Docker Is a platform that enables developers to package, distribute, and run applications as lightweight, portable containers. These containers encapsulate the application along with its dependencies, libraries, and configuration files, ensuring consistency and reproducibility across different environments. Docker uses operating-system-level virtualization to create and manage containers, allowing applications to run in isolated environments without the overhead of full virtual machines.

Differentiate between Containers vs. virtual machines

Containers and virtual machines (VMs) are both technologies used to virtualize computing resources, but they operate at different levels of abstraction and have distinct characteristics. Here's a comparison differentiating between containers and virtual machines:

Abstraction Level:

Virtual Machines (VMs): VMs virtualize the entire hardware stack, including the operating system (OS), allowing

multiple OS instances to run on a single physical machine. Each VM runs its own complete OS and maintains its own kernel, which can lead to overhead in terms of resource consumption.
Containers: Containers virtualize at the OS level, enabling multiple isolated user-space instances (containers) to run on a single OS kernel. Containers share the host OS kernel, leading to lower overhead compared to VMs.
Resource Utilization:

VMs: VMs require a full OS for each instance, including its own memory allocation, disk space, and CPU cycles. This can lead to higher resource utilization and overhead.
Containers: Containers share the host OS kernel and only include the necessary libraries and binaries to run the application, resulting in more efficient resource utilization and faster startup times compared to VMs.

Isolation:

VMs: VMs provide strong isolation between different instances since each VM has its own OS kernel. This isolation makes VMs suitable for running applications with different OS requirements or security concerns.
Containers: Containers provide lightweight isolation at the application level. While they offer less isolation compared to VMs, they are often sufficient for most use cases and provide faster performance due to the reduced overhead.

Portability:

VMs: VMs are less portable compared to containers because they include the entire OS stack, making them larger and more complex to transfer between different environments.
Containers: Containers are highly portable since they encapsulate only the application and its dependencies. This makes it easier to move containers between different

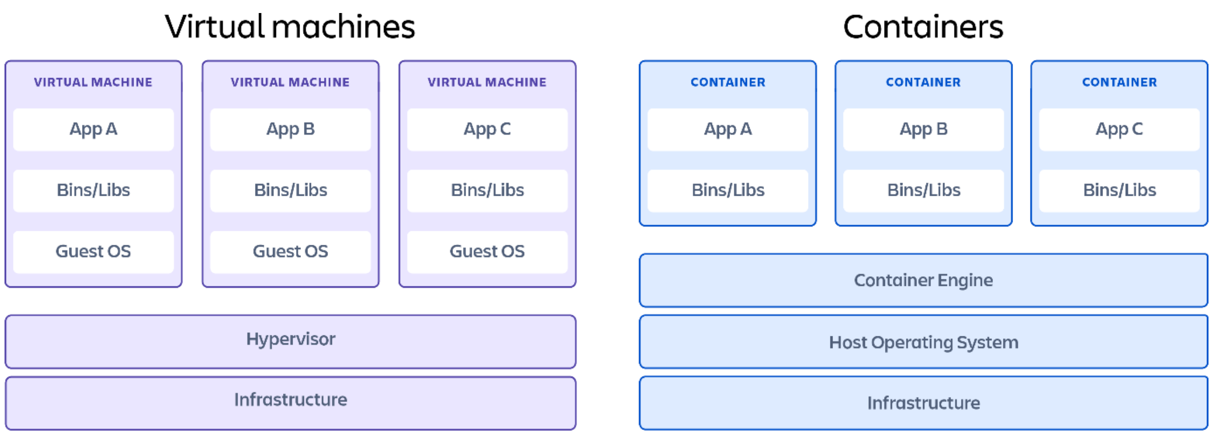
environments, such as development, testing, and production, without worrying about underlying OS differences.

Performance:

VMs: VMs may suffer from performance overhead due to the emulation of hardware and the presence of multiple OS kernels.
Containers: Containers generally offer better performance than VMs due to their lightweight nature and reduced overhead. They share the host OS kernel and do not require emulation of hardware, resulting in faster startup times and lower resource consumption.

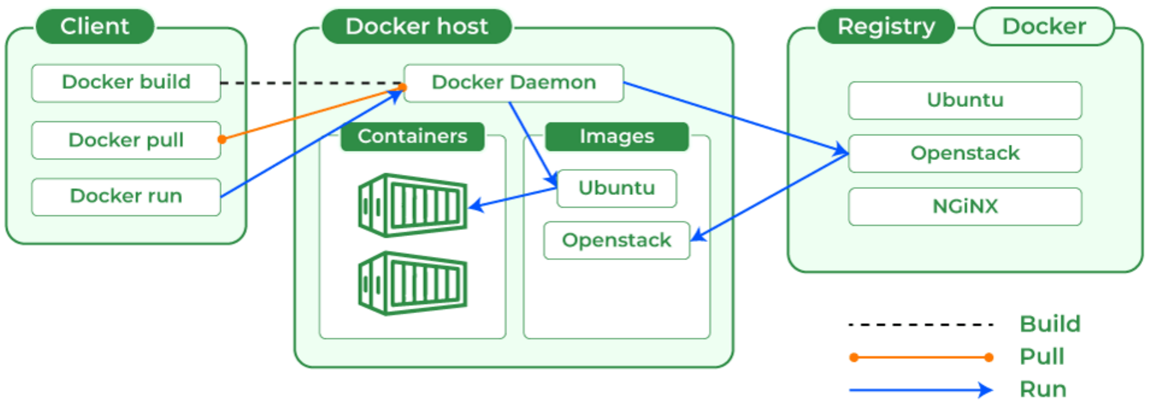
Use Cases:

VMs: VMs are suitable for scenarios where strong isolation, compatibility with different OS environments, or running legacy applications is required.
Containers: Containers are well-suited for microservices architectures, continuous integration/continuous deployment (CI/CD) pipelines, cloud-native applications, and scenarios where scalability, portability, and fast deployment are essential.



Architecture of Docker:

Docker makes use of a client-server architecture. The Docker client talks with the docker daemon which helps in building, running, and distributing the docker containers. The Docker client runs with the daemon on the same system or we can connect the Docker client with the Docker daemon remotely. With the help of REST API over a UNIX socket or a network, the docker client and daemon interact with each other.



Docker - Container and Host:

Docker is a platform that allows developers to package and distribute applications along with all of their dependencies into lightweight containers. These containers can then be run on any system that has Docker installed, providing consistency across different environments.

Container:

- A container is a lightweight, standalone, executable software package that includes everything needed to run a piece of software, including the code, runtime, libraries, and dependencies.

- Containers are isolated from each other and from the underlying host system, ensuring that they run consistently regardless of where they are deployed.
- Docker containers are created from Docker images, which are essentially read-only templates that contain the instructions for creating a container.
- Containers can be started, stopped, moved, and deleted independently of each other.

Host:

- The host is the physical or virtual machine on which Docker is installed and running.
- It provides the necessary infrastructure for running Docker containers, including the Docker Engine, which is responsible for creating and managing containers.
- The host system provides resources such as CPU, memory, storage, and network connectivity to the containers.
- Docker containers run directly on the host's operating system, but they are isolated from it using containerization technologies such as namespaces and control groups.

Docker- Engine:

Docker Engine is the core component of Docker, responsible for creating and managing Docker containers. It provides an environment for running containerized applications and includes several key functionalities:

1. Container Runtime: Docker Engine uses a container runtime to create and manage containers. The default container runtime used by Docker Engine is called containerd, which handles container lifecycle

management, image distribution, and storage management.

2. Image Management: Docker Engine allows users to create, pull, push, and manage Docker images.

3. Networking: Docker Engine provides networking capabilities for containers, allowing them to communicate with each other and with external networks.

4. Storage: Docker Engine allows users to manage storage for containers, including volume mounts for persistent data storage and container storage limits.

5. Security: Docker Engine provides several security features to protect containers and the Docker environment.

Docker- Cloud

Docker Cloud was a hosted service provided by Docker, Inc. It offered a platform for deploying, managing, and scaling containerized applications using Docker containers. Docker Cloud provided a user-friendly interface for managing Dockerized applications across multiple cloud providers and on-premises infrastructure.

1. Application Deployment: Docker Cloud allowed users to deploy containerized applications with just a few clicks. Users could specify the number of containers to run, configure environment variables, set up networking, and define resource constraints.

2. Multi-Cloud Support: Docker Cloud supported deployment across multiple cloud providers, including Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). This allowed users to deploy applications to their preferred cloud provider or mix and match providers as needed.

3. Integration with Docker Hub: Docker Cloud seamlessly integrated with Docker Hub, the official registry for Docker images. Users could easily pull and push Docker images from Docker Hub, making it simple to deploy applications using pre-built images or publish custom images.

4. Automated Builds: Docker Cloud provided automated build capabilities, allowing users to automatically build Docker images whenever changes were pushed to a connected source code repository (e.g., GitHub, Bitbucket). This facilitated continuous integration and deployment (CI/CD) workflows for containerized applications.

5. Scalability and High Availability: Docker Cloud offered features for scaling applications horizontally by adding or removing container instances based on demand. It also provided built-in support for high availability, ensuring that applications remained accessible even in the event of infrastructure failures.

6. Monitoring and Logging: Docker Cloud provided monitoring and logging capabilities for containerized applications, allowing users to track resource usage, view container logs, and set up alerts for performance or availability issues.

7. Security and Access Control: Docker Cloud included features for managing user access and permissions, ensuring that only authorized users could deploy or modify applications.

Installing Docker on Windows
To install Docker on Windows, you can follow these steps:

1. Check System Requirements:

2. Download Docker Desktop:

3. Install Docker Desktop:

4. Start Docker Desktop:

5. Sign in (Optional):

6. Verify Installation:

7. Configure Docker Settings (Optional):

8. Start Using Docker:

Basic Docker Commands:

1. **docker version**:
   - This command displays the version of Docker Engine installed on your system, as well as the version of the client.

2. **docker info**:

- This command provides detailed information about the Docker installation.

3. docker pull \<image>:
For example:

```
docker pull nginx
```

4. docker images:
- This command lists all Docker images stored on your local system.

5. docker run \<image>:
For example:

```
docker run nginx
```

6. **docker ps**:
To view all containers, including stopped ones, use the `-a` flag:

```
docker ps -a
```

7. **docker stop \<container>**:
For example:

```
docker stop my_container
```

8. **docker start \<container>**:
For example:

```
docker start my_container
```

9. **docker rm \<container>**:

```
docker rm my_container
```

10. **docker rmi \<image>**:

```
docker rmi nginx
```

11. **docker exec -it \<container> \<command>**:

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

Working with Containers:

Working with containers in Docker involves various tasks such as creating, running, managing, and interacting with containers. Here's a guide on how to work with containers:

1. Create a Container:

For example:

```
docker run nginx
```

2. Run a Container:

For example:

```
docker start my_container
```

3. Stop a Container:

For example:

```
docker stop my_container
```

4. View Container Logs:

For example:

```
docker logs my_container
```

5. List Running Containers:

To view all containers, including stopped ones, use the `-a` flag:

```
docker ps
```

6. Delete a Container:

For example:

```
docker rm my_container
```

7. Inspect Container Details:

```
docker inspect my_container
```

8. Execute Commands in a Running Container:

For example:

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

9. Copy Files to/from a Container:

```
docker cp my_container:/path/to/file.txt /host/path
```

10. Pause and Unpause a Container:

To unpause it, use the `docker unpause` command.

Building a web server Docker file:
To build a Dockerfile for a basic web server, such as one serving static content or a simple web application, you typically need to:

1. Specify a base image.
2. Copy your web server files into the image.
3. Expose the necessary ports for accessing the web server.
4. Define any commands needed to start the web server.

Here's a simple example of a Dockerfile for setting up an Nginx web server serving static files:

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

# Set the working directory inside the container
WORKDIR /usr/share/nginx/html

# Copy the static files (HTML, CSS, JS, etc.) from the
host into the container
COPY . .

# Expose port 80 to allow outside access to the web
server
EXPOSE 80
```

```
# Define the command to start the Nginx web server
(this will start automatically when the container
starts)
CMD ["nginx", "-g", "daemon off;"]
```

In this Dockerfile:

- `FROM nginx:latest`:
Specifies the base image to use. We're using the latest version of the official Nginx image from Docker Hub.
- `WORKDIR /usr/share/nginx/html`: Sets the working directory inside the container where our web server files will be copied.
- `COPY . .`: Copies all files from the current directory on the host machine (where the Dockerfile is located) into the `/usr/share/nginx/html` directory inside the container.
- `EXPOSE 80`: Informs Docker that the container will listen on port 80 at runtime. This does not actually publish the port; it's just informational.

- `CMD ["nginx", "-g", "daemon off;"]`: Defines the command to start the Nginx web server when the container starts. We use `daemon off;` to ensure that Nginx runs in the foreground and doesn't detach from the container, which would cause the container to exit.

To build an image from this Dockerfile, navigate to the directory containing the Dockerfile and run:

```
docker build -t my-web-server .
```

This command will build a Docker image named `my-web-server` using the Dockerfile in the current directory (`.`).

After building the image, you can run a container based on it using:

~~~
```
docker run -d -p 8080:80 my-web-server
```
~~~

This command will start a container named `my-web-server` based on the image we just built, and it will map port 8080 on the host to port 80 on the container. You can then access the web server by visiting `http://localhost:8080` in your web browser.

Create an example for this:
let's create a simple example of a Docker container running a basic web server that serves a "Hello, World!" message. We'll use Python's built-in HTTP server module for simplicity.

1. Create a directory for your project:

```
bash
mkdir docker-example
cd docker-example
```

2. Create an `index.html` file with the "Hello, World!" message:

```
html
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-
width, initial-scale=1.0">
    <title>Hello, World!</title>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
</html>
```

3. Create a Dockerfile in the same directory:

```
Dockerfile
# Use the official Python image as the base image
FROM python:3.9-slim

# Set the working directory inside the container
WORKDIR /app

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

# Install any dependencies needed for your
application
# (In this case, we don't need any dependencies)

# Expose port 8000 to allow outside access to the web
server
EXPOSE 8000
```

```
# Define the command to start the HTTP server using Python
CMD ["python", "-m", "http.server", "8000"]
```

4. Build the Docker image:

```bash
docker build -t my-web-server .
```

5. Run a container based on the image:

```bash
docker run -d -p 8000:8000 my-web-server
```

6. Access the web server:

Open your web browser and go to `http://localhost:8000`. You should see the "Hello, World!" message displayed.