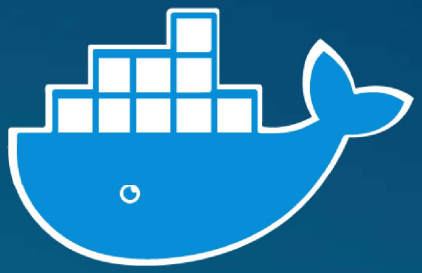
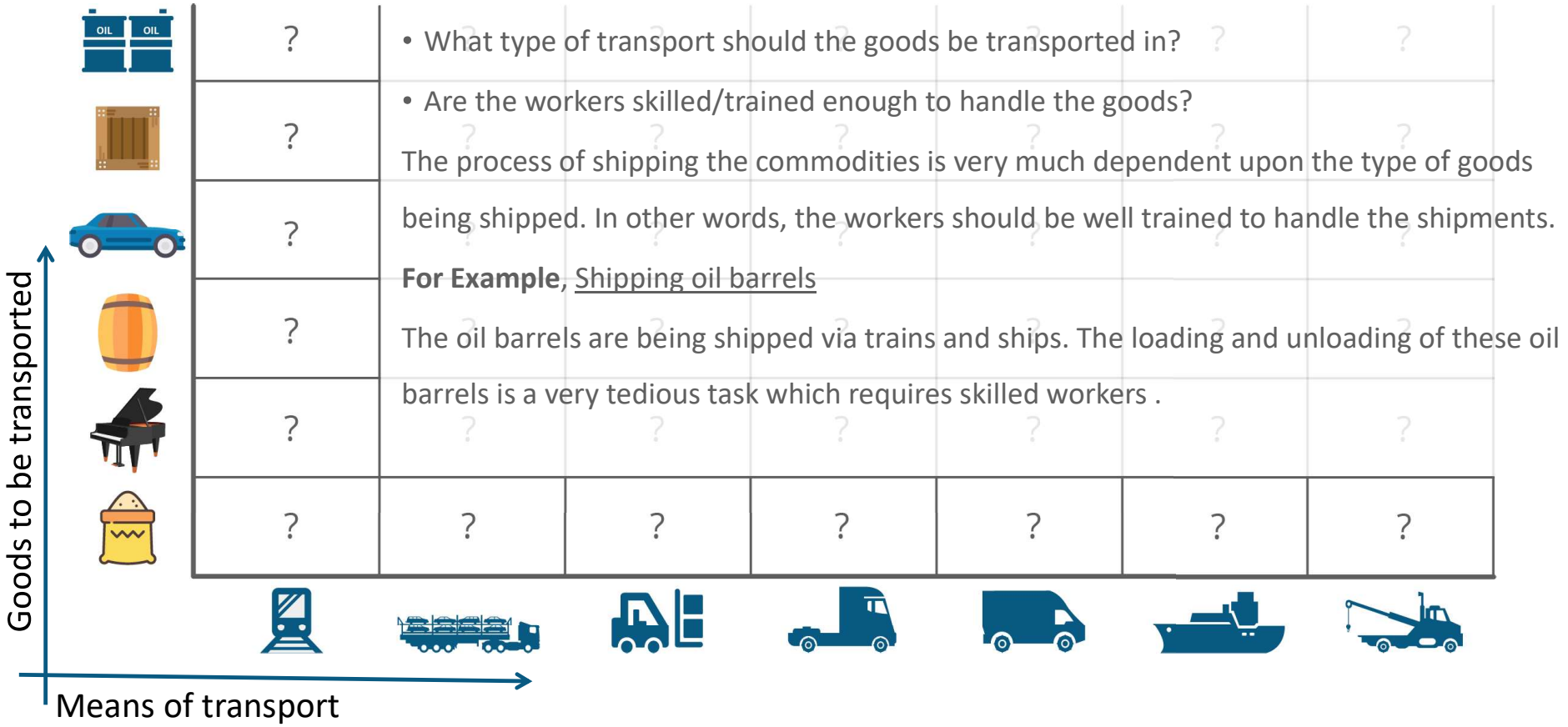
edureka!



Continuous Deployment using Docker

Shipping Challenges



- What type of transport should the goods be transported in?
- Are the workers skilled/trained enough to handle the goods?

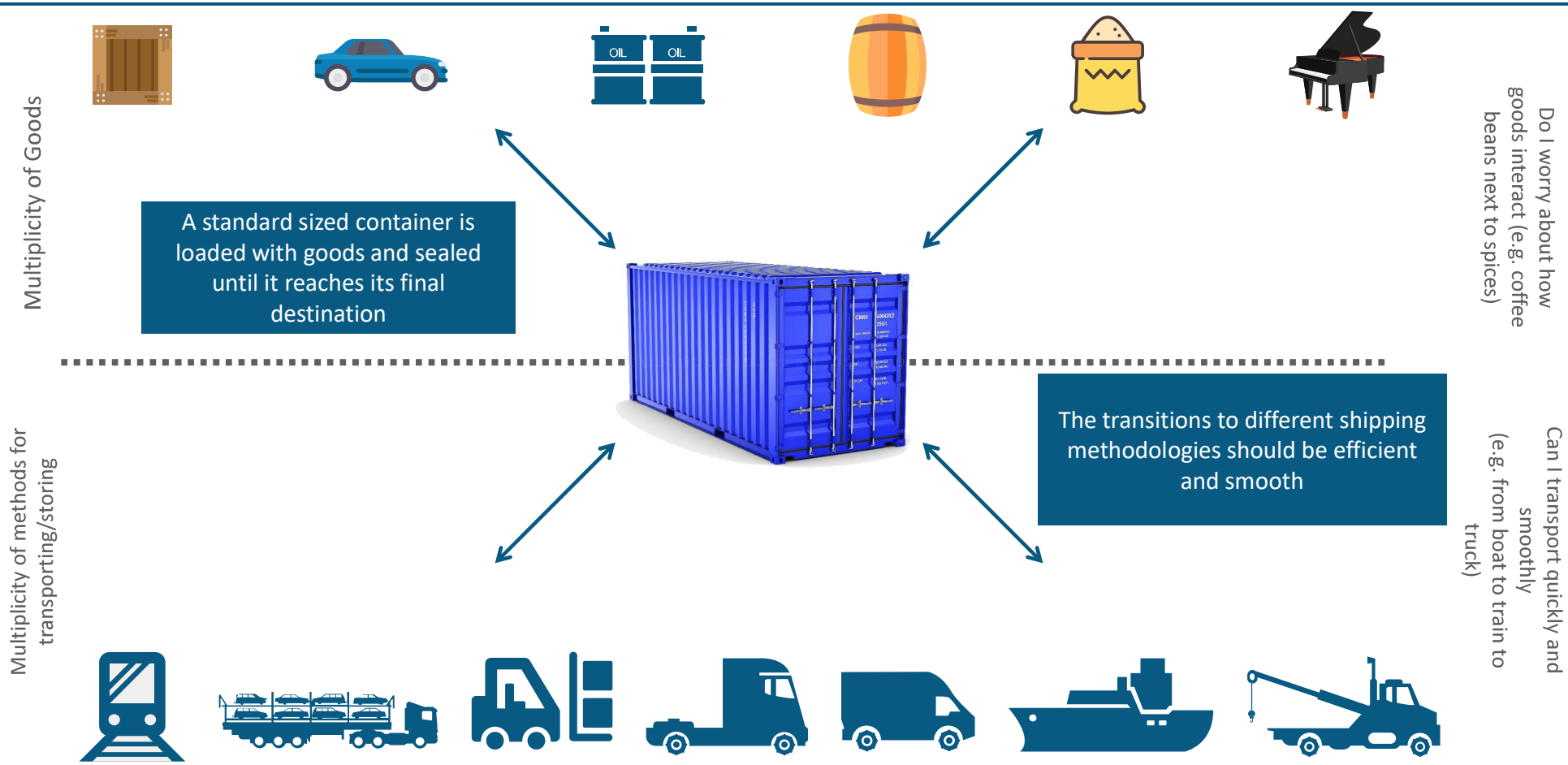
The process of shipping the commodities is very much dependent upon the type of goods being shipped. In other words, the workers should be well trained to handle the shipments.

For Example, Shipping oil barrels

The oil barrels are being shipped via trains and ships. The loading and unloading of these oil barrels is a very tedious task which requires skilled workers .

edureka!

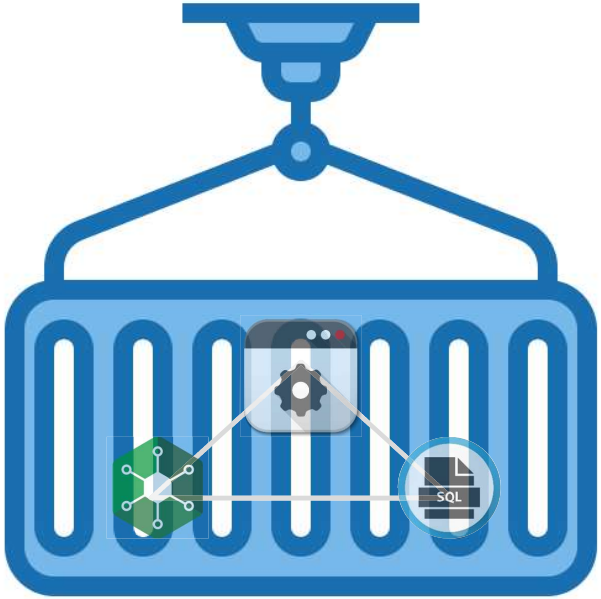
Shipping Solution



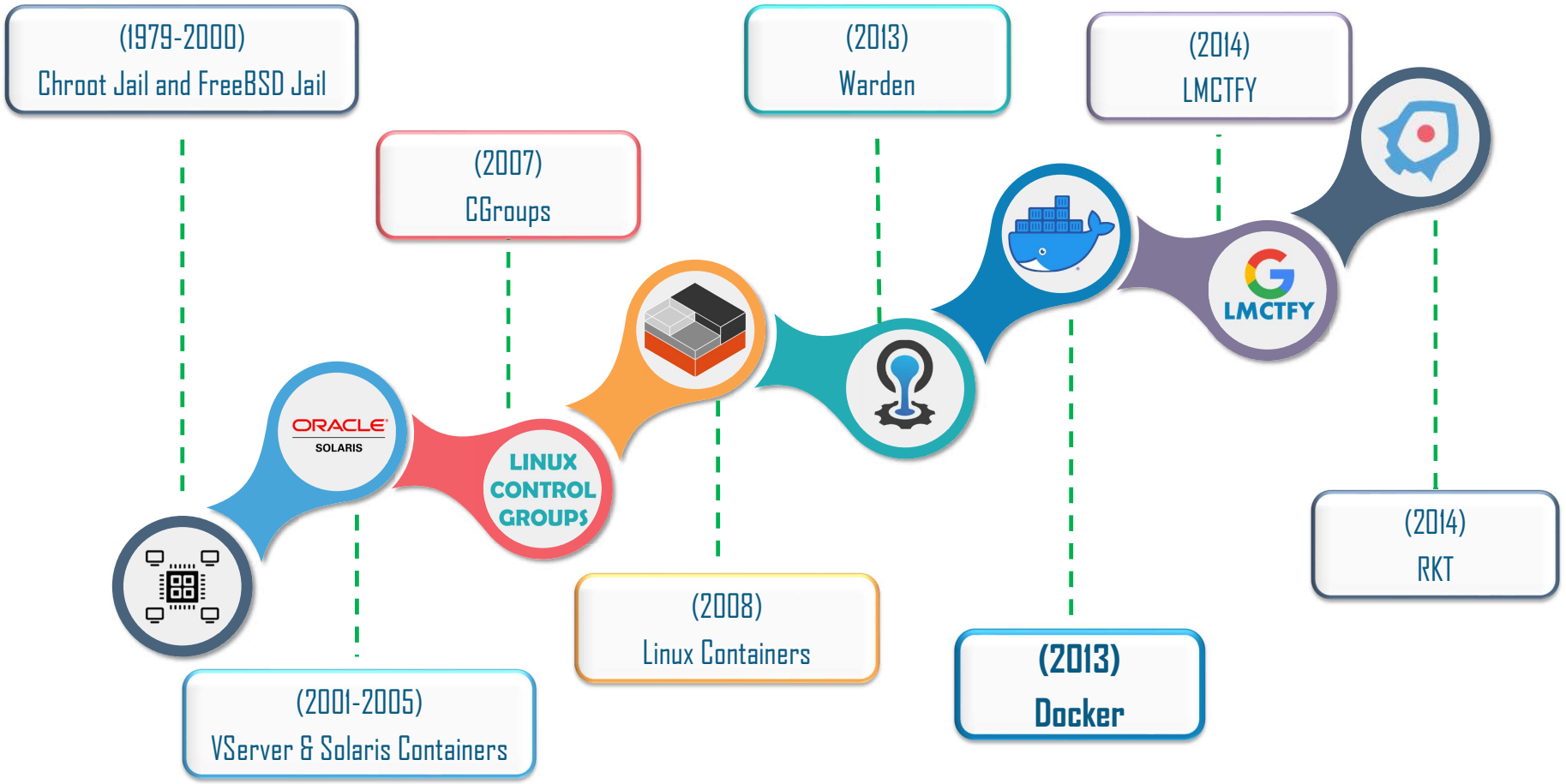
edureka!

What is Containerization?

It is the process of packaging the application and all its dependencies to execute them in an efficient and hassle-free way across different environments. A single unit of this bundled package is known as a **Container**

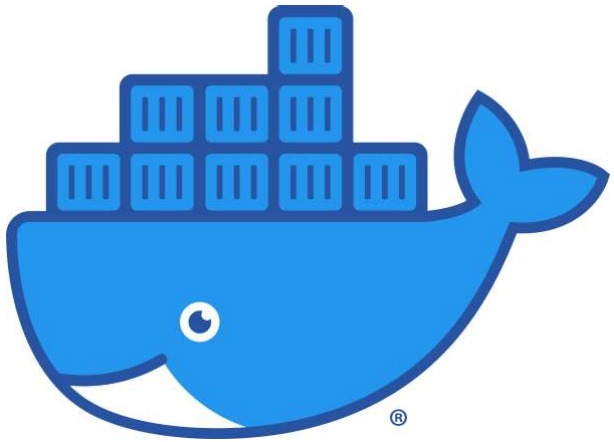


A Short History of Containers (HoC)



edureka!

HoC: Docker



2013

- Docker was made public as an open-source containerization technology in 2013
- It initially used LXC execution engine
- A year later, with the release of Docker v0.9, docker introduced their own component replacing LXC

Why use Containers?

Performance Overhead

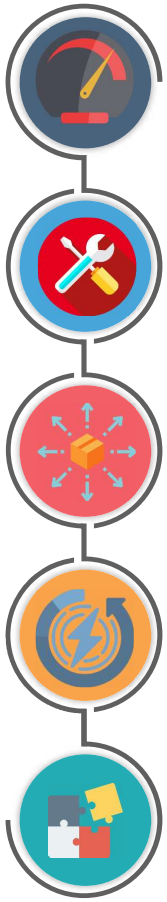
Containers work on top Host OS's Kernel, therefore, there is little to no performance overhead

Platform Independent

Containers can be deployed to platforms with different network topologies and security policies without any hassle

Modularity

Depending upon the approach, containers work seamlessly in a monolithic as well as the microservice environment



Easily Manageable

Since all the dependencies run inside an isolated instance, it is easy to manipulate and make changes to the application

Instant Boot

Containerized application has zero boot time making them available instantaneously

What is a Container?

- It is an Operating System Level Virtualization technology
- Detaches the application and its dependencies from the rest of the system
- utilizes **namespaces(Na)** and **cgroups(CG)** feature of Linux Kernel to isolate processes

Container: Working

Containers utilize two of the Linux Kernel features:

CGROUPS

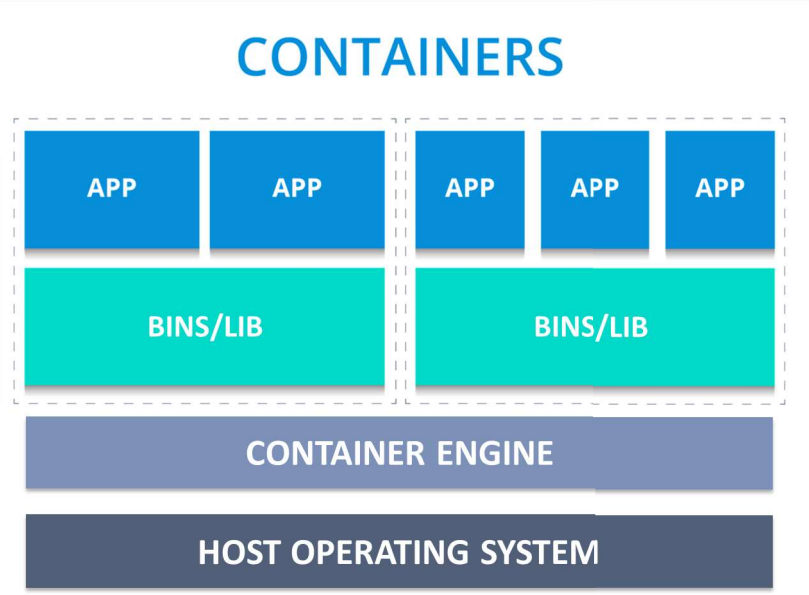
- Control Groups is a Linux Kernel feature
- Allows segregating the processes and the required resources
- Manages the isolated resources and processes as a single module

NAMESPACES

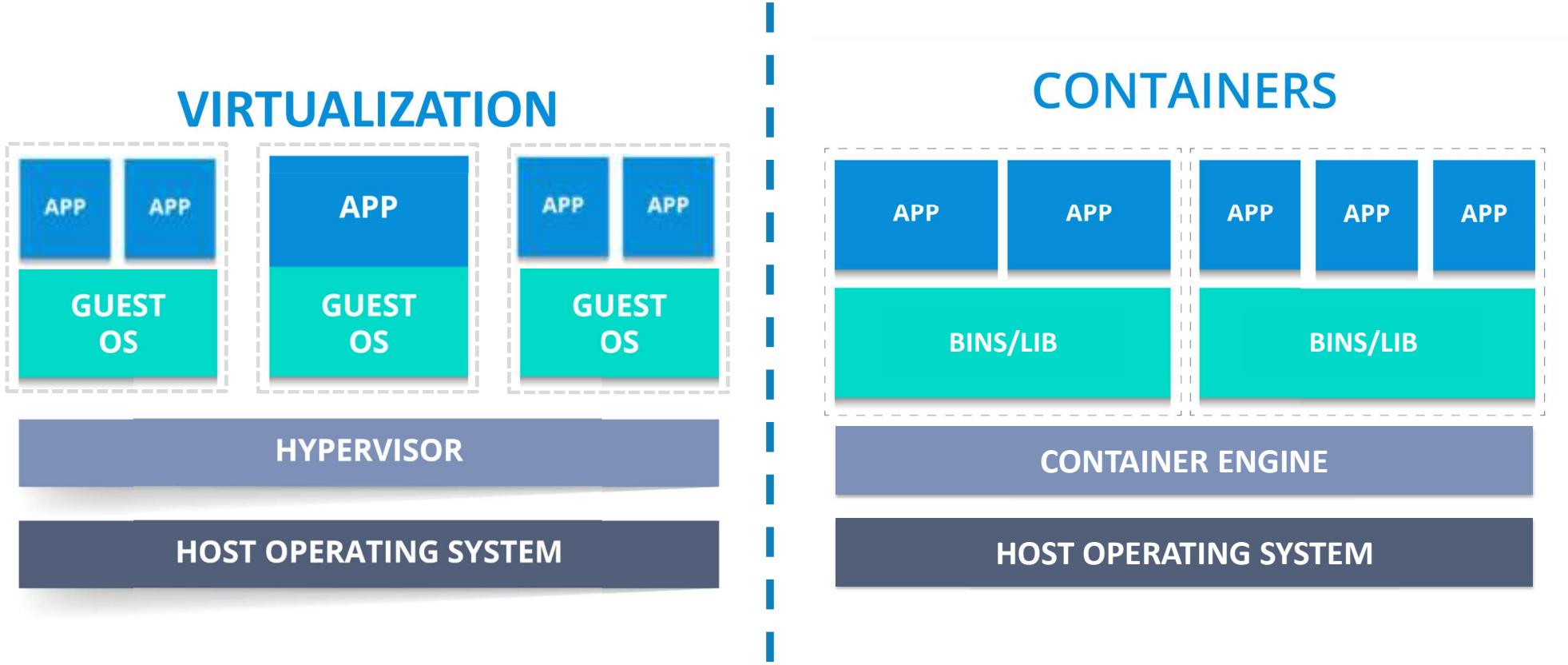
- Used to limit process's access to view the system
- Process is unaware of everything running outside of its own namespace

Container: Working

- A container is just another process for the host operating system
- Works in contained environment under the OS
- Gets restricted view and access to other system processes, resources, and environments



Virtual Machine vs Container

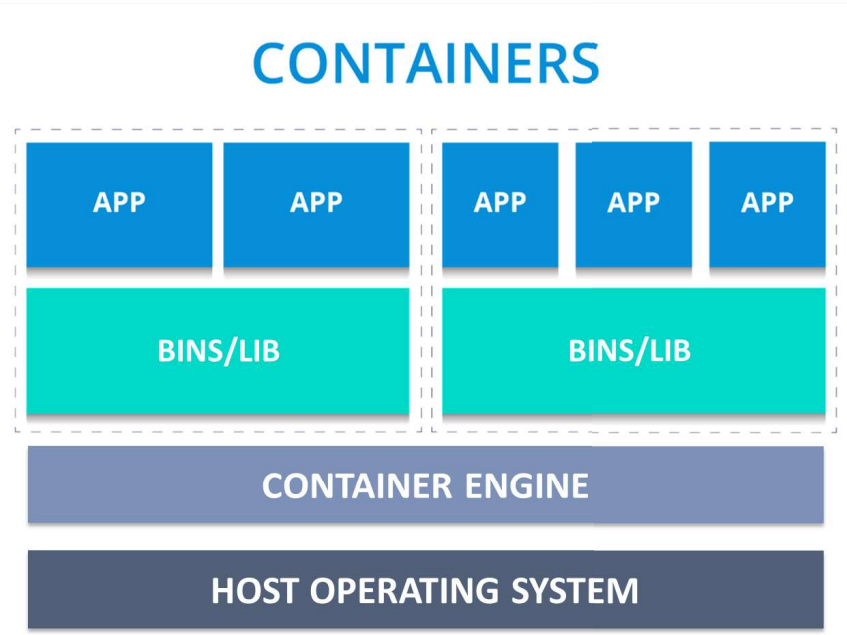


Containers vs Virtual Machines

	Containers	Virtual Machines
Number of Apps on Server	Can run a limited number of applications, depending on VM Configuration	Can run multiple instances of different applications together
Resource Overhead	Light-weight with little to no overhead	Highly resource-intensive with performance overhead of managing multiple Oss
Speed of Deployment	Very fast deployment. Container images require seconds to deploy new instances	Very slow Deployment. Requires setting up OS, app dependencies, etc.
Security	Users usually require root level permissions to execute container related tasks	Provides better security
Portability	Highly portable with emphasis on consistency across environments	Very limited portability

Why use Containers?

- Containers are light compared to a VM, as they directly utilize the Host Operating System's Kernel
- For a single VM instance, the machine hardware deals with 3 separate Kernels



edureka!

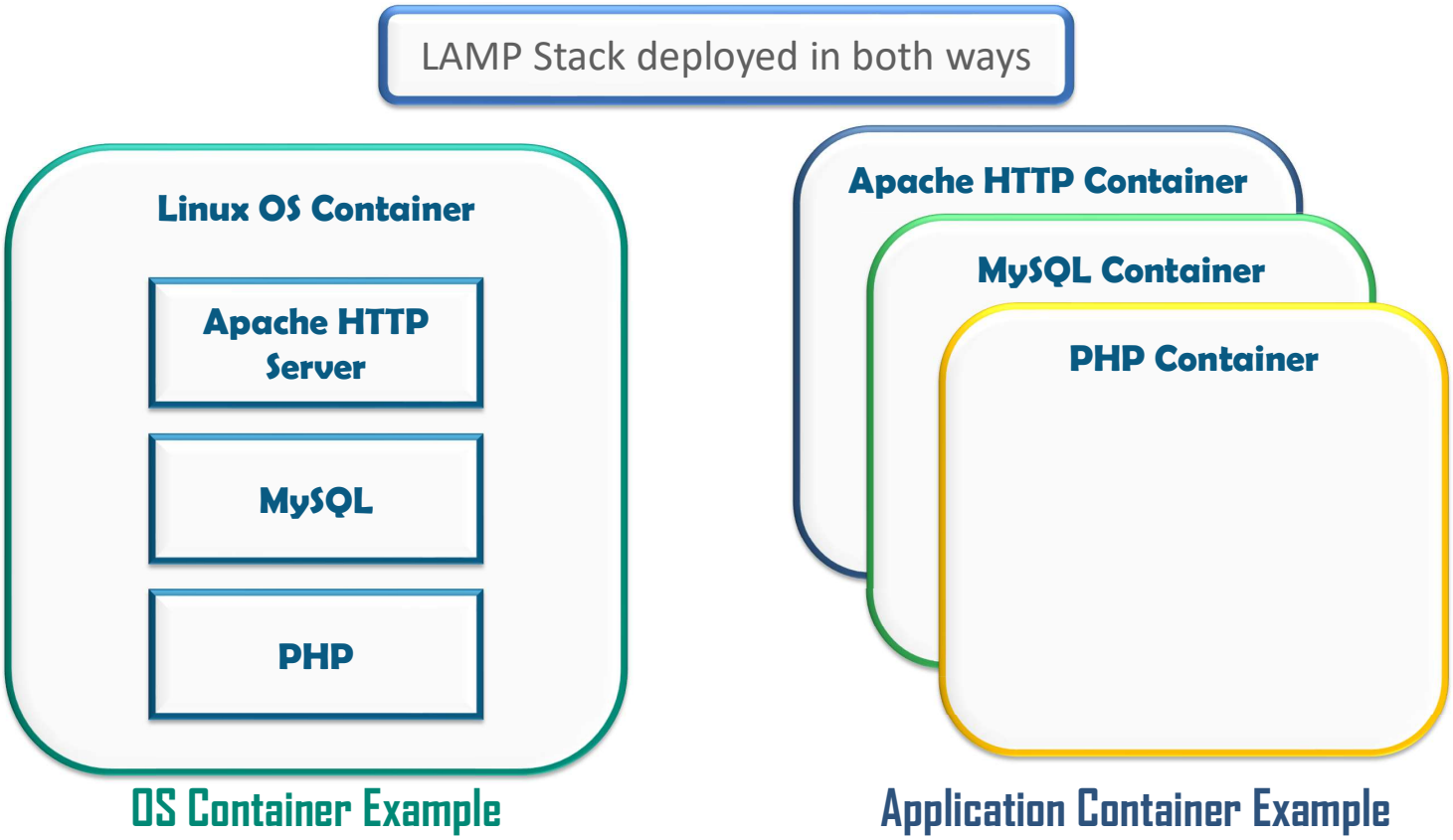
Types of Containers

There are two types of Containers:

OS Containers

Application Containers

Types of Containers: Example

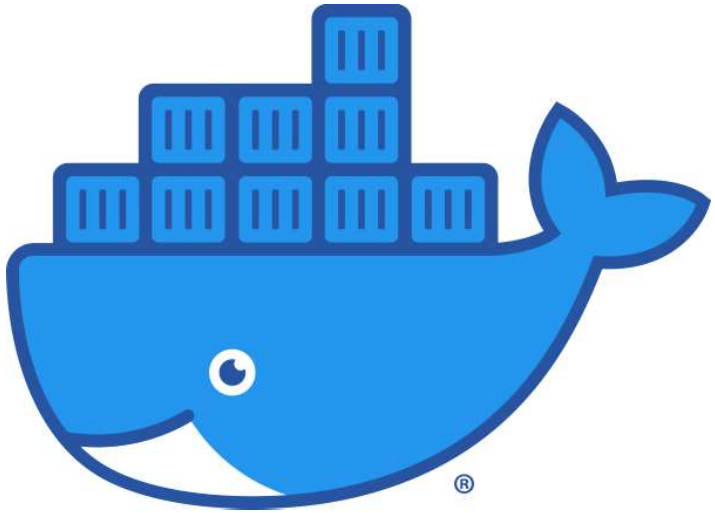


OS Containers vs Application Controller

OS Containers	Application Containers
Runs and functions as a complete OS	Runs as an application isolated from the rest of the system
Executes all the necessary services required for the application in a single container	Runs one single process/application per container
Useful when multiple different distros are required	Useful when process isolation is required to run the application directly
Example: OpenVZ, Solaris Containers, LXC, Linux VServer, etc.	Example: Docker, rkt, etc.

Introduction to Docker

Docker is one of the most popular Container engines today because of the way it handles containers



Docker: Features

Fast Configuration

- The set up process is very quick and easy
- It separates the requirements of the infra from the requirements of the application

Application Isolation

- Provides applications/service isolation
- Applications inside containers execute independent of the rest of the system

Productivity

- Rapid deployment, in turn, increases the productivity
- Reduced resource utilisation is another factor increasing the productivity

Swarm

- It helps clustering and scheduling docker containers
- It enables controlling cluster of docker hosts from a single point

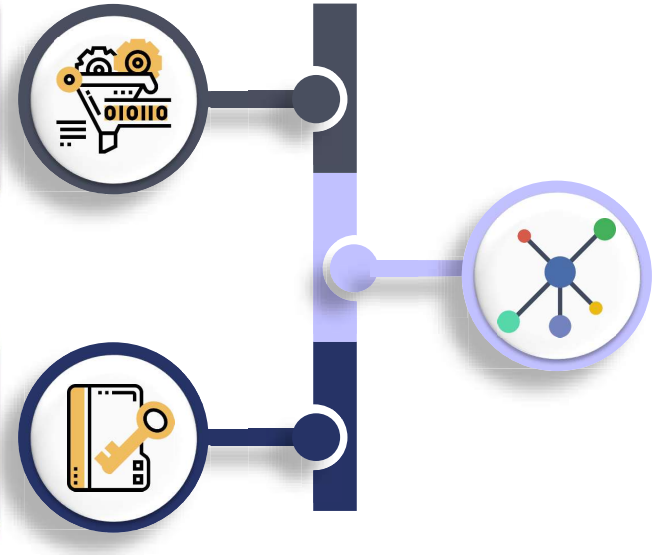
Docker: Features (Contd.)

Services

- Services use a set of tasks to define the state of a container inside a cluster
- Swarm manager takes the defined service as input and schedules it on the nodes

Security

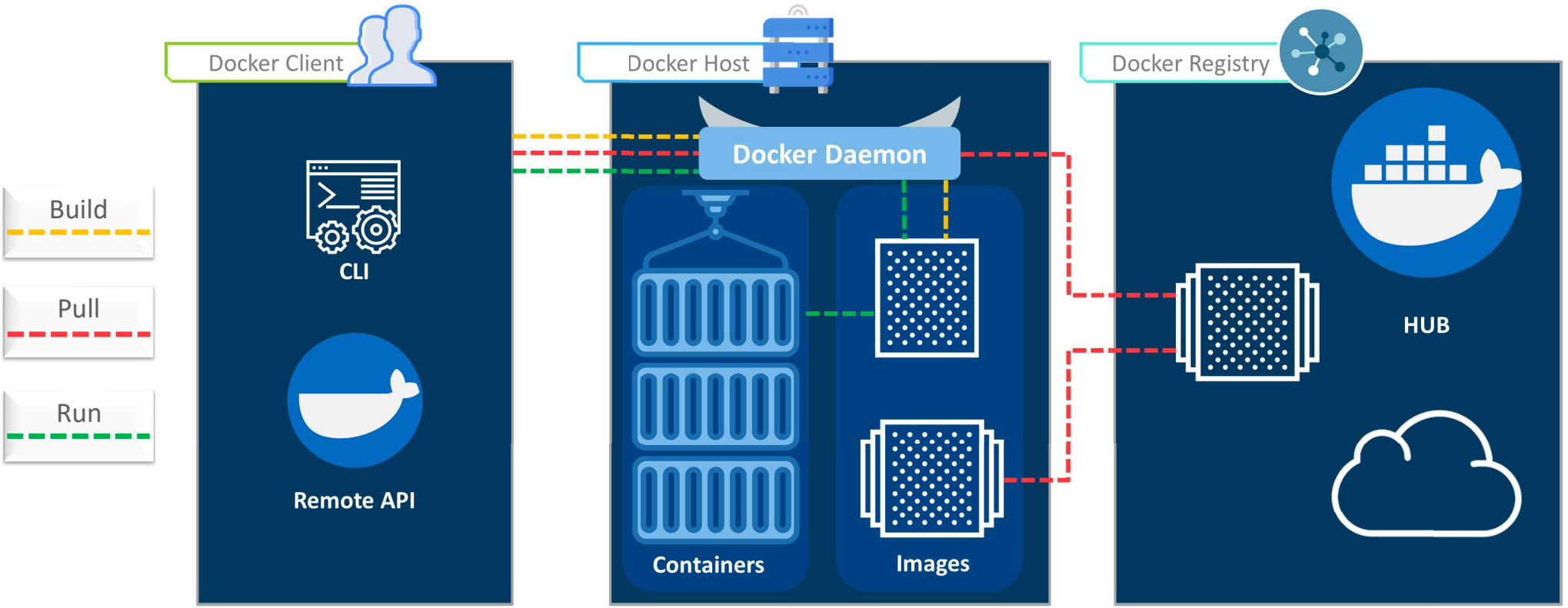
- Allows saving secrets inside the swarm cluster
- These secrets can then be accessed by the required service



Service Discovery

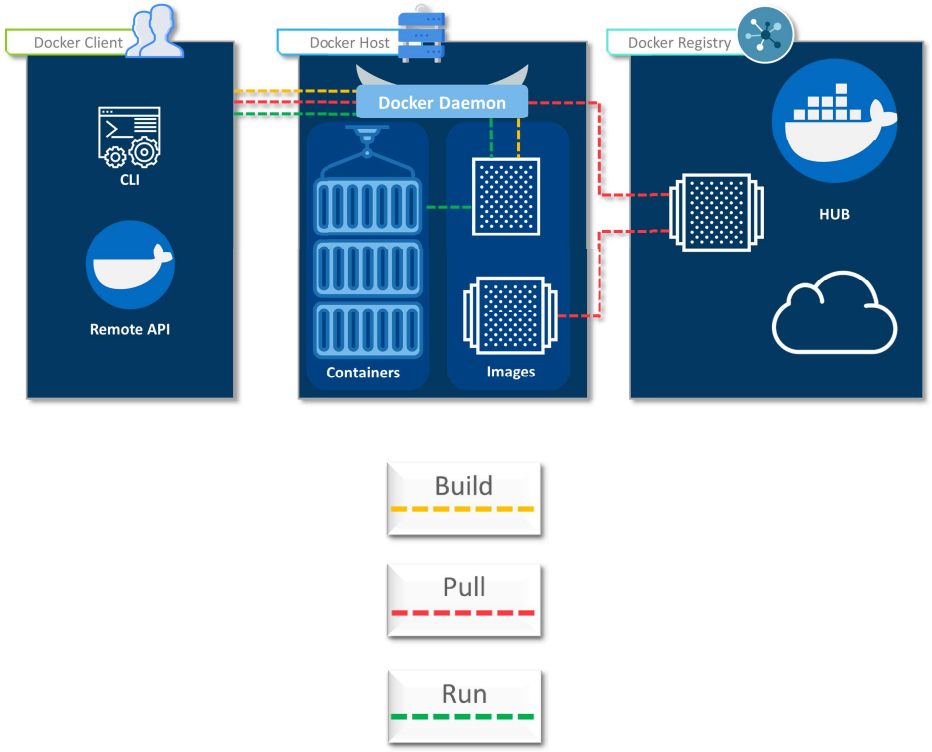
- Mesh allows service discovery through the same port on all the nodes in swarm
- It is possible to reach the node even if the service is not deployed on it

Docker Architecture



Docker Architecture (Contd.)

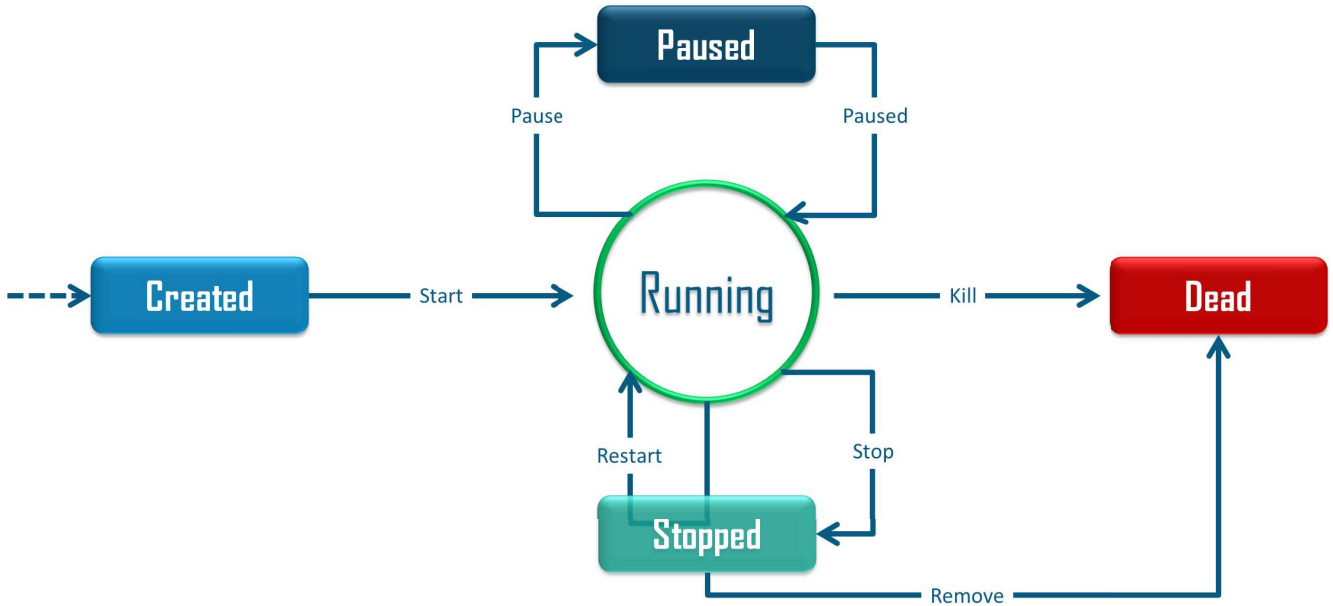
1. Docker works in a client-server architecture
2. The client sends the control commands to the Docker Daemon through a REST API
3. Docker Daemon is responsible for building, running and distributing the containers
4. The Client and Daemon can be on the same system or the client can connect to a remote Daemon



Container Lifecycle

A newly created container can be in one of six states:

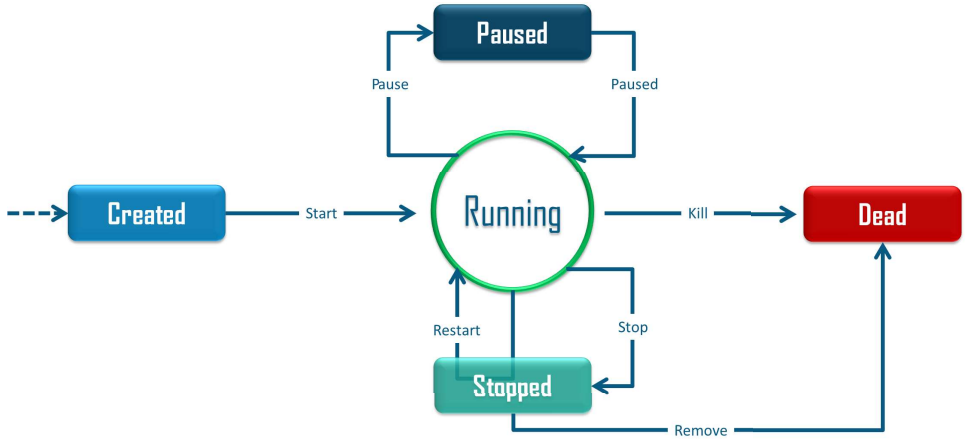
- Created
- Running
- Paused
- Stopped
- Restarted
- Dead



edureka!

Container Lifecycle

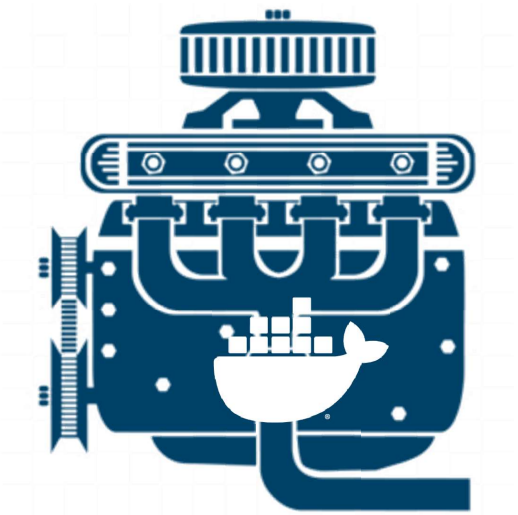
State	Description
Created	A container that has been created but not yet started
Running	A container in its normal working state
Paused	Container whose processes have been paused at the moment
Stopped	A container that is no longer working. Also known as Exited
Restarting	A previously stopped container which is now being restarted
Dead	A container which is no longer in use and is discarded



edureka!

The Docker Engine

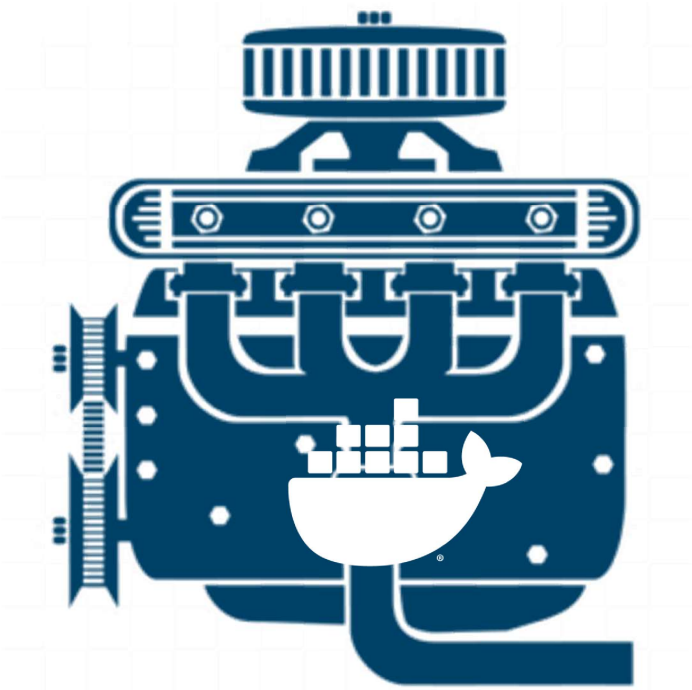
The Docker Engine is responsible for all the building, scheduling, and management of containers



The Docker Engine: Components

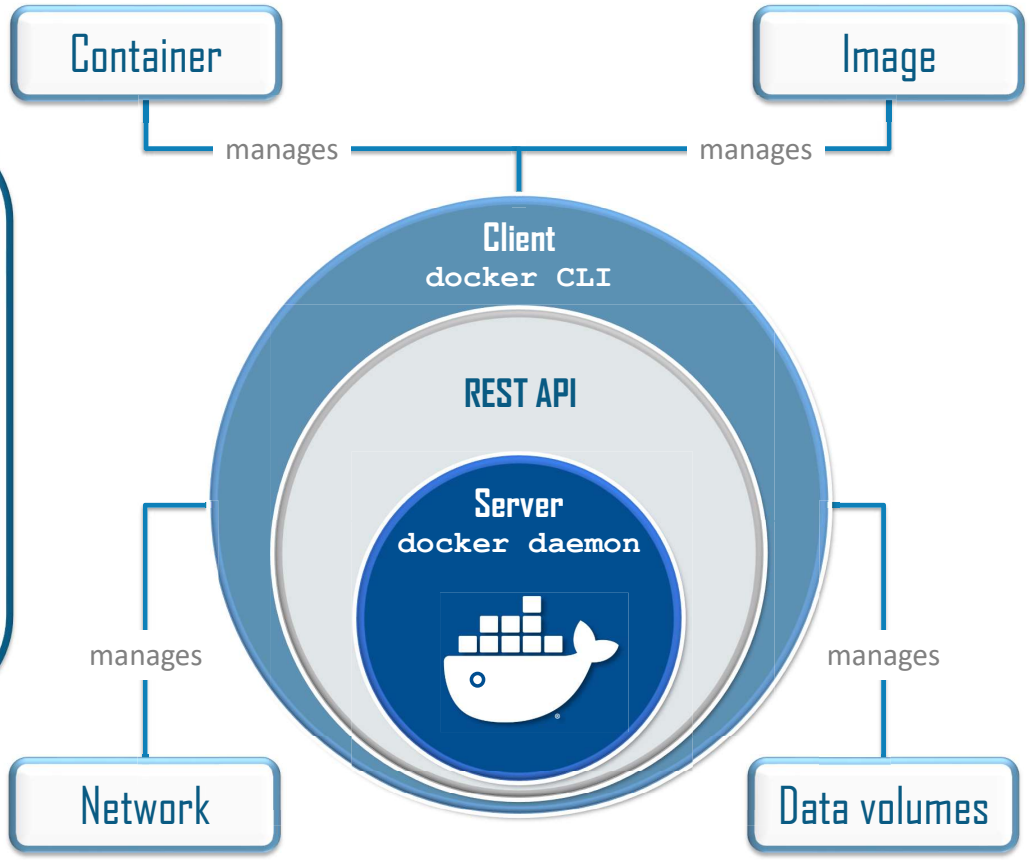
The Docker Engine works in a client-server architecture utilizing the following major components:

- **Server:** A long-running program which is known as a Daemon process (dockerd)
- **REST API:** An API which defines the interfaces the programs can use to communicate with the Daemon
- **CLI:** A Command Line Interface to pass instructions to the Daemon



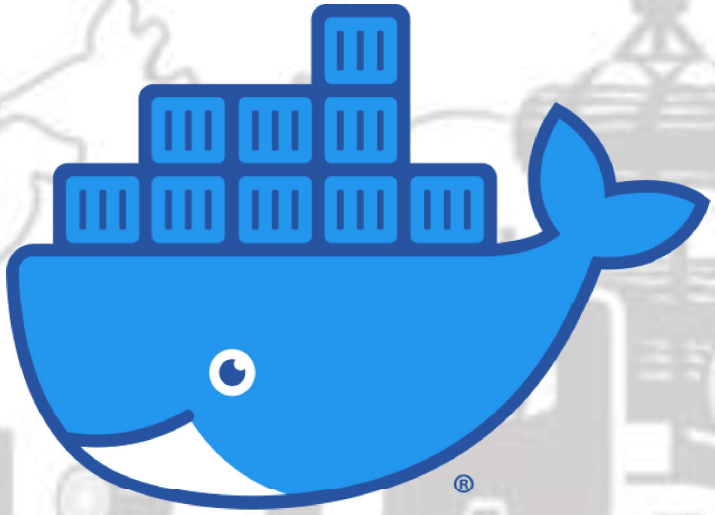
The Docker Engine: Working

- The CLI uses the REST API to control and interact with the Docker Daemon
- CLI does this through either using scripts or direct commands
- Daemon is responsible for creating and managing docker objects such as container, images, network, and volumes



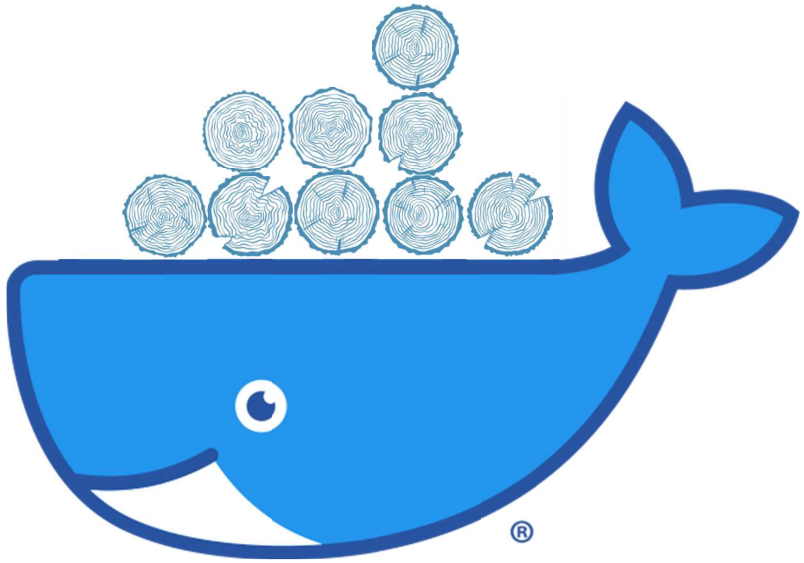
Docker Logging Containers

Docker has a built-in logging mechanism to extract information from containers and services



Docker Logging Drivers

- Each docker daemon has a default logging driver which is common for all the containers
- By default, docker uses the json-file logging driver
- External logging plugins can also be configured with the docker daemon



Logging Drivers: Journald

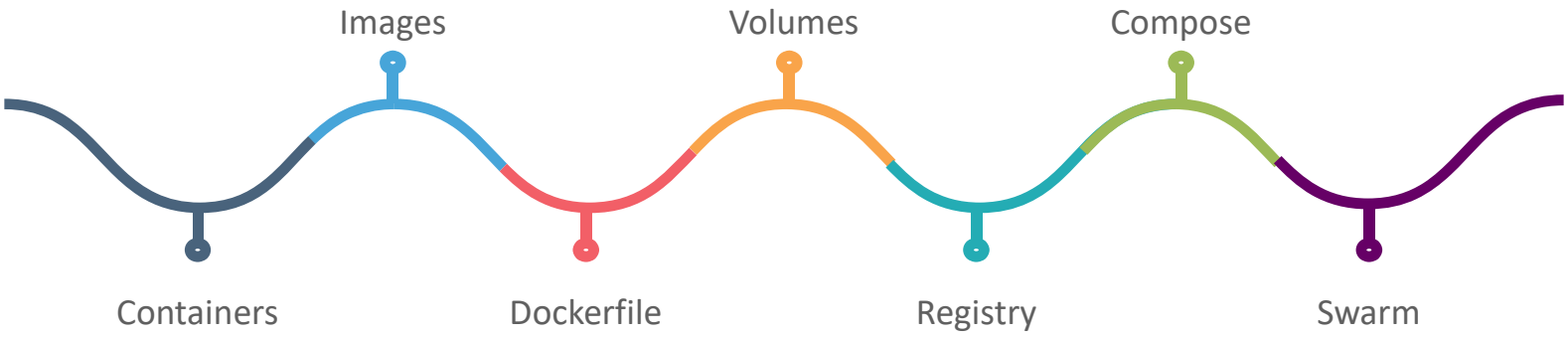
- Journald logging driver uses the system-journald service to collect and store logging data
- It sends the container logs system journal
- The log entries can be captured using `journalctl` command

Logging Drivers: Fluentd

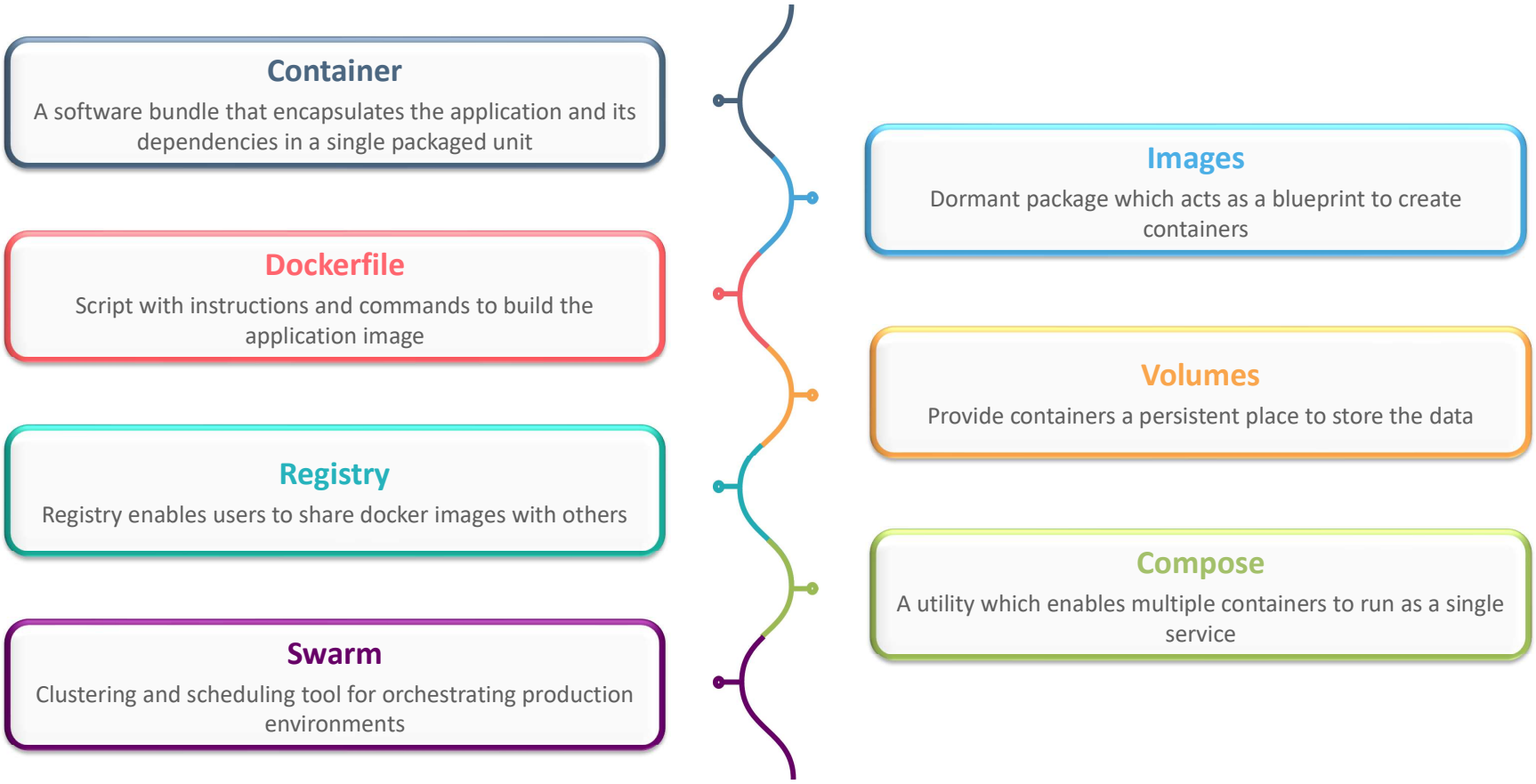
- `fluentd` logging driver uses the fluentd daemon running on the host to collect the log data
- It sends the structured log data to the fluentd collector

Introduction to Docker

Docker is one of the most popular Container engines today because of the following:



Docker Terminology



Why we need Port Binding?

- By default docker containers can connect to the public internet without requiring any configurations
- But the public internet does not know how to connect with the containerized service

What is Port Binding?

The process of exposing the required Docker container port and binding it to a port on the system is known as Port Binding. It is also known as port forwarding or port mapping

Port Binding

- The container port can be exposed using the `-p` flag while starting the container
- This exposes the port on the default System IP i.e. `0.0.0.0`

```
docker run -p <system_port>:<container_port> -d <containerName> <imageName>
```

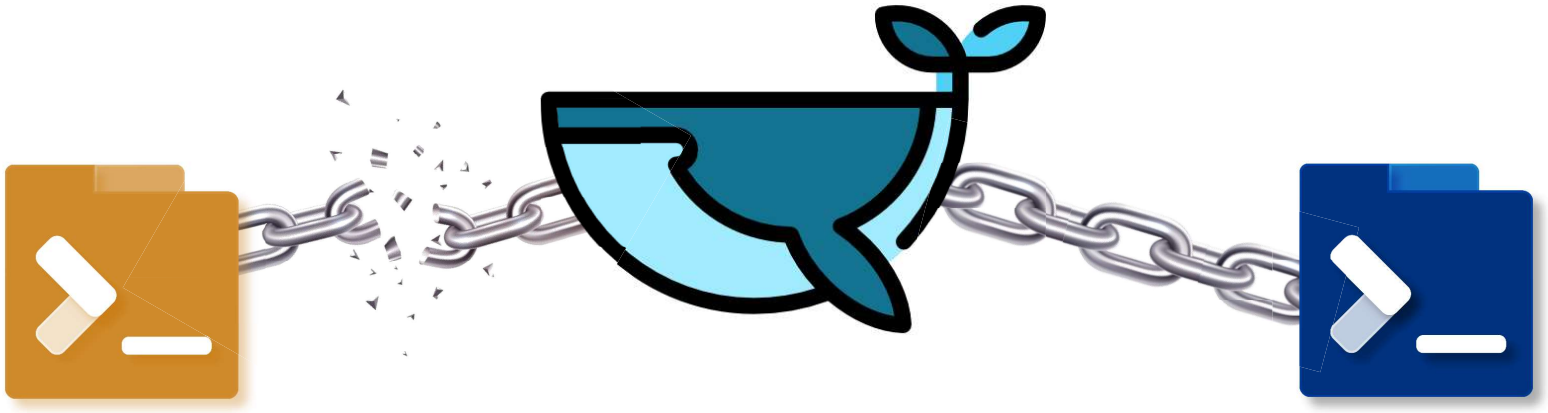
- In order to expose the port on a specific IP, use the following syntax

```
docker run -p <IP>:<system_port>:<container_port> -d <containerName> <imageName>
```

Docker Container Running Mode

Docker provides two modes to run the containers:

- Detached: The container runs in the background
- Foreground or attached: The container attaches itself to the terminal



Detached vs Foreground

Detached

In the detached mode, the container process executes in the background

The container exits when the root process starting the container ends

The '`-d`' flag is used to start the container in detached mode

Example:

```
docker run -d -p 80:80  imageName
```

Foreground

In the attached mode, the container application attaches itself to the console

The process can be connected to any of STDIN, STDOUT, and/or STDERR streams

All containers execute in foreground mode if the '`-d`' flag is not specified

Example:

```
docker run --rm -p 80:80 imageName
```

Detached Mode

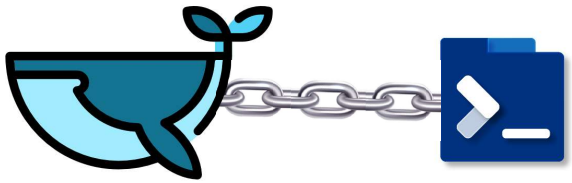
- Using the `--rm` flag with the detached container exits it when the daemon stops
- Input/output actions on detached containers can be done by connecting through network
- A detached container can be reattached to the terminal by using the `attach` command

edureka!

Foreground Mode

- An attached container is usually started in TTY (TeleType) mode, which is done using the `-t` flag
- By default, docker attaches the container to STDOUT and STDERR streams
- The `-t` flag must not be used when providing input through a pipe '|'

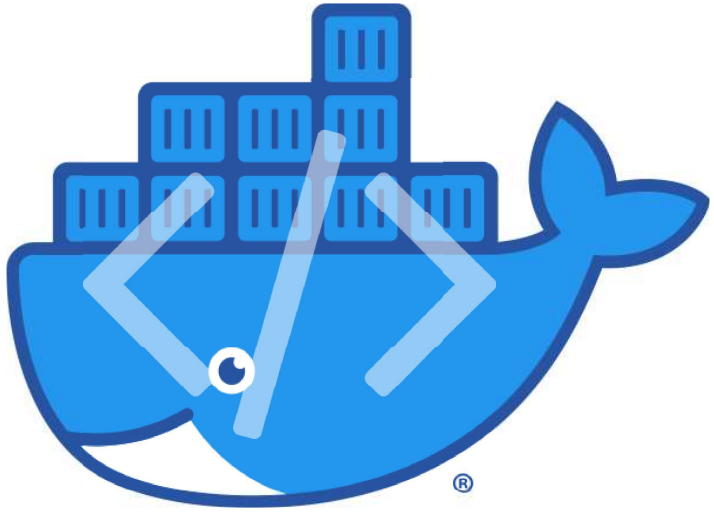
```
Configurable flags
-a  : attach to a 'STDIN', 'STDOUT', and/or 'STDERR'
-t  : allocates a terminal to the container
-i  : keeps the STDIN open, even if not attached
```



edureka!

Docker Command Line Interface (CLI)

Docker offers a Command Line Interface (CLI) to manage and interact with containers. The CLI can also be used to manage remote server operations and the Docker Hub repository operations



Docker CLI Configuration

- Docker commands can be prefaced to run with or without `sudo` permissions
- The command line config files are stored in `.docker` directory inside `$HOME` directory
- The default docker command behavior can be altered using the `config.json` file