Containerization

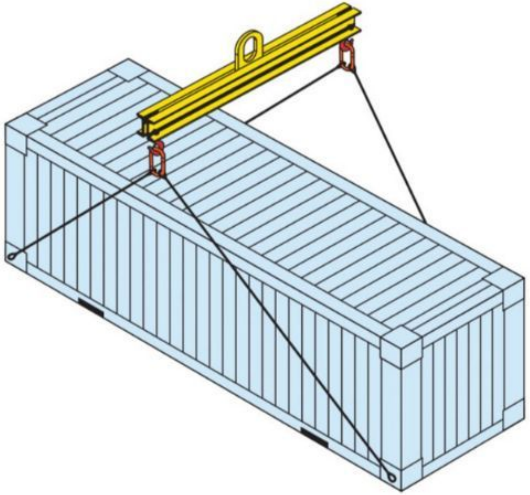
Learning Objectives

The learning objectives are to

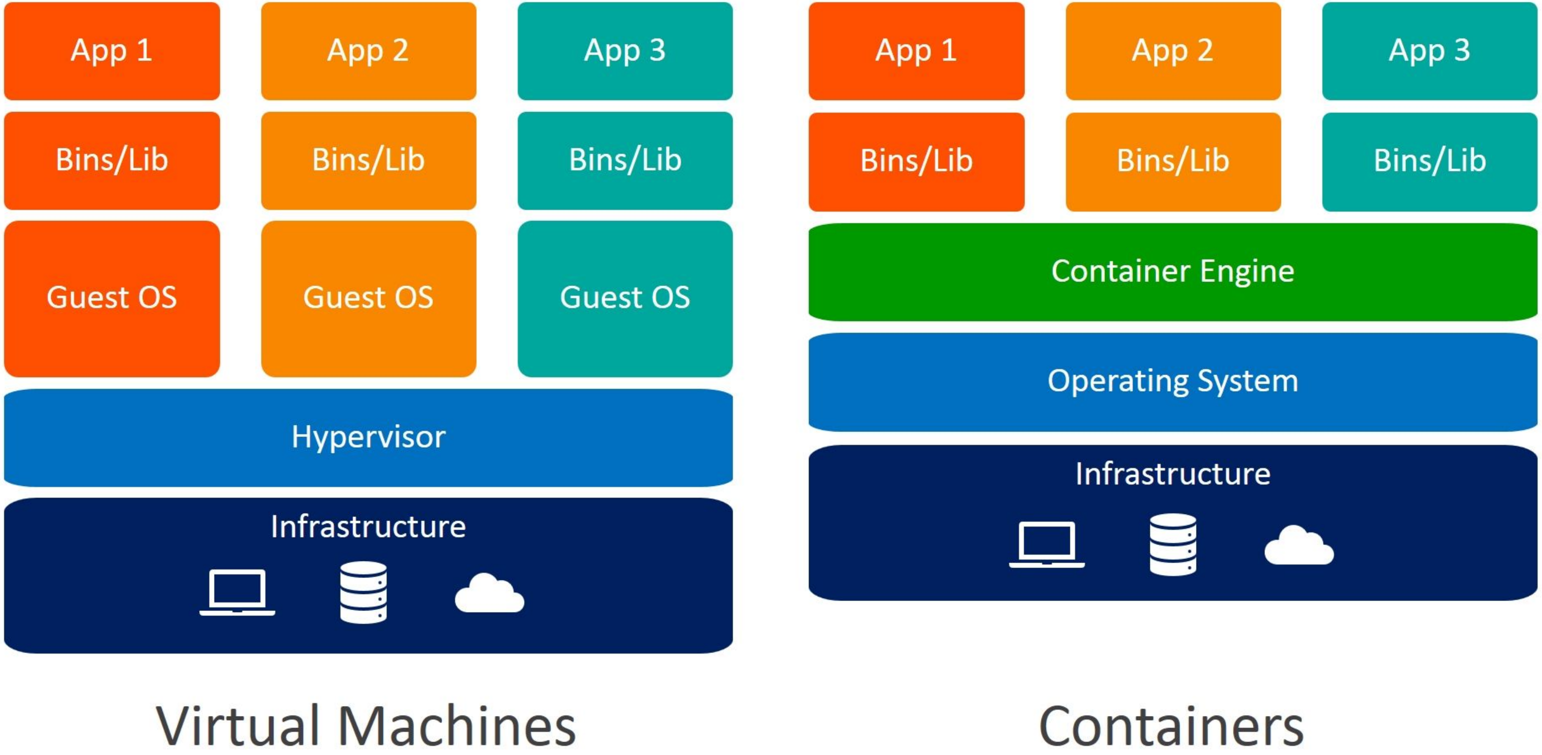
- Understand the concept of containerization and its significance in DevOps.
- Differentiate containers from traditional virtualization.
- Understand Docker's architecture, including images and containers.
- Demonstrate how to install Docker and run a basic container.
- To explain the feature and need of Container orchestration platform

Definition

- A lightweight form of virtualization that allows you to ***package and run applications and their dependencies in isolated, self-sufficient environments called containers.***
- Containers provide a consistent and efficient way to package, distribute, and execute software across different computing environments, such as ***development, testing, and production systems.***
- They encapsulate an application, its code, libraries, and runtime components, ensuring that it runs reliably and consistently across various platforms.



Container vs Virtual Machine



App 1

App 2

App 3

Bins/Lib

Bins/Lib

Bins/Lib

Guest OS

Guest OS

Guest OS

Hypervisor

Infrastructure

Virtual Machines

App 1

App 2

App 3

Bins/Lib

Bins/Lib

Bins/Lib

Container Engine

Operating System

Infrastructure

Containers

Container vs Virtual Machine

	Containerization	Virtualization
Level of Abstraction	Containers operate at the application layer. They encapsulate an application and its dependencies, running as isolated processes on a shared operating system kernel. Containers share the host OS resources, making them lightweight and efficient.	Traditional virtualization uses a hypervisor to emulate an entire operating system, including its kernel. Each virtual machine (VM) runs a complete OS instance. This approach is heavier in terms of resource usage compared to containers.

	Containerization	Virtualization
Isolation	Containers provide process-level isolation, meaning that they are isolated from each other at the application and process level. However, they share the same kernel, which can be a potential security concern if not properly configured.	**VMs provide strong isolation because each VM runs its own full-fledged operating system. They are isolated at both the application and kernel levels, offering higher security but at the cost of increased resource usage.**

Container vs Virtual Machine

	Containerization	Virtualization
Resource Overhead	Containers have minimal resource overhead because they share the host OS's kernel. This makes them highly efficient in terms of memory and CPU usage. Multiple containers can run on a single host with minimal resource wastage..	**VMs have more significant resource overhead due to the emulation of complete OS instances. Each VM includes its kernel, which consumes more memory and CPU resources.**

Container vs Virtual Machine

	Containerization	Virtualization
Portability	Containers are highly portable because they encapsulate applications and their dependencies. Container images can run consistently across different environments, making them ideal for microservices architectures and DevOps practices.	**VMs are less portable due to their larger size and the need for compatibility with specific hypervisors. Moving VMs between different virtualization platforms can be challenging.**

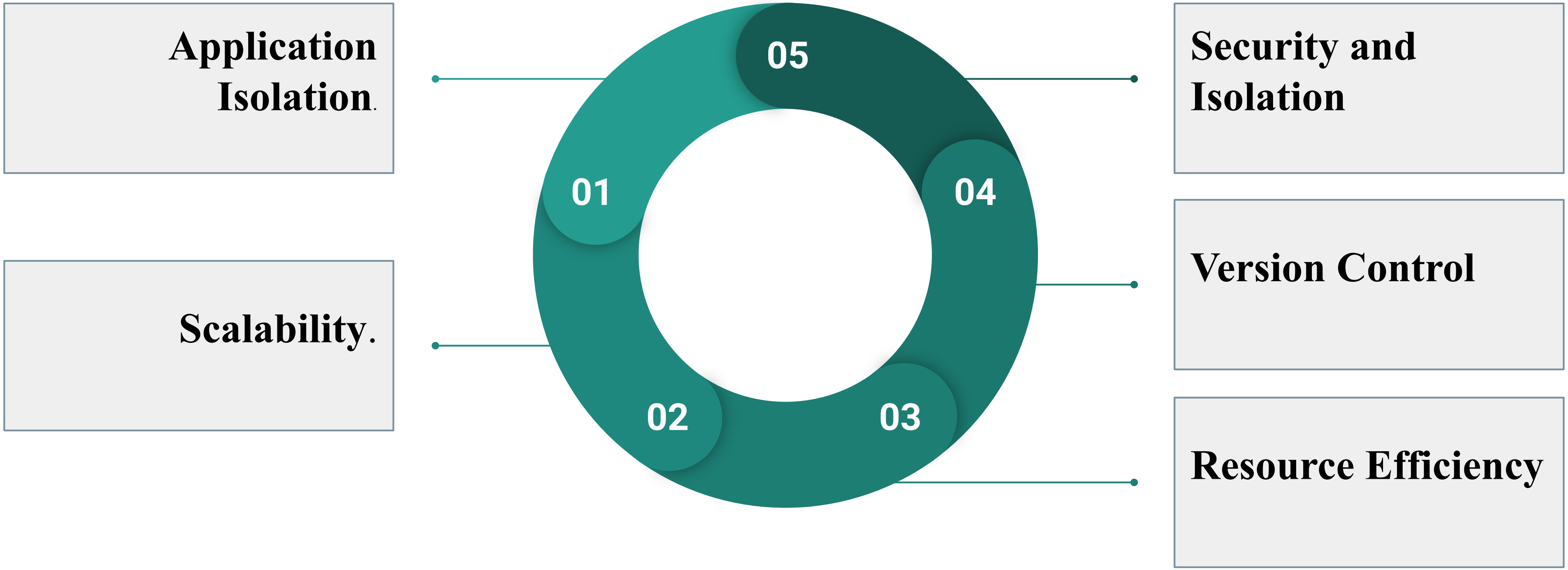
Container vs Virtual Machine

	Containerization	Virtualization
Boottime	Containers start and stop quickly, often in a matter of seconds, making them suitable for dynamic workloads and rapid scaling.	VMs have longer boot times since they need to load an entire OS. Starting a VM can take minutes, which is less suitable for fast-scaling applications.

Container vs Virtual Machine

	Containerization	Virtualization
Management & Orchestration	Containers are typically managed and orchestrated using container orchestration platforms like Kubernetes, Docker Swarm, and container runtimes like Docker. These tools provide automation for deploying, scaling, and managing containers.	VMs are typically managed using virtualization management tools, and their orchestration often involves solutions like VMware vSphere or Microsoft Hyper-V.

Significance of containerization to DevOps



Application Isolation.

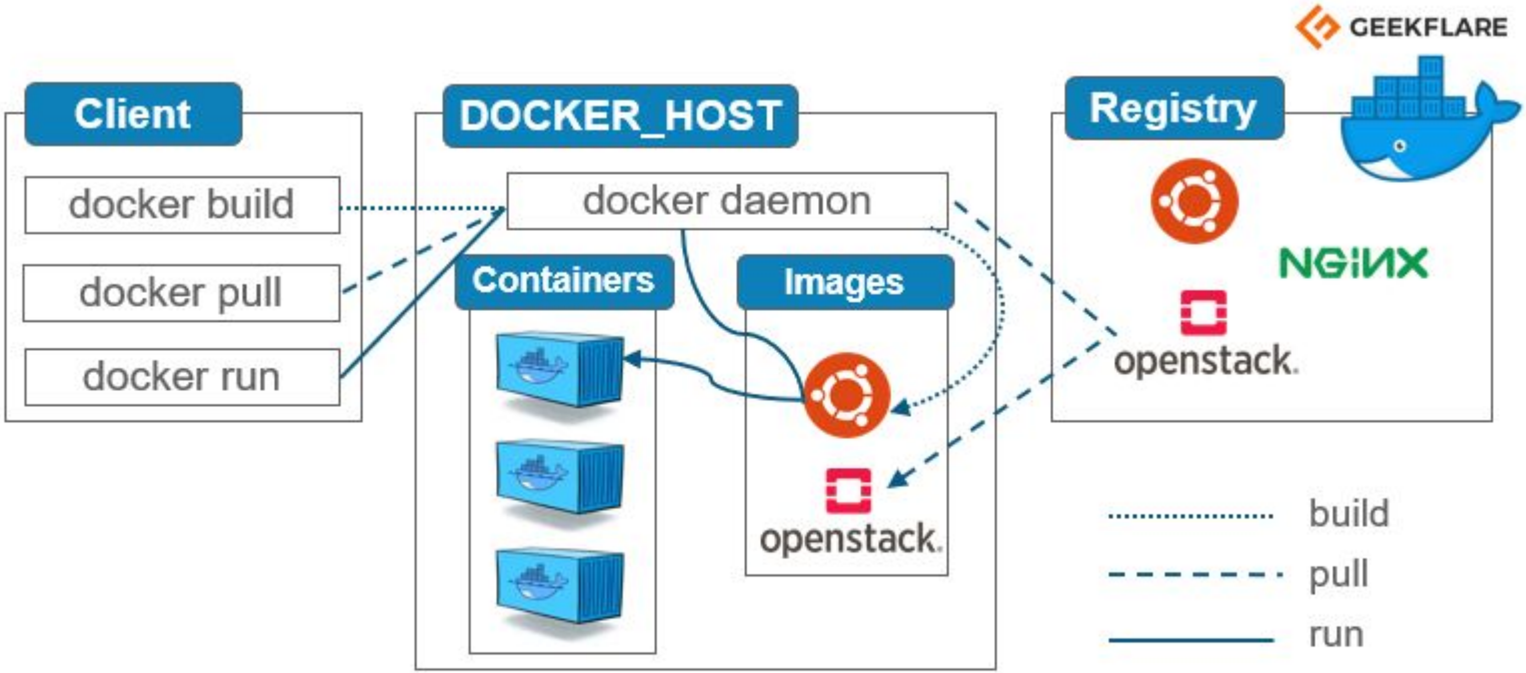
Scalability.

Security and Isolation

Version Control

Resource Efficiency

Docker Architecture



Docker Architecture

Docker Daemon	It is a long-running background process that manages Docker containers on a host machine. It is responsible for building, running, and maintaining containers.
Docker Client:	User interface either CLI/GUI
Docker Images:	Docker images are read-only templates that define an application, its code, libraries, and runtime environment.
Docker Containers	Containers are instances created from Docker images. They are isolated, runnable environments that encapsulate applications and their dependencies.

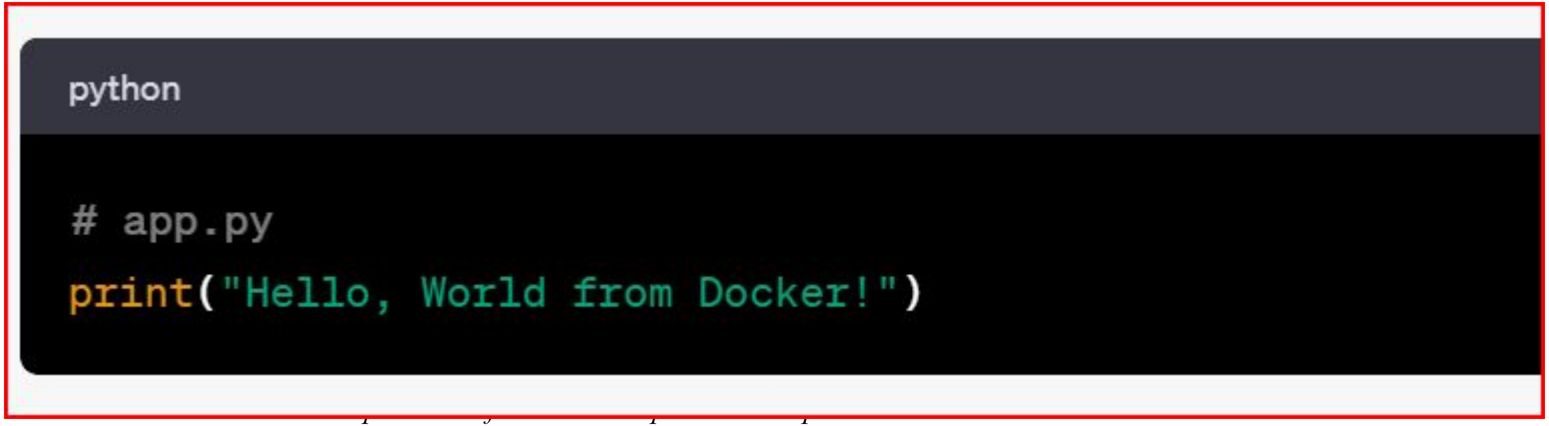
Docker Architecture

Docker Registry	Docker registries are centralized repositories for storing and distributing Docker images. The most well-known registry is Docker Hub, but you can also set up private registries.
Docker Networking	Docker provides networking capabilities that allow containers to communicate with each other and the external world. Containers can be connected to user-defined networks for isolation and flexibility.
Docker Volumes	Docker volumes are mechanisms for persisting data generated and used by containers. They are separate from the container's filesystem and can be mounted within containers.

Dockerize a Python Application

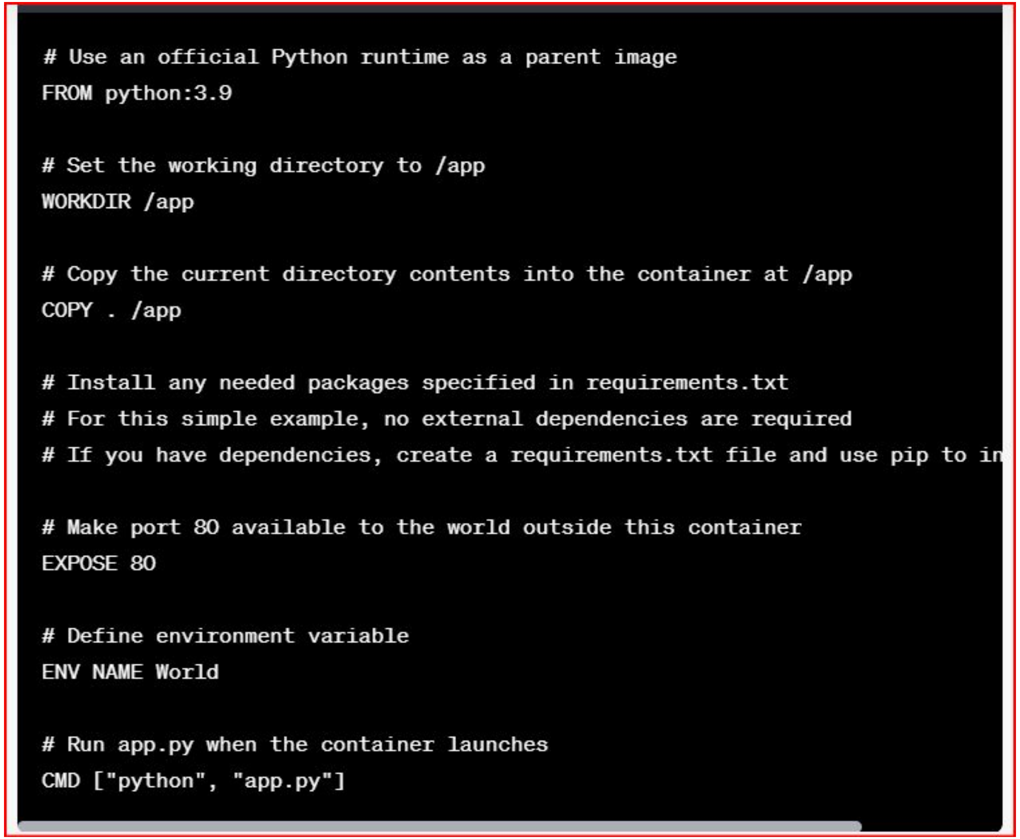
1 Install Docker Desktop on your machine

2 Develop Python Application


python

```python
# app.py
print("Hello, World from Docker!")
```

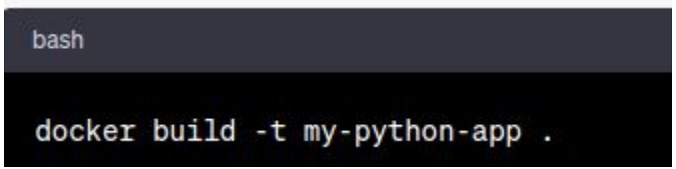
3 Create a Dockerfile



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

# Set the working directory to /app
WORKDIR /app

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

# Install any needed packages specified in requirements.txt
# For this simple example, no external dependencies are required
# If you have dependencies, create a requirements.txt file and use pip to in

# 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"]
```

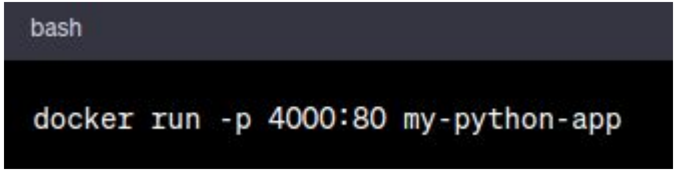
Dockerize a Python Application

4 Build the Docker Image

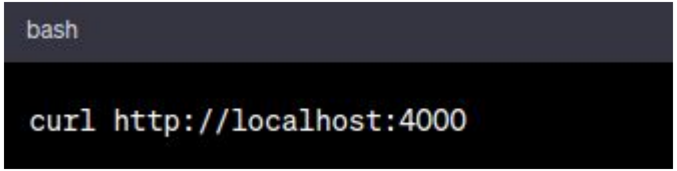


```bash
docker build -t my-python-app .
```

5-6 Run the Docker Container and Access Your Application



```bash
docker run -p 4000:80 my-python-app
```

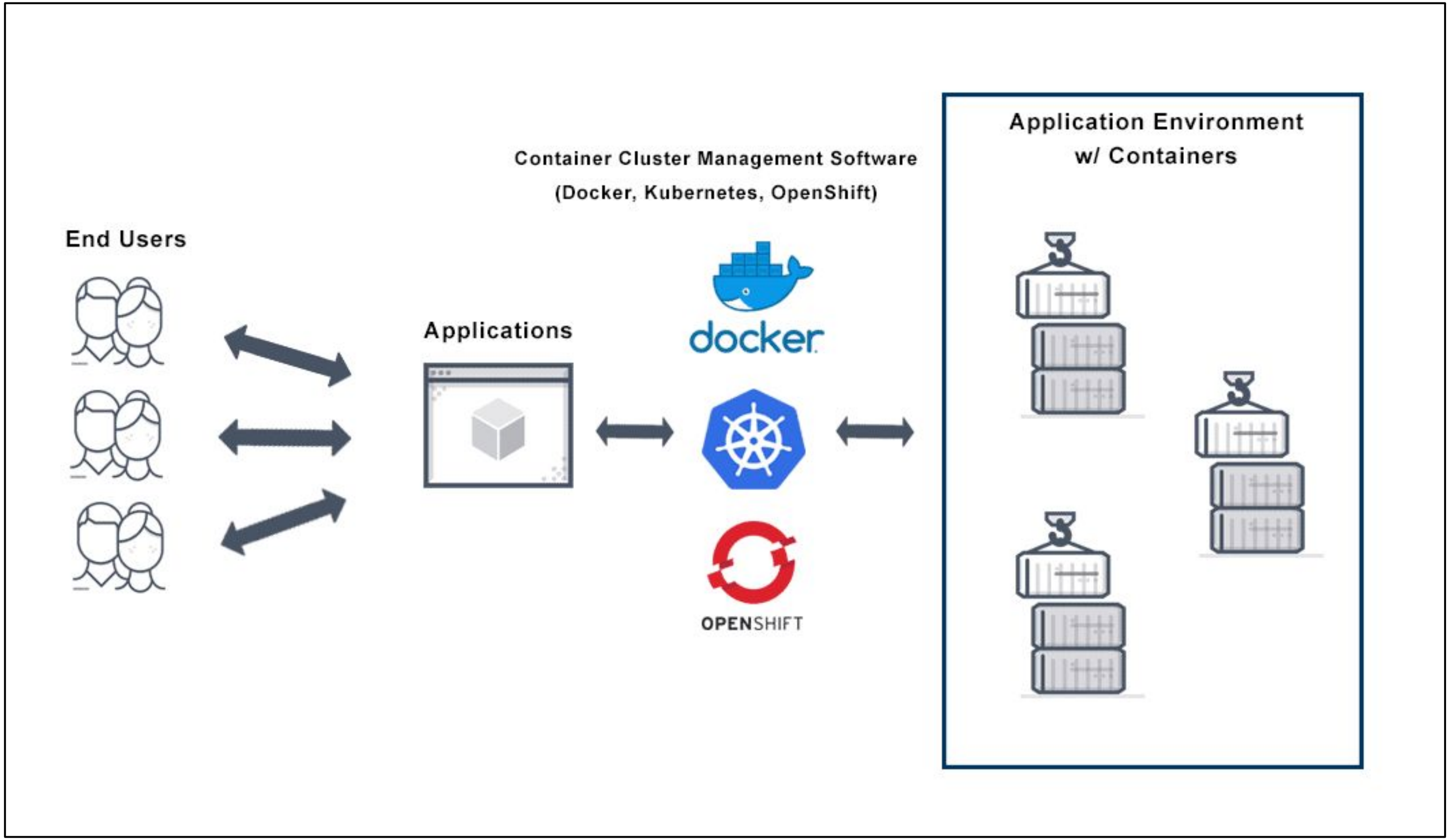


```bash
curl http://localhost:4000
```

Kubernetes:

A Container Orchestration Platform

Why do we need Kubernetes?



End Users

Applications

Container Cluster Management Software (Docker, Kubernetes, OpenShift)

Application Environment w/ Containers

Why do we need Kubernetes?

- Container Technology allows use to design a large monolithic application into smaller microservices which can be accessed through its public API
- Containers are basically small computational units.
- We need a OS like control unit to manage containers and interactions among them.
- Kubernetes is one such control unit/ container orchestration platform.

Why do we need Kubernetes?

- Container technology allows use to decompose a large monolithic application into smaller microservices which can be accessed through its public API
- Containers are basically small computational units.
- We ***need a OS like control unit to manage containers and interactions among them.***
- ***Kubernetes*** is one such platform controlling container deployment, scaling, and interactions among them.

What is Kubernetes?

- Kubernetes is a open source software tool to manage container workload
- It operates at container level (not hardware) and controls deployment, scaling and management of containers.
- It works along with docker.
-

Differences between Docker and Kubernetes

Docker		Kubernetes
Docker is a platform that enables developers to package, distribute, and run applications as lightweight containers.	**Definition**	Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications.

Differences between Docker and Kubernetes

Docker		Kubernetes
Docker follows a client-server architecture. The Docker daemon runs as a background process (server), and the Docker client communicates with it via a REST API.	**Architecture**	Kubernetes follows a master-node architecture. The master node manages the cluster, and worker nodes (minions) execute containers.

Differences between Docker and Kubernetes

Docker		Kubernetes
Ideal for local development, testing, and packaging applications into containers.	**Use Case**	Suited for managing containerized applications at scale in production environments.

Differences between Docker and Kubernetes

Docker		Kubernetes
Docker manages resources at the container level, allowing you to specify resource constraints (CPU, memory) for individual containers.	**Resource Management**	Kubernetes manages resources at the pod level (a group of containers), enabling the coordination of resources between containers within a pod.

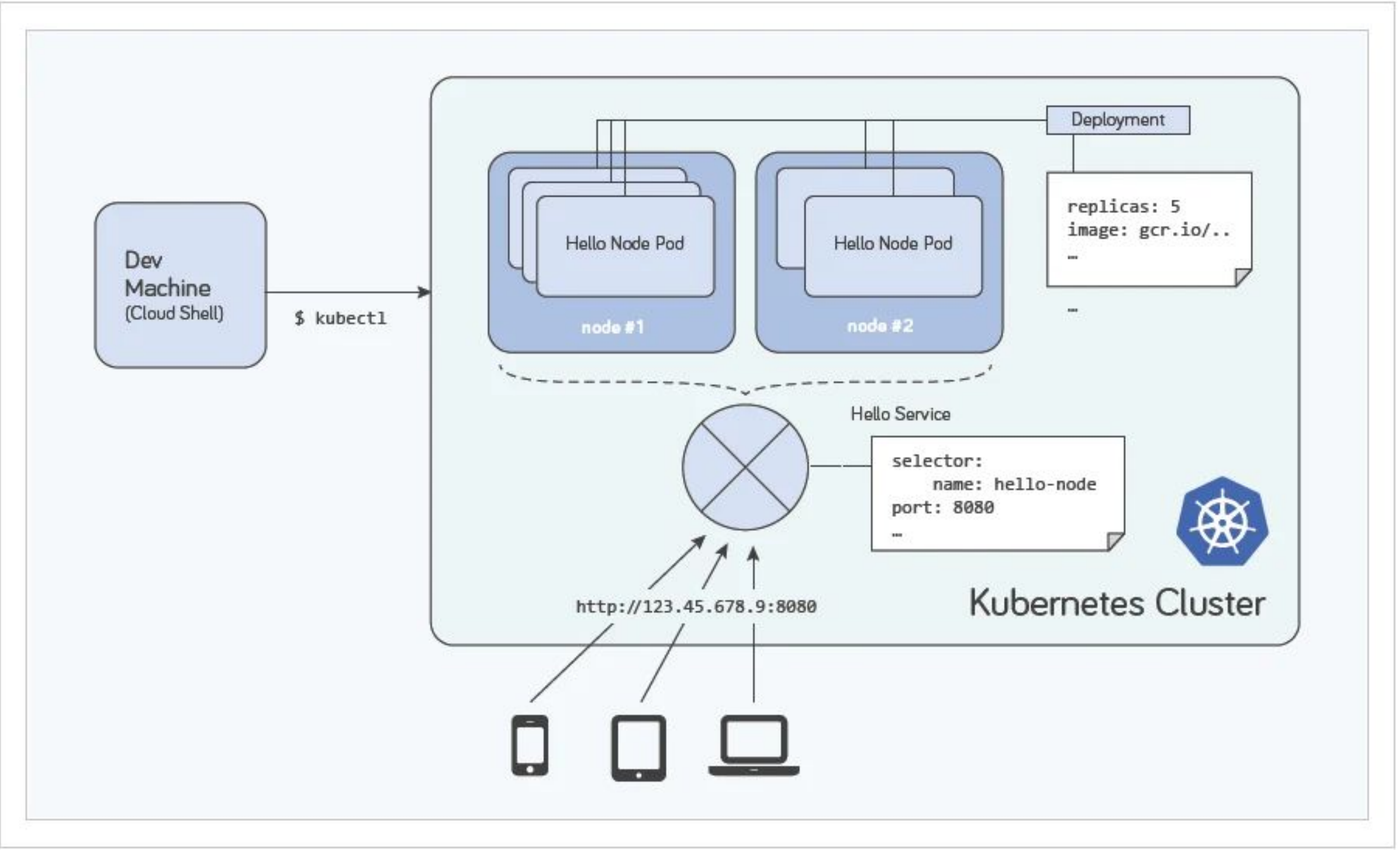
Differences between Docker and Kubernetes

Docker		Kubernetes
Networking between containers is straightforward, especially within the same Docker host.	**Networking**	Kubernetes has a more sophisticated networking model, allowing for communication between containers across nodes.

Differences between Docker and Kubernetes

Docker		Kubernetes
Scaling is typically done manually or through tools like Docker Swarm for orchestration	**Orchestration and Scaling**	Kubernetes provides advanced orchestration features like automated scaling, load balancing, rolling updates, and self-healing.

Main Architectural Elements in Kubernetes



```
replicas: 5
image: gcr.io/..
...
```

```
selector:
    name: hello-node
port: 8080
...
```

Main Architectural Elements in Kubernetes

Element	**Definition**	**Use Case**
Pods	A Pod is the smallest unit in the Kubernetes object model. It represents a single instance of a ***running process*** in a cluster. Pods can contain one or more containers sharing the same network namespace, storage, and have a unique IP address.	Pods are used to deploy and manage individual units of an application or microservices architecture.

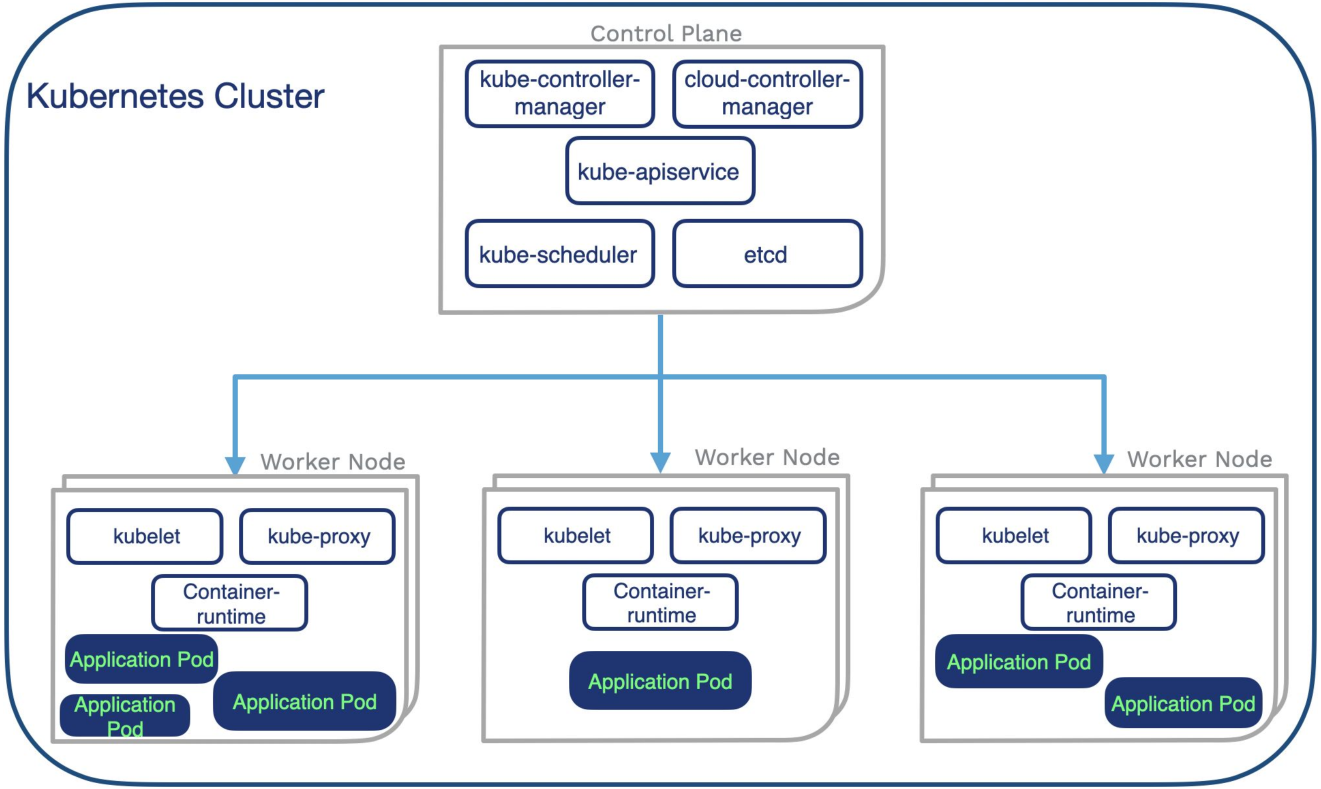
Main Architectural Elements in Kubernetes

Element	**Definition**	**Use Case**
Services	A Service is an abstraction that defines a logical set of Pods. Services provide stable endpoints for applications to communicate with, abstracting away the underlying Pod instances. Services enable load balancing and service discovery.	Services are used to enable communication between different parts of an application or between different applications within a Kubernetes cluster.

Main Architectural Elements in Kubernetes

Element	**Definition**	**Use Case**
Deployments	A Deployment is a higher-level abstraction that enables declarative updates to applications. They handle updates, rollbacks, and scaling automatically, providing a declarative way to manage application deployments.	Deployments are commonly used to manage the deployment and scaling of applications. They abstract away the complexity of directly managing Pods.

Components in Kubernetes Cluster



Kubernetes Cluster

Control Plane

kube-controller-manager

cloud-controller-manager

kube-apiservice

kube-scheduler

etcd

Worker Node

kubelet

kube-proxy

Container-runtime

Application Pod

Application Pod

Application Pod

Worker Node

kubelet

kube-proxy

Container-runtime

Application Pod

Worker Node

kubelet

kube-proxy

Container-runtime

Application Pod

Application Pod

Main Architectural Elements in Kubernetes

Element	**Definition**
Master Node	● The master node is responsible for managing the overall state of the cluster. It acts as the control plane, making decisions about the cluster (scheduling, scaling, etc.) and responding to events from nodes. ● Key components on the master node include the Kubernetes API server, controller manager, scheduler, and etcd.

Main Architectural Elements in Kubernetes

Element	**Definition**
Worker Node	● Worker nodes, or minions, are the machines where containers are actually deployed and run. Each worker node has a container runtime (such as Docker) for managing containers. ● The main component on a worker node is the kubelet, which communicates with the master node and manages containers on the node.

Main Architectural Elements in Kubernetes

Element	Definition
etcd	● **etcd** is a distributed key-value store that is used to store the configuration data of the cluster. It serves as the cluster's source of truth for information about the state of the cluster.

Quiz time

Containers encapsulate

1. Code
2. Runtime Components
3. Libraries needed to execute code
4. All of the above

Quiz time

Which of the following statement TRUE about virtualization and containerization

1. Containers provides process level isolation and virtualization provide os -level isolation
2. Containers provides OS- level isolation and virtualization provide process level isolation
3. Both provide process-level isolation
4. Both provide OS-level isolation

Quiz time

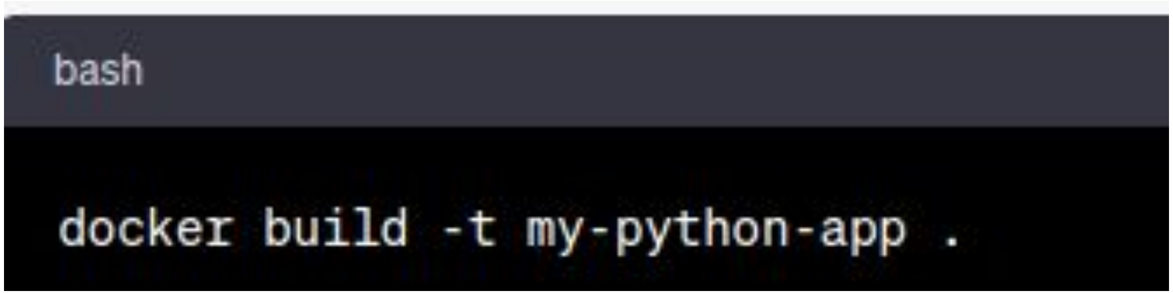
Docker images are read-only templates that define an application, its code, libraries, and runtime environment.

1. TRUE
2. FALS

Quiz time

Following docker command

1. Creates docker image file
2. Tests the python code before creating image files
3. Loads image file
4. Runs image file



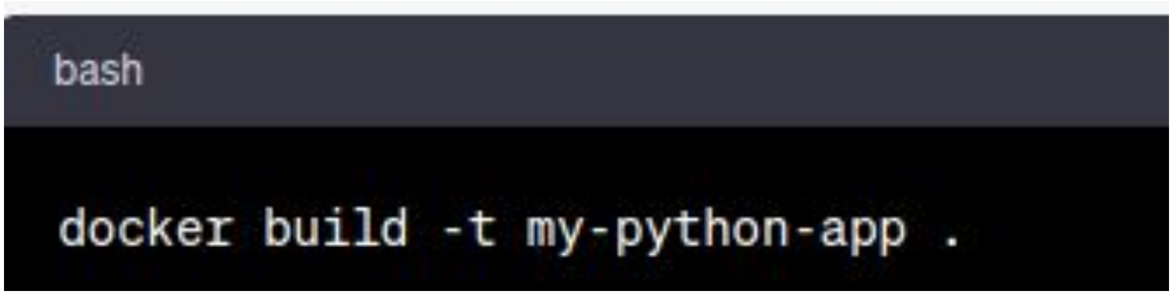
bash

```bash
docker build -t my-python-app .
```

Quiz time

Following docker command

1. Creates docker image file
2. Tests the python code before creating image files
3. Loads image file
4. Runs image file



```bash
docker build -t my-python-app .
```

Quiz time

A containerized application in Kubernetes consists of

1. Pods, Services, Deployments
2. Code, Test cases, deployments
3. Worker node, master node, cluster node
4. Client, server, databases