UNIT III – DEVOPS AND CI/CD CONCEPTS [9 hours]

What is DevOps? Why is it used in industry?,CI/CD – Continuous Integration and Deployment, Introduction to GitHub Actions / Jenkins, Introduction to Docker and Dockerfile, Automating build and test for an app

---

INTRODUCTION TO DOCKER

DOCKER

Docker is an open platform that uses operating-system-level virtualization to deliver software in packages called containers. Docker is an **open-source containerization platform** used to develop, package, ship, and run applications in **lightweight containers**. A Docker container includes the application code along with all its dependencies, libraries, and configuration files, ensuring that the application runs consistently across different environments. The key components are:

- Images: Read-only templates used to create containers, which include the application and all its dependencies.
- Containers: Running instances of an image, isolated from the host system and other containers, but sharing the host OS kernel.
- Docker Hub/Registry: A cloud service or private server for sharing and downloading popular Docker images.

Need for Docker

Before Docker, applications were deployed directly on physical servers or virtual machines, leading to various challenges such as dependency conflicts and environment inconsistencies.

Problems in Traditional Deployment

- **Environment inconsistency** is one of the major problems in traditional deployment. Applications often behave differently in development, testing, and production environments due to differences in operating systems, software versions, or system configurations. This leads to the common issue known as “it works on my machine”,

where an application runs successfully on a developer's system but fails in production, causing delays and unexpected errors.

- **Dependency conflicts** occur when multiple applications require different versions of the same software library or framework on a single system. In traditional deployment, dependencies are installed directly on the host machine, making it difficult to manage multiple versions simultaneously. Installing or upgrading one dependency may break another application, leading to instability and maintenance challenges.
- **Complex installation and configuration** is another drawback of traditional deployment. Applications often require lengthy setup procedures involving manual installation of software, libraries, environment variables, and configuration files. These manual steps increase the chances of human error and make the deployment process time-consuming, especially for new team members or system administrators.
- **Poor resource utilization** is commonly observed in traditional systems. Physical servers or virtual machines are usually dedicated to a single application, leaving CPU, memory, and storage resources underutilized. Virtual machines also require separate operating systems, which consume additional resources and increase infrastructure costs.
- **Slow deployment and startup time** is a significant issue in traditional deployment. Installing applications and their dependencies takes considerable time, and virtual machines can take several minutes to boot. As a result, application updates, patches, or scaling operations lead to delays and sometimes require downtime, reducing overall system efficiency.
- **Difficult scalability** limits the ability of traditional deployment models to handle changing workloads. Scaling an application requires manual provisioning of new servers or virtual machines and configuring them appropriately. This process is slow and inflexible, making it hard to respond quickly to sudden increases or decreases in user demand.
- **Lack of application isolation** creates stability and security concerns. Since multiple applications share the same operating system and system resources, a failure or

security breach in one application can affect others. This shared environment makes it difficult to ensure reliable and secure application execution.

- **Inconsistent testing and debugging** arises because test environments rarely match production environments exactly. Differences in system configurations or installed software can cause bugs that appear only after deployment. This makes debugging difficult and reduces confidence in testing results.
- **Difficult rollback and version management** is another challenge in traditional deployment. Reverting to a previous version of an application often requires uninstalling or reconfiguring software manually. Managing multiple versions of an application simultaneously is complex and time-consuming.
- **High maintenance overhead** increases operational costs and effort. Systems require frequent updates, patching, and monitoring to maintain performance and security. Over time, configuration drift occurs, where systems deviate from their original setup, making maintenance even more difficult.
- **Limited portability** restricts the movement of applications between different environments or platforms. Applications are tightly coupled with the underlying system, making migration to new servers or cloud platforms difficult and requiring reinstallation and reconfiguration.
- **Security challenges** are amplified in traditional deployment due to shared environments and manual security management. Applying security patches and enforcing application-specific security policies is complex, increasing the risk of vulnerabilities and system compromise.

How Docker Solves Traditional Deployment Problems

- **Docker packages applications with their dependencies**, which is one of its most important advantages. Using Docker images, an application is bundled along with all required libraries, frameworks, runtime environments, and configuration files. This eliminates dependency conflicts because each application carries its own dependencies instead of relying on the host system. As a result, applications run

independently without interfering with one another, even if they require different software versions.

- **Docker ensures environment consistency** by providing the same runtime environment across development, testing, and production. Since Docker containers run based on images, the application behaves identically regardless of where the container is deployed. This removes the "it works on my machine" problem and ensures reliable testing and deployment across different platforms and operating systems.
- **Docker enables faster application startup** compared to traditional virtual machines. Containers share the host operating system kernel instead of running a full guest OS, which significantly reduces startup time. As a result, containers can start in seconds, allowing developers to deploy, stop, and restart applications quickly, improving productivity and reducing downtime.
- **Docker provides better resource utilization** by allowing multiple containers to run on the same host system while sharing system resources efficiently. Unlike virtual machines, containers do not require separate operating systems, which reduces memory and CPU overhead. This lightweight nature allows more applications to run on the same hardware, lowering infrastructure costs and improving performance.
- **Docker makes scaling and deployment easy and efficient.** Containers can be quickly replicated to handle increased workloads, enabling horizontal scaling. With tools like Docker Compose and orchestration platforms, applications can be deployed, updated, or rolled back with minimal effort. This flexibility allows organizations to respond quickly to changing user demands and supports modern DevOps and microservices architectures.

Containerization

Containerization is a lightweight virtualization technique in which applications run in **isolated user spaces called containers**, sharing the host operating system kernel.

Key Characteristics

- **Each container runs independently**, meaning that applications inside one container are isolated from applications in other containers. Every container has its own file system, processes, network interfaces, and environment variables. This isolation ensures that a failure, crash, or security issue in one container does not affect other containers running on the same host system, thereby improving application stability and reliability.
- **Containers use fewer resources** than virtual machines because they share the host operating system kernel instead of running separate guest operating systems. Unlike virtual machines, which require their own OS and consume significant memory and CPU resources, containers are lightweight and efficient. This allows multiple containers to run on the same system with minimal overhead, resulting in better performance and reduced infrastructure costs.
- **Containers start and stop quickly** due to their lightweight nature. Since containers do not need to boot a full operating system, they can be started or stopped in a matter of seconds. This fast startup time is particularly useful for rapid application deployment, continuous integration and delivery (CI/CD), scaling applications, and quick recovery from failures.
- **Containers are portable across platforms**, meaning they can run consistently on different environments such as development machines, testing servers, cloud platforms, and production systems. As long as Docker or a compatible container runtime is available, the same container image can be executed without modification. This portability ensures consistent behavior across platforms and eliminates environment-related issues.

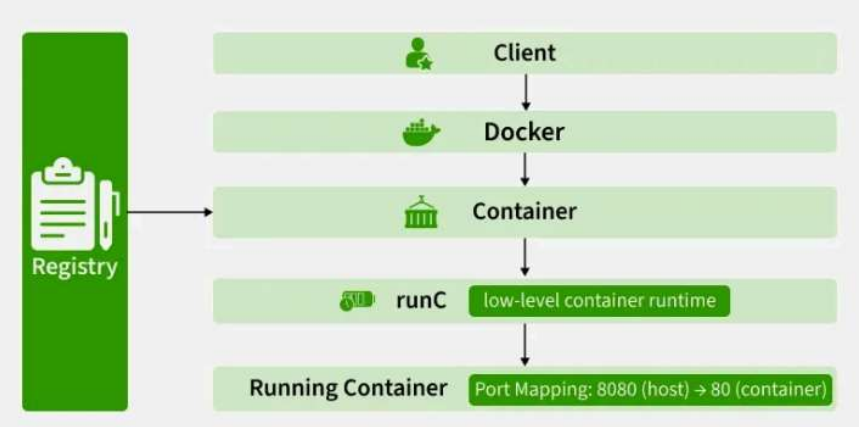
Docker vs Virtual Machine

Feature	Docker	Virtual Machine
Architecture	Shares host OS kernel	Has separate guest OS
Startup time	Seconds	Minutes

Resource usage	Low	High
Size	Lightweight	Heavy
Performance	Near native	Slower

Docker Architecture

Docker follows a client-server architecture. The Docker client communicates with a background process, the Docker Daemon, which does the heavy lifting of building, running, and managing your containers. This communication happens over a REST API, typically via UNIX sockets on Linux (e.g., /var/run/docker.sock) or a network interface for remote management.



The Core Architectural Model

- **Docker Client:** This is your command center. When you type commands like docker run or docker build, you're using the Docker Client.
- **Docker Host:** This is the machine where the magic happens. It runs the Docker Daemon (dockerd) and provides the environment to execute and run containers.

- **Docker Registry:** This is a remote repository for storing and distributing your Docker images.

This interaction forms a simple yet powerful loop: you use the **Client** to issue commands to the **Daemon** on the **Host**, which can pull images from a **Registry** to run as containers.

Core Components:

1. The Docker Daemon (dockerd):

The Docker Daemon is the persistent background process that acts as the brain of your Docker installation.

- It runs on the **Docker Host**.
- It listens for API requests from the Docker Client.
- It manages all **Docker objects**: images, containers, networks, and volumes.
- It can communicate with other daemons to manage Docker services in a multi-host environment (like a Docker Swarm cluster).

2. The Docker Client:

The Docker Client is the primary interface through which users interact with Docker. This is most commonly the Command Line Interface (CLI).

- It translates user commands like docker ps into REST API requests.
- These requests are sent to the Docker Daemon for processing.
- A single client can communicate with multiple daemons.

Common Commands:

- docker build: Builds an image from a Dockerfile.
- docker pull: Pulls an image from a registry.
- docker run: Creates and starts a container from an image.

3. The Docker Host

The Docker Host is the physical or virtual machine that provides the complete environment for executing and running containers. It comprises:

- The Operating System (and its kernel).
- The Docker Daemon.
- Images that have been pulled or built.

- Running Containers.
- Networks and Storage components.

4. The Docker Registry

A Docker Registry is a stateless, scalable storage system for Docker images.

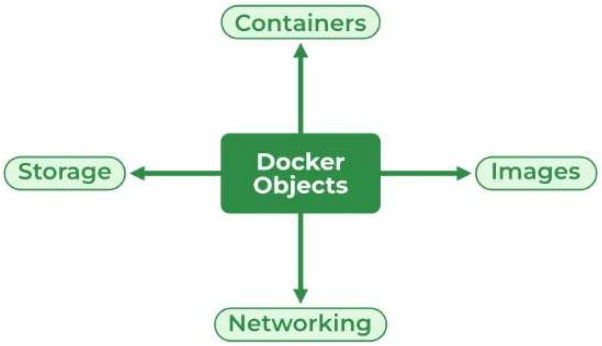
- **Public Registry:** The default public registry is **Docker Hub**, which contains a vast collection of community and official images.
- **Private Registries:** Organizations often use private registries (like Harbor, AWS ECR, or Google Artifact Registry) to store proprietary images for security and control.

Image Lifecycle Commands:

- docker pull <image_name>: Downloads an image from a configured registry to your local Docker Host.
- docker push <image_name>: Uploads a local image to a registry.

Docker Objects

Whenever we are using docker, we are creating and using images, containers, volumes, networks, and other objects.



1. Images

An image is a read-only, inert template that contains the instructions for creating a Docker container. Think of it as a blueprint or a class in object-oriented programming.

- It's built from a Dockerfile, a simple text file defining the steps to assemble the image.

- Images are built in layers, where each instruction in the Dockerfile corresponds to a layer. This layered architecture makes builds and distribution incredibly efficient.

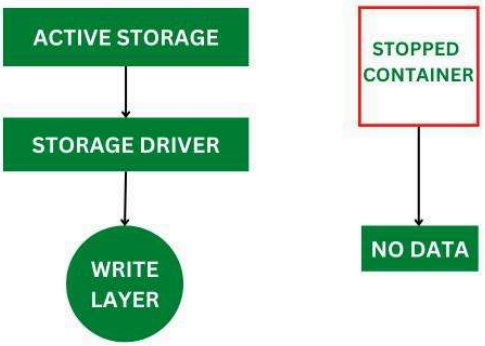
2. Containers

A container is a runnable, live instance of an image. If an image is the blueprint, a container is the house built from that blueprint.

- You can create, start, stop, move, or delete containers using the Docker API or CLI.
- Each container is isolated from other containers and the host machine, having its own filesystem, networking, and process space.
- You can run multiple containers from the same image.

3. Storage

Since a container's writable layer is ephemeral (data is lost when the container is deleted), Docker provides robust solutions for data persistence. Storage driver controls and manages the images and containers on our docker host.



Types of Docker Storage

Docker provides multiple storage options to persist, share, and manage data across containers and hosts.

- **Volumes:** The preferred mechanism. Volumes are managed by Docker and stored in a dedicated area on the host filesystem (e.g., /var/lib/docker/volumes/ on Linux). They are designed to survive the container lifecycle.

- **Bind Mounts:** Allow you to map a file or directory from the host machine directly into a container. This is very useful for development, where you might want to share source code with a container.
- **tmpfs Mounts:** In-memory storage that is temporary and never written to the host filesystem. Useful for sensitive data or high-performance temporary files.

Docker Networking

Docker networking provides complete isolation for docker containers. It means a user can link a docker container to many networks. It requires very few OS instances to run the workload.

Types of Docker Network

1. **Bridge:** It is the default network driver. We can use this when different containers communicate with the same docker host.
2. **Host:** When you don't need any isolation between the container and host then it is used.
3. **Overlay:** For communication with each other, it will enable the swarm services.
4. **None:** It disables all networking.
5. **macvlan**: Assigns a unique MAC address to a container, making it appear as a physical device on your network.

Example:

You run the command: docker run -d -p 80:80 nginx

1. **Client:** The Docker Client sends a REST API request to the Docker Daemon to create and run a container from the nginx image.
2. **Daemon:** The Daemon receives the request. It first checks if the nginx image exists locally on the Host.
3. **Registry (Pull):** If the image is not found locally, the Daemon contacts the configured Registry (Docker Hub by default) and pulls the nginx image.
4. **Runtime (containerd):** The Daemon hands the image and run-configuration over to containerd.

5. **Runtime (runc):** containerd uses runc to create a new container. runc interfaces with the Linux kernel to create isolated **namespaces** and limit resources with **cgroups**.
6. **Execution:** The container is started. Docker maps port 80 of the host to port 80 of the nginx container, as requested by the -p 80:80 flag. The Nginx process runs as PID 1 inside the container's isolated PID namespace.

Basic Docker Commands

Some commonly used Docker commands are:

1. docker version

The docker version command is used to display detailed information about the installed Docker software. It shows the version of the Docker client and Docker server (daemon), along with details such as API version, build time, and platform. This command helps verify whether Docker is installed correctly and running on the system.

2. docker pull image_name

The docker pull command is used to download a Docker image from a Docker registry such as Docker Hub. If the specified image is not available locally, Docker fetches it from the registry and stores it on the local system. This command is commonly used before running a container to ensure the required image is available.

3. docker images

The docker images command lists all Docker images present on the local machine. It displays information such as the image name, tag, image ID, creation date, and size. This command helps users manage and identify available images that can be used to create containers.

4. docker run image_name

The docker run command is used to create and start a new container from a specified Docker image. If the image is not available locally, Docker automatically pulls it from the registry before running the container. This command also allows options such as port mapping, volume mounting, and environment variable configuration.

5. docker ps

The docker ps command displays a list of all currently running containers. It provides information such as container ID, image name, command, status, and port mappings. This command is useful for monitoring active containers and managing their execution.

6. docker stop container_id

The docker stop command is used to safely stop a running container. It sends a termination signal to the container's main process, allowing it to shut down gracefully before stopping. This ensures that applications running inside the container are properly closed.

7. docker rm container_id

The docker rm command removes one or more stopped containers from the system. It helps free up system resources and keep the Docker environment clean. A container must be stopped before it can be removed using this command.

Advantages of Docker

- **Platform independent:** Docker containers package the application along with its dependencies, allowing it to run consistently across different operating systems and environments. This ensures the same behavior in development, testing, and production.
- **Faster deployment:** Docker containers are lightweight and start within seconds since they do not require a full operating system. This significantly reduces deployment time and speeds up application releases.
- **Easy scalability:** Docker allows applications to be scaled easily by running multiple container instances. Containers can be quickly started or stopped to handle varying workloads.
- **Efficient resource usage:** Containers share the host operating system kernel, which reduces memory and CPU overhead. This enables better utilization of system resources compared to virtual machines.
- **Supports microservices architecture:** Docker allows each microservice to run in its own container with independent dependencies. This improves flexibility, fault isolation, and easier maintenance of applications.

- **Improves DevOps workflow:** Docker enables seamless integration between development and operations teams by providing consistent environments. It supports continuous integration and continuous deployment (CI/CD) practices.