DataEngineerGuys Workshops

# docker

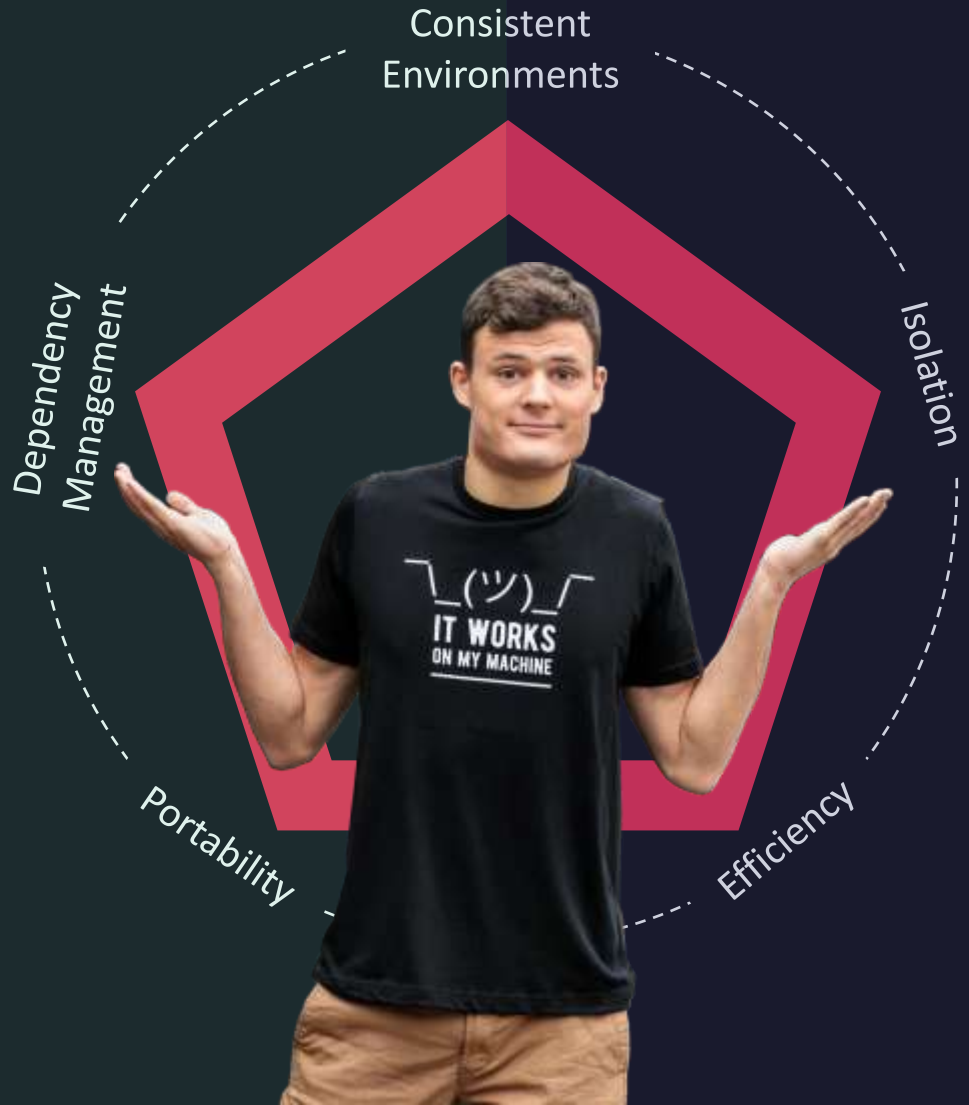
For Data Engineers

behnam-yazdanpanahi

## Software Development Challenges

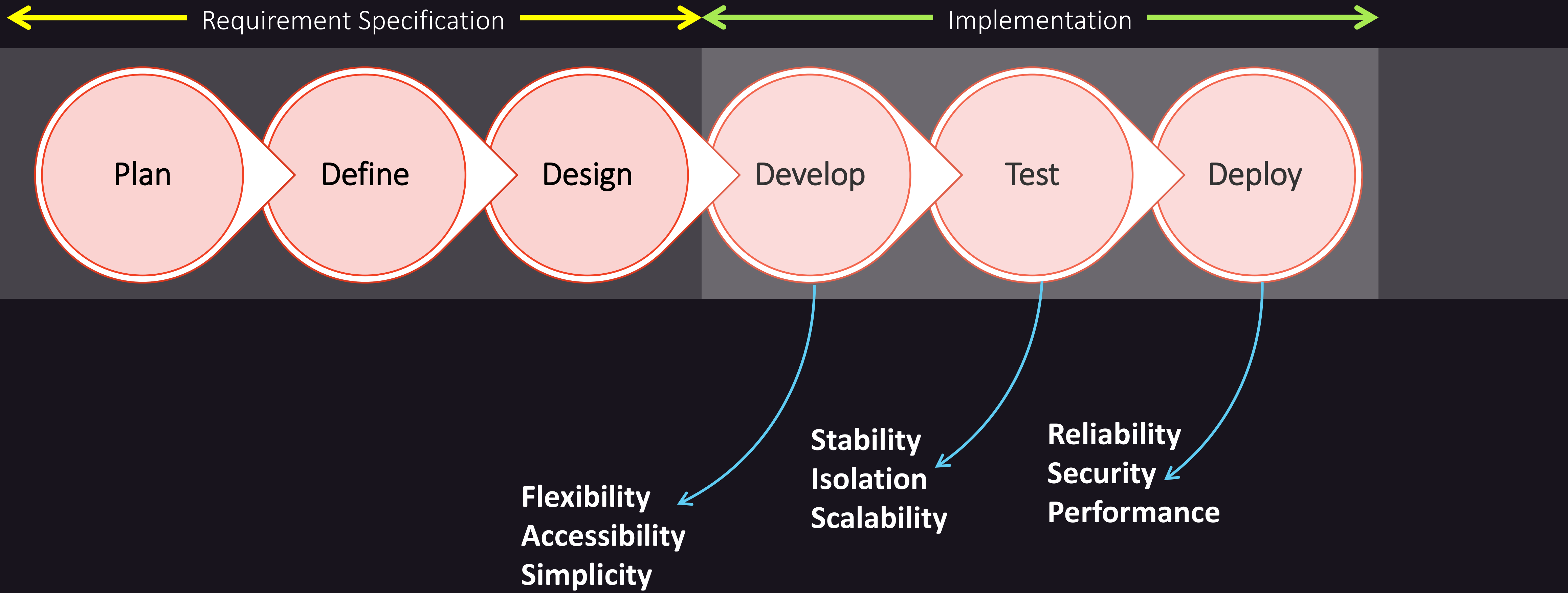
Developer

End-User

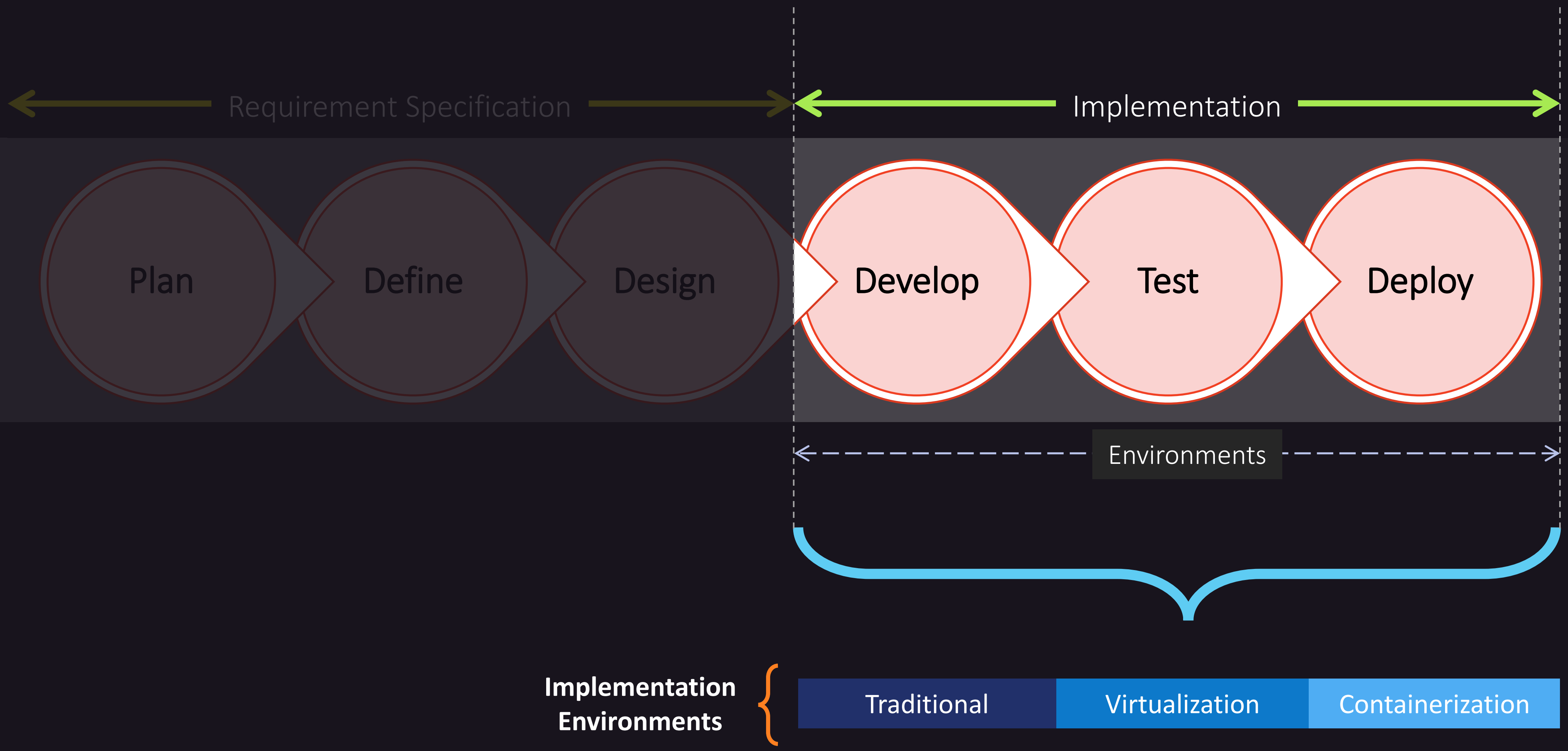


## Software Development Life Cycle

In software development, different environments serve distinct purposes throughout the software development lifecycle (SDLC), from initial coding to test and deployment.

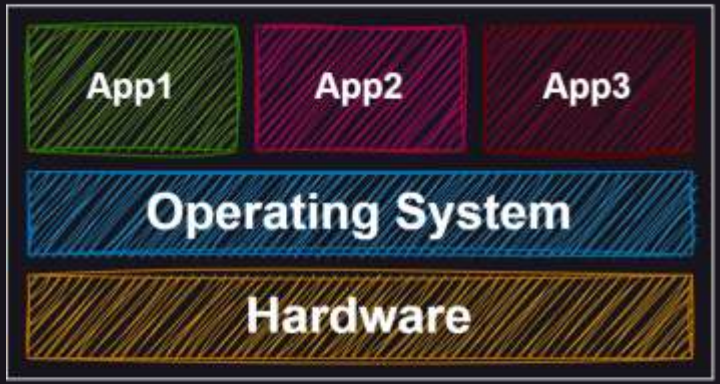


# Software Development Life Cycle

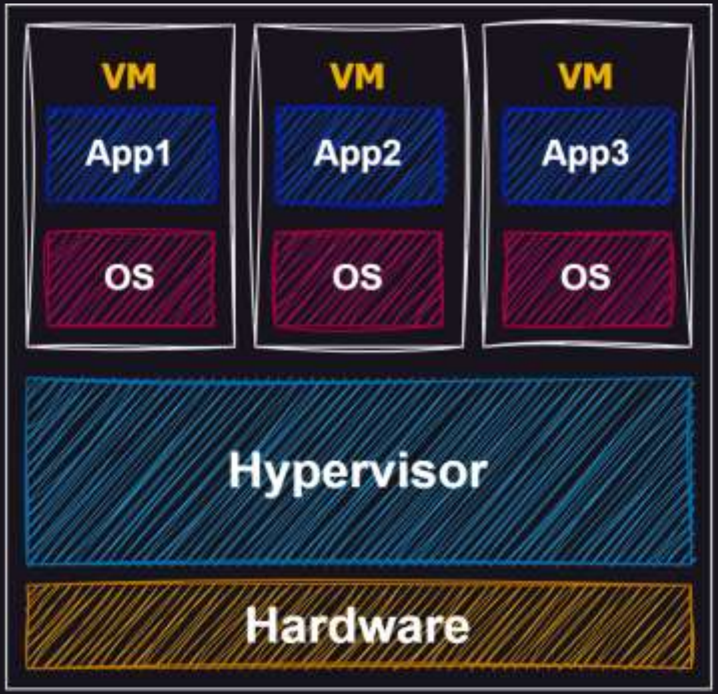


# Deployment Methods

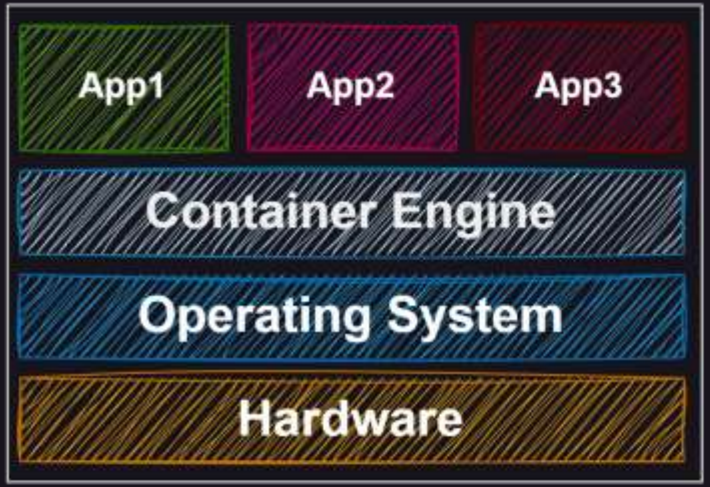
Traditional vs Virtualization vs Containerization



Traditional Method



Virtualization



Containerization

# Deployment Methods

Traditional vs Virtualization vs Containerization

| | Resource Utilization | Isolation and Flexibility | Portability and Scalability | Management and Automation |
|---|---|---|---|---|
| Traditional | Acceptable resource utilization but potential underutilization due to dedicated servers for each application. | Limited isolation between applications, leading to potential conflicts and dependencies. | Low portability and scalability with manual configuration and dedicated servers, leading to slower deployment times and scalability challenges. | Limited automation and management capabilities, relying on manual configuration and deployment processes. |
| Virtualization | Decent resource utilization with overhead from running multiple guest operating systems. | Decent isolation between virtual machines but may have performance overhead. | Moderate portability and scalability due to overhead from running multiple guest operating systems. | Management and automation tools available but may require more setup compared to containerization. |
| Containerization | Efficient resource utilization by sharing the host operating system's kernel among containers. | Good isolation between containers with minimal overhead, suitable for microservices architectures. | High portability and scalability facilitated by lightweight containers and efficient resource utilization. | Excellent management and automation capabilities with built-in tools like Docker and Kubernetes. |

## Explain containerization with an example

Imagine you're preparing to bake a cake. In a traditional kitchen setup, you might gather all your ingredients, utensils, and tools each time you want to bake a cake. This process can be time-consuming and messy, especially if you're working on multiple recipes simultaneously.

?

## Explain with an example

### Now, let's consider containerization.

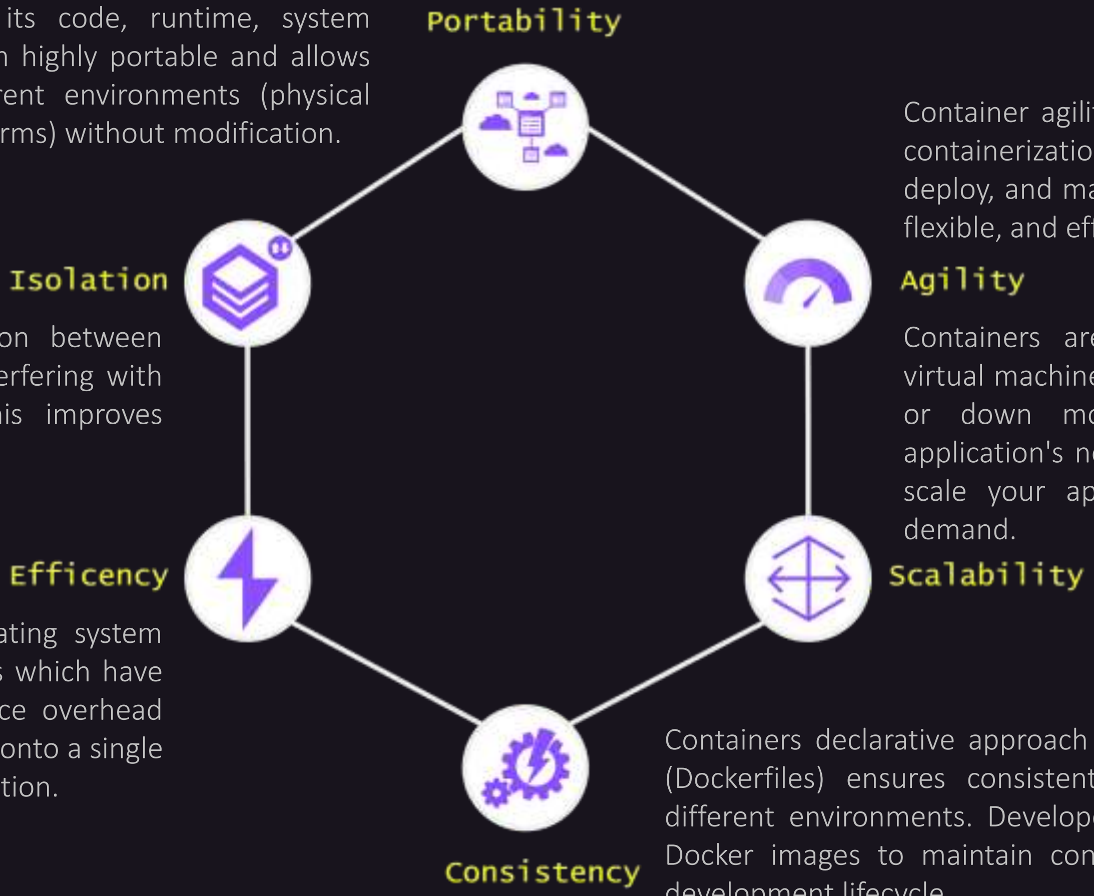
Instead of preparing everything from scratch every time you bake a cake, containerization allows you to package all the necessary ingredients and tools into a single, self-contained container. This container includes the cake batter, frosting, baking pan, spatula, and any other items you need. With containerization, you can simply grab the pre-packaged container whenever you want to bake a cake, saving time and ensuring consistency in your baking process.

## Containerization

Containerization simplifies software development, deployment, and management by packaging applications and their dependencies into lightweight, portable containers. With containerization, developers can build, ship, and run applications consistently across different environments, leading to faster development cycles, improved scalability, and enhanced reliability.

# Containerization Advantage

Containers are lightweight and package all the dependencies an application needs to run, including its code, runtime, system libraries, and settings. This makes them highly portable and allows them to run seamlessly across different environments (physical machines, virtual machines, cloud platforms) without modification.



Container agility is one of the key advantages of containerization. It refers to the ability to develop, deploy, and manage applications in a faster, more flexible, and efficient way.

Containers provide a layer of isolation between applications, preventing them from interfering with each other or the host system. This improves security and stability.

Containers are very lightweight compared to virtual machines. This allows you to easily spin up or down more containers based on your application's needs. This makes it much easier to scale your applications up or down to meet demand.

Containers share the underlying operating system kernel of the host machine, unlike VMs which have their own kernel. This reduces resource overhead and allows you to pack more containers onto a single server, leading to better hardware utilization.

Containers declarative approach to defining infrastructure as code (Dockerfiles) ensures consistent and reproducible builds across different environments. Developers can create, share, and version Docker images to maintain consistency throughout the software development lifecycle.

## Containerization Solutions

docker®

Leading containerization platform simplifying creation, deployment, and management of containers.

## Before Docker

Before Docker, developers and operations teams often faced several challenges related to dependency management, environment inconsistencies, and deployment processes:

| | Problem | Consequences | Solution | Benefits |
|---|---|---|---|---|
| Dependency Issues | Managing dependencies across different environments leads to conflicts and version mismatches. | Dependency conflicts result in runtime errors and compatibility issues. | Docker containerization encapsulates applications and dependencies. | Containers ensure consistent dependencies, accelerating development and improving application reliability. |
| Environment Inconsistencies | Inconsistent environments cause "it works on my machine" scenarios and deployment discrepancies. | Difficulty reproducing production issues and deploying consistently. | Docker provides consistent runtime environments across stages. | Reproducible environments streamline debugging and deployment, enhancing reliability. |
| Deployment Challenges | Manual deployment processes are time-consuming and error-prone. | Deployment failures and scalability limitations hinder agility. | Docker enables automated deployment pipelines with immutable infrastructure. | Accelerated deployment, reduced failures, and scalable applications with Docker. |

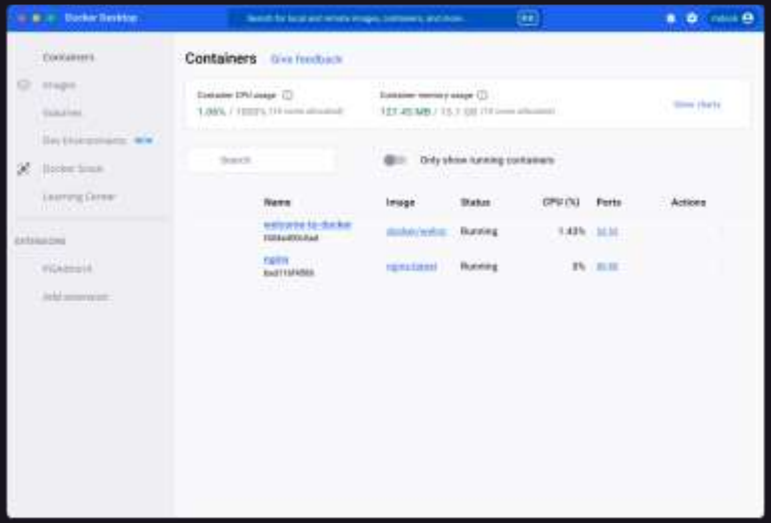
# Hands-on Docker

Part-01

## Docker Installation

Ubuntu on WSL2

https://learn.microsoft.com/en-us/windows/wsl/



Docker Desktop

Containers Give feedback

Container CPU usage 1.06% / 1000% (10 cores allocated)

Container memory usage 121.45 MB / 15.1 GB (10 cores allocated)

Only show running containers

| Name | Image | Status | CPU (%) | Ports | Actions |
|---|---|---|---|---|---|
| welcome-to-docker [illegible] | docker/welcome | Running | 1.43% | [illegible] | |
| nginx [illegible] | nginx:latest | Running | 0% | [illegible] | |

Docker Desktop

https://www.docker.com/products/docker-desktop/

Windows Terminal

1

2

3

## Getting Started with Docker (1)

### 1 Pull Image

Open a terminal or command prompt and pull the Redis image from Docker Hub using the following command:

```
> docker pull redis
```

### 2 Run Container (Detach mode)

Run a Redis container using the pulled image

```
> docker run --name my-redis1 -d redis
```

`--name my-redis1`: Assigns a name "my-redis1" to the container.
`-d`: Runs the container in detached mode, meaning it runs in the background.

### 3 Connect to Container

To interact with the Redis container in detached mode, you can use the ***docker exec*** command to execute commands inside the container:

```
> docker exec -it my-redis1 redis-cli
```

This command opens a Redis command line interface connected to the Redis server running inside the container. You can now execute Redis commands such as SET, GET, etc.

**Note:**

The `-it` flag in Docker's `docker run` command stands for **"interactive"** and **"tty"** (teletype). It's actually a combination of two separate flags:

> `-i`: This flag allows you to keep STDIN (standard input) open even if not attached. It enables you to interact with the container's STDIN, which is useful for providing input to processes running inside the container.
>
> `-t`: This flag allocates a pseudo-TTY (teletype) for the container, which simulates a terminal. It ensures that the output from the container is formatted properly and makes it easier to read.

When used together as `-it`, this flag combination enables interactive mode, allowing you to interact with the container's command-line interface. It's commonly used when you want to run commands inside a container and interact with them in real-time, such as when using interactive shells or command-line tools.

# Getting Started with Docker (2)

## 4 Expose Ports and Connect to Container (Attach Mode)

To connect to the Redis server from your local machine, you need to expose the Redis port:

```
> docker run --name my-redis2 -p 6379:6379 redis
```

```
-p 6379:6379: Maps port 6379 on your local machine to port 6379 in the container.
```

## 5 Clean Up

When you're done experimenting, you can stop and remove the Redis container

```
> docker stop my-redis1 my-redis2
> docker rm my-redis1 my-redis2
```

```
--name my-redis: Assigns a name "my-redis" to the container.
-d: Runs the container in detached mode, meaning it runs in the background.
```

## Congratulations!

This tutorial covers pulling Docker images, creating and running containers, connecting to containers in attach and detach modes, exposing ports, and interacting with containers using Redis.

## Note:

**Port exposing** in Docker allows you to make services running inside a container accessible from outside. By using the -p flag during container creation, you map a port on the host machine to a port in the container. This enables access to the container's services from your host machine or other containers. Exposing ports facilitates communication between the container and external clients, making it easy to run and access networked applications inside Docker containers. Here's how it works:
Sure!

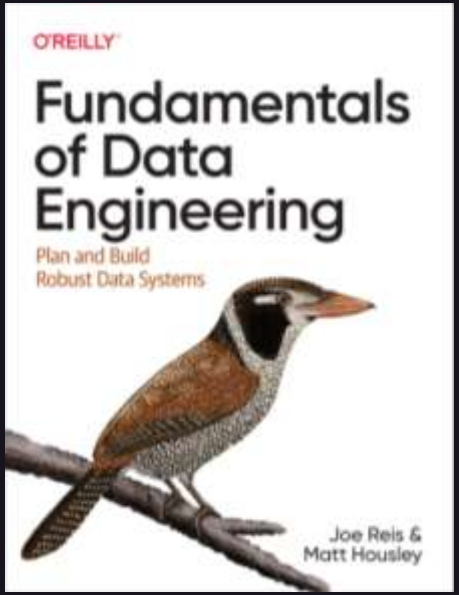
1. Specify Ports: Use the `-p` flag during container creation to map ports between the host and the container. For example, `-p 8080:80` maps port 80 in the container to port 8080 on the host.

2. Access Services: Access services running inside the container from your host machine by directing traffic to the mapped port. For instance, if a web server runs on port 80 inside the container, you can access it from your browser at `http://localhost:8080`.

3. Automatic Routing: Docker sets up routing rules automatically to forward traffic between the host and the container, allowing seamless communication without manual configuration.

## Role of Docker in Data Engineering

## Data Engineering Defined



**Data engineering** is the development, implementation, and maintenance of systems and processes that take in raw data and produce high-quality, consistent information that supports downstream use cases, such as analysis and machine learning.

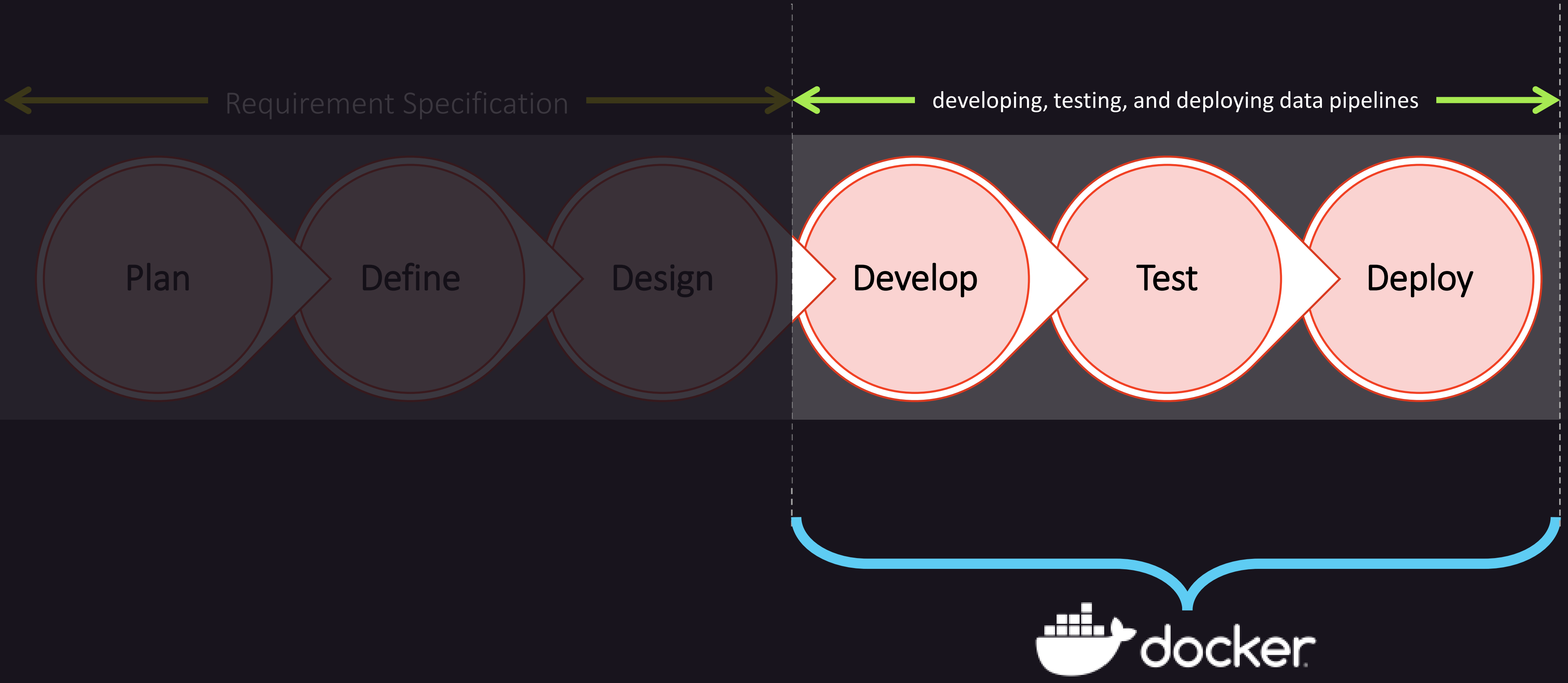
Data engineering is the intersection of security, data management, DataOps, data architecture, orchestration, and software engineering.

**A data engineer** manages the *data engineering lifecycle*, beginning with getting data from source systems and ending with serving data for use cases, such as analysis or machine learning.

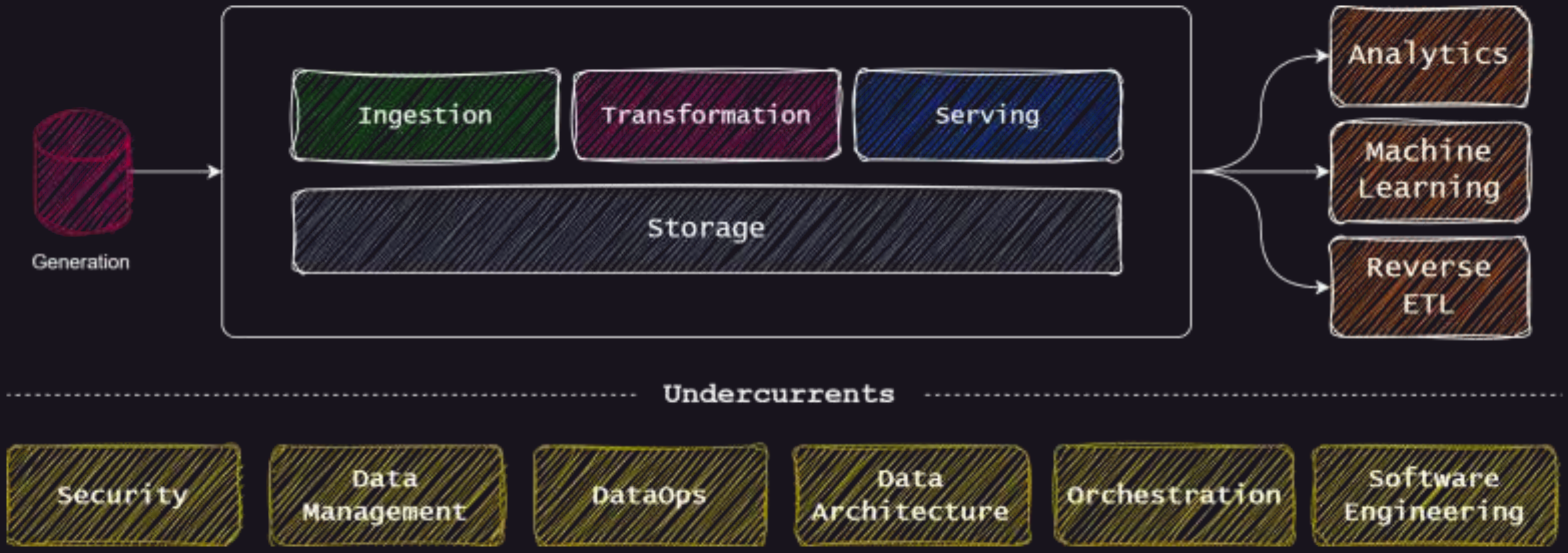
## Data Engineering Defined

Data engineering is the development, implementation, and maintenance of systems and processes that take in raw data and produce high-quality, consistent information that supports downstream use cases, such as analysis and machine learning.

*Fundamentals of Data Engineering*


Requirement Specification
developing, testing, and deploying data pipelines
Plan
Define
Design
Develop
Test
Deploy
docker

# Docker in Data Engineering Lifecycle



## Ingestion:

Docker can be used to create containers for data ingestion tools and platforms such as **Apache Airflow**, **Apache NiFi**, or custom data connectors.

## Transformation:

Docker facilitates the deployment of data processing frameworks and tools like **Apache Spark**, **Apache Flink** and etc.
Developers can package data processing algorithms and libraries into Docker containers, ensuring reproducibility and consistency in data transformation tasks.

## Serving and Storage:

Docker containers are used to deploy and manage databases, data warehouses, and storage solutions such as **PostgreSQL**, **MySQL**, **MongoDB**, or **Apache Hadoop**.
Docker volumes provide persistent storage for data-intensive applications, ensuring data durability and resilience across container restarts or updates.

## Monitoring and Maintenance:

Docker facilitates the deployment of monitoring and logging solutions such as Prometheus, Grafana, ELK stack
Containerized monitoring agents can collect metrics and logs from Dockerized applications and infrastructure components, providing visibility into system performance and health.

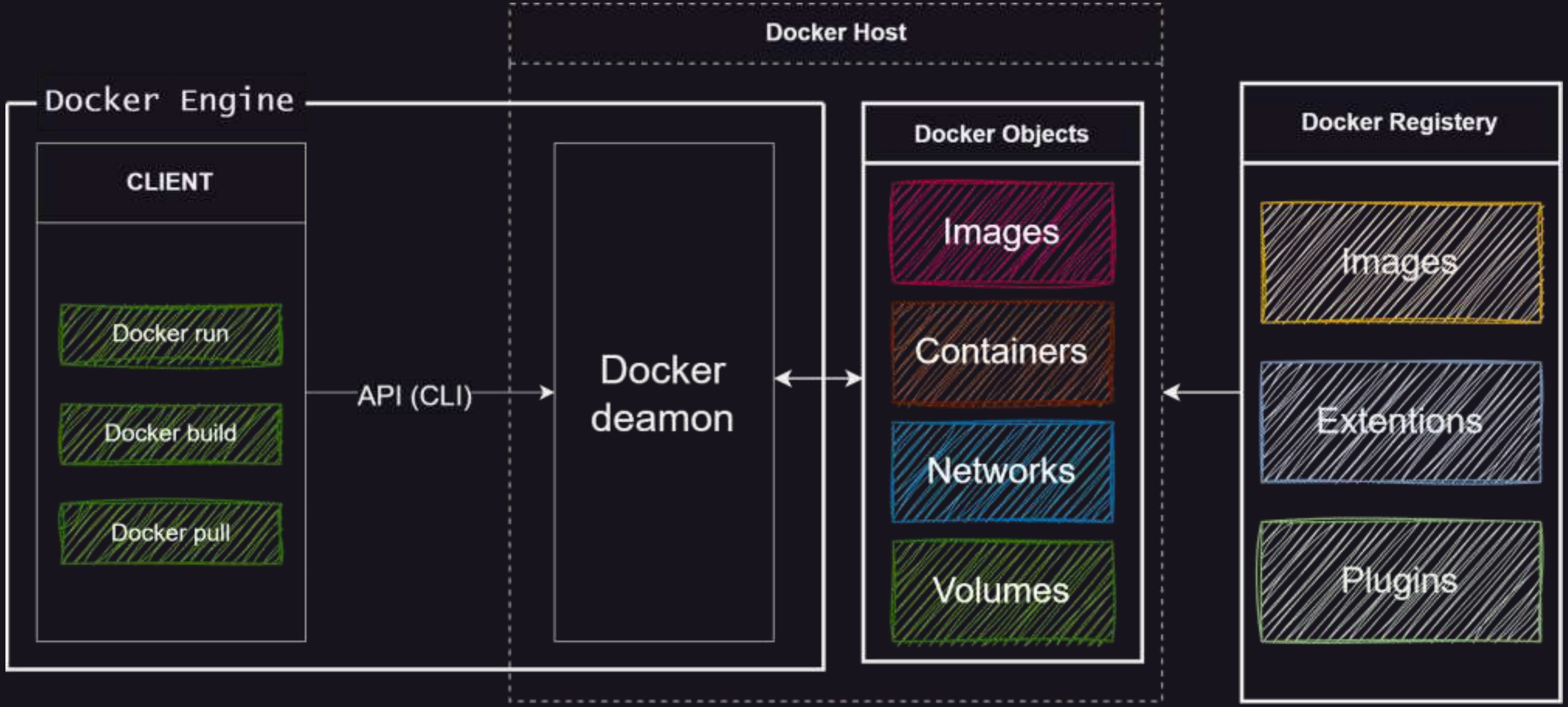
## Docker Basics

## Docker Basic Definitions

| | Description |
|---|---|
| **Docker Engine** | Core component enabling containerization, consisting of Docker daemon (dockerd), Docker API and Command Line Interface (CLI). |
| **Docker Image** | Read-only template containing everything needed to run an application, built from a Dockerfile. |
| **Docker Container** | Lightweight, executable package created from a Docker image, running in an isolated environment on a host system. |
| **Dockerfile** | Text file with instructions for building a Docker image, defining base image, dependencies, and commands to run. |
| **Docker Registry** | Centralized repository storing and distributing Docker images, commonly Docker Hub or private registries. |
| **Docker Compose** | Tool for defining and running multi-container Docker applications using YAML configuration (docker-compose.yml) |
| **Docker Swarm** | Built-in orchestration tool for managing clusters of Docker hosts, enabling deployment, scaling, and management of containerized applications. |
| **Docker Networking** | Enables communication between containers and other networked resources, providing virtual networks and port exposure. |
| **Docker Volumes** | Persistent storage mechanisms used to persist data generated by containers, allowing data sharing and management independently of container lifecycle. |
| **Container Lifecycle** | The lifecycle of Docker containers, including creation, start, stop, deletion, and how to manage container states. |
| **Build** | Process of creating a Docker image from a Dockerfile, executing instructions to assemble the image layer by layer. |

## Docker Engine

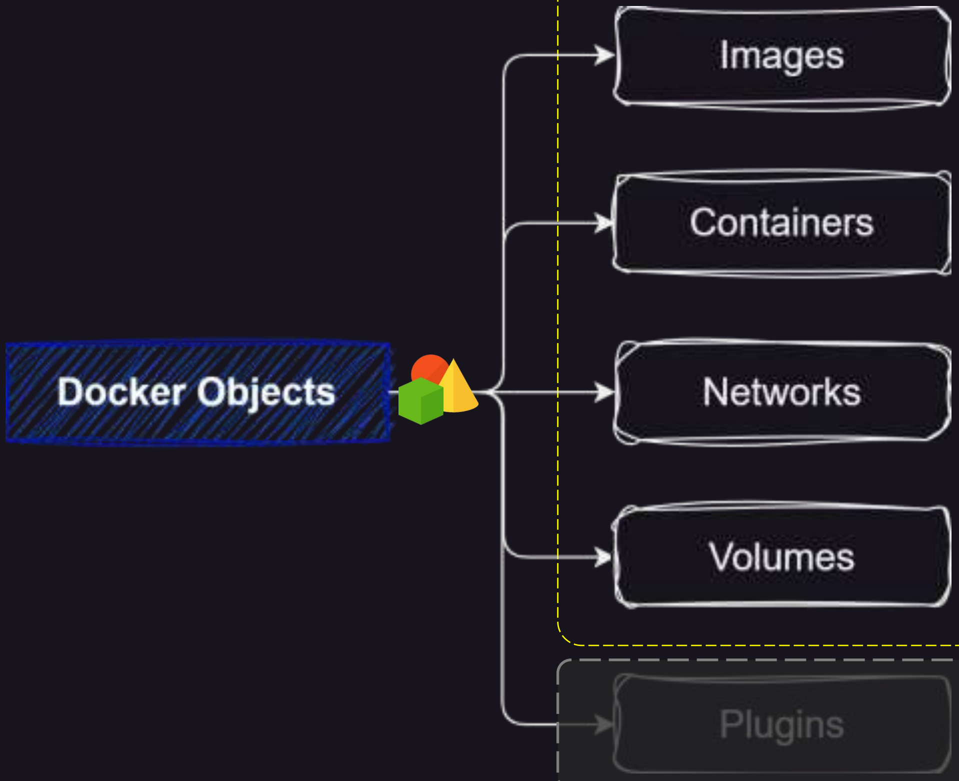
The Docker Engine is the core of Docker. It is a **client-server** application comprising a long-running server process called the Docker daemon (dockerd) and a command-line interface called the Docker client (docker).



## Docker Objects

## Docker Objects

Docker objects create the core of the whole system and enable containerization in general.



Docker images are read-only templates used to create Docker containers. Images contain application code, runtime dependencies, libraries, and other configuration settings required to run an application. Images are built using Dockerfiles and can be stored in Docker registries.

Docker containers are lightweight, standalone, and executable instances of Docker images. Containers encapsulate an application and its dependencies, running in an isolated environment on a host system.

Docker networks provide communication channels for containers running on the same host or across different hosts. Docker supports various network drivers, including bridge networks, overlay networks, and host networks. Networks enable containers to communicate with each other and with external systems.

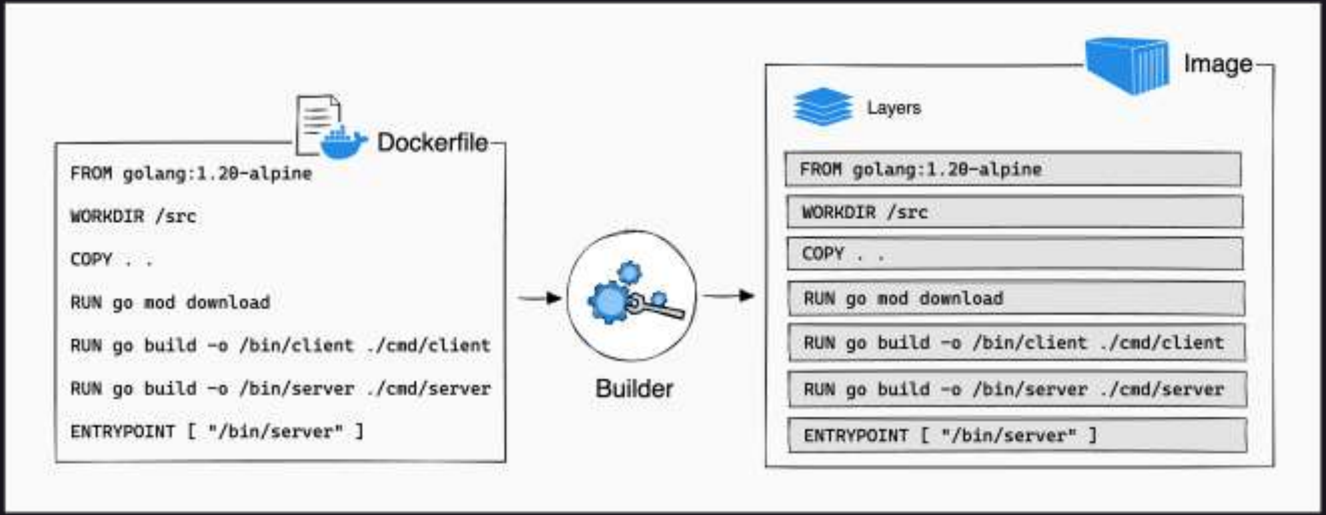
Docker volumes are persistent storage mechanisms used to persist data generated by containers. Volumes provide a way to share and manage data between containers and the host system. They allow data to persist across container restarts and be managed independently of the container lifecycle.

Docker plugins extend the functionality of Docker by adding new capabilities and features. Plugins can be used to integrate Docker with external storage systems, logging solutions, networking providers, and more. They allow users to customize and extend Docker to meet specific requirements.

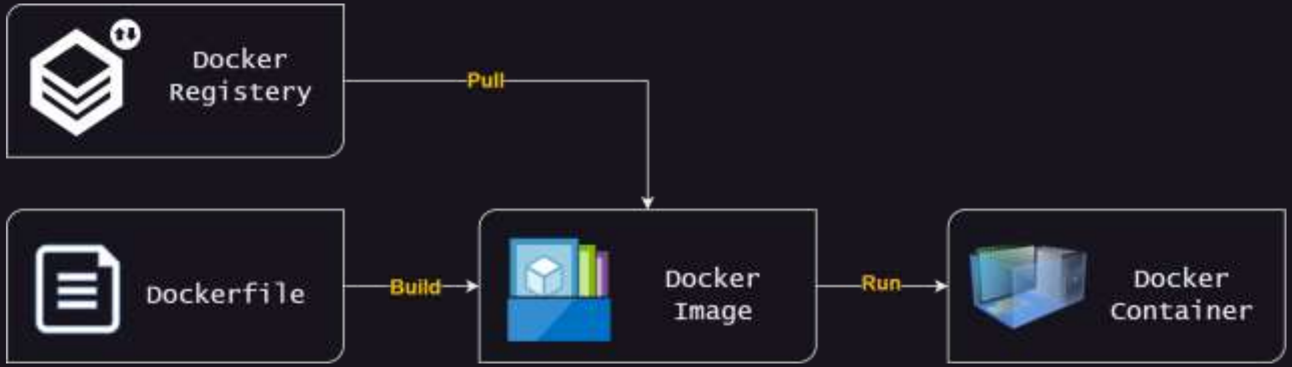
# Docker Image

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

*You might create your own images or you might only use those created by others and published in a registry.* To build your own image, you create a **Dockerfile** with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.



```
FROM golang:1.20-alpine
WORKDIR /src
COPY . .
RUN go mod download
RUN go build -o /bin/client ./cmd/client
RUN go build -o /bin/server ./cmd/server
ENTRYPOINT [ "/bin/server" ]
```

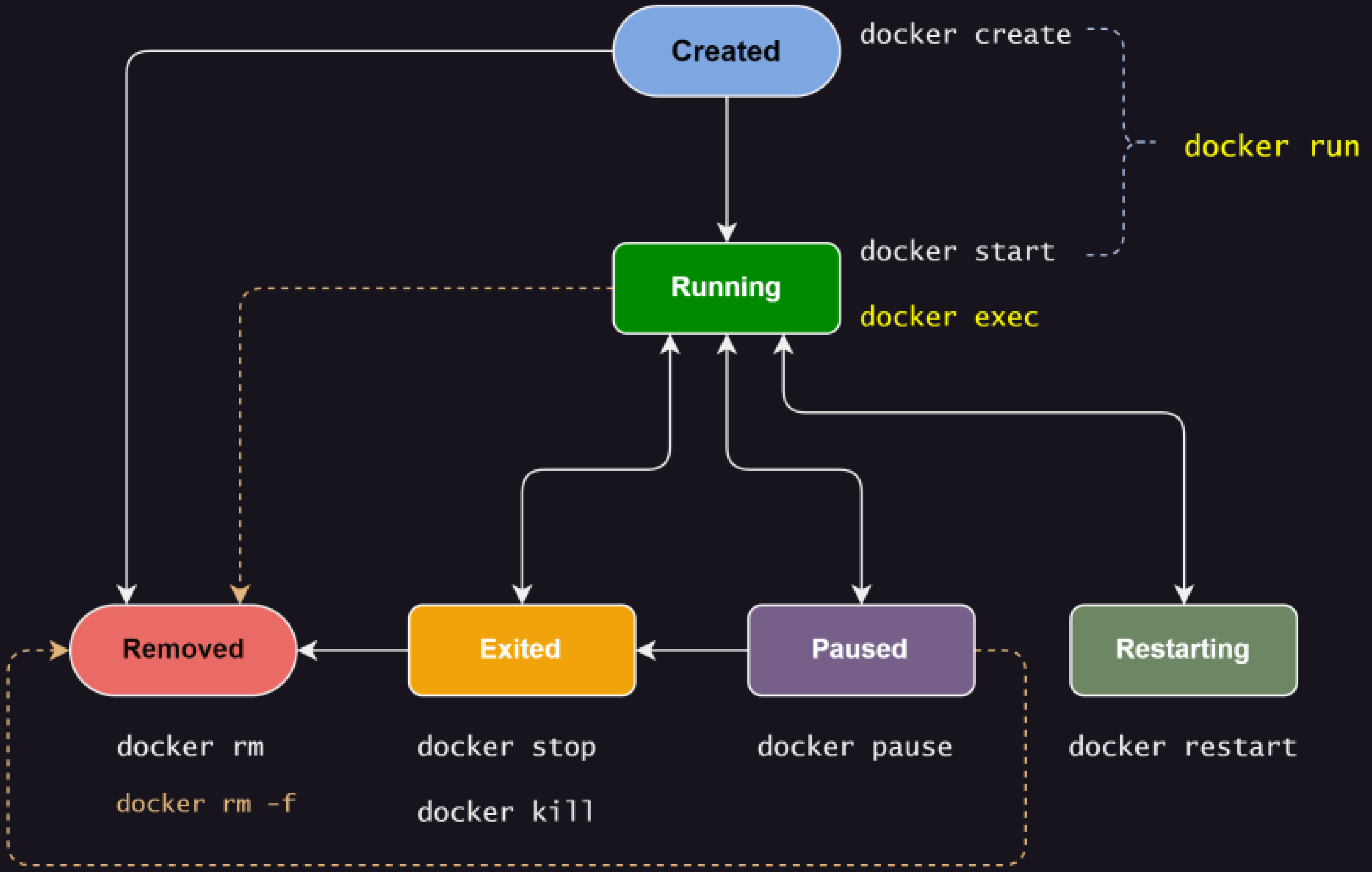


# Docker Container

Docker containers are lightweight, standalone, and executable instances of Docker images.

Each container encapsulates an application and its dependencies, running in an isolated environment on a host system.

Containers can be started, stopped, deleted, and managed using Docker commands.



# Docker Volumes (Storage)

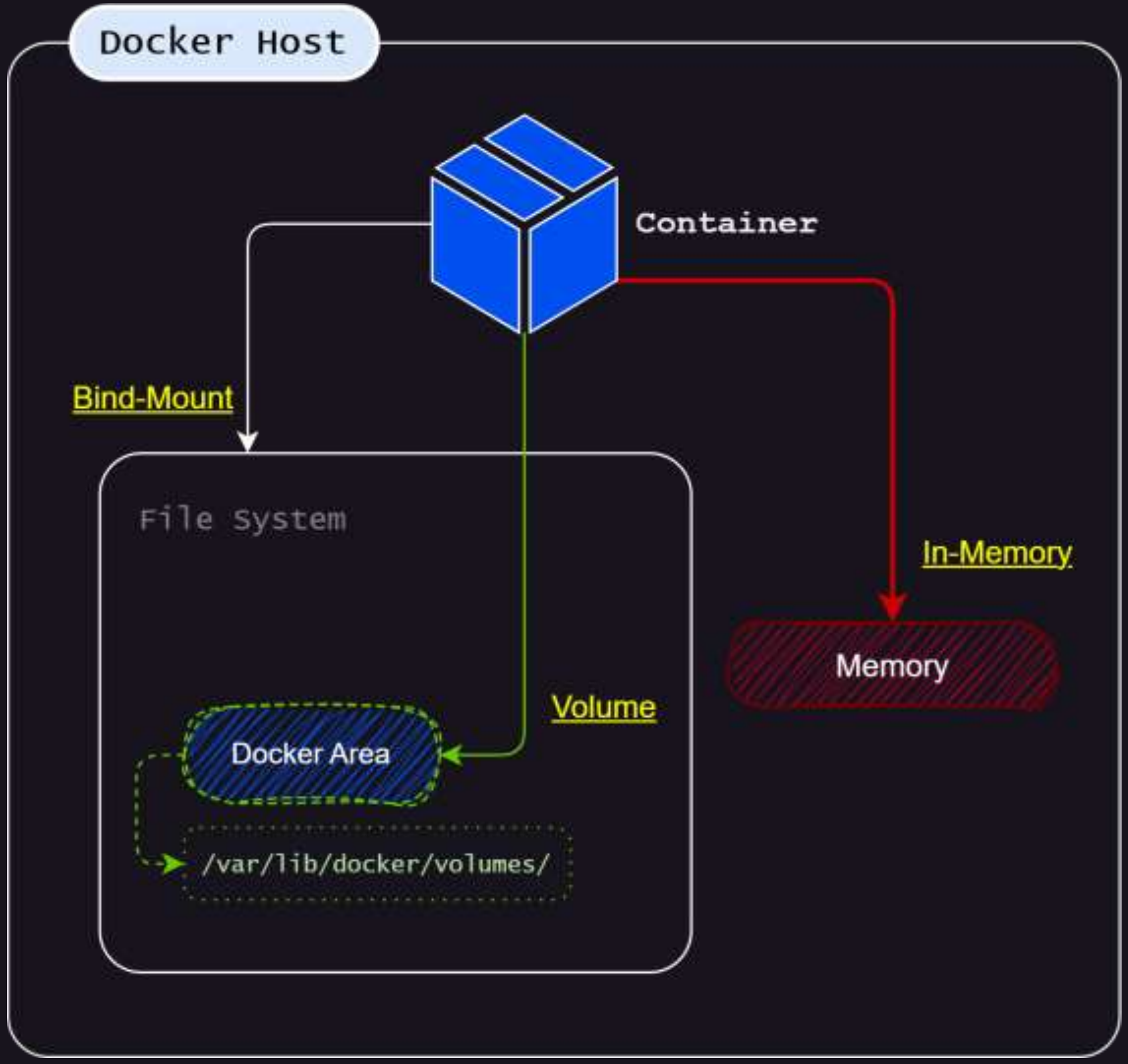
Docker containers are designed to be ephemeral, meaning they are meant to run for a specific purpose and can be easily recreated.
what about data? If your application relies on persistent data storage, Docker offers a couple of mechanisms to handle it:

**1- Volumes:**
Volumes in Docker provide a dedicated storage layer, ensuring data persistence independent of container lifecycle. By creating and mounting volumes to specific directories, data remains intact even through container restarts or removals.

**2- Bind Mounts:**
Bind mounts allow you to map a directory on the Docker host machine directly into a container's file system. Compared to volumes, bind mounts require less configuration as you directly link a host directory.

---

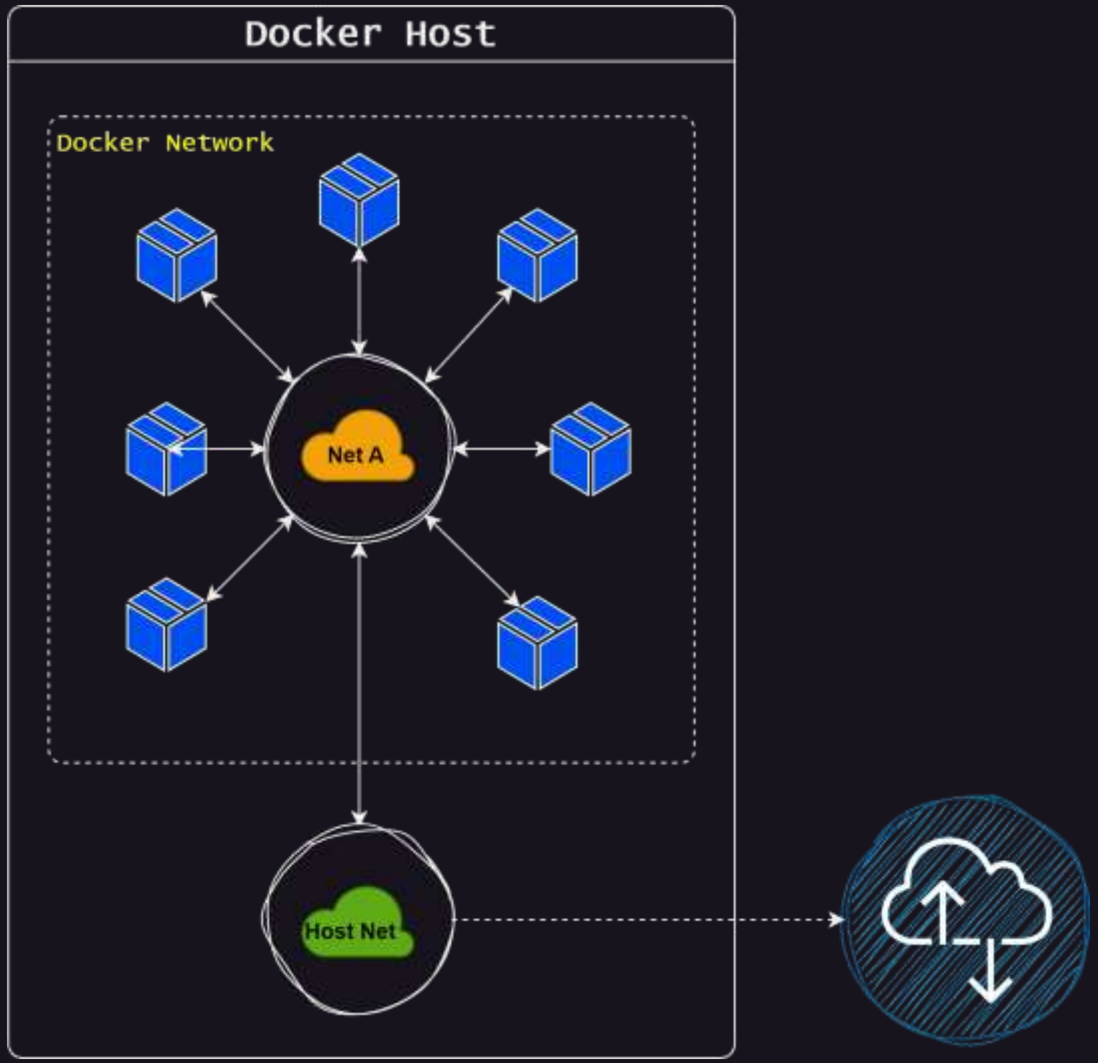
In most cases, volumes are the way to go for data persistence in Docker. They offer better flexibility, data management, and portability compared to bind mounts. Bind mounts might be suitable for simple scenarios where you need to access a specific host directory within your container, but for persistent data storage, volumes are the recommended approach.



# Docker Networks

**Networks:** A group of containers that can communicate with each other. Docker provides different network drivers like bridge, overlay, and macvlan to create various network types. Docker networking enables communication between Docker containers, between containers and the host system, and even between containers across different Docker hosts.

| Driver | Description |
|---|---|
| bridge | The default network driver. |
| host | Remove network isolation between the container and the Docker host. |
| none | Completely isolate a container from the host and other containers. |
| overlay | Overlay networks connect multiple Docker daemons together. |
| ipvlan | IPvlan networks provide full control over both IPv4 and IPv6 addressing. |
| macvlan | Assign a MAC address to a container. |



# Hands-on Docker

Part-02

## Interacting with Docker

**Docker provides two primary interfaces for interacting with Docker**

### Command-line interface (CLI)

The Docker CLI is a command-line tool that allows users to interact with Docker through a set of commands. It provides a convenient and intuitive way to manage Docker resources such as images, containers, networks, and volumes.

### Docker Engine API / SDK for Go and Python

Docker provides an API for interacting with the Docker daemon (called the Docker Engine **API**), as well as **SDKs** for Go and Python. The SDKs allow you to efficiently build and scale Docker apps and solutions. If Go or Python don't work for you, you can use the Docker Engine API directly.
The Docker Engine API is a RESTful API accessed by an HTTP client such as **wget** or **curl**, or the HTTP library which is part of most modern programming languages.

## Docker Command Structure

```
docker <command> <subcommand> [options] [arguments]
```

This is the main action you want Docker to perform. Examples include run, pull, build, exec, ps, stop, start, etc.
Each command and subcommand corresponds to a specific operation or task you want to carry out with Docker.

Options/Flags modify the behavior of the command. They are preceded by one or two dashes (- or --). Options provide additional instructions to Docker on how to execute the command. Some options are required, while others are optional. Examples of options include -it, --rm, -p, -v, --network, --name, --restart, etc.

Arguments are typically the elements of the command that follow the options. They provide specific inputs or parameters required for the command to execute properly. The arguments can vary depending on the command being used. For example, with the docker run command, the argument is usually the name of the Docker image you want to run as a container.

**Example**

$$\texttt{docker}\ \underbrace{\texttt{run}}_{\text{Command}}\ \underbrace{\texttt{-it --rm}}_{\text{Options}}\ \underbrace{\texttt{ubuntu:latest bash}}_{\text{Arguments}}$$

## Common Commands

**docker info:** Provides detailed information about the Docker installation, including the number of containers and images, storage driver, kernel version, etc.

```
docker info
```

**docker version:** Displays the Docker version information, including the client and server versions.

```
docker version
```

**docker ps:** Lists all running containers, including their container IDs, names, status, and ports.

**docker ps -a:** Lists all containers (both running and stopped).

```
docker ps
```

```
docker ps -a
```

**docker images:** Lists all Docker images available locally, including their repository, tag, and size.

```
docker images
docker image ls
docker image list
```

**docker pull:** This command is used to pull Docker images from a registry

```
docker pull
```

**docker run:** This command creates and runs a Docker container from a specified image.

```
docker run
```

**docker stop:** This command stops a running Docker container.

```
docker stop
```

**docker rm:** This command removes one or more Docker containers.

```
docker rm
```

# > docker info

The docker info command provides detailed information about the Docker installation and the environment it's running in. The output of docker info includes various sections that give insights into different aspects of the Docker setup. Here's a breakdown of the typical output and the information it provides:

| Sections | Description |
|---|---|
| Containers | Information about the number of containers running, including how many are currently running, paused, or stopped. |
| Images | Details about the number of images available locally, including the number of dangling (unused) images. |
| Server Version | Docker server version information, including the version number and the API version it supports. |
| Storage Driver | The storage driver used by Docker for managing images and containers. This can include information about the specific storage driver configuration. |
| Logging Driver | The logging driver configured for Docker containers, which determines how container logs are handled. |
| Cgroup Driver | The control group (cgroup) driver used by Docker, which is responsible for resource allocation and management. |
| Plugins | Information about any plugins installed and enabled in the Docker environment, such as volume or network plugins. |
| Memory | Details about the total available memory and the memory limit for containers, if any. |
| Kernel Version | The version of the Linux kernel running on the host system where Docker is installed. |
| Operating System | The name and version of the operating system running on the host. |
| CPUs | Information about the number of CPUs available and their architecture. |
| Registry Mirrors | Any registry mirrors configured for Docker to use for pulling images. |
| Isolation | The isolation technology used by Docker, such as default or hyperv. |

## > docker version

The docker version command provides detailed information about the Docker Engine installed on your system, including the client and server versions, build information, and API version compatibility. When you run the docker version command in your terminal or command prompt, it displays the following information:

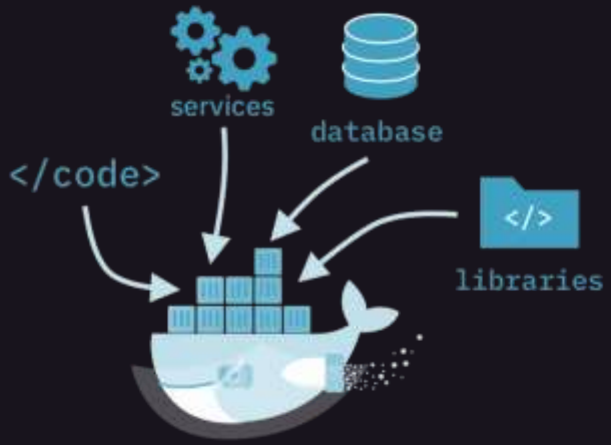
| Sections | Description |
|---|---|
| Client Version | This section provides details about the Docker client installed on your system.<br>▪ Version: The version of the Docker client.<br>▪ API version: The version of the Docker API used by the client.<br>▪ Go version: The version of the Go programming language used to build the Docker client.<br>▪ Git commit: The unique identifier of the Git commit used to build the Docker client.<br>▪ Built: The date and time when the Docker client was built.<br>▪ OS/Arch: The operating system and architecture for which the Docker client was built. |
| Server Version | This section provides details about the Docker server (daemon) installed on your system.<br>▪ Engine: The version of the Docker Engine (daemon) running on the system.<br>▪ API version: The version of the Docker API supported by the server.<br>▪ Go version: The version of the Go programming language used to build the Docker server.<br>▪ Git commit: The unique identifier of the Git commit used to build the Docker server.<br>▪ Built: The date and time when the Docker server was built.<br>▪ OS/Arch: The operating system and architecture on which the Docker server is running.<br>▪ **Built:** The date and time when the Docker server was built.<br>▪ **OS/Arch:** The operating system and architecture on which the Docker server is running.<br>▪ **Experimental:** Indicates whether experimental features are enabled on the Docker server. |

## > docker ps

The `docker ps` command is used to list Docker containers that are currently running on your system. It provides various flags that allow you to customize the output and filter the containers based on different criteria.

| flags | Description | example |
|---|---|---|
| -a, --all | This flag shows all containers, including those that are stopped. By default, `docker ps` only displays running containers. | docker ps -a |
| -q, --quiet | This flag only displays the container IDs, making the output more concise. | docker ps -q |
| -n, --last n | This flag limits the output to the last n containers, where n is a numeric value. | docker ps -n 5 |
| --no-trunc | By default, the output of `docker ps` truncates long container names or IDs. This flag disables truncation, showing the full names or IDs | docker ps --no-trunc |
| -s, --size | This flag displays the total file sizes of the containers. It shows the virtual size and the disk space used by each container. | docker ps -s |
| --format | This flag allows you to customize the output format using a Go template. You can specify which fields to include and their formatting. | docker ps --format "{{.ID}} \| {{.Names}} \| {{.Status}}" |
| --filte | This flag filters the output based on specific criteria, such as status, label, ID, name, etc. You can combine multiple filters using a comma-separated list. | docker ps --filter "status=running" --filter "name=my-container" |
| --latest, -l | This flag shows the latest created container, including all states. | docker ps --latest |

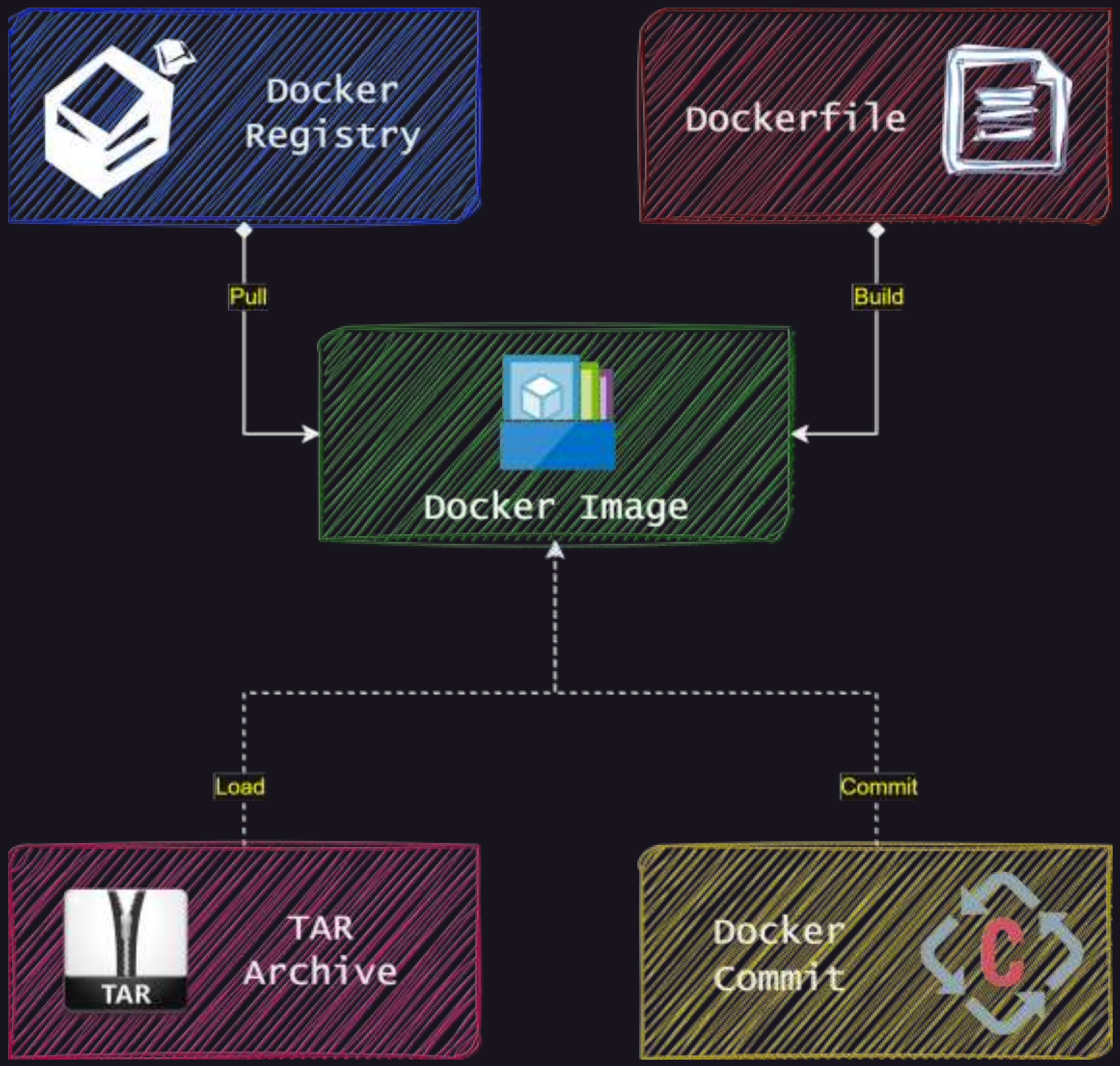
These flags provide flexibility in how you view and filter Docker containers using the docker ps command, allowing you to tailor the output to your specific needs.



## Docker Image

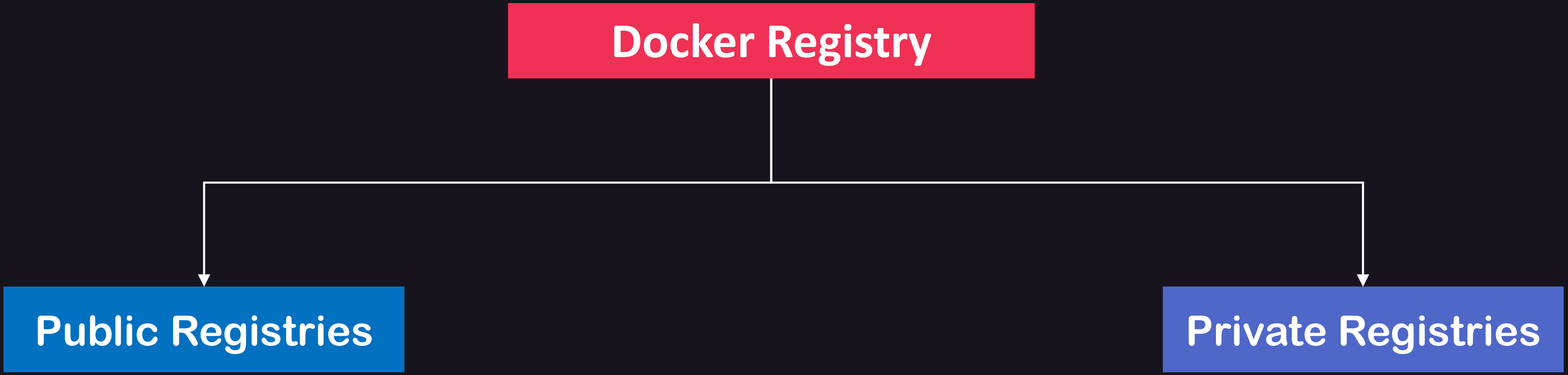
## Docker Image

In Docker, **images** serve as the building blocks for containers.
Images contain everything needed to run a container, including the application code, runtime, libraries, environment variables, and configuration files.



## Docker Registry

A Docker registry is a repository for storing, distributing, and managing Docker images. It serves as a central hub where Docker users can push, pull, and share container images.

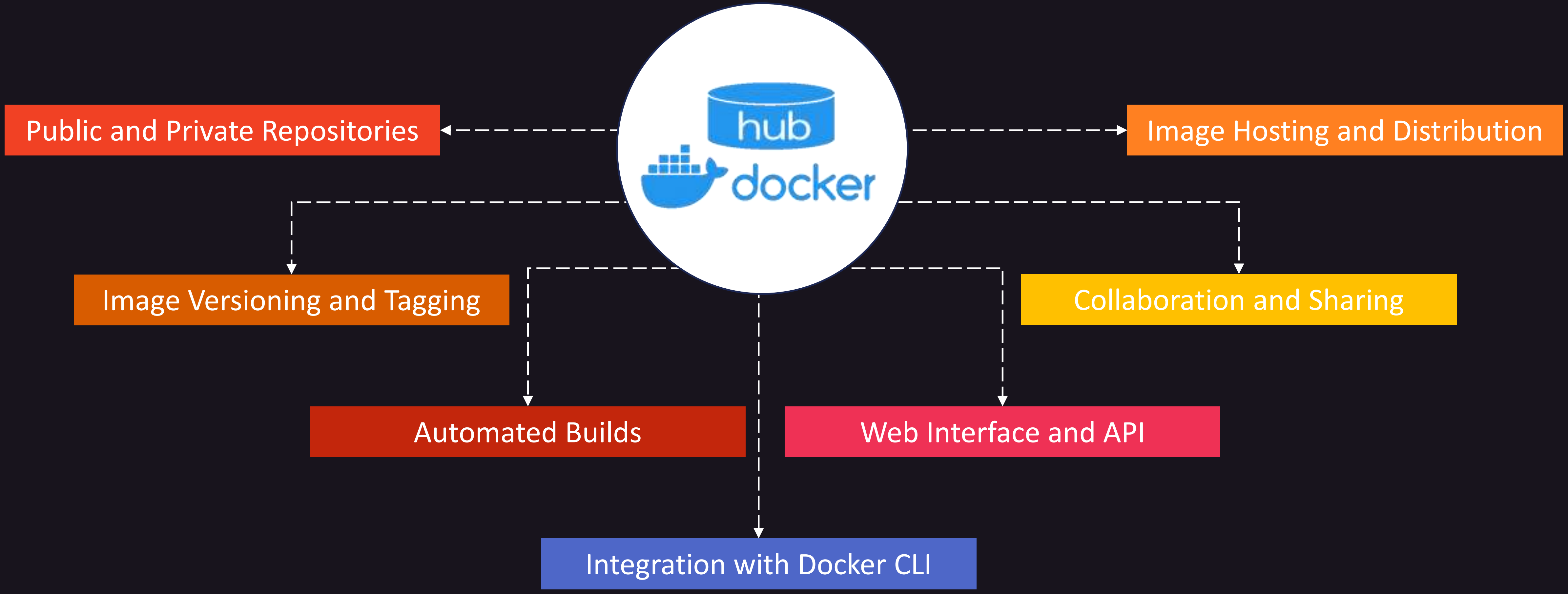


Public registries, such as Docker Hub, are openly accessible and host a vast collection of Docker images contributed by the community.
Docker Hub is the default public registry for Docker images, maintained by Docker, Inc

Private registries are secure repositories for storing proprietary or sensitive Docker images within an organization.
They provide control over image distribution, access control, and image lifecycle management.
Popular private registry solutions include Docker Trusted Registry (DTR), Amazon Elastic Container Registry (ECR), Nexus, and etc.

## Docker Hub

Docker Hub is a cloud-based registry service provided by Docker, Inc. It serves as a centralized repository for storing, sharing, and distributing Docker images.


hub
docker
Public and Private Repositories
Image Hosting and Distribution
Image Versioning and Tagging
Collaboration and Sharing
Automated Builds
Web Interface and API
Integration with Docker CLI

# Hands-on Docker

Part-03

## Pull Images from Docker Hub

1

### Login to Docker Hub

This command prompts you to enter your Docker Hub credentials (username and password) to authenticate and login to Docker Hub.

```
> docker login
```

2

### Search for Images on Docker Hub

This command searches Docker Hub for images that match the specified search term.

```
> docker search <search_term>
```

3

### Pull an Image from Docker Hub

This command downloads the specified Docker image from Docker Hub to your local machine.

```
> docker pull <image_name>
```

4

### List Docker Images

This command lists all Docker images available on your local machine, including images pulled from Docker Hub.

```
> docker images
```

# Docker Image Tags

In Docker, image tags are labels applied to Docker images to differentiate between different versions, variants, or configurations of the same image. Tags provide a way to manage and reference specific versions of Docker images within Docker Hub or other Docker registries.

| | Description |
|---|---|
| Naming Convention | ▪ Docker image tags are appended to the image name using a colon (:) separator.<br>▪ The basic syntax for tagging a Docker image is **image_name:tag.** |
| Purpose | ▪ Tags provide a human-readable and meaningful way to identify and reference different versions or variants of Docker images.<br>▪ They help users and developers to easily manage and track changes to Docker images over time.<br>▪ Tags enable users to pull specific versions of Docker images from Docker Hub or other registries, ensuring consistency and reproducibility in container deployments. |
| Versioning | ▪ Tags are commonly used to denote version numbers or release labels for Docker images.<br>▪ Version tags typically follow semantic versioning conventions (e.g., v1.0, 2.3.1, latest) to indicate major, minor, and patch versions of the image.<br>▪ Semantic versioning helps users understand the compatibility and impact of different versions of Docker images. |
| Aliases and Labels | ▪ In addition to version numbers, tags can also be used to define aliases or labels for Docker images.<br>▪ Aliases provide alternative names for Docker images, making them easier to reference or identify (e.g., ubuntu:latest is an alias for the latest version of the Ubuntu base image).<br>▪ Labels can be used to describe the purpose, configuration, or environment of Docker images (e.g., nginx:alpine refers to the Nginx image based on the Alpine Linux distribution). |
| Default Tag | If no tag is specified when pulling or referencing a Docker image, Docker defaults to using the **latest** tag. |
| Best Practices | ▪ Use descriptive and meaningful tags to identify Docker images and their versions.<br>▪ Follow semantic versioning conventions for version tags to ensure clarity and consistency.<br>▪ Avoid using ambiguous or generic tags like latest for production deployments, as they may lead to unexpected or unintended behavior.<br>▪ Regularly update and manage Docker image tags to track changes, maintain compatibility, and ensure security. |

## Common types of image tags

Docker image tags, such as alpine, slim, and others, denote different variants or configurations of Docker base images. These tags are commonly used to specify specific distributions, flavors, or optimizations for Docker images.

### Alpine:

- Images tagged with alpine are based on the Alpine Linux distribution, which is known for its lightweight and minimalistic nature.
- Alpine Linux images are popular in the Docker ecosystem due to their small size and efficient resource utilization.
- Alpine-based images are often used to create lightweight Docker containers, especially for applications where minimalism and performance are priorities.

### Bookworm:

- Images tagged with buster typically refer to the Debian Buster release, which is a newer version of the Debian Linux distribution.
- Debian Buster images offer updated software packages and libraries compared to Debian Stretch, providing users with access to the latest features and improvements.
- Debian distribution codenames are based on the names of characters from the Toy Story films. Debian's unstable trunk is named after Sid, a character who regularly destroyed his toys.

### Slim:

- Images tagged with slim typically refer to slimmed-down versions of popular base images, such as Debian or Ubuntu.
- Slim images are optimized to reduce image size and minimize dependencies, making them suitable for resource-constrained environments or minimizing attack surfaces.

### Latest:

- The latest tag is a common convention used to refer to the most recent version of a Docker image.
- Images tagged with latest may not necessarily be the most recent or the best choice for production deployments, as they may lack version specificity and stability guarantees.

## Create Docker Image

- https://docs.docker.com/reference/dockerfile/
- https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

# Create Docker Image

A Docker image consists of read-only layers each of which represents a Dockerfile instruction.
The layers are stacked and each one is a delta of the changes from the previous layer.

**Read-Only Layers:**

Docker images are built in a series of steps, each corresponding to an instruction in the Dockerfile. Each step creates a new read-only layer that adds, modifies, or deletes files relative to the previous layer.

**Delta (Change Set):**

The term "delta" emphasizes that each layer contains only the changes from the previous layer, not the entire filesystem. For instance:

- If a layer installs a software package, the delta for that layer would include the files added by the installation.
- If a layer modifies a configuration file, the delta would include the new version of that file.
- If a layer deletes a file, the delta would include the record of that deletion.

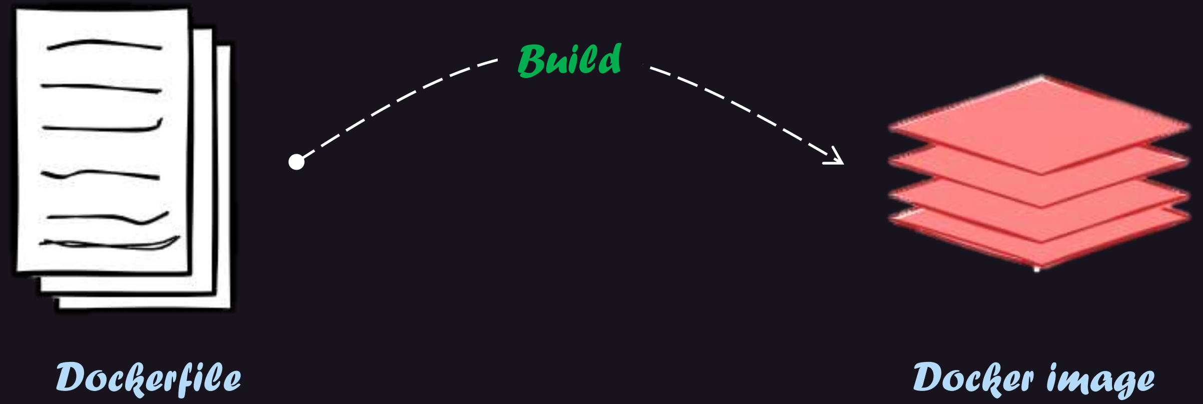
**Layer Stacking:**

Layers are stacked on top of each other in a specific order, starting from the base image layer. When a container is created from an image, it combines these layers into a single unified filesystem. The container's view of the filesystem is the result of applying all the deltas in sequence.

## Create Docker Image

### What is a Dockerfile?

Docker builds images automatically by reading the instructions from a Dockerfile which is a text file that contains all commands, in order, needed to build a given image.

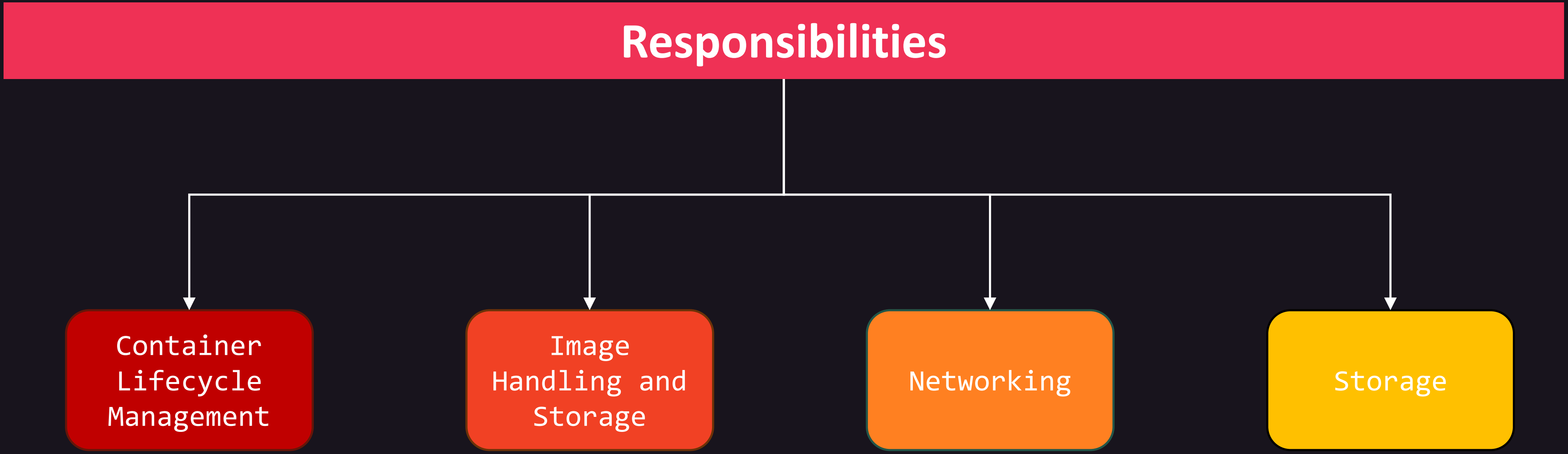


## Docker demon

- https://docs.docker.com/engine/api/

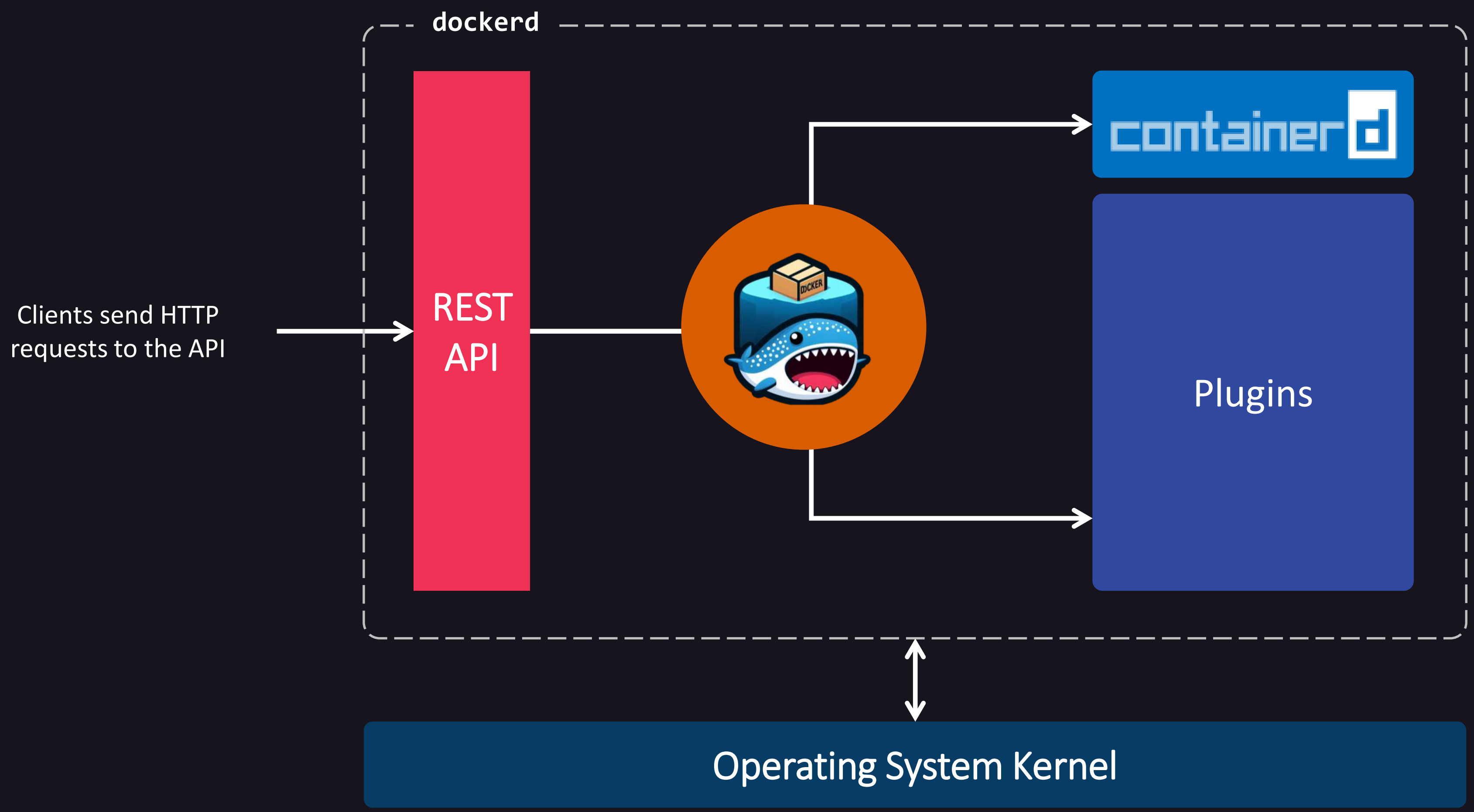
## Docker Daemon

Docker Daemon (dockerd) is the **persistent process** (Long-Running Process) that manages Docker objects such as images, containers, networks, and volumes. It runs on the host machine and handles requests from the Docker client (docker), executing them against the Docker API.



| | | | |
|---|---|---|---|
| Docker Daemon creates, starts, stops, and removes containers as per user commands. | It manages Docker images, pulling them from registries, storing them locally, and distributing them to containers. | Docker Daemon configures network interfaces for containers, enabling communication between containers and the external world. | It manages storage volumes used by containers, allowing for persistent data storage beyond the container lifecycle. |

## Architecture of Docker Daemon

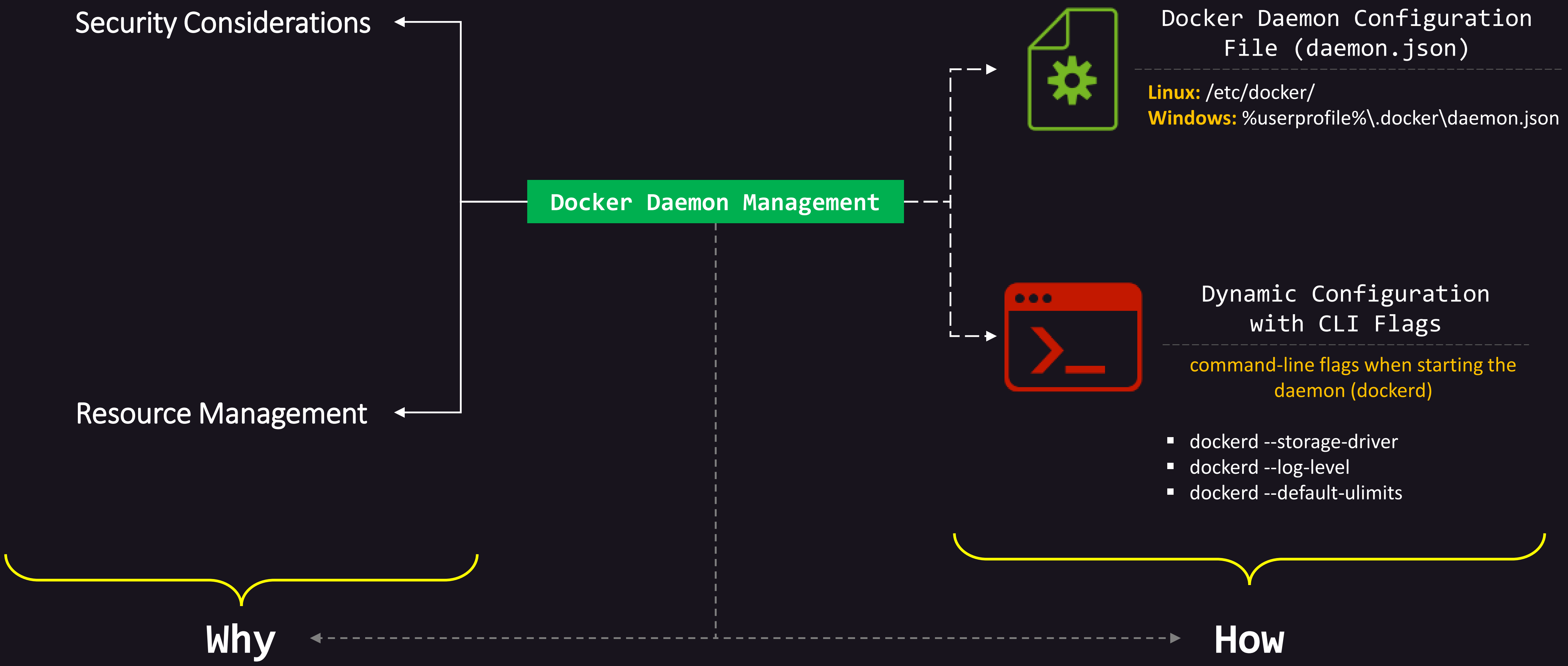


High-level container runtime that manages container lifecycle operations.

Optional extensions that add functionality to Docker Daemon, such as volume plugins for custom storage solutions.

Docker Daemon communicates with the underlying operating system kernel to create and manage containers. It interacts with container runtimes like containerd to execute container lifecycle operations and plugins to extend its functionality.

# Docker Daemon Configuration and Settings


Security Considerations
Resource Management
Docker Daemon Management
Docker Daemon Configuration File (daemon.json)
Dynamic Configuration with CLI Flags
Why
How

**Linux:** /etc/docker/
**Windows:** %userprofile%\.docker\daemon.json

command-line flags when starting the daemon (dockerd)

- dockerd --storage-driver
- dockerd --log-level
- dockerd --default-ulimits

## Docker Daemon Configuration and Settings

docker

### Backup Configuration

Regularly backup the daemon.json file to ensure you can restore Dockerd's configuration in case of accidental changes or system failures.

### Test Changes

Before applying configuration changes to production environments, test them in a development or staging environment to ensure they work as expected and do not cause unexpected issues.

### Monitor Dockerd

Use monitoring tools to track Dockerd's performance, resource usage, and logging output. This helps identify potential issues and optimize Dockerd's configuration.

### Recommended

# Hands-on Docker

Part-04