapshots (can be build from scratch)
  - Images are downloaded via any **Docker registry**
  - Every image extends a base image (e.g. ubuntu, alpine)
  - **Dockerfile:** instructions to create an image file
  - Each image consists of series of layers
    - New layer built on every application update
    - Layering makes it easier to distributes updates to a dockerized application since **only updated layer is transferred over the network**
  - **Docker container:** runnable instance of an image, created when images are started with “run” command
  - The most popular Images
    - https://hub.docker.com/search?q=&type=image

Docker

- **Images & layers**
  - Example:

```
FROM ubuntu:15.04
RUN apt-get -y install nginx
COPY index.html /var/www/html
CMD ["nginx","-g","daemon off;"]
```



Container
(based on ubuntu:15.04 image)

- Each layer is **only a layer of differences** from the layer before it
- The layers are **stacked** on top of each other
- When you create a new container, you add **a new writable layer on top** of the underlying layers. This layer is often called the “container layer”
- All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer

Docker

- **Images & layers**
  - A storage driver handles the details about the way these layers interact with each other
  - Different storage drivers are available (with diff. pros/cons)
    - overlay2, devicemappers, aufs, btrfs, etc.
    - overlay2 is the default one
  - You can view the contents of each layer on the Docker host at `/var/lib/docker/overlay2/diff`
  - Layers are important because they can be re-used by multiple images
    - Means saving disk space and reducing time to build images while maintaining their integrity

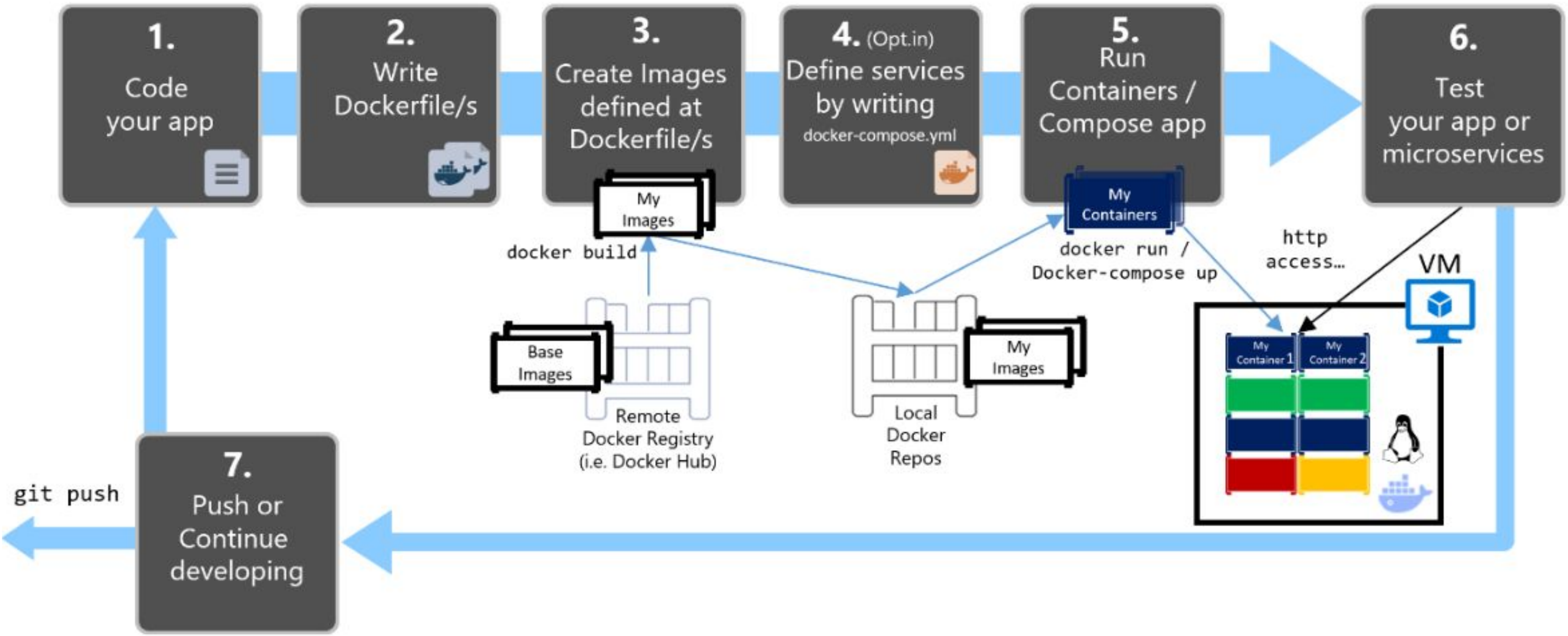
OverlayFS is a union mount filesystem implementation for Linux. It combines multiple different underlying mount points into one, resulting in single directory structure that contains underlining files and sub-directories from all sources.

Docker

- **Volumes**
    - Data directory which can be initialized within the container
        - Can be initialized via image at runtime or configured in Dockerfile
    - Data volume is shared with the host machine in `/var/lib/docker/volumes` directory
    - Any change in data directory within the container is **reflected in real-time** in the host machine and vice versa
    - Can also mount a directory from your Docker engine's host into a container
        - This helps data volumes to be shared among multiple containers simultaneously
    - Persist even if container is deleted

Docker

- **Application development workflow**



source:
https://docs.microsoft.com/en-us/dotnet/architecture/microservices/docker-application-development-process/docker-app-development-workflow

Docker

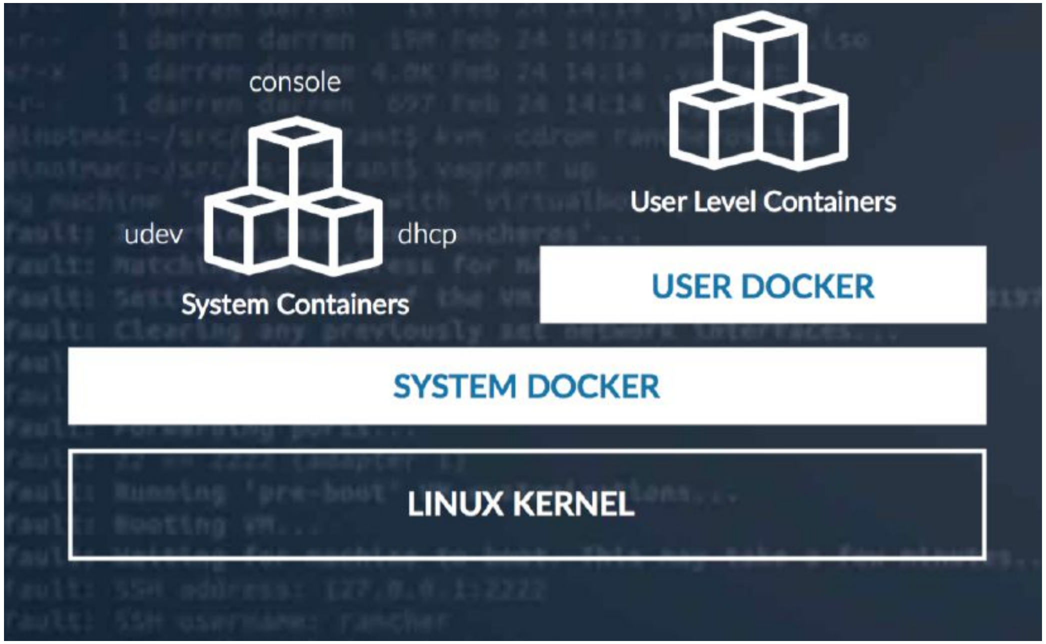
- **Pros & Use Cases**
  - Simplifies distribution, shipping and deployment of apps
  - Build once, run anywhere
  - No worries of missing dependencies, installing and configuring the application during subsequent deployments
  - Running of multiple/similar versions of same app/library in same host machine (Each application runs in its own isolated container)
  - Easier to scale applications (already packed and installed, just run it)
  - Easier to run you application as a failsafe long running service

Minimalistic (Container-Opt.) OS

- **Features**
  - Small footprint, Atomic updates
    - The Container-Optimized OS kernel is locked down; you'll be unable to install third-party kernel modules or drivers
  - No package management
    - Container-Optimized OS does not include a package manager; as such, you'll be unable to install software packages directly on an instance
  - Everything runs on a container
    - Container-Optimized OS does not support execution of non-containerized applications
- **Examples**
  - CoreOS (Google → Red Hat)
  - Atomic (Red Hat)
  - Snappy Ubuntu Core (Canonical)
  - Photon OS (VMware)
  - RancherOS (Rancher Labs)

Minimalistic (Container-Opt.) OS

- **Ex: RancherOS**
  - Every process in RancherOS is a container managed by Docker (This includes system services e.g. `udev`, `syslog`)
    - Security benefits + `docker rm -f $(docker ps -qa)` don't delete the entire OS
  - Docker as PID 1, ~20MB in total



Container Orchestration Tools

- **Example Case**
  - Assume an online and on-demand entertainment movie streaming company, called "NetPly"
    - Delivers **video streams** to your favorite devices and provides personalized movie recommendations to their customers based on their previous activities, such as sharing or rating a movie
    - NetPly runs **15,000 production servers** worldwide and follow **agile methodology to deploy** new features and bug fixes to the production environment
  - NetPly has been struggling with two fundamental issues in their software development lifecycle
    - **Issue 1-** Code that runs perfectly in a development box, sometimes fails on test and/or production environments
    - **Issue 2-** Viewers experience a lot of lags as well as poor quality and degraded performance for video streams during weekends, nights, and holidays, when incoming requests spike

source: https://rancher.com/blog/2018/2018-10-09-kubernetes-versus-docker/

Container Orchestration Tools

- **Need**
  - A container is standardized & self-contained software package. Hence it provides platform independence & operational simplicity
    - Docker has become the most popular container technology worldwide, despite a host of other options, including RKT from CoreOS, LXC, LXD from Canonical, OpenVZ, and Windows Containers
  - However, container technology alone is not enough to reduce the complexity of managing containerized applications
    - Software projects get more and more complex and require the use tens of thousands of containers
    - A few containers to manage → we're fine with Docker CLI (+scripting maybe)
    - Hundreds of containers → Container orchestration tool
      - For instance, think of architecture with several microservices, all with distinct scalability and availability requirements.

Container Orchestration Tools

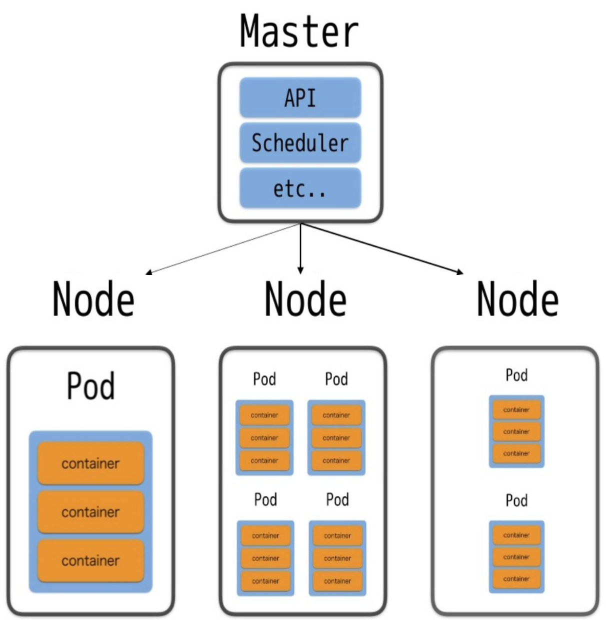
- **Definition**
  - A container orchestration system treats a cluster of machines with a multi-container application as a single deployment entity. **It provides automation** from initial deployment, scheduling, updates to other features like monitoring, scaling, and failover
- **Examples**
  - Docker EE (Docker Swarm)
  - Apache Mesos (Mesosphere Marathon)
  - Kubernetes
  - Nomad

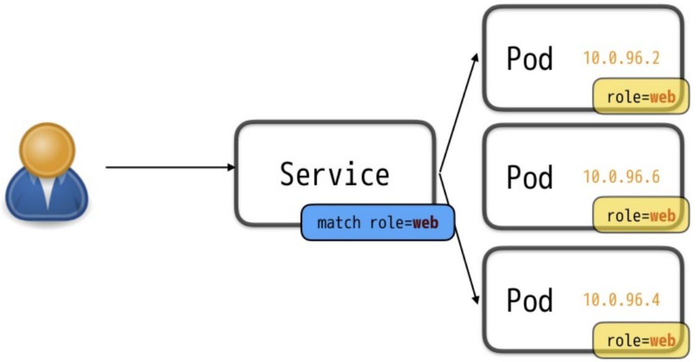
Container Orchestration Tools

- **Kubernetes**
  - Kubernetes is an open-source container-orchestration system for automating application deployment, scaling, and management
  - It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation (CNCF)
  - Fundamental features
    - Load balancing
    - Configuration management
    - Automatic IP assignment
    - Container scheduling
    - Health checks and self healing
    - Storage management
    - Auto rollback and rollout
    - Auto scaling

Container Orchestration Tools

- **Kubernetes**
  - Basics
    - Node: Hosts running k8s daemons
    - Pod: Basic unit of deployment (co-scheduled, co-located, etc.)
      - Stateless, has its own IP
    - Service: logical set of pods
    - Namespace: virtual clusters (backed by the same physical cluster)
  - Higher-level abstractions
    - Deployment
    - DaemonSet
    - StatefulSet
    - ReplicaSet
    - Job
    - etc.





source: https://www.slideshare.net/lestrrat/kubernetes-in-30-minutes-20170310

Scalable Container Management Services

- **Amazon ECS (Elastic Container Service)**
    - Highly scalable and fast container management service
    - Easily deploy and scale Docker containers on cluster of EC2 instances
    - Supports auto-scaling
    - Create task definitions using hosted Docker images
    - Run tasks or create services from this task definition to run on **EC2 cluster**
    - Start service via aws console or aws-cli
    - Integrated with **Amazon ECR** (fully managed docker container registry)
        - Eliminates the need to operate your own container repositories
        - Pay only for data stored in repos & transferred to internet

Scalable Container Management Services

- **Azure Container Service**
  - Allows you to quickly deploy a production ready Kubernetes, DC/OS, or Docker Swarm cluster
  - Deprecated by January 31, 2020. Migration path:
    - ACS with Kubernetes → Azure Kubernetes Service or aks-engine open-source project
    - ACS with Docker → Docker Enterprise Edition for Azure solution template
    - ACS with DC/OS → Mesosphere DC/OS Enterprise or Mesosphere DC/OS Open Source solution template
  - "AKS Engine" is not covered by the Microsoft Azure support policy
    - https://support.microsoft.com/en-us/help/2941892/support-for-linux-and-open-source-technology-in-azure
    - The AKS Engine project maintainers will respond to the best of their abilities
- **Azure Kubernetes Service**
  - Managed Kubernetes that reduces the complexity for deployment and core management tasks

Scalable Container Management Services

- **Google Cloud Computing**
  - App Engine Flexible Environment
    - Containers on fully-managed VM-based PaaS to run an application in **one container**
    - Automatic high availability with built-in autoscaling and load balancing
  - Cloud Run
    - Containers on a fully-managed serverless environment
    - Built on Knative, enabling portability of your workloads across platforms
  - Compute Engine
    - **Containers on VMs** and on Managed Instance Groups (No container orchestration)
    - VM-level autoscaling, autohealing. Integrate a containerized application into your existing IaaS infrastructure
    - Direct access to specialized hardware, including local SSDs, GPUs, & TPUs
  - Kubernetes Engine
    - Managed Kubernetes
    - Automates container orchestration, including service health monitoring, node auto repair, autoscaling, auto upgrades, and rollbacks

Q/A