The Great Escape: Why Containers Won't Save You

(and How MicroVMs Will)

Part 2: Underwater

Laurent Bobelin

laurent.bobelin@insa-cvl.fr

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE-VAL DE LOIRE

ToC

1. Introduction

2. A Deeper Dive into Container Management and Docker Example
2.1 Overview
2.2 Runtime

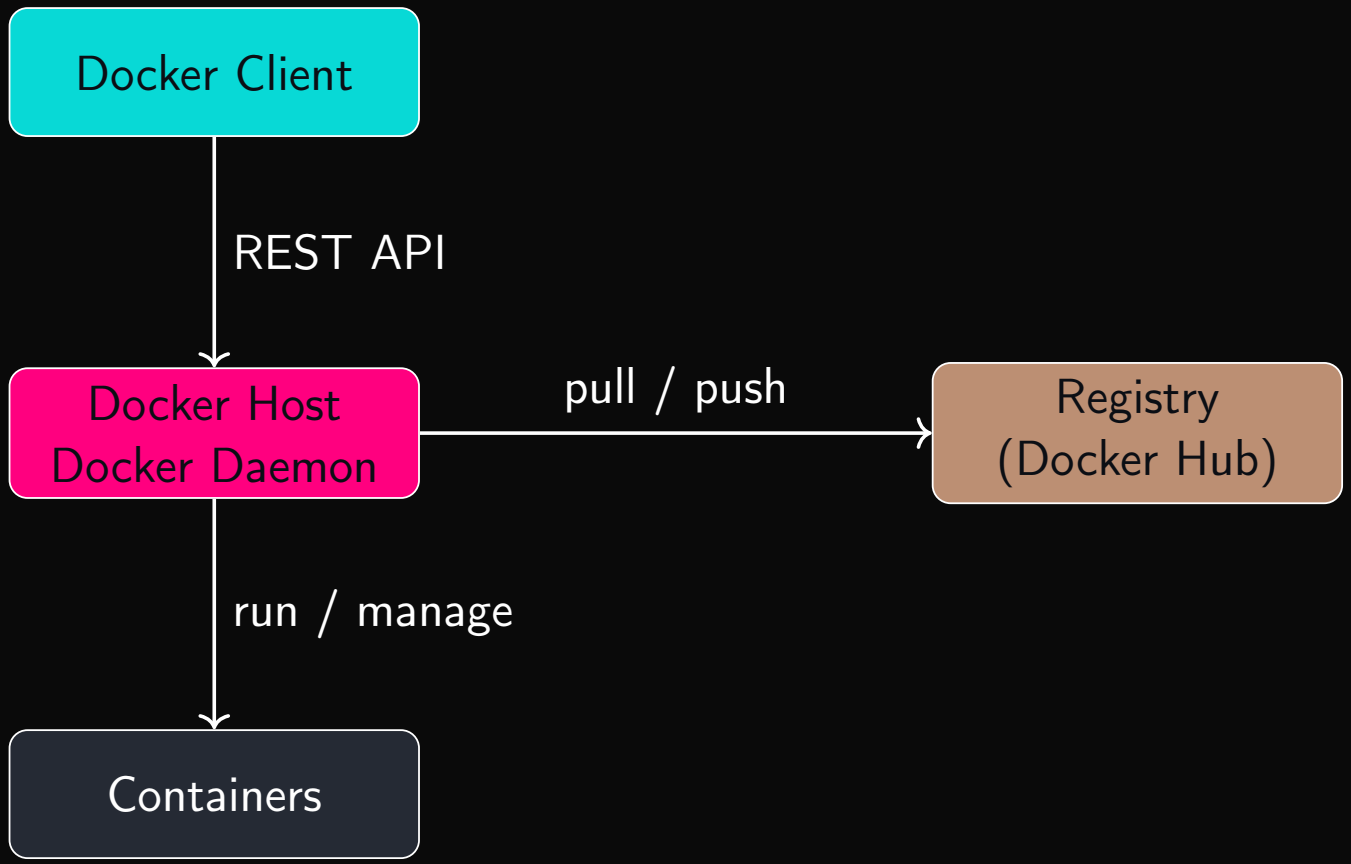
3. OCI Specifications
3.1 OCI Runtime
3.2 OCI Image
3.3 OCI Distribution Specification

4. Conclusion

Features of a Container Management Tool

- Manage image repositories
- Prepare images
- Create and manipulate instances
- Configure instances
- Manage applications launched within containers
- When several machines host containers, allow the application to be distributed between these machines (orchestrate)
- ...

Docker Architecture



Docker

The previous description does say you how to manipulate your containers but:

- ... it does not say how the are managed
- ... Neither what are the underneath tools used

We discussed isolation mechanisms are used but:

- ... How are they effectively instanciated
- ... Neither if there are some standards for doing it.

ToC

1. Introduction

2. A Deeper Dive into Container Management and Docker Example
2.1 Overview
2.2 Runtime

3. OCI Specifications
3.1 OCI Runtime
3.2 OCI Image
3.3 OCI Distribution Specification

4. Conclusion

ToC

1. Introduction

2. A Deeper Dive into Container Management and Docker Example
2.1 Overview
2.2 Runtime

3. OCI Specifications
3.1 OCI Runtime
3.2 OCI Image
3.3 OCI Distribution Specification

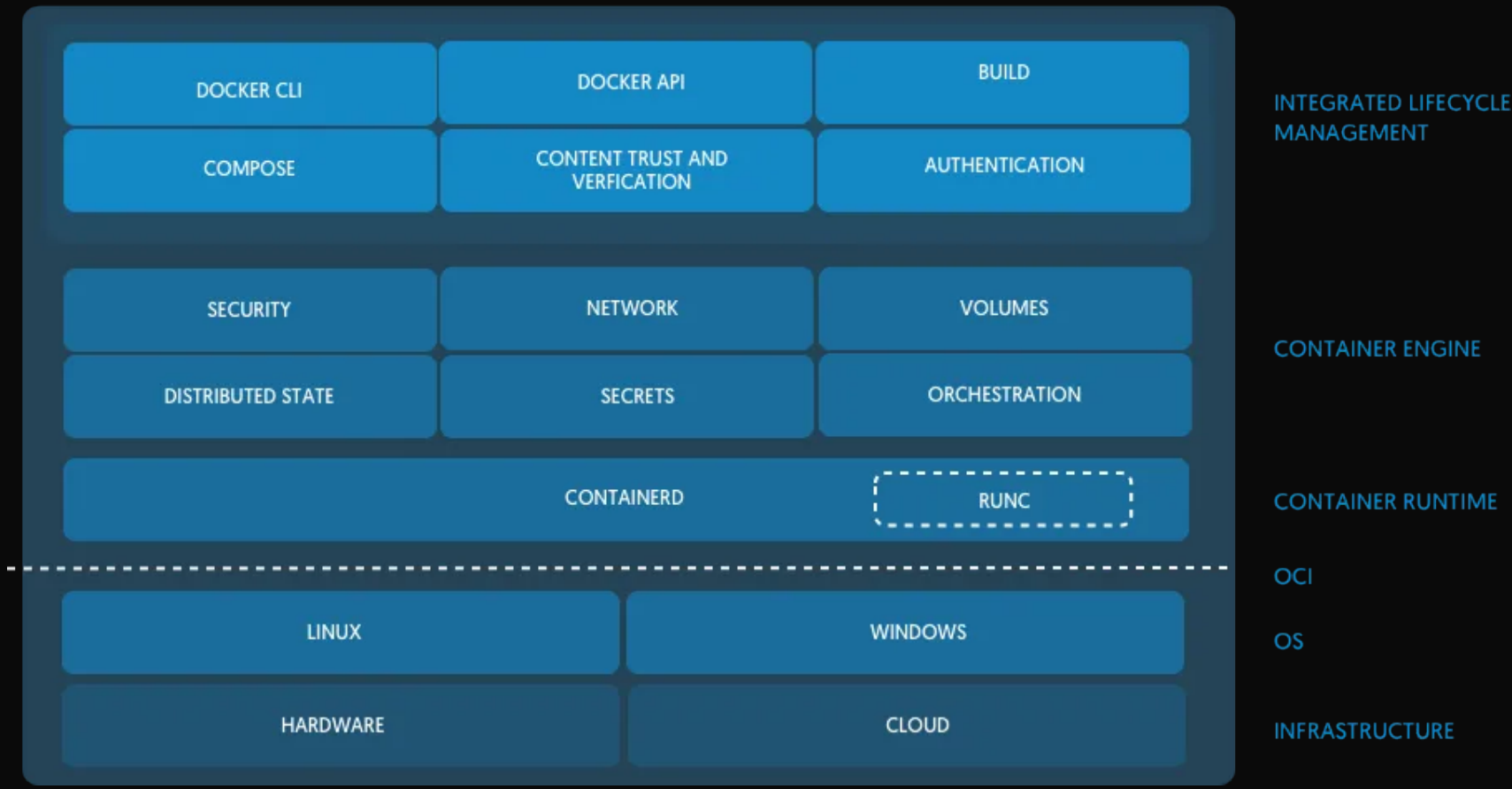
4. Conclusion

Overview

Docker is divided into different layers:

- Lifecycle management: command line interface , api, how to build image, orchestrate containers, authenticate users...
- Container engine: all usefull stuff like volumes management, network, security management, and so on...
- Container runtime: create/delete/pause/restart a container... Two kinds in Docker:
  - High-level container runtime, that manages the whole container lifecyle
  - Low-level container runtime, that just runs the container process

The Big Picture



Daemon Point of View

- Docker daemon communicates with upper client (CLI for example) ; it does high level operation for orchestrating local resources.
- Containerd daemon handle the compatibility with a standard (OCI) to help Docker daemon to orchestrate things
- Containerd will discuss with container runtimes that does start/stop containers via a shim, that does the interface between containerd and runtimes.

Docker and low-level runtimes:

- no dependency!
- Low-level runtimes can be changed and configured dynamically
- runc is just the default runtime for Docker

Integrated Lifecycle Management

Docker tools:

- Compose: how to orchestrate the behaviour of containers on the same host
- Swarm: how to orchestrate containers hosted by more than one computer.

Non-Docker tools includes kubernetes, openshift...

Container Engine

Responsibilities:

- Handling input (CLI, API ...)
- Pulling the Container Images from the Registry Server
- Decompressing and expanding the container image on disk
- Preparing a container mount point
- Preparing the metadata which will be passed to the container Container Runtime to start the Container correctly
- Parse defaults from the container image (ex.ArchX86)
- Parse user input overriding defaults
- Use these parameters to call the Container Runtime chosen by user.

ToC

1. Introduction

2. A Deeper Dive into Container Management and Docker Example
2.1 Overview
2.2 Runtime

3. OCI Specifications
3.1 OCI Runtime
3.2 OCI Image
3.3 OCI Distribution Specification

4. Conclusion

High-Level vs. Low-Level Runtimes

Feature	runc	containerd
Type	Low-level (OCI)	High-level (Daemon)
Scope	Process execution	Lifecycle management
Persistence	Exits after start	Runs as background service
Capabilities	Creates/Starts containers	Pulls images, manages storage
Standard	OCI Specification	CRI (for Kubernetes)

- **runc** is the "Doer" (executes the recipe).
- **containerd** is the "Manager" (oversees the kitchen).

Key Architectural Concepts

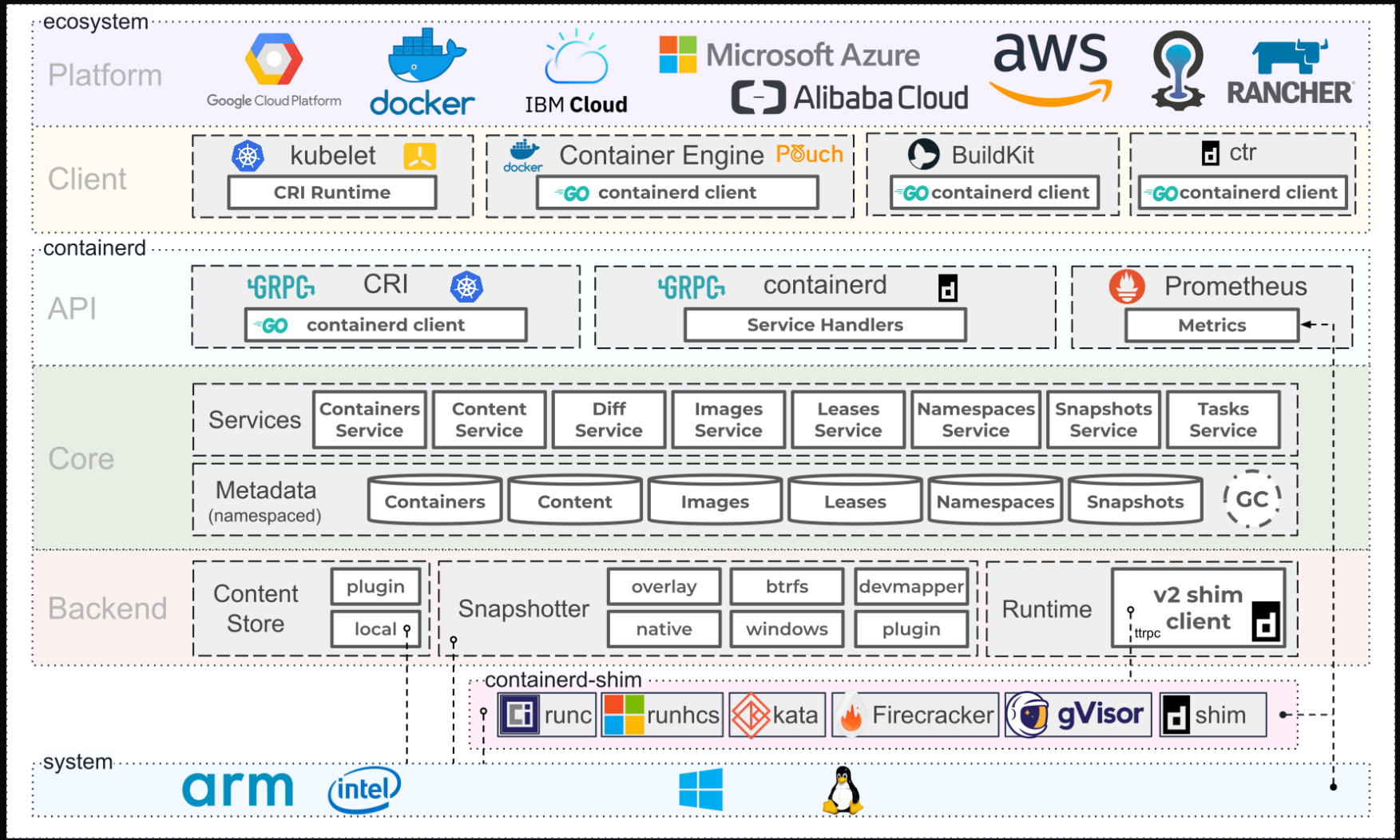
The Role of runc (The "Doer")

- Lightweight CLI tool interacting directly with the Linux kernel.
- Handles namespaces, cgroups, and security profiles.
- Does not know how to pull images or manage networks.

The Role of containerd (The "Manager")

- Persistent daemon that sits between the orchestrator and runc.
- Manages image pushing/pulling and local storage snapshots.
- Instructs runc on how and when to execute containers.

The Big Picture: Containerd



Containerd Features

- OCI Image Spec support
- Image push and pull support
- Network primitives for creation, modification, and deletion of interfaces
- Multi-tenant supported with CAS storage for global images
- OCI Runtime Spec support (aka runC)
- Container runtime and lifecycle support
- Management of network namespaces containers to join existing namespaces

Low-level Runtime

Purpose: actually run the container. Docker's default is runc:

- Interface with the kernel
- Construct namespaces
- Configure cgroups
- Configure appArmor/SELinux/...
- Container process is started as a child process of runc, and as soon as it starts, runc exits.

Shim

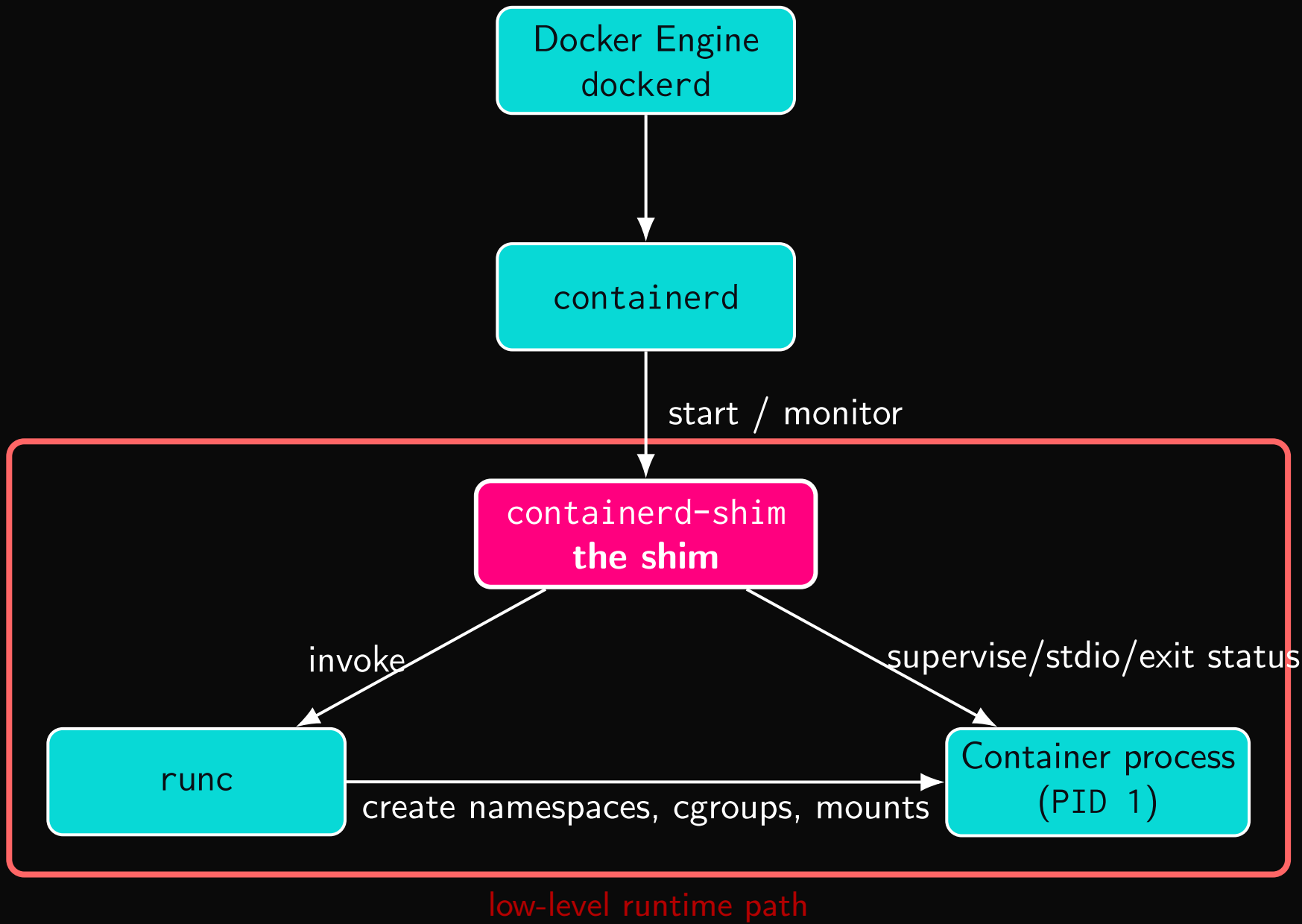
Decouple running containers from the daemon for things like daemon upgrades:

- new container: new runc instance
- Once created: runc exits
- runc exits: containerd-shim becomes the new parent.

What shim does:

- Keep STDIN & STDOUT streams open, even if daemon restarts
- Reports the container's exit status back to the daemon
- ...

What is a Shim in the Container Stack?



ToC

1. Introduction

2. A Deeper Dive into Container Management and Docker Example
2.1 Overview
2.2 Runtime

3. OCI Specifications
3.1 OCI Runtime
3.2 OCI Image
3.3 OCI Distribution Specification

4. Conclusion

OCI Specifications

Three specifications:

- Runtime Specification (runtime-spec): how to run a "filesystem bundle" that is unpacked on disk.
- Image Specification (image-spec): how to construct the bundle
- Distribution Specification (distribution-spec): how to distribute the bundle

At a high-level an OCI implementation would download an OCI Image then unpack that image into an OCI Runtime filesystem bundle. At this point the OCI Runtime Bundle would be run by an OCI Runtime.

ToC

1. Introduction

2. A Deeper Dive into Container Management and Docker Example
2.1 Overview
2.2 Runtime

3. OCI Specifications
3.1 OCI Runtime
3.2 OCI Image
3.3 OCI Distribution Specification

4. Conclusion

Runtime

The Open Container Initiative Runtime Specification aims to specify the configuration, execution environment, and lifecycle of a container.

- configuration file formats
- a set of standard operations
- an execution environment.

Runtime Specification Principle

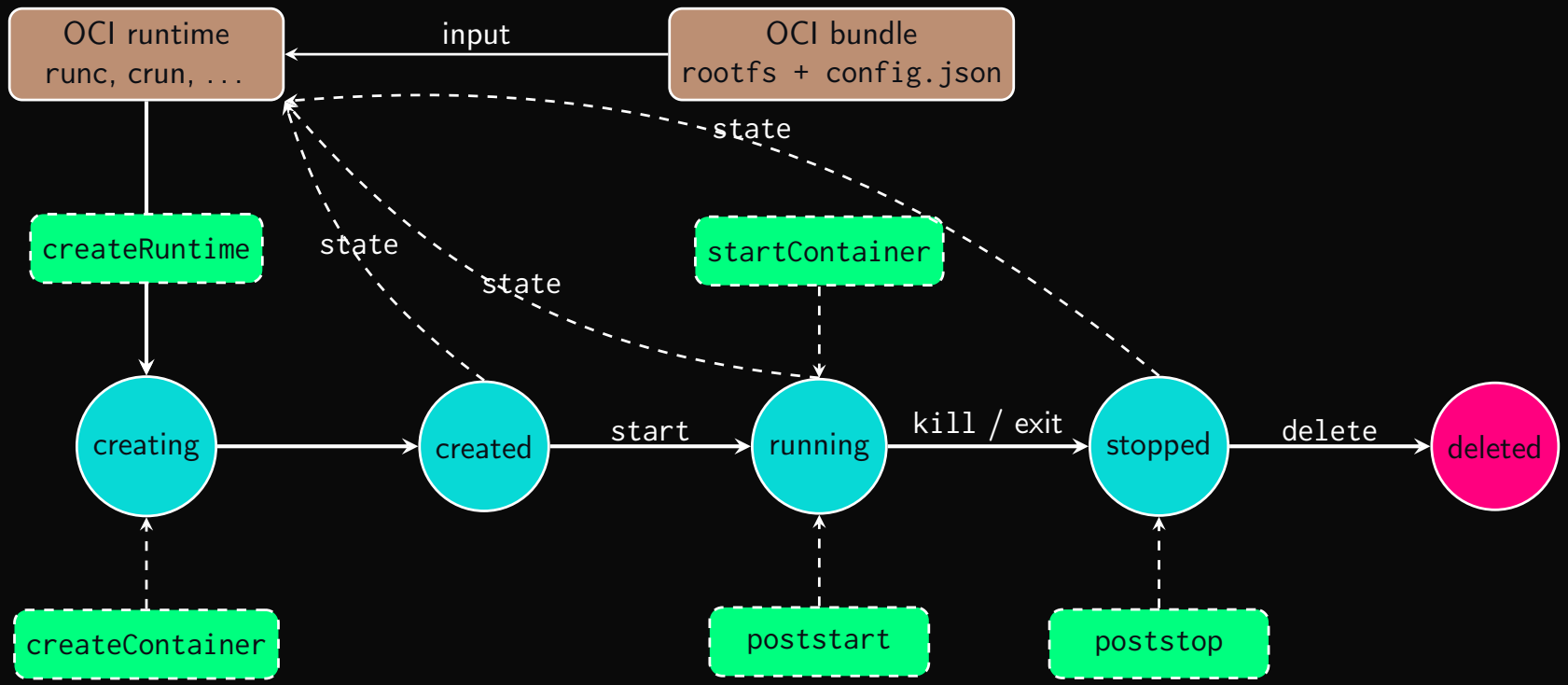
Runtime specifications are written based on 5 main principles:

- Standard operations: created, start, stop, copy, snapshot, download and upload
- Content agnostic: all standard operations have the same effect regardless of the contents
- Infrastructure agnostic: can be run in any OCI supported infrastructure
- Designed for automation: same standard operations regardless of content and infrastructure.
- Industrial-grade delivery: enabling large and small enterprises to streamline and automate their software delivery pipelines.

OCI Bundle

- An **OCI bundle** is the directory given to the OCI runtime.
- It contains:
  - `config.json` – container configuration (namespaces, mounts, cgroups, hooks, process)
  - `rootfs/` – the container root filesystem
- The runtime reads the bundle when executing commands such as: `runc create <container>`
- Bundles make container execution **portable and reproducible**.

OCI Container Lifecycle



OCI Hooks: Purpose and Execution Points

- **Hooks** are external programs executed by the OCI runtime at specific lifecycle events.
- They allow **custom logic without modifying the runtime**.
- Defined in the container's `config.json`.
- Typical uses:
  - network configuration
  - device injection (e.g., GPUs)
  - security setup (SELinux/AppArmor)
  - logging or monitoring
- Hooks receive container metadata through environment variables and JSON on stdin.

ToC

1. Introduction

2. A Deeper Dive into Container Management and Docker Example
2.1 Overview
2.2 Runtime

3. OCI Specifications
3.1 OCI Runtime
3.2 OCI Image
3.3 OCI Distribution Specification

4. Conclusion

OCI Image Specifications

OCI Images are a standardized format for packaging container software.

- Provide a portable way to distribute containerized applications.
- Used by container runtimes and platforms such as:
  - Docker
  - containerd
  - CRI-O
  - Kubernetes
- OCI images are stored in **container registries**.
- An image is composed of:
  - **metadata describing the container**
  - **filesystem layers**
- Images are **content-addressed**: each component is identified by a cryptographic digest.

Goal: interoperability between tools and runtimes.

OCI Image Structure

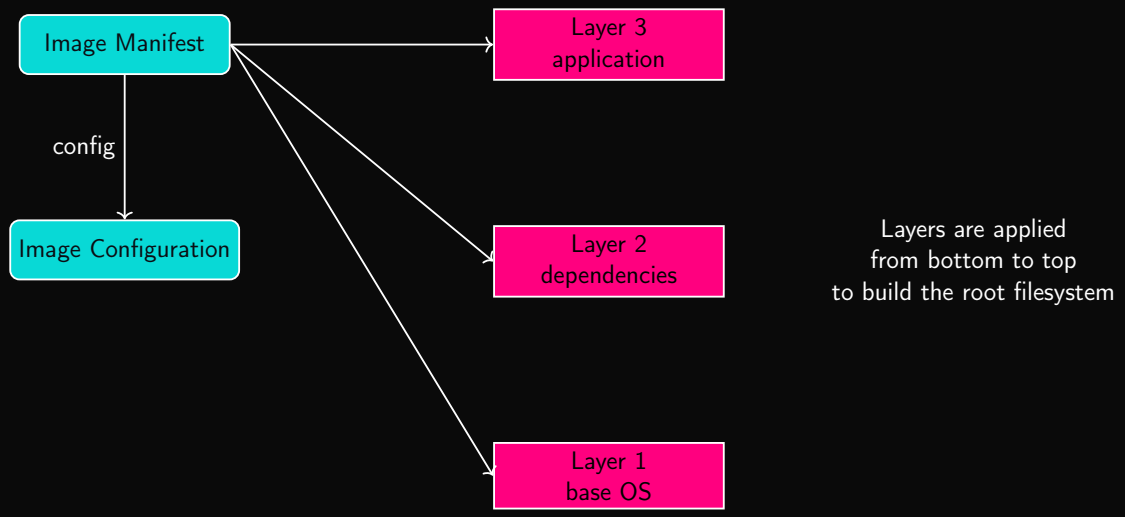
An OCI image is composed of several objects stored in a registry:

- **Image Manifest**
  - Describes the image structure
  - Lists configuration object and layers
- **Image Configuration**
  - Runtime configuration of the container
  - Environment variables
  - Entrypoint / command
  - Metadata about layers
- **Filesystem Layers**
  - Tar archives containing filesystem changes
  - Stacked to form the container root filesystem

Manifest example:

- reference to configuration object
- ordered list of layers
- media types
- cryptographic digests

Example OCI Image



- The **manifest** references the configuration and the layers.
- The **configuration** describes runtime parameters (env, entrypoint, etc.).
- Layers are **stacked** to form the container filesystem.
- Layers can be **shared between images**.

ToC

1. Introduction

2. A Deeper Dive into Container Management and Docker Example
2.1 Overview
2.2 Runtime

3. OCI Specifications
3.1 OCI Runtime
3.2 OCI Image
3.3 OCI Distribution Specification

4. Conclusion

OCI Distribution Specification

- The **OCI Distribution Specification** defines how container images are **stored and retrieved from registries**.
- It standardizes the **HTTP API** used by container registries.
- Main goal: ensure **interoperability** between:
  - container engines (Docker, Podman, containerd)
  - registries (Docker Hub, Harbor, GitHub Container Registry, etc.)
- Operations supported by the specification include:
  - pushing images
  - pulling images
  - retrieving manifests and layers
- It is largely derived from the **Docker Registry HTTP API v2**.

How Image Distribution Works

- Container images are stored in **registries** as:
  - a **manifest**
  - a **configuration object**
  - multiple **filesystem layers**
- Images are identified by:
  - **repository name** (e.g., `nginx`)
  - **tag** (e.g., `latest`)
  - or a **content digest** (SHA256)
- When pulling an image:
  1. The client retrieves the **manifest**
  2. It downloads the required **layers**
  3. Layers are verified using their **content hash**
- Layer reuse enables **efficient storage and fast distribution**.

ToC

1. Introduction

2. A Deeper Dive into Container Management and Docker Example
2.1 Overview
2.2 Runtime

3. OCI Specifications
3.1 OCI Runtime
3.2 OCI Image
3.3 OCI Distribution Specification

4. Conclusion

Conclusion

- Containers are built on a layered stack: `dockerd` / `containerd`, shims, and a low-level runtime such as `runc`.
- The OCI specifications separate **how to run** a container from **how to package** it, making runtimes and images interoperable.
- An OCI image is a content-addressed bundle made of a manifest, a configuration object, and reusable filesystem layers.
- This architecture makes containers portable and efficient, but it also shows that they remain a process-isolation mechanism built on the host kernel.
- Understanding these internals is key to seeing why stronger isolation mechanisms, such as **MicroVMs**, can address some of the limits of containers.