Cloud Computing Applications and Services

(Aplicações e Serviços de Computação em Nuvem)

Virtualization (Part II)

University of Minho

2026-2027



Containers

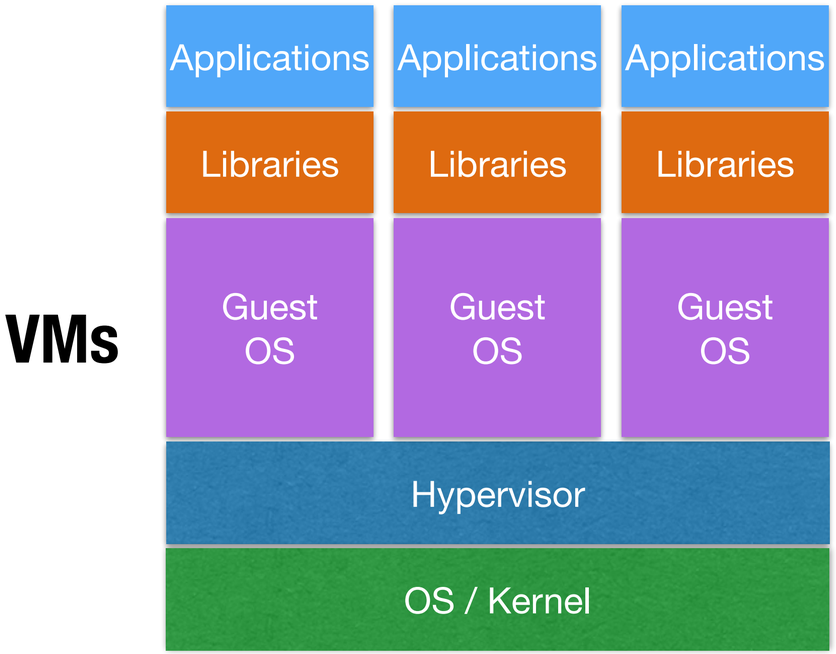


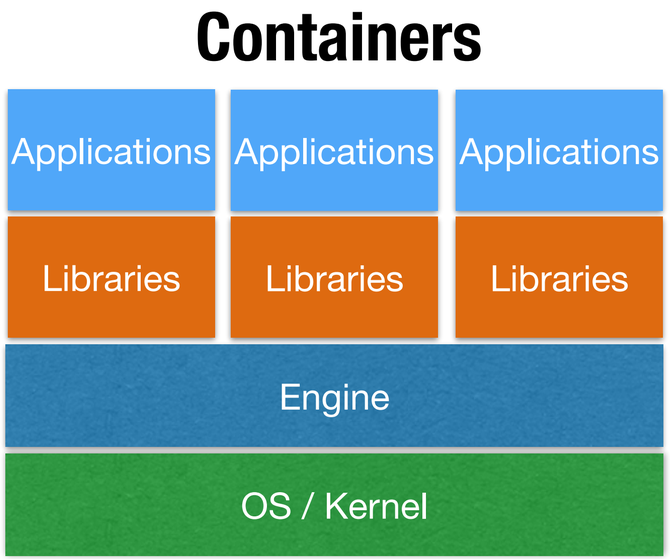
- **Lightweight virtual environment** that **groups and isolates a set of processes and resources** (memory, CPU, disk, …), from the host and any other containers
- Why are containers useful?
  - Running different isolated versions of the same software/application (*e.g.*, database) in a shared OS/Kernel environment
  - Portability/migration across servers
  - Easy packaging of software, applications and their dependencies

Linux Containers

Architecture

- Containers are lightweight, **not requiring full virtualization** of CPU, RAM, network, and storage requests (as in VMs)

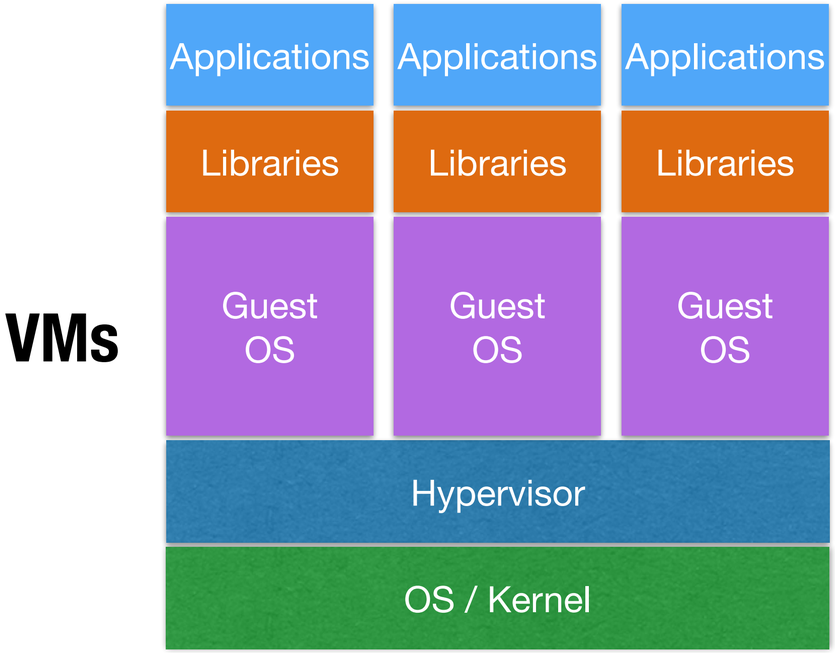


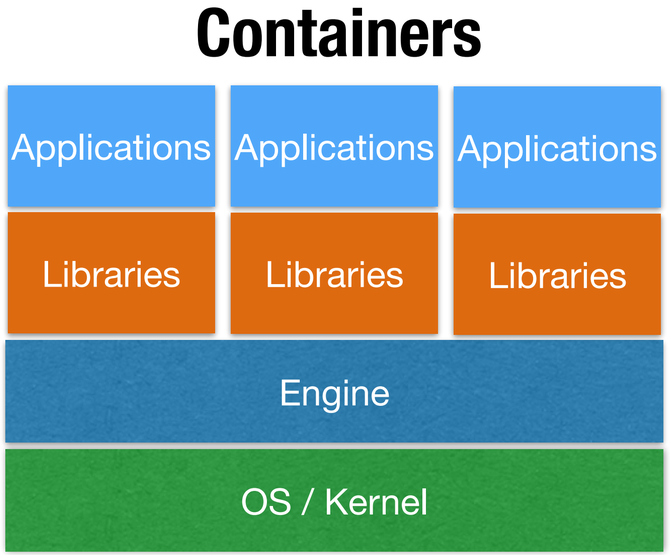


Linux Containers

Engine

- The **engine** isolates and configures resources
  - Containers are isolated from each other, i.e., the host is compartmentalized in terms CPU, RAM, memory, disk
  - Each container shares the hardware and kernel/OS with the host system





Linux Containers

Building Blocks (components of Linux kernel)

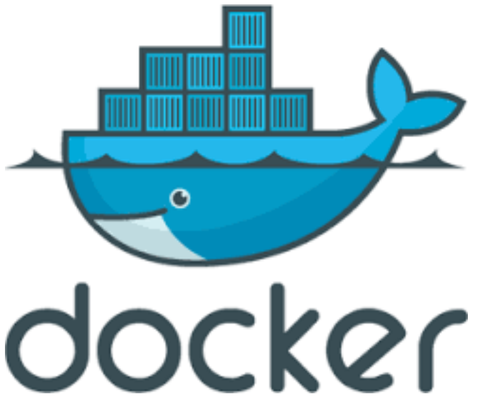
- **Namespaces** (Isolation)
  - Host resources (e.g., network, filesystem) are **partitioned** into dedicated resources that are only accessible by a certain group of processes (under the same namespace)
  - Allow **isolating** host resources across different containers
- **Control Groups** (Resource Management)
  - Allow **allocating resources** (CPU, RAM, Disk, network) among groups of processes
  - **Limit the amount of resources** per container (*e.g.*, CPU cores, RAM/storage)
- **SELinux** (Security)
  - Provides **additional security** over namespaces so that a container is not able to compromise the host system and other containers
  - Enforces **access control and security policies**

Linux Containers

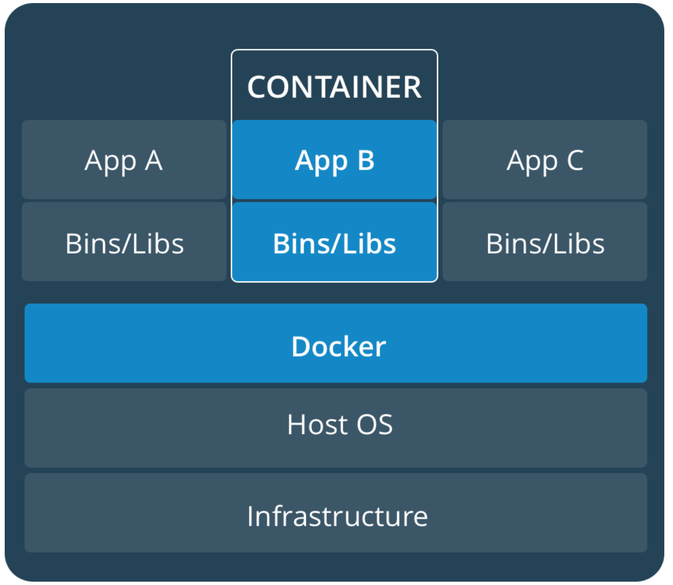
Types

- **OS Containers** (*e.g.*, LXC)
  - Containers run multiple processes and simulate a “lightweight” operating system
- **Application Containers** (*e.g.*, Docker)
  - Focused on deploying applications
  - Each application is seen as an independent process

Docker

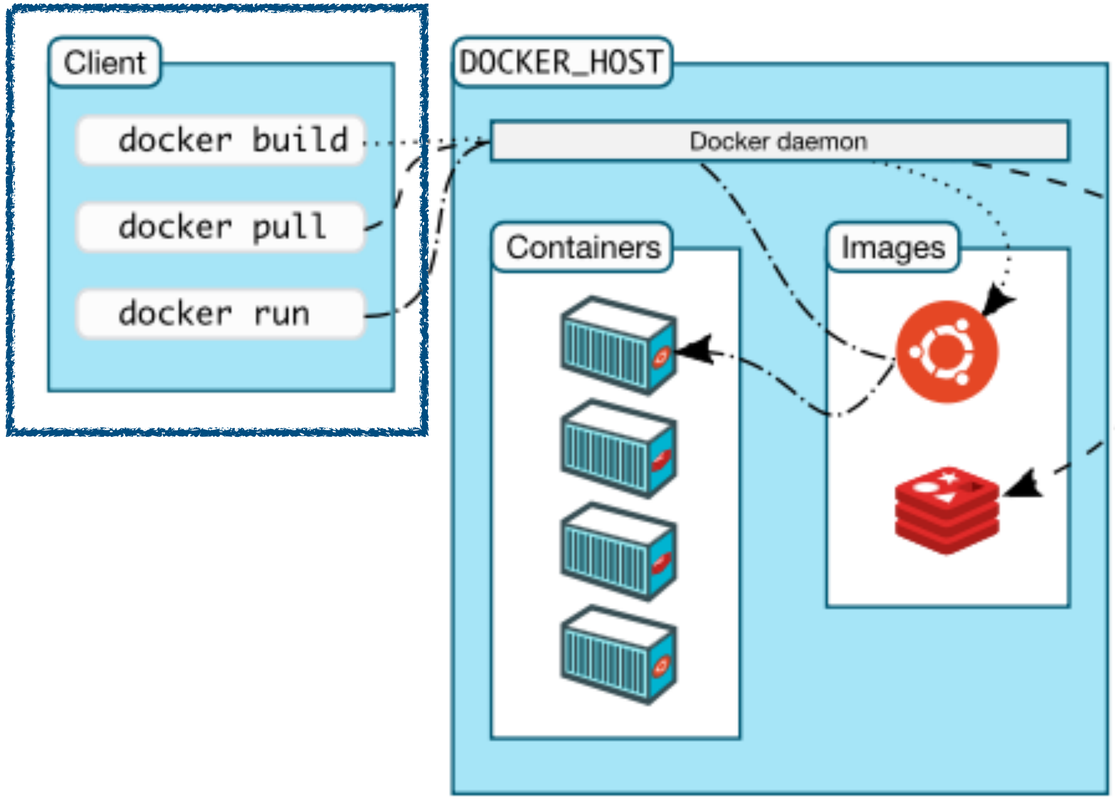


- Most widely-known container platform
- Supports Ubuntu, Fedora, RHEL, CentOS, Windows, etc
- https://www.docker.com



Docker
Docker Client

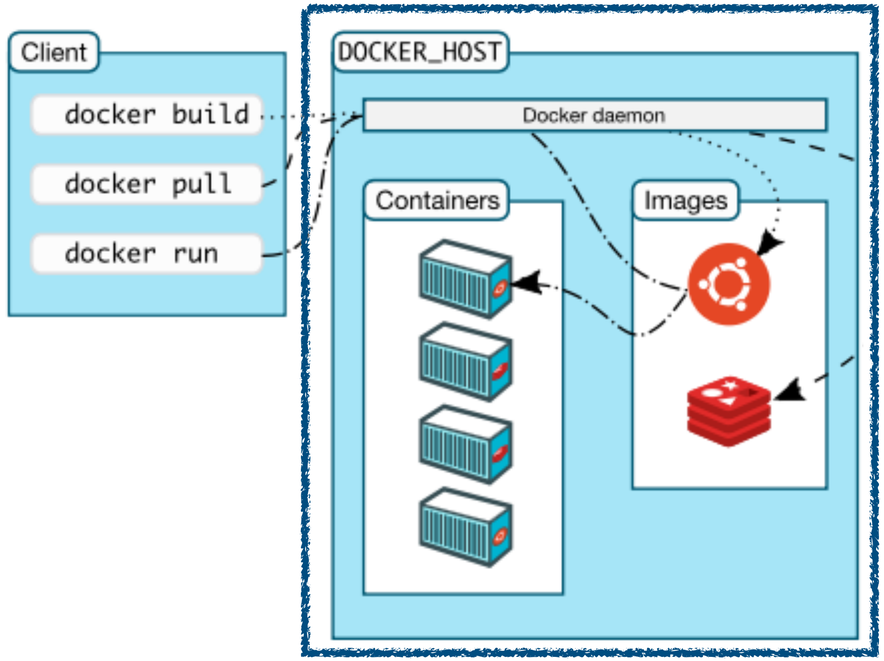
- Component used by users to interact with the Docker platform (daemon)
- Exposes the **Docker API** for:
  - Running and managing containers
  - Managing networks and volumes
  - Reading logs and metrics
  - Pulling and managing images
  - …



Docker

Docker Daemon and Objects

- Docker Daemon
  - Uses the Docker API for receiving requests from the Docker Client
  - **Manages Docker** Images, Containers, Volumes, Networks
- Docker Objects
  - **Image:** immutable (unchangeable) file that contains the source code, libraries, and other files needed for an application to run
  - **Container:** runnable instance of an image



Docker
Volumes

- Containers have an **internal file system** that is **ephemeral**
  i.e., the container's data is deleted once the container is removed

- Containers can **mount a file or directory from the host machine**. Stored data is independent from the container's internal file system and **persisted even if the container is removed**
  - **Bind mount:** Generic directory from the host. Any container or host process can access it
  - **Volume:** Special host directory managed by Docker and accessible only by containers

Find more about Docker Volumes at: https://docs.docker.com/storage/volumes/

Docker
Network

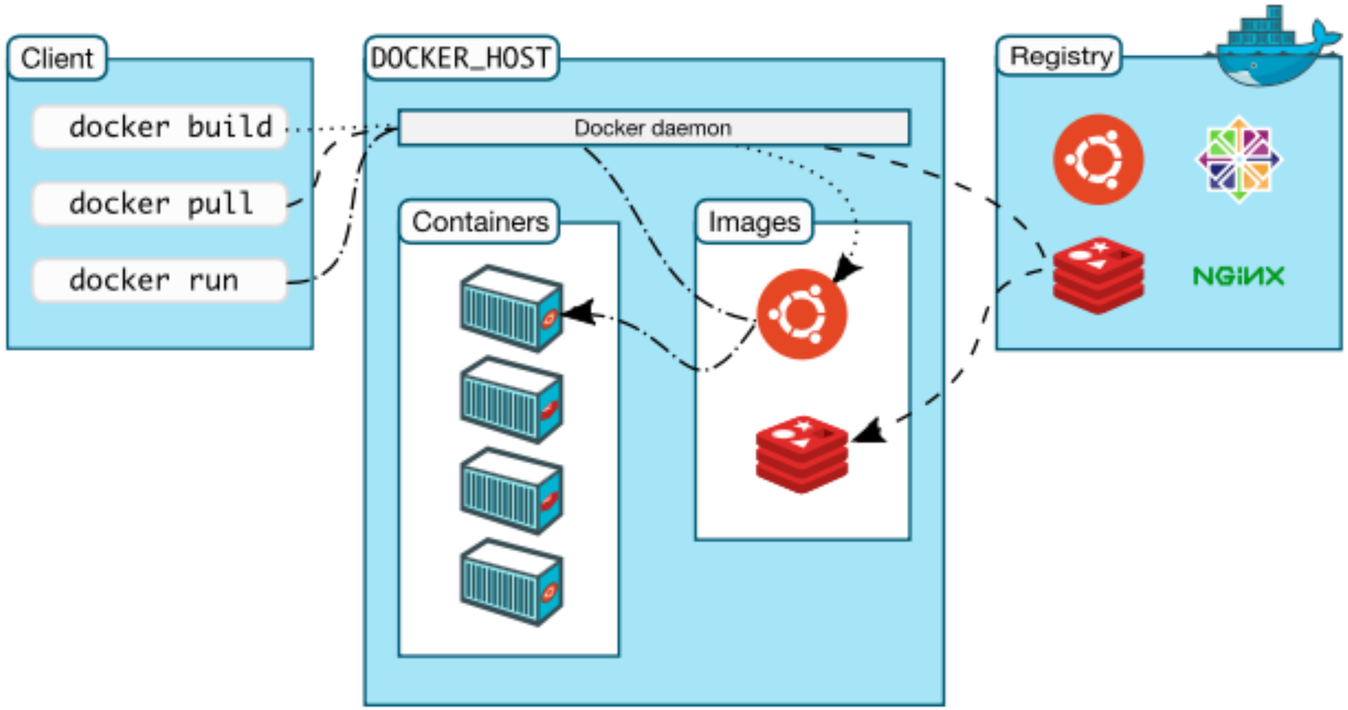
- **Host**:
  - Shares the host networking namespace
  - Container services are presented in the network as if run by the host
  - Ports are shared (e.g., port 80)
- **Bridge**:
  - The container is seen as another node in the physical network

Find more about Docker Networks at: https://docs.docker.com/network/

Docker

Docker Registry

- Repository of Docker Images
- Example: https://store.docker.com



Kubernetes (aka K8s)



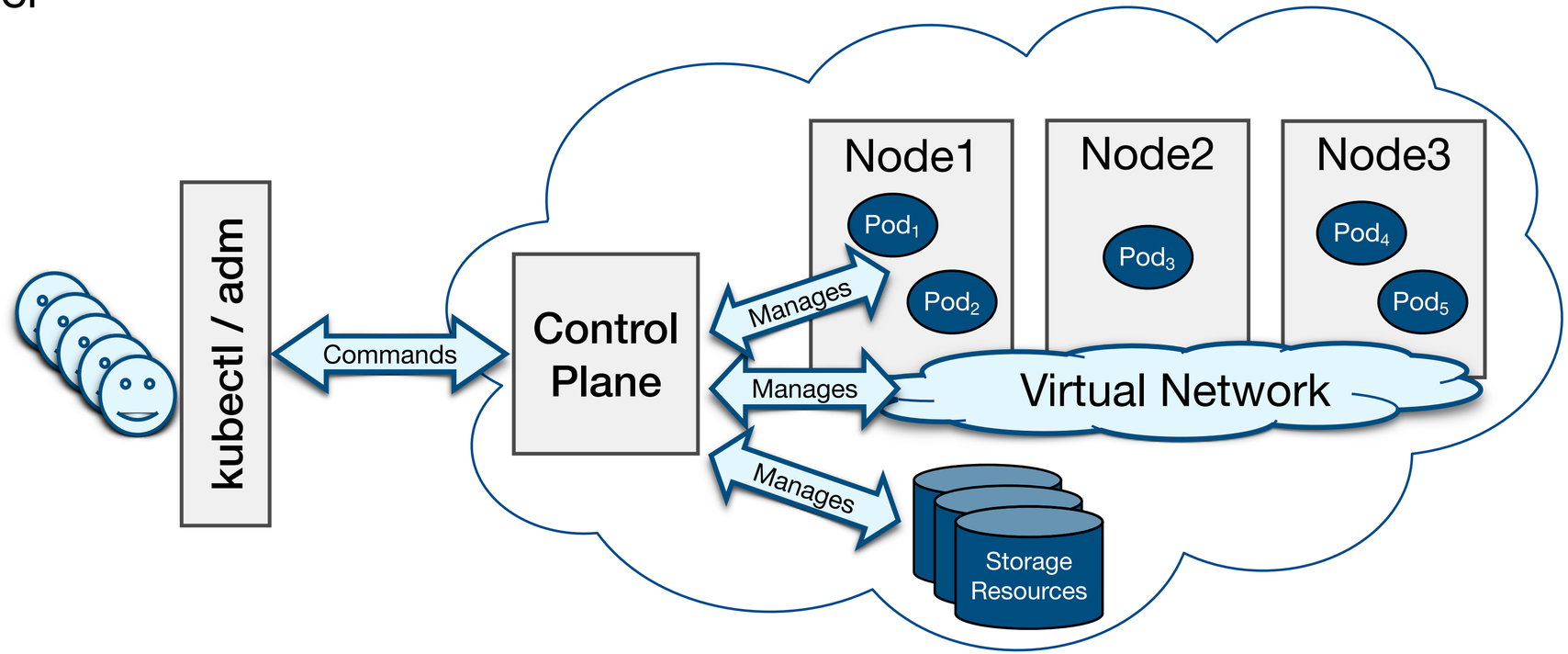
- Container orchestration platform
  - Automates the deployment, scaling, and management of containers
- Interesting Features:
  - Simplified and automated **scheduling**, **self-healing,** and **scaling** of containers
  - Network management (*e.g.*, service discovery, load balancing)
  - Modular storage orchestration (*e.g.*, iSCSI, NFS, Ceph, AWS, GCP)

- https://kubernetes.io

Kubernetes Cluster

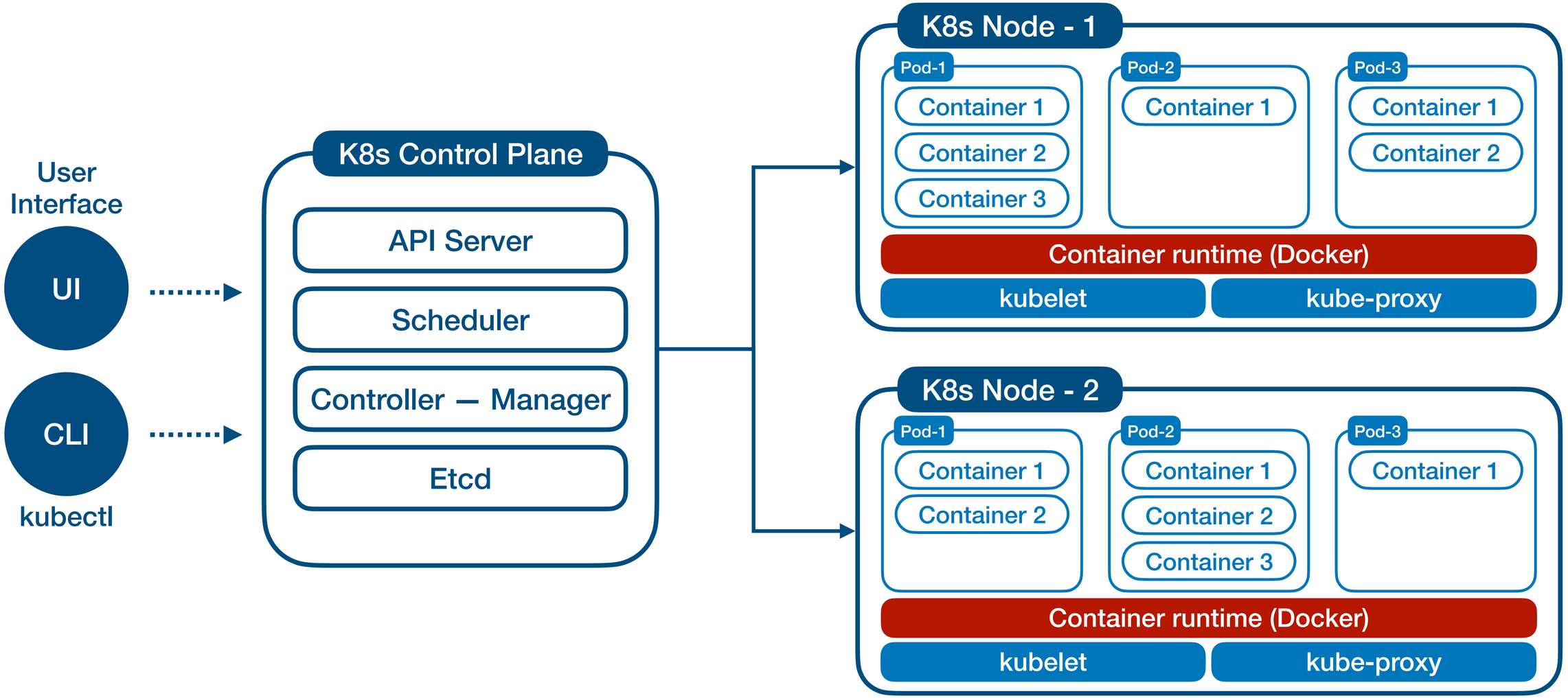
Overview

- A K8s cluster is a group of **a Control Plane** and a **set of Nodes**
  - The **Nodes** host Pods that run containerized applications
  - The **Control Plane** manages the Nodes, Network, Storage and Pod resources in the cluster



Kubernetes Components

More detail



Kubernetes Components

In brief…

- Control Plane Components
  - **API server:** The core component server that exposes the K8s HTTP API
  - **etcd:** Key-value store for cluster configurations and state
  - **Scheduler:** Distributes unscheduled pods across the available nodes
  - **Controller—Manager:** Runs controllers (e.g., for replication, namespaces, …) to implement K8s API behavior
- Node Components
  - **Kubelet:** Manages containers based on incoming Pods specifications
  - **Kube-proxy:** Accepts and controls network connections to node's Pods
  - **Container runtime:** Software responsible for running containers (e.g., Docker)

Kubernetes Cluster

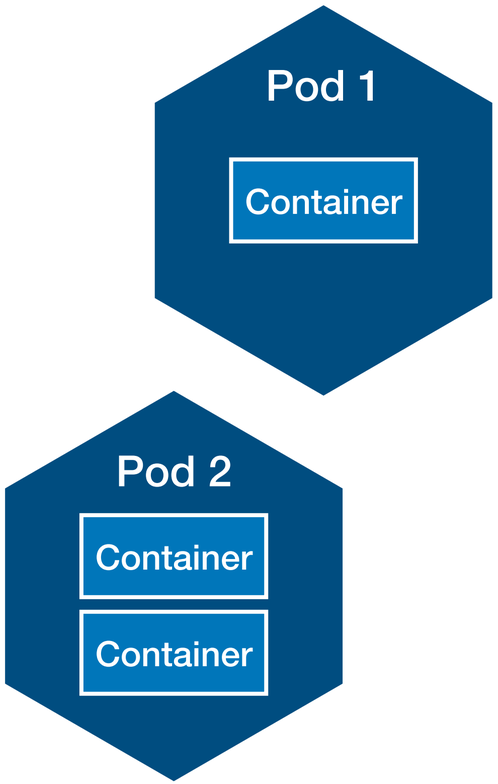
Usage

- An **administrator** uses the **kubeadm** command-line tool to set up a cluster:
  - To initialize a cluster: kubeadm init
  - To destroy a cluster: kubeadm reset
- **Clients** interact with the cluster through the Control Plane using the **kubectl** command-line tool
- Workflow
  1. You submit a configuration file (YAML) describing what you want
  2. The Control Plane validates it and stores the state in etcd
  3. The Scheduler assigns Pods to nodes
  4. The Kubelet ensures those Pods are running
  5. The Controller Manager automatically keeps everything in sync

Kubernetes Workloads

The Pod computing unit

- A **workload** is an application running on Kubernetes
  - Whether it is a single component or several that work together, on Kubernetes you run these inside a set of **Pods**
- A **Pod is** a group of **one or more containers** with shared storage and network resources, and a specification for how to run the containers
  - Smallest deployable unit in Kubernetes

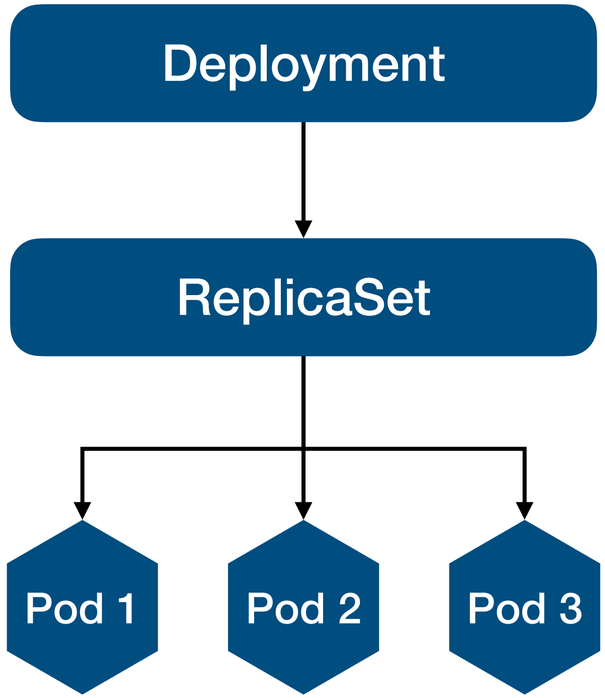


Find more about Pods at: https://kubernetes.io/docs/concepts/workloads/pods/

Kubernetes Workloads Resources

Runtime environment for Pods

- **Deployments** provide a declarative way to manage and scale a set of identical Pods (replicas)
  - These define a desired state for the application (e.g., the desired number of replicas) and manage the lifecycle of the corresponding Pods
- **ReplicaSets** ensure that a specified number of Pod replicas are running at any given time
  - The Deployment is a higher-level concept that automatically manages ReplicaSets.

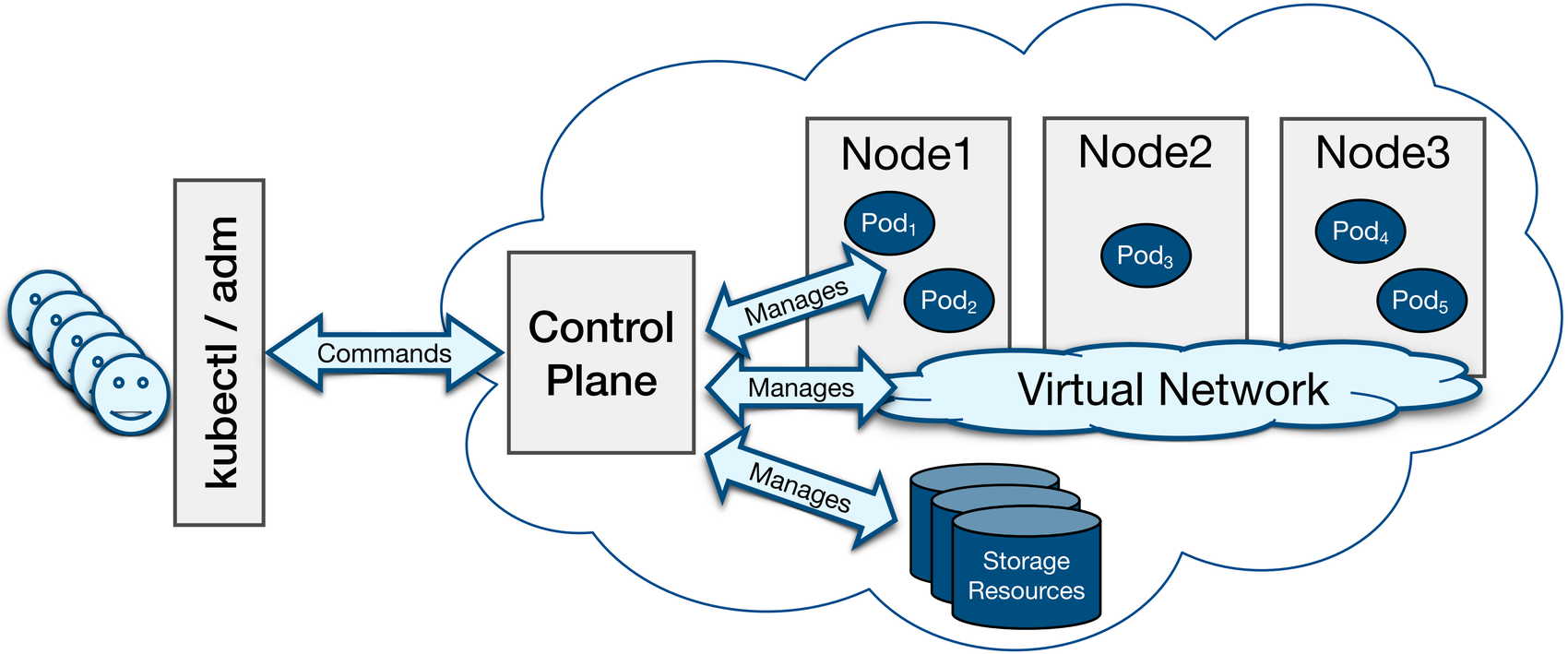


Find more about Workload Resources at: https://kubernetes.io/docs/concepts/workloads/controllers/

Kubernetes Network

Network Overlay

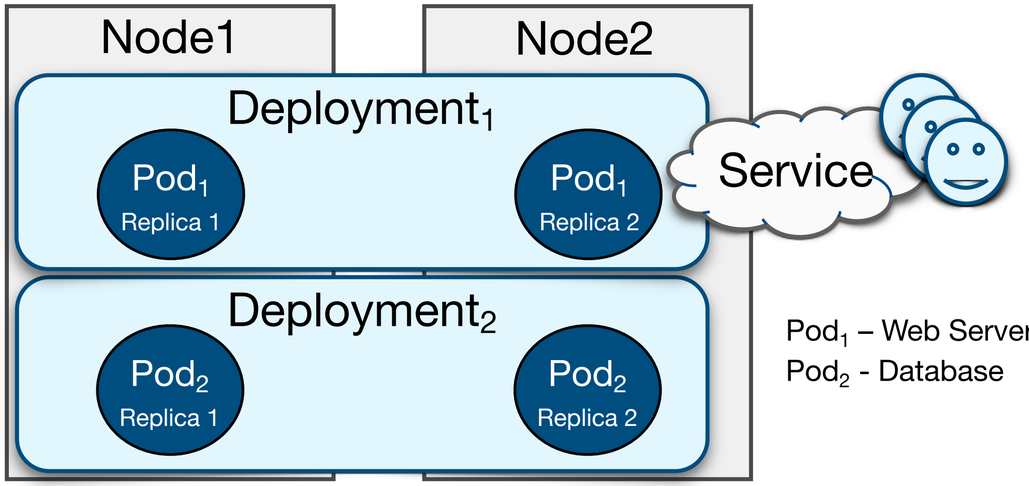
- Each pod has a unique cluster-wide IP address
  - **Networking across pods** (even in distinct worker nodes) is managed through **network overlays** (*e.g.*, Flannel, Calico)



Kubernetes Services

Exposing Pods as a network service

- A **Service** is an abstraction, built on top of the network overlay, for exposing groups of Pods over the network
  - Each Service object defines a logical set of endpoints (e.g., Pods) and a policy about how to make those pods accessible
  - Different types of Service policies:
    - **ClusterIP:** Exposes the Service on a cluster-internal IP
    - **NodePort:** Exposes the Service on each Node's IP at a static port
    - **LoadBalancer:** Exposes the Service externally using an external load balancer

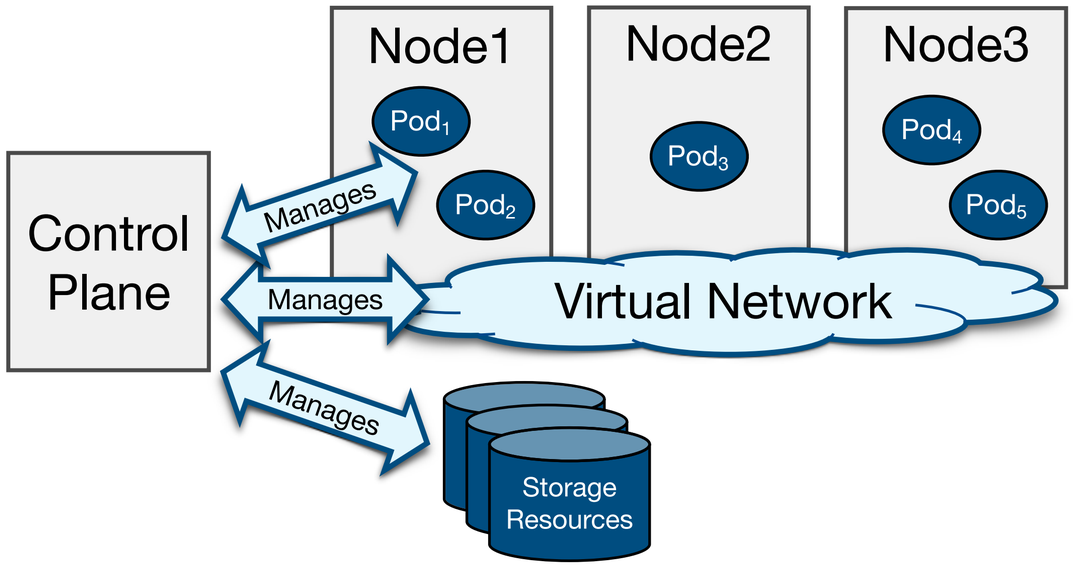


Find more about Services at: https://kubernetes.io/docs/concepts/services-networking/service/

Kubernetes Storage

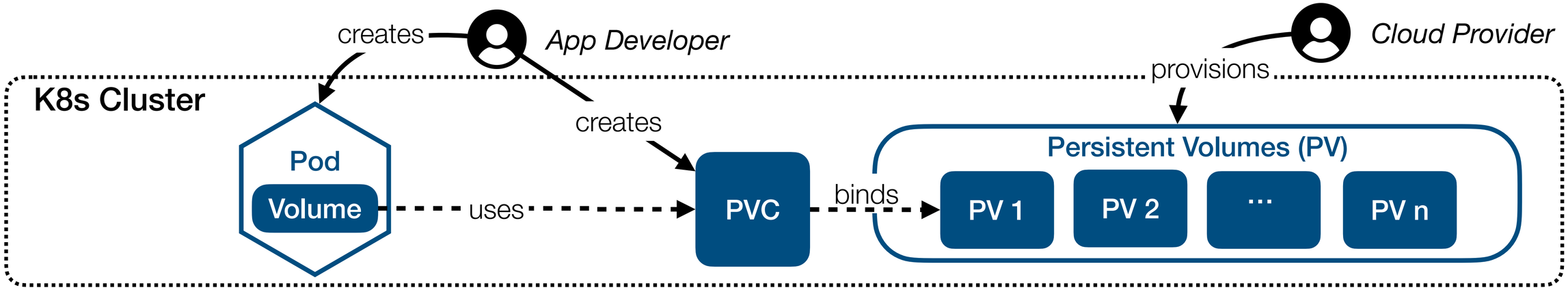
Storage Types

- Pods can access storage volumes provided by different storage backends (*e.g.*, iSCSI, NFS, AWS, GCP)
  - **Ephemeral** storage – available during the pod's lifetime
  - **Persistent** storage – available even if the pod is terminated



Kubernetes Volumes
for Pods

- **PersistentVolume (PV):** A piece of storage in the cluster provisioned by an administrator or dynamically provisioned using **Storage Classes**
- **PersistentVolumeClaim (PVC):** A user request of storage for a Pod
- **Storage Class (SC):** Provides a way for administrators to describe the types of storage they offer (e.g., faster/slower, local/remote storage)



Find more about Persistent Volumes at: https://kubernetes.io/docs/concepts/storage/persistent-volumes/

Kubernetes (level up!)

Enhancing your Pods configurations and security

- **ConfigMaps**
  - A k8s object for storing and updating **non-confidential** Pod configurations in a key-value pair format
    (e.g., environmental variables, command-line arguments)
- **Secrets**
  - A K8s object to safely store **confidential** data from Pods
    (e.g., passwords, tokens, keys)
- Useful for the practical assignment!

Find more at: https://kubernetes.io/docs/concepts/configuration/configmap/
and https://kubernetes.io/docs/concepts/configuration/secret/

Summary

Containers vs VMs – Disclaimer!

Each technology is built with different goals and the best one depends on the targeted use case!

- VMs are useful when **full server (OS) virtualization** is needed
- Containers are useful for managing virtual environments with heterogeneous libraries and/or applications

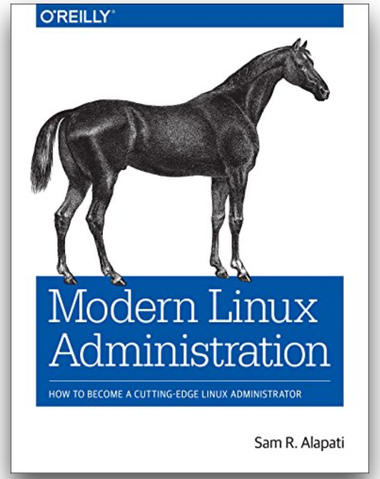
Advantages and Disadvantages

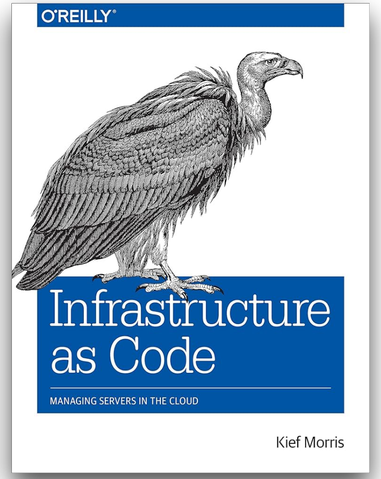
Containers vs VMs

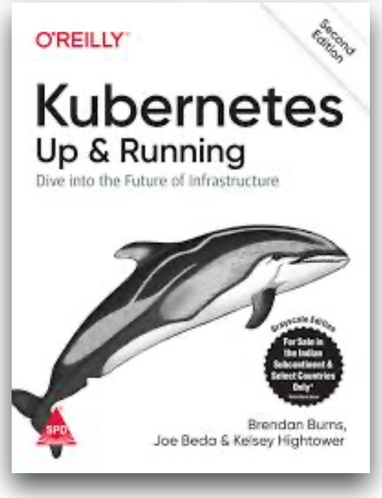
- **Advantages** of Containers over VMs
  - **Faster testing/provisioning/migration** (containers are more lightweight!)
  - **Better resource utilization and performance** (again, more lightweight…)
  - Can easily be **deployed** on both **physical and virtualized servers**
    - Some VM hypervisors provide nested virtualization (i.e., running a VM inside a VM)
- **Disadvantages** of Containers over VMs
  - **Weaker isolation/security** (remember that OS/Kernel are shared)
  - **Less flexibility in running different OSs** (*e.g.*, Linux, Windows, BSD, ...)

Further Reading

- S. Alapati. ***Modern Linux Administration: How to Become a Cutting-edge Linux Administrator***. O'Reilly, 2016
- K. Morris. ***Infrastructure as Code: Managing Servers in the Cloud***. O'Reilly, 2016
- B. Burns. ***Kubernetes Up & Running (Second Edition)***. O'Reilly, 2019.







Questions?