Docker

Docker is an open-source platform designed to automate the deployment, scaling, and management of applications within lightweight, portable containers. Containers encapsulate an application and its dependencies, ensuring consistency across various environments. It simplifies the development, testing, CICD pipelines, Microservices and deployment of applications.

Key Components and Concepts of Docker

Containers:

Containers are the core unit of Docker. They are lightweight, standalone, executable packages that include everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings. Containers share the same operating system kernel, making them more efficient than traditional virtual machines.

Images:

A Docker image is a read-only template used to create containers. Images contain the application code, libraries, dependencies, and other necessary files. Images are built from a set of instructions defined in a Dockerfile.

Dockerfile:

A Dockerfile is a text file that contains a series of instructions on how to build a Docker image. Each instruction in a Dockerfile corresponds to a command that the Docker engine executes to assemble the image.

Docker Engine:

Docker Engine is the runtime that enables containers to run. It consists of the Docker daemon (dockerd), an API that provides interaction with the daemon, and a command-line interface (CLI).

Docker Hub:

Docker Hub is a cloud-based repository where Docker users can create, test, store, and distribute Docker images. It allows for easy sharing and collaboration.

Volumes:

Volumes are used to persist data generated by and used by Docker containers. They are independent of the container’s lifecycle, ensuring data is not lost when a container is removed.

Networks:
Docker provides various networking options for containers to communicate with each other, with other applications, and with the outside world. This includes bridge networks, overlay networks, and more.

VIRTUALIZATION VS. CONTAINERIZATION

What is Virtualization?

Virtualization is achieved using a hypervisor, which splits CPU, RAM, and storage resources between multiple virtual machines (VMs). Each user on the hypervisor gets their own operating system environment.

It should be noted that none of the individual VMs interact with one another, but they all benefit from the same hardware. This means cloud platforms like AWS can maximize resource utilization per server with multiple tenants, enabling lower prices for enterprises through economies of scale

What is Containerization?

Containerization is a form of virtualization. Virtualization aims to run multiple OS instances on a single server, whereas containerization runs a single OS instance, with multiple user spaces to isolate processes from one another. This means containerization makes sense for one AWS cloud user that plans to run multiple processes simultaneously.

Containerization is achieved by packaging software code, libraries, frameworks, and other dependencies together in an isolated user space called a container. This container is portable and can be used on any infrastructure in any environment that supports the container technology, such as Docker and Kubernetes.

How is Containerization Related to Microservices?

Microservice architectures involve decoupling the main components of an application into singular, isolated components. Since the components can operate independently of one another, it reduces the risk of errors or complete service outages.

A container holds a single function for a specific task, or a microservice. By splitting each individual application function into a container, microservices improve enterprise service resilience and scalability.

Containerization also allows single application components to be updated in isolation, without affecting the rest of the technology stack. This ensures that security and feature updates are applied rapidly, with minimal disruption to overall operations.

Differences Between Virtualization and Containerization

At a technical level, both environments use similar properties while having different outcomes. Here are the primary differences between the two techniques.

1. Isolation

Virtualization results in a fully isolated OS and VM instance, while containerization isolates the host operating system machine and containers from one another. However, all containers are at risk if an attacker controls the host.

2. Different Operating Systems

Virtualization can host more than one complete operating system, each with its own kernel, whereas containerization runs all containers via user mode on one OS.

3. Guest Support

Virtualization allows for a range of operating systems to be used on the same server or machine. On the other hand, containerization is reliant on the host OS, meaning Linux containers cannot be run on Windows and vice-versa.

4. Deployment

Virtualization means each virtual machine has its own hypervisor. With containerization, either Docker is used to deploy an individual container, or Kubernetes is used to orchestrate multiple containers across multiple systems.

5. Persistent Virtual Storage

Virtualization assigns a virtual hard disk (VHD) to each individual virtual machine, or a server message block (SMB) if shared storage is used across multiple servers. With containerization, the local hard disk is used for storage per node, with SMB for shared storage across multiple nodes.

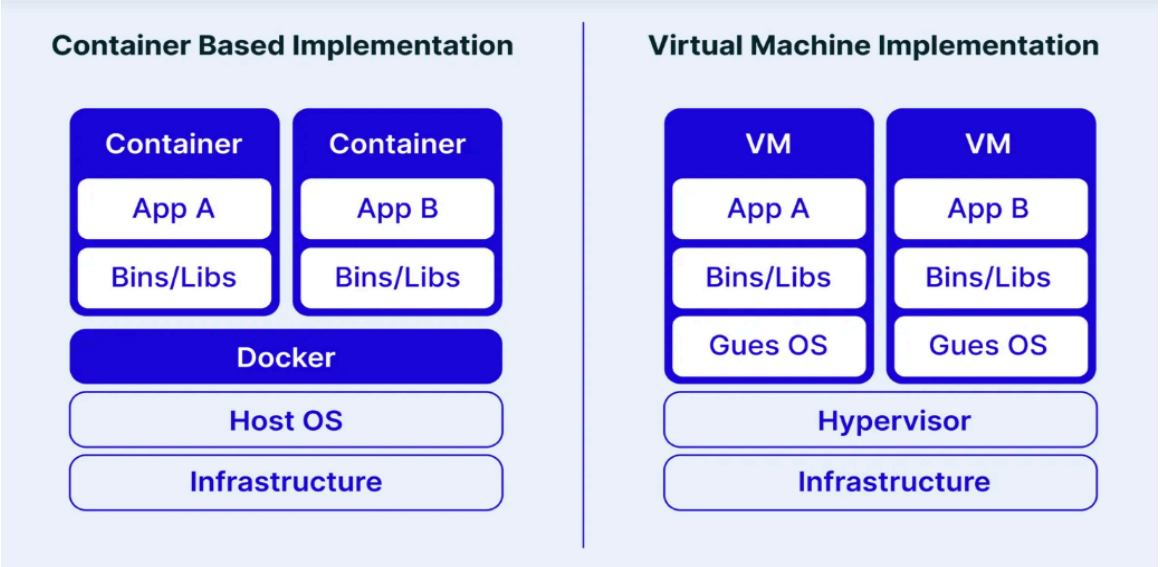
6. Virtual Load Balancing

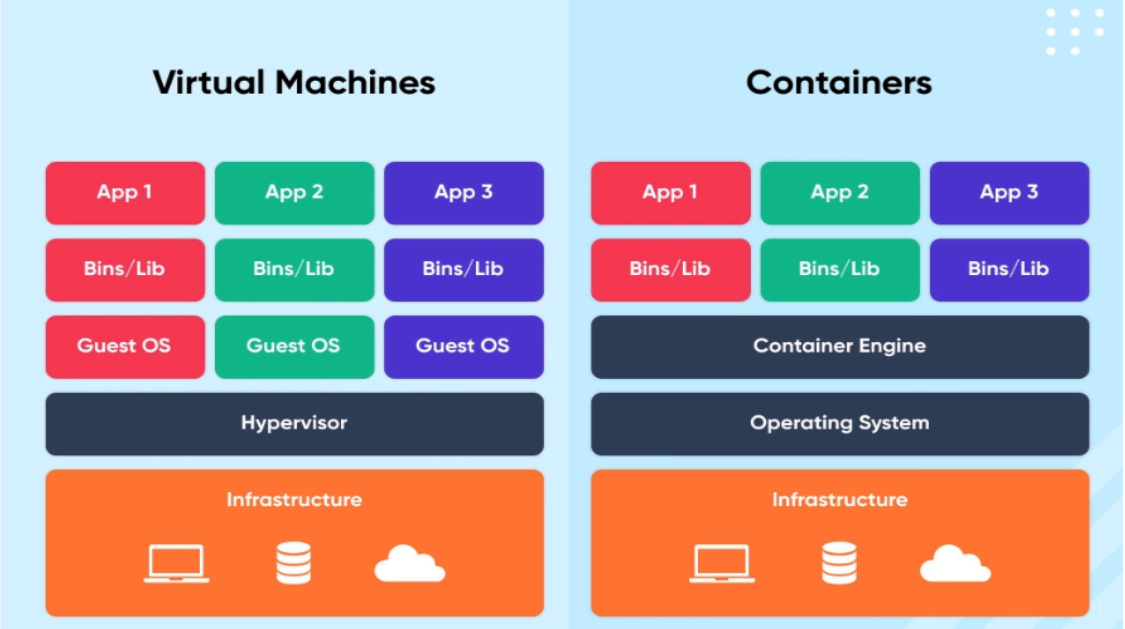
Virtualization means failover clusters are used to run VMs with load balancing support. Since containerization uses orchestration via Docker or Kubernetes to start and stop containers, it maximizes resource utilization. However, decommissioning for load balancing with containerization occurs when limits on available resources are reached.

7. Virtualized Networking

Virtualization uses virtual network adaptors (VNA) to facilitate networking, running through a master network interface card (NIC). With containerization, the VNA is split into multiple isolated views for lightweight network virtualization.

Feature	Container	Virtual machine
Operating system	Shares the host operating system's kernel	Has its own kernel
Portability	More portable	Less portable
Speed	Faster to start up and shut down	Slower to start up and shut down
Resource usage	Uses fewer resources	Uses more resources
Use cases	Good for portable and scalable applications	Good for isolated applications





DOCKER INSTALLATION AND RUNNING DAEMON

```
sudo apt-get update

sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release

sudo mkdir -p /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
```

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin

apt-cache madison docker-ce

sudo apt-get install docker-ce=<VERSION_STRING> docker-ce-cli=<VERSION_STRING>
containerd.io docker-compose-plugin

sudo docker run hello-world

sudo systemctl start docker && systemctl enable docker

newgrp docker

usermod -aG docker $USER
```

DOCKER and DOCKERFILE

Getting Started

The getting started guide on Docker has detailed instructions for setting up Docker.

After setup is complete, run the following commands to verify the success of installation:

PLEASE NOTE POST INSTALLATION STEPS BELOW IF YOU HAVE TO PREPEND SUDO TO EVERY COMMAND

- **docker version** - provides full description of docker version
- **docker info** - display system wide information
- **docker -v** - provides a short description of docker version
- **docker run hello-world** - pull hello-world container from registry and run it
- **docker compose version**

Optional Post Installation Steps

To create the docker group and add your user:

- **`sudo groupadd docker`**
- **`sudo usermod -aG docker $USER`**
- **`Log out and log back in so that your group membership is re-evaluated.`**
- **`Verify that you can run docker commands without sudo.`**

DOCKERFILE

A Dockerfile typically includes the following instructions:

FROM: Specifies the base image to use for the new image for any registry.

MAINTAINER: (Deprecated, use **LABEL** instead) Specifies the author of the image.

LABEL: Adds metadata to the image.

RUN: Executes commands in a new layer on top of the current image and commits the results.

COPY or ADD: Copies files/directories from the host into the image.

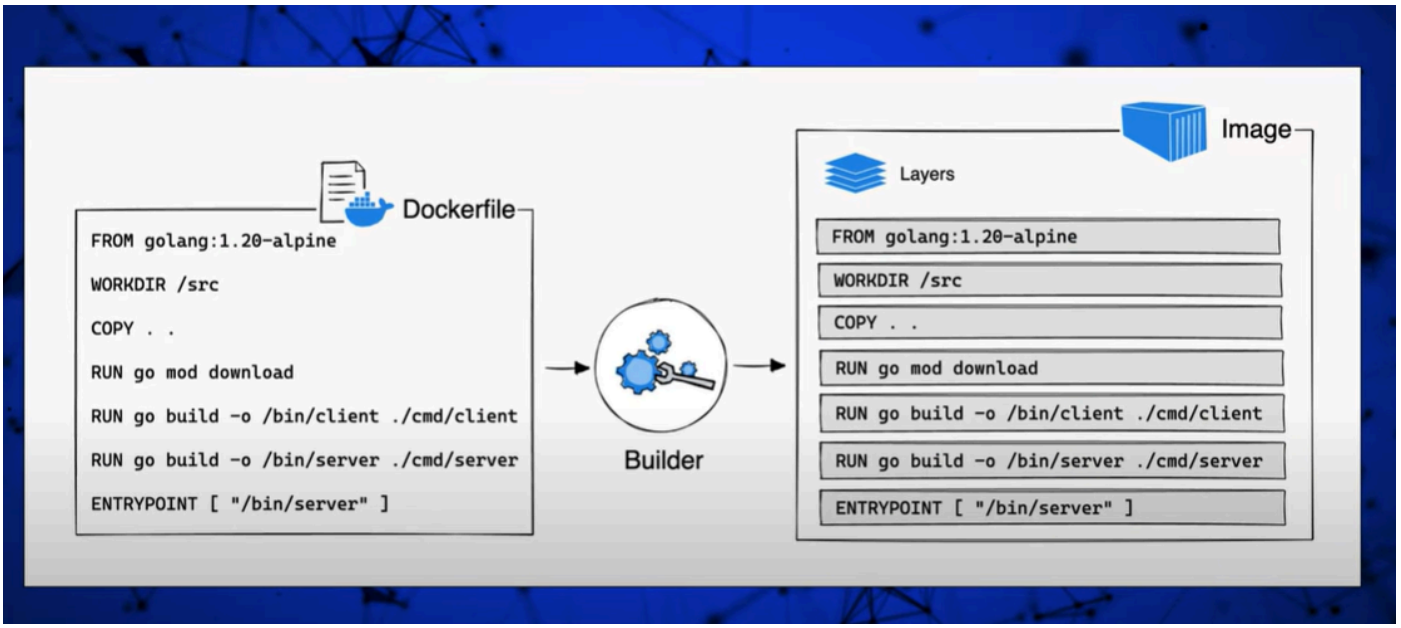
WORKDIR: Sets the working directory for any subsequent RUN, CMD, ENTRYPOINT, COPY, and ADD instructions.

ENV: Sets environment variables.

EXPOSE: Informs Docker that the container listens on the specified network ports at runtime.

CMD: Provides the default command to run when the container starts.

ENTRYPOINT: Configures a container that will run as an executable**.**



Example Dockerfile

Use the official Node.js image as the base image
FROM node:14

Create and change to the app directory
WORKDIR /usr/src/app

Copy the package.json and package-lock.json files
COPY package*.json ./

Install the dependencies
RUN npm install

Copy the rest of the application code
COPY . .

Expose port 8080
EXPOSE 8080

Define environment variables
ENV NODE_ENV=production

Run the application
CMD ["node", "app.js"]

Best Practices writing Dockerfile

Minimize Layers: Combine multiple `RUN` instructions into a single instruction to reduce the number of layers in the image.

```
RUN apt-get update && apt-get install -y \
    package1 \
    package2 \
    package3
```

Leverage .dockerignore: Use a `.dockerignore` file to exclude files and directories from the build context that are not needed in the final image.

```
node_modules

npm-debug.log
```

Use Specific Version Tags: Avoid using `latest` tags as they can lead to inconsistent builds. Specify a specific version or tag.

```
FROM node:14.17.0
```

Keep Images Small: Remove unnecessary dependencies and clean up after installations to keep the image size minimal.

```
RUN apt-get update && apt-get install -y package && \
```

```
apt-get clean && \
rm -rf /var/lib/apt/lists/*
```

Security: Regularly update the base image and dependencies to include the latest security patches.

DOCKER SWARM

Docker Swarm is an orchestration management tool that runs on Docker applications. It allows you to create and deploy a cluster of Docker nodes. Each node in a Docker Swarm is a Docker daemon, and these daemons interact using the Docker API. With Docker Swarm, you can manage multiple nodes (referred to as clusters) and maintain your containers across them. It's a powerful way to scale and manage containerized applications. 🐳

Cluster Management: Docker Swarm allows users to create and manage a cluster of Docker nodes (physical or virtual machines). This cluster is referred to as a "Swarm".

Decentralized Design: Swarm mode uses a decentralized design, where the workload can be distributed among multiple manager and worker nodes.

Service Definitions: In Docker Swarm, applications are deployed as services. A service is a task that Docker executes on one or more nodes in the Swarm.

Scaling: Swarm makes it easy to scale services up or down by adjusting the number of task replicas. Swarm automatically distributes the tasks across nodes in the most efficient way.

Load Balancing: Docker Swarm includes built-in load balancing. When you deploy a service, Swarm automatically assigns each service a unique DNS name and distributes the requests among the service replicas.

Service Discovery: Nodes in the Swarm can discover and communicate with other services through an in-built DNS service.

High Availability: Docker Swarm ensures high availability of services by maintaining multiple manager nodes and automatically reassigning tasks from failed nodes to healthy ones.

Security: Swarm mode includes several security features, such as mutual TLS authentication between nodes and encryption of data in transit.

Rolling Updates: Docker Swarm supports rolling updates, allowing you to update services with zero downtime. This means you can gradually update your application without affecting the service availability.

Declarative Service Model: Users define the desired state of the services, and Docker Swarm ensures that the actual state matches the desired state, maintaining consistency across the cluster.

Basic Components of Docker Swarm

- **Nodes**: These are individual Docker engines participating in the Swarm, categorized as manager nodes and worker nodes.

- **Manager Nodes**: Responsible for the overall management of the Swarm, including maintaining the desired state of the cluster and handling requests from clients.
- **Worker Nodes**: Execute tasks assigned to them by manager nodes. Worker nodes do not participate in the orchestration of the cluster.
- **Services**: High-level definitions of containers. Services define how containers are distributed across the nodes.
- **Tasks**: Individual instances of a service running on nodes in the Swarm.

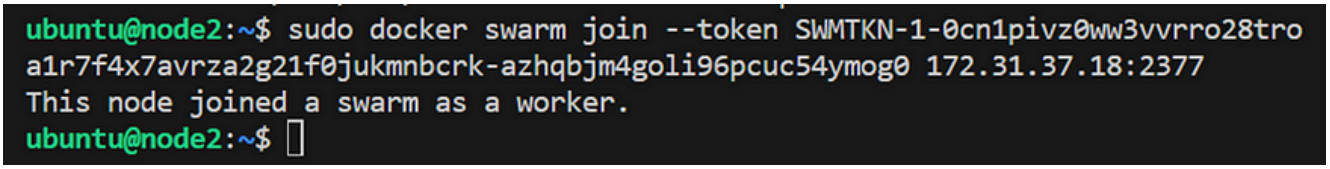
Getting Started with Docker Swarm

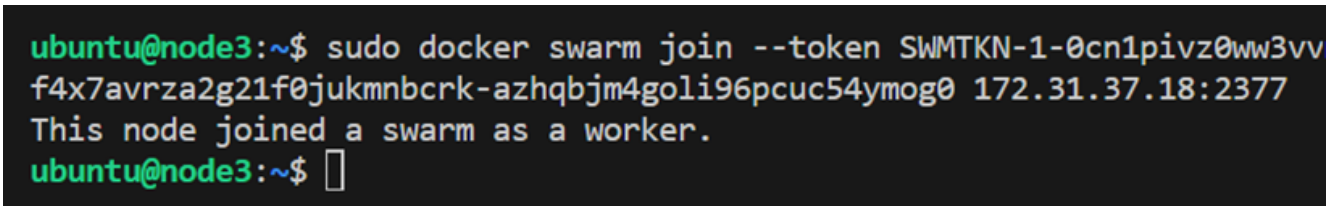
In Node 1, we want to input the below command. This will initialize, or begin, the swarm.

sudo docker swarm init --advertise-addr [Input node1 Private IP]

The output above literally gives us the exact command we need to run to get Node 2 and 3 added to the swarm.

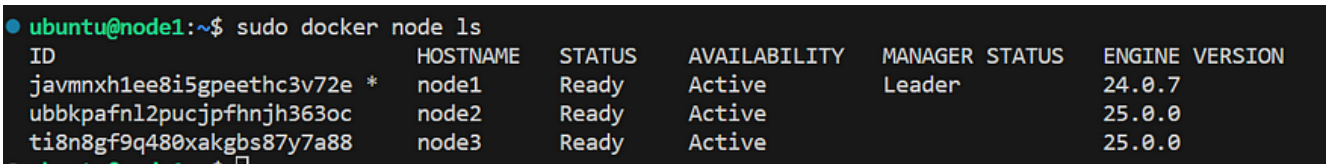
Note: we will need to add “sudo” to the front of this command. Copy the command into each terminal for Node 2 and 3 (shown below). This will complete the process for each node to join the swarm the manager (Node 1) created.





Lastly, in the manager node (Node 1) check the status of each node.

sudo docker node ls



We can see above that each node is active. So, that's it! We successfully set up a Docker Swarm with one manager node and two worker nodes.

Cleanup

It is important to remember to leave the swarm and exit the ssh connection in each node. Input the two below commands into each node. You will want to leave the manager swarm last.

#leave the swarm

sudo docker swarm leave --force

DOCKER SECURITY

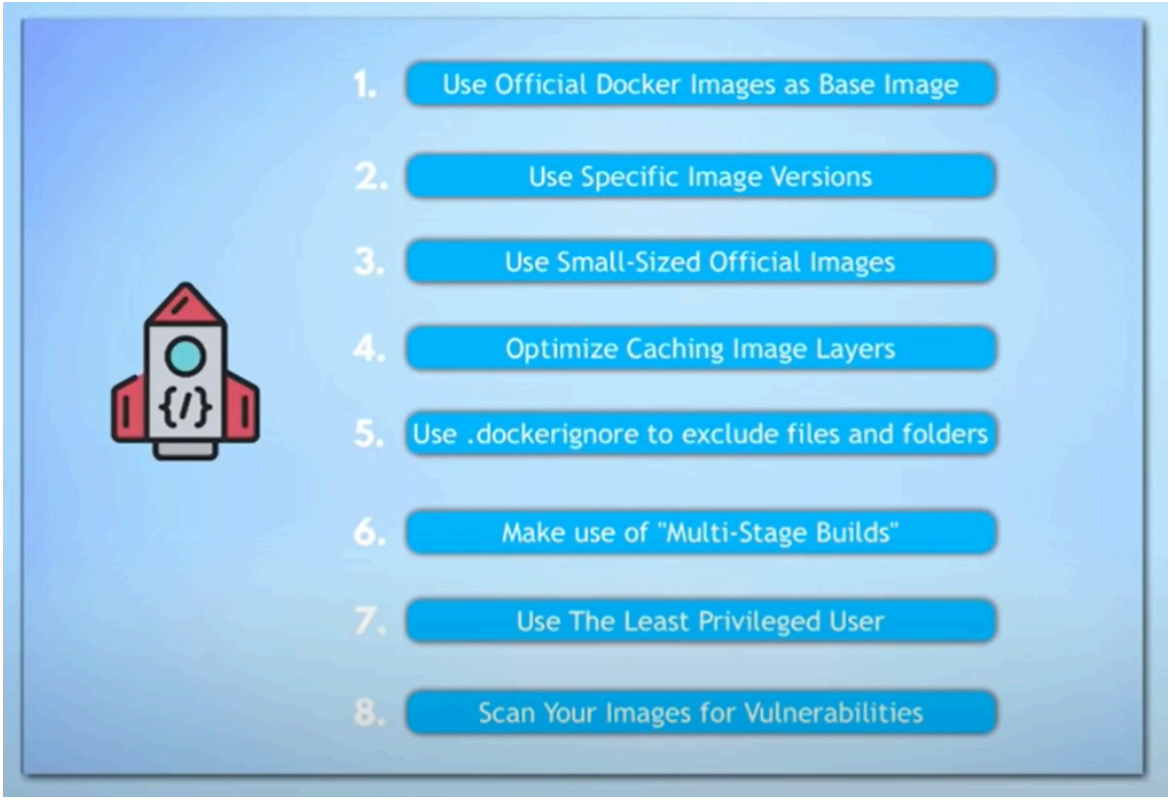
Tip #1: Don't run the container as the root user.

Tip #2: Use a multi-stage build + distroless base image.

Tip #3: Harden the security of the host system.

Tip #4: Use a container image scanner to detect vulnerabilities.

Tip #5: Don't install/configure things within the Dockerfile without understanding the potential risks



Difference Between Kubernetes And Docker Swarm

Kubernetes

- Complex installation
- More complex with a high learning curve, but more powerful
- Supports auto-scaling
- Built in monitoring
- Manual setup of load balancer
- Need for a separate CLI tool

Docker Swarm

- Easier installation
- More lightweight and easier to use, but limited functionality
- Manual scaling
- Needs third party tools for monitoring
- Auto load balancing
- Integrates docker CLI

DOCKER CHEAT SHEET

Basic Docker Concepts and Terms:

Docker Image: 🖼️
A lightweight, stand-alone, executable package that includes everything needed to run a piece of software.

Docker Container: 🏃
A runtime instance of a Docker image.

Docker Hub: ☁️
A cloud-based registry service where Docker users and partners create, test, store, and distribute container images.

Dockerfile: 📄
A text document that contains all the commands a user could call on the command line to assemble an image.

Docker Compose: 🎶🐳
A tool for defining and running multi-container Docker applications.

Basic Docker Commands: 🔧🏁

- **docker --version:** Display Docker version.
- **docker info:** Display system-wide information.
- **docker run image:** Run a Docker container from an image.
- **docker ps:** List running Docker containers.
- **docker ps -a:** List all Docker containers.
- **docker stop container_id:** Stop a running container.
- **docker rm container_id:** Remove a Docker container.
- **docker images:** List Docker images.
- **docker rmi image_id:** Remove a Docker image.

Intermediate Docker Commands: 🛠️🚀

- **docker run -d image:** Run a Docker container in detached mode.
- **docker run -p host_port:**container_port image: Map a port from the host to a container.
- **docker run -v host_volume:**container_volume image: Mount a volume from the host to a container.
- **docker run -e VAR=VALUE image:** Set environment variables in a container.
- **docker inspect container_id/image_id:** Return low-level information on Docker objects.
- **docker build -t tag .:** Build a Docker image with a tag from a Dockerfile in the current directory.
- **docker tag image new_tag:** Tag an image with a new tag.

Dockerfile Commands: 🚢🛠️

- **FROM image:** Set the base image.
- **RUN command:** Run a command.
- **CMD command:** Set a default command that will run when the container starts.
- **ENV VAR=VALUE:** Set environment variables.
- **ADD source destination:** Copy files from source to the container's filesystem at the destination.
- **COPY source destination:** Copy new files or directories from source and add them to the filesystem of the container.
- **ENTRYPOINT command:** Allow you to configure a container that will run as an executable.
- **LABEL:** Adds metadata to an image.
- **EXPOSE:** Informs Docker that the container listens on the specified network ports at runtime.

.

Docker Compose Commands: 🎶🐋🎵

- **docker-compose up:** Create and start containers.
- **docker-compose down:** Stop and remove containers, networks, images, and volumes.
- **docker-compose build:** Build or rebuild services.
- **docker-compose logs:** View output from containers.
- **docker-compose restart:** Restart services.

Docker Networking: 🌐🔗

- **docker network ls:** List networks.
- **docker network create network:** Create a network.
- **docker network rm network:** Remove a network.
- **Bridge:** Docker's default networking driver.
- **Host:** For standalone containers, removes network isolation between the container and the Docker host.
- **Overlay:** Networks connect multiple Docker daemons together and enable swarm services to communicate with each other.
- **Macvlan:** Assigns a MAC address to a container, making it appear as a physical device on your network.

Docker Volumes: 🗄️📁

- **docker volume ls:** List volumes.
- **docker volume create volume:** Create a volume.
- **docker volume rm volume:** Remove a volume.

Docker Object Commands: 📦🔨

- **docker image:** Manages images.
- **docker container:** Manages containers.
- **docker network:** Manages networks.
- **docker volume:** Manages volumes.
- **docker secret:** Manages Docker secrets.
- **docker plugin:** Manages plugins.

Docker Advanced Commands: 🚀🛠️

- **docker history image:** Show the history of an image.
- **docker save image > file:** Save an image to a tar archive.
- **docker load < file:** Load an image from a tar archive.
- **docker commit container image:** Create a new image from a container's changes.

Docker System Commands: 💻🔍

- **docker info:** Displays system-wide information.
- **docker version:** Shows the Docker version information.
- **docker system df:** Shows Docker disk usage.
- **docker system events:** Gets real-time events from the server.
- **docker system prune:** Removes unused data.

Docker Swarm Commands: 🐝🚀

- **docker swarm init:** Initialize a swarm.
- **docker swarm join:** Join a node to a swarm.
- **docker node ls:** List nodes in a swarm.
- **docker service create image:** Create a service.
- **docker service ls:** List services in a swarm.
- **docker service rm service:** Remove a service.
- **docker swarm:** Manages Swarm.
- **docker node:** Manages Swarm nodes.
- **docker stack:** Manages Docker stacks.
- **docker service:** Manages services.

Container Orchestration with Docker Swarm: 🐝🎶🐳

- **Services:** 🎶🐳
  The definition of the tasks to execute on the manager or worker nodes.
- **Tasks:** 🚀🔧
  A single runnable instance of a service.

- **Worker nodes:** 👷
  Nodes that receive and execute tasks dispatched from manager nodes.
- **Manager nodes:** 👨‍💼
  The only nodes that can execute Docker commands or authorize other nodes to join the swarm.
- **Raft Consensus Algorithm:** 👑📜
  Manager nodes use the Raft Consensus Algorithm to agree on task scheduling and status updates.
- **Services scaling:** 🔢🔧
  In Docker Swarm mode, you can scale your services up or down for optimal resource utilization.

Docker Security: 🔒🛡️

- **docker secret create secret file:** Create a secret from a file.
- **docker secret ls:** List secrets.
- **docker secret rm secret:** Remove a secret.
- **Docker Security Scanning:** 🔍🛡️
  A security feature that you can use in Docker repositories.
- **Docker Content Trust:** 🤝🔒
  Provides the ability to use digital signatures for data sent to and received from remote Docker registries.
- **Docker Secrets:** 🤫🔐
  Allows you to manage sensitive data, such as passwords, SSH private keys, SSL certificates, and other data.

Docker Troubleshooting and Monitoring: 🐛🔧👀

- **docker stats:** Display a live stream of container(s) resource usage statistics.
- **docker system df:** Display the space usage of Docker daemon entities.
- **docker inspect:** Return low-level information on Docker objects.
- **docker events:** Get real-time events from the server.
- **docker logs:** Fetch the logs of a container.
- **docker healthcheck:** Checks the health of a running container.

Docker Registries and Repositories: 📦🔍🏛️

- **Docker Hub:** ☁️📦
  Docker's public registry instance.
- **Docker Trusted Registry (DTR):** 🔒📦
  Docker's commercially supported storage for Docker images.
- **Docker Content Trust (DCT):** 🤝🔒📦
  Provides the ability to use digital signatures for data sent to and received from remote Docker registries.

Docker and CI/CD: 🛠️🚀🔧

- **Docker in Jenkins:** 🔧🏭
  Jenkins provides built-in Docker integration for CI/CD workflows.

- **Docker in Travis CI:** 🔧🚂
  Travis CI also provides Docker integration for CI/CD workflows.
- **Docker in GitLab CI:** 🔧🏭
  GitLab CI has native Docker support for CI/CD workflows.
- **Docker in CircleCI:** 🔧🔄
  CircleCI offers Docker support to build and push Docker images.
- **Docker in Azure DevOps:** 🔧🏭🔧
  Azure DevOps can build, push, or run Docker images, or run a Docker command.

Docker and the Cloud: ☁️🐳🚀

- **Docker on AWS:** 🌐🔧
  AWS provides services like Amazon Elastic Container Service (ECS) and AWS Fargate for running Docker containers.
- **Docker on Azure:** 🌐🔧
  Azure provides Azure Kubernetes Service (AKS) for running Docker containers.
- **Docker on Google Cloud:** 🌐🔧
  Google Cloud provides Google Kubernetes Engine (GKE) for running Docker containers.

Docker Best Practices: 👨‍🏫🛠️🚀

- **Container immutability:** 🧱🔒
  The idea that you never update a running container; instead, you should always create a new one.
- **Single process per container:** 🚀🔧
  Each container should address a single concern and do it well.
- **Minimize layer counts in Dockerfiles:** 🔄🔨
  The fewer commands that create layers, the smaller your image is likely to be.
- **Leverage build cache:** 🏗️🔍
  Docker will cache the results of the first build of a Dockerfile, allowing subsequent builds to be super fast.
- **Use .dockerignore:** 🙈🔍
  Prevents sending unnecessary files to the daemon when building images.
- **Use specific tags for production images:** 🏷️🚀
  Using specific versions of an image ensures that your application consistently works as expected.
- **Always use the latest version of Docker:** 🔧🆕
  Each new version of Docker includes security improvements, bug fixes, and new features.

Docker and Microservices: 🐳🔧🏭

- **Service discovery:** 🔍🔭
  Docker Swarm Mode has a built-in DNS server that other containers can use to resolve the service name to an IP address.
- **Service scaling:** 🔢🔧
  In Docker Swarm Mode, you can scale your services up or down.
- **Load balancing:** ⚖️🚀
  Docker has a built-in load balancer that can distribute network connections to all instances of a replicated service.

- **Secure communication between services:** 🔒📞
  Docker Swarm Mode has a built-in routing mesh that provides secure communication between services.

Docker Plugins: 🧩🔌

- **Storage Plugins:** 🗄️🔌
  These plugins provide storage capabilities to Docker containers.
- **Network Plugins:** 🔗🔌
  These plugins provide networking capabilities to Docker containers.
- **Authorization Plugins:** 🔐🔌
  These plugins restrict the Docker APIs that can be accessed.

Docker API: 🚀📡

- **Docker REST API:** 🔧📡
  An API used by applications to interact with the Docker daemon.
- **Docker SDK:** 🔧👨‍💻
  SDKs for Go and Python, built on top of the Docker REST API.
- **Docker Engine API:** 🔧📡
  The API Docker clients use to communicate with the Docker daemon.

Docker Editions: 🎁🚀

- **Docker Community Edition (CE):** 🏠🐳
  Ideal for individual developers and small teams looking to get started with Docker and experimenting with container-based apps.
- **Docker Enterprise Edition (EE):** 🏢🐳
  Designed for enterprise development and IT teams who build, ship, and run business-critical applications in production at scale.

Docker Architecture: 🏗️🚀

- **Docker Engine:** 🔧📡💻
  A client-server application with three major components: a server, a REST API, and a command-line interface (CLI).
- **Docker Daemon:** 👹🚀
  Listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
- **Docker Client:** 👨‍💻📡
  The primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out.
- **Docker Images:** 🖼️🏞️
  The basis of containers. An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime.
- **Docker Containers:** 🏃‍♂️📦
  A runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI.
- **Docker Services:** 🎶🐳🏭
  Allows you to scale containers across multiple Docker daemons, which all work together as a swarm with multiple managers and workers.