Lecture #10
Containers

Spring 2024

Containers: A Modern Way to Deploy Applications

- Containers are lightweight, portable, and isolated from each other.
- They can be used to deploy applications on a variety of platforms, including servers, clouds, and even on devices.
- Containers are a popular choice for deploying microservices architectures.

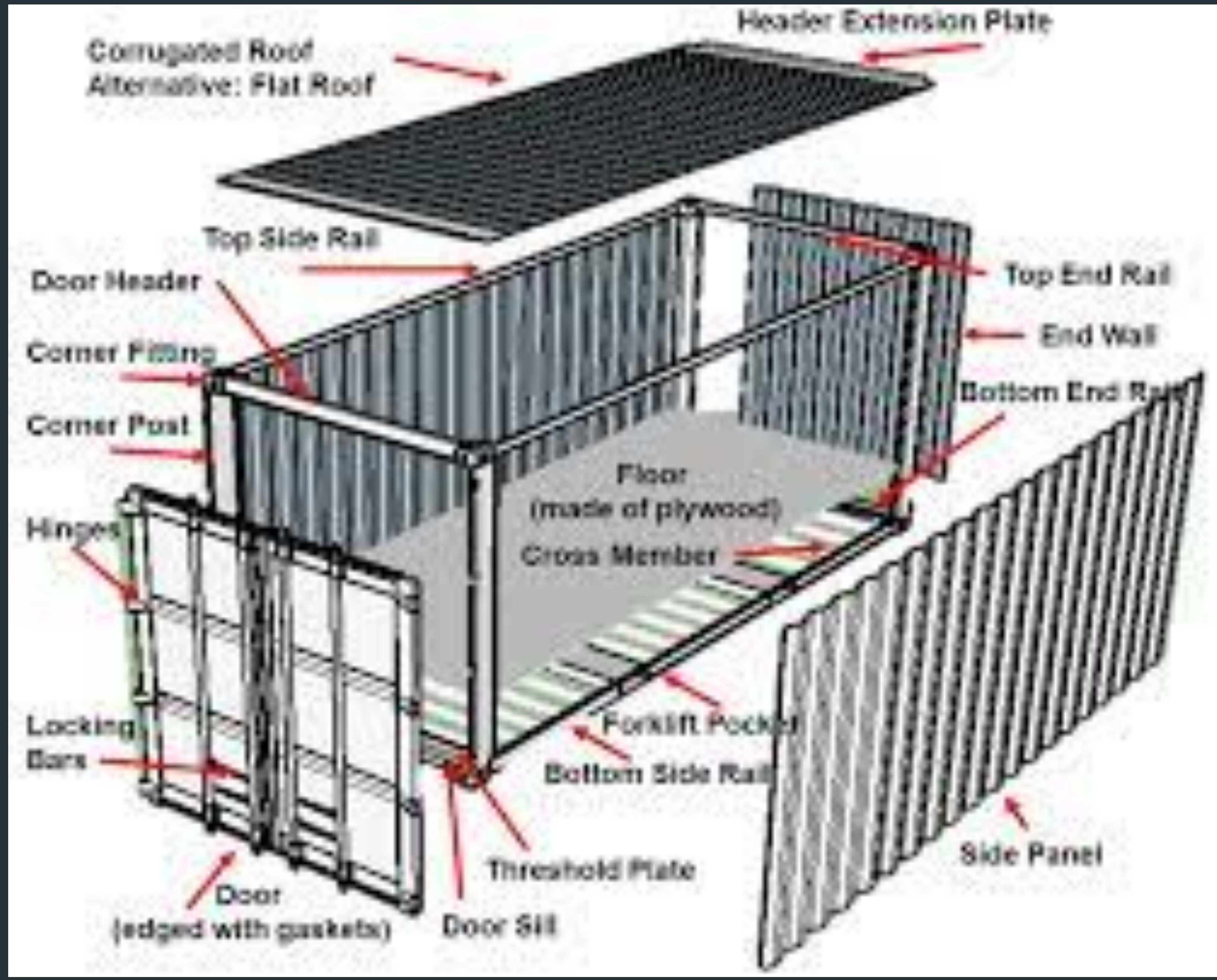
Structure

- **Image**: The image is a read-only template that contains the code, runtime, system tools, system libraries, and settings for a container.

- **Container**: A container is a runnable instance of an image.

- **Container runtime**: The container runtime is responsible for managing the lifecycle of containers.

- **Container orchestration platform**: A container orchestration platform is a tool that helps to manage and scale containerized applications.

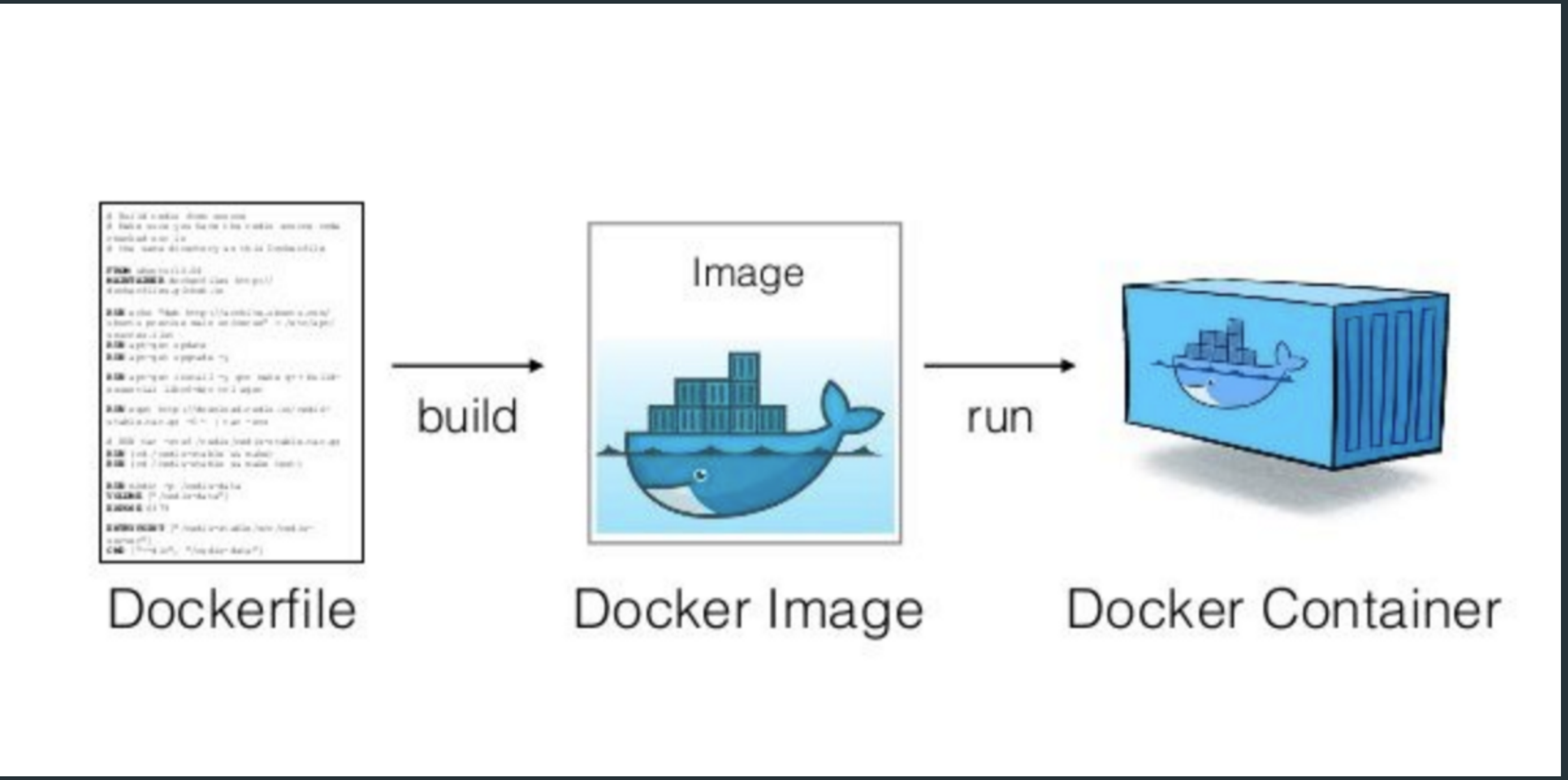


Image

- An image is a read-only template that contains the code, runtime, system tools, system libraries, and settings for a container.

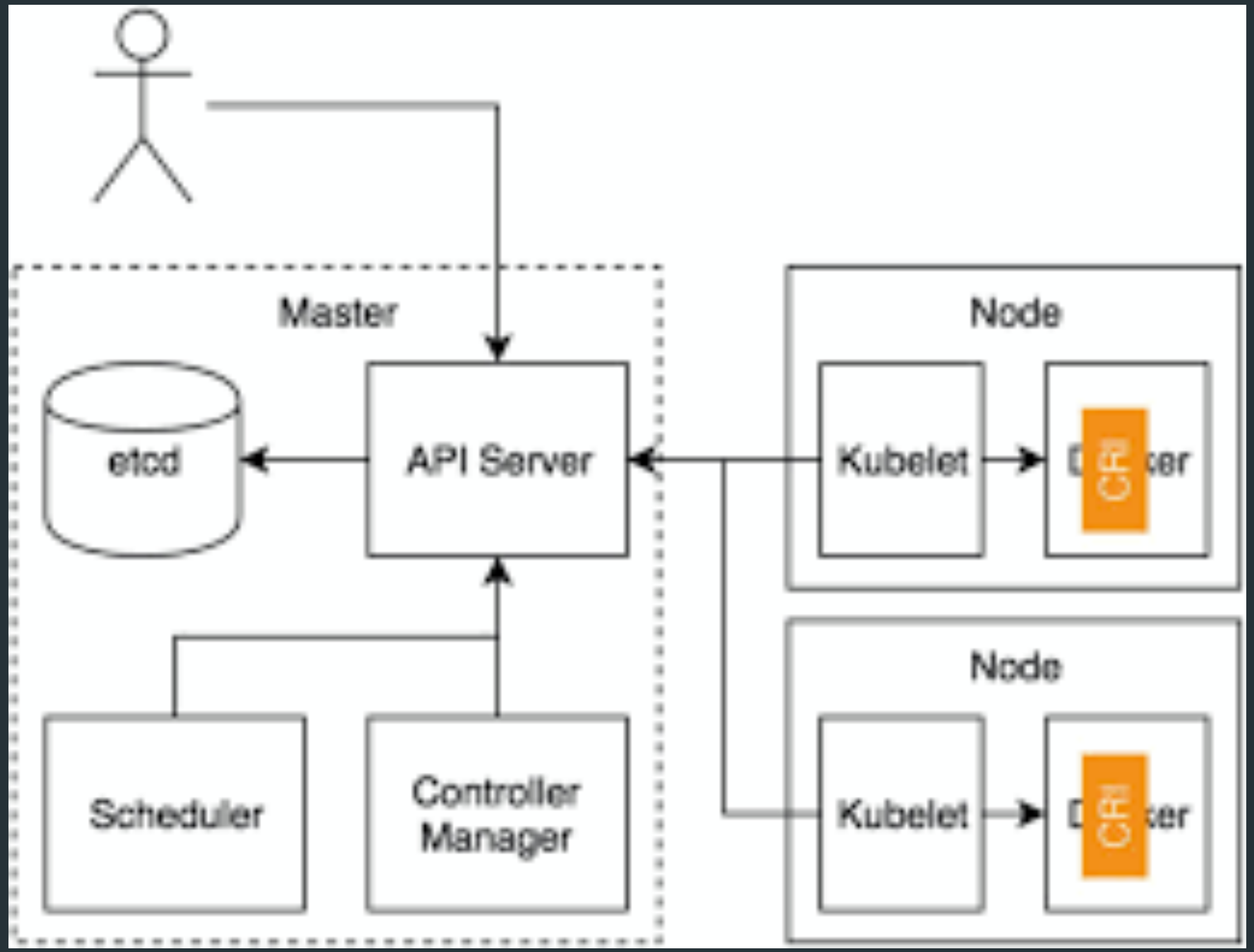
- Images are created using a tool called a Dockerfile.

- Images can be shared and reused.



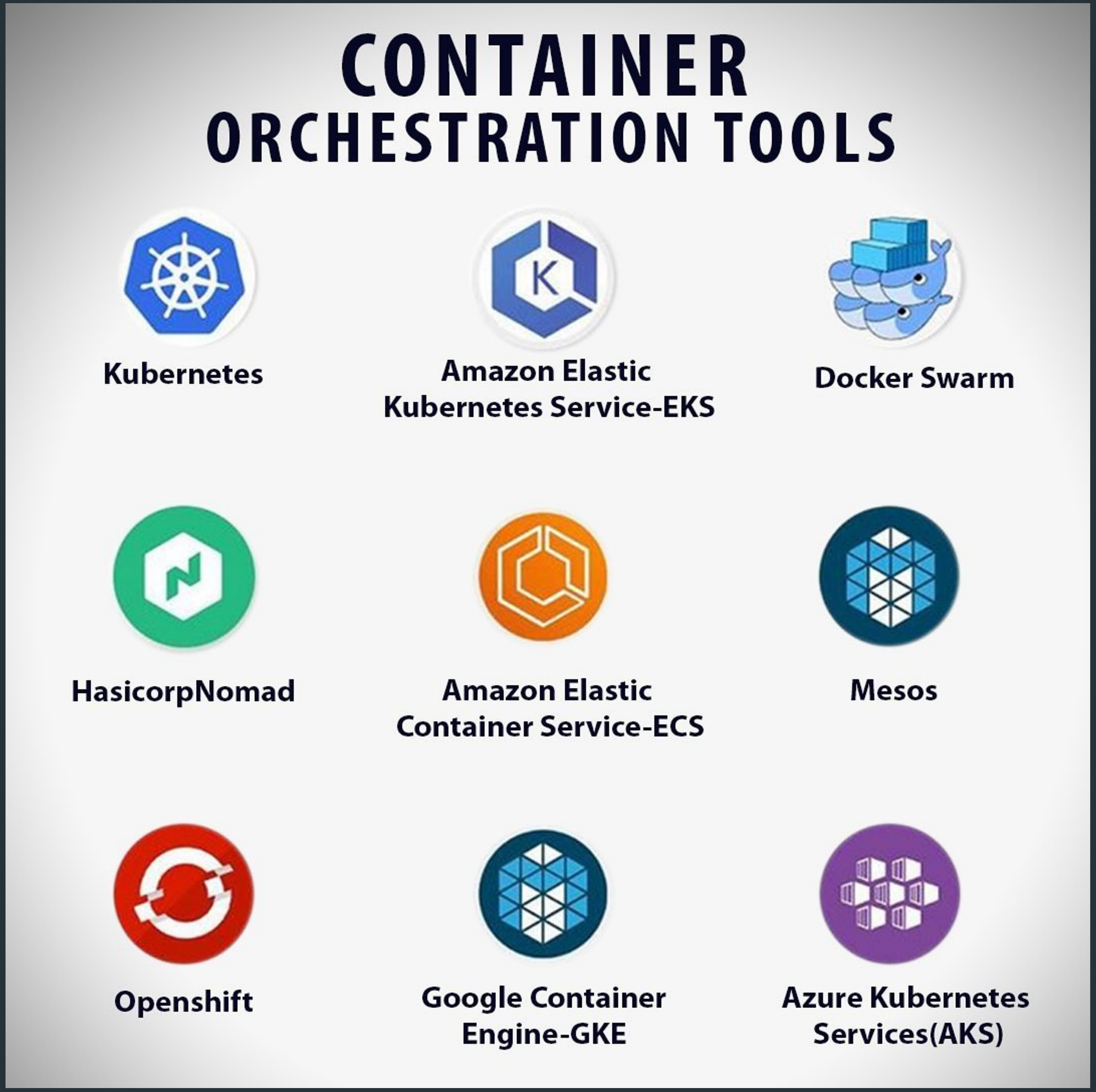
Runtime

- Responsible for managing the lifecycle of containers.
- Creates, starts, stops, and destroys containers.
- Provides isolation between containers.



Container Orchestration Platform

- A container orchestration platform is a tool that helps to manage and scale containerized applications.
- Container orchestration platforms provide features such as:
  - Automatic deployment
  - Load balancing
  - Autoscaling
  - Monitoring



Objectives

- Containers are a way to package and run applications.
- They are lightweight, portable, and isolated from each other.
- They can be used to deploy applications on a variety of platforms, including servers, clouds, and even on devices.

Benefits

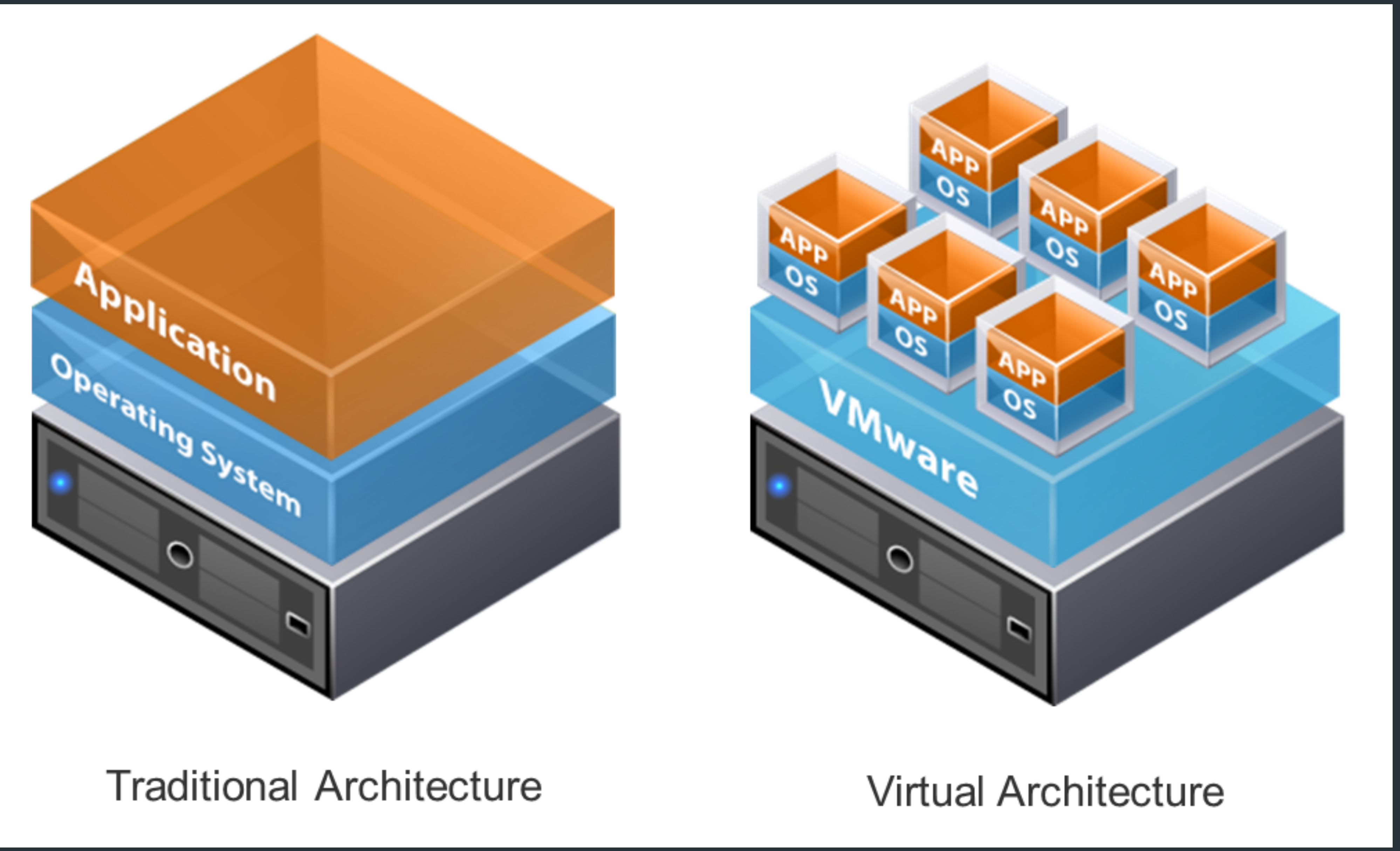
- Portability: Containers can be easily moved from one environment to another.
- Isolation: Containers are isolated from each other, which means that they cannot affect each other.
- Efficiency: Containers share the underlying operating system kernel, which makes them more efficient than virtual machines.

Examples of Containers

- Docker
- Kubernetes
- OpenShift

Containers and Virtualization

- Containers are a form of virtualization.
- Containers are more lightweight and portable than virtual machines.



Virtual Machines

- Virtual machines are a way to create isolated operating system environments.

- They are more heavyweight than containers, but they offer more features, such as the ability to run different operating systems on the same physical hardware.

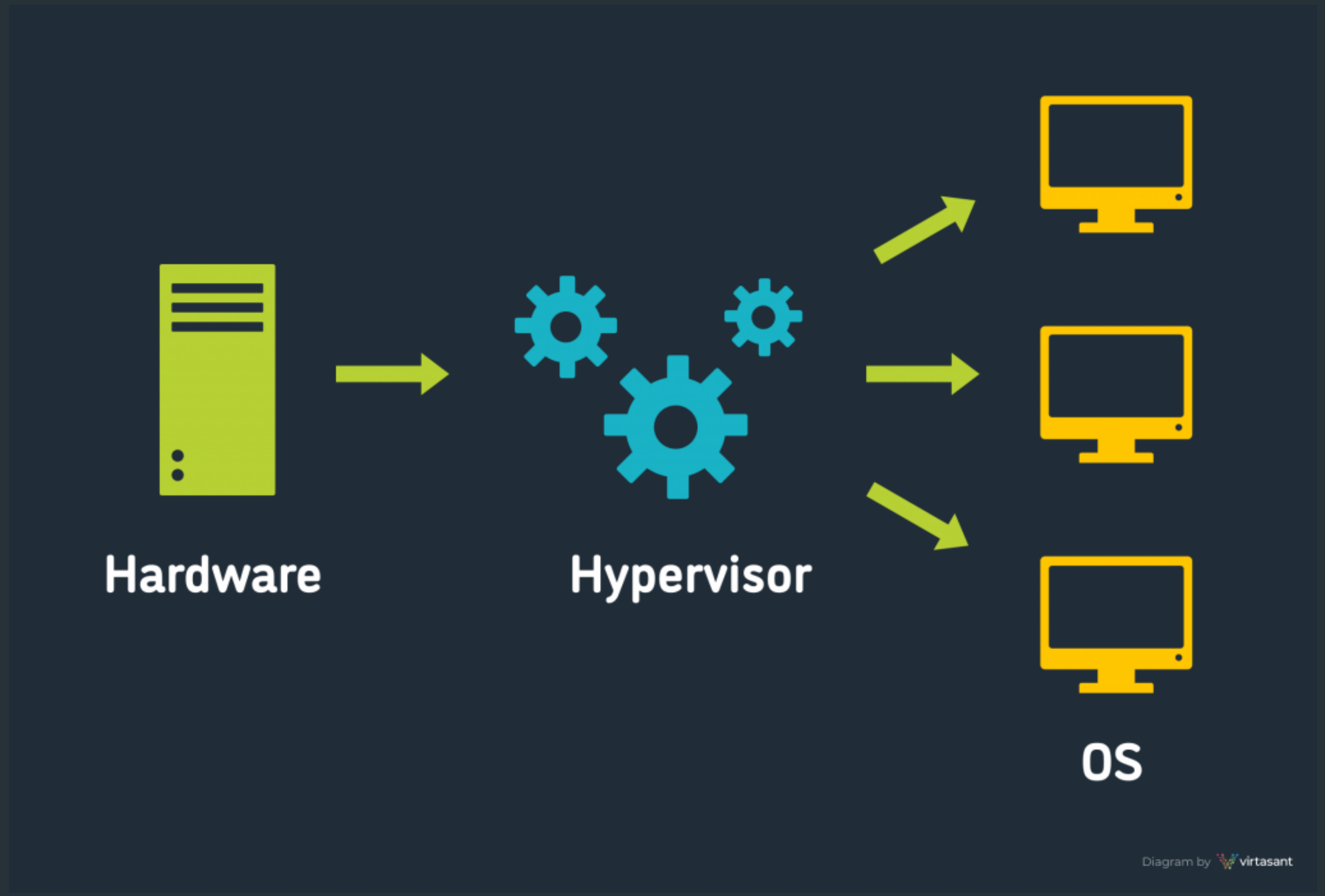
- Virtual machines are a good choice for running legacy applications or applications that require a specific operating system.

Hypervisors

- Hypervisors are the software that allows virtual machines to run on the same physical hardware.

- There are two main types of hypervisors: Type 1 and Type 2.

- Type 1 hypervisors run directly on the physical hardware, while Type 2 hypervisors run on top of an operating system.

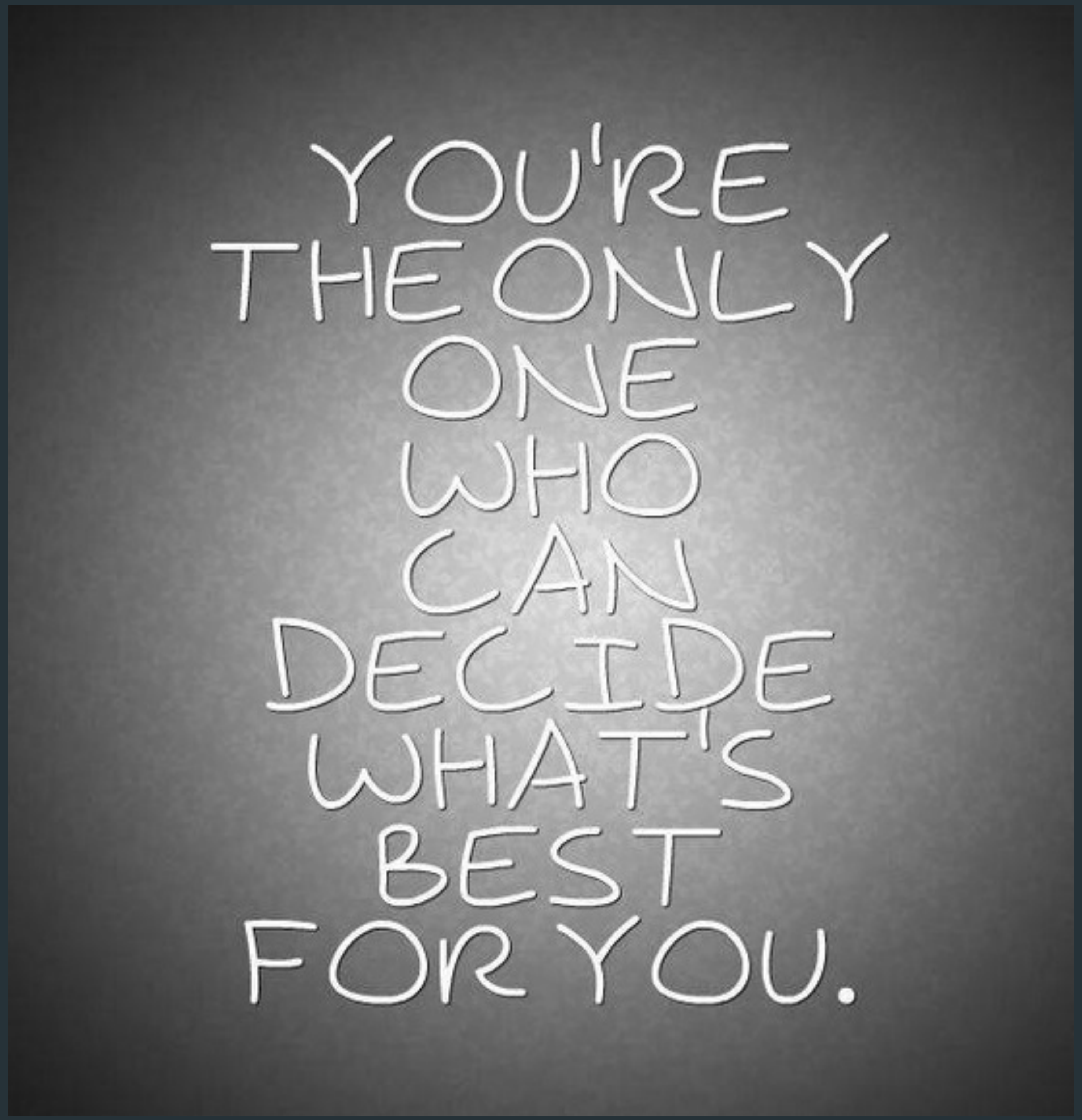


Which is right for you?

- Choose containers if you need a lightweight and portable way to run applications.

- Choose virtual machines if you need a more heavyweight solution that offers more features.

- Choose a hypervisor if you need to run multiple virtual machines on the same physical hardware.



DIY: Running a Container on Linux

Step 1: Prepare the System

- Install a Linux distribution that supports containers
- Update the system
- Install the necessary tools
- Create a user group for containers
- Add your user to the container group

```
sudo apt install build-essential
sudo apt update
sudo apt install lxc-utils
sudo groupadd containers
sudo usermod -aG containers $USER
```

Step 2: Obtaining the Container

- There are two ways to obtain a container image:
- Pull it from a registry
- Build it yourself

```
# Pull an image from a registry
docker pull nginx

# Build an image yourself
docker build -t my-nginx .
```

Step 2b: Build and image yourself

```
FROM nginx:latest

# Copy the nginx configuration file to the container
COPY nginx.conf /etc/nginx/nginx.conf

# Expose port 80 to the host machine
EXPOSE 80

# Start the nginx web server
CMD ["nginx", "-g", "daemon off;"]
```

```
docker build -t my-nginx .
```

Step 3: Setup the group

- Control groups (cgroups) are a Linux kernel feature that allows you to limit and manage resources for groups of processes.
- Cgroups can be used to limit CPU usage, memory usage, disk I/O, and network bandwidth.
- To set up cgroups, you will need to create a cgroup hierarchy.

```
# Create a cgroup hierarchy
mkdir -p /cgroup
mkdir -p /cgroup/cpu
mkdir -p /cgroup/memory
mkdir -p /cgroup/disk
mkdir -p /cgroup/network
```

Step 3: Setup the group

- Control groups (cgroups) are a Linux kernel feature that allows you to limit and manage resources for groups of processes.
- Cgroups can be used to limit CPU usage, memory usage, disk I/O, and network bandwidth.
- To set up cgroups, you will need to create a cgroup hierarchy.

```
# Create a cgroup hierarchy
mkdir -p /cgroup
mkdir -p /cgroup/cpu
mkdir -p /cgroup/memory
mkdir -p /cgroup/disk
mkdir -p /cgroup/network
```

```
cgroups add -g cpu0 nginx
```

Step 3: Setup the group

- Control groups (cgroups) are a Linux kernel feature that allows you to limit and manage resources for groups of processes.
- Cgroups can be used to limit CPU usage, memory usage, disk I/O, and network bandwidth.
- To set up cgroups, you will need to create a cgroup hierarchy.

```
# Create a cgroup hierarchy
mkdir -p /cgroup
mkdir -p /cgroup/cpu
mkdir -p /cgroup/memory
mkdir -p /cgroup/disk
mkdir -p /cgroup/network
```

```
cgroups add -g cpu0 nginx

cgroups set -r cpu.shares=50 nginx

cgroups stat -c cpu nginx
```

Step 3: Run the Container

- To run a container, you will need to use the docker run command.

- The docker run command takes a number of arguments, including the name of the image to run, the command to run inside the container, and the environment variables to set.

- For example, to run the nginx web server on port 80, you would use the following command:

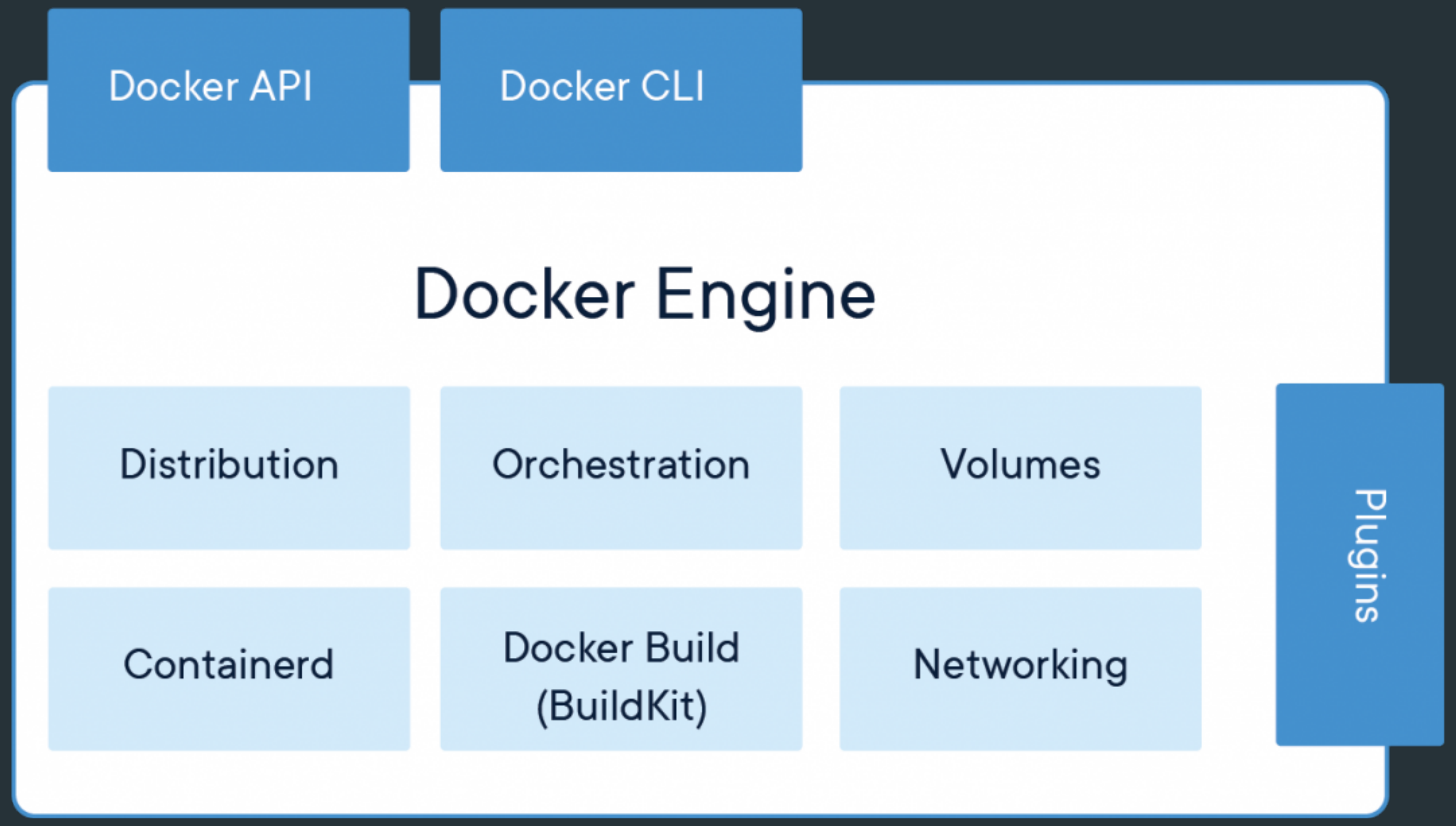
```
docker run -p 80:80 nginx
```

Docker

- Engine
- Daemon
- Registry
- Client

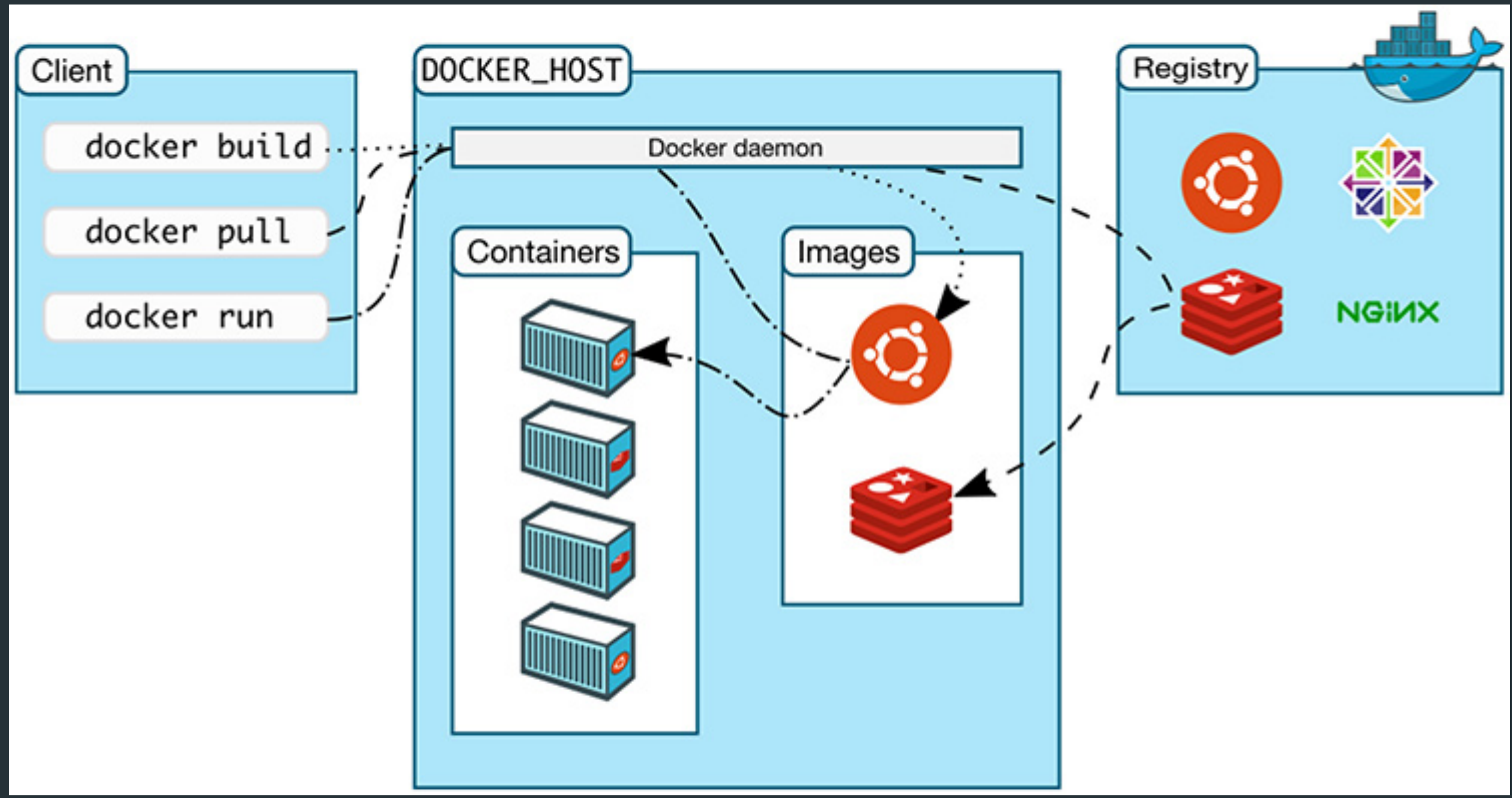
Docker Engine

- Is the core component of Docker that manages containers.
- Responsible for creating, starting, stopping, and removing containers.
- Manages the resources that are allocated to containers.



Docker Daemon

- A background process that runs on the host machine and manages the Docker Engine.
- Responsible for communicating with the Docker Registry and downloading Docker images.
- Manages the containers that are running on the host machine.

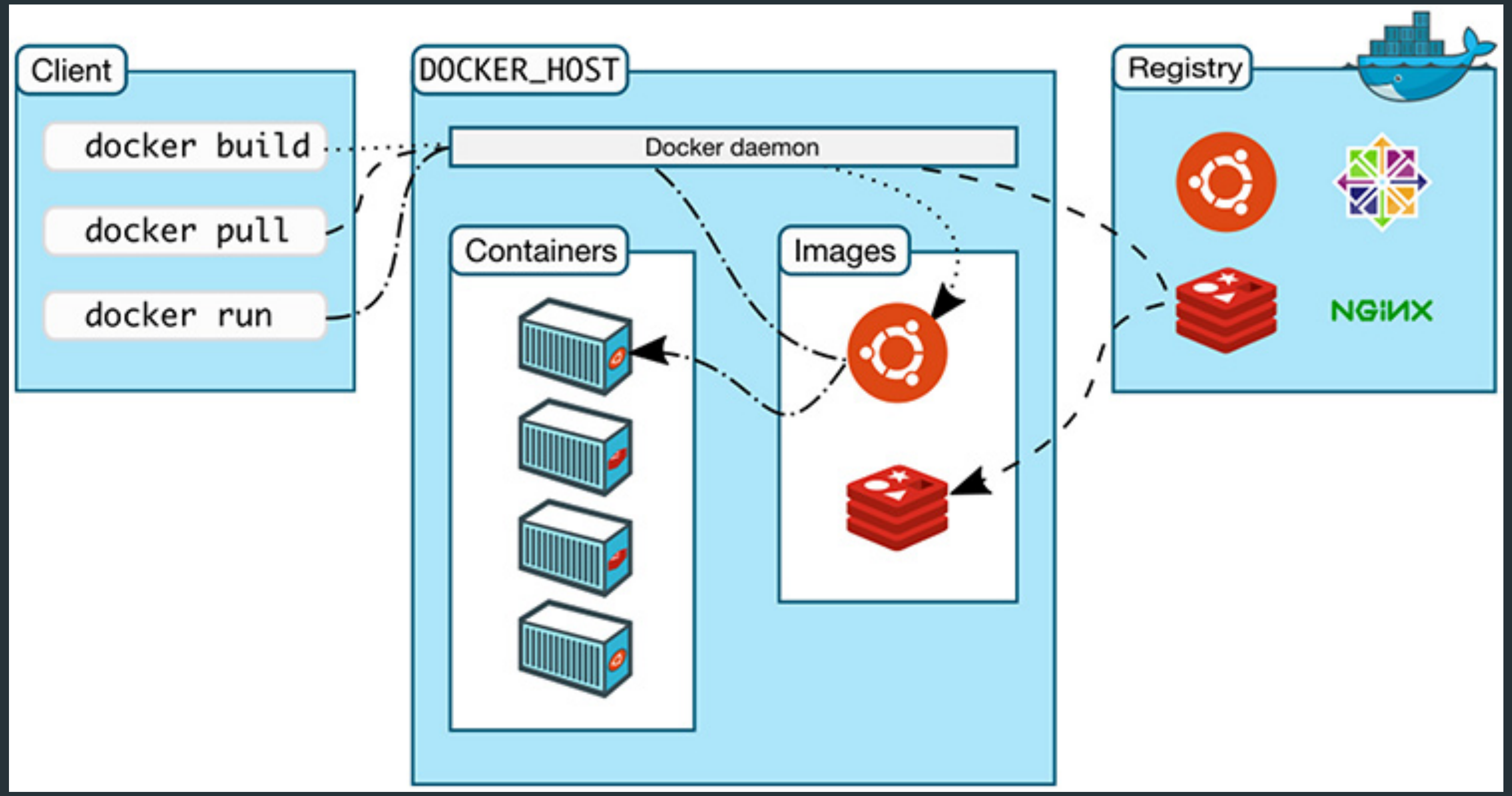


Docker Registry

- A central repository where Docker images can be stored and shared.

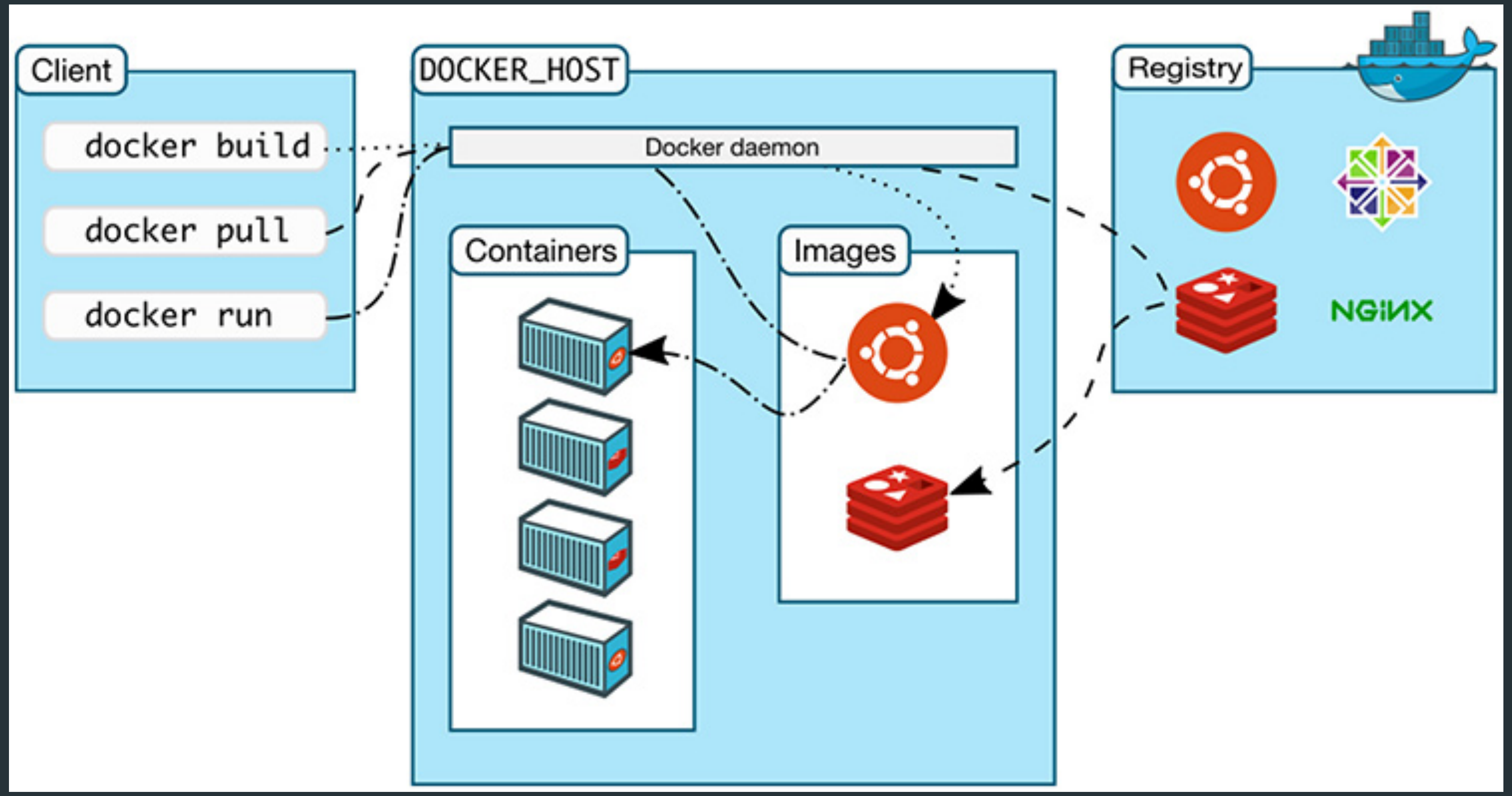
- Docker images are essentially read-only templates that can be used to create containers.

- Docker images can be created from scratch or they can be built from other Docker images.



Docker Client

- A command-line tool that allows you to interact with the Docker Engine.
- Used to create, start, stop, and remove containers.
- Used to manage the resources that are allocated to containers.



Writing Docker Files

- A Docker file is a text file that contains the instructions for building a Docker image.
- Docker files are written in a simple, human-readable format.
- Docker files can be used to build Docker images for any application.

```
FROM nginx:latest

COPY . /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Writing Docker Files

- A typical Docker file will include the following steps:
  - FROM: This specifies the base image that the Docker image will be built on.
  - RUN: This specifies commands that will be run when the Docker image is built.
  - COPY: This specifies files or directories that will be copied into the Docker image.
  - EXPOSE: This specifies ports that will be exposed by the Docker image.
  - CMD: This specifies the command that will be run when the Docker image is started.

```
FROM nginx:latest

COPY . /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

More Docker Instructions

- ARG: Defines an environment variable that can be used in the Docker file.
- AND: Specifies that the following commands should be run only if the previous command succeeds.
- ENTRYPOINT: The command that will be run when the Docker image is started.
- WORKDIR: The working directory for the Docker image.
- STOPSIGNAL: The signal that will be sent to the Docker container when it is stopped.
- USER: The user that will run the commands in the Docker image.
- LABEL: A label for the Docker image.
- ENV: An environment variable for the Docker image.

```
ARG VERSION=1.0

RUN apt-get update && apt-get install -y nginx

ENTRYPOINT nginx

WORKDIR /usr/share/nginx/html

STOPSIGNAL SIGTERM

USER nginx

LABEL app=nginx

ENV PORT=80
```

Docker Commands

- Some of the most common Docker commands include:
  - docker build: Builds a Docker image from a Dockerfile.
  - docker run: Starts a Docker container from an image.
  - docker stop: Stops a Docker container.
  - docker remove: Removes a Docker container.
  - docker inspect: Inspects a Docker container.
  - docker logs: Displays the logs for a Docker container.

```
# Build a Docker image from a Dockerfile
docker build -t my-app .

# Start a Docker container from an image
docker run -p 80:80 my-app

# Stop a Docker container
docker stop my-app

# Remove a Docker container
docker rm my-app

# Inspect a Docker container
docker inspect my-app

# Display the logs for a Docker container
docker logs my-app
```

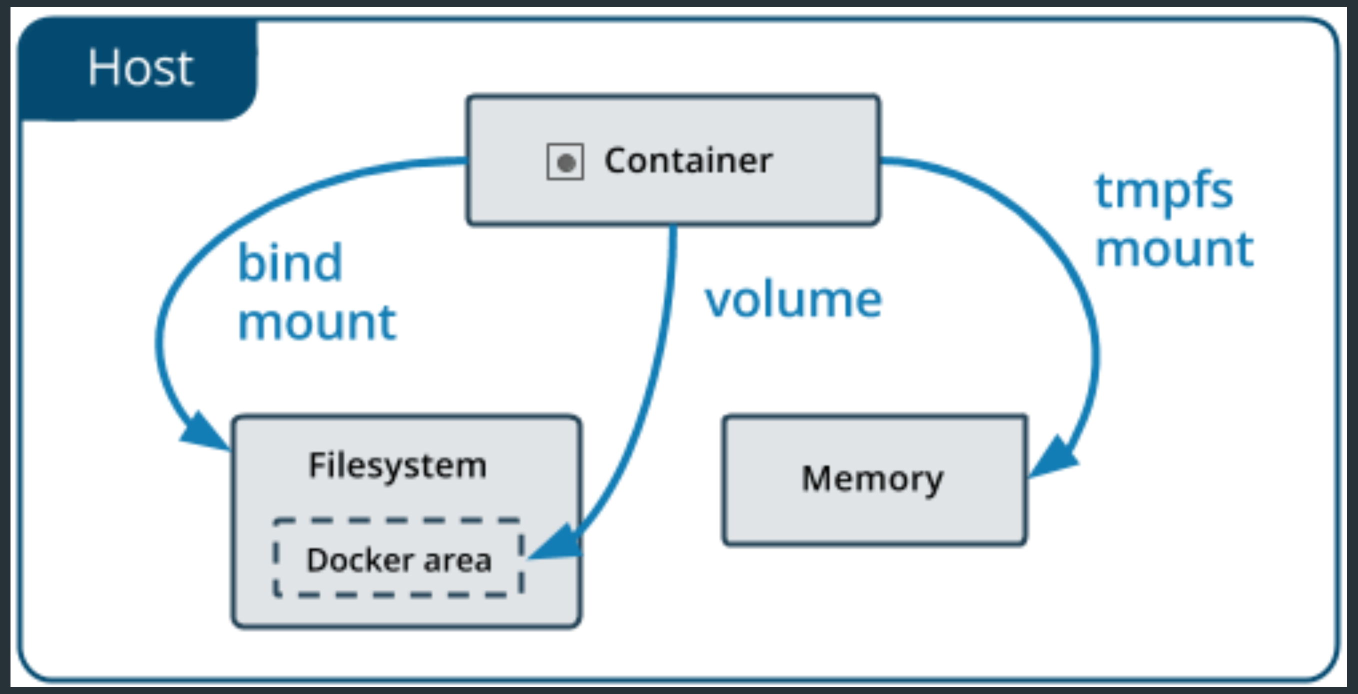
More Docker Commands

```
# List all containers
docker ls

# Show the history of a container
docker history my-container

# Search for Docker images
docker search nginx

# Pull an image from a registry
docker pull nginx

# Tag an image
docker tag nginx:latest my-nginx

# Push an image to a registry
docker push my-nginx

# Rename a container
docker rename my-container my-new-container

# Attach to a running container
docker attach my-container
```

```
# Tag an image
docker tag nginx:latest my-nginx

# Push an image to a registry
docker push my-nginx

# Rename a container
docker rename my-container my-new-container

# Attach to a running container
docker attach my-container

# Commit a container's changes to an image
docker commit my-container my-new-image

# Get statistics for a container
docker stats my-container

# Wait for a container to finish
docker wait my-container

# Diff the changes between a container's image and its filesystem
docker diff my-container

# Copy files from a container to the host
docker cp my-container:/path/to/file /path/to/destination
```

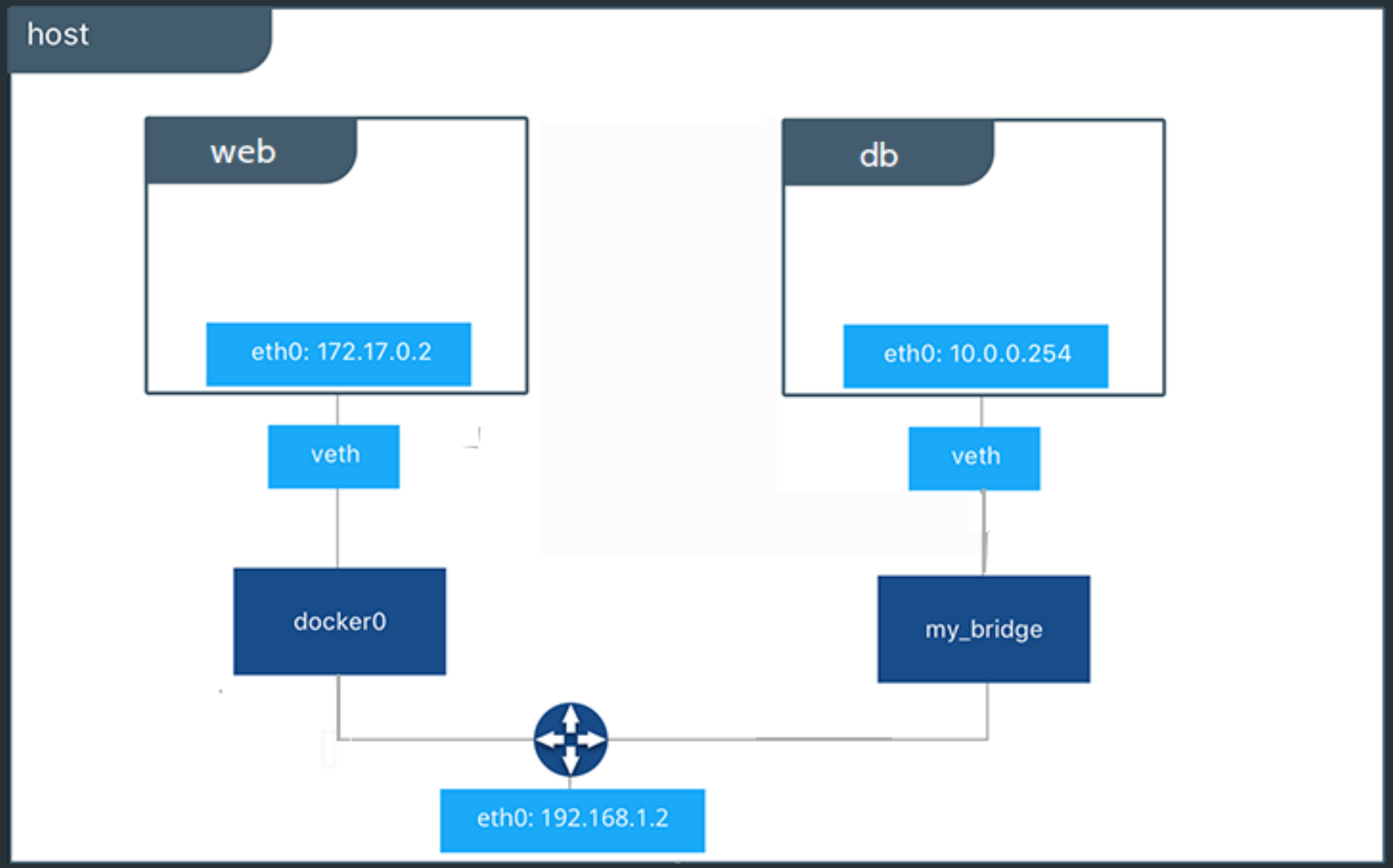
Connectivity and Storage

- Docker links: Docker links allow containers to communicate with each other by name.
- Docker networks: Docker networks allow containers to communicate with each other over a network.
- Docker volumes: Docker volumes allow containers to share data with each other and with the host machine.



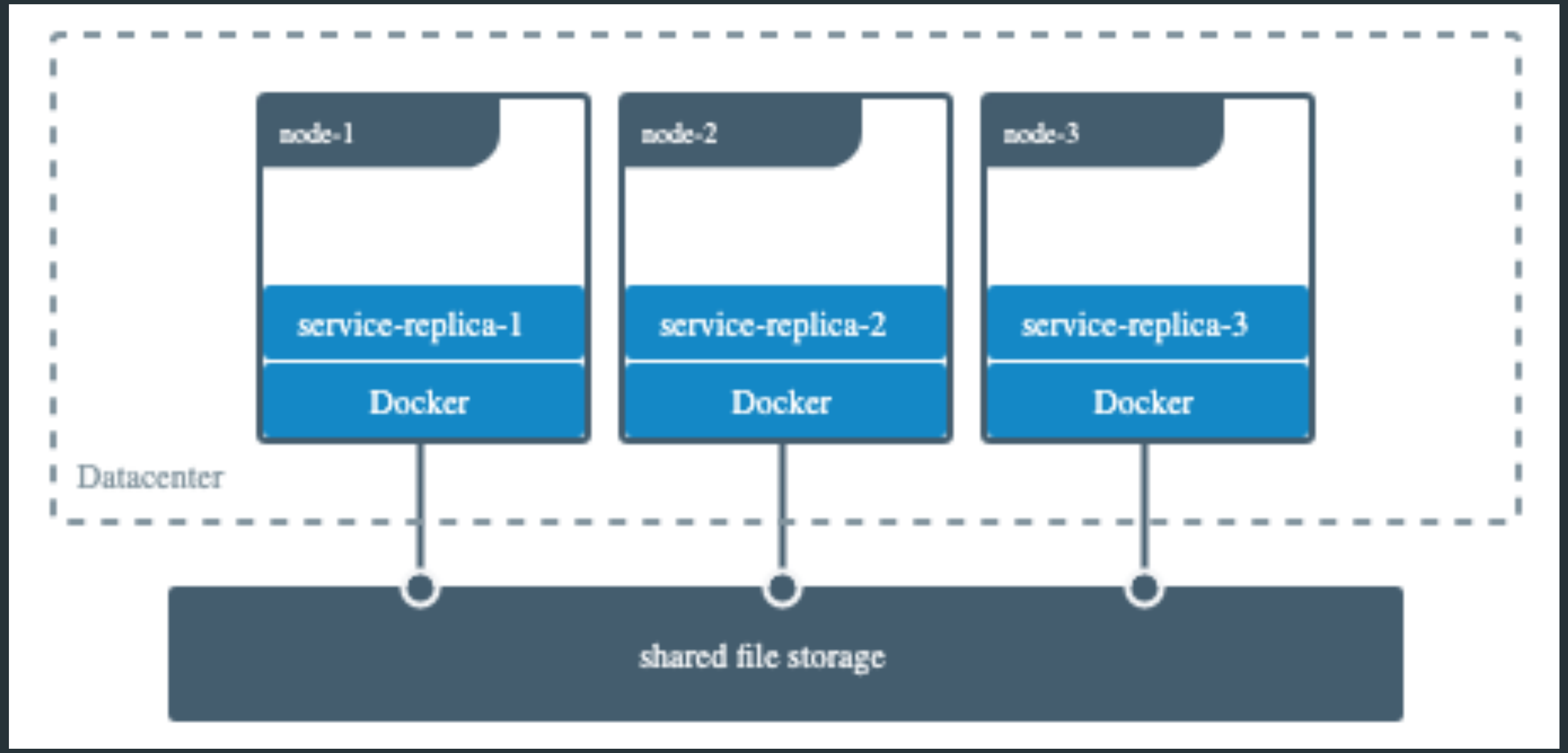
Docker Connectivity

- Docker containers can be connected to the host machine's network using a variety of methods, such as:
  - Bridged networking: Bridged networking allows containers to connect to the host machine's network as if they were physical machines.
  - Host networking: Host networking allows containers to share the host machine's network interface.
  - Overlay networks: Overlay networks allow containers to connect to a network that is shared by multiple hosts.



Docker Storage

- Docker volumes can be created as:
  - Local volumes: Local volumes are stored on the host machine's filesystem.
  - Named volumes: Named volumes are stored in a shared location, such as a Docker registry.
  - Tmpfs volumes: Tmpfs volumes are stored in temporary memory.

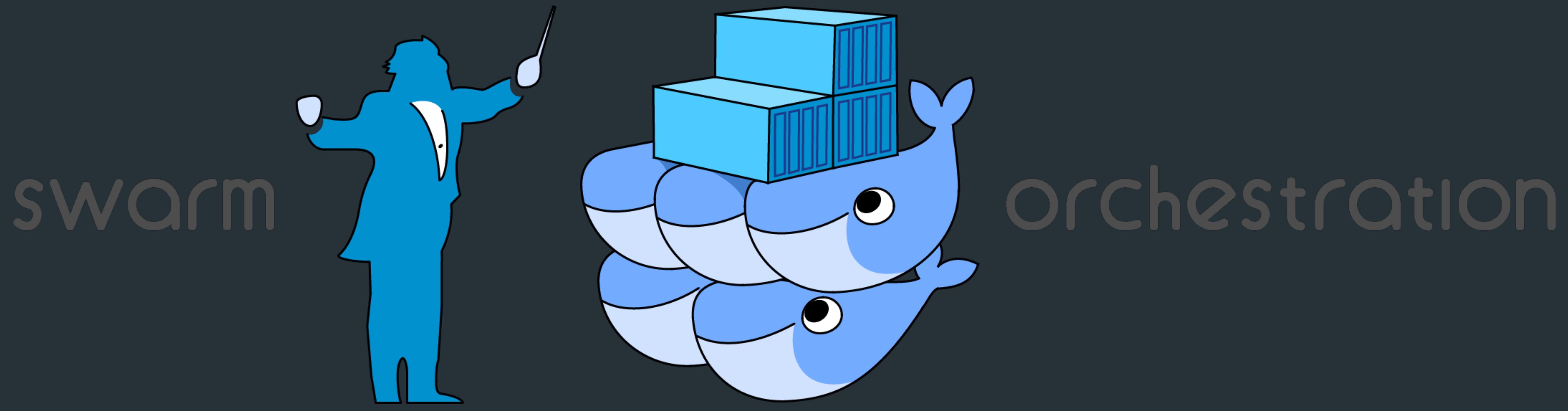


Docker Swarm

- A container orchestration system that allows you to run and manage Docker containers across multiple machines.

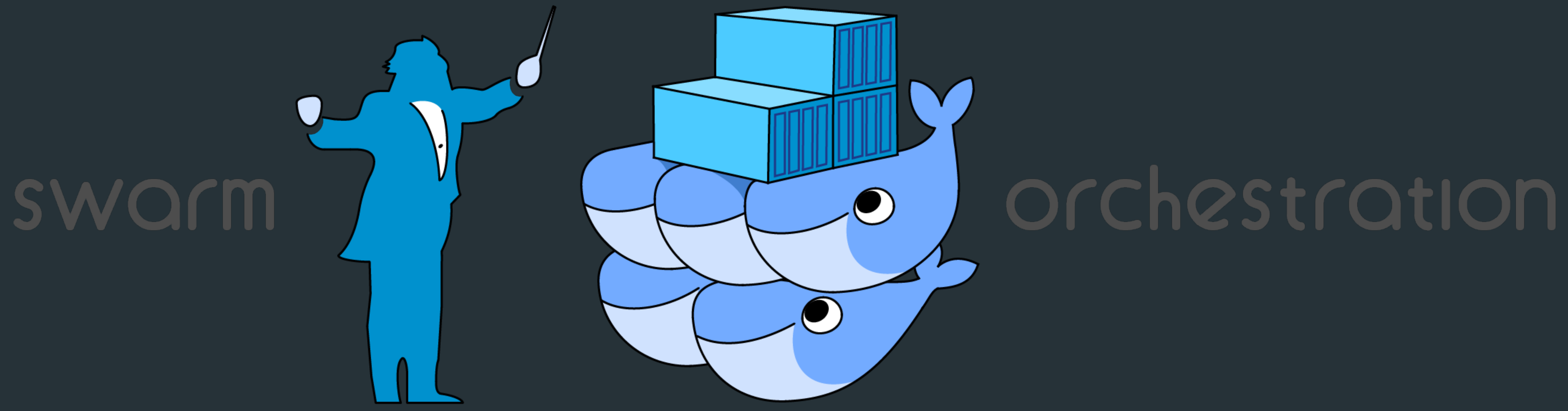
- A powerful tool that can be used to deploy and manage large-scale applications.

- Easy to use and can be managed using the Docker CLI or the Docker Swarm UI.



Benefits of Using Docker Swarm

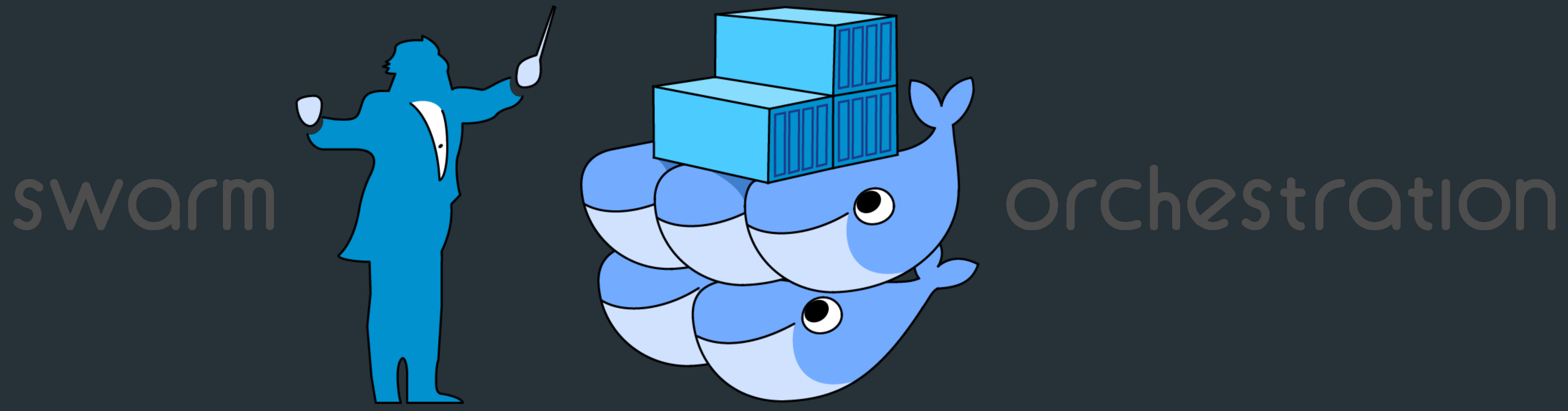
- Scalability
- High availability
- Ease of use
- Cost-effectiveness



How to Use Docker Swarm

- To use Docker Swarm, you first need to create a Docker Swarm cluster.
- Once you have created a Docker Swarm cluster, you can then deploy applications to the cluster.
- You can manage Docker Swarm clusters using the Docker CLI or the Docker Swarm UI.

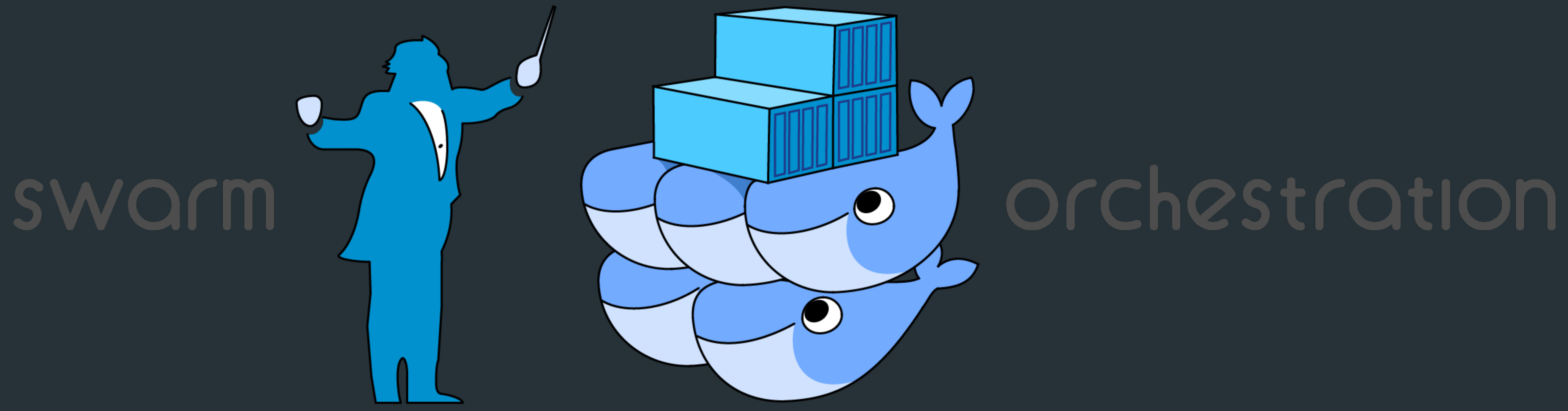
docker swarm init



How to Use Docker Swarm

- To use Docker Swarm, you first need to create a Docker Swarm cluster.
- Once you have created a Docker Swarm cluster, you can then deploy applications to the cluster.
- You can manage Docker Swarm clusters using the Docker CLI or the Docker Swarm UI.

```
docker swarm init

docker stack deploy -c <stack-file> <stack-name>
```



Introduction to Kubernets

- An open-source container orchestration system.
- Used to automate deployment, scaling, and management of containerized applications.
- Used to deploy applications on a single machine or on a cluster of machines.

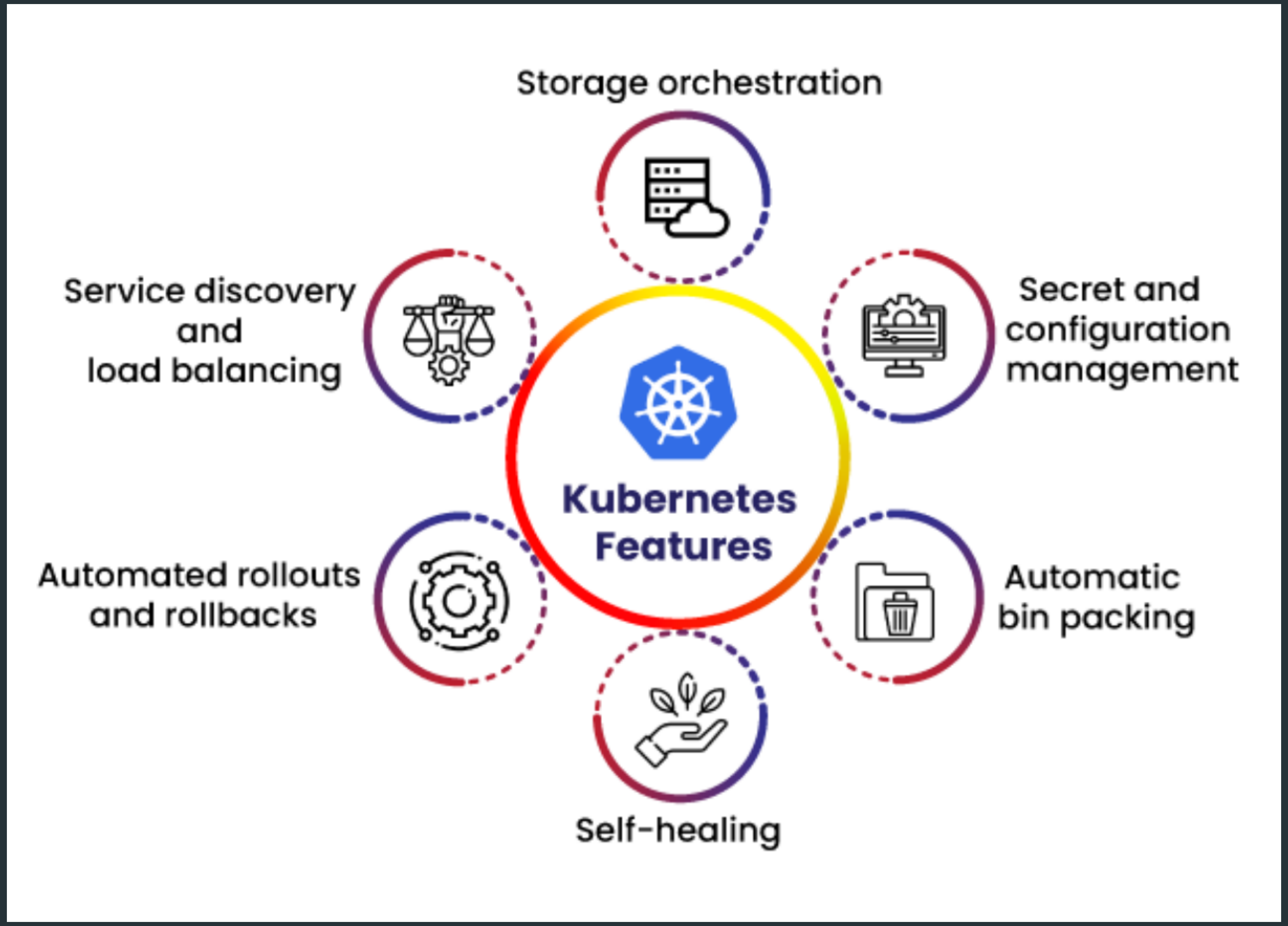
Benefits of Using Kubernetes

- Scalability: Kubernetes is highly scalable and can be used to deploy and manage applications of any size.

- High availability: Kubernetes is highly available and can continue to run applications even if some of the underlying nodes fail.

- Ease of use: Kubernetes is easy to use and can be managed using a variety of tools, including the command-line, the web UI, and the kubectl command-line tool.

- Cost-effectiveness: Kubernetes is cost-effective and can be used to save money on hardware and infrastructure costs.



How to Use Kubernetes

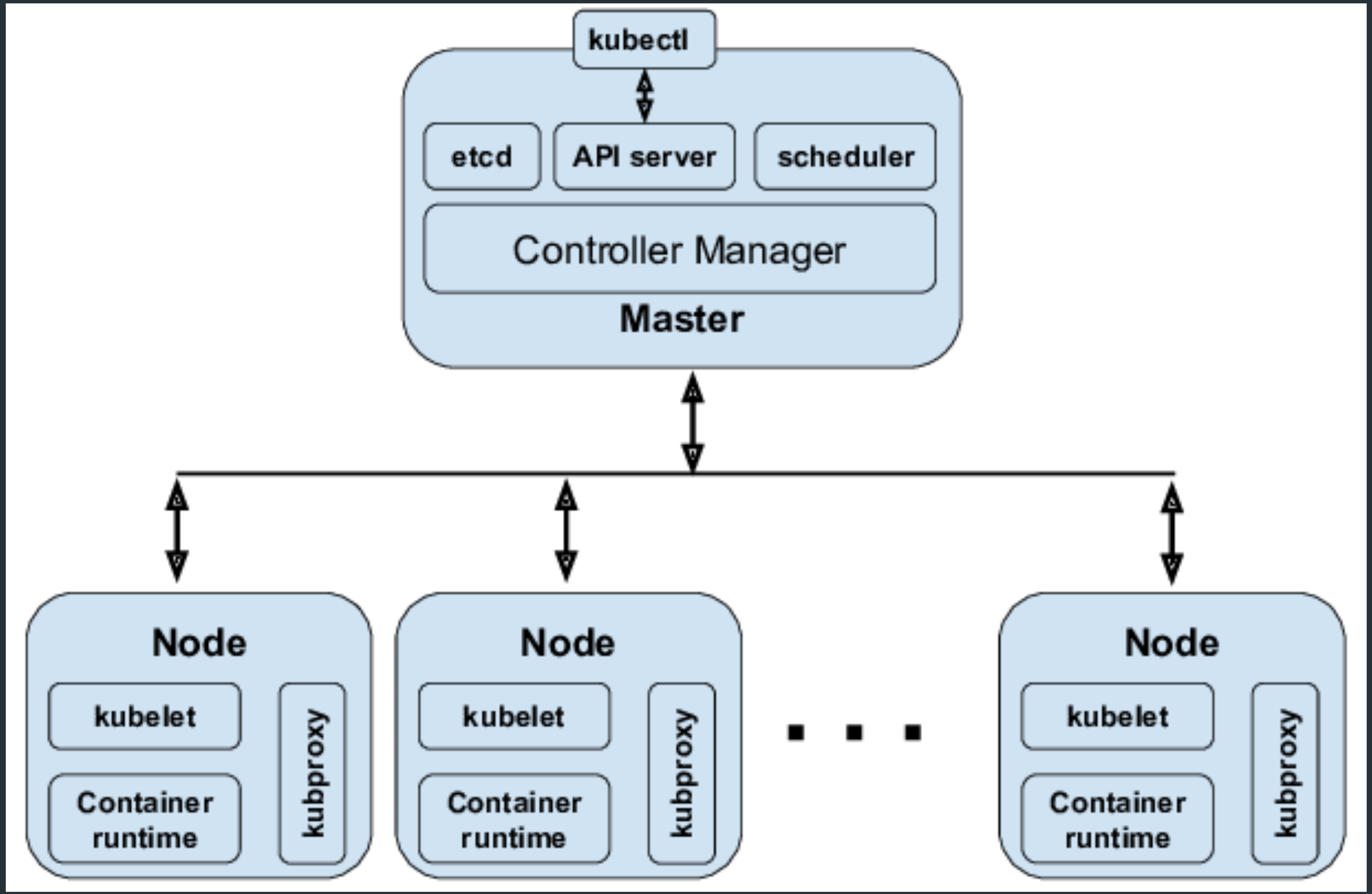
- To use Kubernetes, you first need to create a Kubernetes cluster.
- Once you have created a Kubernetes cluster, you can then deploy applications to the cluster.
- You can manage Kubernetes clusters using the command-line, the web UI, or the kubectl command-line tool.

kubectl create cluster

kubectl apply -f <manifest-file>

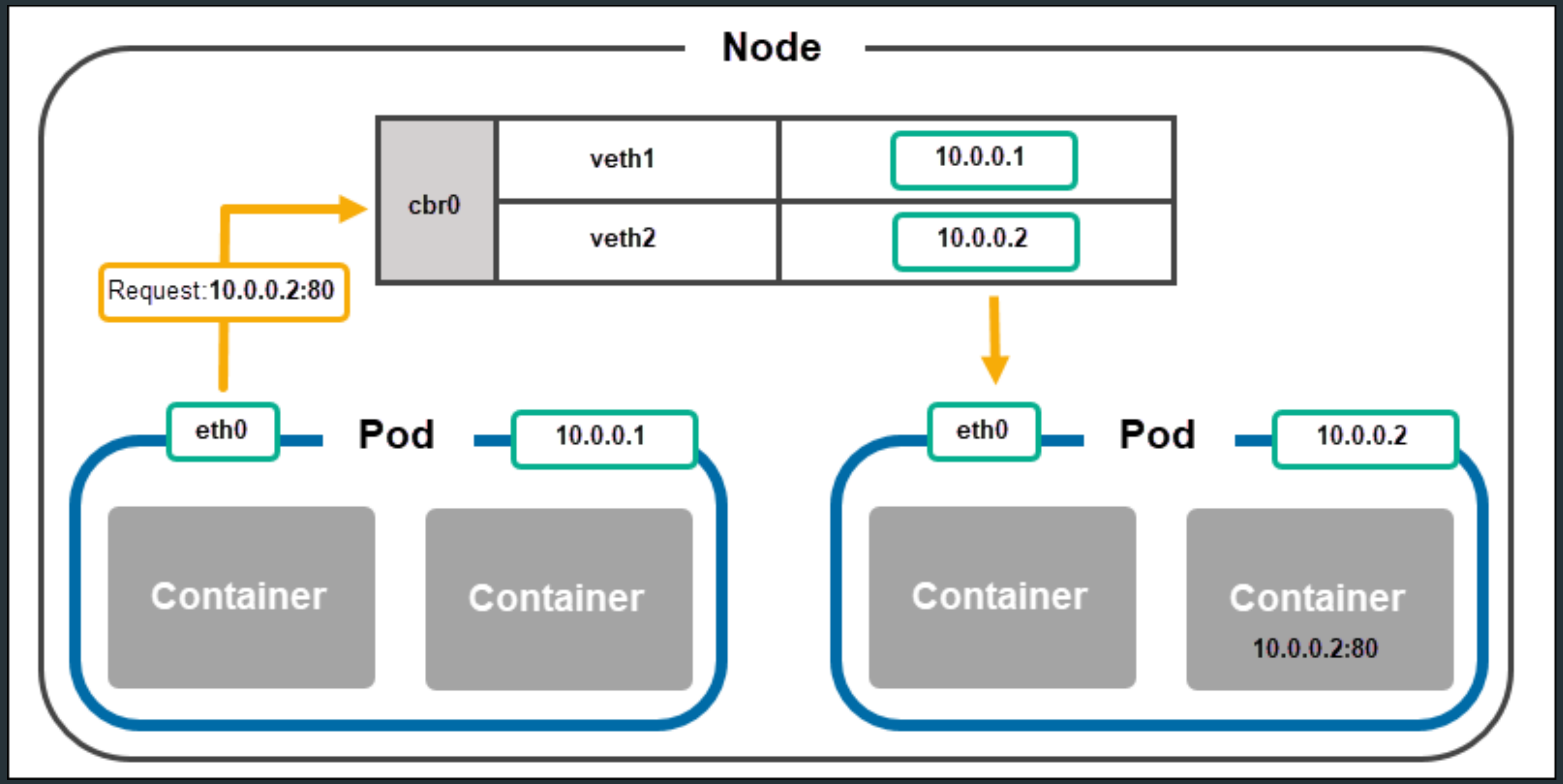
Kubernetes Components

- The Kubernetes architecture is made up of the following components:
  - Nodes
  - Pods
  - Services
  - Controllers
  - Schedulers
  - API Server
  - etcd
  - Kubelet
  - Kubectl



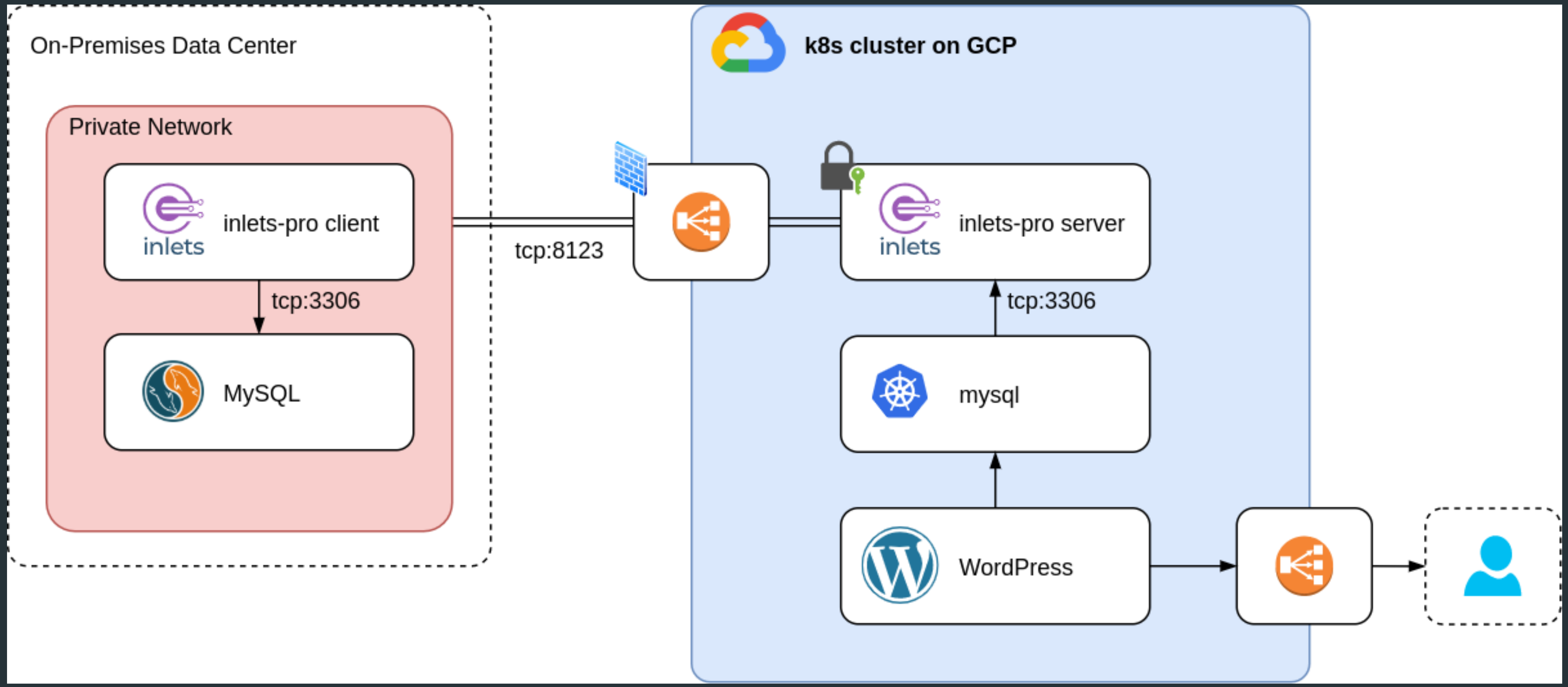
Kubernetes Communication

- Kubernetes components communicate with each other using a variety of protocols, including:
    - REST: The API Server uses the REST protocol to expose resources to clients.
    - gRPC: Controllers and kubelets use the gRPC protocol to communicate with each other.
    - etcd: etcd uses the Raft protocol to store configuration data.



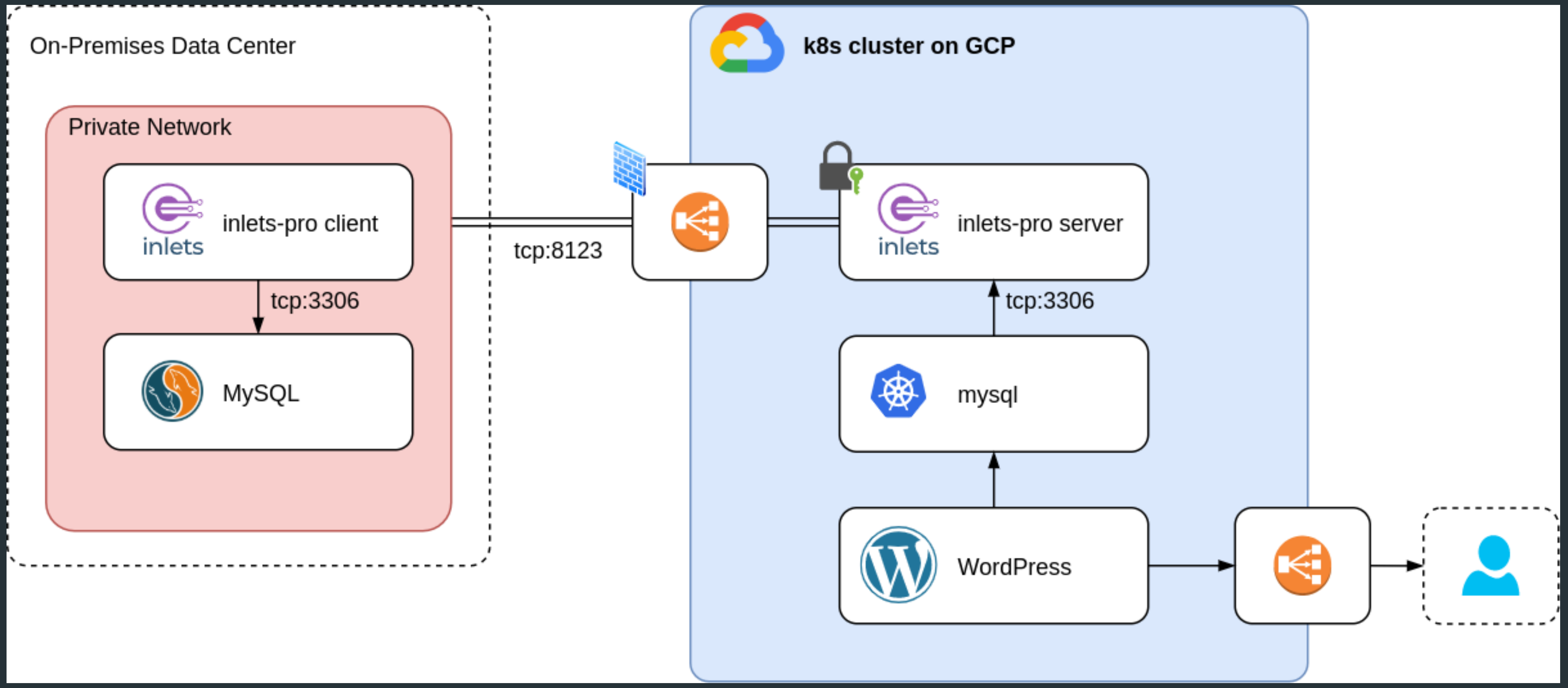
Setting Up a Kubernetes Cluster

- On-Premises
  - Are typically set up on a dedicated set of machines.
  - Offer the highest level of control and security.
  - Can be more expensive to set up and maintain than other types of clusters.



Setting Up a Kubernetes Cluster

- Cloud-Based
  - Are hosted by a cloud provider.
  - Typically more scalable and easier to manage than on-premises clusters.
  - Can be more expensive than on-premises clusters, depending on the provider and the features you need.



Setting Up a Kubernetes Cluster

- Managed
  - Provided by a third-party vendor.
  - Offer a turnkey solution for deploying and managing Kubernetes clusters.
  - Can be more expensive than self-managed clusters, but they can save you time and hassle.

Lecture outcomes

- Containers
- Virtualization
- Docker
- Kubernets