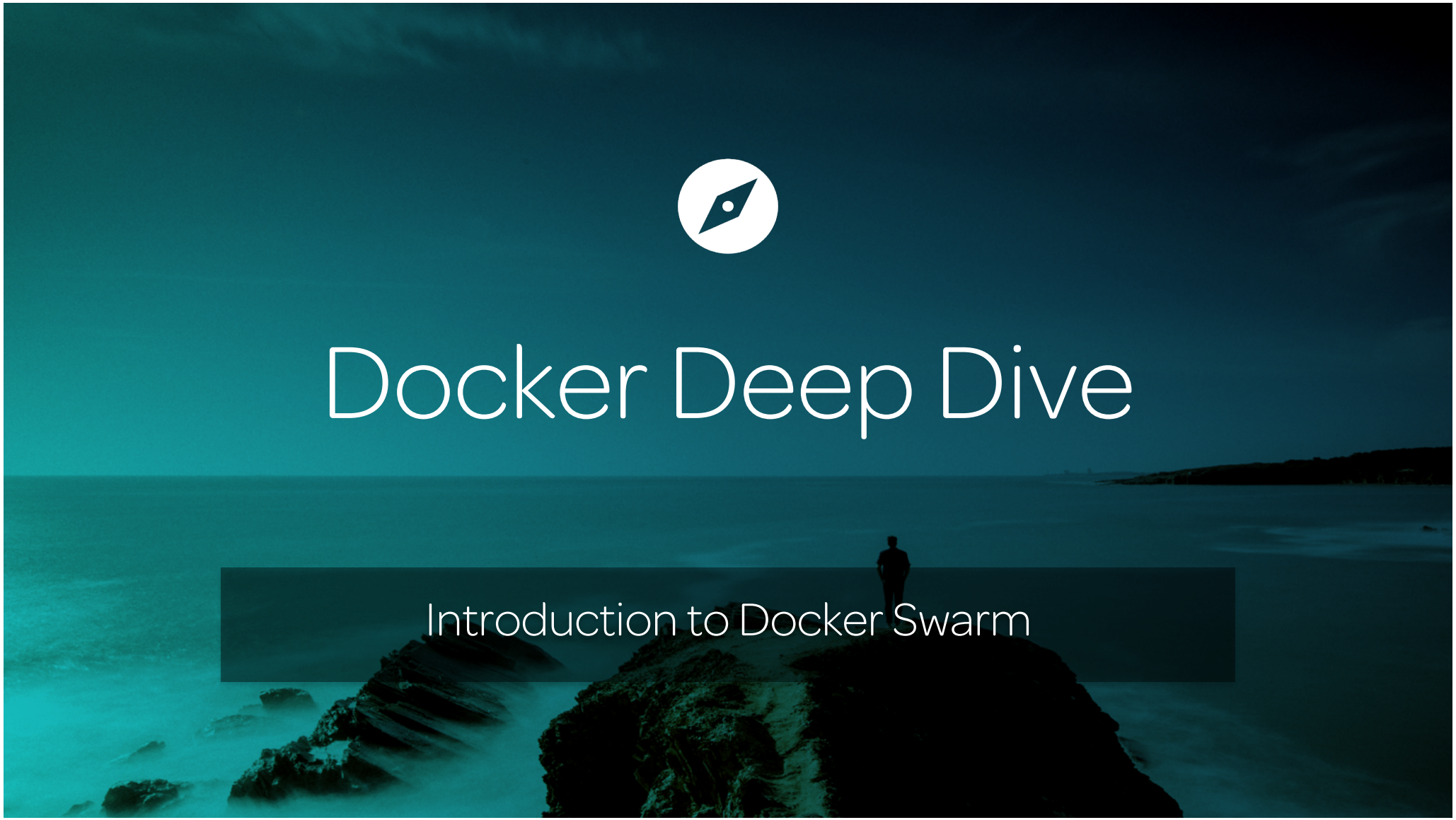

Docker Deep Dive
Introduction to Docker Swarm

Swarm 101

Swarm has two major components :

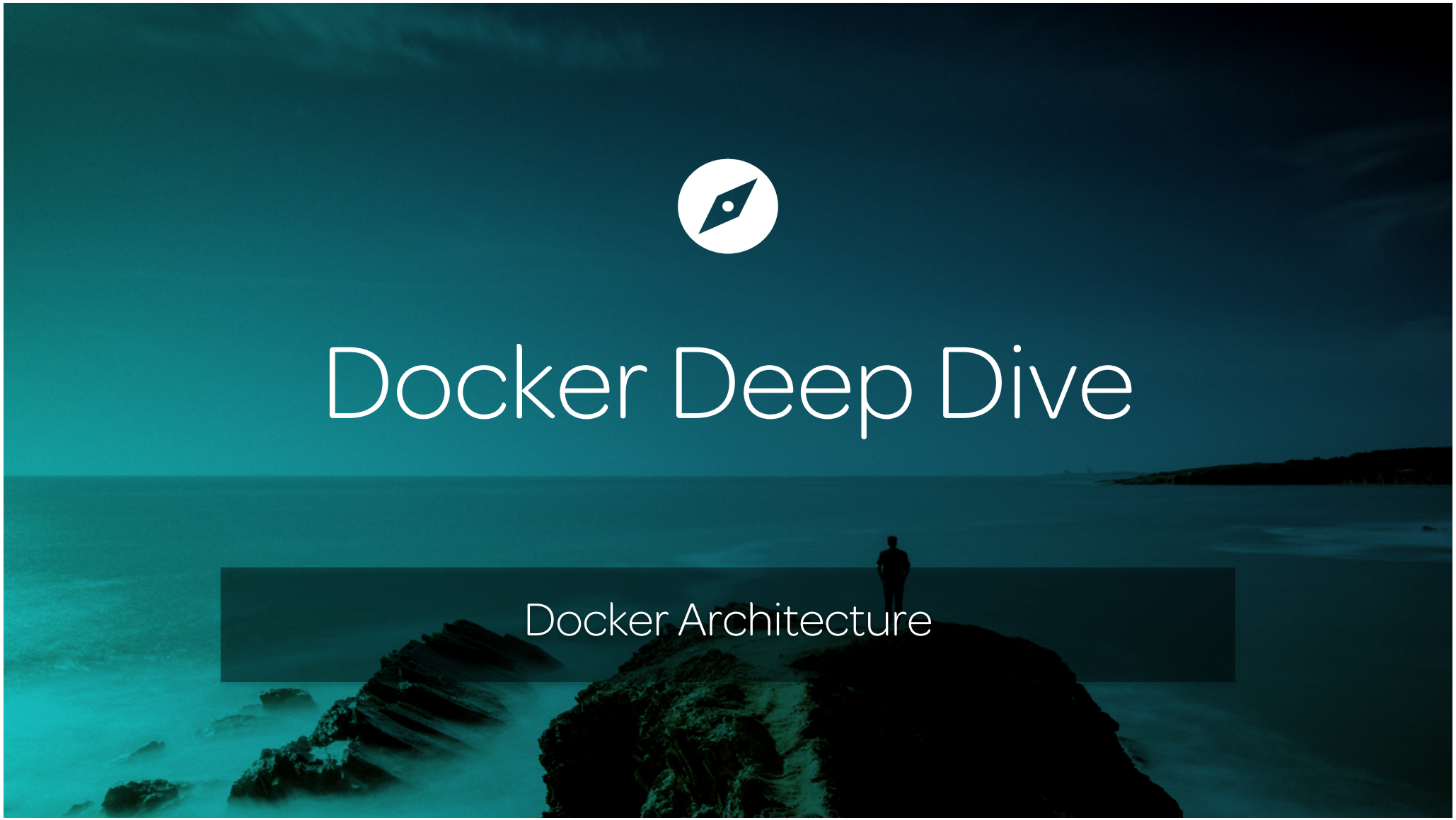
- An enterprise-grade secure cluster:
  - Manage one or more Docker nodes as a cluster
  - Encrypted distributed cluster store
  - Encrypted networks
  - Secure join tokens
- An orchestration engine for creating microservices:
  - API for deploying and managing microservices
  - Define apps in a declarative manifest files
  - Perform rolling updates, rollbacks, and scale apps
- Swarm was initially separate product layered on Docker
- Since Docker 1.12 it became a part of the engine

The Cluster

- A swarm consists of one or more Docker nodes
- Nodes are either a manager or a worker
- Managers:
  - Manage the state of the cluster
  - Dispatches tasks to workers
- Workers:
  - Accepts and execute tasks
- State is held in etcd
- Swarm uses Transport Layer Security (TLS)
  - Encrypted communication
  - Authenticate Nodes
  - Authorize roles

Orchestration

- The atomic unit of scheduling is a swarm service
- The service construct adds the following to a container:
  - scaling
  - rolling updates
  - rollback
- A container wrapped in a service is a task or a replica


Docker Deep Dive
Docker Architecture

Architecture Overview

Docker architecture:

- Client-server architecture
- The client talks to the Docker deamon
- The Docker deamon handles:
  - Building
  - Running
  - Distributing
- Both communicate using a REST API:
  - UNIX sockets
  - Network interface

Architecture Overview (cont.)

The Docker daemon (`dockerd`):

- Listens for Docker API requests and manages Docker objects:
  - Images
  - Containers
  - Networks
  - Volumes

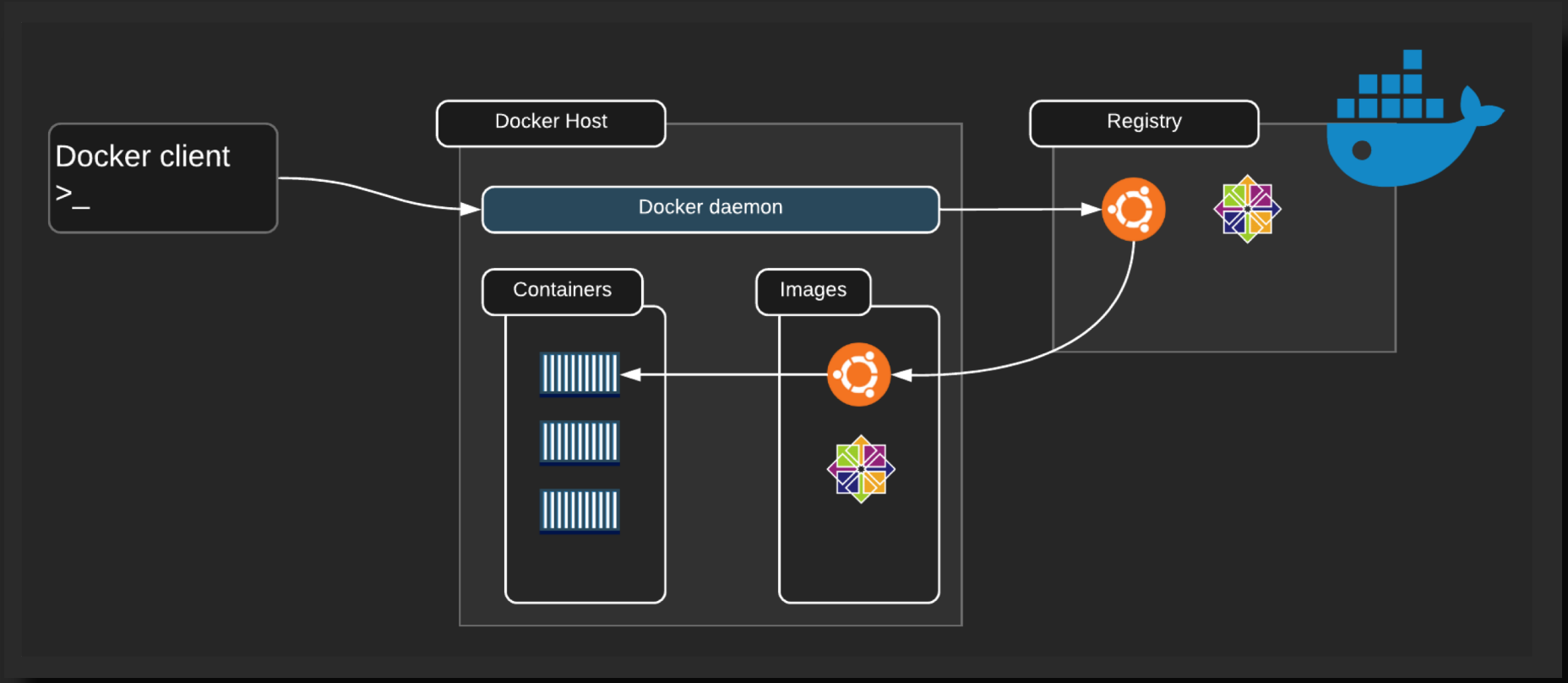
The Docker client (docker):

- Is how users interact with Docker
- The client sends these commands to `dockerd`

Docker registries:

- Stores Docker images
- Public registry such as DockerHub
- Run your own private registry

Architecture Overview (cont.)



Architecture Overview (cont.)

Docker objects:

- Images
  - Read-only template with instructions for creating a Docker container
  - Image is based on another image
  - Create your own images
  - Use a Dockerfile to build images
- Containers
  - Runnable instance of an image
  - Connect a container to networks
  - Attach storage
  - Create a new image based on its current state
  - Isolated from other containers and the host machine

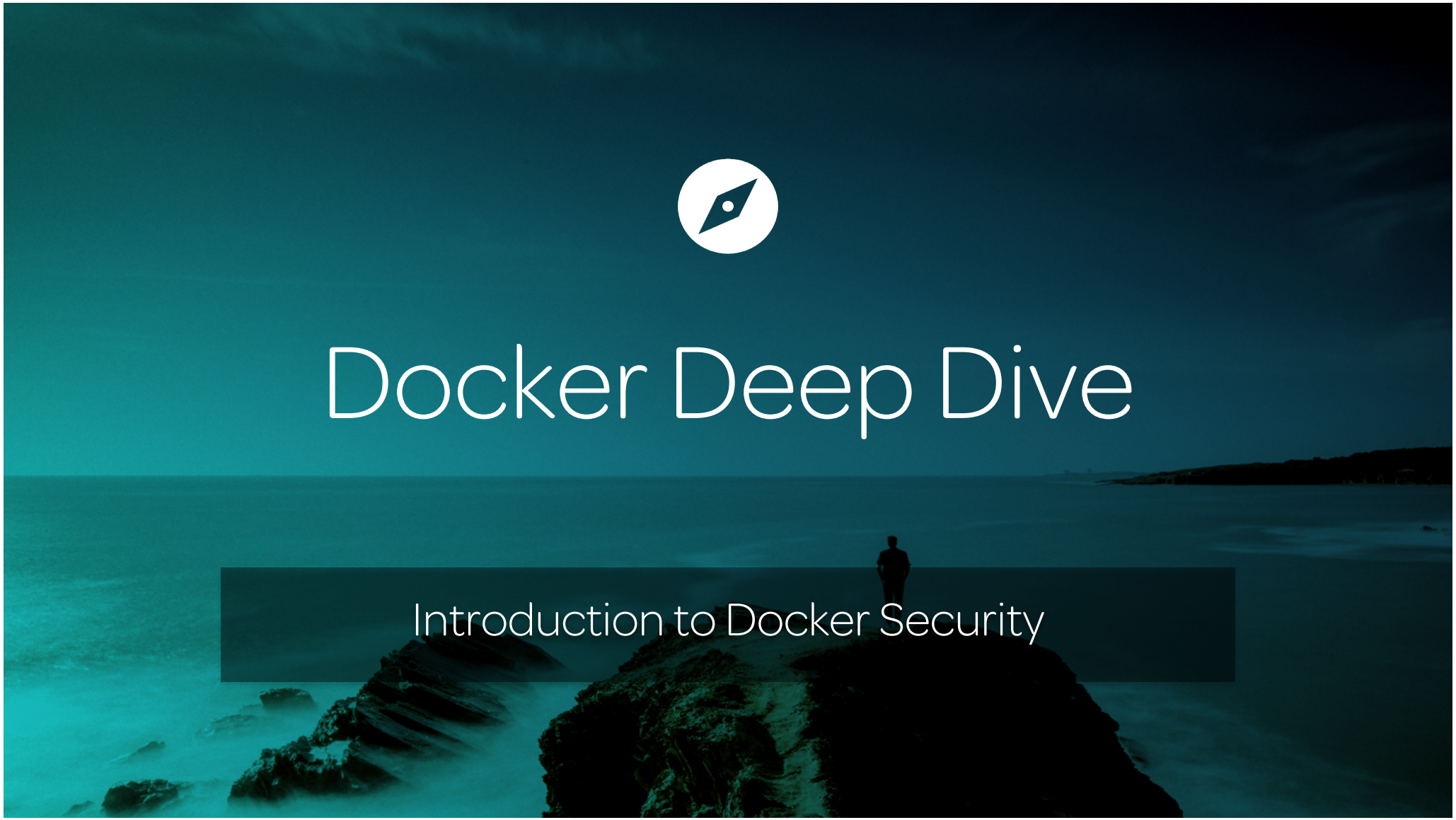
Architecture Overview (cont.)

Docker objects:

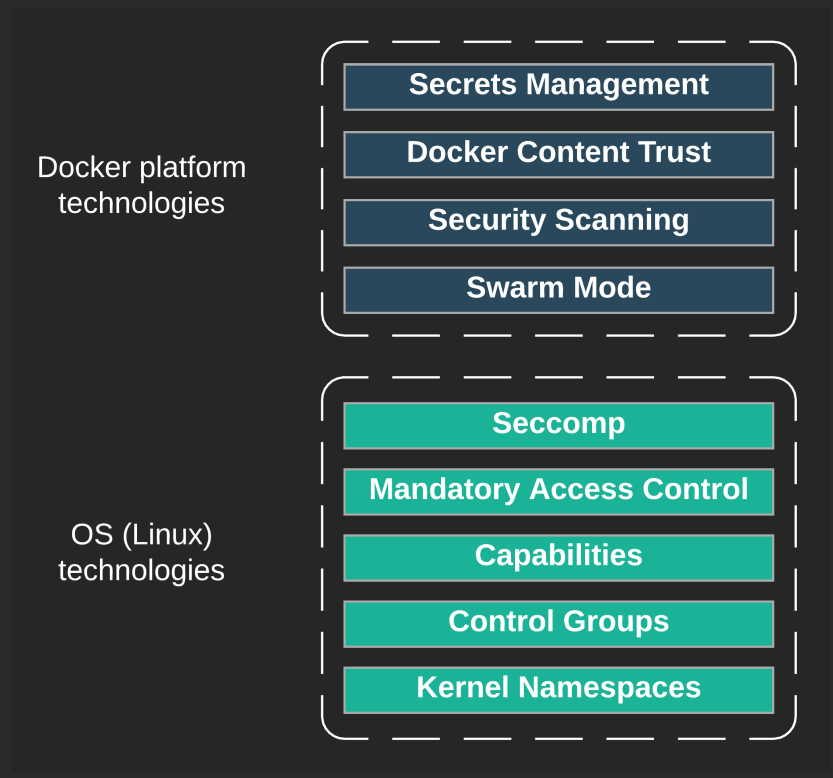
- Services
  - Scale containers across multiple Docker daemons
  - Docker Swarm
  - Define the desired state
  - Service is load-balanced

Docker Swarm:

- Multiple Docker daemon (Master and Workers)
- The daemons all communicate using the Docker API
- Supported in Docker 1.12 and higher


Docker Deep Dive
Introduction to Docker Security

Docker Security 101



Namespaces

- Docker on Linux Namespaces:
  - Process ID (pid)
  - Network (net)
  - Filesystem/mount (mount)
  - Inter-process Communication (ipc)
  - User (user)
  - UTS (uts)

Docker Swarm

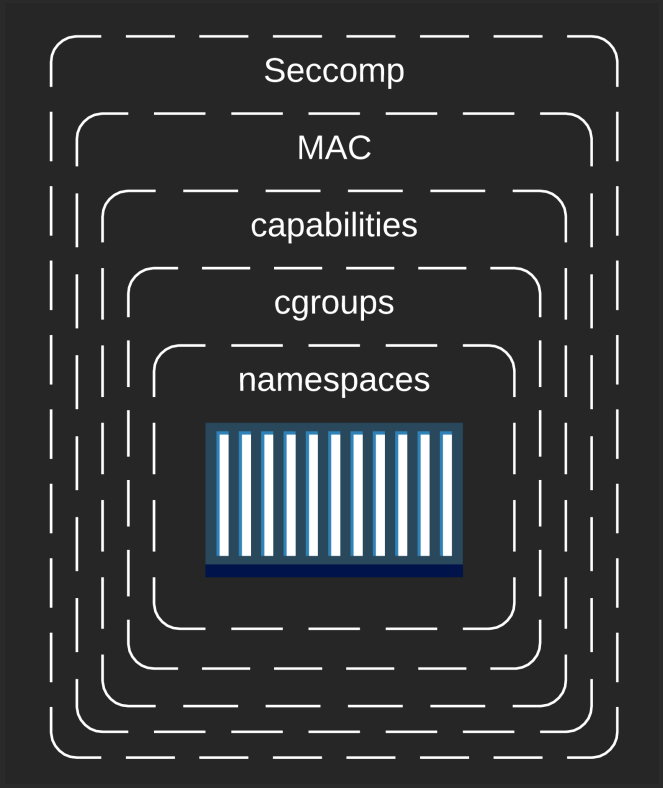
- Cryptographic node Ids
- Mutual authentication via TLS
- Secure join tokens
- CA configuration with automatic certificate rotation
- Encrypted cluster store
- Encrypted networks

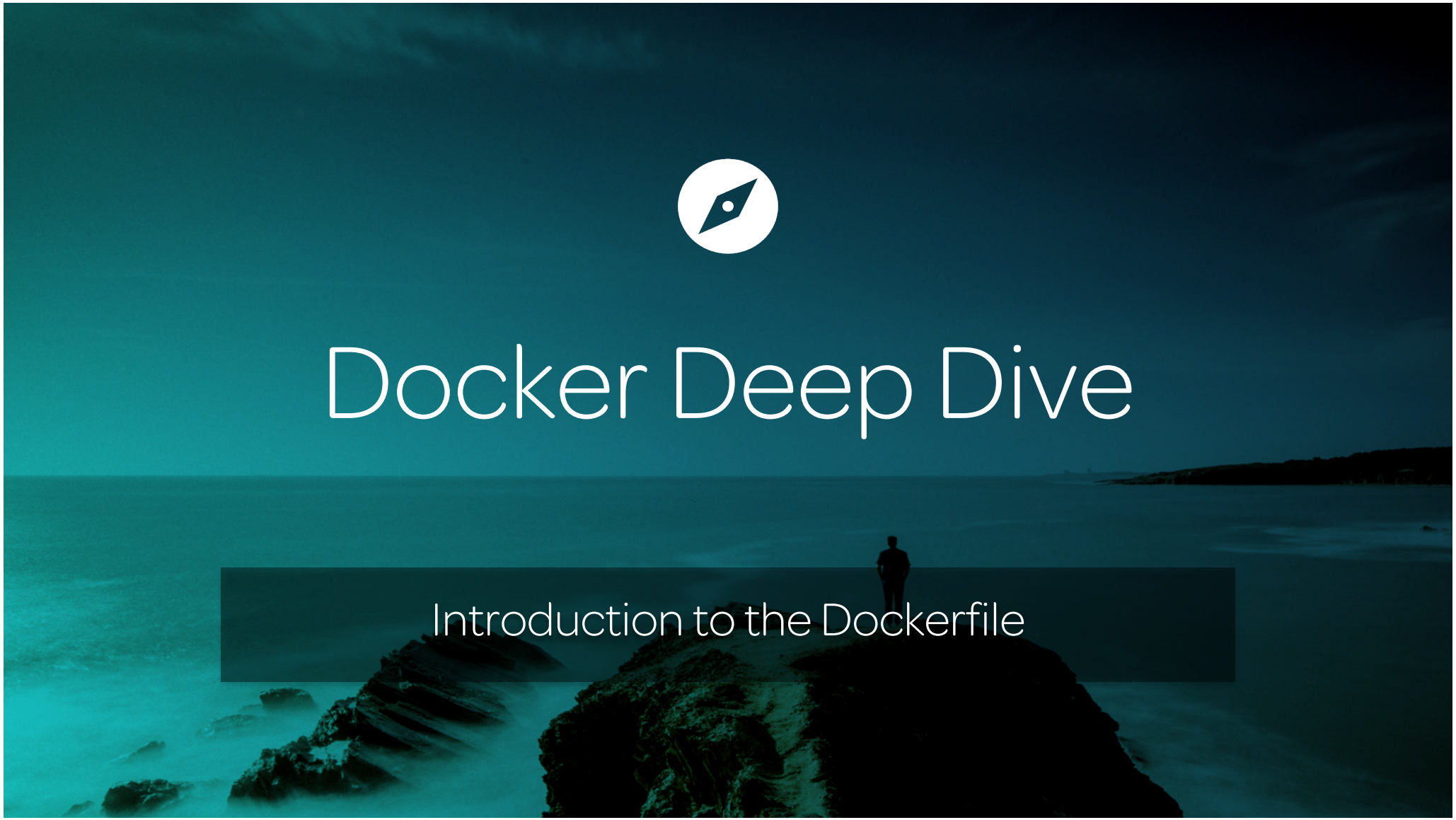
Docker Secrets

Secrets Workflow:

- A secret is created and posted to the Swarm.
- The secret is encrypted and stored.
- A service is created and the secret is attached.
- Secrets are encrypted in-flight.
- The secret is mounted into the container of a service.
- When the task is complete, the in-memory is torn down.

Docker Security 101 (cont.)




Docker Deep Dive
Introduction to the Dockerfile

What is the Dockerfile?

Dockerfiles are instructions on how to build an image.

The file contains all commands used to start a container:

- Docker image consists of read-only layers.
- Each layer represents a Dockerfile instruction.
- Layers are stacked.
- Each layer is a delta of the changes from the previous layer.
- Images are built using the `docker image build` command.

Dockerfile Layers

Dockerfile

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

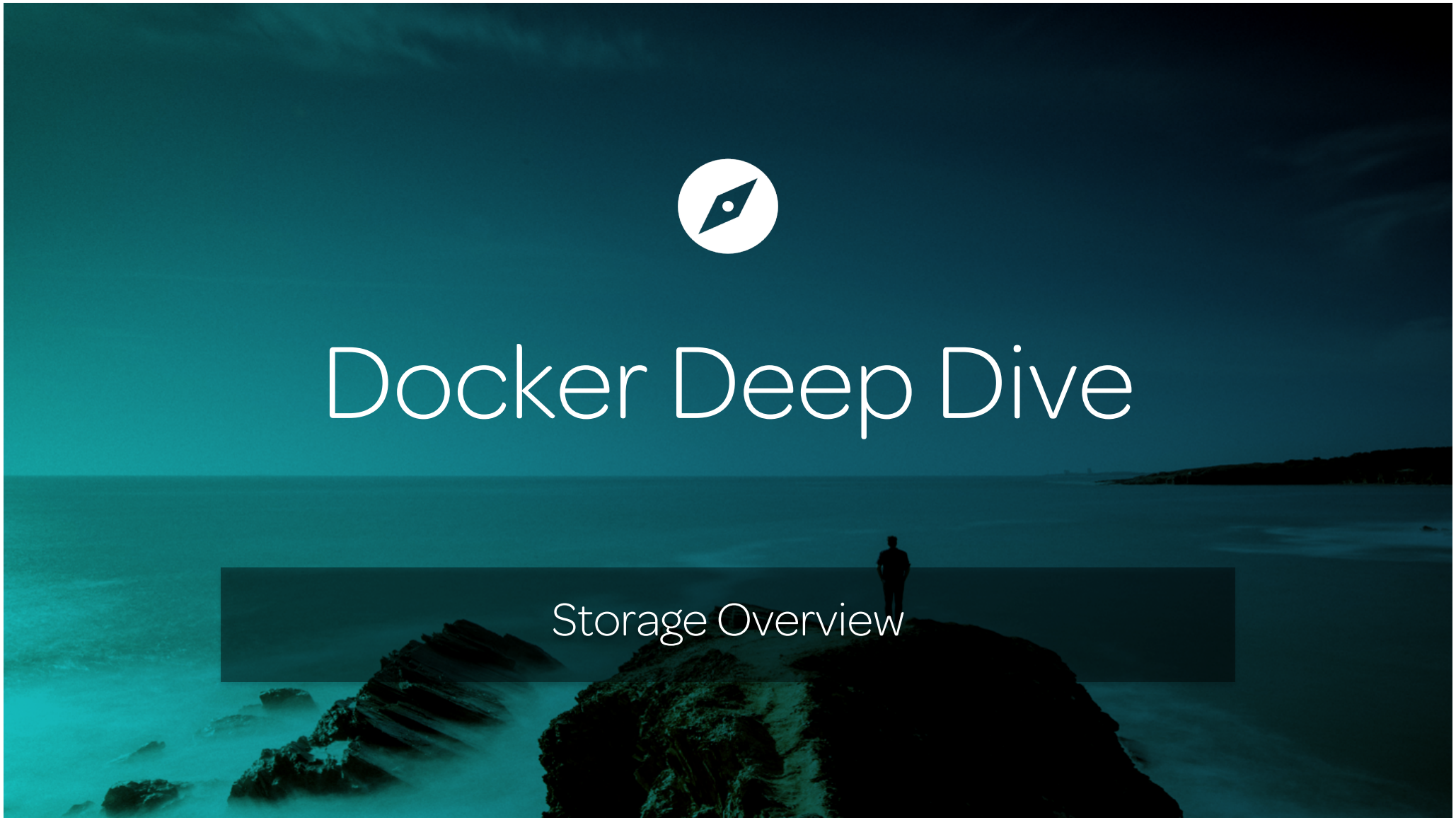
Layers:

- **FROM** creates a layer from the ubuntu:15.04 Docker image.
- **COPY** adds files from your Docker client's current directory.
- **RUN** builds your application with make.
- **CMD** specifies what command to run within the container.

Best Practices

General guidelines:

- Keep containers as ephemeral as possible.
- Follow Principle 6 of the 12 Factor App.
- Avoid including unnecessary file.
- Use **.dockerignore**.
- Use multi-stage builds.
- Don't install unnecessary packages.
- Decouple applications.
- Minimize the number of layers.
- Sort multi-line arguments.
- Leverage the build cache.


Docker Deep Dive
Storage Overview

Docker Storage 101

Categories of data storage:

- Non-persistent
  - Data that is ephemeral
  - Every container has it
  - Tied to the lifecycle of the container
- Persistent
  - Volumes
  - Volumes are decoupled from containers

Non-persistent Data

Non-persistent data:

- By default all container use local storage
- Storage locations:
  - linux: `/var/lib/docker/<STORAGE-DRIVER>/`
  - windows: `c:\ProgramData\Docker\windowsfilter\`
- Storage Drivers:
  - RHEL uses overlay2
  - Ubuntu uses overlay2 or aufs
  - SUSE uses btrfs
  - Windows uses it's own

Persistent Data Using Volumes

Volumes:

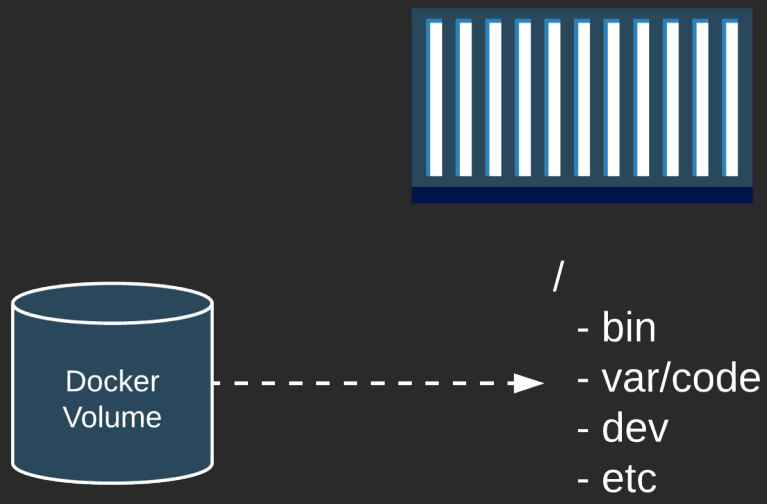
- Use a volume for persistent data
  1. Create the volume.
  2. Create your container.
- Mounted to a directory in the container
- Data is written to the volume
- Deleting a container does not delete the volume
- First-class citizens
- Uses the local driver

Persistent Data Using Volumes (cont.)

Volumes:

- Third party drivers
  - Block storage
  - File storage
  - Object storage
- Storage locations
  - linux: `/var/lib/docker/volumes/`
  - windows: `c:\ProgramData\Docker\volumes`

Persistent Data Using Volumes (cont.)




Docker Deep Dive
Networking Overview

Docker Networking 101

Docker Networking:

- Container Network Model (CNM)
- The libnetwork implements CNM
- Driver extend the model by network topologies

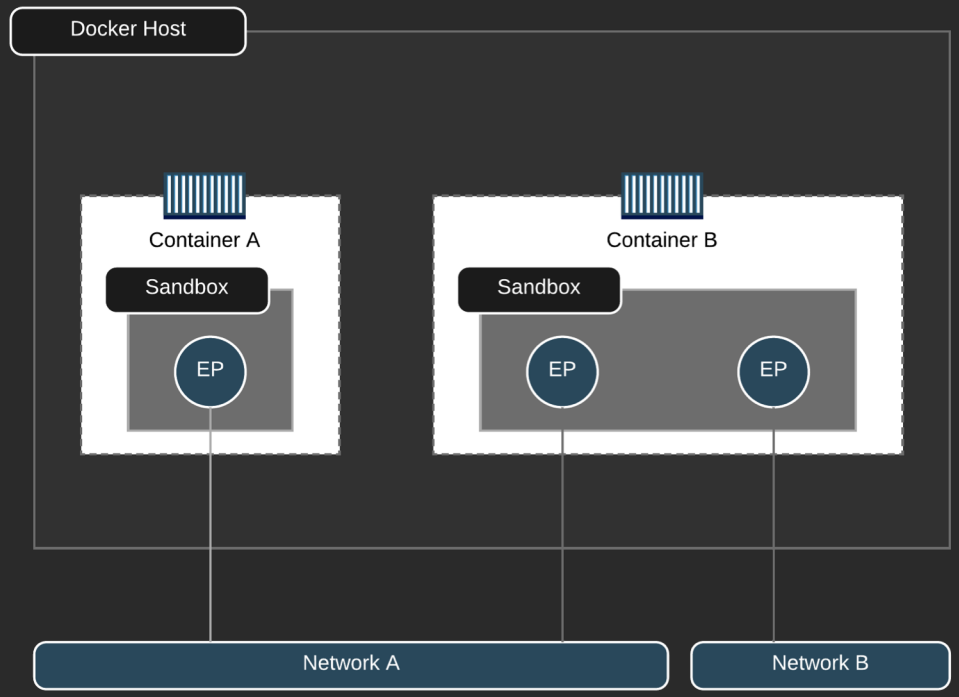
Network Drivers:

- bridge
- host
- overlay
- macvlan
- none
- Network plugins

Container Network Model

Defines three building blocks:

- Sandboxes
- Endpoints
- Networks

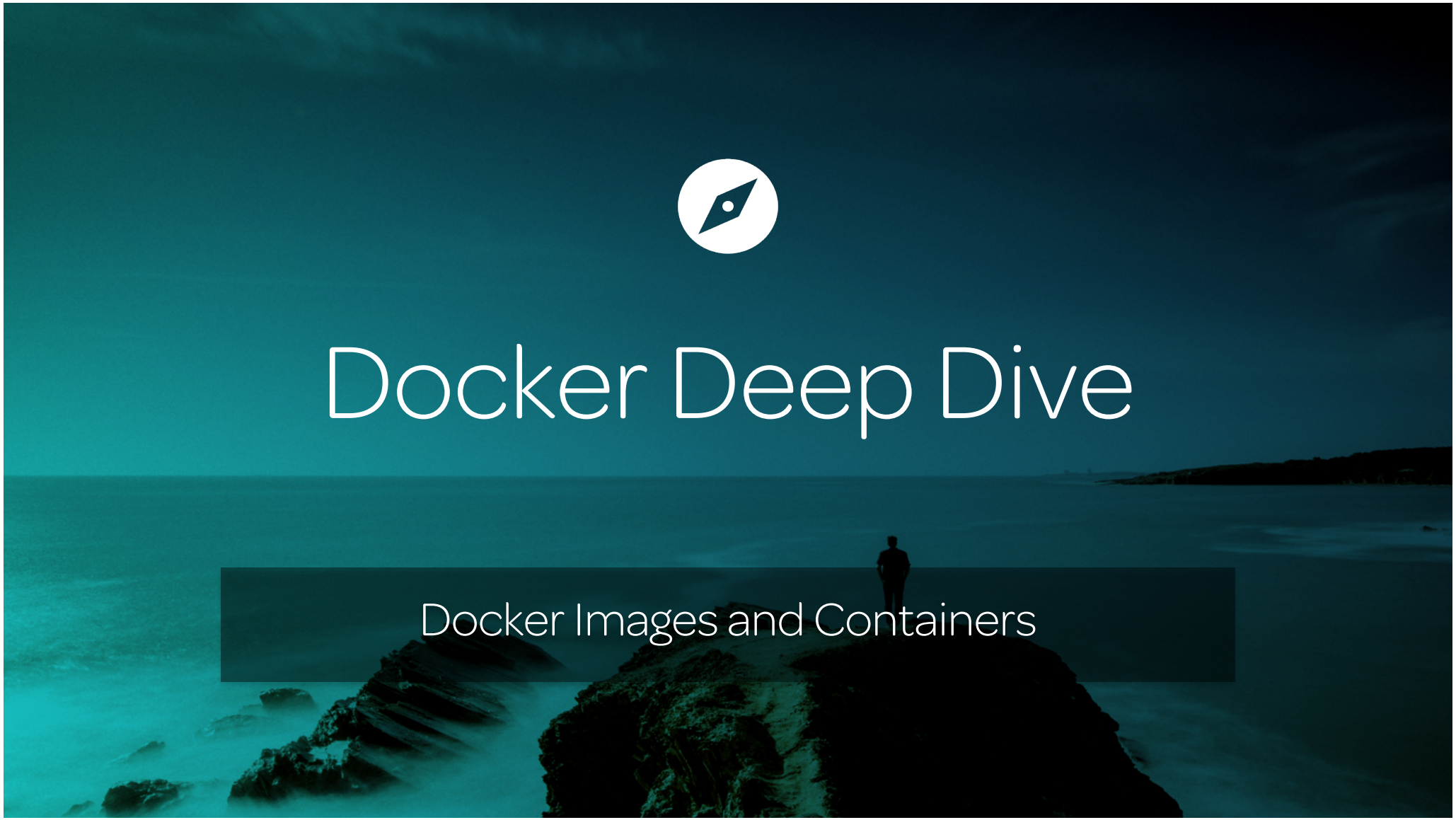



Docker Deep Dive
Docker Hub

What is Docker Hub?

Docker Hub:

- Public Docker registry
- Provided by Docker
- Features:
  - Repositories
  - Teams and Organizations
  - Official Images
  - Publisher Images
  - Builds
  - Webhooks
- https://hub.docker.com/signup


Docker Deep Dive
Docker Images and Containers

What are Docker Images?

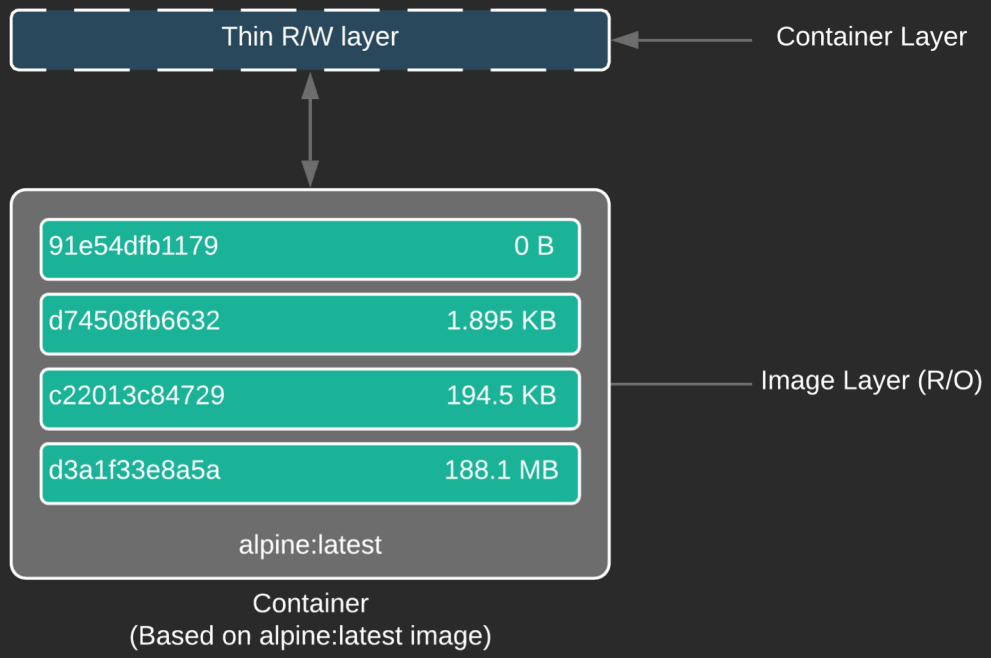
Docker Images:

- Use images to create an instance of a container
- Comprised of multiple layers
- Build time constructs
- Built from the instructions

Docker Images and Layers

- Images are made of multiple layers
- Each layer represents an instruction in the image's Dockerfile
- Each layer, except the very last one, is read-only
- Each layer is only a set of differences from the layer before it
- Containers add a new writable layer on top of the underlying layers
- All changes made to a running container are made to the Container layer

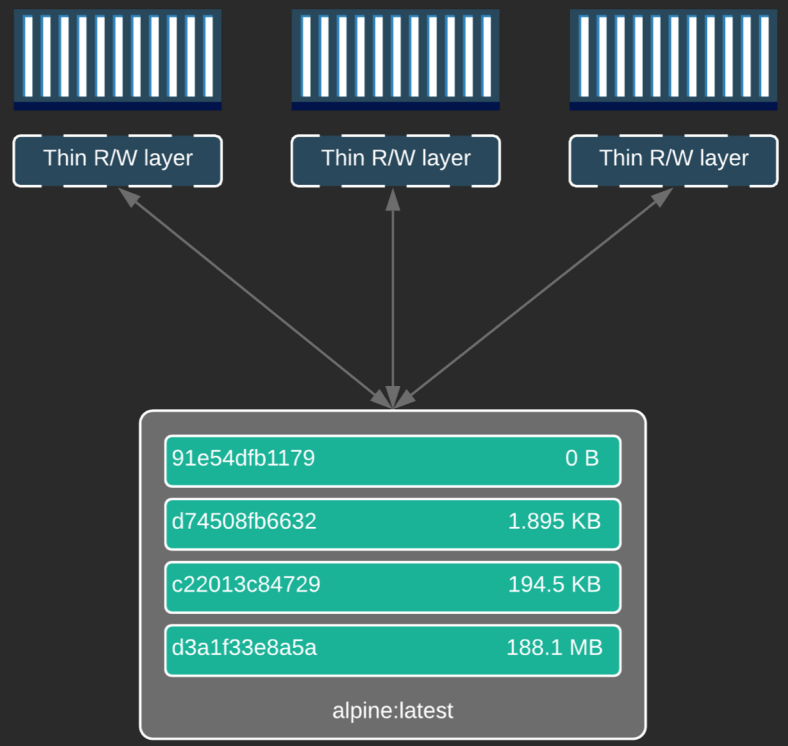
Docker Images and Layers (cont.)



Container and Layers

- Top writable layer
- All changes are stored in the writable layer
- The writable layer is deleted when the container is deleted
- The image remains unchanged

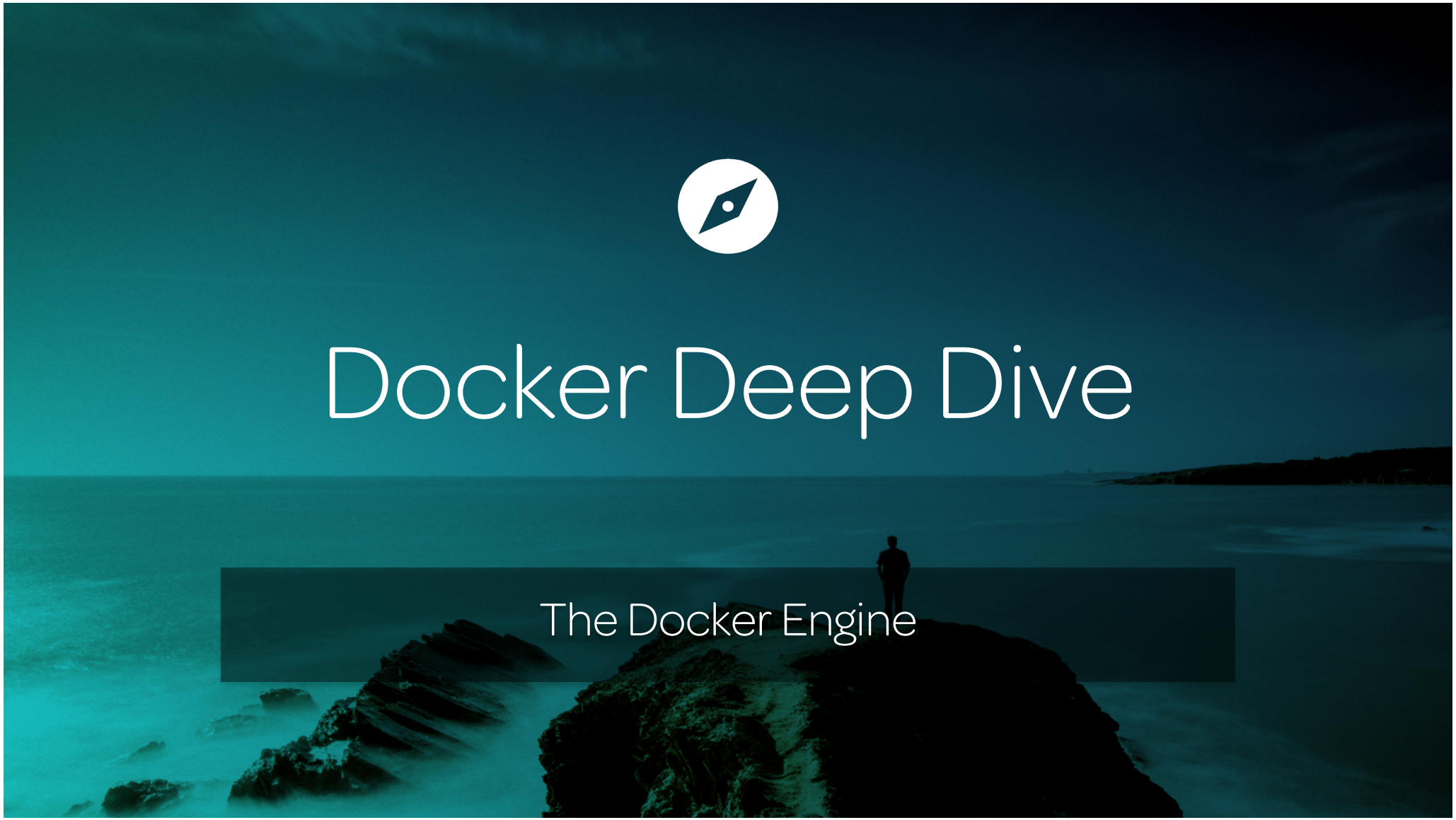
Container and Layers (cont.)



Containers

Container Use Cases:

- Tasks
  - Used to perform one-off jobs
- Long-running processes (LRP)
  - Run until terminated

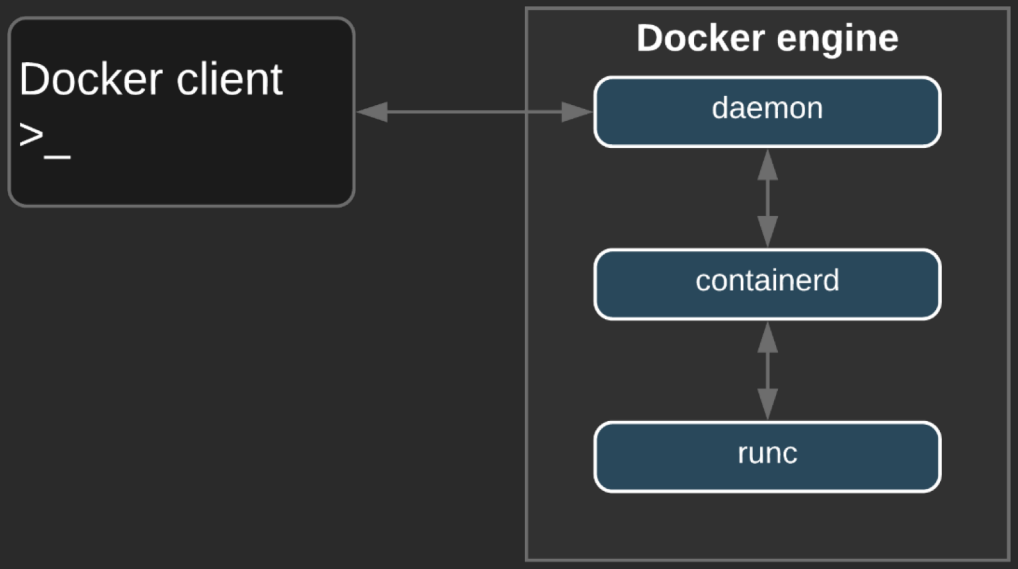

Docker Deep Dive
The Docker Engine

Under The Hood

Docker engine :

- Modular in design:
  - Batteries included but replaceable
- Based on an open-standards outline by the Open Container Initiative
- The major components:
  - Docker client
  - Docker daemon
  - `containerd`
  - `runc`
- The components work together to create and run containers

Under The Hood (cont.)

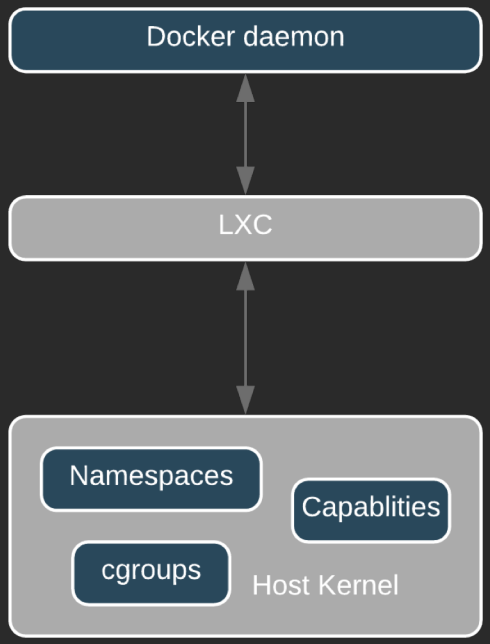


A Brief History of the Docker Engine

The first release of Docker

- The Docker daemon
  - Docker client
  - Docker daemon
  - Monolithic binary
  - Docker client
  - Docker API
  - Container runtime
  - Image builds
  - Much more...
- LXC
  - Namespaces
  - Control groups (cgroups)
  - Linux-specific

A Brief History of the Docker Engine (cont.)



Refactoring of the Docker Engine

LXC was later replaced with libcontainer:

- Docker 0.9
- Platform agnostic

Issues with the monolithic Docker daemon:

- Harder to innovate
- Slow
- Not what the ecosystem wanted

Docker became more modular:

- Smaller more specialized tools
- Pluggable architecture

Refactoring of the Docker Engine (cont.)

Open Container Initiative:

- Image spec
- Container runtime spec
- Version 1.0 released in 2017
- Docker, Inc. heavily contributed
- Docker 1.11 (2016) used the specification as much as possible

Refactoring of the Docker Engine (cont.)

`runc`

- Implementation of the OCI container-runtime-spec
- Lightweight CLI wrapper for libcontainer
- Create containers

`containerd`

- Manage container lifecycle:
  - Start
  - Stop
  - Pause
  - Delete
- Image management
- Part of the 1.11 release

Refactoring of the Docker Engine (cont.)

`shim`

- Implementation of daemonless Containers
- `containerd` forks an instance of `runc` for each new container
- `runc` process exits after the container is created
- `shim` process becomes the container parent
- Responsible for:
  - STDIN and STDOUT
  - Reporting exit status to the Docker daemon

Running Containers

```
docker container run -it --name <NAME> <IMAGE>:<TAG>
```

Creating a container:

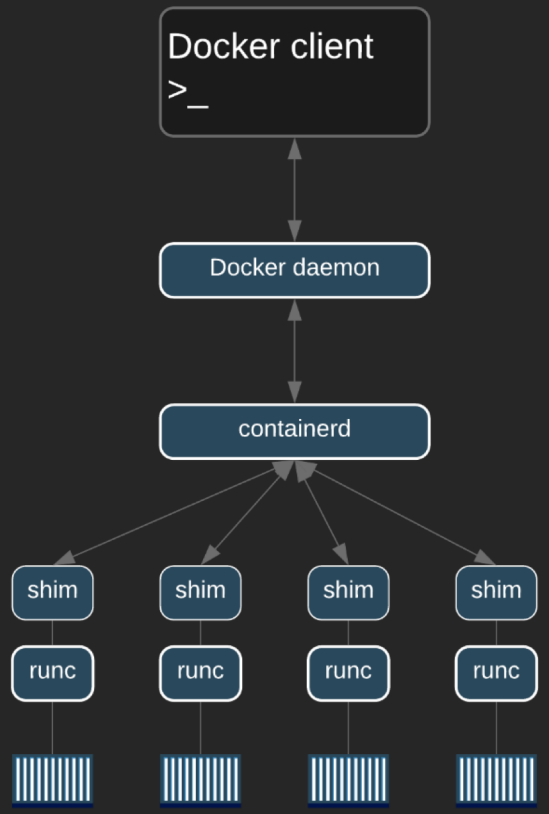
- Use the CLI to execute a command.
- Docker client uses the appropriate API payload.
- The command POSTs to the correct API endpoint.
- The Docker daemon receives instructions.
- The Docker daemon calls `containerd` to start a new container.
- The Docker daemon uses `gRPC`, a CRUD style API.
- `containerd` creates an OCI bundle from the Docker image.
- Tells `runc` to create a container using the OCI bundle
- `runc` interfaces with the OS kernal to get the constructs needed to create a container

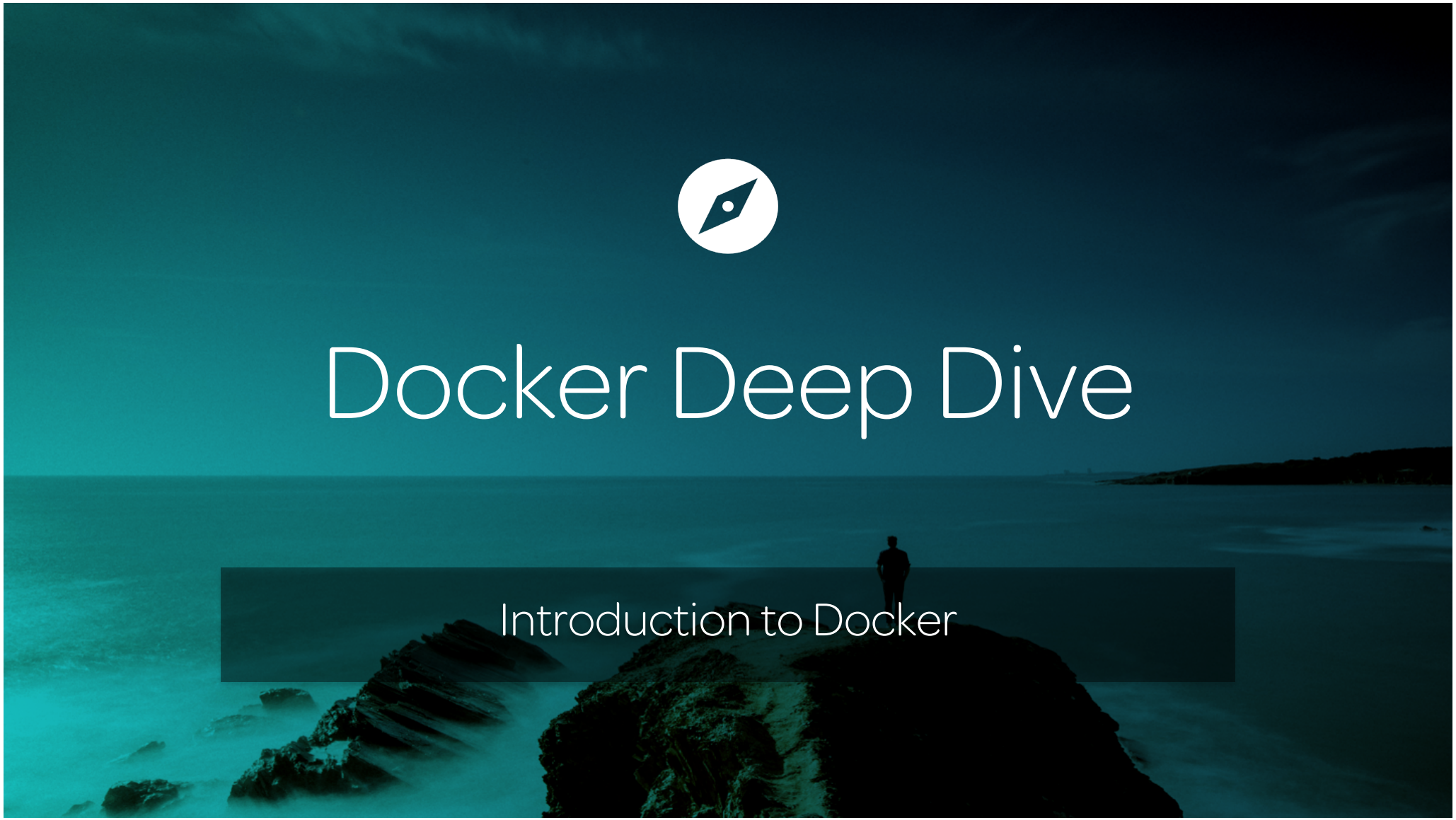
Running Containers (cont.)

Creating a container:

- This includes namespaces, `cgroups`, etc.
- The container process is started as a child process
- Once the container starts `runc` will exit
- Which completes the process and the container is now running

Process Summary




Docker Deep Dive
Introduction to Docker

What is Docker?

Docker :

- The company Docker, Inc.
- Docker the container runtime and orchestration engine
- Docker the open source project (Moby)

The Company

Docker, Inc. :

- Based out of San Francisco
- Found by Solomon Hykes
- Started as a PaaS provider called dotCloud
- dotCloud leveraged Linux containers
- Their internal tool used to manage containers was nick-named Docker
- In 2013, dotCloud was rebranded as Docker

The Runtime and Orchestration Engine

The Docker runtime and orchestration engine :

- Most people are referring to the Docker Engine
- Two main editions:
  - Enterprise Edition (EE)
  - Community Edition (CE)
- Both are released quarterly
  - CE is supported for 4 months
  - EE is supported for 12 months

The Open-Source Project

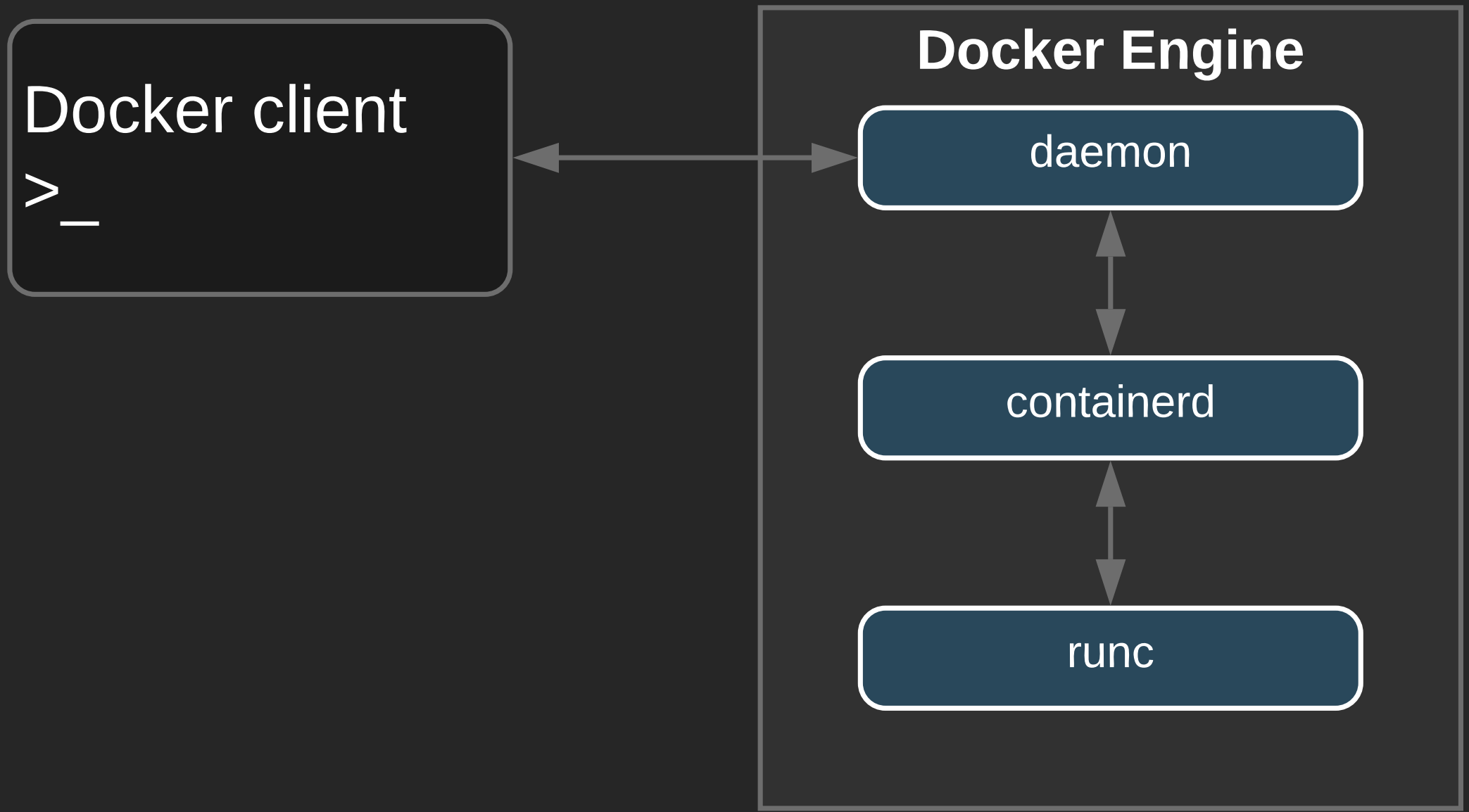
Moby:

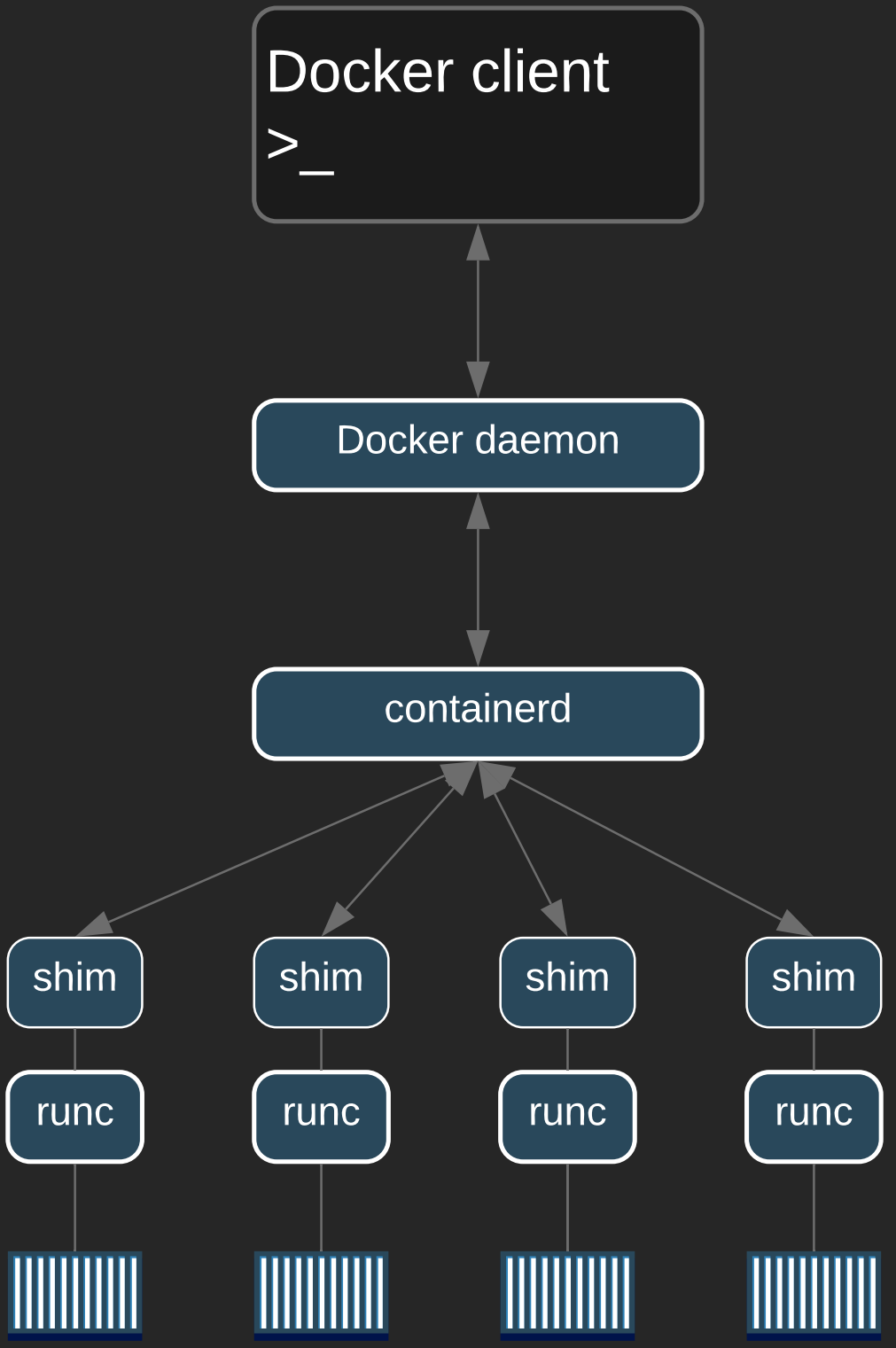
- The upstream project of Docker
- Breaks Docker down into more modular components
- Code is available on GitHub: https://github.com/moby/moby

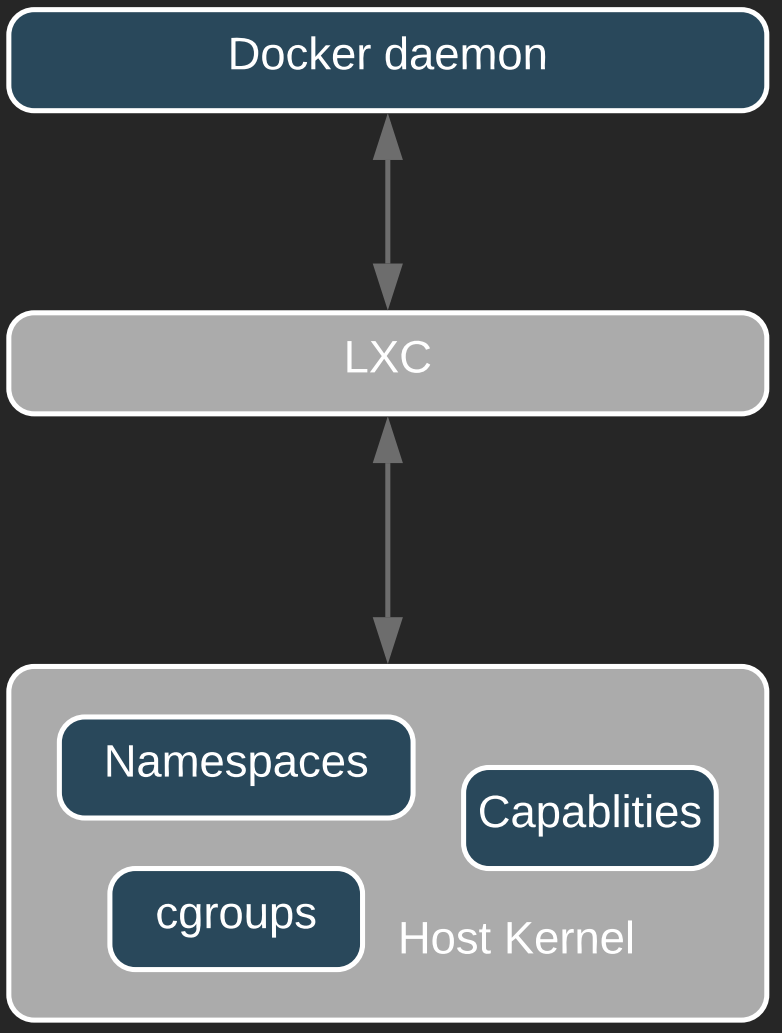
Why Use Docker?

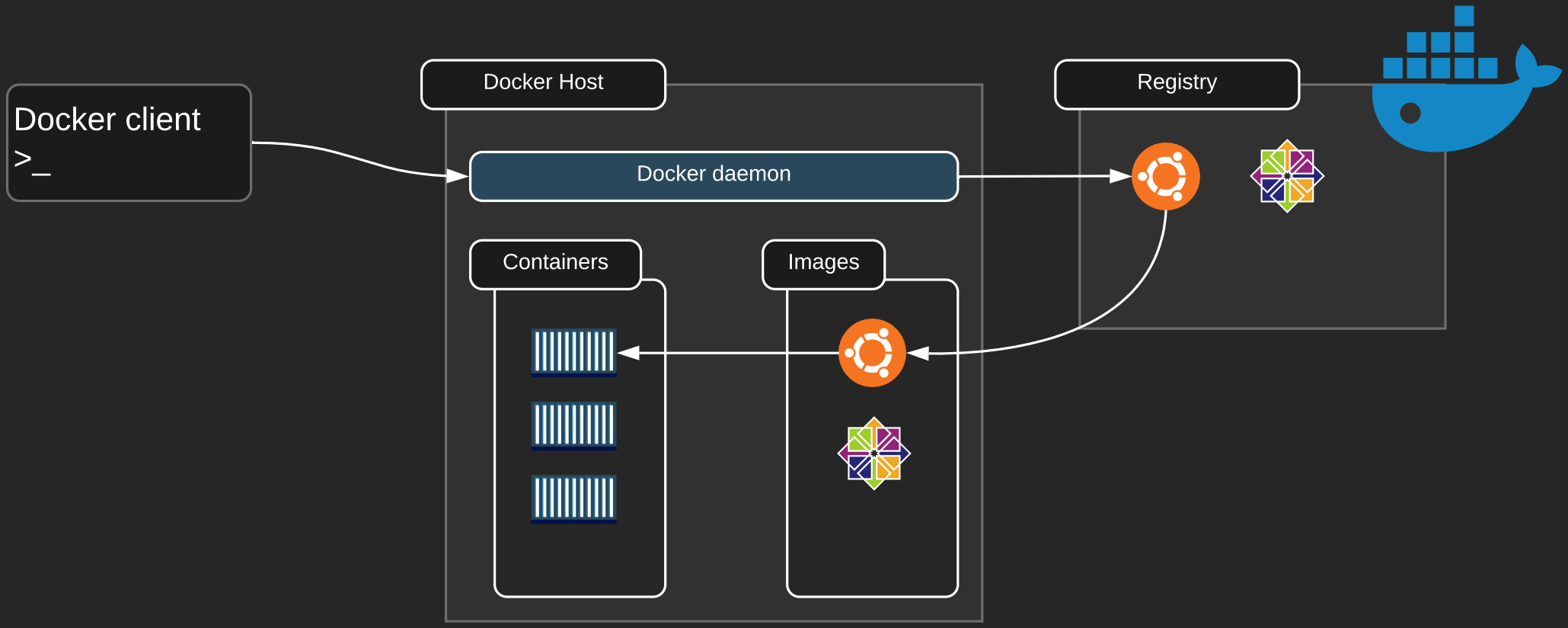
Docker Use Cases:

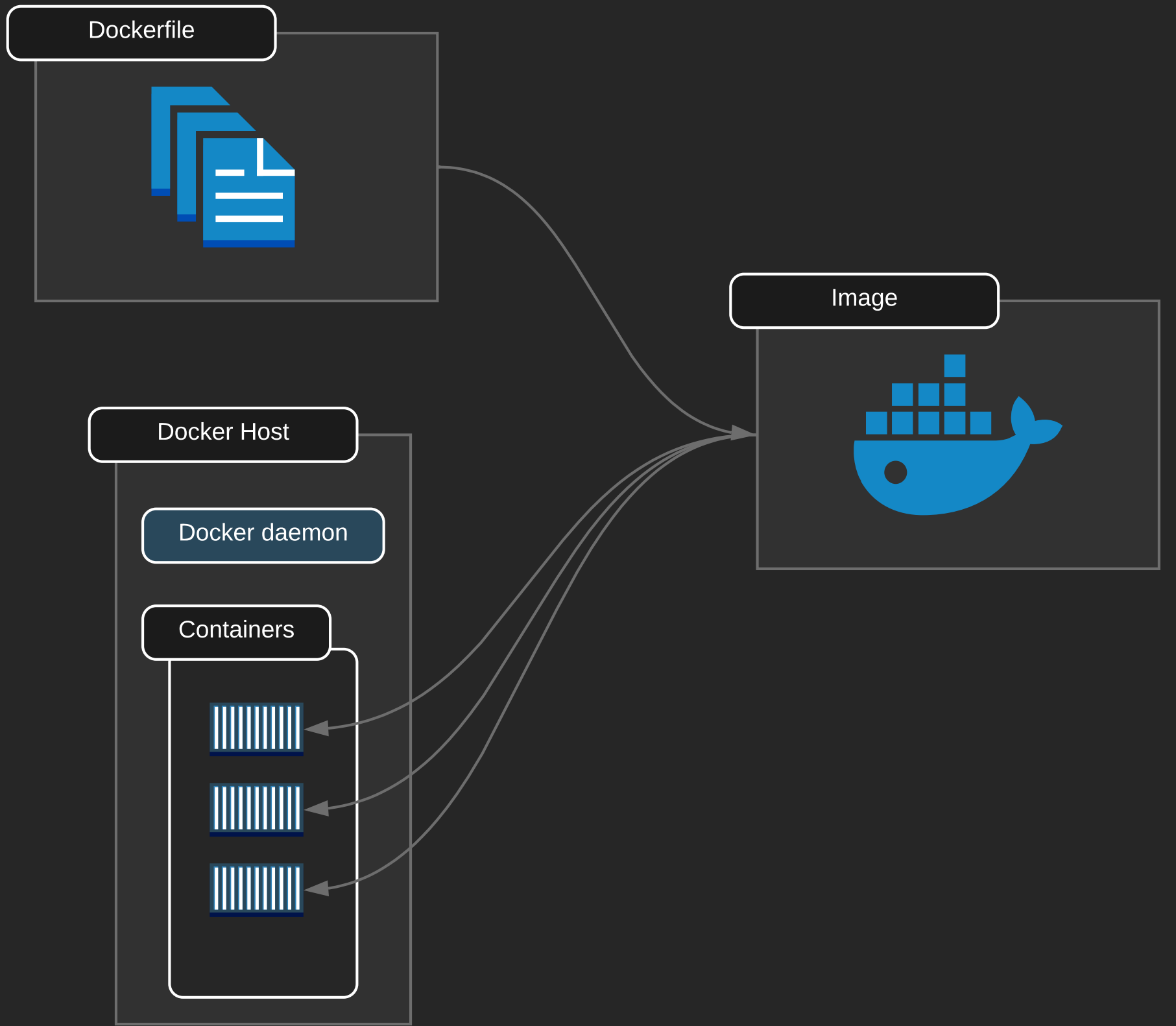
- Dev/Prod parody
- Simplifying Configuration
- Code Pipeline Management
- Developer Productivity
- App Isolation
- Server Consolidation
- Debugging Capabilities
- Multi-tenancy

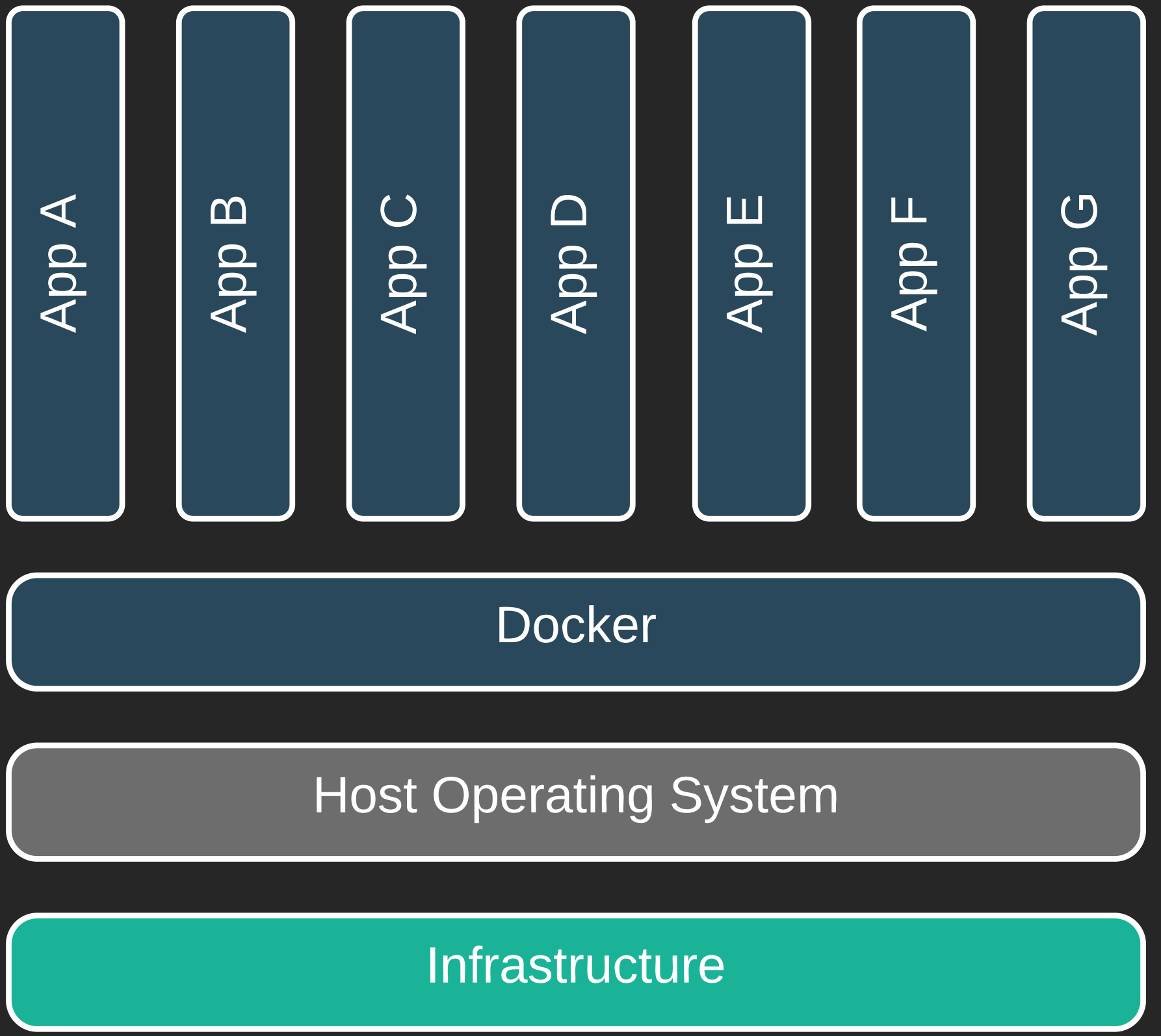

Docker client
>_
Docker Engine
daemon
containerd
runc

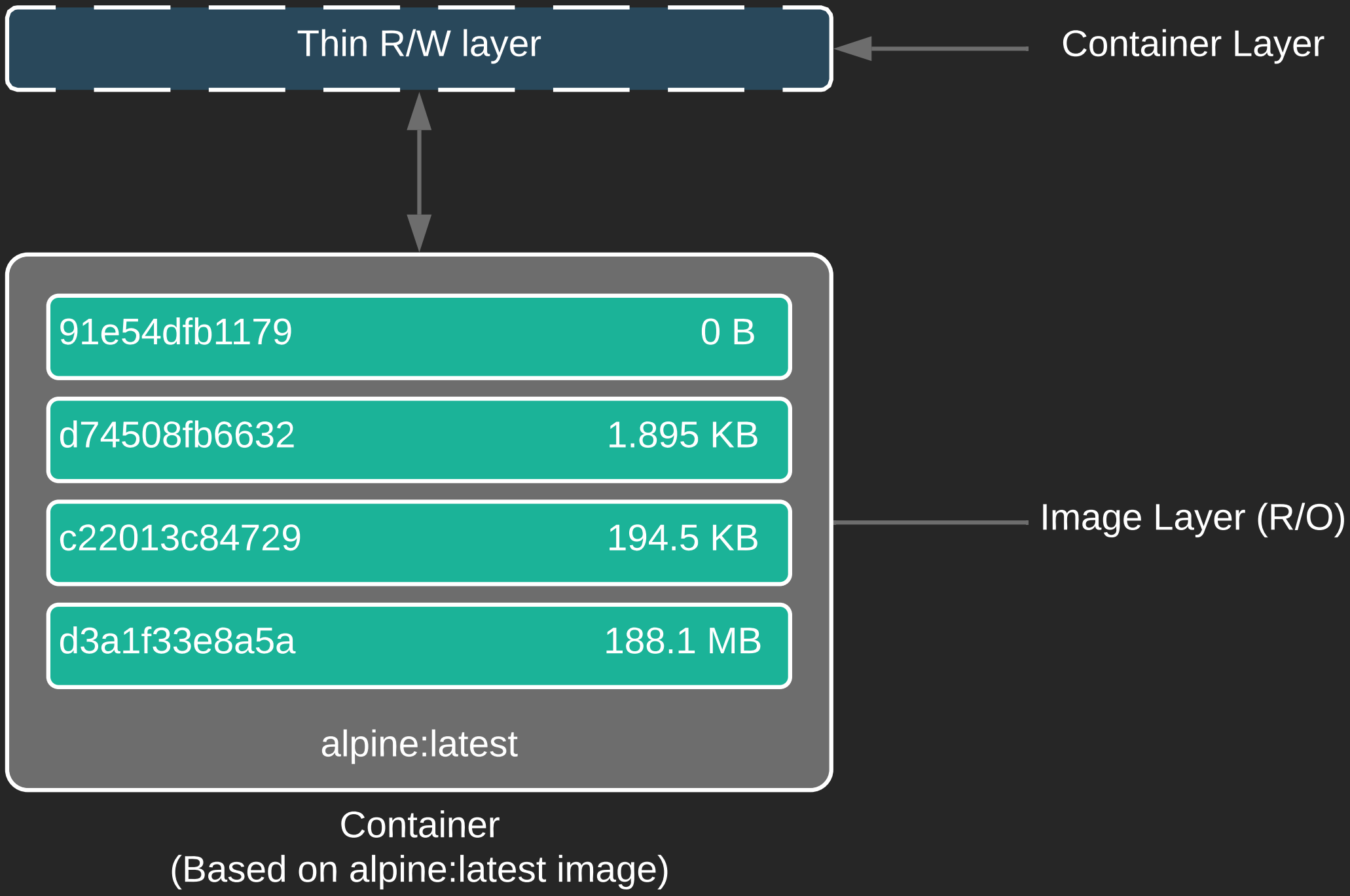

Docker client
>_
Docker daemon
containerd
shim
shim
shim
shim
runc
runc
runc
runc

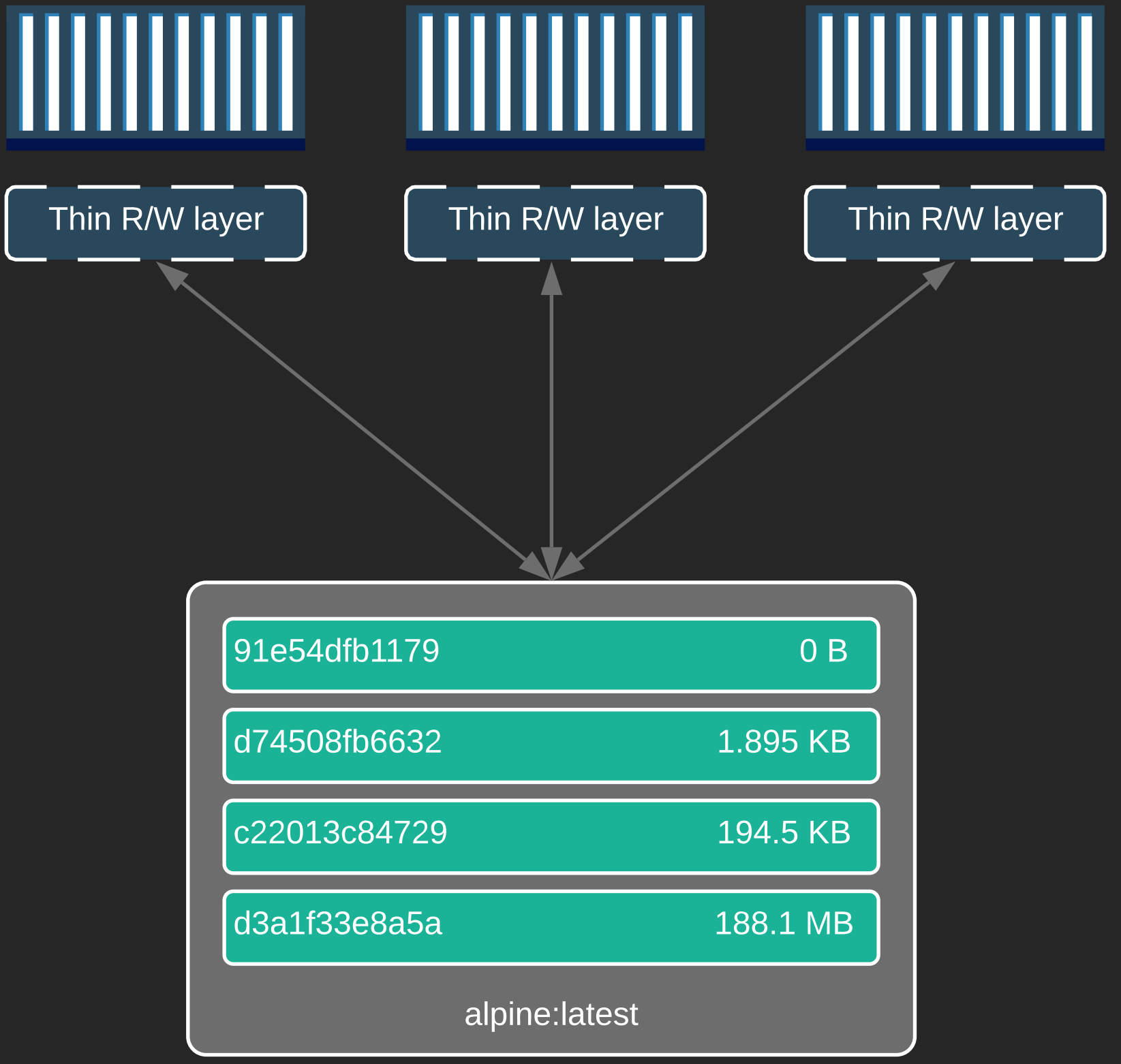

Docker daemon
LXC
Namespaces
Capablities
cgroups
Host Kernel

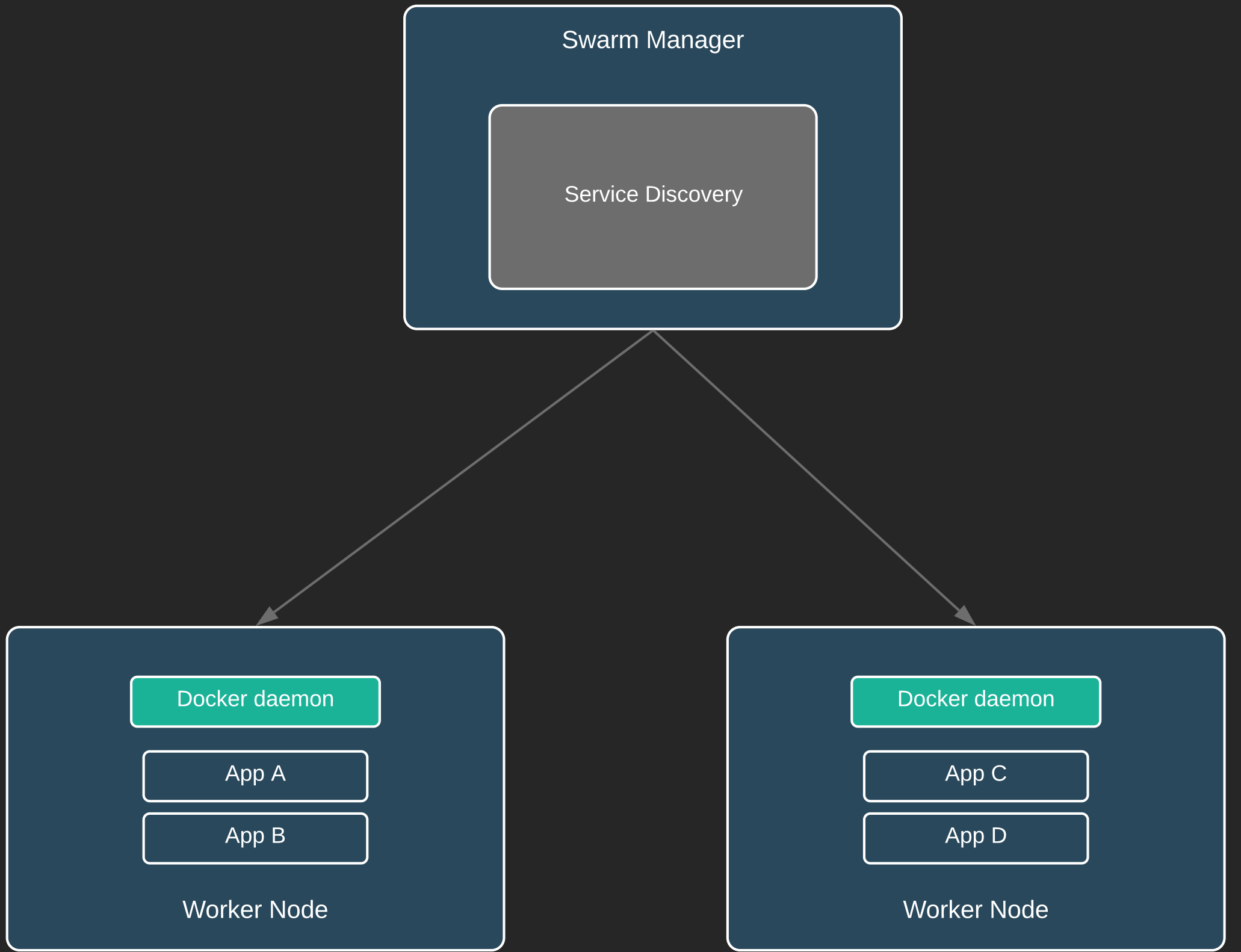

Docker client
>_
Docker Host
Docker daemon
Containers
Images
Registry

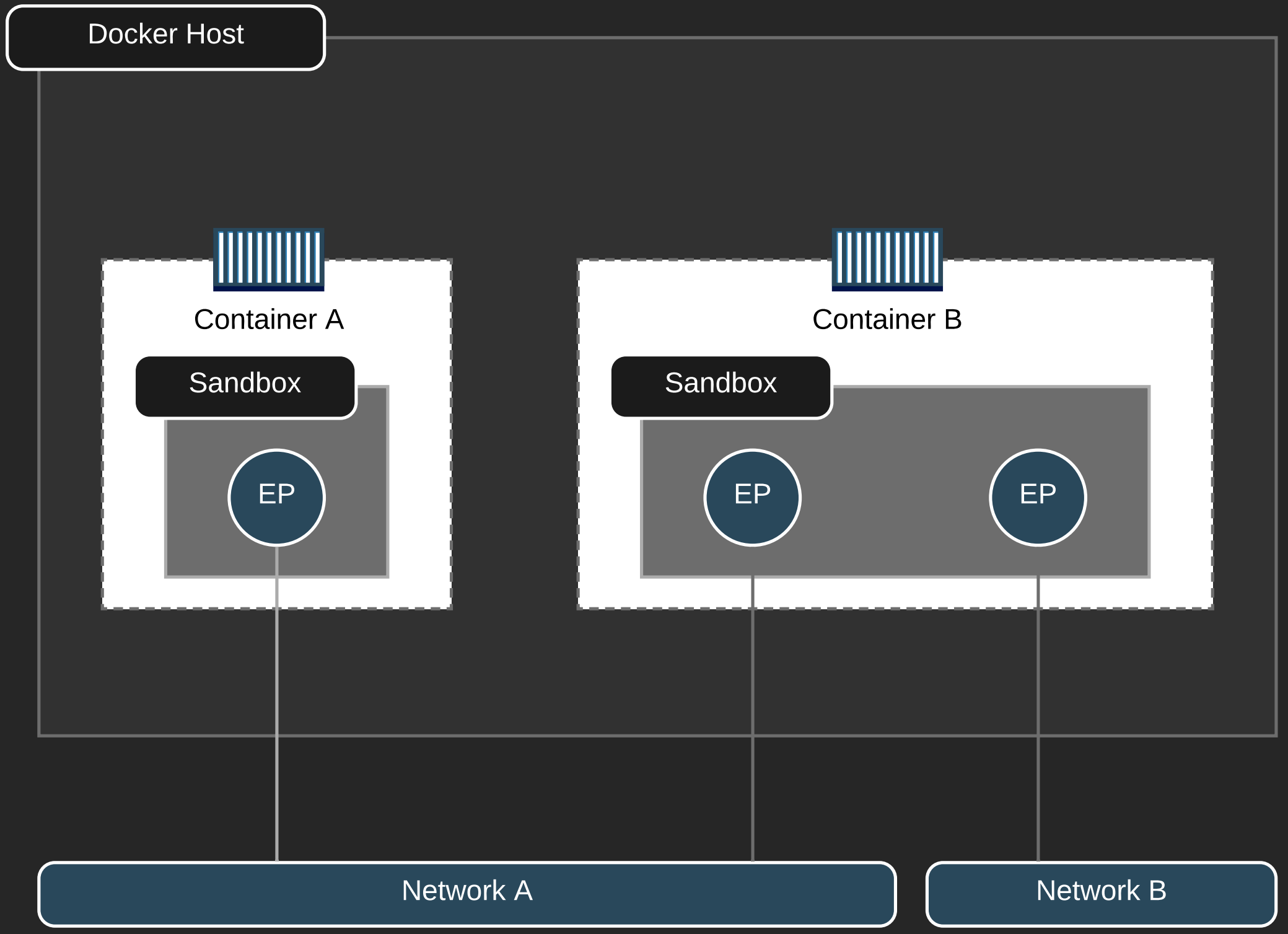

Dockerfile
Image
Docker Host
Docker daemon
Containers

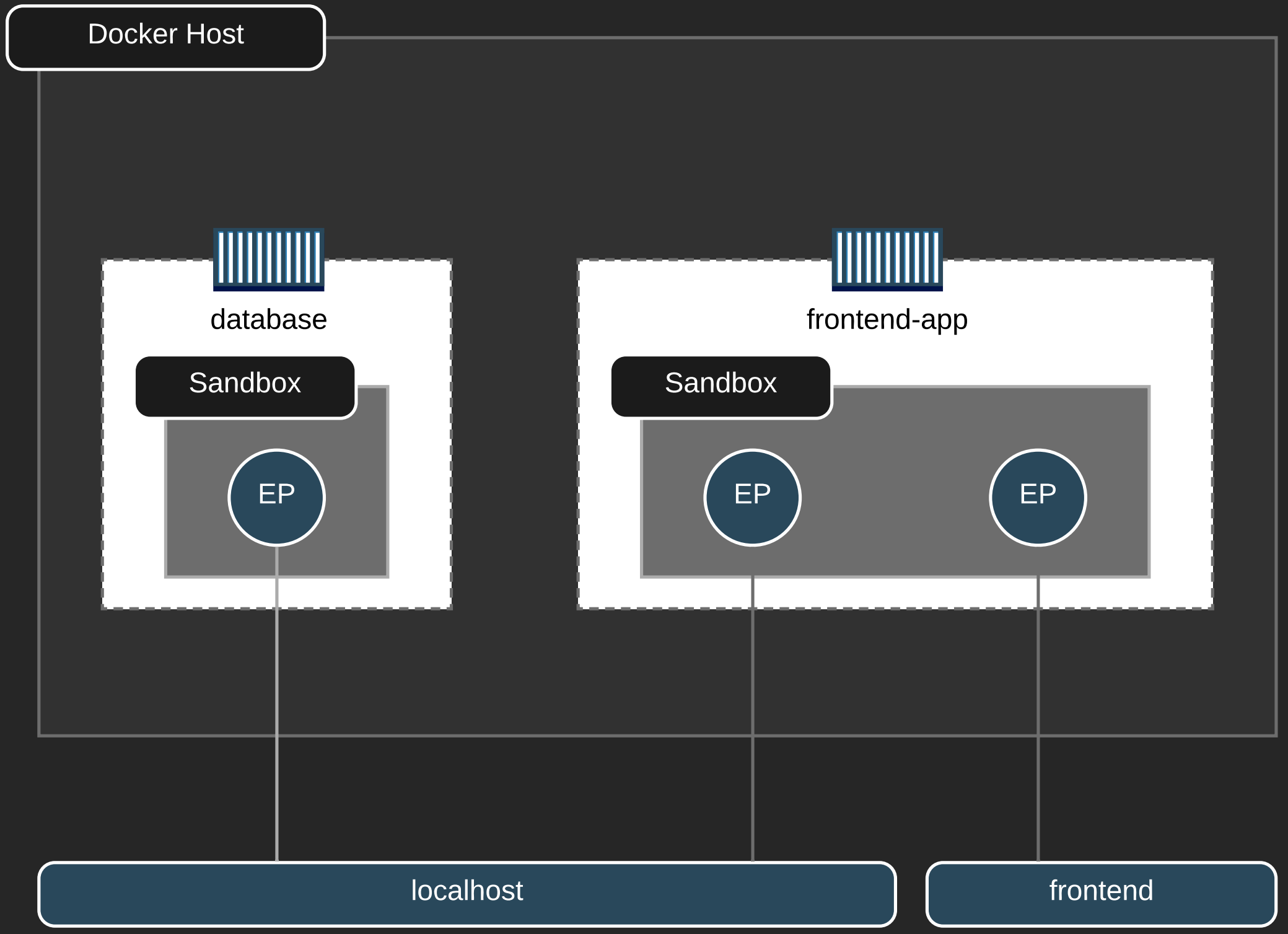

App A
App B
App C
App D
App E
App F
App G
Docker
Host Operating System
Infrastructure

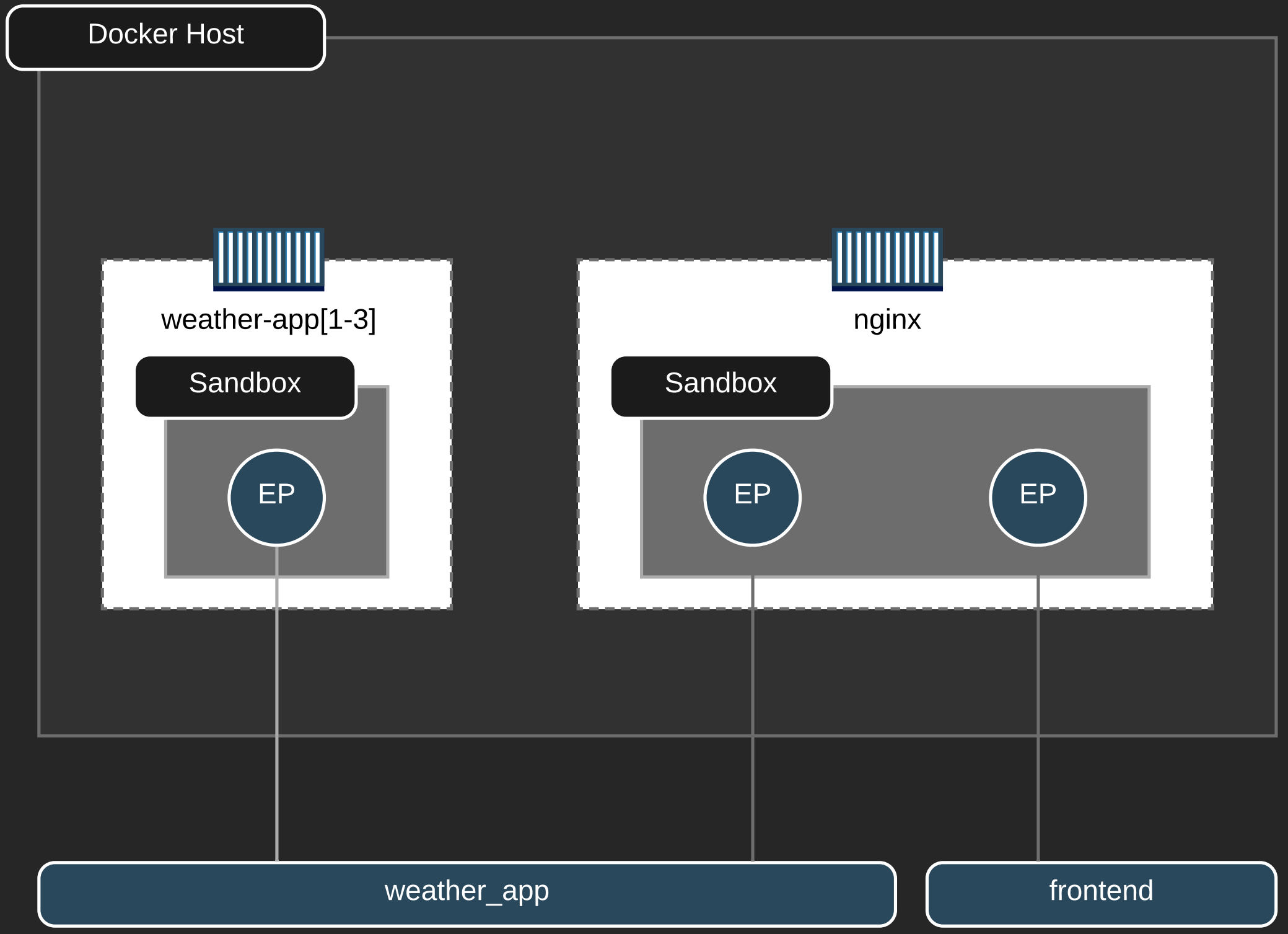

Thin R/W layer
Container Layer
91e54dfb1179 0 B
d74508fb6632 1.895 KB
c22013c84729 194.5 KB
d3a1f33e8a5a 188.1 MB
Image Layer (R/O)
alpine:latest
Container
(Based on alpine:latest image)

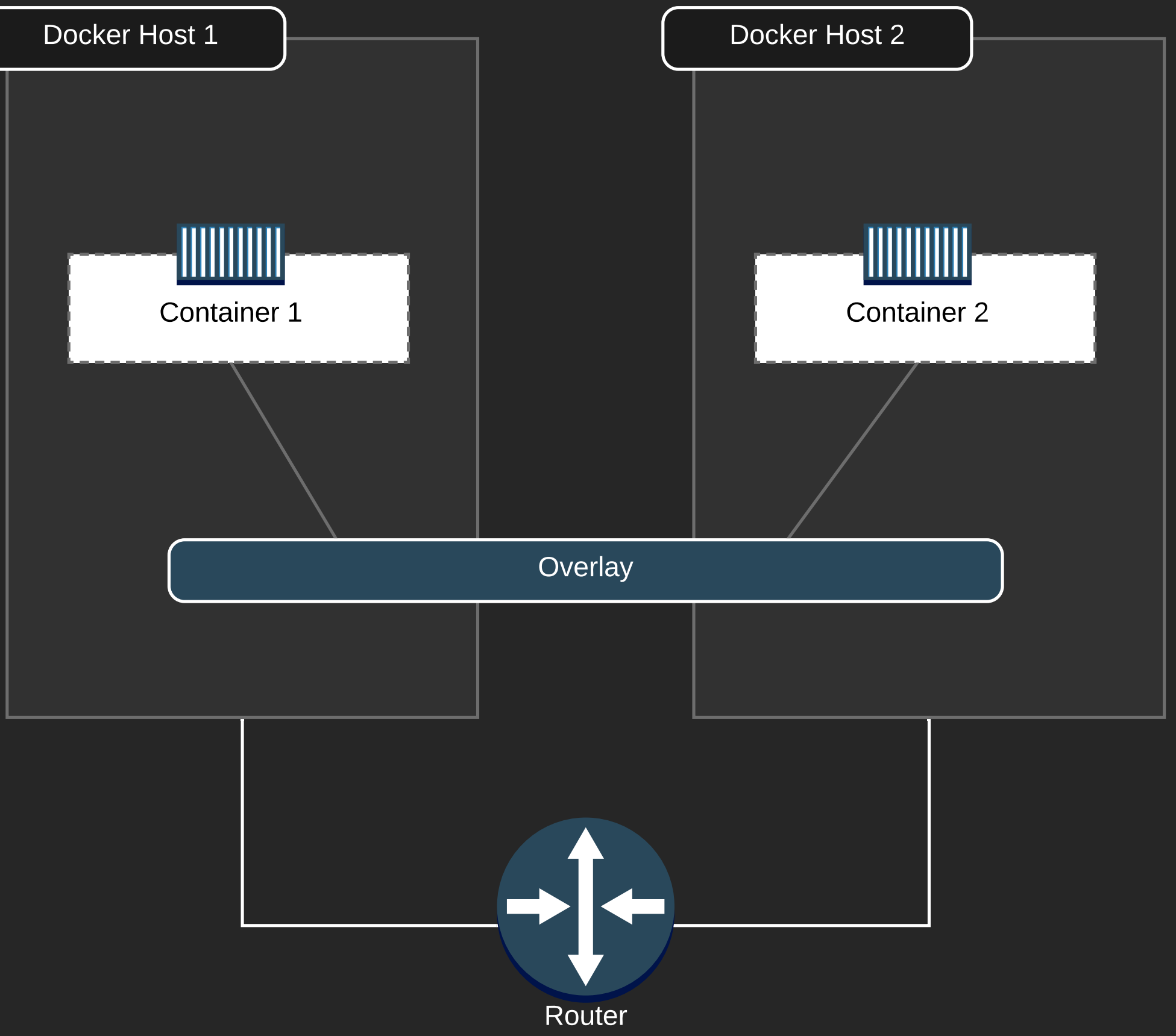

Thin R/W layer
Thin R/W layer
Thin R/W layer
91e54dfb1179 0 B
d74508fb6632 1.895 KB
c22013c84729 194.5 KB
d3a1f33e8a5a 188.1 MB
alpine:latest

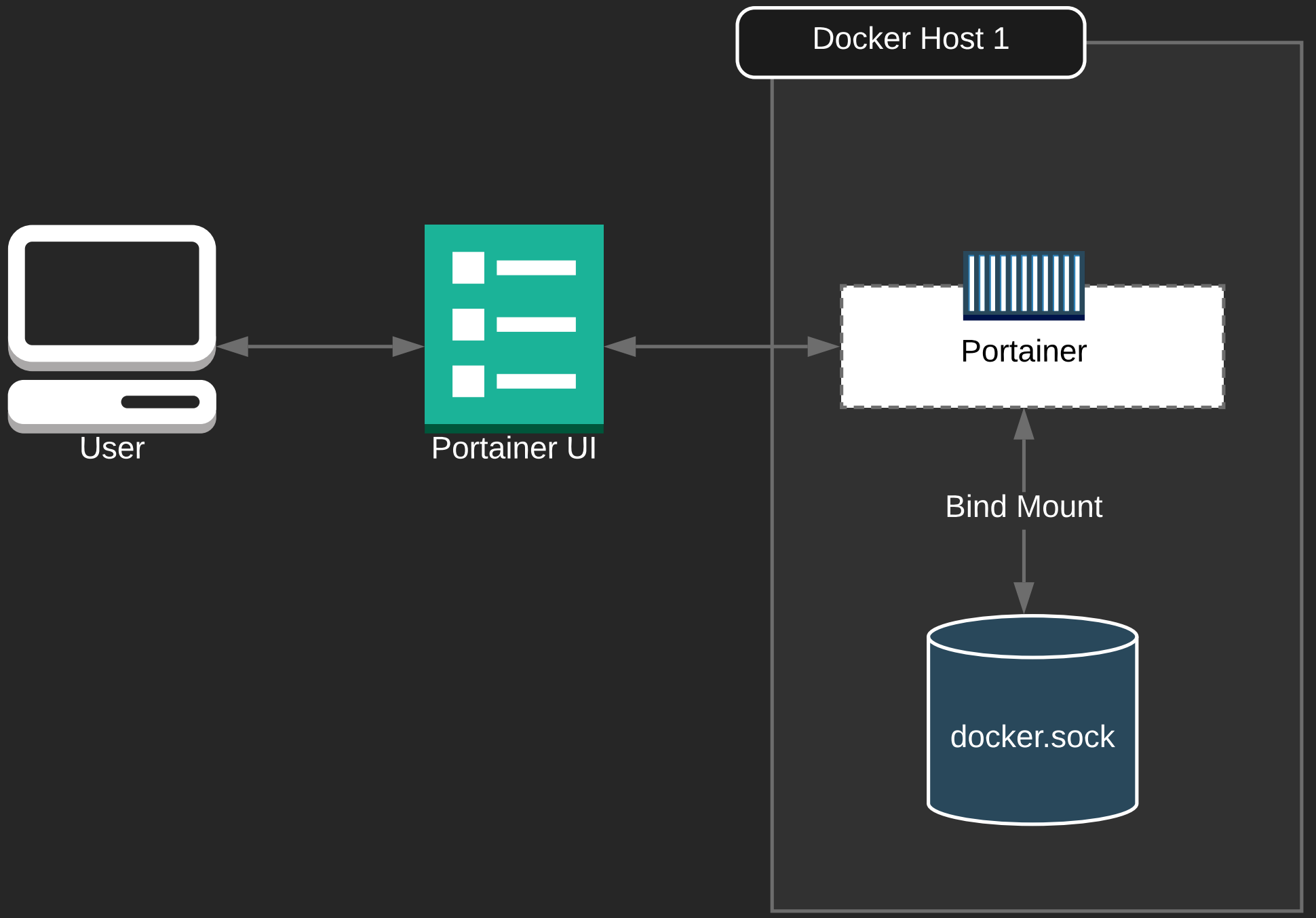

Swarm Manager
Service Discovery
Docker daemon
App A
App B
Worker Node
Docker daemon
App C
App D
Worker Node

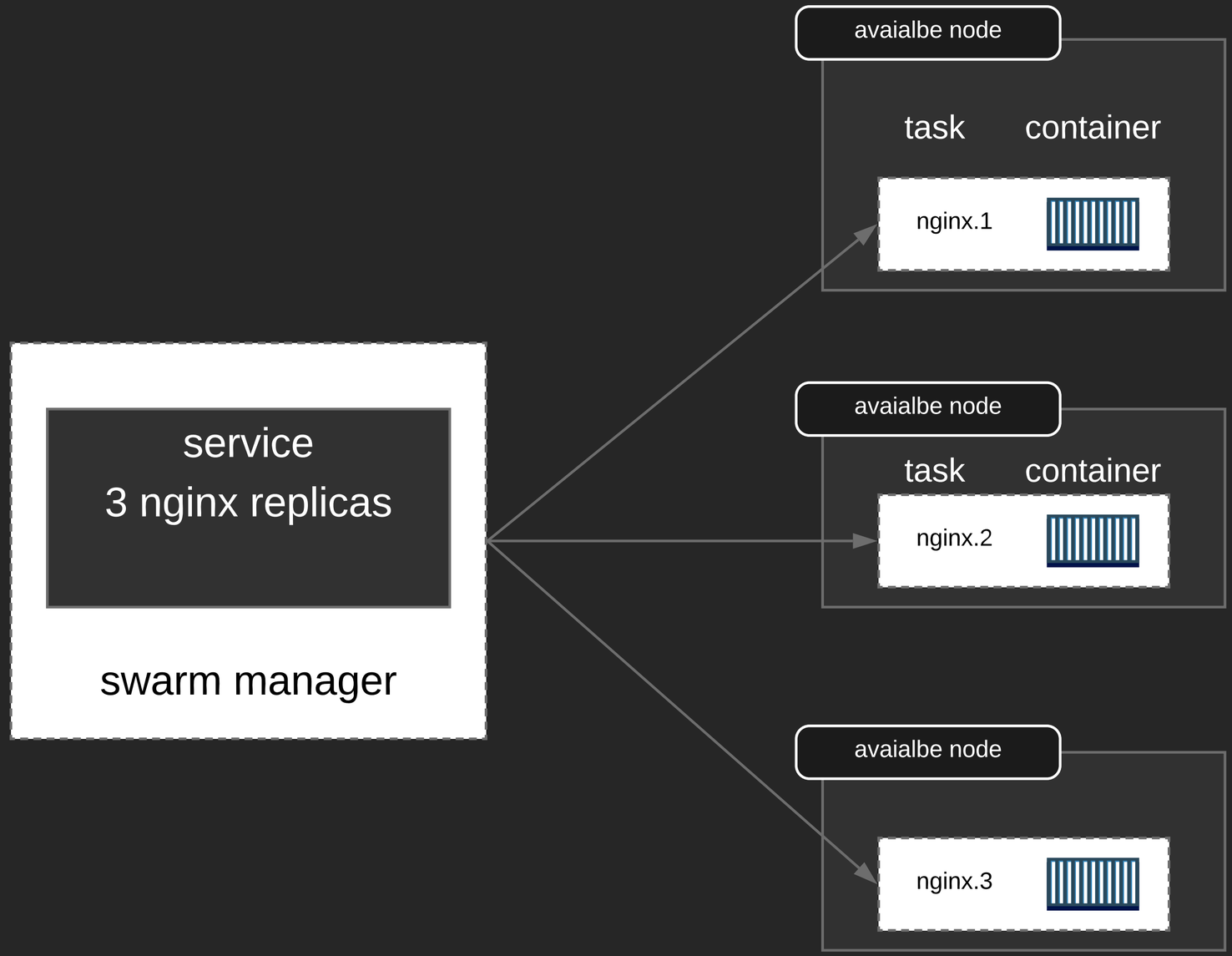

Docker Host
Container A
Sandbox
EP
Container B
Sandbox
EP
EP
Network A
Network B


Docker Host
database
Sandbox
EP
frontend-app
Sandbox
EP
EP
localhost
frontend


Docker Host
weather-app[1-3]
Sandbox
EP
nginx
Sandbox
EP
EP
weather_app
frontend


Docker Host 1
Docker Host 2
Container 1
Container 2
Overlay
Router

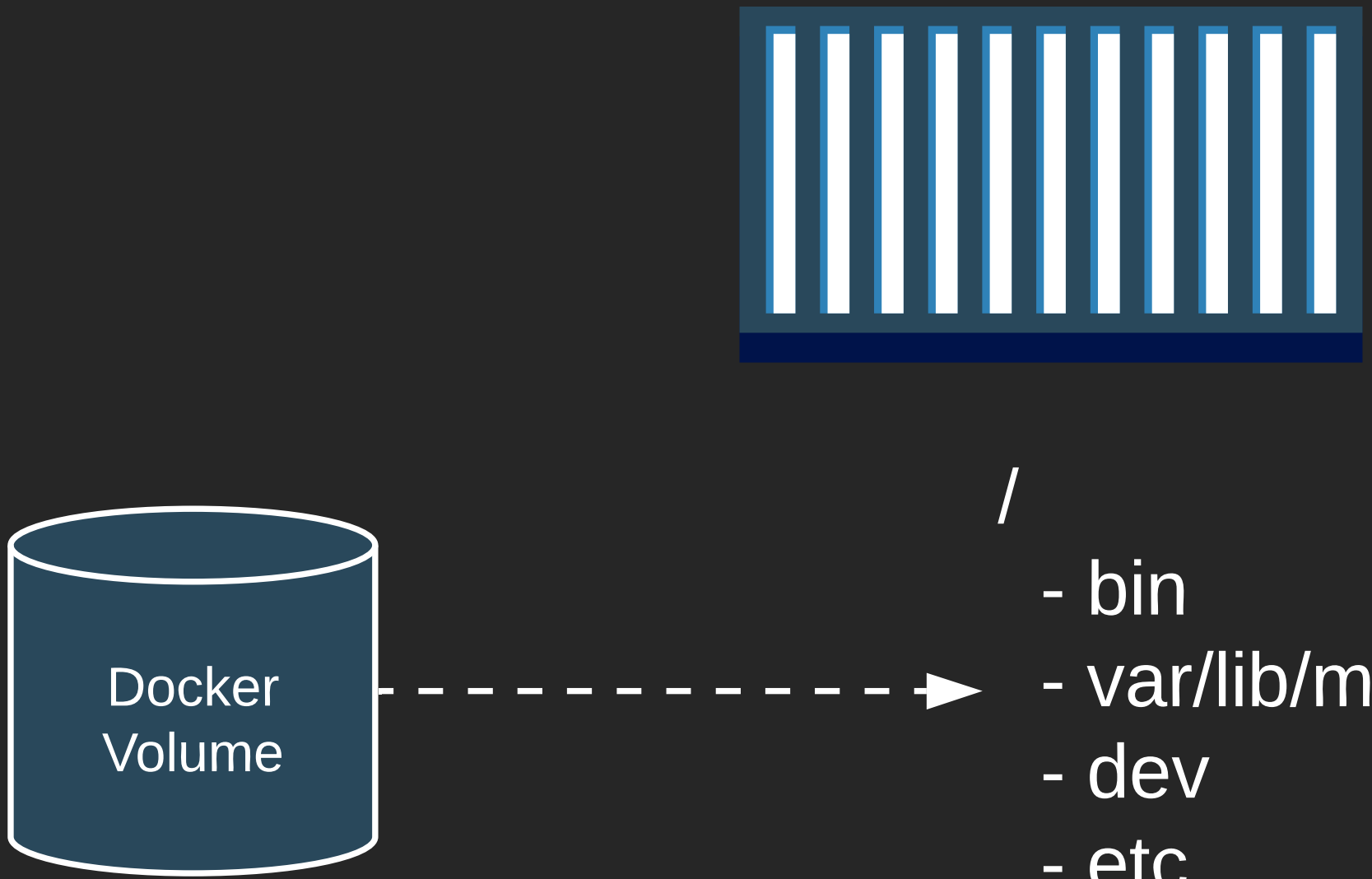

Docker Host 1
User
Portainer UI
Portainer
Bind Mount
docker.sock

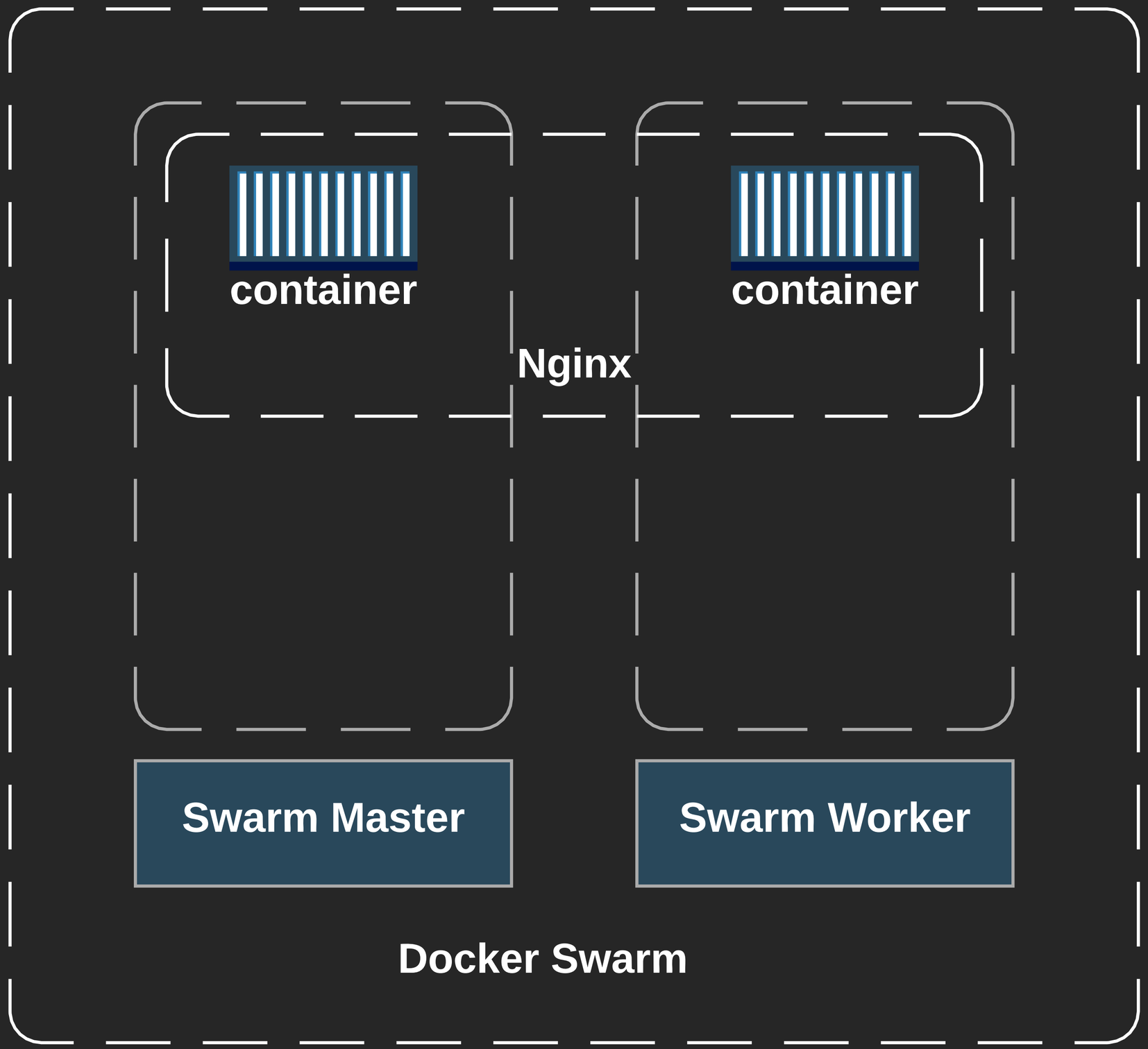

avaialbe node
task
container
nginx.1
avaialbe node
task
container
nginx.2
avaialbe node
nginx.3
service
3 nginx replicas
swarm manager

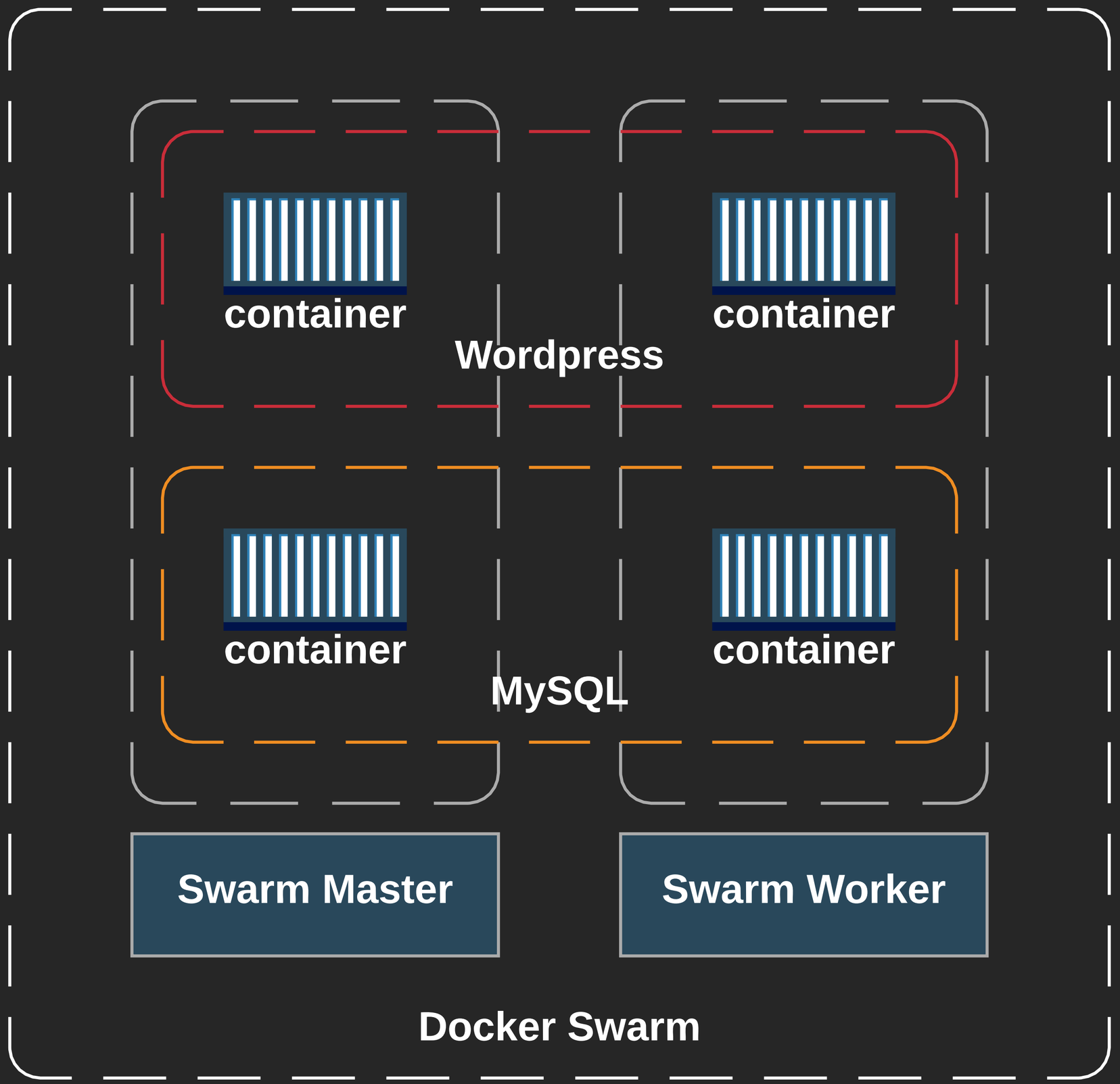
/

- bin
- var/code
- dev
- etc

Docker Volume


Docker
Volume
/
- bin
- var/lib/mysql
- dev
- etc


container
container
Nginx
Swarm Master
Swarm Worker
Docker Swarm


container
container
Wordpress
container
container
MySQL
Swarm Master
Swarm Worker
Docker Swarm

Docker platform technologies

Secrets Management

Docker Content Trust

Security Scanning

Swarm Mode

OS (Linux) technologies

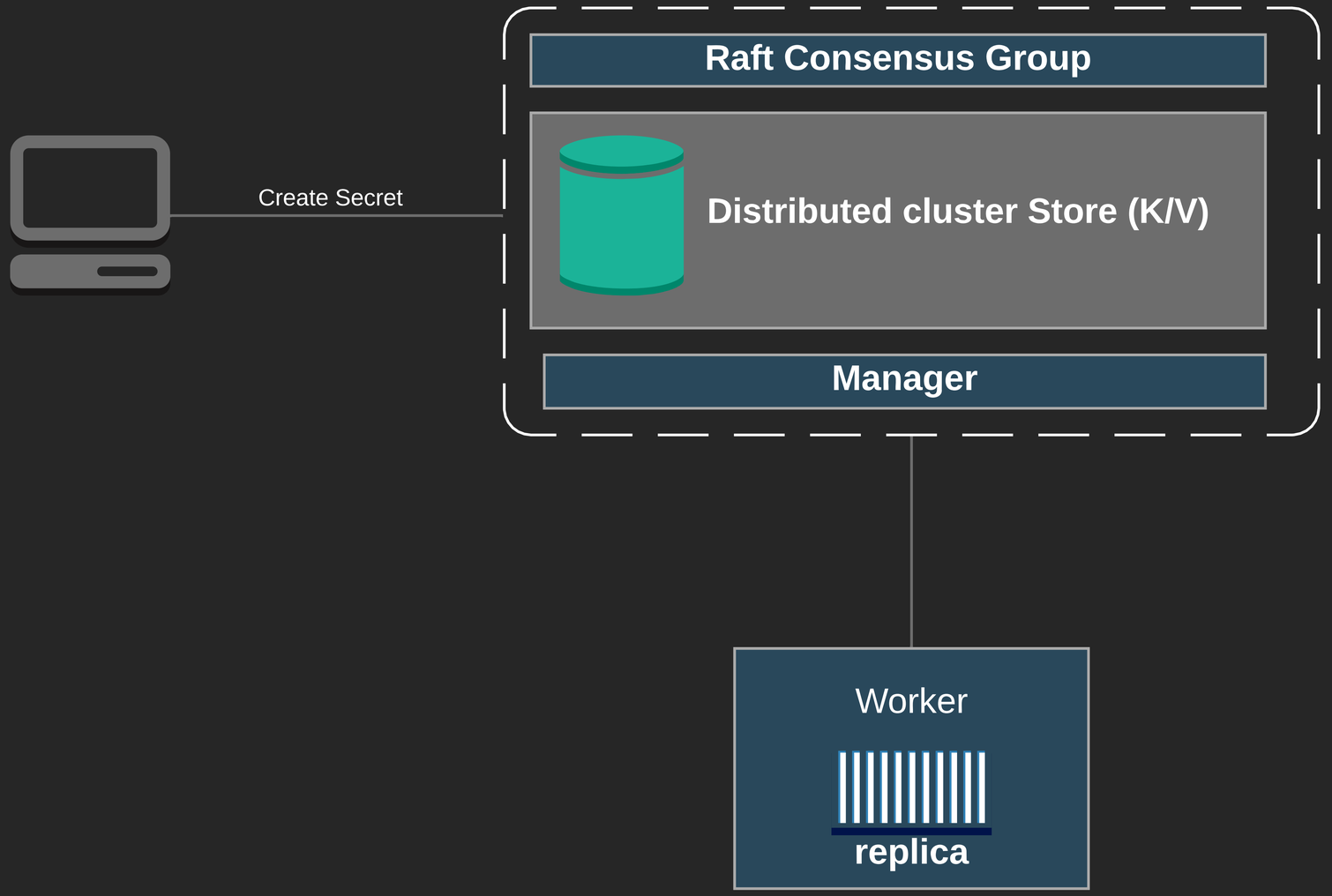
Seccomp

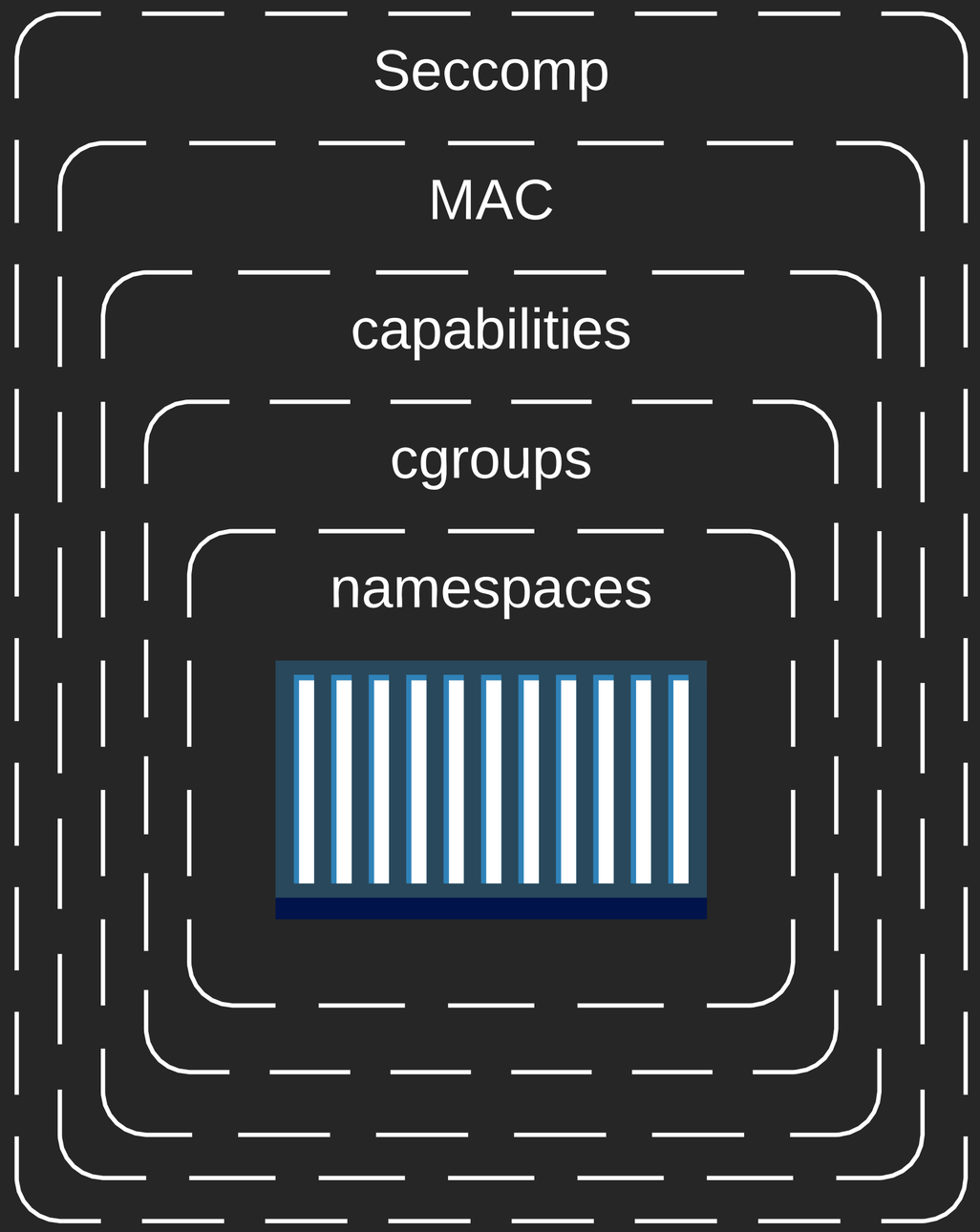
Mandatory Access Control

Capabilities

Control Groups

Kernel Namespaces


Raft Consensus Group
Distributed cluster Store (K/V)
Create Secret
Manager
Worker
replica


Seccomp
MAC
capabilities
cgroups
namespaces