OS Containers – Container runtimes

Renaud Lachaize

Univ. Grenoble Alpes
M2 MoSIG & Phelma-SEOC
December 2025

Some reminders about previous lectures

The concept of OS containers:

- Unlike (system-level) virtual machines, **containers support virtualization at the level of the OS (kernel) interface** (rather than at the hardware interface).

- Roughly speaking, a container is akin to a **"process group"** in a traditional OS … yet with **more isolation guarantees** regarding security, performance and software configuration.

- Different (guest) containers running on the same machine **share the same underlying host kernel**. There are no guest kernels.

Some reminders [continued]

- We focus on Linux containers.
- **On Linux, the concept of containers is not a unified facility provided by the operating system.**
- The management of containers is achieved by **combining a set of independent facilities** provided by distinct Linux kernel mechanisms and user-level tools.
- We have already studied the main kernel-level mechanisms (such as namespaces and cgroups).

Container tools

The container ecosystem also includes **tools and facilities to simplify the management of container images** (i.e., the files to be included in the file system within a container).

- **Application packaging**
  - Management of executables, libraries, and configuration files
  - Management of version numbers and dependencies
  - Layered file system, allowing to define new images based on existing ones, in a simple and space-efficient way
- **Distribution and sharing of images**
  - Repository (“hub”) of existing images
  - Facilitated by the fact that most container images are lightweight (and layered)

Containers tools [continued]

In practice, the management of containers is addressed via a set of software tools, which encompass **different needs**:

- Building **container images**, managing images, sharing and downloading images
- Managing **container instances**, running containers

There exists several tools with roughly similar features, like Docker and Podman.

The above-mentioned tools are themselves based on a modular architecture, based on several building blocks, among which, so-called **“low-level runtimes”** and **“high-level runtimes”** (detailed later), which serve different purposes.

OCI: Open Container Initiative (1/2)

- The OCI is (since 2015) "*an open governance structure for the express purpose of creating open industry standards around container formats and runtimes"*.
- So far, 3 main specifications have been produced (image, runtime, distribution).
- **Image format specification**:
  - "Defines the requirements for an OCI Image (container image), which consists of a manifest, an optional image index, a set of filesystem layers, and a configuration."
- **Runtime specification**:
  - "Specifies the configuration, execution environment, and lifecycle of a container." In particular, "defines how to properly run a container *"filesystem bundle"* which fully adheres to the OCI Image Format Specification."
- **Distribution specification**:
  - "Defines an API protocol to facilitate and standardize the distribution of content. It was launched in April 2018 to standardize container image distribution around the specification for the Docker Registry HTTP API V2 protocol, which supports the pushing and pulling of container images."

OCI: Open Container Initiative (2/2)

- For more details:
  - I. Velichko. **What Is a Standard Container (2021 edition)**. https://iximiuz.com/en/posts/oci-containers/
  - B. Chen. **Open Container Initiative (OCI) Specifications.** 2019. https://alibaba-cloud.medium.com/open-container-initiative-oci-specifications-375b96658f55
  - A. Krajewska. **Container image formats under the hood.** 2020. https://snyk.io/blog/container-image-formats/
  - J. Webb. **Docker and the OCI container ecosystem.** LWN. 2022. https://lwn.net/Articles/902049/

  - OCI image spec: https://github.com/opencontainers/image-spec/
  - OCI runtime spec: https://github.com/opencontainers/runtime-spec
  - OCI distribution spec: https://github.com/opencontainers/distribution-spec

OCI runtime specification

- “A Standard Container is an environment for executing processes with configurable isolation and resource limitations.”

- “[To] define a unit of software delivery ... The **goal of a Standard Container is to encapsulate a software component and all its dependencies** in a format that is self-describing and portable, so that **any compliant runtime can run it** without extra dependencies, regardless of the underlying machine and the contents of the container.”

- “[Containers] can be created, started, and stopped using standard container tools; copied and snapshotted using standard filesystem tools; and downloaded and uploaded using standard network tools.”

OCI runtime specification [continued]

Runtime and lifecycle:

- (See https://github.com/opencontainers/runtime-spec/blob/main/runtime.md)
- **A container can be in the following states:**
  - Creating
  - Created
  - Running
  - Stopped
- **The runtime must support the following operations:**
  - QueryState
  - Create
  - Start
  - Kill (signal)
  - Delete
- **Hooks:**
  - Allow for additional actions to be taken before or after each operation
  - https://github.com/opencontainers/runtime-spec/blob/main/config.md#posix-platform-hooks

OCI runtime specification [continued]

- **The OCI runtime specification defines the expected format for a "runtime bundle":**
  - Configuration file (config.json)
  - Root file system tree
- **Such a bundle must be:**
  - Prepared by a high-level container runtime
  - Provided as input to a low-level container runtime

- "The definition of a bundle is only concerned with how a container, and its configuration data, are stored on a local filesystem so that it can be consumed by a compliant runtime."
- "A Standard Container bundle contains all the information needed to load and run a container."

- Details: https://github.com/opencontainers/runtime-spec/blob/main/bundle.md

OCI runtime specification [continued]

- **Bundle Configuration file (config.json)**
  - (See https://github.com/opencontainers/runtime-spec/blob/main/config.md)
  - “This configuration file contains metadata necessary to implement standard operations against the container. This includes the process to run, environment variables to inject, sandboxing features to use, etc.”
  - Contains different sections: common section + platform-specific section

OCI runtime specification [continued]

- **Bundle Configuration file (config.json) [continued]**
  - **Common aspects**
    - Provides information such as as OCI version, rootfs path, list of additional mount points (beyond rootfs) + mount options, information about main process (terminal/console settings, environment variables, args, hostname, domainname, …)
  - **Platform-specific aspects (here we focus on POSIX+Linux)**
    - Provides additional settings about the process (capabilities, scheduling policy, CPU affinity …)
    - UID/GID in the container namespace
    - Lifecycle hooks
    - Annotations (arbitrary/extensible metadata for the container)
    - Linux container configuration (https://github.com/opencontainers/runtime-spec/blob/main/config-linux.md)
      - Provides settings for namespaces (various types), devices, cgroups, seccomp, …
  - Note: Among the various sub-specifications for platform-specific "container" aspects (including FreeBSD, Windows, …), there is also one about "virtual machines". (https://github.com/opencontainers/runtime-spec/blob/main/config-vm.md)

“Low-level” container runtimes

- **Such a low-level runtime is typically:**
  - Focused on a small feature set
  - Compliant with the OCI “runtime specification”
  - Not intended to be used directly by end users … but instead to be leveraged by other tools such as “high-level” container runtimes
- **Roles:**
  - Receiving as input an image bundle
  - Interacting with the host OS kernel to create and set up a new container instance based on the bundle, using facilities such as namespaces and cgroups
  - Launching the execution of the desired application/process(es) inside the container

"Low-level" container runtimes [continued]

Typical design – Important remarks:

- Such a low-level runtime typically does not run as a daemon (unlike a "high-level" runtime).
- A new (low-level) runtime instance is created upon each creation of a new container instance.
- **A low-level container runtime does not necessarily run/exist continuously throughout the lifespan of its corresponding container**.
  - Actually, such a behavior may vary according to the chosen operating mode for the runtime.
  - For example, in the case of the "runc" runtime, "foreground" versus "detached" mode (https://github.com/opencontainers/runc/blob/201b06374548b64212f4ceb1529688d435e42899/docs/terminals.md)
- More precisely:
  - **The low-level runtime typically terminates once it has finished setting up the container and launched the first/main (application) process inside the container.**
  - **After that, it is up to higher-level tools to monitor the container and manage its lifecycle.** Typically, this role is played by a container shim/monitor and/or a "high-level" container runtime.

"Low-level" container runtimes [continued]

Some examples of low-level runtimes:

- **runc**
  - One of the first and most popular "low-level" container runtimes.
  - Initially developed by Docker and donated to the OCI.
  - Written in Golang.
  - https://github.com/opencontainers/runc

- **runhcs**
  - A fork of runc supporting native Windows containers on Windows machines
  - https://learn.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/containerd

"Low-level" container runtimes [continued]

Some examples of low-level runtimes: [continued]

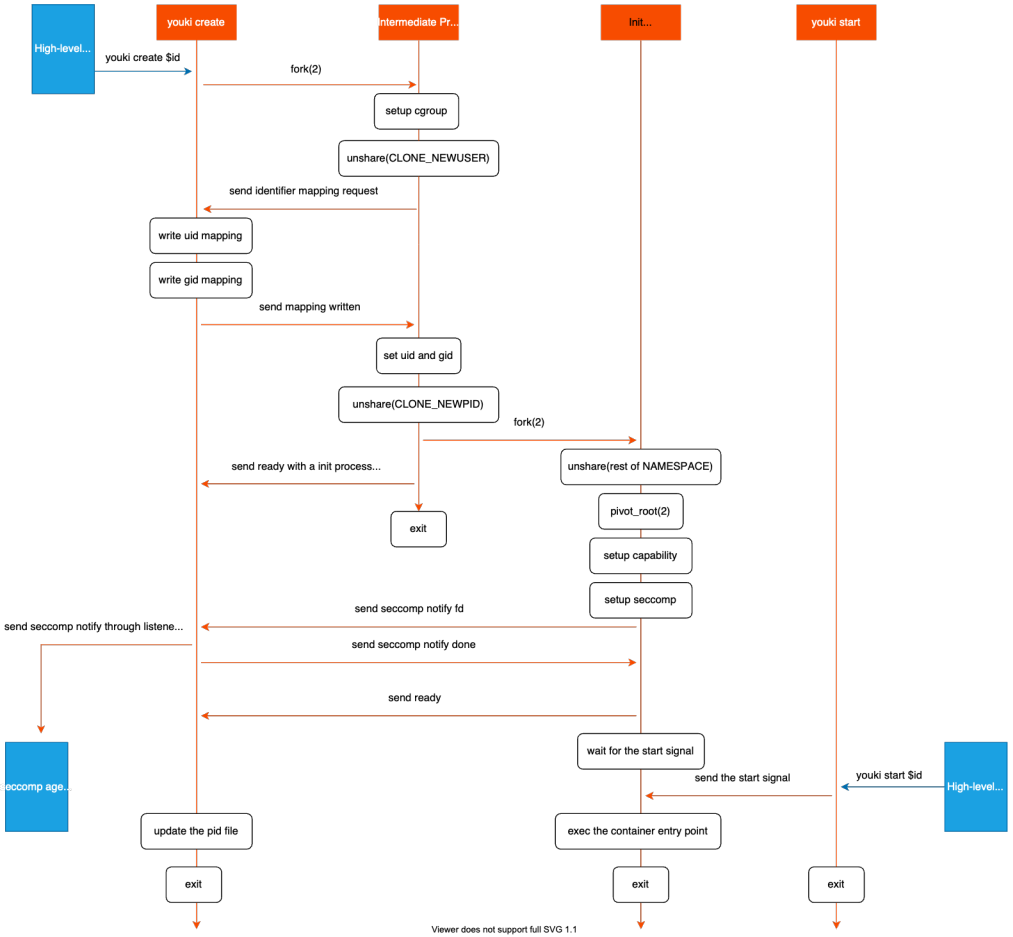
- **crun**
  - A lightweight and fast "low-level" runtime.
  - Written in C (better match than the Golang model for interactions with the Linux kernel)
  - Binary size ~ 300 kB (compared to ~15M for runc).
  - https://github.com/containers/crun
  - https://www.redhat.com/en/blog/introduction-crun
  - https://www.redhat.com/en/blog/speed-oci-containers
- **youki**
  - Motivation & characteristics quite similar to crun but is written in Rust.
  - https://github.com/youki-dev/youki

“Low-level” container runtimes [continued]

An illustration of the main setup steps performed by a typical low-level runtime (here Youki) leveraging “traditional” Linux kernel facilities:

- configures namespaces, cgroups, capabilities
- changes the root mount
- execs the container entry point

Source: Youki authors (https://github.com/youki-dev/)



"Low-level" container runtimes [continued]

Some examples of "non traditional" low-level runtimes:

- **Kata containers (a.k.a "Kata")**
  - Provides more security isolation than "standard" Linux containers …
  - … by **running each container as a distinct (system-level) virtual machine**
  - Leverages hardware-assisted virtualization and lightweight virtualization stacks ("micro VMs")
  - Supports multiple multiple virtualization stacks and multiple hardware architectures
  - On Linux, typically leverages the KVM hypervisor + the optimized Firecracker VMM
  - Remarks:
    - Unlike other kinds of containers, this design uses a per-container guest kernel.
    - Requires running on bare metal or having support for nested virtualization.
  - For more information:
    - https://katacontainers.io
    - https://firecracker-microvm.github.io

"Low-level" container runtimes [continued]

Some examples of "non traditional" low-level runtimes: [continued]

- **gVisor (a.k.a. runsc)**
  - Developed by Google
  - Provides additional security facilities to protect the host kernel from attacks.
  - Runs the container above a so-called **"application kernel"** (written in Golang) running in user mode and emulating (a substantial portion of) the Linux system call interface.
  - Supports several versions/implementations relying on distinct mechanisms for the interception of system calls:
    - Using the Linux KVM hypervisor (Note: this requires running on bare metal or having support for nested virtualization)
    - Using the systrap mechanism, itself based on Linux Seccomp
    - Using the ptrace facility (ubiquitously supported but slower)
  - https://gvisor.dev

"Low-level" container runtimes [continued]

Some examples of "non traditional" low-level runtimes: [continued]

- **runWasi**
  - Runs a container inside a WebAssembly (a.k.a. Wasm) sandbox/runtime
  - Implemented as a containerd shim, i.e., acting as a bridge between a high-level container runtime and a Wasm runtime.
  - Wasm code is portable code format, executed by a low-level runtime (a virtual-stack based machine, akin to the Java Virtual Machine).
  - WASI (WebAssembly System Interface) is an interface (API and ABI), intended to be portable and to provide POSIX-like operations (such as file I/O).
  - This topic will not be studied in depth in this lecture.
  - https://runwasi.dev
  - https://www.nigelpoulton.com/post/webassembly-and-containerd-how-it-works

"High-level" container runtimes

- Sometimes also named "**container engines**" or "**container managers**".
- **Such a high-level runtime typically:**
  - **Runs as a daemon**
  - **Relies on a low-level runtime** (+ shim layer, detailed later) for the creation and execution of containers
  - **Receives requests issued via humans or higher-levels tools/middleware**, using a CLI (command-line interface) or a network API
- **Main roles:**
  - Retrieving/downloading container images and unpacking them
  - Preparing the image bundle format to be provided to the low-level runtime
  - Possibly also managing some (lower-level but cross-container) aspects.
    - For example: setting up and managing the network configuration of containers
  - Managing the lifecyle of containers (especially post launch)

containerd

- One of the most popular high-level runtimes
- Initially developed by Docker (the company)
  - When the initial (monolithic) docker implementation was refactored into modular components
  - Still used today within the docker tool (and also in several other container-oriented tools)
- Runs as a daemon
- Provides CLI client applications (“ctr”, “nerdctl”), and a network API
- Is focused on running containers. Does not support other features such as building container images.

containerd [continued]

- To better understand the operations/commands supported by the API of containerd, it is recommended to take a look at the documentations/tutorials about the ctr CLI client.

- See for example:
  - https://hexshift.medium.com/mastering-ctr-practical-usage-of-containerds-native-cli-783706aa1153
  - https://labs.iximiuz.com/courses/containerd-cli/
  - https://www.mankier.com/8/ctr#

Container shims

- Sometimes also called **“container monitors”**.

- A **container shim** is typically **a small code layer run within a specific/dedicated process** to **facilitate the design and operation of a software stack for container management**.

- It essentially **acts as a bridge** between several components:
  - (1) a high-level container runtime (a.k.a. **“container manager”**)
  - (2) a low-level runtime instance + its associated container instance

Container shims [continued]

- There is typically **one container shim instance** per **container instance**.
  - But there are some exceptions (for example, in the case of Kubernetes pods).
- **A container shim:**
  - **is a long-lived process, whose lifespan is ~aligned with the one its associated container instance**. (For this reason, a shim is sometimes referred to as a "daemon process" but this terminology can be confusing.)
  - is tightly bound to the processes that it monitors: the low-level container runtime + the process(es) running inside the container
  - **is running detached from the high-level container "manager" process**
  - **mediates all the interactions between the manager and the container**

Container shims [continued]

- **Main roles of a container shim:**
  - **Abstracting away the machinery** of low-level container runtimes
  - **Acting as a direct parent of the running container** (once the low-level container runtime has exited after the container launch), using the Linux "subreaper" facility (https://iximiuz.com/en/posts/dealing-with-processes-termination-in-Linux/)
  - **Intercepting the container's stdin/stdout/stderr streams** and redirecting them (e.g., to log files)
  - **Reporting the container creation status** to the higher-level container runtime
  - **Reporting container termination info** (exit code) to the high-level container runtime
  - **Performing some cleanup** upon container termination
  - **Supporting dynamic attachment to a container** (e.g., for features such as "docker attach"), including the aspects related to pseudo-terminals
  - Dealing with situations where some of the higher-level components (e.g., the container manager) temporarily disappear (e.g., due to a restart or crash)

Container shims [continued]

- For some additional details, see:
  - https://iximiuz.com/en/posts/implementing-container-runtime-shim/
  - And more generally, the whole articles series: https://iximiuz.com/en/series/implementing-container-manager/

- **Examples:**
  - **containerd runtime shims** (for example, containerd-shim-runc-v2, a bridge between containerd and runc), based on the **containerd-shim API**: https://github.com/containerd/containerd/blob/main/core/runtime/v2/README.md)
  - **conmon** (used by Podman and CRI-O): https://github.com/containers/conmon

Container shims [continued]

- **For more technical details about containerd shims:**
  - Overview: https://container42.com/2022/01/10/shim-shiminey-shim-shiminey/
  - containerd Shim RPC API v2: https://github.com/containerd/containerd/blob/v1.5.8/runtime/v2/task/shim.proto
  - Code of the containerd-shim-runc-v2 shim: https://github.com/containerd/containerd/blob/v1.5.8/runtime/v2/runc/v2/service.go

Docker

- **Docker is a feature-rich and still very popular container management tool.**
- Since Docker 1.11 (2016), the architecture is modular.
- **The docker engine is a daemon (dockerd), which internally relies on containerd, controlled via a gRPC API.**
- **The docker command-line application (CLI):**
  - Acts a client application that interacts with the docker daemon via a local (Unix domain) socket, using a ~REST API (some operations are not RESTful).
  - **Warning: Having access to that socket from a process (running inside or outside a container) grants ~ root privileges on the machine!**
- The dockerd daemon handles authentication, networking and storage aspects.
- containerd manages the container images + the lifecycle of containers.
- As we have seen before, containerd itself relies on a shim + a low-level runtime such as runc.

Global architecture wrap-up: Docker case study

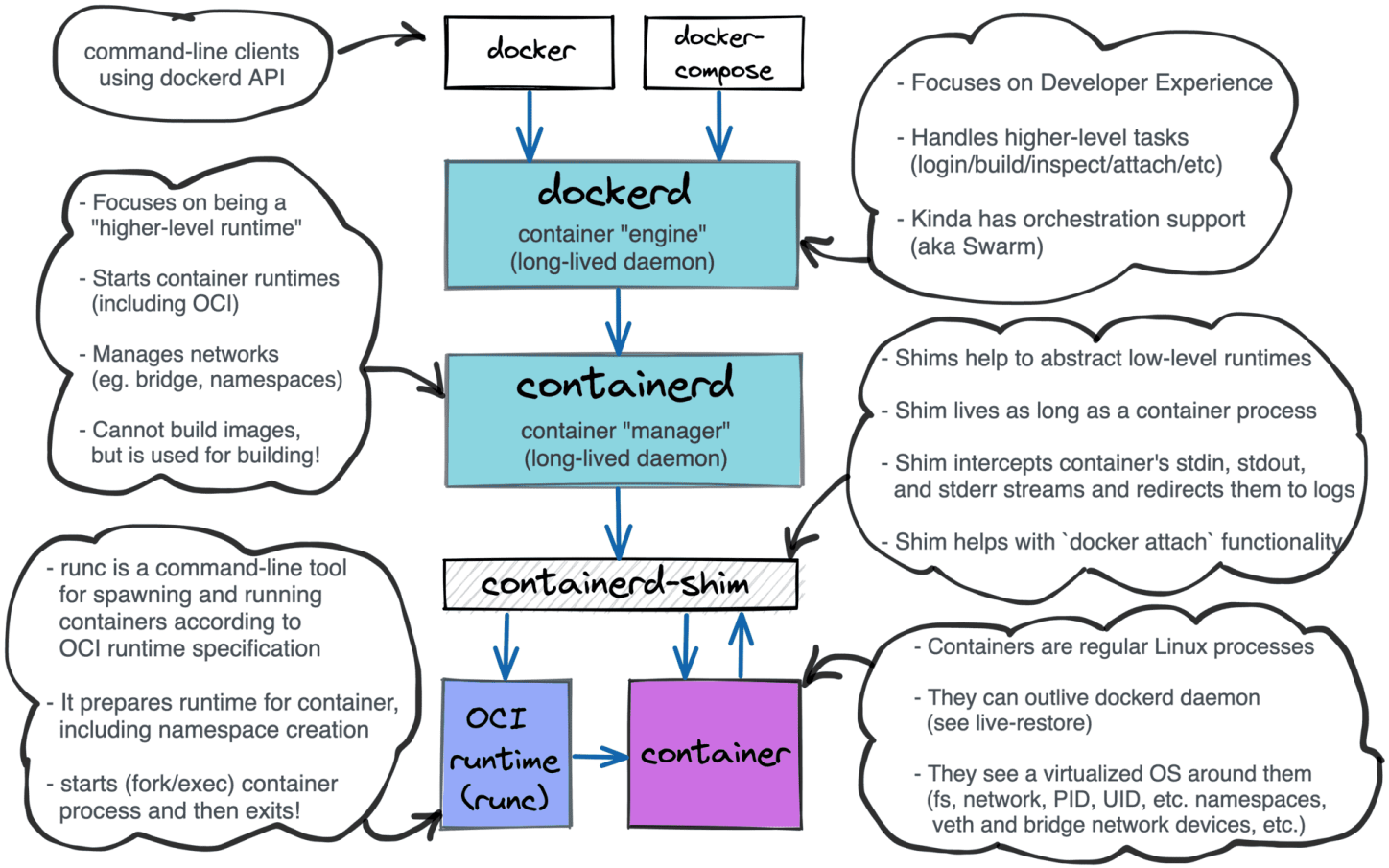


Figure by Ivan Velichko

Source: https://iximiuz.com/en/series/implementing-container-manager/

Rootless containers

- In the early days, running containers on Linux required having (more or less directly) root privileges (“rootful containers”).
- Progressively, new facilities (such as user namespaces) have been introduced in the Linux kernel to enable “rootless” operation: i.e., launching a container without having root privileges.
- Today:
  - Docker still uses a rootful mode by default (but also supports rootless mode).
  - Podman uses rootless mode by default.
  - In some cases, using rootless containers can still raise some difficulties/issues (e.g., regarding ease of use/setup and performance.)
  - Also, it is not always straightforward/possible to reuse/adapt rootful configurations and images to a rootless setup.
  - Running a container with access to a (real) network privileged port (i.e., port numbers < 1024) requires root privileges.

Rootless containers [continued]

- **Potential security advantages of rootless operation:**
  - More flexible management of users w.r.t. container tools (principle of least privilege)
  - Containerized applications/processes run without superuser privileges w.r.t. host/initial namespaces.
    - This allows limiting the impact of attacks if a malicious/compromised process manages to escape from a container.
    - Note that (as we have seen when studying user namespaces), this does not prevent from granting superuser privileges to the processes within the container w.r.t. to the logical resources owned by this container.
  - Containerized applications can be more isolated w.r.t. each other (if they are associated to different user IDs).

- Note: In some situations, there are also arguments against rootless operation (not detailed here).

Rootless containers [continued]

- **Container networking**
  - In rootful mode, a typical setup (convenient, flexible, efficient) for container networking relies on using virtual ethernet devices (see `man 4 veth` and `man 7 network_namespaces`) + a network bridge.
  - The above setup is not possible in rootless mode. An alternative setup consists in using a user-level networking stack (less efficient than the kernel stack), such as slirp4netns or passt/pasta.

- **Rootless images**
  - Another, complementary approach in terms of security precautions, consists in using “rootless images”, i.e., images configured in such a way that the processes running inside the container do not requires superuser privileges.

Podman

- **Podman essentially target the same use cases as Docker** (i.e., a feature-rich local container management tool, mainly aimed at developer and administrators).
- **Like Docker, Podman:**
  - Can be used to create, launch and manage containers
  - Consumes dockerfiles, docker (CLI) commands, docker REST API requests (Podman also introduces its own commands/files for its non-docker features)
  - Supports full management of container lifecycle
  - Supports management of container networking
- **Unlike Docker, Podman:**
  - Has been designed from the start with a **daemon-less architecture** and **built-in support for rootless containers**
  - **Does not rely on a high-level container runtime** (such as containerd) and instead directly interacts with a low-level runtime such as runc/crun
  - **Supports the concept of "pod" (~ groups of containers that share resources and are managed together), even without Kubernetes.**
  - **Does not provide image creation/building features. These features are provided by a separate tool (Buildah).**

Podman

- **Podman manages pods, but :**
  - Defines this notion in a flexible manner that is compatible with the K8s specifications … yet independant from them
  - It does not require K8s components and does support higher-level K8s concepts/controllers (such as Deployments, ReplicaSet, StatefulSet, etc.)
  - It does not itself use/support the CRI
  - It can also work with containers outside of pods

- For more information:
  - https://podman.io
  - https://github.com/containers/podman
  - https://www.redhat.com/en/blog/5-reasons-choose-podman-2025

Podman

- **Rootless operation**
  - Podman can be run as a normal user U (without requiring a setuid executable file).
  - In such a rootless setup, Podman leverages (Linux) user namespaces, in order to let U obtain superuser privileges inside containers but not outside of them (i.e., in the initial/host namespaces).
  - Rootless networking requires using a user-level networking stack (such as slirp4netns or pasta).
  - Almost all the normal Podman features are supported but the current implementation still has some limitations (https://github.com/containers/podman/blob/main/rootless.md)
  - For more information, see: What happens behind the scenes of a rottless podman container (https://www.redhat.com/en/blog/behind-scenes-podman)

Types of containers

- **Old containers (pre-Docker era) aimed at encapsulating a complete operating system:**
  - Including an init process but also various daemons (sshd, syslogd, etc.)
  - And possibly also running several (cooperating) applications within the same container (e.g., a Web application stack)
- **Docker philosophy:**
  - Run only one service/application (i.e., quite often just a single process) per container
  - Generally, no daemons (in particular, no “init” process – i.e., the initial application process owns PID 1)
- **Today:**
  - Some aspects of the Docker philosophy remain popular (lightweight containers, each with only one service/application). However, **there are various designs choices regarding the inclusion of a dedicated “Init” process and/or other daemons.**
  - Running without a real “init” process inside the container main raise problems (e.g. regarding the management of signals and process termination) so some containers include a simple “init” process, such as DumbInit (https://github.com/Yelp/dumb-init).
  - Some low-level runtimes enable Docker container to run a complete virtual server, featuring a complex “init” process such as system and/or complex services such as Docker or Kubernetes. See for example Sysbox (https://github.com/nestybox/sysbox).

Container orchestration

- A **container orchestrator** is:
  - **a software system** in charge of **simplifying** (through automation) **the management of a fleet of container-based applications on a cluster** of (virtual or physical) machines.

- **Examples:**
  - Docker Swarm
  - Kubernetes (more popular, more feature-rich, but also more complex)

- In the remainder of this discussion, we will focus on Kubernetes.

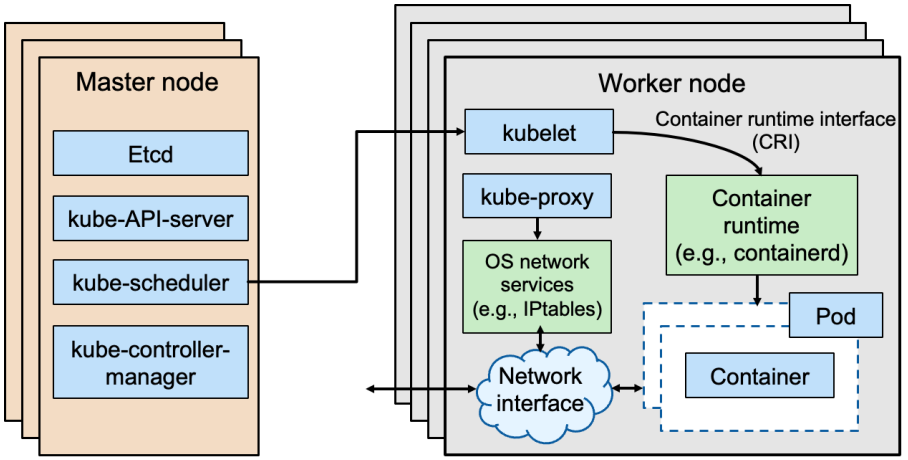
Container orchestration [continued]

- The **main duties** of a container orchestrator include:
  - The **provisioning** and **deployment** of containers
  - The setup of the **network configuration** of the containers
  - The **placement decisions** for the containers on the cluster nodes
  - **Health monitoring** (for containers and nodes) and appropriate reactions when needed
  - **Load balancing** between containers
  - **Autoscaling** decisions (mostly for horizontal scaling but also possibly for vertical scaling)

- **In this lecture, we will only consider the aspects directly related to the execution of containers on the machines of the cluster.**

Container runtime interface (CRI)

- CRI is another specification about container runtimes, which **has a different scope** than the (previously discussed) OCI runtime specification.
- **CRI is an interface specifically defined for the internal design of the Kubernetes container orchestration system.**
- Kubernetes (a.k.a. “K8s”) aims at managing distributed applications (based on containers) deployed on a cluster of “worker” nodes.
  - On each worker node, a specific K8s component named “kubelet” receives external commands issued by the K8s “control plane”.
  - The kubelet interacts with a local container (high-level) runtime.
  - **CRI corresponds to the specification of the interface between the kubelet and the container runtime.**



Container runtime interface (CRI) [continued]

Position of the CRI specification w.r.t. the OCI runtime specification:

- **The OCI runtime specification** defines the configuration, execution environment, and lifecycle of a container on a local machine. In particular, it defines how to properly run a container *"filesystem bundle"* which fully adheres to the OCI Image Format Specification.
- **The container runtime interface (CRI) is an API for container runtimes to integrate with the kubelet component on each compute/worker node.**
- An implementation of the Kubernetes CRI typically leverages OCI-compliant runtimes.
- A CRI-compliant runtime is aimed at providing the functionality required to manage containers on the cluster nodes a dynamic cloud environment.

Container runtime interface (CRI) [continued]

In K8s, application deployments are based on “pods”

- **A pod is an abstraction encapsulating one or several containerized components.**
- A pod is the basic scheduling unit for Kubernetes.
- **All the containers within a pod are guaranteed to be deployed on the same machine.**
- **Pod share resources, among which:**
  - **Networking:**
    - Containers within the same pod share the same network namespace.
    - Each pod is assigned a unique IP address.
    - Containers within the same pod can communicate using localhost.
  - **Storage:**
    - Containers within a pod have independent file systems … but a pod can specify a set of shared storage volumes that can be accessed by all the containers.
  - **Cgroups**

Container runtime interface (CRI) [continued]

- **The CRI defines several remote procedure calls (RPCs based on gRPC protocol) and message types.**

- **Roughly speaking, The CRI interface can be thought of as a gRPC interface providing:**
  - a subset of the features necessary in a general-purpose container engine
  - + support for some specific features, such as:
    - "Pods" as a first-class abstraction
    - Container checkpoints
    - Reporting of pod metrics/statistics

- **A CRI-compliant runtime must support the following features:**
  - Image management (including downloads from an image registry)
  - Managing individual containers
  - Managing pods

- **For more details:**
  - Overview: https://kubernetes.io/docs/concepts/containers/cri/
  - Specification: https://github.com/kubernetes/cri-api/blob/master/pkg/apis/runtime/v1/api.proto

Container runtime interface (CRI) [continued]

- **crictl:**
  - The “crictl” utility **allows sending CRI requests directly from the command line**. This is useful to test/debug a CRI implementation without to set up K8s (i.e., without deploying a full-blown K8s cluster or even a kubelet).
  - Note that crictl is not designed for (capable of) managing containers outside of pods because K8s has no concept of container outside of a pod.
  - **For a full list of commands**, see: https://github.com/kubernetes-sigs/cri-tools/blob/master/docs/crictl.md

Container runtime interface (CRI) [continued]

Some examples of the main operations covered by CRI:

- **Images:**
  - Pull/remove image
  - Check image status/info
  - List images
  - …
- **Pods:**
  - Create/stop/delete pod
  - Display status of pod(s)
  - Forward local port to a pod
  - List pod statistics
  - …
- **Containers:**
  - Create/delete container (inside a pod)
  - Start/stop container(s)
  - Checkpoint container(s)
  - Display status of container(s)
  - Fetch the logs of a container
  - List container(s) resource usage statistics
  - Attach to a running container
  - Execute a command inside a running container
  - …

Note: CRI does not include a function for restarting a container once it is stopped.

Container runtime interface (CRI) [continued]

What are the typical engines used (for running containers) in Kubernetes setups?

- **Docker [warning: deprecated]**
  - In the early years of Kubernetes, Docker was the main container engine used in K8s setups.
  - A component named Dockershim provided a bridge between the CRI (gRPC requests) and Docker (Docker REST API).
  - Docker was eventually deprecated as a CRI engine in 2020 due to various reasons
    - https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/
    - https://www.redhat.com/en/blog/kubernetes-is-removing-docker-support-kubernetes-is-not-removing-docker-support
- **containerd**
  - Using a CRI plugin, providing a gRPC (CRI API) server on top of containerd
- **CRI-O**
  - See next slide
- **Notes:**
  - As discussed before, containerd (and also CRI-O) can be plugged above diverse implementation of low-level runtimes leveraging either traditional container facilities (e.g., runc, crun, youki) or other isolation techniques (e.g., gVisor, Kata).
  - Podman is not designed to act as a CRI-compliant engine.

CRI-O

- Developed/sponsored by Red Hat and other companies.
- As the name suggests, CRI-O is a CRI-compliant, Kubernetes-specific high-level container runtime.
  - “The name derives from CRI plus Open Container Initiative (OCI), because CRI-O is strictly focused on OCI-compliant runtimes and container images.”
  - “The scope for CRI-O is to work with Kubernetes, to manage and run OCI containers. It’s not meant as a developer-facing tool.”
  - “Building images, for example, is out of scope for CRI-O and that’s left to tools like Docker’s build command or Buildah.”
  - “While CRI-O does include a command line interface (CLI), it’s provided mainly for testing CRI-O and not really as a method for managing containers in a production environment. ”

CRI-O [continued]

- Handles images, storage and networking management.
- Relies on conmon for the shim layer
- Can use various low-level OCI runtimes
- CRI-O is based on the same libraries (containers/storage and containers/image) as Skopeo, Buildah, and Podman and can be used in conjunction with these tools
- Compared to containerd (focused on extensibility), CRI-O is more focused on simplicity
- **For more information:**
  - https://cri-o.io
  - https://www.redhat.com/en/blog/introducing-cri-o-10
  - https://github.com/cri-o/cri-o/blob/main/awesome.md

Container runtime interface (CRI) [continued]

The concept of "pod sandbox":

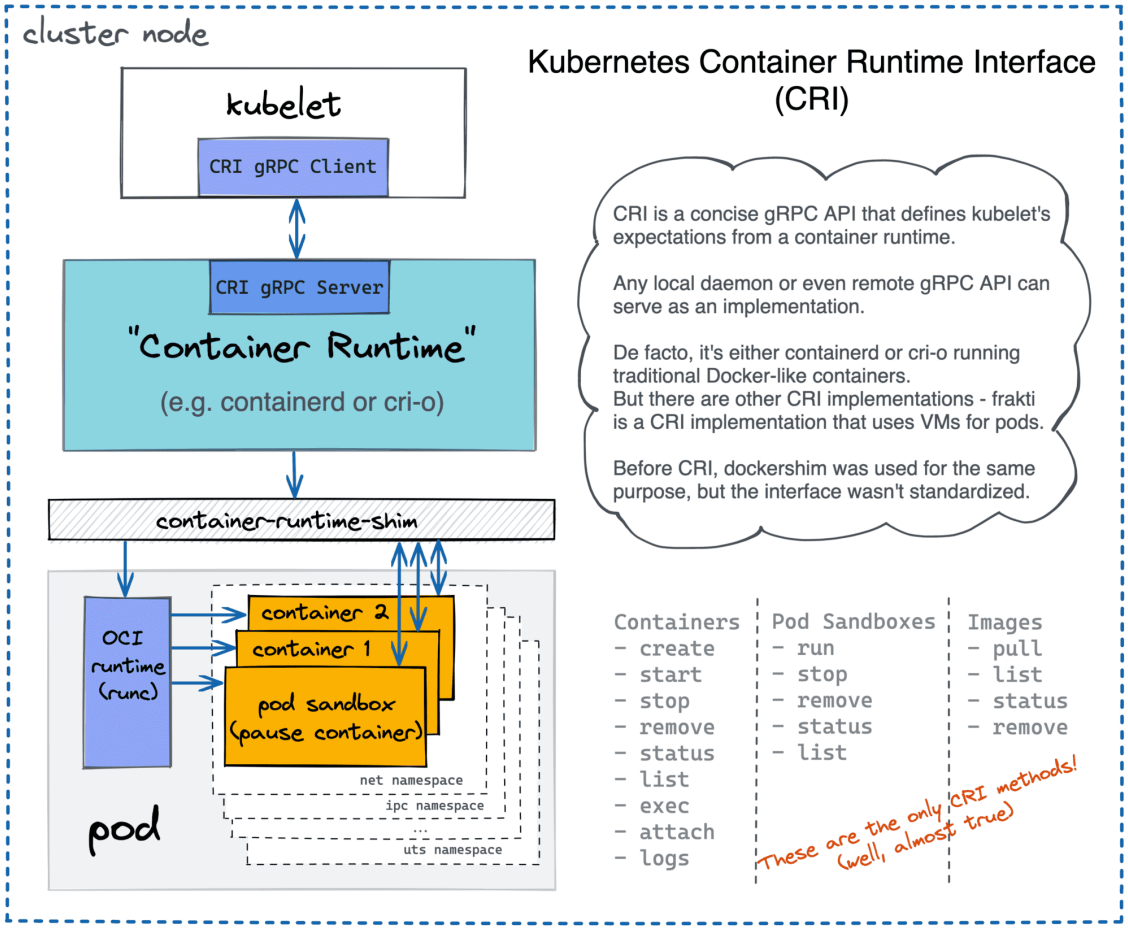
- (Warning: This name and concept are somewhat confusing.)
- **According to the CRI designers:**
  - "A Pod is composed of a group of application containers in an isolated environment with resource constraints. In CRI, this environment is called PodSandbox. We intentionally leave some room for the container runtimes to interpret the PodSandbox differently based on how they operate internally. For hypervisor-based runtimes, PodSandbox might represent a virtual machine. For others, such as ~~Docker~~ containerd, it might be Linux namespaces."
  - "Before starting a pod, kubelet calls RuntimeService.RunPodSandbox to create the environment. This includes setting up networking for a pod (e.g., allocating an IP). Once the PodSandbox is active, individual containers can be created/started/stopped/removed independently."
  - Source: https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/
- Understanding some implementation details regarding the notion of PodSandbox provides some insights that are useful when monitoring/debugging K8s applications running on traditional containers (see next slide …)

Container runtime interface (CRI) [continued]

The concept of "pod sandbox": [continued]

- When using K8s configured with a runtime such as containerd or CRI-O:
  - We can sometimes observe error messages about the creation of a new PodSandbox, mentioning a container image whose name contains "pause".
  - Even when everything works correctly, we can notice that each worker node typically runs a large number of containers running a "pause" image, whose code does almost nothing (~ a loop invoking the "pause" system call).
- **What is the underlying reason for these "pause" containers?**
  - **When the CRI-compliant runtime creates a new pod, it systematically creates a first/special container (i.e., the "pause" container) inside the pod's namespaces and cgroups.**
  - Essentially, **this container serves as a "parent container" for all the other containers (created later) within the pod**. It plays to main roles:
    - It acts as a placeholder that helps the other containers to join and share the same namespaces.
    - (When containers share the same PID namespace) It acts as the "init" process (PID=1) for all the containers in the pod and takes care of reaping zombie processes.
- For more information:
  - https://sklar.rocks/what-is-a-pod-sandbox/
  - https://www.ianlewis.org/en/almighty-pause-container
  - https://github.com/containerd/containerd/blob/main/docs/cri/architecture.md

Container runtime interface (CRI) [continued]



Source: Ivan Velichko. https://iximiuz.com/en/posts/conman-the-container-manager-inception/

References & Acknowledgments

- Scott McCarty. **A practical introduction to container terminology.** Red Hat developer blog. 2018. https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction#

- Ian Lewis. **Article series on Container runtimes (Parts 1- 4) .** 2017-2019.
  - https://www.ianlewis.org/en/container-runtimes-part-1-introduction-container-r
  - https://www.ianlewis.org/en/container-runtimes-part-2-anatomy-low-level-contai
  - https://www.ianlewis.org/en/container-runtimes-part-3-high-level-runtimes
  - https://www.ianlewis.org/en/container-runtimes-part-4-kubernetes-container-run

- Jerôme Petazzoni. **Introduction to Docker and Containers (self-paced tutorial)**. [See especially Parts 8 and 9]. Date unknown. https://container.training/intro-selfpaced.yml.html

References & Acknowledgments [continued]

- E. Baker. **A comprehensive container runtime comparison**. 2020. https://www.capitalone.com/tech/cloud/container-runtime/
- I. Velichko. **What Is a standard container (2021 edition)**. https://iximiuz.com/en/posts/oci-containers/
- I. Velichko. **Implementing container manager – Learning series**. 2020. https://iximiuz.com/en/series/implementing-container-manager/
- I. Velichko. **Debunking container myths – Learning series**. 2022. https://iximiuz.com/en/series/debunking-container-myths/
- J. Webb. **Docker and the OCI container ecosystem**. 2022. https://lwn.net/Articles/902049/
- A.Suda. **The internals and the latest trends of container runtimes**. 2023. https://medium.com/nttlabs/the-internals-and-the-latest-trends-of-container-runtimes-2023-22aa111d7a93
- Daniel Walsh. **Podman in Action**. Manning. 2023. [eBook freely available from Red Hat]