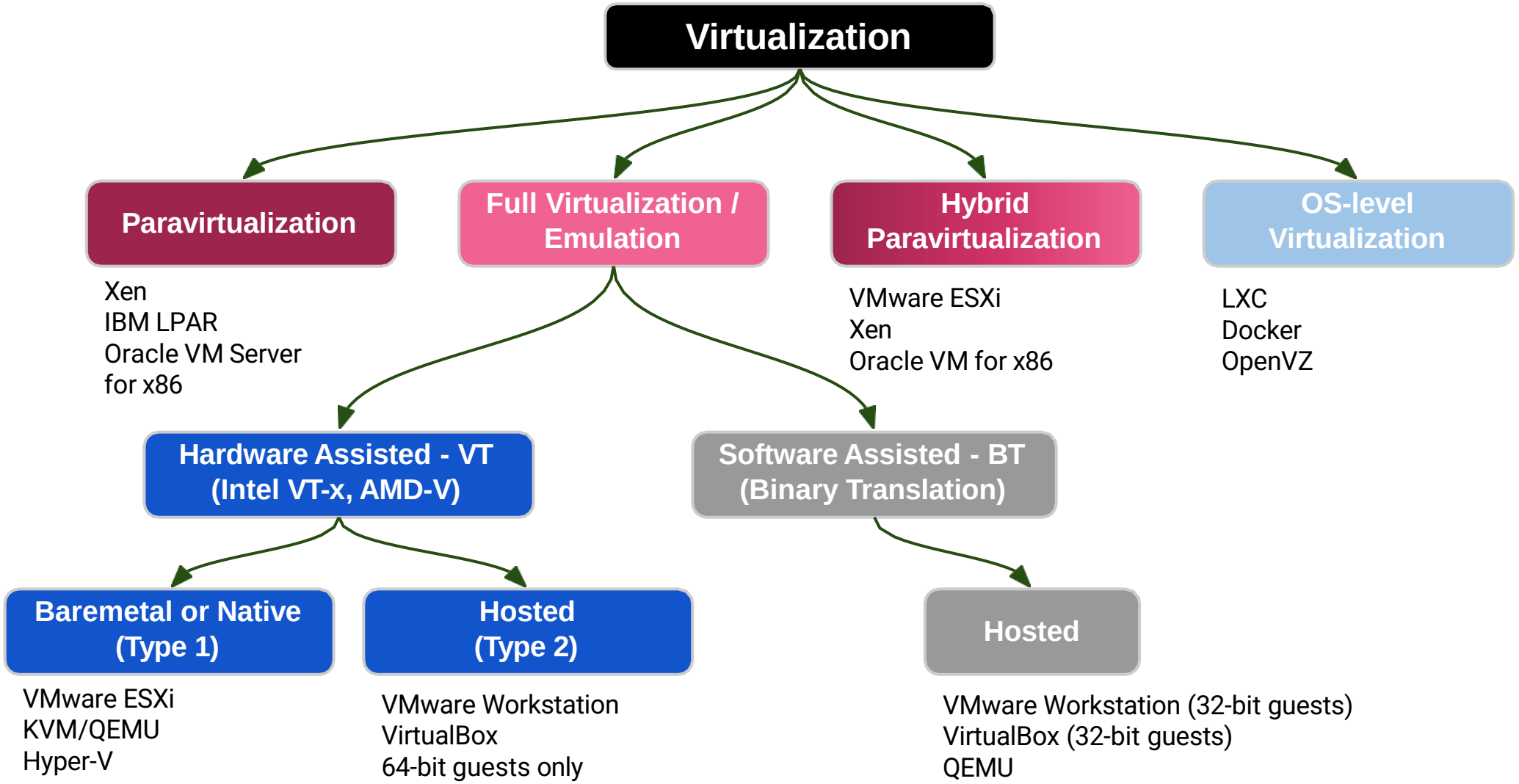
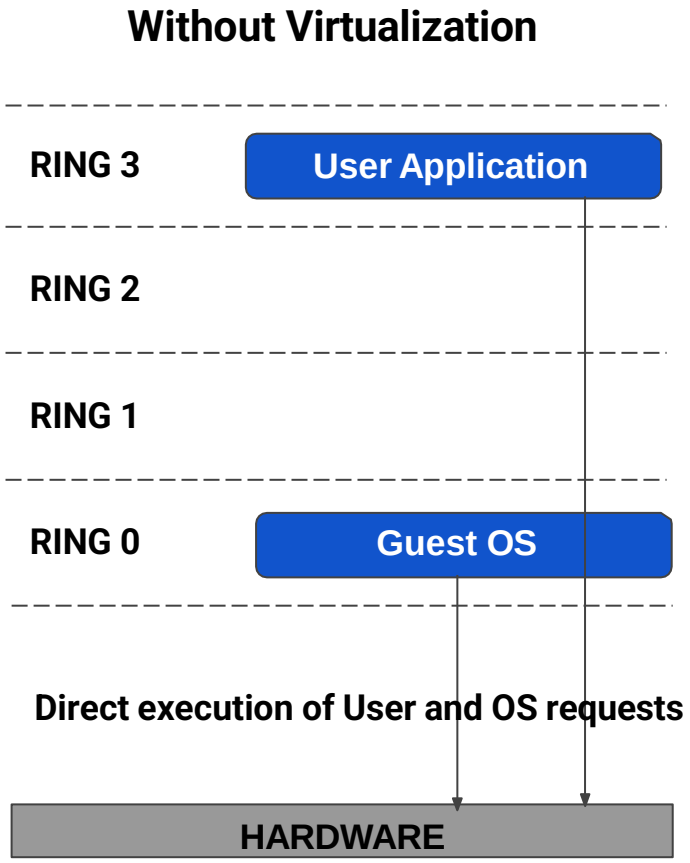
Last Lesson Recaps

Virtualization Landscape



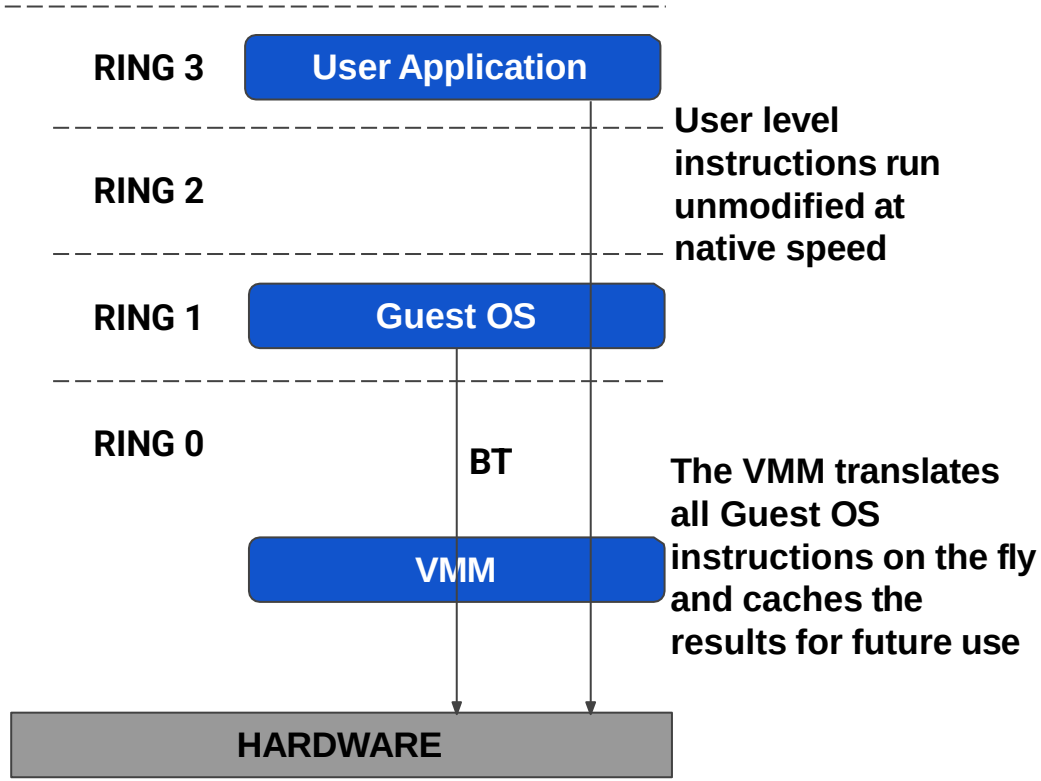
CPU Virtualization Techniques & Protection Ring

The x86 architecture offers **four levels of privilege** known as **Ring 0, 1, 2 and 3** to operating systems and applications to **manage access to the computer hardware**

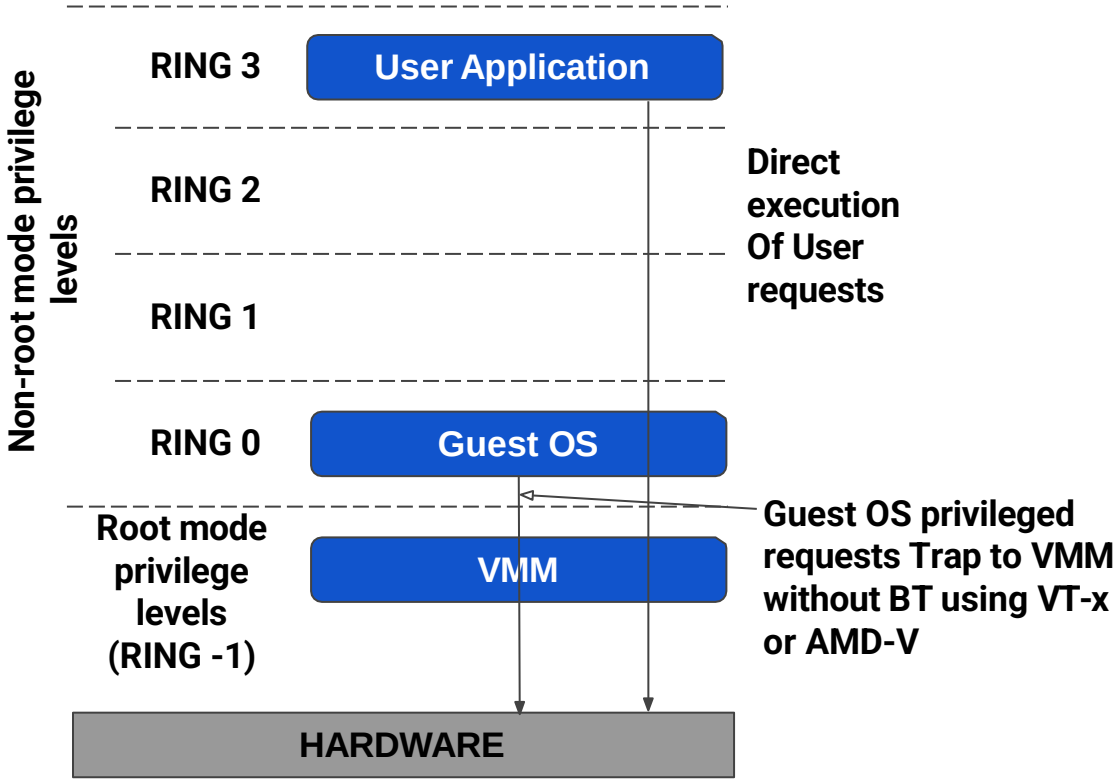


CPU Virtualization Techniques & Protection Ring

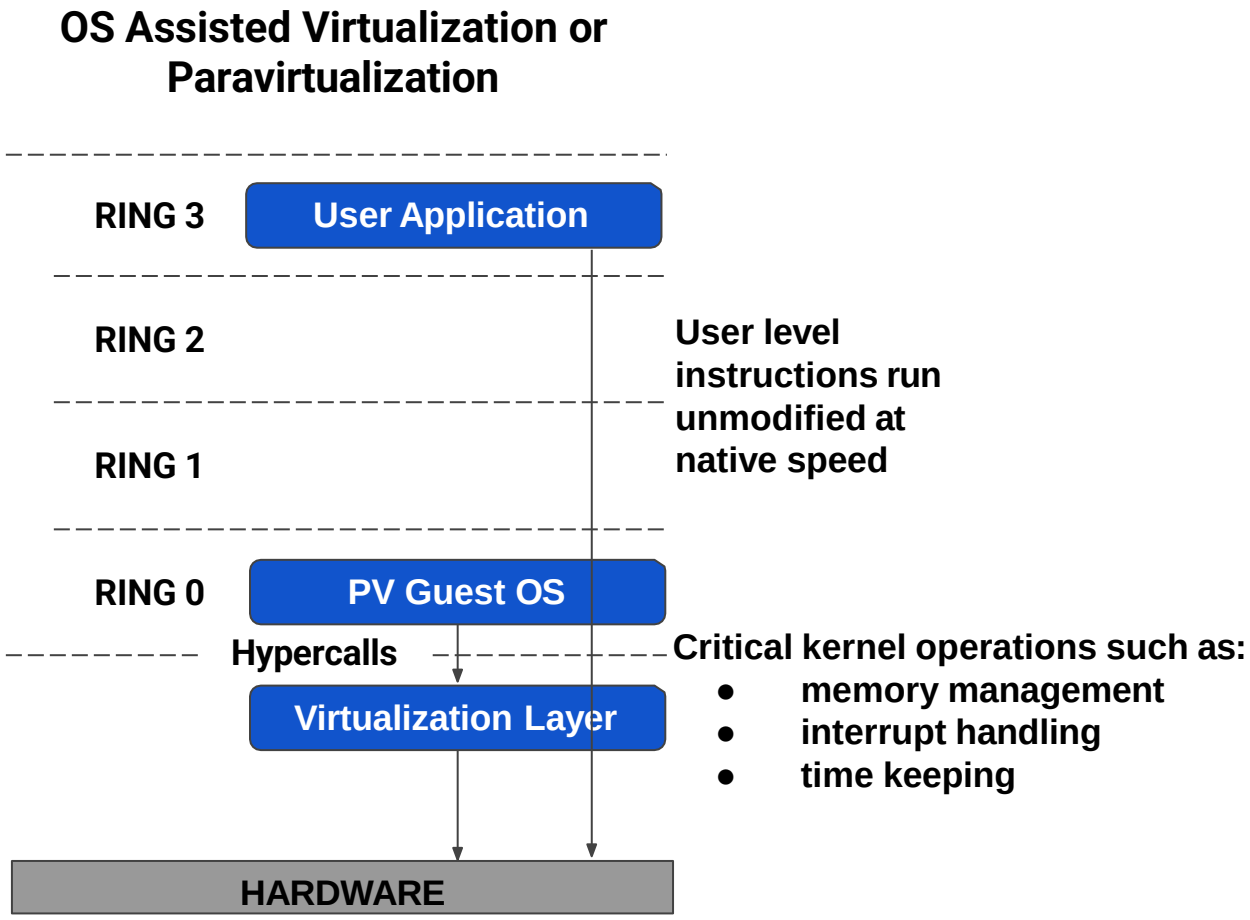
Full Virtualization - Binary Translation (BT)



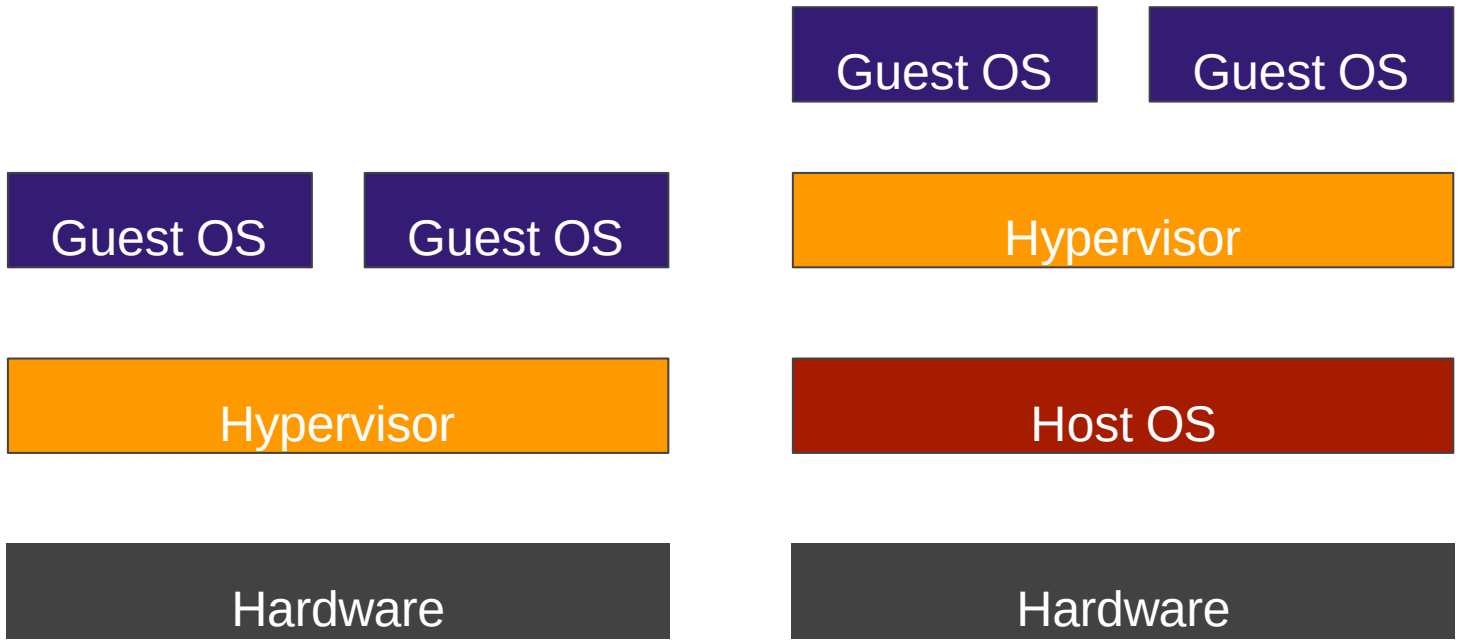
Full Virtualization - Hardware Assisted (VT)



CPU Virtualization Techniques & Protection Ring



How it works ?



TYPE 1 HYPERVISOR

Baremetal

TYPE 2 HYPERVISOR

Hosted

Why call it a “Hypervisor”?

Initially, it was about resource allocation, and the goal was to try to utilize areas of memory that were not normally accessible to programmers. The code produced was successful and was dubbed a hypervisor because, at the time, operating systems were called supervisors, and this code could supersede them.

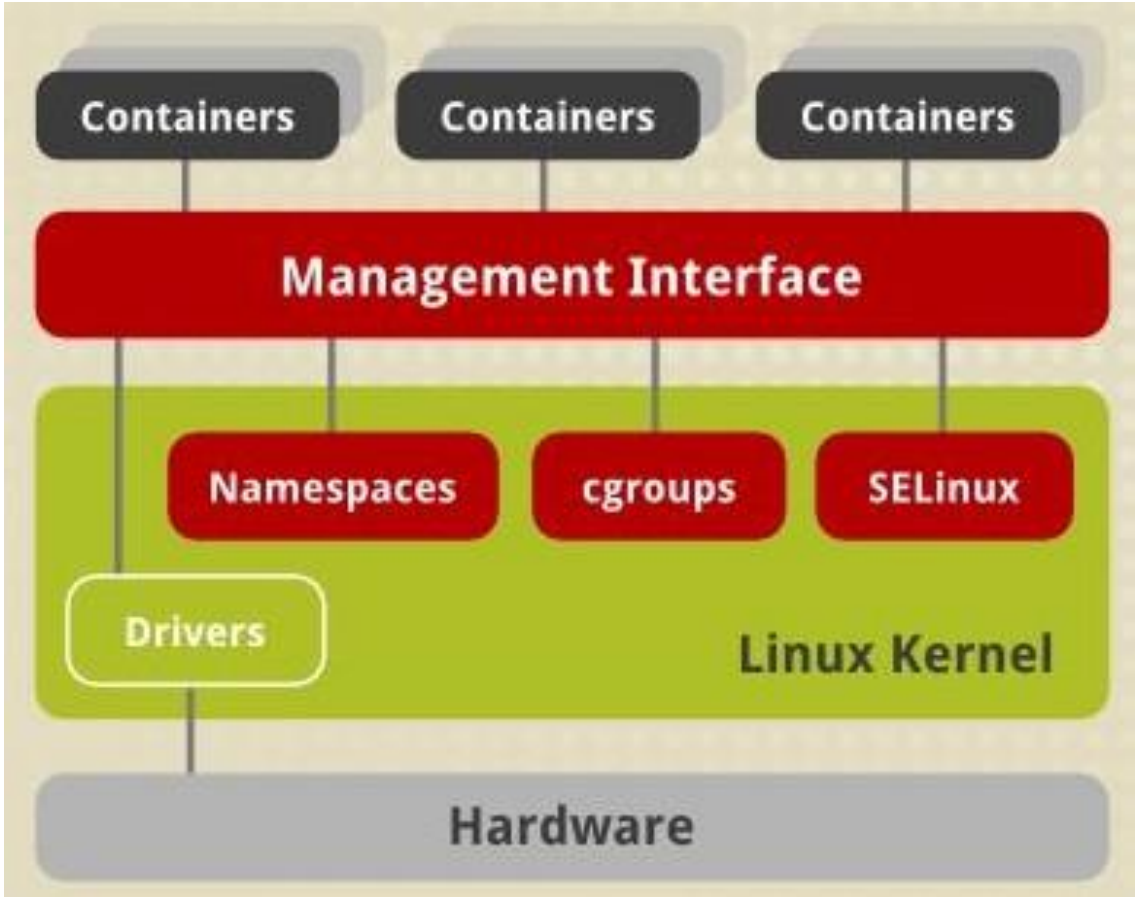
What are containers? [11]

- 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

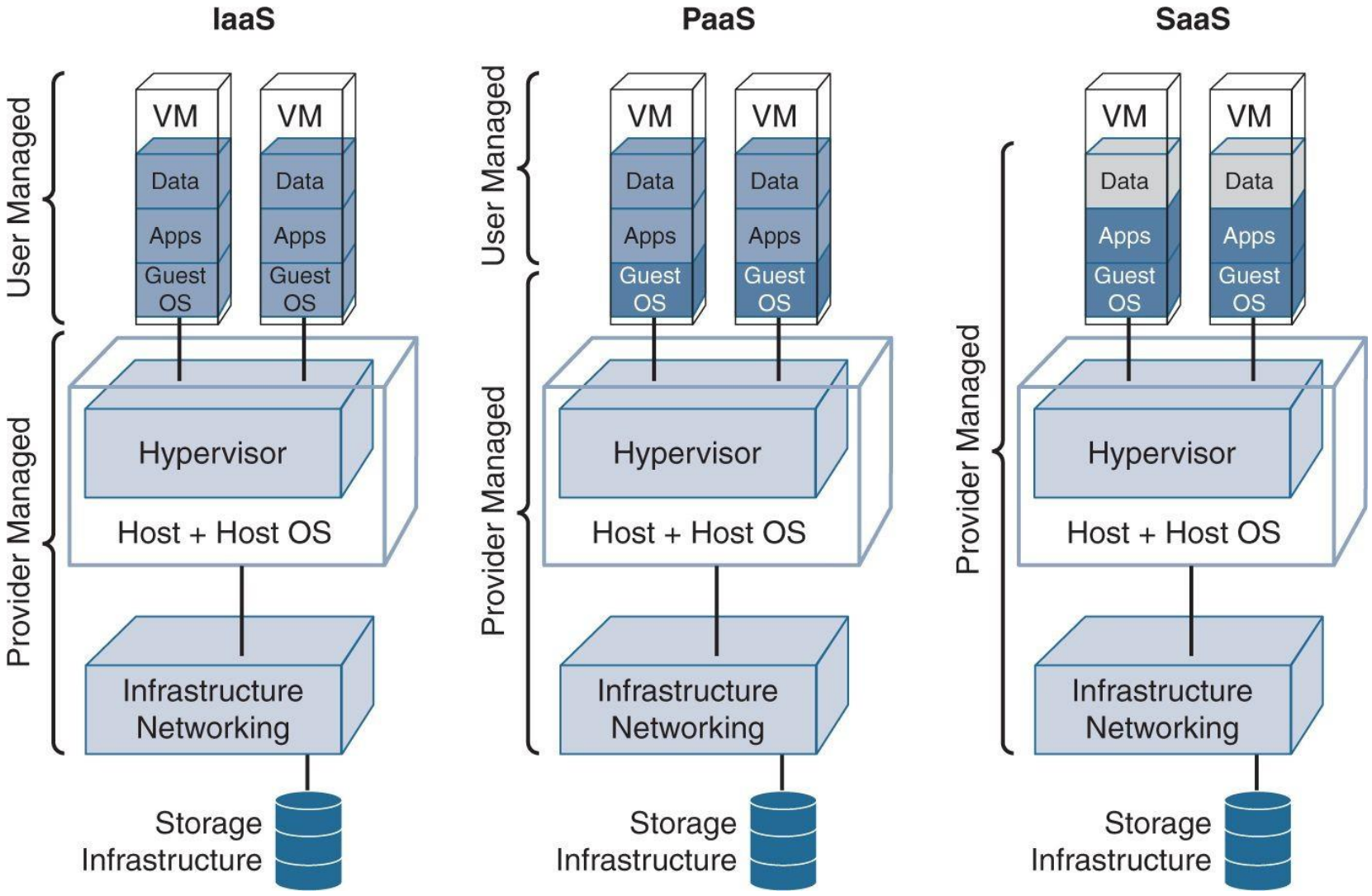
Linux Cgroups, Namespaces, SELinux

- **Namespaces**: abstract a particular global system resource and make it appear as a separated instance to processes within a namespace (e.g. Network, Mount, PID, IPC namespace, etc.)
- **Cgroups**: allows limiting and monitoring CPU time, system memory, network bandwidth of a set of processes
- **SELinux** (Security-Enhanced Linux): enforces container isolation by applying SELinux policy and labels
- Others: Netfilter, netlink, apparmor, etc.

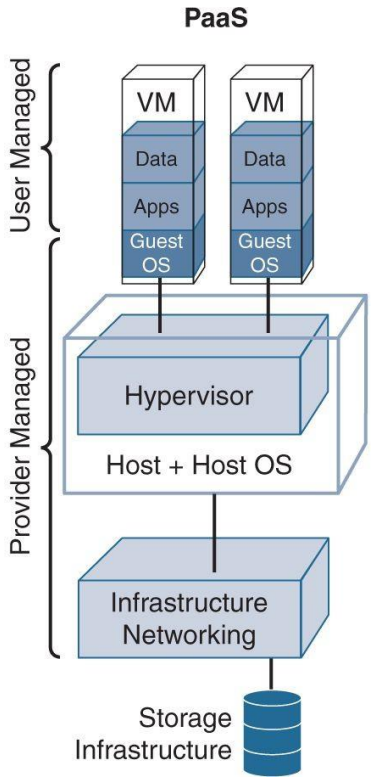


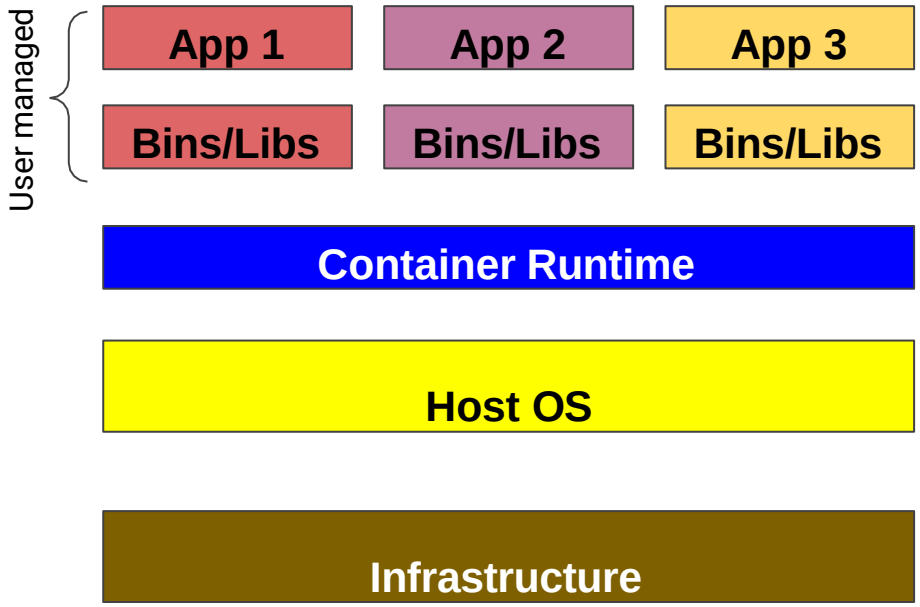
redhat.com

Cloud Computing Models: IaaS, PaaS, SaaS

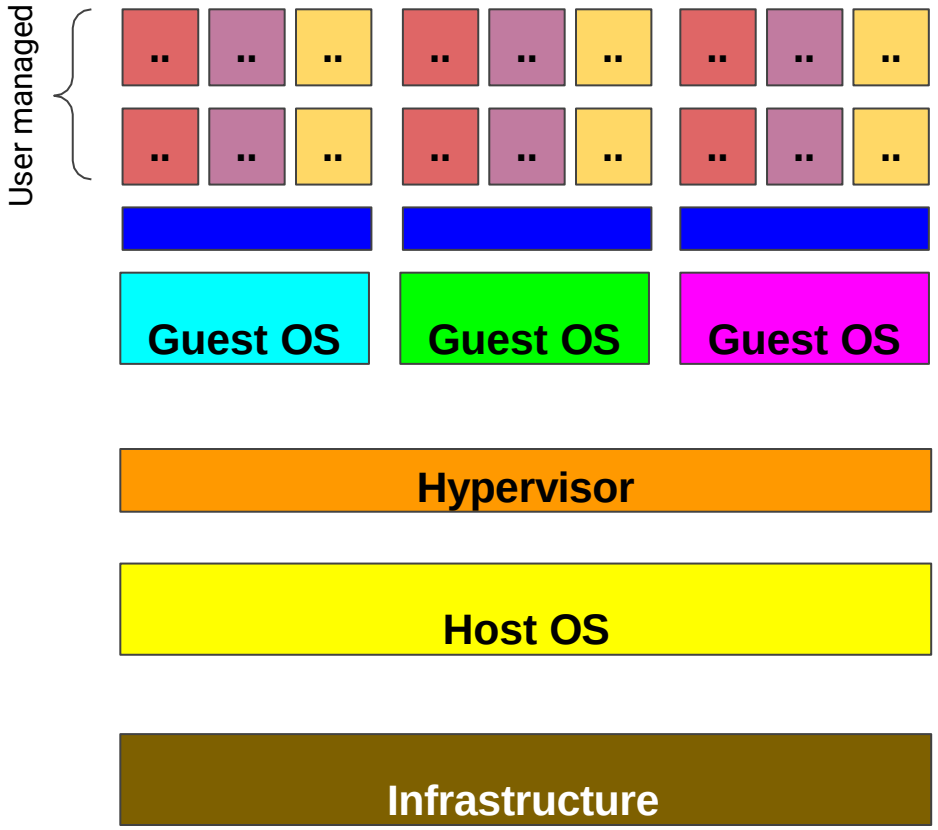


Container-as-a-Service (CaaS)



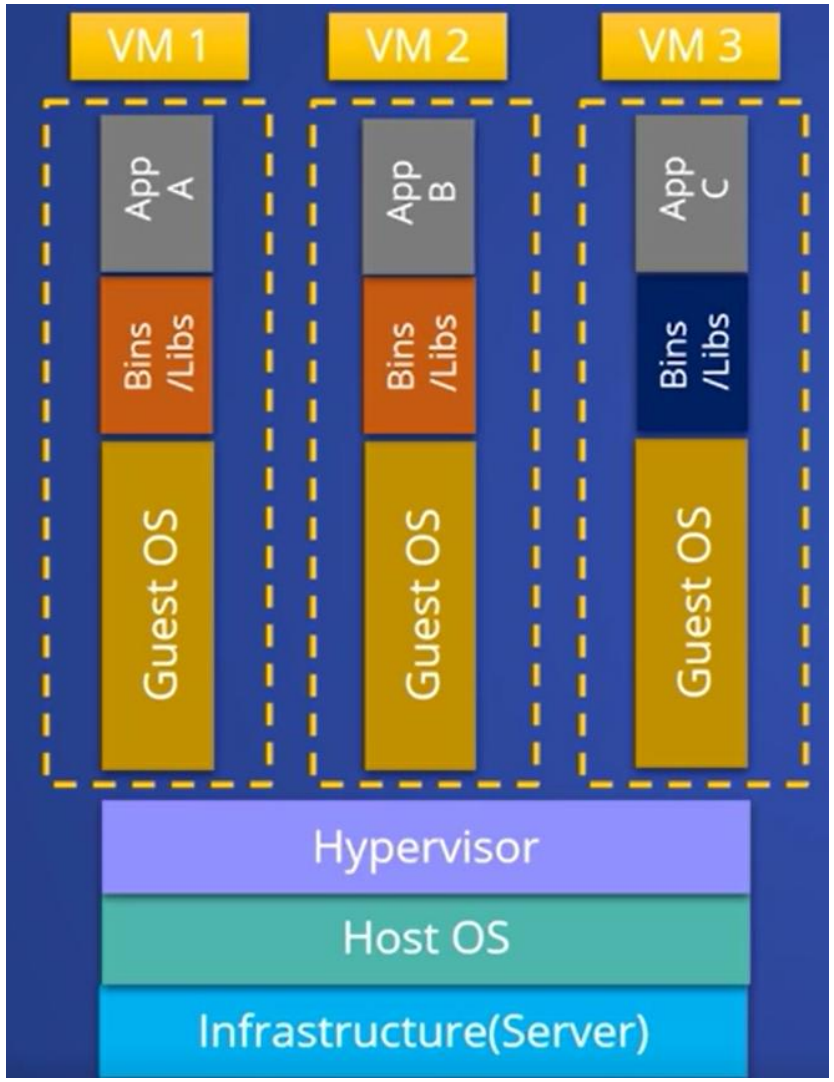


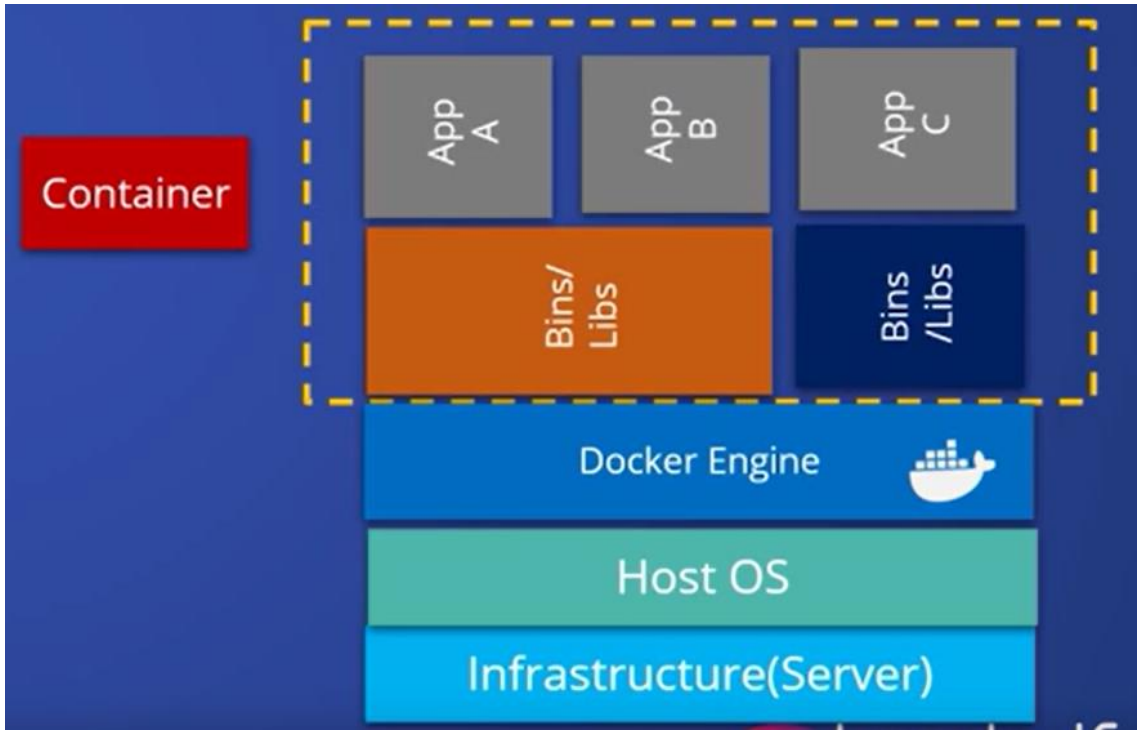
Option 1: Container on Baremetal



Option 2: Container in VM

Container VS VM

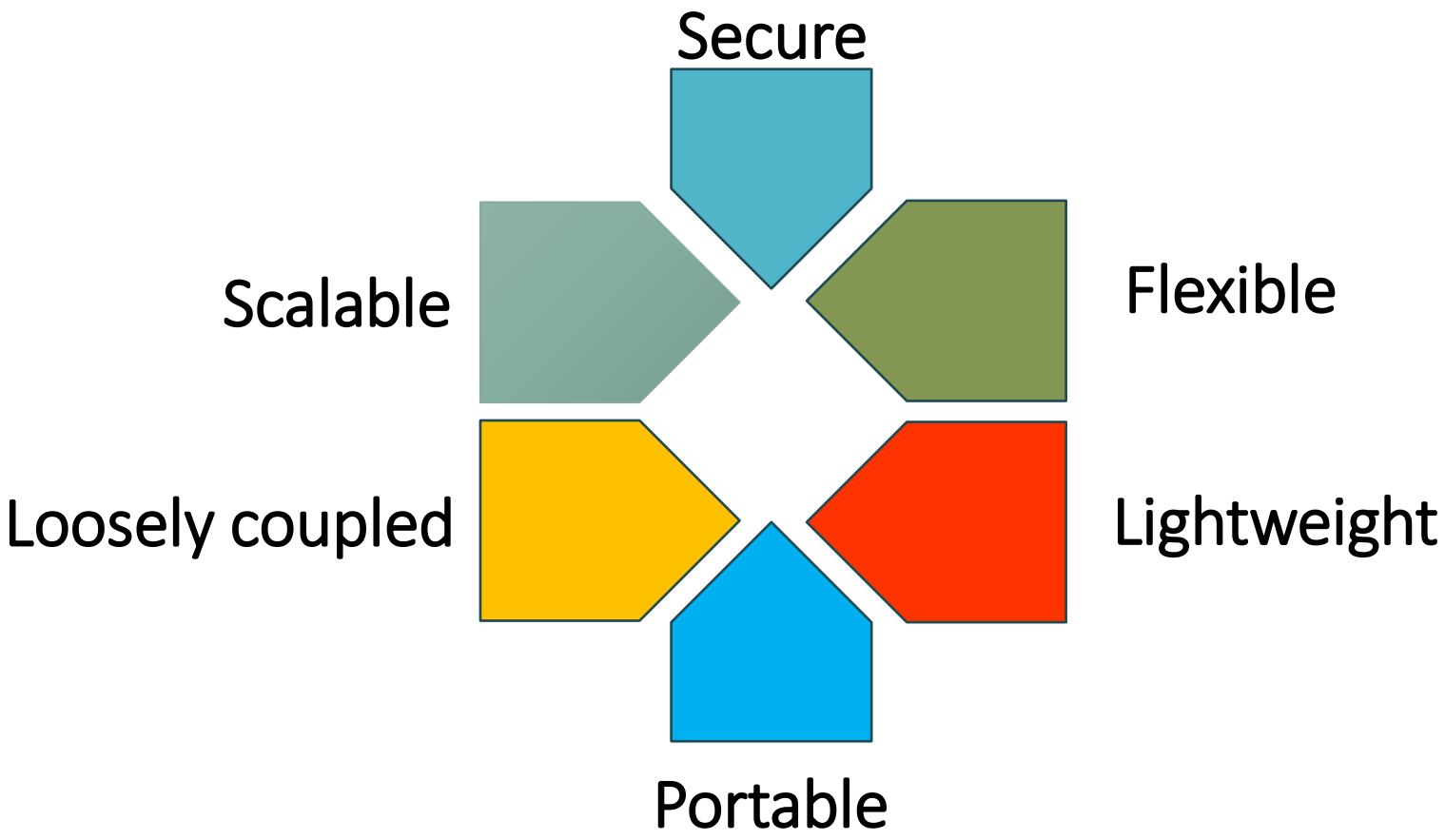




Containerization Benefits

- Greater density
  - several GB in size for a VM vs few dozen of MB for a Container
  - a maximum of 6000 containers can run theoretically on a host machine
- Faster to launch and easy to scale up/down
  - a container is just another process running on the Host OS
- Uses far fewer resources than a VM
  - higher utilization of compute resources than traditional or hardware VMs
- Well adapted for microservice architecture and design pattern
- Lower I/O latency and CPU overhead
- Increased portability
  - application container can run in a predictable/reproducible way on different OSs and environments
- Operational simplicity
  - Container engines provide a very simple yet powerful CLIs to Life Cycle Manage containers (create, start, stop, scale, destroy, attach, etc.)

Containerization Benefits



2.
Virtualisation concepts 2/2

- What are microservices?
- Container runtimes?
- How to deploy containers with docker?

Monolithic vs Microservice Architecture

Monolithic Architecture:

A single, unified application where all components are interconnected and run as a single service.

Benefits:

•Easier to develop and deploy initially.

Disadvantages:

•All components share the same codebase.

•Scaling requires scaling the entire application.

•Changes in one part can affect the whole system.

Monolithic vs Microservice Architecture

Microservices Architecture:

An architectural style that structures an application as a collection of loosely coupled services, each responsible for a specific function.

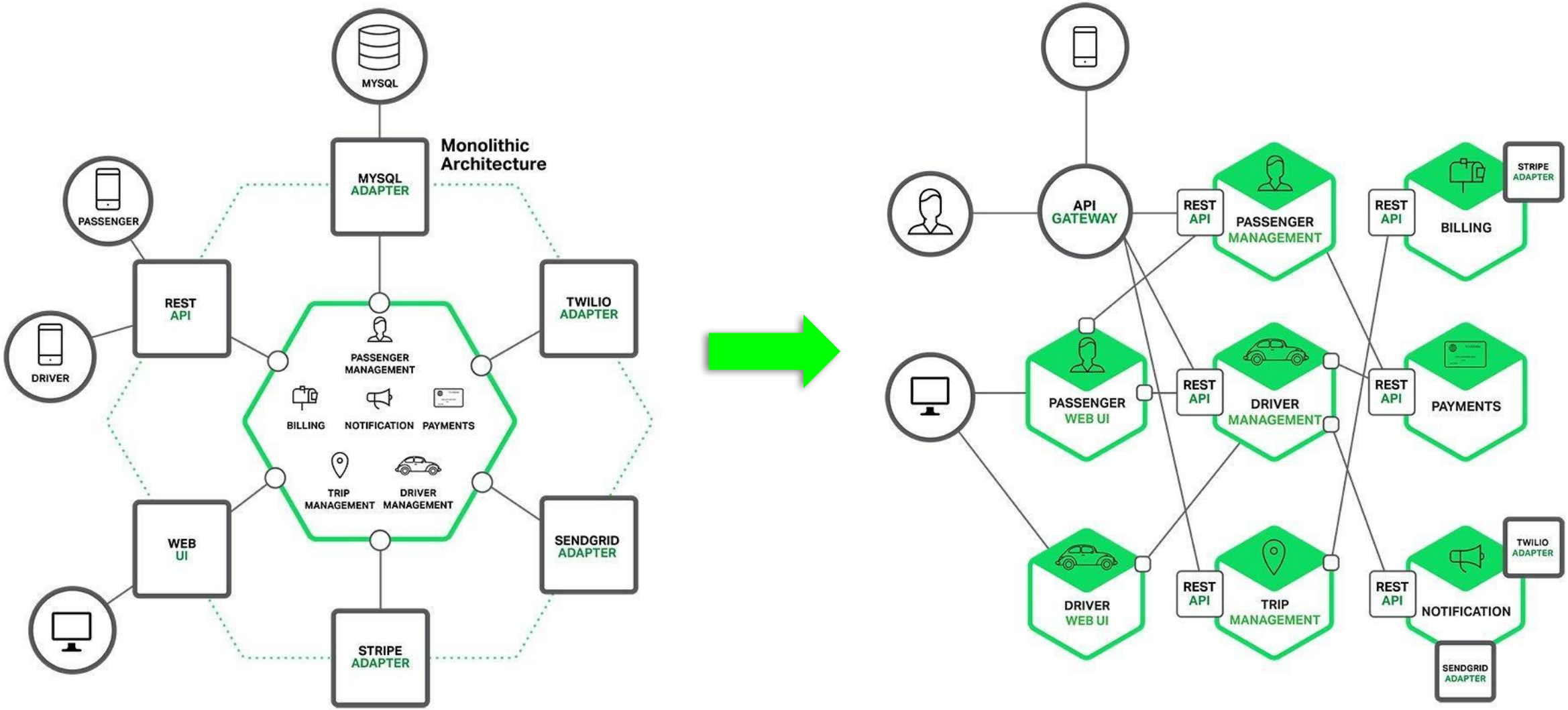
Benefits:

- Each service can be developed, deployed, and scaled independently.
- Technologies and languages can vary between services.
- Better fault isolation; issues in one service do not necessarily affect others.

Disadvantages:

- More complex to manage due to inter-service communication

Microservice Architecture Example [10]



The Twelve-Factor App Methodology [9]

1. **Codebase**: One codebase tracked in revision control, many deployments
2. **Dependencies**: Explicitly declare and isolate dependencies (e.g. requirements.txt, go.mod, etc.)
3. **Config**: Store config in environment variable (e.g. MYSQL_HOST=mysql)
4. **Backing services**: Treat backing services as attached resources
5. **Build, release, run**: Strictly separate build, release and run stages
6. **Processes**: Execute the app as one or more stateless processes
7. **Port binding**: Export services via port binding in a declarative way
8. **Concurrency**: Scale out via the process model by deploying more copies of the app rather giving it more resources and making it larger
9. **Disposability**: Maximize robustness against crashes with fast startup and graceful shutdown
10. **Dev/prod parity**: Keep development, staging, and production as similar as possible
11. **Logs**: Treat logs as event streams. Simply write the events to stdout/err without any concern about how logs are managed.
12. **Admin processes**: Run admin/management tasks as one-off processes. Admin operations/scripts are part of the app codebase

Container Runtimes

- The most widely known runtime is Docker, but it is not alone

The startup that made OS Containers accessible to everyone

Developed by Docker and donated to the CNCF in 2017

Developed by Docker and donated to the Open Container Initiative (OCI) living under the CNCF

Rocket Developed by the former CoreOS (now acquired by RedHat)

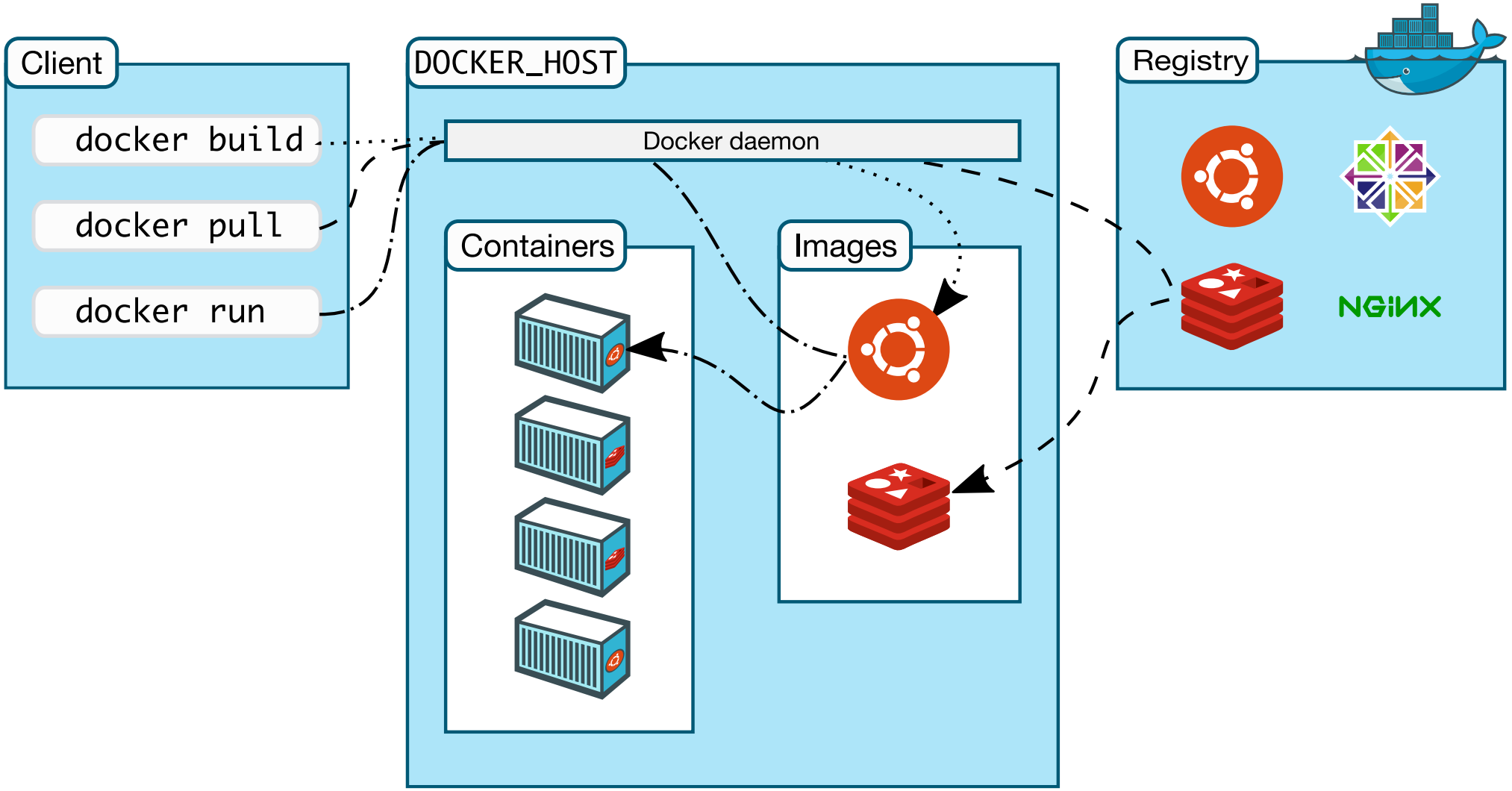
Developed and maintained by the OCI/CNCF

Container Runtimes

A **container runtime** is software that enables the execution and management of containers. It provides the necessary environment for running containerized applications by handling tasks such as:

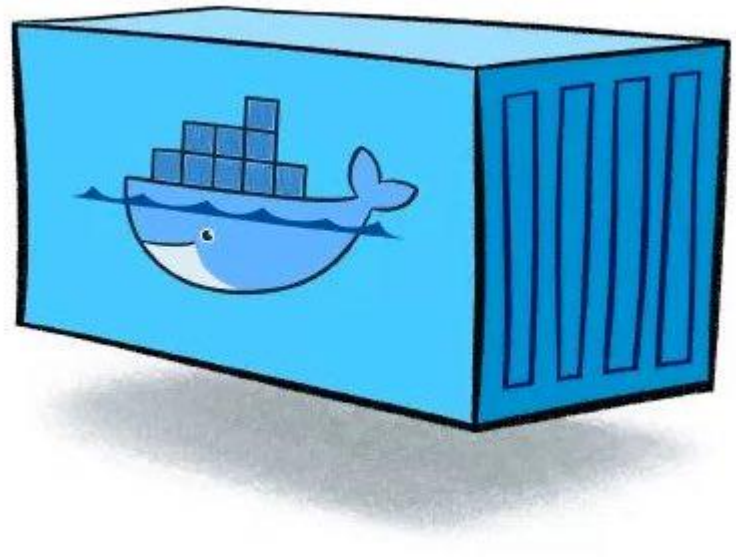
- **Image Management**: Downloading and storing container images.
- **Container Lifecycle Management**: Starting, stopping, and deleting containers.
- **Resource Allocation**: Managing CPU, memory, and storage resources for containers.
- **Networking**: Configuring network settings for containers.

Docker Architecture



Docker Container

A Docker container is a compact, self-sufficient executable package that contains all the necessary components to run an application, including code, runtime, system tools, libraries, and configurations.



Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . .

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Dockerfile

- FROM: Specifies the base image to use. In this case, it's a slim version of Python 3.9.
- WORKDIR: Sets the working directory inside the container. All subsequent commands will be run in this directory.
- COPY: Copies files from the host machine into the container. Here, it copies everything from the current directory to /app.
- RUN: Executes commands in the container. This installs Python packages listed in requirements.txt.
- EXPOSE: Documents the port that the container listens on at runtime. It does not publish the port; you still need to use -p when running the container.
- ENV: Sets an environment variable inside the container.
- CMD: Specifies the command to run when the container starts.

Docker build

```
$ docker build -t my_image:latest .
```

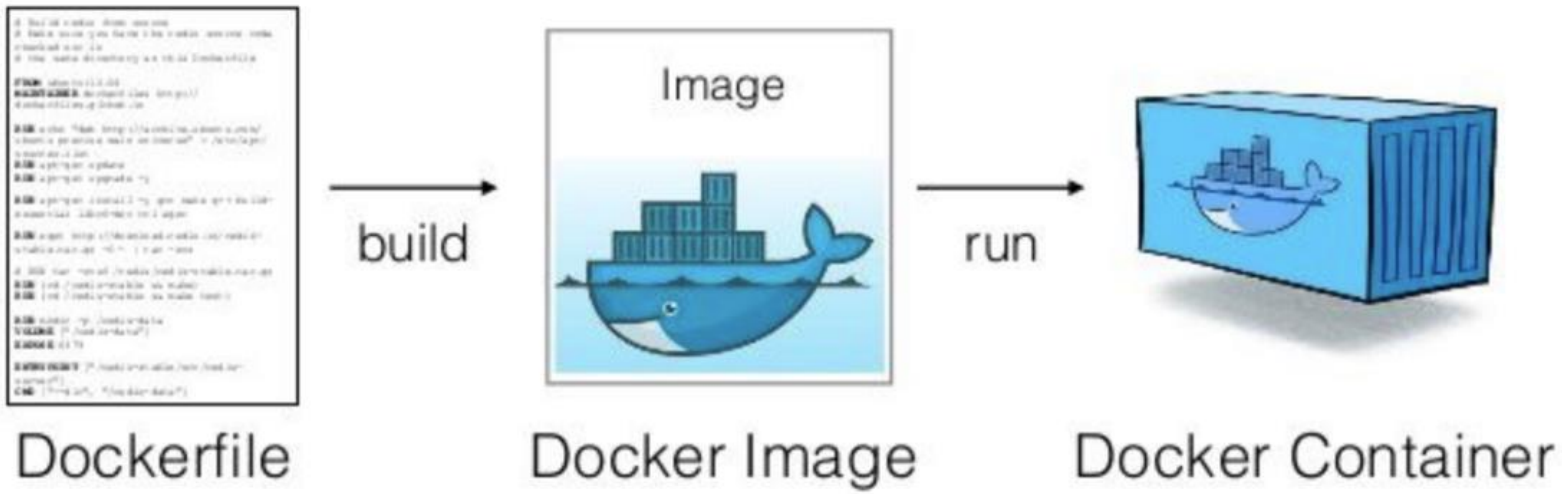
-t: Tag the image with a name (e.g., -t my_image:latest).
-f: Specify a Dockerfile to use (if not named Dockerfile).

Docker run

```
$ docker run -d -p 80:80 --name my_container my_image:latest
```

-d: Run the container in detached mode (in the background).
-p: Map ports from the container to the host (e.g., -p host_port:container_port).
-v: Mount a volume (e.g., -v volume_name:/path/in/container).
--name: Assign a name to the container.
-e: Set environment variables.

Docker image



Docker image storage

- Locally
- Remotely: Registry such as Docker hub

Docker pull/push

```
$ docker pull ubuntu:latest
```

```
$ docker push myusername/my_image:latest
```

This command pushes the my_image with the tag latest to the Docker Hub repository associated with the user myusername.

Dockerfile with an image

dockerfile

```
# Use the official Nginx image as a base
FROM nginx:latest

# Copy custom configuration file (if needed)
COPY nginx.conf /etc/nginx/nginx.conf

# Copy static website files to the Nginx HTML directory
COPY html /usr/share/nginx/html

# Expose port 80
EXPOSE 80
```

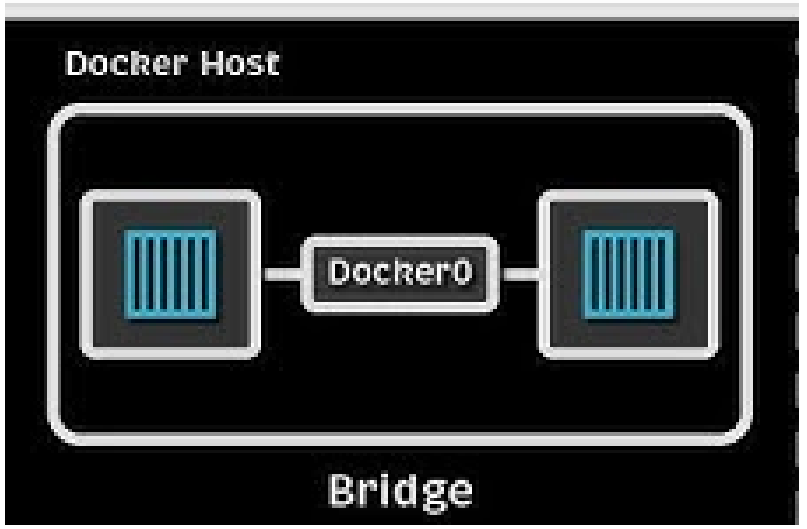
Docker Network

A Docker network is a virtual network that allows containers to communicate with each other and with external systems. It provides isolation and enables different networking modes, such as bridge, host, and overlay, to facilitate container connectivity and management.

```
$ docker network ls
```

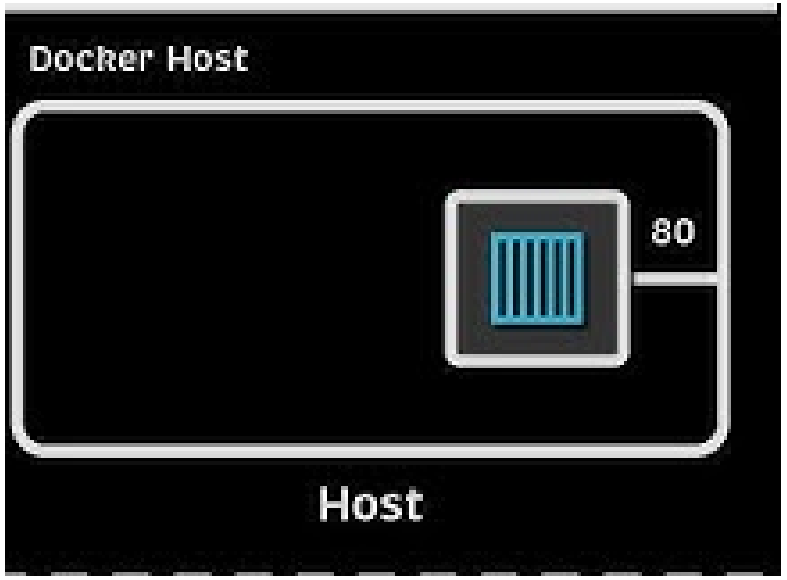
Docker Network: bridge

- Connects containers to each other and to the host system
- It creates a private internal network, allowing containers to communicate using their IP addresses while isolating them from the external network.
- By default, we have docker0 created for the bridge.
- We can create customized bridges



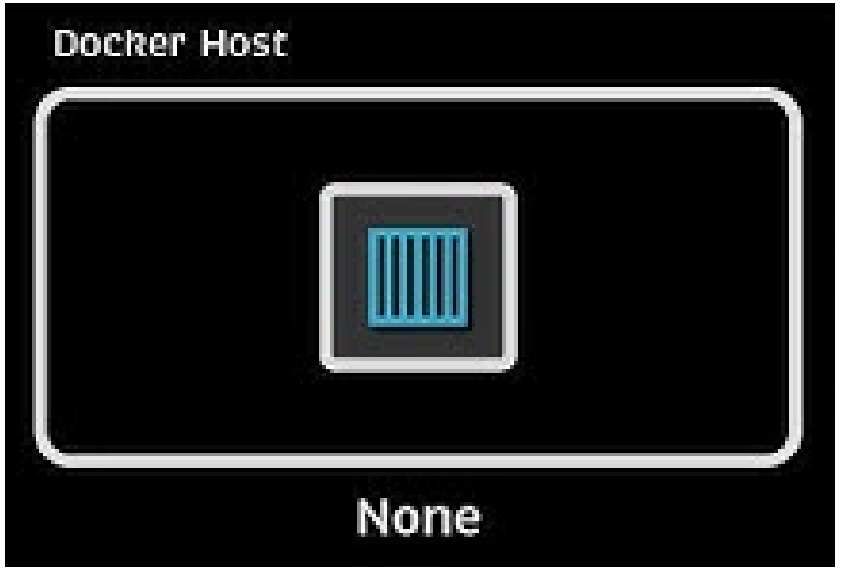

```
$ docker network inspect bridge
$ docker network create --driver bridge <network_name> % to create new bridge
$ docker run -d -p 80:80 --name my_container nginx  %by default the docker0 bridge will be considered
```

Docker Network: host



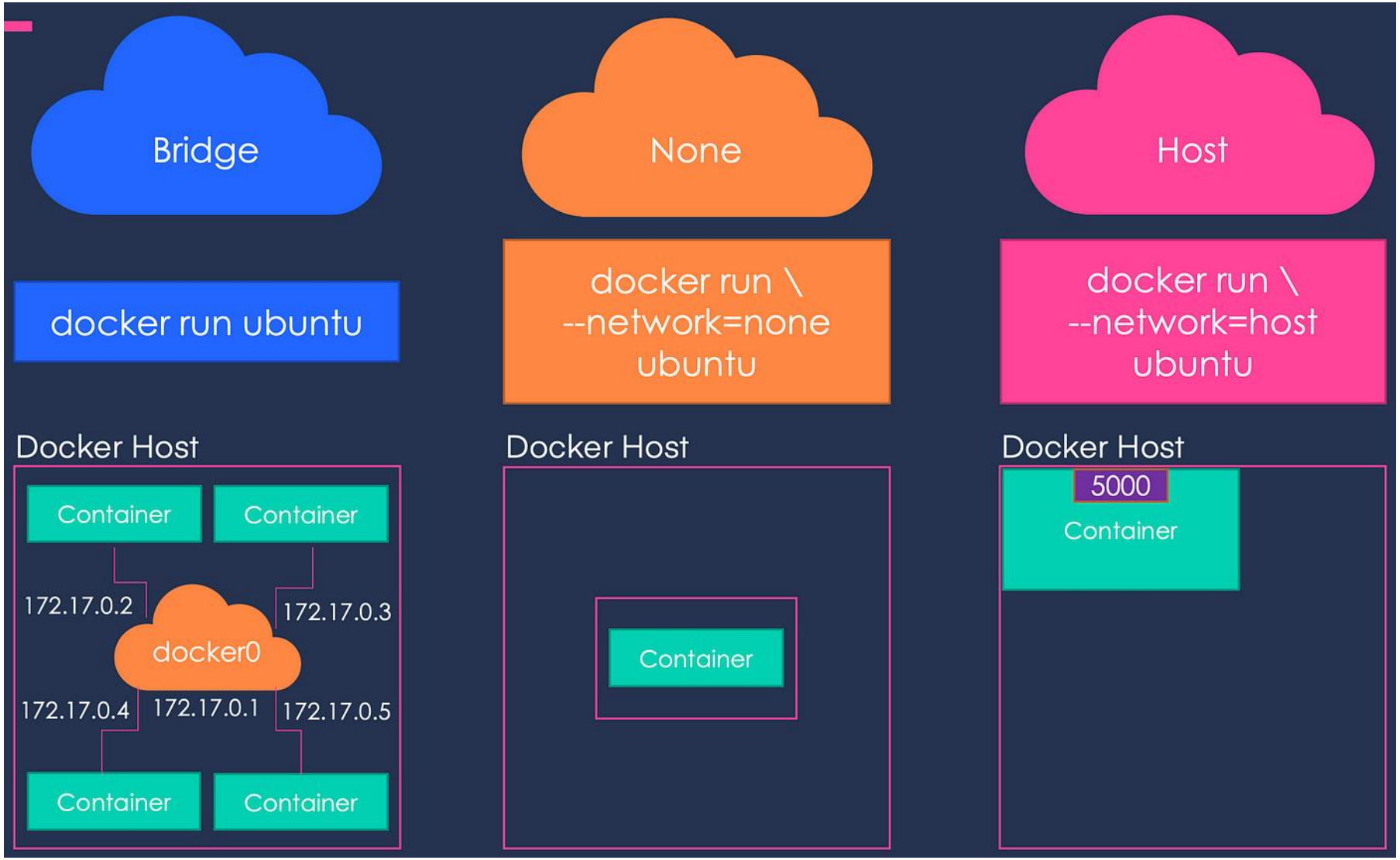

```
$ docker run --network=host
```

Docker Network: None

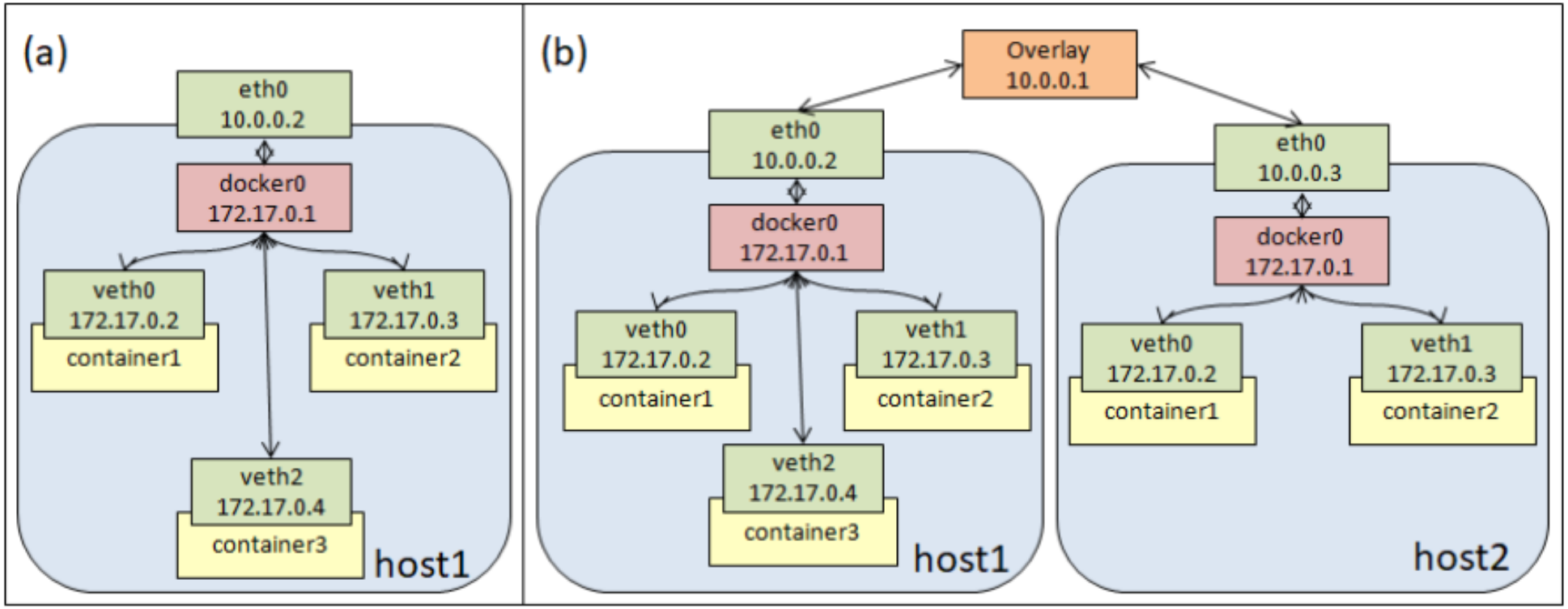



```
$ docker run –network=none
```

Docker Network



Docker Network: use case of two containers



Container Network Interface (CNI) & Network Plugins

- CNI is a set of specification and libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins
- CNI handles network connectivity of containers and removes allocated resources when the container is deleted

Full list: https://github.com/containernetworking/cni

Docker data volumes

- A Docker volume is a persistent storage mechanism used to store data generated and used by Docker containers
- Volumes can be shared between dockers

```
$ docker volume ls
$ docker volume create <volume_name>
$ docker run -v <volume_name>:/path/in/container --name <new_container_name> <image_name>
```

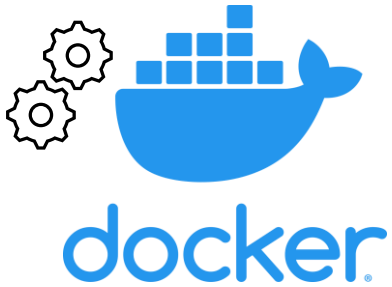
Docker Engine

Docker Engine is the core software that enables the creation, management, and running of containers.

Components:

Client: The command-line interface (CLI) used to interact with Docker.
Server: The daemon that manages Docker containers and images.
REST API: Allows communication between the client and the server.

Docker Engine

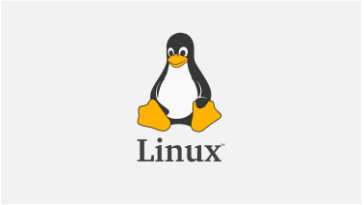


Docker Engine

The administrator configures the desired state

Docker engine changes settings and conditions automatically to keep the desired state

Docker Engine OS



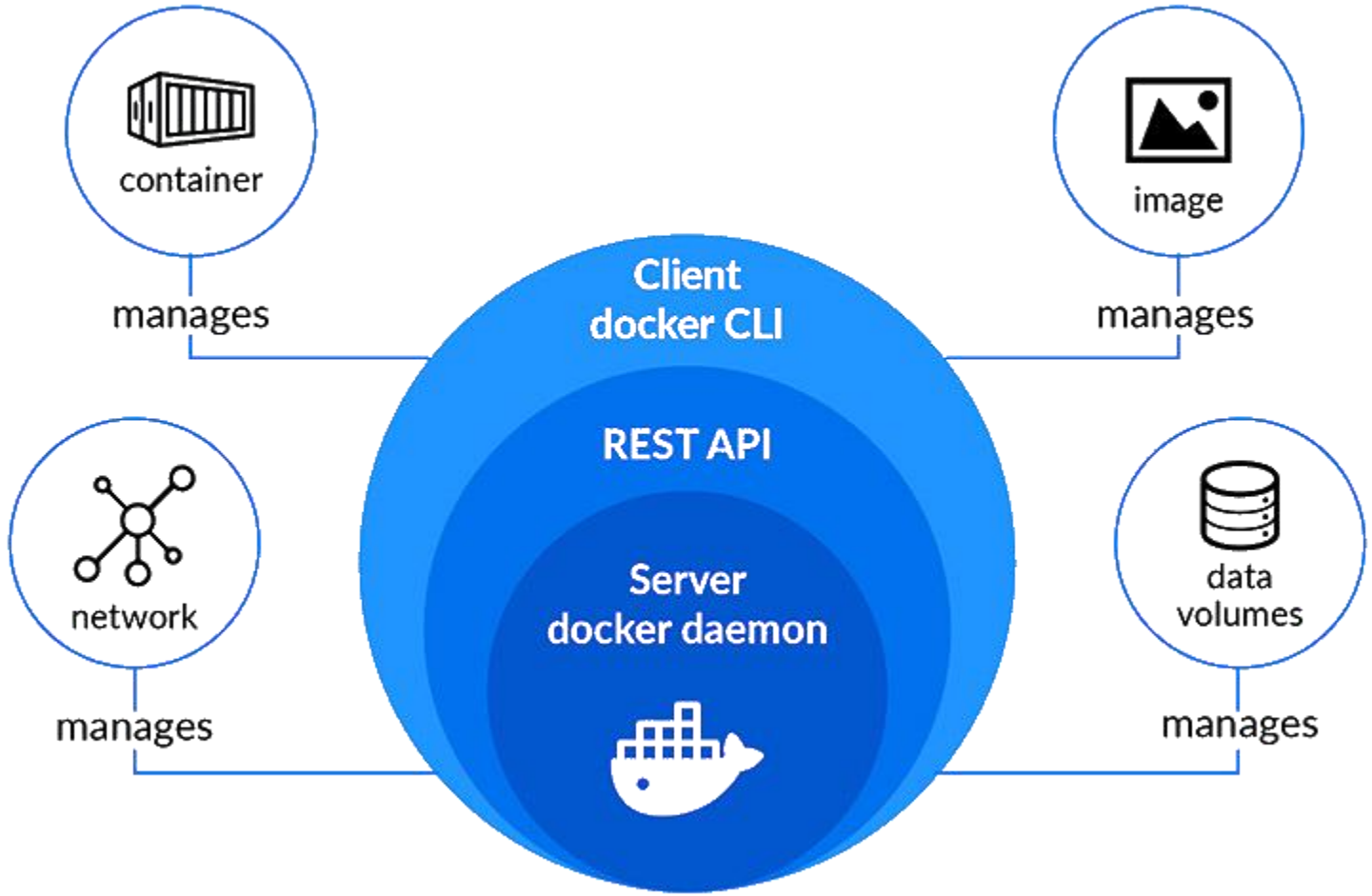
https://docs.docker.com/engine/install/

Docker Engine vs Docker Desktop

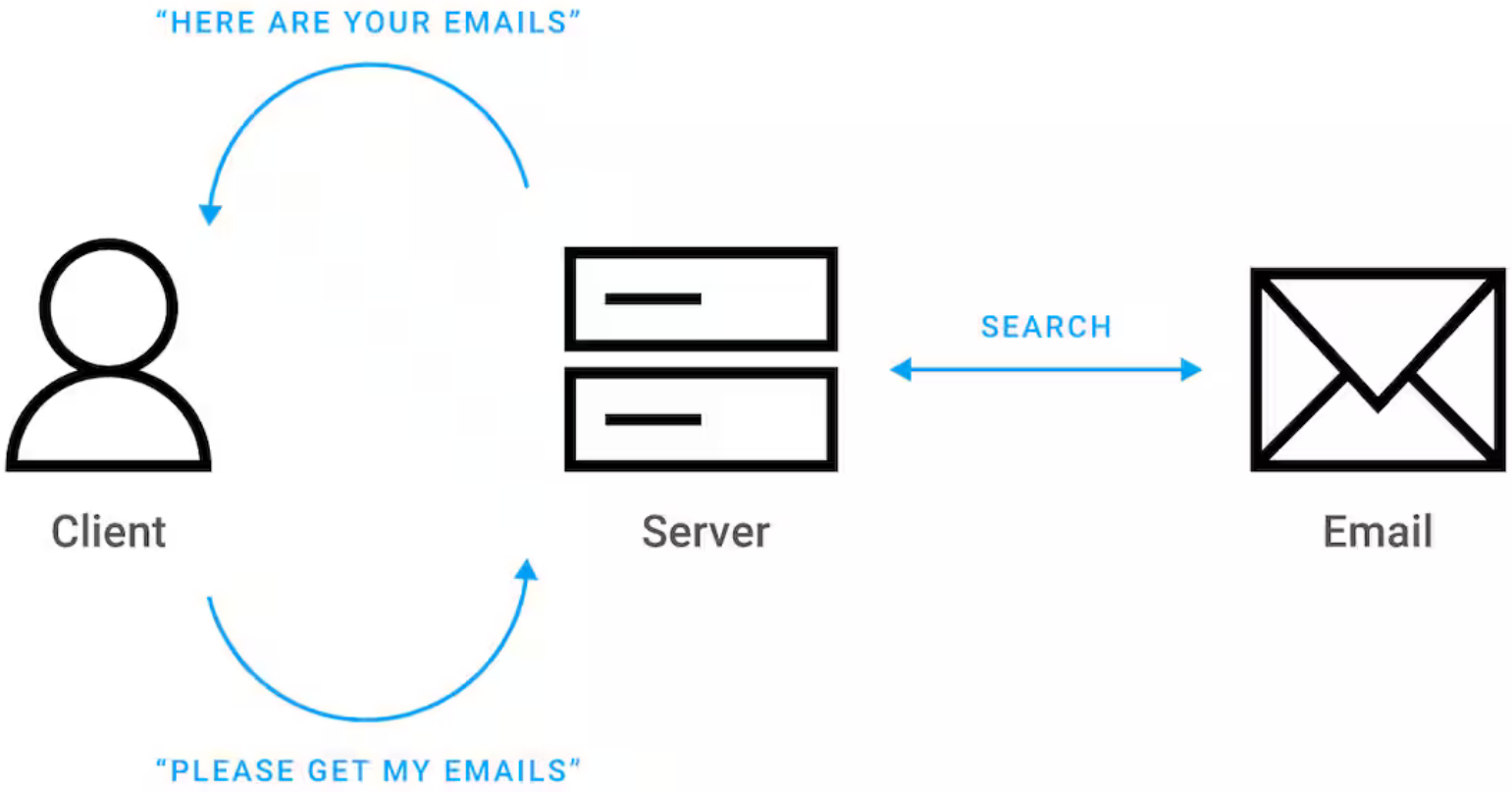
Feature	Docker Desktop for Linux	Docker Engine for Linux
Purpose	GUI for managing Docker	Core container management
Interface	User-friendly GUI	Command-line interface
Installation	Bundled with GUI tools	Installed via package manager
Kubernetes Support	Integrated	Requires separate installation

Note that Docker Desktop was developed to enable MAC and Windows Docker installation

Docker Engine



Client-Server Model



Docker CLI

The Docker Command Line Interface (CLI) is a command-line tool that allows users to interact with the Docker Engine. It provides commands for managing containers, images, networks, and volumes.

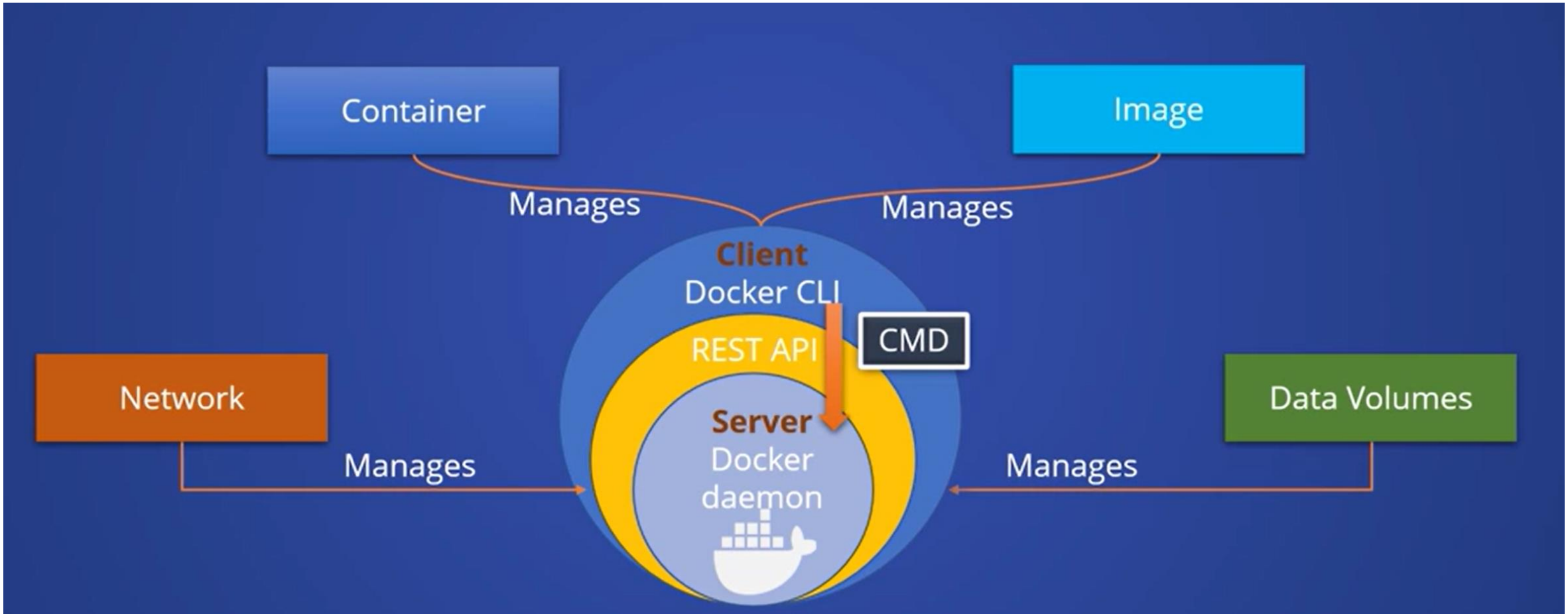
Features:

Container Management: Create, start, stop, and remove containers.
Image Management: Pull, build, tag, and delete Docker images.
Network Management: Create and manage networks for container communication.
Volume Management: Create and manage data volumes for persistent storage.

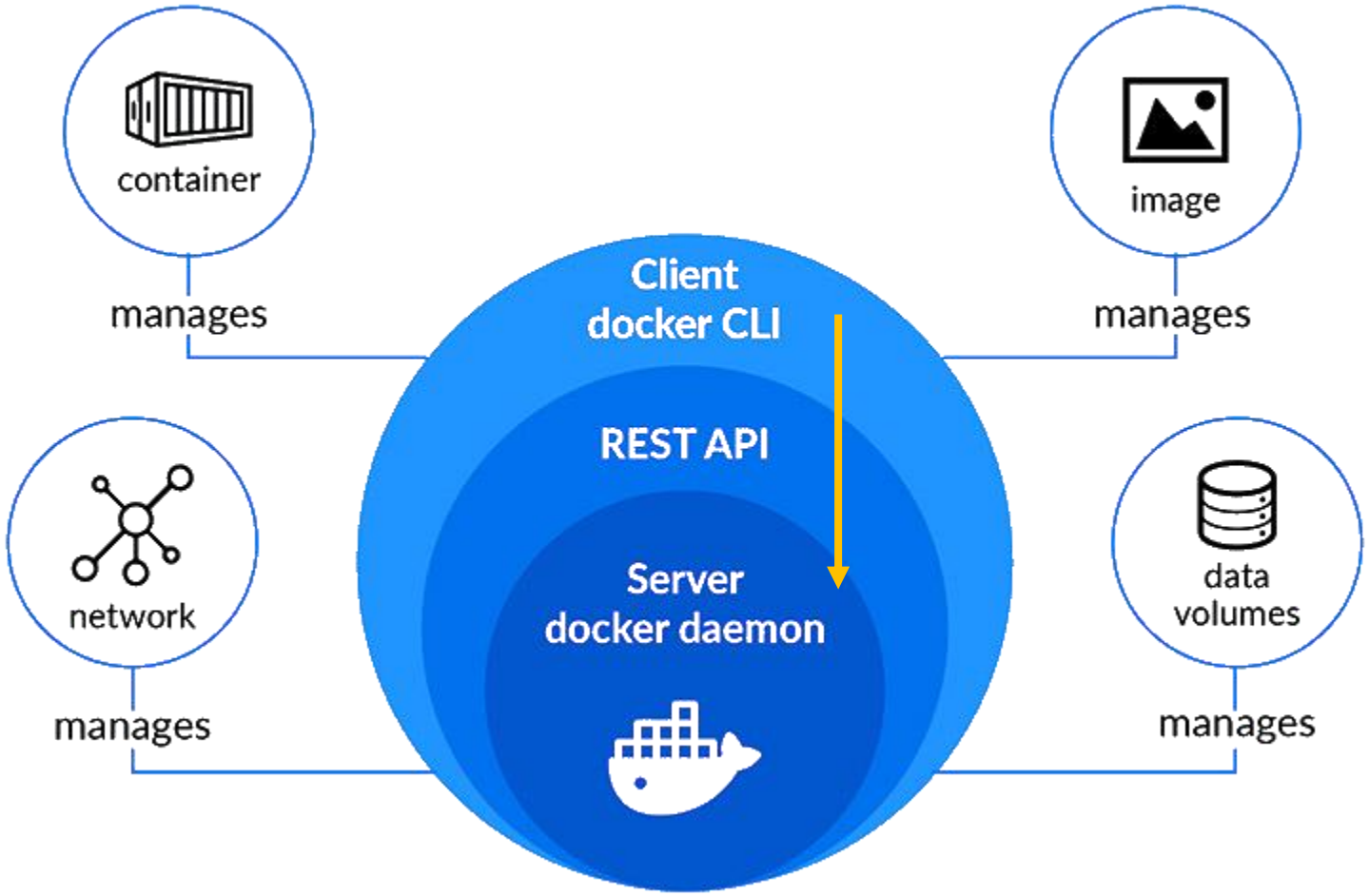
Some Commands:

docker run: Create and start a container.
docker ps: List running containers.
docker images: List available images.
docker pull: Download an image from a registry.
docker build: Build an image from a Dockerfile.
docker exec: Run a command in a running container.

Docker CLI


Container
Image
Manages
Manages
Client
Docker CLI
REST API
CMD
Network
Data Volumes
Server
Docker
daemon
Manages
Manages

Docker Engine: Docker deamon



Docker server/deamon

The Docker daemon (dockerd) is a background service that manages Docker containers, images, networks, and volumes.

Responsibilities:

- Handles container lifecycle operations (create, start, stop, remove).
- Manages images and their storage.
- Listens for API requests and communicates with the Docker client.
- Manages networking and storage for containers.

Docker REST API

The Docker Engine REST API is a set of HTTP endpoints that allows users to interact programmatically with the Docker Engine.

Common Endpoints

Containers:

GET /containers/json: List all containers.

POST /containers/create: Create a new container.

DELETE /containers/{id}: Remove a container.

Images:

GET /images/json: List all images.

POST /images/create: Pull an image from a registry.

DELETE /images/{name}: Remove an image.

Networks:

GET /networks: List all networks.

POST /networks/create: Create a new network.

Docker list

```
$ docker ps  % shows only running containers
$ docker ps –a % Shows all containers, regardless of their state (running, stopped, exited).
```

Output:

- Container ID
- Image name
- Command being run
- Creation time
- Status
- Ports
- Names

Docker start/stop/rm

```
$ docker start my_container
$ docker stop my_container
$ docker rm my_container
```

Docker exec

```
$ docker exec -it my_container bash
```

https://create.kahoot.it/share/docker/87a25f3b-3ec7-4350-b557-480da6b4c01c

Manage Docker clusters

- Docker swarm
- Dokcer-compose

Docker compose

Docker Compose is a tool for defining and running multi-container Docker applications using a simple YAML file.

Key Features:

- Multi-Container Management: Define multiple services, networks, and volumes in a single file.
- Simplified Configuration: Use a docker-compose.yml file to configure your application's services.
- Single Command Deployment: Start all services with a single command (docker-compose up).

Docker Compose: For defining and running multi-container applications locally.

Docker swarm

Docker Swarm is a native clustering and orchestration tool for Docker. It allows you to manage a cluster of Docker engines (nodes) as a single virtual system.

Key Features:

- Clustering: Combines multiple Docker hosts into a single virtual host.
- Load Balancing: Distributes incoming requests across multiple containers.
- Scaling: Easily scale services up or down.
- High Availability: Ensures that services are running and can recover from failures.

Docker Swarm: For orchestrating and managing clusters of Docker containers in production.

References

1 ETSI GS NFV-IFA 040 V4.1.1 (2020-11) Network Functions Virtualisation (NFV) Release 4; Management and Orchestration; Requirements for service interfaces and object model for OS container management and orchestration specification

2 https://www.openstack.org/

3 https://kubernetes.io/

4 https://www.docker.com/

5 https://hub.docker.com/

6 https://quay.io/

7 https://www.cncf.io/

8 https://microservices.io

9 https://12factor.net/

10 https://www.nginx.com/blog/introduction-to-microservices/

11 https://cloud.google.com/containers