Containerd From The Bottom Up

A reader-grade tour of containerd, runc, namespaces, cgroups, and the Linux primitives behind every running container.

Contents

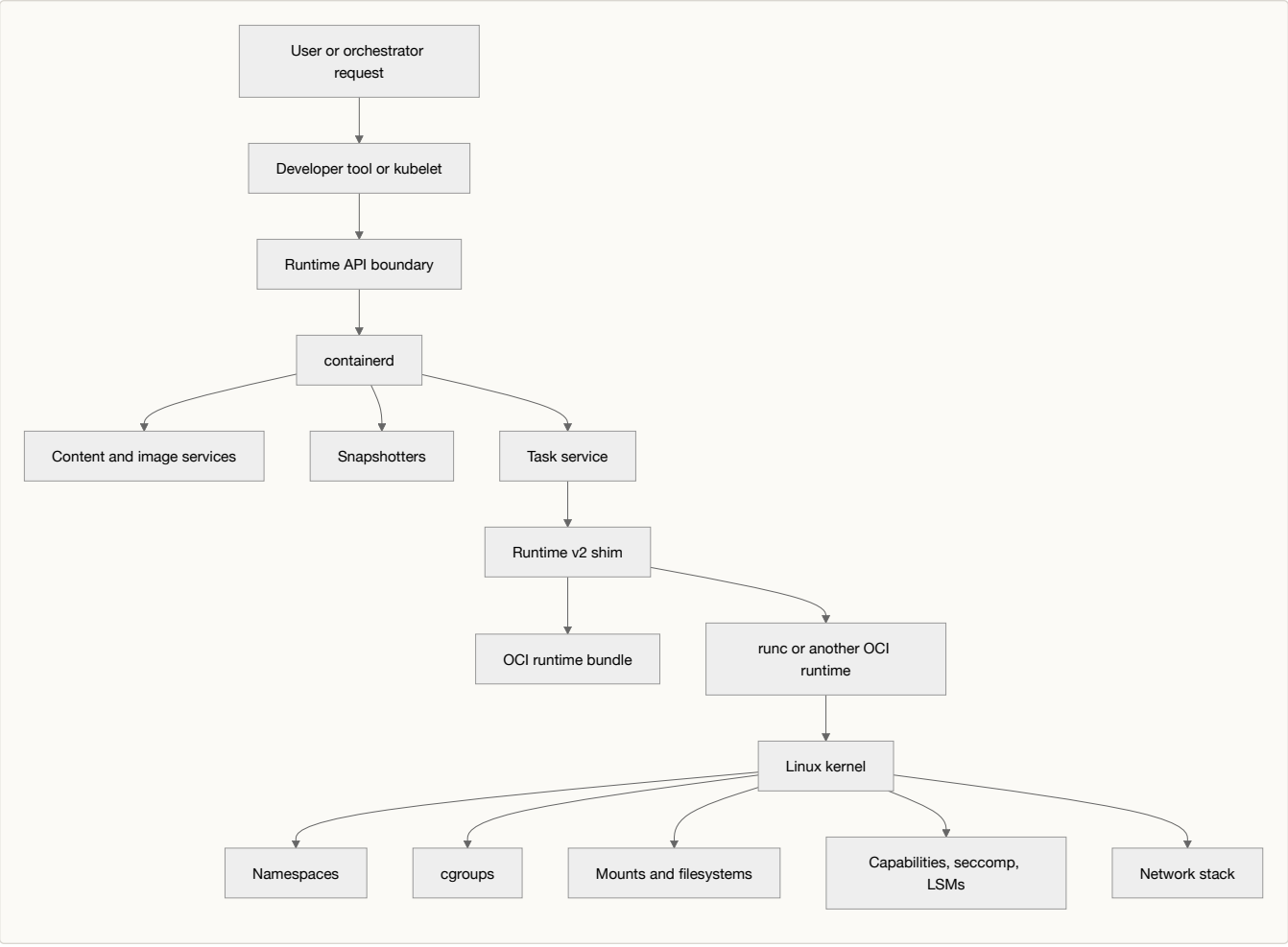
PART I — ORIENTATION

Chapter 1: The Container Stack Map

The word "container" gets used at every layer of the stack. A `docker run` command is a "container." A Kubernetes `Pod` "contains" containers. `runc` "creates a container." A `clone3(2)` call with namespace flags creates something the kernel does not call a container at all. The same word names a developer workflow, a Kubernetes resource, a runtime command, and a kernel construction — and people use the meanings interchangeably. That is how the same conversation can be technically correct on every line and still leave the room confused.

This book walks the runtime stack from the kernel up: namespaces and cgroups (control groups) and mount setup, the OCI (Open Container Initiative) runtime spec and `runc`, the runtime v2 shim, containerd's content store and snapshotters and task service, the Container Runtime Interface (CRI), and the Kubernetes objects that finally land on those primitives. The map below is what the rest of the chapters fill in.

A container system turns a request — typed by a developer or issued by an orchestrator — into a Linux process running with a configured environment. Along the way image references become local content, image layers become a mountable filesystem, runtime metadata becomes an OCI spec, and that spec becomes kernel state: namespaces, cgroups, mounts, security policy, and a network stack.



Docker, Kubernetes, `nerdctl`, `ctr`, Podman, and direct containerd clients all enter the stack at different layers. What stays consistent is the delegation pattern: high-level tools express intent, lower layers produce content and runtime state, and the kernel enforces the result.

A Short History

Containers were not designed; they accreted over four decades of Unix work. The modern stack — kernel primitives at the bottom, an OCI runtime, a per-container shim, a daemon, an orchestrator API, and developer tooling at the top — is a record of which problem each generation solved and which it deferred to the next.

Filesystem Isolation: 1979 To 2005

Unix `chroot(2)` shipped in V7 in 1979. It changes the apparent root directory of a process so pathname resolution starts somewhere other than `/`. That is a useful primitive — it lets a sandbox or a build environment see a smaller filesystem — and it is not a container. `chroot` does not isolate process IDs, networking, IPC, users, or privileges. Root inside a `chroot` is root on the host, and a root process can break out of one without much effort.

FreeBSD jails (FreeBSD 4.0, March 2000) extended the idea into something closer to OS-level virtualization. A jail is `chroot` plus its own hostname, IP-binding constraints, restricted process visibility, and a privilege model that downgrades a jailed root. Solaris Zones (Solaris 10, February 2005) went further: separate process trees, separate networking stacks (with "exclusive-IP" zones added in 2007), and separate user spaces, all sharing one kernel. Both systems demonstrated, several years before Linux containers were practical, that "many isolated user spaces on one kernel" was a real production model.

Linux took longer. Through the 2000s the kernel grew the same set of capabilities one feature at a time, and the userland followed.

The Linux Kernel Catches Up: 2002 To 2020

Linux containers did not arrive as a single feature. The kernel grew the parts list across more than a decade:

Feature	Kernel	Year
POSIX capabilities	2.2	1999
Mount namespace	2.4.19	2002
SELinux	2.6.0	2003
seccomp (mode 1)	2.6.12	2005
UTS and IPC namespaces	2.6.19	2006
Process containers (cgroups)	2.6.24	2008
PID namespace	2.6.24	2008
Network namespace	2.6.24 onward	2008–2009
AppArmor	2.6.36	2010
seccomp-bpf	3.5	2012
User namespace	3.8	2013
OverlayFS	3.18	2014
cgroup v2	4.5	2016
cgroup namespace	4.6	2016
Time namespace	5.6	2020

A short gloss on each name in the table:

- **POSIX capabilities** — root-equivalent authority broken into ~40 individually grantable units (`CAP_NET_ADMIN` , `CAP_SYS_ADMIN` , etc.).

- **Namespaces** — kernel mechanism that gives a process a private view of one global resource (process IDs, mounts, network stack, hostname, IPC, user IDs, cgroup tree, time offsets).
- **cgroups (control groups)** — group processes into a hierarchy and account for or limit their CPU, memory, IO, and pids consumption.
- **SELinux and AppArmor** — Linux Security Modules (LSMs) that enforce mandatory access control. SELinux is label-based; AppArmor is path-based.
- **seccomp** — secure computing mode. The kernel feature that filters syscalls; `seccomp-bpf` lets userspace install a Berkeley Packet Filter program for the syscall decision.
- **OverlayFS** — kernel filesystem that stacks read-only layers under a writable upper layer; the standard backend for layered container images.

cgroups arrived under the name "process containers" — Google's Paul Menage and Rohit Seth proposed them in 2006, and they merged in 2.6.24 in 2008. The name was changed to avoid colliding with the rest of the kernel's "container" usage. (The collision lost anyway.)

Through this period, Linux had container parts but no consensus container. OpenVZ — an open-source OS-level virtualization system distributed as a separate kernel patch set — had been running production VPS hosting since 2005, but never made it to mainline. Google had been running Borg, its internal cluster manager, on cgroups since the late 2000s. Neither was something a developer could install on a laptop.

LXC, Docker, And libcontainer: 2008 To 2014

LXC — short for Linux Containers — was the first userspace toolkit to assemble Linux's accumulating namespace and cgroup features into a usable container model. LXC 0.1.0 shipped in August 2008. It drove the kernel directly, exposed a CLI (`lxc-create` , `lxc-start` , `lxc-attach`), and worked — but it left the workflow problems open. There was no standard image format, no layered build model, no registry, no portable way to "ship a container."

Docker (March 2013) closed the workflow gap. The original 2013 release used LXC underneath; what Docker added was the part above the kernel: a daemon (`dockerd`), a developer-facing CLI (`docker`), a layered image format with build instructions (`Dockerfile`), and a registry protocol (Docker Registry, later Distribution). Containers became commodity infrastructure within eighteen months of Docker's launch — not because the kernel changed, but because the workflow finally fit a developer's day.

Docker 0.9 (March 2014) replaced LXC with libcontainer, a Go library that drove the kernel directly through namespace and cgroup syscalls. The motivations were operational: independence from a separate userland project, the ability to manage namespaces and cgroups directly from Go, and a path to platform-independent execution drivers. libcontainer is the codebase that became `runc` .

CoreOS launched rkt (pronounced "rocket") in November 2014 with a different model: no central daemon, an `appc` (App Container) image format, and a process-per-invocation supervision shape. rkt did not survive as a runtime, but its existence pushed the ecosystem toward open, multi-vendor standards instead of Docker-defined ones.

Standards: OCI, 2015

The Open Container Initiative formed at DockerCon on June 22, 2015, under the Linux Foundation. The mandate was open specifications for container formats and runtimes; Docker donated runc (a brand-new extraction of libcontainer) as the reference implementation. OCI publishes three specifications, each versioned independently:

- **Runtime Specification** — what an OCI runtime consumes and how it behaves. Defines the OCI bundle format and the lifecycle commands.
- **Image Specification** — how images are packaged: manifests, image configs, layers, and digests.
- **Distribution Specification** — how images move through registries. Reached 1.0 in 2020 by formalizing Docker Registry HTTP API V2 as an open standard.

The three specs separate concerns the Docker daemon had bundled together: building, distributing, and running an image are now three contracts owned by three standards.

`runc` 1.0 took until June 2021. Real production deployments had been using pre-1.0 runc for years; the version number was an acknowledgement of stability, not a moment of arrival.

containerd Becomes Its Own Layer: 2015 To 2020

containerd started inside Docker in early 2015 as a refactor that pulled lifecycle and supervision out of `dockerd` and into a separate daemon. By December 2015 it was a public project; in March 2017 Docker donated it to the Cloud Native Computing Foundation (CNCF), the Linux Foundation sub-project that hosts Kubernetes; CNCF announced graduation on February 28, 2019.

The split mattered because Kubernetes — by then the dominant orchestrator — did not want to depend on Docker the product. containerd 1.0 (December 2017) gave Kubernetes a daemon focused on the work in the middle: pulling and storing image content, snapshot management, container metadata, task supervision, and runtime integration through shims. Docker remained the developer-facing product on top; containerd became reusable infrastructure underneath.

containerd's runtime v2 shim model (containerd 1.2, October 2018) is the boundary that lets a single daemon work with runc, crun (a C reimplementation of runc by Red Hat), gVisor (a Google userspace kernel that intercepts syscalls), Kata Containers (each container in a lightweight VM), Wasm (WebAssembly) runtimes, and Windows host process containers through one ttrpc API. ttrpc is a smaller-footprint variant of gRPC for local Unix-socket transport, designed for the per-container shim use case. Runtime v1 was deprecated in containerd 1.4 (September 2020); current installations are all v2.

Kubernetes And CRI: 2016 To 2022

Kubernetes shipped in 2014 carrying the vocabulary of Borg, Google's internal cluster manager that Kubernetes was modelled after — pods, controllers, schedulers — and an in-tree integration with Docker Engine. By late 2016 the in-tree integration had become a maintenance problem: every container-runtime change required a Kubernetes patch. The Container Runtime Interface (CRI) was the answer. CRI v1alpha shipped in Kubernetes 1.5 (December 2016) as a gRPC API kubelet calls, and any runtime that implements it can plug into Kubernetes.

CRI is also where the word "runtime" splits in two. From Kubernetes' point of view, a "container runtime" is anything that implements CRI: containerd (via its CRI plugin), CRI-O. From OCI's point of view, the "runtime" is what consumes an OCI bundle: runc, crun, youki, runsc, kata-runtime. Both meanings are correct; the rest of the book uses the precise term.

For several years kubelet talked to Docker through an in-tree adapter called dockershim, which presented Docker Engine as if it were a CRI runtime. dockershim was deprecated in Kubernetes 1.20 (December 2020) and removed in 1.24 (May 3, 2022). The change did not break Docker-built images — those are OCI images, identical to what any other tool produces — but it ended the special case where Kubernetes called Docker. Kubernetes nodes now talk to containerd or CRI-O directly.

Why The Stack Looks Like This

The current shape — kernel primitives, OCI runtime, runtime shim, containerd, CRI plugin, kubelet, Kubernetes API — is the residue of those decisions. Each layer marks a boundary that someone, at some point, had to redraw because the layer above wanted to be replaceable.

- The kernel stayed the kernel. Containers compose its primitives without changing them.
- OCI exists because Docker-defined formats were not the stable contract a multi-vendor ecosystem needed.
- runc exists because the kernel-driving code had to live in a small, language-agnostic binary that did not bundle a registry or a daemon.
- The runtime v2 shim exists because containerd needs to survive its own restarts without killing running workloads.
- containerd exists because Docker's product surface and Kubernetes' runtime needs are different problems.
- CRI exists because Kubernetes wanted to stop tracking individual runtime APIs.

That layering is what the rest of the book inspects. Most of the time the layers are invisible — a developer types `kubectl apply -f deploy.yaml` and a process starts somewhere — but when something breaks, "where in the stack" is the only question that matters.

"Runtime" Means Two Things

The CRI/OCI split appears at every layer of the stack from chapter 3 onward. It deserves its own definition table.

Phrase	Meaning	Examples
CRI runtime	A service kubelet calls over gRPC. Implements `RuntimeService` and `ImageService`.	containerd (via its CRI plugin), CRI-O
OCI runtime	A program that consumes an OCI bundle and produces a configured process.	runc, crun, youki, runsc, kata-runtime
containerd runtime plugin	containerd's configured runtime path; usually a runtime v2 shim binary.	`io.containerd.runc.v2`, `io.containerd.runhcs.v1`
Runtime v2 shim	The supervisor process between containerd and the OCI runtime.	`containerd-shim-runc-v2`, `containerd-shim-kata-v2`

A request from a Kubernetes pod walks all four. kubelet asks the CRI runtime (containerd), which delegates to a runtime v2 shim (`containerd-shim-runc-v2`), which calls the OCI runtime (`runc`).

The Layers, From The Top

Intent

A person types `docker run nginx`, or `nerdctl run nginx`, or `podman run nginx`, or applies a Kubernetes Deployment that eventually causes kubelet to ask for a pod. The interfaces differ; the intent is the same — run a process from an image with a particular filesystem, environment, network, and resource policy.

Developer-facing tools name containers, expose ports, attach logs, and hide the runtime machinery. Hiding it does not eliminate it; it means the tool is doing translation. `docker run nginx` becomes a registry pull, a snapshot prepare, a container record, a task create, a shim launch, a runc invocation, and a `clone3(2)` call. Everything from the second word onward is what the rest of the book is about.

Docker

Docker is a developer-facing product: a CLI (`docker`), a daemon (`dockerd`), and a set of workflows for images, containers, networks, and volumes. Since Docker 1.11 (April 2016), `dockerd` has not done its own container lifecycle work — it delegates to containerd, and containerd delegates to runc. A `docker run` command and "a container runtime" are two different things at two different layers; the daemon is the workflow tier, the runtime is what configures the kernel.

Docker is one entry point into the stack. `nerdctl` is a Docker-compatible CLI for containerd. Podman is a daemonless Docker-compatible CLI that drives runc directly through `conmon` (container monitor), a small per-container supervisor that plays the same role as containerd's shim. The entry points differ; the layers underneath are the same.

Kubernetes

Kubernetes does not exist to run a single container conveniently. It exists to manage desired state across many machines. A user declares that a workload should exist; controllers and the scheduler decide where it runs; kubelet on each node makes local runtime calls to create pods and containers.

kubelet talks to the runtime over CRI, a gRPC API defined in `k8s.io/cri-api`. The methods break into a `RuntimeService` (pod sandbox lifecycle, container lifecycle, exec, attach, port-forward, status, stats, logs) and an `ImageService` (list, status, pull, remove, filesystem usage). Anything kubelet wants the runtime to do crosses one of those two interfaces.

A pod is a Kubernetes-only concept; CRI invents a "pod sandbox" to represent it at the runtime layer. The sandbox holds the network namespace, runtime endpoint, labels, and (on Linux) a `pause` container that pins the namespaces while the pod's workload containers come and go. Chapter 13 walks the full sandbox lifecycle.

containerd

containerd is where image references become content, content becomes a snapshot, and a snapshot plus a spec becomes a running task — three boundaries the rest of the book follows.

It is a daemon with a graph of plugins: a content store for digest-addressed bytes, snapshotters for filesystem state, an image service for name-to-descriptor mapping, a container metadata store, a task service, runtime v2, a CRI plugin, an events service, and a leases service for protecting in-flight work from garbage collection. Clients reach each plugin through gRPC.

containerd also enforces a distinction that cuts through the rest of the book's vocabulary: a *container* is metadata — image, labels, snapshot key, runtime config, OCI spec — while a *task* is the live process derived from that metadata. A container can exist without a task. A task can exit while the container record stays. Untangling them is the first step toward understanding what the daemon is actually doing.

Images, Content, And Snapshots

When the user says "run nginx," nothing about that reference is yet usable. The reference is a name; the kernel needs a filesystem.

Producing one takes six steps:

1. resolve the reference to a manifest;
2. fetch the manifest, image config, and layer blobs from a registry;
3. store blobs by digest in the content store;
4. unpack the layers;
5. ask a snapshotter to prepare a mountable view, plus a writable layer for this container;
6. mount that view as the root for the process.

The content store and the snapshotter are deliberately separate concerns. The content store holds digest-addressed image data, immutable and shareable across containers and across images. The snapshotter turns layers into filesystem state. Image pulls can succeed even when unpacking fails; many containers can share one set of immutable layers; and garbage collection has to reason about content, snapshots, containers, and leases at once. Chapter 11 covers the whole pipeline.

Shims

In the runtime v2 model, containerd launches one shim process per container (or per pod sandbox); for the standard Linux runc path, the binary is `containerd-shim-runc-v2`. The shim owns runtime-specific behavior and the lifecycle of the actual container process: it invokes runc, manages the IO pipes, reaps the process, and reports the exit status back to containerd. On a host running one container, `pstree` shows the layering:

```
systemd
├── containerd
└── containerd-shim-runc-v2
    └── nginx        ← the workload
```

The shim is not a duplicate of containerd. It is the supervision boundary that lets containerd be restarted without killing running workloads — the shim keeps holding the workload's stdout, exit status, and signal path until the daemon reconnects. Runtime-specific code lives behind the shim v2 task API over ttrpc (a lightweight gRPC variant for local Unix-socket transport), so swapping runc for crun, gVisor, or Kata is a configuration change at the containerd layer, not a code change.

runc

runc is a low-level OCI runtime. Its job is to take an OCI bundle — a directory containing `config.json` and a root filesystem — and produce a configured process. It does not talk to registries, schedule pods, or host a developer workflow.

`config.json` describes everything the kernel needs to know: the process to run, environment and working directory, root filesystem path, mounts, namespaces to enter or create, cgroup settings, capabilities, seccomp filter, hooks, and annotations. runc translates that into Linux operations — `clone(2)` or `clone3(2)` with the right namespace flags, mount setup, cgroup writes,

credential and capability adjustments — and then `execve` s the configured process.

runc is one OCI runtime. crun is a C reimplementation maintained by Red Hat with lower memory and faster startup. youki is a Rust reimplementation. runsc (gVisor) intercepts syscalls in a userspace kernel for stronger isolation. kata-runtime runs each container inside a lightweight VM. All of them satisfy the same OCI bundle contract.

The Kernel

A container is built by configuring several Linux primitives at once:

- namespaces change what a process can see;
- cgroups account for and limit what it can use;
- mount setup determines what filesystem it sees as `/`;
- capabilities and seccomp reduce its authority;
- LSMs (AppArmor, SELinux) enforce mandatory access control;
- network namespaces and virtual devices route its traffic.

The kernel does not have a "container" type. It runs processes with credentials, mount tables, and namespace memberships — and the runtime configures those primitives so that, taken together, they behave like the abstraction the layers above promised.

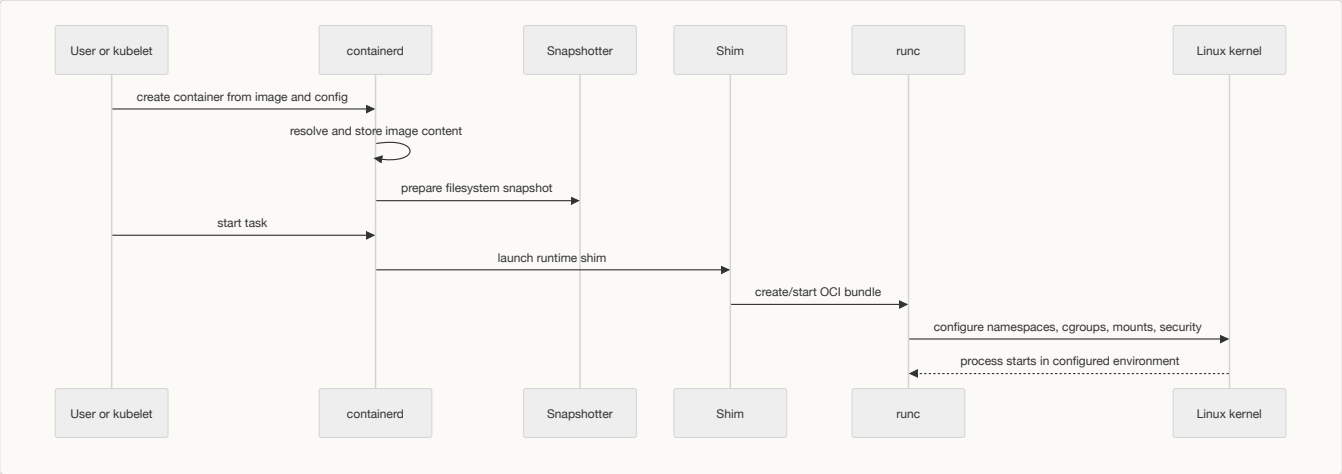
What "Container" Means At Each Layer

The same word names different objects at different layers. When two engineers disagree about containers, they are usually pointing at different rows of this table:

Layer	"A container" is...	The actual object
Kubernetes	An entry in a `Pod` spec.	A `Container` inside `PodSpec.containers`.
CRI	A runtime-level container, scoped to a pod sandbox.	A container ID returned by `RuntimeService.CreateContainer`.
containerd	Persistent metadata.	A row in the container metadata store: ID, image, snapshot key, runtime, OCI spec.
Runtime v2 / shim	A supervised task.	A task service ID with an attached shim process and runc state.
OCI	A bundle in the `created` or `running` state.	A directory with `config.json` plus a runtime state file.
Linux kernel	Nothing.	A process tree with namespace memberships, cgroup placement, mount table, credentials, and security policy.

Most "is X a container" arguments resolve once both speakers agree which row they mean.

Following One Request Down



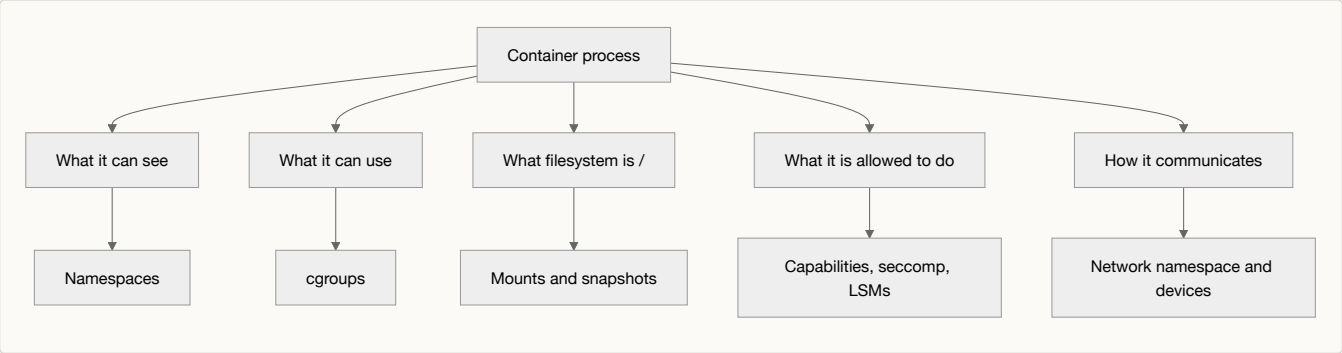
The exact calls vary by client, runtime configuration, and host. The rest of Part I expands two of the diagram's nouns: chapter 2 zooms in on the kernel-side question — what a container actually is — and chapter 3 covers the contracts that hold the stack together: the OCI runtime spec and the runtime v2 shim API.

Sources And Further Reading

- Docker overview: https://docs.docker.com/get-started/docker-overview/
- Docker 0.9 / libcontainer announcement: https://www.docker.com/blog/docker-0-9-introducing-execution-drivers-and-libcontainer/
- Open Container Initiative: https://opencontainers.org/
- OCI Runtime Specification: https://github.com/opencontainers/runtime-spec
- OCI Image Specification: https://github.com/opencontainers/image-spec
- OCI Distribution Specification: https://github.com/opencontainers/distribution-spec
- containerd docs: https://containerd.io/docs/main/
- containerd graduation announcement: https://www.cncf.io/announcements/2019/02/28/cncf-announces-containerd-graduation/
- containerd runtime v2: https://github.com/containerd/containerd/blob/main/docs/runtime-v2.md
- Kubernetes container runtimes: https://kubernetes.io/docs/setup/production-environment/container-runtimes/
- Kubernetes CRI introduction: https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/
- Kubernetes dockershim removal FAQ: https://kubernetes.io/blog/2022/02/17/dockershim-faq/
- runc README: https://github.com/opencontainers/runc
- LXC project: https://linuxcontainers.org/lxc/introduction/
- FreeBSD jail(8): https://man.freebsd.org/cgi/man.cgi?query=jail&sektion=8
- Solaris Zones introduction: https://docs.oracle.com/cd/E19044-01/sol.containers/817-1592/zones.intro-1/index.html

Chapter 2: What A Container Actually Is

A Linux container is a process — or a process tree — running with a deliberately configured environment. The kernel schedules it like any other process. What makes it a container is the configuration around it: what it can see, what it can use, which filesystem it sees as `/`, what authority it has, and how it reaches the network.



Why The Boundary Is Not A Hypervisor

VMs and containers both isolate workloads, but at different layers. A VM runs a guest kernel on hardware virtualization. A Linux container shares the host kernel — which is why startup is fast (process creation, not a boot) and why the container is tied to that kernel's version, config, and security state. A workload that needs a different kernel needs something more than a Linux container.

The container boundary is the assembled effect of namespaces, cgroups, mount setup, credentials, capabilities, seccomp, LSMs, device rules, and runtime policy. It is strong when the configuration is right, but its shape is nothing like a hypervisor's: a hypervisor isolates with a separate guest kernel and a virtual hardware interface, while a container is the host kernel applying scoping rules to one of its own processes. The practical consequence is that a kernel bug like CVE-2022-0185 (filesystem-context heap overflow) is a container-escape primitive; the same bug under a VM stops at the guest kernel.

Not An Image

An image is packaged content and metadata — in OCI terms, a manifest, an image config, and a set of layer blobs, all addressed by digest. It can sit in a registry or a local content store with no process running anywhere.

A container is what happens when that content is combined with runtime configuration to start a process. "Run an image" is shorthand for a fixed sequence: the runtime resolves the reference, fetches the manifest and layers, stores them by digest, hands them to a snapshotter that produces a writable root filesystem, and starts a process inside that filesystem under an OCI runtime spec. The image is an input. The container is the running result.

The Process At The Center

A container has an ordinary Linux process at its center, with everything any other Linux process has: a command, environment, credentials, file descriptors, signal handlers, parents and children, an exit status. The runtime does not replace any of that; it configures the world around it.

The consequences are mundane. When the configured process exits, the container's task is over. When it spawns children, the runtime and kernel track the whole tree. A signal from outside has to become a Linux signal delivered to the right PID. Bytes written to stdout have to land somewhere the runtime stack has wired up.

Namespaces: What The Process Can See

Namespaces wrap global system resources so a process sees a scoped instance of each one. Eight namespace types are relevant to containers:

- **mount** — the mount table;
- **PID** — the process ID space;
- **network** — interfaces, addresses, routes, firewall state, ports;
- **UTS** — hostname and domain name;
- **IPC** — System V IPC and POSIX message queues;
- **user** — UID and GID mappings;
- **cgroup** — the cgroup hierarchy view;
- **time** — offsets for `CLOCK_MONOTONIC` and `CLOCK_BOOTTIME` (added in Linux 5.6).

Inside a container, a process might believe it is PID 1; on the host it has another PID, and both are real within their respective views. Namespaces on their own do not limit memory, CPU, or process count, and an isolated mount namespace can still expose dangerous host paths if a careless bind mount puts them there.

cgroups: What The Process Can Use

cgroups organize processes into a hierarchy and attach resource controllers to it. Where namespaces govern visibility, cgroups govern consumption: memory limits and OOM behavior, CPU weights and quotas, cpuset placement, IO controls, pids limits, and the accounting that makes any of it observable.

Linux has shipped cgroup v2 since 4.5 (2016); systemd made it the default in v243 (2019), and major distributions followed by 2022 (RHEL 9). v2 is a unified hierarchy with one consistent controller model, replacing v1's per-controller hierarchies. On systemd hosts, runtimes ask systemd for a transient scope or slice over D-Bus and let systemd create the cgroup with `Delegate=yes`; the runtime itself never writes arbitrary paths under `/sys/fs/cgroup`.

The split to hold onto:

- namespaces answer *what world does this process see?*
- cgroups answer *what resources can this process group use?*

The Root Filesystem

Inside a container, `/` is almost never the host's `/`. It is a prepared root filesystem assembled from image layers and runtime mounts.

Producing it follows a fixed sequence: layers arrive as content; the snapshotter unpacks them and stacks them, typically with overlayfs; a writable upper layer is added for this container; the runtime sets up the mount table inside a fresh mount namespace; and `pivot_root(2)` swaps the prepared tree in as the new `/`. Bind mounts then expose specific host paths, tmpfs is mounted where needed, certain paths are masked or read-only, and device nodes are restricted.

The image is the repeatable base; the runtime decides what host-specific mounts, secrets, devices, and writable paths show up.

Security Controls: What The Process May Do

In Unix, visibility and permission are different problems. A container process can see a path it cannot read, run as UID 0 inside a user namespace while mapping to an unprivileged UID on the host, hold a shrunken set of capabilities, and have most of its syscalls denied by seccomp and most of its file accesses denied by AppArmor or SELinux.

The usual controls:

- **capabilities** — split root-equivalent authority into individually grantable pieces;

- **seccomp** — filter or block system calls;
- **AppArmor / SELinux** — apply mandatory access control policy;
- **user namespaces** — map UIDs and GIDs;
- **read-only and masked mounts** — limit filesystem exposure;
- **device cgroup rules** — control which device nodes work.

The boundary is the combined configuration. Dropping `CAP_NET_ADMIN` is fine for a web server but breaks a CNI plugin that needs to manage routes inside the netns; turning off seccomp's default deny list lets a workload call `keyctl(2)` again, which is fine for some images and a privilege-escalation vector for others.

Networking: How The Process Communicates

A container usually runs in its own network namespace, with its own loopback device, addresses, routes, firewall state, and ports. Connecting it to anything else is left to the runtime: typical solutions include veth pairs into a bridge, routed setups, NAT, overlay networks, eBPF datapaths, or direct device assignment.

Kubernetes treats this slightly differently. It creates one network namespace per pod sandbox, and every container in that pod runs inside the same namespace. This is why containers in a pod can talk to each other over `localhost` — they share the namespace.

The Container Network Interface (CNI) fits at this layer. CNI is a plugin specification, not a container or a runtime. The runtime creates (or receives) a network namespace, then invokes plugin binaries with `ADD` to set it up and `DEL` to tear it down. A concrete `ADD` call: containerd executes `/opt/cni/bin/bridge`, passes the namespace path in `CNI_NETNS` and a JSON config on stdin, and the bridge plugin creates a veth pair, moves one end into the namespace, attaches the other to a Linux bridge on the host, and delegates address allocation to an IPAM plugin. Part V walks through this in detail.

The Metadata Outside

The running process is only half the story. Outside it, the runtime tracks the image reference, labels and annotations, the snapshot key, the runtime name, the OCI spec, task status, the shim process, IO pipes and logs, exit status, and the leases that keep garbage collection from reclaiming content still in use.

This outside state is why containerd separates a *container* from a *task*. The container object is metadata that can exist without any process; the task is the live execution of that metadata.

A Thought Experiment

Take `/bin/sh` on a Linux host. Run it normally; it sees the host's process tree, mounts, network, and resource environment. Now change one thing at a time. The commands below add isolation in roughly the order a runtime does. Run them on a disposable Linux VM as root — each one mutates kernel state — and refer to chapter 4 onward for the flag-by-flag detail.

Swap the root filesystem (mount namespace plus `pivot_root`): `/` means something different.

```
# Build a tiny rootfs and enter a shell whose / is that directory.
mkdir -p /tmp/rootfs && docker export $(docker create alpine:3.20) | tar -x -C /tmp/rootfs
sudo unshare --mount --uts --pid --fork --mount-proc=/tmp/rootfs/proc \
  chroot /tmp/rootfs /bin/sh
# Inside: ls / shows alpine's layout, not the host's.
```

Add a UTS namespace: it gets its own hostname.

```
sudo unshare --uts -- /bin/sh -c 'hostname container-demo; hostname; exit'
hostname   # unchanged on the host
```

Add a PID namespace: it sees a process tree where it might be PID 1.

```
sudo unshare --pid --fork --mount-proc -- /bin/sh -c 'echo "I am PID $$"; ps -ef'
# I am PID 1
# UID  PID  PPID ... CMD
#   0    1     0 ... /bin/sh -c echo "I am PID $$"; ps -ef
```

Add a cgroup with limits: its memory and CPU are bounded and accounted.

```
sudo mkdir /sys/fs/cgroup/demo
echo "100M"     | sudo tee /sys/fs/cgroup/demo/memory.max
echo "50000 100000" | sudo tee /sys/fs/cgroup/demo/cpu.max   # 50% of one CPU
sudo unshare --pid --fork --mount-proc -- /bin/sh -c '
  echo $$ > /sys/fs/cgroup/demo/cgroup.procs
  cat /sys/fs/cgroup/demo/memory.current
'
sudo rmdir /sys/fs/cgroup/demo   # cleanup
```

Add a network namespace and a veth pair: it has its own network stack, plumbed to the host.

```
sudo ip netns add demo
sudo ip link add veth-h type veth peer name veth-c
sudo ip link set veth-c netns demo
sudo ip addr add 10.20.0.1/24 dev veth-h && sudo ip link set veth-h up
sudo ip -n demo addr add 10.20.0.2/24 dev veth-c
sudo ip -n demo link set veth-c up && sudo ip -n demo link set lo up
sudo ip netns exec demo /bin/sh -c 'ip addr; ping -c1 10.20.0.1'
sudo ip netns del demo && sudo ip link del veth-h 2>/dev/null
```

Drop capabilities, attach a seccomp profile, and apply an LSM policy: it has less authority even in a tree it appears to own.

```
# Capabilities: run /bin/sh with no caps in any set.
sudo capsh --drop=all --caps="" -- -c 'grep ^Cap /proc/self/status; id'

# Seccomp: deny mkdir(2) for this shell only (requires runc/docker for full profiles).
docker run --rm --security-opt seccomp=<(echo '{
  "defaultAction":"SCMP_ACT_ALLOW",
  "syscalls":[{"names":["mkdir","mkdirat"],"action":"SCMP_ACT_ERRNO"}]
}') alpine:3.20 sh -c 'mkdir /tmp/x || echo "mkdir blocked"'

# LSM: confine /bin/sh under the docker-default AppArmor profile.
docker run --rm --security-opt apparmor=docker-default alpine:3.20 \
  sh -c 'cat /proc/self/attr/current'
# docker-default (enforce)
```

Each command on its own is a small kernel call. Stacked, they are what a runtime hands the kernel when it starts a container.

A Working Definition

The word *container* gets attached to all of these: a tarball, an image in a registry, a `chroot`, a single namespace or cgroup, a `docker run` command, a Kubernetes pod, even a VM. Each names a real part of the picture, and none of them is the whole thing. Mistaking the part for the whole is how the word loses meaning.

> *A container is a process or process tree, started from packaged content and a runtime spec, isolated and constrained by host kernel primitives, and tracked by metadata that lives outside it.*

Chapter 3 picks up the contracts that hold these pieces together: the OCI runtime spec and the runtime v2 shim API.

Sources And Further Reading

- Linux namespaces: https://man7.org/linux/man-pages/man7/namespaces.7.html
- Linux cgroups: https://man7.org/linux/man-pages/man7/cgroups.7.html
- Linux cgroup v2 docs: https://kernel.org/doc/html/next/admin-guide/cgroup-v2.html
- OCI Runtime Specification: https://github.com/opencontainers/runtime-spec
- OCI Image Specification: https://github.com/opencontainers/image-spec
- CNI specification: https://www.cni.dev/docs/spec/

Chapter 3: Standards — OCI and Runtime v2

Each arrow is a different contract. CRI is owned by Kubernetes. Runtime v2 is owned by containerd. The OCI runtime spec is owned by the OCI. They were designed at different times by different groups.

What OCI Is

The Open Container Initiative is a Linux Foundation project formed at DockerCon on June 22, 2015. Its mandate is to publish open specifications for container formats and runtimes. The point is interoperability: a runtime, a registry, or a builder should be replaceable without rewriting everything above it.

OCI publishes three specifications, each versioned independently:

- **Runtime Specification** — what a runtime consumes and how it behaves.
- **Image Specification** — what an image is.
- **Distribution Specification** — how images move through registries.

The Runtime and Image specs were the original two. The Distribution spec, derived from Docker's Registry HTTP API V2, reached 1.0 in 2020 and turned an existing de facto standard into an open one.

This chapter focuses on the Runtime Specification.

The OCI Runtime Specification

The runtime spec defines two artifacts and a small lifecycle.

The Bundle

A **bundle** is a directory on disk containing two things: a `config.json` file and a root filesystem the runtime will use as `/`.

`config.json` describes the desired environment in fields that map to the chapter 2 model:

- `process` — args, env, cwd, user, capabilities, rlimits, terminal, `noNewPrivileges`.
- `root` — path to the root filesystem and a `readonly` flag.
- `mounts` — destination, type, source, options. The mount table the runtime should set up before exec.
- `hostname`.
- `linux` — the Linux-specific block: namespaces to enter or create, UID and GID mappings, devices, `cgroupsPath`, `resources` (per-controller cgroup settings), `seccomp` filter, `maskedPaths` and `readonlyPaths`, `rootfsPropagation`, sysctls.
- `hooks` — code to run at fixed points in the lifecycle.
- `annotations` — opaque key/value metadata.

`ociVersion` pins the spec version the config targets; runc rejects bundles whose major version it does not recognize.

A trimmed but realistic `config.json` — the kind of file `runc spec` generates and the shim hands runc — looks like this:

```
{
  "ociVersion": "1.2.0",
  "process": {
    "terminal": false,
    "user": { "uid": 0, "gid": 0 },
    "args": ["/bin/sh"],
    "env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "TERM=xterm"
    ],
    "cwd": "/",
    "capabilities": {
      "bounding":  ["CAP_AUDIT_WRITE", "CAP_KILL", "CAP_NET_BIND_SERVICE"],
      "effective": ["CAP_AUDIT_WRITE", "CAP_KILL", "CAP_NET_BIND_SERVICE"],
      "permitted": ["CAP_AUDIT_WRITE", "CAP_KILL", "CAP_NET_BIND_SERVICE"]
    },
    "rlimits": [
      { "type": "RLIMIT_NOFILE", "hard": 1024, "soft": 1024 }
    ],
    "noNewPrivileges": true
  },
  "root": { "path": "rootfs", "readonly": true },
  "hostname": "runc",
  "mounts": [
    { "destination": "/proc", "type": "proc", "source": "proc" },
    { "destination": "/dev",  "type": "tmpfs", "source": "tmpfs",
      "options": ["nosuid", "strictatime", "mode=755", "size=65536k"] },
    { "destination": "/sys",  "type": "sysfs", "source": "sysfs",
      "options": ["nosuid", "noexec", "nodev", "ro"] }
  ],
  "linux": {
    "namespaces": [
      { "type": "pid" }, { "type": "network" }, { "type": "ipc" },
      { "type": "uts" }, { "type": "mount" }
    ],
    "uidMappings": [{ "containerID": 0, "hostID": 100000, "size": 65536 }],
    "gidMappings": [{ "containerID": 0, "hostID": 100000, "size": 65536 }],
    "resources": {
      "memory": { "limit": 268435456 },
      "cpu":    { "shares": 512, "quota": 50000, "period": 100000 },
      "devices": [{ "allow": false, "access": "rwm" }]
    },
    "maskedPaths":  ["/proc/kcore", "/proc/keys", "/sys/firmware"],
    "readonlyPaths": ["/proc/asound", "/proc/bus", "/proc/sys"],
    "seccomp": { "defaultAction": "SCMP_ACT_ERRNO" }
  }
}
```

Every chapter-2 concept has a slot here: namespaces under `linux.namespaces`, cgroup limits under `linux.resources`, the rootfs under `root`, capabilities and `noNewPrivileges` under `process`, mounts and masked paths in their own sections. Generate a fully-populated default in any directory with `runc spec`, then edit it down — that is the path most runtime work takes.

The Lifecycle

A compliant OCI runtime exposes five lifecycle commands:

- `create` — set up namespaces, mounts, cgroups, security policy. Stop at the entry point of the container's init process and wait. The user process is not yet running.
- `start` — exec the user process.
- `state` — report the container's status as JSON: `id`, `status`, `pid`, `bundle`.
- `kill` — send a signal.

- `delete` — release any state retained by the runtime.

`create` and `start` are split so the caller can do work between resource setup and process execution: attach IO, set up a network namespace from outside, run hooks. That is also why containerd's shim uses the two-phase form rather than `runc run`.

Hooks

Hooks let external code run at fixed points in the container's lifecycle. The current spec defines five:

- `createRuntime` — runs in the runtime's namespace, after namespaces are created but before `pivot_root`.
- `createContainer` — runs inside the container's namespace, before `pivot_root`.
- `startContainer` — runs inside the container's namespace, just before `exec`.
- `poststart` — after the user process starts.
- `poststop` — after the user process exits.

There is also a `prestart` hook that was deprecated in spec 1.0.2 in favor of the more granular hooks above; runtimes still support it for compatibility. Hooks are how networking, GPU device injection, and a lot of runtime extension behavior get wired in without modifying the runtime itself.

OCI Runtimes In Practice

An OCI runtime is anything that consumes a valid bundle and implements those lifecycle commands. The interchangeability is the point: the layers above the runtime — containerd, Docker, Kubernetes — pick a runtime by configuration, not by code.

The common implementations:

- **runc** (Go) — the reference implementation, donated by Docker; uses libcontainer for the Linux setup.
- **crun** (C) — smaller and faster than runc; lower memory and quicker startup; maintained by Red Hat.
- **youki** (Rust) — runc-equivalent semantics in a different language.
- **runsc** (gVisor) — Google's user-space kernel; intercepts syscalls in a sandbox process. Trades performance for a stronger isolation boundary.
- **kata-containers** — runs each container inside a lightweight VM; OCI-compatible from the caller's perspective.
- **nvidia-container-runtime** — a thin wrapper around runc that injects GPU devices via a hook.

Containerd's Runtime v2

containerd does not exec the OCI runtime directly. It launches a **shim** per task, and the shim drives the runtime on its behalf. This is the runtime v2 model, defined in `docs/runtime-v2.md` in the containerd repo.

Why The Shim Exists

If containerd were the direct parent of every container process, three things would break:

1. **Restarting containerd would kill containers.** When the parent dies, the children either die with it or are reparented to init.
2. **IO and exit handling would leak into containerd.** Stdin/stdout pipes, exit notification, and signal forwarding are runtime-specific.
3. **Runtime swaps would require daemon changes.** With the shim contract, swapping runc for crun is a configuration change.

v1 Versus v2

The original v1 shim was defined inside containerd and predated a stable wire protocol. Runtime v2 is a versioned API: the shim binary speaks a defined ttrpc Task service, and any runtime can be wrapped behind it without touching containerd. v2 also supports a single shim serving multiple containers — typically one shim per pod sandbox in CRI mode — which cuts per-container overhead in Kubernetes.

containerd 1.0 (2017) shipped v1; 1.4 (2020) deprecated it; v2 is what every current installation uses.

Naming

A v2 shim is identified by a name like `io.containerd.runc.v2`. The binary name is the same string with dots replaced by dashes: `containerd-shim-runc-v2`. containerd resolves the name to a binary on `$PATH` when starting a task.

The shims you are likely to see:

Runtime name	Binary	Purpose
`io.containerd.runc.v2`	`containerd-shim-runc-v2`	Standard Linux runc path
`io.containerd.runhcs.v1`	`containerd-shim-runhcs-v1`	Windows HCS
Vendor-specific	`containerd-shim-kata-v2`, `containerd-shim-runsc-v1`, etc.	Kata, gVisor, Firecracker

The Wire Protocols

The four arrows in this chapter's opening diagram each use a different transport. Walking from top to bottom:

CRI gRPC: kubelet → containerd. The kubelet calls containerd over a Unix domain socket — `/run/containerd/containerd.sock` by default — using the `RuntimeService` and `ImageService` definitions in `cri-api`. These are full gRPC over HTTP/2: requests like `RunPodSandbox`, `CreateContainer`, and `StartContainer` are unary, while `Attach`, `Exec`, and `PortForward` use streaming. The kubelet picks the socket up from `--container-runtime-endpoint`; the path is the only handshake.

ttrpc Task service: containerd → shim. Once a task exists, containerd talks to its shim over **ttrpc** — a slimmer gRPC variant maintained at `github.com/containerd/ttrpc`. The protobuf definitions are gRPC-shaped, but the wire skips HTTP/2: framed protobuf over a Unix socket, no streaming, smaller binary, smaller memory. ttrpc fits here because shims are cheap and there is one per workload (or per pod) — a full HTTP/2 stack per shim would cost more than the shim itself. The Task service the shim implements covers the operations a daemon needs:

- `Create`, `Start`, `Delete` — task lifecycle.
- `Exec`, `Kill`, `ResizePty`, `CloseIO` — interactive control.
- `State`, `Pids`, `Wait`, `Stats` — status and observation.
- `Pause`, `Resume`, `Update`, `Checkpoint` — runtime control.
- `Connect`, `Shutdown` — shim-level lifecycle.

When the shim is sandbox-aware (i.e. it can host a Kubernetes pod), a separate Sandbox service sits alongside Task.

Events: shim → containerd, via a publish binary. Containers produce asynchronous events — `TaskExit`, `TaskOOM`, `TaskCreate` — that the shim has to push back to containerd. Rather than open a second long-lived ttrpc connection, the shim invokes a small binary it was given at startup. containerd execs the shim with `-publish-binary /usr/local/bin/containerd` and `-address <main socket>`; the shim runs `containerd publish --topic=/tasks/exit --namespace=<ns>` for each event, with a serialized protobuf envelope on stdin. The publish binary is a thin client that forwards the event over ttrpc to containerd's Events service and exits. From the shim's perspective, an event is one `fork` / `exec`.

OCI runtime CLI: shim → runc. The shim does not link runc as a library. It writes the bundle to disk and invokes runc as a subprocess for each lifecycle step: `runc create`, `runc start`, `runc kill`, `runc delete`. The "protocol" is the CLI — arguments and exit codes — with several out-of-band channels:

- **stdio** is plumbed through inherited file descriptors. When `process.terminal` is true, runc allocates a pseudo-terminal and sends the master fd to the shim over a Unix socket whose path the shim passes as `--console-socket`.
- **The bundle directory** is the input. runc reads `config.json` and the rootfs from it; the path is the last positional argument to `runc create`.

- **The pid file** (`--pid-file`) is where runc writes the init process's host PID after `create` returns. The shim opens it, reads the PID, and uses `wait4(2)` (or pidfd) to detect exit.
- **The state directory** (default `/run/runc/<id>/`) holds runc's own `state.json` for each container, so subsequent `runc kill` or `runc delete` calls find the container without the shim having to keep any in-memory handle.

Part III walks through the runc side of these calls in detail.

The Startup Handshake

containerd starts a shim by execing its binary with the subcommand `start` and a fixed set of flags: `-namespace` (a containerd namespace, not a Linux one), `-id` (the task id), `-address` (containerd's main ttrpc socket), and `-publish-binary` (the events client). The shim then:

1. Creates its own ttrpc socket — a Unix abstract socket under containerd's state directory, e.g. `/run/containerd/s/<random>` .
2. Forks itself; the child runs the ttrpc server, the parent prints the socket address on stdout and exits.
3. containerd reads the address, dials it, and from then on drives the Task service over ttrpc.

The shim process — not containerd — is the parent of the container's init process. Containerd holds a ttrpc connection to the shim, the shim holds the runc invocations and the init PID, and runc has already exited by the time `Start` returns. If containerd restarts, the shim keeps running and keeps holding the container; on reconnect, containerd dials the existing socket and resumes management without restarting the workload.

Part III opens up the OCI runtime spec and runc. Part IV opens up containerd, which is where CRI on top and runtime v2 underneath both meet.

Sources And Further Reading

- OCI homepage: https://opencontainers.org/
- OCI Runtime Specification: https://github.com/opencontainers/runtime-spec
- OCI Image Specification: https://github.com/opencontainers/image-spec
- OCI Distribution Specification: https://github.com/opencontainers/distribution-spec
- runc: https://github.com/opencontainers/runc
- crun: https://github.com/containers/crun
- youki: https://github.com/containers/youki
- gVisor: https://gvisor.dev/
- Kata Containers: https://katacontainers.io/
- containerd runtime v2: https://github.com/containerd/containerd/blob/main/docs/runtime-v2.md
- containerd ttrpc: https://github.com/containerd/ttrpc

PART II — LINUX PRIMITIVES

Chapter 4: Namespaces

> ***Safety:*** *the commands below mutate kernel state and most need root. Use a disposable Linux VM. Examples here were checked on Ubuntu 24.04 with kernel 6.8 and the* `util-linux` *package supplying* `unshare`*,* `nsenter`*, and* `lsns`*.*

What Lives In `/proc/<pid>/ns/`

Every process exposes its current namespaces as symlinks under `/proc/<pid>/ns/`. Two processes share a namespace exactly when their symlinks point at the same inode.

```
ls -l /proc/self/ns/
# lrwxrwxrwx 1 me me 0 ... cgroup -> 'cgroup:[4026531835]'
# lrwxrwxrwx 1 me me 0 ... ipc    -> 'ipc:[4026531839]'
# lrwxrwxrwx 1 me me 0 ... mnt    -> 'mnt:[4026531840]'
# lrwxrwxrwx 1 me me 0 ... net    -> 'net:[4026531992]'
# lrwxrwxrwx 1 me me 0 ... pid    -> 'pid:[4026531836]'
# lrwxrwxrwx 1 me me 0 ... pid_for_children -> 'pid:[4026531836]'
# lrwxrwxrwx 1 me me 0 ... time   -> 'time:[4026531834]'
# lrwxrwxrwx 1 me me 0 ... time_for_children -> 'time:[4026531834]'
# lrwxrwxrwx 1 me me 0 ... user   -> 'user:[4026531837]'
# lrwxrwxrwx 1 me me 0 ... uts    -> 'uts:[4026531838]'
```

The `_for_children` entries apply to processes the current process spawns. PID and time namespace changes take effect at the next `fork(2)`, not immediately, so the kernel exposes both the current value and the value newly-forked children will inherit.

The same information in tabular form:

```
lsns
```

`lsns` walks `/proc/*/ns/` and groups processes by namespace; it is the easiest way to figure out which container is using which namespace in a debug session.

Creating Namespaces With `unshare`

`unshare(1)` is the user-space wrapper for `unshare(2)`. It creates new namespaces and execs a command inside them.

A UTS namespace needs `CAP_SYS_ADMIN` to create directly:

```
sudo unshare --uts -- bash -c 'hostname inside; hostname'
# inside
hostname
# (unchanged on host)
```

To get the same behavior unprivileged, wrap it in a user namespace — the one namespace that does not require pre-existing privilege to create:

```
unshare --user --map-root-user --uts -- bash -c 'hostname inside; hostname'
# inside
hostname
# (unchanged on host)
```

`--map-root-user` writes a UID/GID mapping that maps the caller to UID 0 inside the new user namespace. The shell believes it is root; from the host it is the same unprivileged process.

PID Namespace And The PID 1 Problem

A new PID namespace makes the first process inside it PID 1.

```
sudo unshare --pid --fork --mount-proc -- bash -c 'echo "pid: $$"; ps -ef'
# pid: 1
# UID  PID  PPID ... CMD
#   0    1     0 ... bash -c echo "pid: $$"; ps -ef
#   0    2     1 ... ps -ef
```

`--fork` is required because the calling process cannot move into the new PID namespace itself; it must fork a child that becomes PID 1. `--mount-proc` remounts `/proc` inside the new mount namespace so `ps` reflects the new PID space.

PID 1 is special, and the special treatment is the reason container processes routinely fail to exit on `docker stop` or `kubectl delete pod`. The kernel does not deliver default signal actions to PID 1: `SIGTERM` will not terminate it unless the process installs a handler. To see the failure:

```
sudo unshare --pid --fork --mount-proc -- bash -c '
  echo "pid 1 = $$"
  while :; do sleep 1; done
'
# In another terminal:
# kill -TERM <pid-of-the-bash-process-on-host>
# bash continues running
```

The kernel will not deliver the default-action `SIGTERM`. `SIGKILL` works because it cannot be ignored. Container runtimes work around this by injecting a tiny init like `tini` or `dumb-init` that installs handlers and forwards signals to its children, and reaps zombies on their behalf.

Mount Namespaces And Propagation

A mount namespace governs which mounts a process sees. To watch a mount appear and disappear:

```
sudo unshare --mount -- bash
# Inside the new namespace:
mount -t tmpfs tmpfs /mnt
mount | grep /mnt
# tmpfs on /mnt type tmpfs (rw,relatime)
exit
# Back on the host:
mount | grep /mnt
# (nothing)
```

The host never saw the mount, because the mount namespace's mount table is private. The catch is propagation. Many distributions configure `/` as a shared mount, which would make the new namespace inherit shared peers and propagate events back to the host. `unshare --mount` defaults to `--propagation private` on the new mount namespace's root to prevent that.

To inspect propagation:

```
findmnt -o TARGET,PROPAGATION /
# TARGET PROPAGATION
# /      shared
```

When runc sets up a container, it sets the new mount namespace root to `private` (or whatever `linux.rootfsPropagation` requests), then mounts the rootfs and special filesystems before `pivot_root(2)` swaps it in. Chapter 6 walks through that sequence.

Network Namespaces

Network namespaces are visible enough to deserve their own subcommand, `ip netns`. Unlike `unshare`, `ip netns add` keeps the namespace alive after the calling process exits by bind-mounting it to `/var/run/netns/<name>`.

```
sudo ip netns add demo
sudo ip netns list
# demo
ls /var/run/netns/
# demo
```

Each namespace starts with a loopback interface, which is *down*:

```
sudo ip netns exec demo ip link
# 1: lo: <LOOPBACK> mtu 65536 state DOWN ...
sudo ip netns exec demo ip link set lo up
sudo ip netns exec demo ping -c1 127.0.0.1
```

Connecting two namespaces requires a virtual link. The classic recipe is a `veth` pair:

```
sudo ip netns add a
sudo ip netns add b

# Create a pair with two ends.
sudo ip link add va type veth peer name vb

# Move each end into a namespace.
sudo ip link set va netns a
sudo ip link set vb netns b

# Bring up loopback and the veths.
sudo ip -n a link set lo up
sudo ip -n b link set lo up
sudo ip -n a link set va up
sudo ip -n b link set vb up

# Address each side.
sudo ip -n a addr add 10.10.0.1/24 dev va
sudo ip -n b addr add 10.10.0.2/24 dev vb

# Test.
sudo ip netns exec a ping -c1 10.10.0.2
```

This is one bridge away from the standard container networking pattern: replace `vb` going into namespace `b` with `vb` attached to a host bridge, and add NAT rules.

Joining An Existing Namespace With `nsenter`

`nsenter(1)` calls `setns(2)` on a target namespace and execs a command. To run a command inside a process's mount namespace:

```
# Find a target pid.
pgrep -f some-container-process

# Enter its mount and PID namespaces.
sudo nsenter -t <pid> -m -p -- ls /proc
```

`kubectl exec`, `crictl exec`, and `docker exec` all reduce to a `setns` chain plus an `exec`. The reason it has to chain in a specific order is that some namespace transitions invalidate previously-set state — notably, joining a mount namespace makes paths from the old namespace unresolvable, so PID namespace switches that need to read `/proc` must come first.

User Namespaces And Privilege Scoping

User namespaces are what make rootless containers possible: they let an unprivileged user gain root-equivalent capabilities scoped to a namespace they own.

To create one and observe the mapping:

```
unshare --user --map-root-user -- bash -c '
  echo "id inside: $(id)"
  cat /proc/self/uid_map
'
# id inside: uid=0(root) gid=0(root) groups=0(root)
# 0          1000          1
```

The mapping reads as `<inside-uid> <outside-uid> <length>`. UID 0 inside maps to UID 1000 outside, for one UID. The shell believes it is root inside; from the host it is the same unprivileged user.

A user namespace gives root-equivalent capabilities only over resources owned by *that* user namespace. Try to do something that requires actual host root:

```
unshare --user --map-root-user -- bash -c '
  # Try to mount something that requires CAP_SYS_ADMIN on the host.
  mount -t tmpfs tmpfs /mnt
'
# mount: /mnt: permission denied. (only privileged user can mount)
```

The same `CAP_SYS_ADMIN`, scoped to the user namespace, is enough to create more namespaces:

```
unshare --user --map-root-user -- bash -c '
  unshare --uts -- bash -c "hostname inside; hostname"
'
# inside
# inside
```

Mounts owned by the host's user namespace remain off-limits, but namespace creation that requires `CAP_SYS_ADMIN` on the *new* namespace's owner works.

For unprivileged users to map UIDs other than their own, they need `newuidmap(1)` and `newgidmap(1)` (setuid helpers from `shadow-utils`) and entries in `/etc/subuid` and `/etc/subgid`:

```
grep "^$(whoami):" /etc/subuid /etc/subgid
# /etc/subuid:me:100000:65536
# /etc/subgid:me:100000:65536
```

These say: the user `me` may map host UIDs 100000 through 165535 inside any user namespace they own. This is what gives a rootless container a 64K-UID range to allocate inside.

Time Namespaces

Time namespaces offset only `CLOCK_MONOTONIC` and `CLOCK_BOOTTIME`. `CLOCK_REALTIME` is shared with the host and cannot be changed per-namespace.

```
sudo unshare --time --fork -- bash -c '
  echo "Before offset:"
  cat /proc/uptime
  # Apply offsets via /proc/<pid>/timens_offsets.
  # Format: <clock_id> <secs> <nanosecs>
  # CLOCK_MONOTONIC=1, CLOCK_BOOTTIME=7
  echo "1 -100 0" > /proc/$$/timens_offsets
  echo "After offset:"
  cat /proc/uptime
'
```

`timens_offsets` must be written before any process executes inside the namespace, which is why runtimes write it in the brief window after `unshare(CLONE_NEWTIME)` and before `exec`.

Putting It Together: A Hand-Rolled Container

The same assembly without runc: a shell with its own user, PID, mount, UTS, IPC, and net namespaces, and an Alpine rootfs as `/`.

```
# Get a small rootfs.
mkdir alpine-rootfs
docker export $(docker create alpine:3.20) | tar -C alpine-rootfs -xf -

# Run a shell in fresh namespaces with that as /.
sudo unshare \
  --user --map-root-user \
  --pid --fork --mount-proc \
  --mount --uts --ipc --net \
  -- chroot alpine-rootfs /bin/sh

# Inside:
# / # hostname
# (empty -- new UTS namespace)
# / # ps
# PID   USER     TIME  COMMAND
#     1 root      0:00 /bin/sh
#     2 root      0:00 ps
# / # ls /
# bin etc lib mnt proc root sbin sys tmp usr var
```

This is intentionally crude. There is no cgroup yet, no seccomp, no capability dropping, no `pivot_root` (just `chroot`), no IO or signal supervision, no networking inside the new netns.

OCI Mapping

The runtime spec's `linux.namespaces` array names which namespaces to enter or create. Each entry is `{type, path}`; if `path` is set, the runtime calls `setns(2)` on it instead of creating a new one.

This is how Kubernetes shares a pod's network namespace across containers — every container in the pod has its config's `network` entry pointing at the same `/proc/<pid>/ns/net` path. The first container creates the namespace; the rest join it.

The `linux.uidMappings` and `linux.gidMappings` arrays carry the user-namespace ID mappings. `linux.timeOffsets` carries the time-namespace clock offsets.

Where This Goes

Namespaces by themselves do not constrain CPU or memory and do not prevent a process from forking until the host runs out of process slots. The next chapter — cgroups v2 — covers the resource side.

Sources And Further Reading

- `namespaces(7)`: https://man7.org/linux/man-pages/man7/namespaces.7.html
- `unshare(1)`: https://man7.org/linux/man-pages/man1/unshare.1.html
- `nsenter(1)`: https://man7.org/linux/man-pages/man1/nsenter.1.html
- `user_namespaces(7)`: https://man7.org/linux/man-pages/man7/user_namespaces.7.html
- `pid_namespaces(7)`: https://man7.org/linux/man-pages/man7/pid_namespaces.7.html
- `mount_namespaces(7)`: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html
- `network_namespaces(7)`: https://man7.org/linux/man-pages/man7/network_namespaces.7.html
- `time_namespaces(7)`: https://man7.org/linux/man-pages/man7/time_namespaces.7.html
- OCI Runtime Spec, namespaces: https://github.com/opencontainers/runtime-spec/blob/main/config-linux.md#namespaces

Chapter 5: Cgroups v2

Namespaces decide what a process sees. Cgroups decide what it can use.

> ***Safety:*** *writing to `/sys/fs/cgroup` requires root. The example that triggers an OOM kill is harmless on a VM but will cause a real OOM event in the kernel log. Use a disposable Linux VM. Examples were checked on Ubuntu 24.04 with kernel 6.8 and systemd 255 in cgroup v2 unified mode.*

Confirm v2 Is Active

Ubuntu 22.04+, Fedora 31+, Debian 11+, and RHEL 9 default to v2. The v2 unified hierarchy is mounted as `cgroup2` at `/sys/fs/cgroup`:

```
mount | grep cgroup2
# cgroup2 on /sys/fs/cgroup type cgroup2 (rw,nosuid,nodev,noexec,relatime,nsdelegate,memory_recursiveprot)
```

If you see additional `cgroup` mounts under `/sys/fs/cgroup/<controller>/`, the system is in legacy v1 or hybrid mode and the rest of this chapter does not apply directly. To force unified mode, set `systemd.unified_cgroup_hierarchy=1` on the kernel command line and reboot.

Walk The Tree

Every directory under `/sys/fs/cgroup` is a cgroup. The root is `/sys/fs/cgroup` itself.

```
ls /sys/fs/cgroup/ | head
# cgroup.controllers
# cgroup.max.depth
# cgroup.max.descendants
# cgroup.procs
# cgroup.subtree_control
# cgroup.threads
# cpu.pressure
# cpu.stat
# init.scope
# io.pressure
# ...
cat /sys/fs/cgroup/cgroup.controllers
# cpuset cpu io memory hugetlb pids rdma misc
```

`cgroup.controllers` lists what is *available* to this cgroup; `cgroup.subtree_control` lists what is *enabled* for children:

```
cat /sys/fs/cgroup/cgroup.subtree_control
# cpuset cpu io memory pids
```

Walking the tree as systemd has shaped it:

```
systemd-cgls --no-pager | head -30
```

The standard layout: `system.slice/` for system services, `user.slice/user-<uid>.slice/` for user sessions, and (when present) `kubepods.slice/` or `machine.slice/` for orchestrated workloads.

Make A Cgroup By Hand

Cgroups are directories. Creating one is `mkdir`:

```
sudo mkdir /sys/fs/cgroup/demo
ls /sys/fs/cgroup/demo/
# cgroup.controllers  cgroup.events  cgroup.procs  cgroup.stat
# cgroup.subtree_control  cgroup.threads  cgroup.type
# cpu.stat  cpu.weight  io.pressure  memory.pressure  ...
```

The controller-specific files visible here come from whichever controllers the *parent* has enabled in its `subtree_control`. To enable more, write the diff to the parent's `subtree_control` (write `+` to add, `-` to remove):

```
echo +memory +pids > /sys/fs/cgroup/cgroup.subtree_control 2>/dev/null \
  || sudo sh -c 'echo "+memory +pids" > /sys/fs/cgroup/cgroup.subtree_control'
```

A controller is enabled for a cgroup's children, not for the cgroup itself; the kernel docs call this the top-down constraint.

Move A Process In

Put a process into a cgroup by writing its PID to `cgroup.procs`:

```
sudo sh -c 'sleep 600 & echo $! > /sys/fs/cgroup/demo/cgroup.procs; wait'
# In another terminal:
cat /sys/fs/cgroup/demo/cgroup.procs
# 12345
```

A process can only be in one cgroup at a time. Writing a PID to a different cgroup's `cgroup.procs` moves it out of its previous cgroup atomically.

`cgroup.threads` works the same way but for individual threads, and only on cgroups whose `cgroup.type` is `threaded`.

Set A Memory Limit, Watch It OOM

Set a small memory limit, run a process that allocates more than that, and watch the kernel kill it within the cgroup:

```
sudo mkdir -p /sys/fs/cgroup/oom-demo
echo "+memory" | sudo tee /sys/fs/cgroup/cgroup.subtree_control > /dev/null
echo 50M | sudo tee /sys/fs/cgroup/oom-demo/memory.max > /dev/null

# Move the current shell into the cgroup, then exec a Python that allocates.
sudo sh -c 'echo $$ > /sys/fs/cgroup/oom-demo/cgroup.procs; \
  exec python3 -c "x=bytearray(200*1024*1024); print(\"alive\")"'
# Killed
echo $?
# 137  (128 + SIGKILL)
```

Confirm the OOM was scoped to the cgroup and did not affect anything else:

```
cat /sys/fs/cgroup/oom-demo/memory.events
# low 0
# high 0
# max 1
# oom 1
# oom_kill 1
```

`memory.max` is a hard limit. The kernel reclaims when the cgroup approaches it; if reclaim cannot free enough memory, the OOM killer fires *inside the cgroup* and picks a victim from the cgroup's processes.

For softer pushback before OOM, set `memory.high`:

```
echo 30M | sudo tee /sys/fs/cgroup/oom-demo/memory.high > /dev/null
```

The kernel will throttle allocations and force reclaim on this cgroup when it exceeds 30M, but will not kill anything. Combined with a higher `memory.max`, this gives the workload time to react.

CPU Caps And Weights

Two CPU files matter most:

- `cpu.weight` — proportional share when CPUs are oversubscribed. Default 100. Range 1–10000.
- `cpu.max` — `<quota> <period>` in microseconds. Default `max 100000`, meaning unlimited.

Set a half-CPU cap:

```
sudo mkdir -p /sys/fs/cgroup/cpu-demo
echo "+cpu" | sudo tee /sys/fs/cgroup/cgroup.subtree_control > /dev/null
echo "50000 100000" | sudo tee /sys/fs/cgroup/cpu-demo/cpu.max > /dev/null

# Run a busy loop in the cgroup.
sudo sh -c 'echo $$ > /sys/fs/cgroup/cpu-demo/cgroup.procs; \
  exec timeout 5 sh -c "while :; do :; done"'

# After it ends, look at cpu.stat:
cat /sys/fs/cgroup/cpu-demo/cpu.stat
# usage_usec     2500000
# user_usec      2500000
# system_usec       0
# nr_periods      ~50
# nr_throttled    ~50
# throttled_usec ~2500000
```

`nr_throttled` and `throttled_usec` show the cgroup hit its quota every period and was paused for the remainder. Kubernetes' CPU throttling alerts read these counters.

Limit Process Count

Forks happen all the time. A bug or attack can exhaust the host's pid space; `pids.max` defends against that:

```
sudo mkdir -p /sys/fs/cgroup/pid-demo
echo "+pids" | sudo tee /sys/fs/cgroup/cgroup.subtree_control > /dev/null
echo 5 | sudo tee /sys/fs/cgroup/pid-demo/pids.max > /dev/null

sudo sh -c 'echo $$ > /sys/fs/cgroup/pid-demo/cgroup.procs; \
  for i in 1 2 3 4 5 6 7 8; do \
    sleep 30 & echo "started $!"; \
  done'
# After the limit:
# sh: fork: retry: Resource temporarily unavailable
```

`pids.events` records the rejections:

```
cat /sys/fs/cgroup/pid-demo/pids.events
# max 4   <- the count of fork rejections
```

Read Pressure

PSI (Pressure Stall Information) is exposed at `cpu.pressure`, `memory.pressure`, and `io.pressure` in every cgroup. The values report the percentage of time some / all processes in the cgroup were stalled on the resource. PSI reflects contention rather than nominal load — a CPU at 80% utilization with no waiting tasks shows zero pressure, while a CPU at 40% with queued tasks shows non-zero:

```
cat /sys/fs/cgroup/oom-demo/memory.pressure
# some avg10=0.00 avg60=0.00 avg300=0.00 total=0
# full avg10=0.00 avg60=0.00 avg300=0.00 total=0
```

`some` is "at least one task stalled"; `full` is "all tasks stalled."

"No Internal Processes" In Practice

A non-root cgroup cannot both contain processes and have controllers enabled in its `subtree_control` that propagate to children. To see the rule fire:

```
sudo mkdir -p /sys/fs/cgroup/parent/child
sudo sh -c 'echo $$ > /sys/fs/cgroup/parent/cgroup.procs'
echo "+memory" | sudo tee /sys/fs/cgroup/parent/cgroup.subtree_control
# tee: /sys/fs/cgroup/parent/cgroup.subtree_control: Device or resource busy
```

Move the process out first, then the write succeeds:

```
sudo sh -c 'echo $$ > /sys/fs/cgroup/parent/child/cgroup.procs'
echo "+memory" | sudo tee /sys/fs/cgroup/parent/cgroup.subtree_control
# +memory
```

Containers always sit in leaf cgroups: the runtime creates a hierarchy of intermediate cgroups (slice → kubepods → pod → container), each of which has children but no processes, and puts the container's processes only in the leaf.

systemd Owns The Tree

On systemd-managed hosts (most production Linux), systemd owns the cgroup tree. The kernel enforces a single-writer model per cgroup directory — if two processes both try to manage the same cgroup, one will lose. systemd's answer is **delegation**: it creates a parent with `Delegate=yes`, marks it as owned by another manager, and stops touching what is underneath.

`systemd-run` is the convenient way to launch a transient unit and observe its cgroup placement:

```
sudo systemd-run --slice=demo.slice --unit=oneshot.service --scope sleep 60 &
systemctl status oneshot.service
# Look for "CGroup: /demo.slice/oneshot.service"
cat /sys/fs/cgroup/demo.slice/oneshot.service/cgroup.procs
# <pid of sleep>
```

When containerd is configured with the `systemd` cgroup driver, it does not write cgroup files directly. It asks systemd over D-Bus to create a transient scope for each container shim, and lets systemd populate the resource files. The OCI `linux.cgroupsPath` in this mode is a systemd path:

```
kubepods-besteffort.slice:cri-containerd:<container-id>
```

read as `<slice>:<prefix>:<id>`. The runtime creates a transient scope `cri-containerd-<id>.scope` under `kubepods-besteffort.slice/`.

The "cgroup driver" config in containerd, kubelet, and runc must agree. A common production failure: kubelet uses `systemd`, containerd uses `cgroupfs`, and pods fail to start because each side is trying to create cgroups the other does not see.

Delegation Enables Rootless

cgroup v2's delegation model is what gives a rootless container resource limits. Without v2 delegation, only the root user could write to cgroup files, and rootless containers would have no enforced limits. systemd creates a delegated subtree under `user.slice/user-<uid>.slice/user@<uid>.service/` and grants the user ownership:

```
ls -ld /sys/fs/cgroup/user.slice/user-1000.slice/user@1000.service
# drwxr-xr-x 2 1000 1000 ... user@1000.service
```

Inside the delegated subtree, the user can `mkdir`, write `+cpu +pids +memory` to `subtree_control` (subject to what systemd enabled at the boundary), and place processes — without root.

```
# As an unprivileged user:
mkdir /sys/fs/cgroup/user.slice/user-$(id -u).slice/user@$(id -u).service/me-demo
echo $$ > /sys/fs/cgroup/user.slice/user-$(id -u).slice/user@$(id -u).service/me-demo/cgroup.procs
```

Rootless podman, buildah, and rootless containerd put their containers under this kind of subtree.

Device Access Is BPF, Not Files

cgroup v2 does not have `devices.allow` or `devices.deny` files. Device access policy is enforced by an eBPF program of type `BPF_PROG_TYPE_CGROUP_DEVICE` attached to the cgroup. runc compiles the OCI `linux.resources.devices` list into a BPF program at container start.

To see the attached program on a running container's cgroup:

```
sudo bpftool cgroup tree | head
# CgroupPath
# ID       AttachType      AttachFlags     Name
# /sys/fs/cgroup/system.slice/docker-<id>.scope
#     12  cgroup_device                   <prog-name>
```

v1 exposed device policy as `devices.allow` and `devices.deny` files; v2 exposes nothing in the cgroup directory. Tooling that walked the v1 files has to load the BPF program with `bpftool cgroup show` instead.

OCI Resource Mapping

The relevant `linux.resources` fields in `config.json` and where they land:

OCI field	v2 file
`memory.limit`	`memory.max`
`memory.reservation`	`memory.low`
`memory.swap`	`memory.swap.max`
`cpu.shares`	`cpu.weight` (rescaled from 1024 → 100 default)
`cpu.quota` / `cpu.period`	`cpu.max`
`cpu.cpus` / `cpu.mems`	`cpuset.cpus` / `cpuset.mems`
`pids.limit`	`pids.max`
`blockIO.weight`, `throttleReadBpsDevice`, etc.	`io.weight`, `io.max`
`devices`	BPF program attached via `BPF_PROG_TYPE_CGROUP_DEVICE`

The remap from v1 names to v2 files is the runtime's job, not the user's. The OCI spec keeps the v1-style names for compatibility; runc, crun, and youki translate.

Where This Goes

The next chapter covers the rootfs: content addressing, snapshotters, and how runc gets a process to see a custom `/`. Cgroups reappear in chapter 7 when the device cgroup BPF program comes up under the security boundary.

Sources And Further Reading

- Linux cgroup v2 admin docs: https://kernel.org/doc/html/next/admin-guide/cgroup-v2.html
- `cgroups(7)`: https://man7.org/linux/man-pages/man7/cgroups.7.html
- systemd cgroup delegation: https://systemd.io/CGROUP_DELEGATION/
- PSI docs: https://docs.kernel.org/accounting/psi.html
- BPF cgroup device program: https://docs.kernel.org/bpf/prog_cgroup_device.html
- OCI runtime spec, Linux resources: https://github.com/opencontainers/runtime-spec/blob/main/config-linux.md#control-groups

Chapter 6: Container Filesystems

The container's filesystem is a three-stage pipeline: image content arrives as digest-addressed blobs, a snapshotter turns those blobs into a stack of directories, and runc swaps that stack in as `/` inside a fresh mount namespace.

> ***Safety:*** *the OverlayFS and `pivot_root` examples mutate kernel mount state and need root. The image-inspection examples are safe and read-only. Use a disposable Linux VM for the privileged sections. Examples were checked on Ubuntu 24.04 with kernel 6.8 and `containerd` 1.7.*

What An OCI Image Looks Like On Disk

An OCI image is a content-addressed bundle: one manifest, one image config, and a list of layers, all addressed by SHA-256 digest. The easiest way to see the structure is to pull an image into an OCI layout directory:

```
sudo apt-get install -y skopeo jq
skopeo copy docker://alpine:3.20 oci:./alpine-oci:3.20
ls alpine-oci/
# blobs/  index.json  oci-layout
ls alpine-oci/blobs/sha256/ | head
```

`oci-layout` is a marker file. `index.json` is the entry point — for a single-arch image, it points at one manifest; for a multi-arch image, it points at an image index that selects per `{architecture, os}`.

Walk the chain:

```
# index.json -> manifest digest
MANIFEST_DIGEST=$(jq -r '.manifests[0].digest' alpine-oci/index.json | sed 's/sha256://')
jq . alpine-oci/blobs/sha256/$MANIFEST_DIGEST

# manifest -> config digest and layer digests
jq '{config: .config.digest, layers: [.layers[].digest]}' \
  alpine-oci/blobs/sha256/$MANIFEST_DIGEST

# config -> rootfs.diff_ids
CONFIG_DIGEST=$(jq -r '.config.digest' alpine-oci/blobs/sha256/$MANIFEST_DIGEST | sed 's/sha256://')
jq '.rootfs' alpine-oci/blobs/sha256/$CONFIG_DIGEST
# {
#   "type": "layers",
#   "diff_ids": ["sha256:..."]
# }
```

Three digest types appear in image storage. Confusing them is the source of most layer-mismatch and snapshot-deduplication bugs:

- **manifest layer digest** — SHA-256 of the *compressed* layer bytes (gzip or zstd).
- **DiffID** — SHA-256 of the *uncompressed* tar. Listed in the image config.
- **ChainID** — recursive hash that identifies a layer plus its ancestors. Snapshotters key snapshots by ChainID, which is why two images that share a base get one snapshot, not two.

To verify a layer's DiffID by hand:

```
LAYER_DIGEST=$(jq -r '.layers[0].digest' \
  alpine-oci/blobs/sha256/$MANIFEST_DIGEST | sed 's/sha256://')
zcat alpine-oci/blobs/sha256/$LAYER_DIGEST | sha256sum
# This should match the diff_ids[0] entry in the image config.
```

If the registry's compressed-layer digest does not produce a tar whose uncompressed SHA-256 matches the config's DiffID, the snapshotter refuses to use the result.

Layer Tar Conventions

A layer is a tar archive with two encoded modifications:

- A file `<dir>/.wh.<name>` is a **whiteout**: it deletes `<name>` from any lower layer.
- A file `<dir>/.wh..wh..opq` is an **opaque marker**: lower layers' contents of `<dir>` are entirely hidden.

To see them in a real image, build one that deletes a file:

```
mkdir build && cat > build/Dockerfile <<'EOF'
FROM alpine:3.20
RUN rm /etc/motd
EOF
docker buildx build --output type=oci,dest=motd.tar build/
mkdir motd-oci && tar -C motd-oci -xf motd.tar

# Find the top layer of the new image.
MANIFEST=$(jq -r '.manifests[0].digest' motd-oci/index.json | sed 's/sha256://')
TOP_LAYER=$(jq -r '.layers[-1].digest' motd-oci/blobs/sha256/$MANIFEST | sed 's/sha256://')

# List its contents.
zcat motd-oci/blobs/sha256/$TOP_LAYER | tar -tvf - | grep -E '\.wh\.|motd'
# -rw-r--r-- 0/0 0 ... etc/.wh.motd
```

The zero-byte `etc/.wh.motd` is how `RUN rm /etc/motd` is encoded into a layer the snapshotter can apply.

containerd Content Store

When containerd pulls an image, it stores every blob in its content store, addressed by digest. The default location:

```
/var/lib/containerd/io.containerd.content.v1.content/
  blobs/sha256/<digest>
  ingest/<id>/
```

To see the content of a running containerd instance:

```
sudo ls /var/lib/containerd/io.containerd.content.v1.content/blobs/sha256/ | head
sudo ctr content ls | head
```

The content store treats every blob as opaque bytes addressed by digest and verifies them on read. The image service is what gives the bytes meaning: it tracks that `alpine:3.20` resolves to a particular manifest digest, which references a config and a list of layers.

Garbage collection treats content and snapshots together. A blob is reachable if some image, lease, or snapshot references it. Leases keep arbitrary content alive during in-flight operations:

```
sudo ctr leases ls
```

Snapshotters

The snapshotter turns image layers into a mountable filesystem. containerd defines an interface; plugins implement it. The default on Linux is the OverlayFS snapshotter.

```
sudo ctr snapshot ls | head
# KEY                                                                     PARENT
KIND
# sha256:abcd... (chainID for layer 0)
Committed
# sha256:efgh... (chainID for layer 0+1)                                  sha256:abcd...
Committed
# k8s.io/12/<container-id>                                                sha256:efgh...
Active
```

Three kinds of snapshot:

- **Committed** — immutable, named by ChainID, produced by unpacking one layer on top of another.
- **Active** — writable, used as the upper layer for a running container.
- **View** — read-only checkout of a committed chain, used by tools that just need to read.

The snapshotter's directory layout:

```
/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/
  metadata.db
  snapshots/<id>/fs/        # the layer's directory
  snapshots/<id>/work/      # OverlayFS work dir (only for active)
```

`fs/` is the data; `work/` is OverlayFS's scratch area for atomic operations, required to be on the same filesystem as the upper layer.

Build An Overlay By Hand

To see the OverlayFS internals that the snapshotter wraps, mount one yourself:

```
sudo mkdir -p /tmp/ov/{lower1,lower2,upper,work,merged}

# Lower1 = base layer
echo "from lower1" | sudo tee /tmp/ov/lower1/file-a > /dev/null
echo "lower1 only" | sudo tee /tmp/ov/lower1/file-b > /dev/null

# Lower2 = stacked above lower1 (right-most = top)
echo "from lower2 (overrides lower1)" | sudo tee /tmp/ov/lower2/file-a > /dev/null

# Upper = writable
sudo mount -t overlay overlay \
  -o lowerdir=/tmp/ov/lower2:/tmp/ov/lower1,upperdir=/tmp/ov/upper,workdir=/tmp/ov/work \
  /tmp/ov/merged

# What the merged view shows:
cat /tmp/ov/merged/file-a  # from lower2 (overrides lower1)
cat /tmp/ov/merged/file-b  # lower1 only

# Write to merged. Changes go to upper.
echo "new line" | sudo tee -a /tmp/ov/merged/file-a > /dev/null
ls /tmp/ov/upper/
# file-a
```

The `lowerdir` syntax reads right-to-left: the last directory is the bottom of the stack, the first is the top. Writes always land in `upperdir`.

To delete a lower-layer file from the merged view:

```
sudo rm /tmp/ov/merged/file-b
ls /tmp/ov/merged/
# file-a
ls -la /tmp/ov/upper/
# c--------- 0 0 ... file-b   <- character device, major:minor 0:0
```

OverlayFS represents whiteouts as character devices with `major:minor 0:0`, *not* as the OCI tar `.wh.<name>` convention. The unpacker translates between them at unpack time.

To clean up:

```
sudo umount /tmp/ov/merged
sudo rm -rf /tmp/ov
```

Rootless OverlayFS

`trusted.*` xattrs (used by OverlayFS for opaque markers and redirects) require `CAP_SYS_ADMIN` *on the host*. That means rootless OverlayFS does not work on older kernels. Linux 5.11 added support for OverlayFS in user namespaces; on older kernels, rootless tooling (rootless Docker, podman) falls back to `fuse-overlayfs` — a userspace reimplementation that uses `user.*` xattrs and accepts the performance cost.

To check which one is in use under a rootless engine:

```
podman info --format '{{.Store.GraphDriverName}}'
# overlay (kernel) or overlay (fuse-overlayfs)
```

Mount Setup In runc

When runc starts a container, it executes a fixed sequence inside a fresh mount namespace. Watching it from outside is awkward (the work happens between `clone(2)` and `execve(2)` of the user process); reading the kernel-side view is easier. After a container has started, look at its mounts via `/proc/<pid>/mountinfo`:

```
docker run --rm -d --name demo alpine:3.20 sleep 600
PID=$(docker inspect -f '{{.State.Pid}}' demo)
sudo cat /proc/$PID/mountinfo
# 1 0 0:34 / / rw,relatime master:1 - overlay overlay rw,lowerdir=...
# 2 1 0:35 / /proc rw,nosuid,nodev,noexec,relatime - proc proc ...
# 3 1 0:36 /pts /dev/pts rw,nosuid,noexec,relatime - devpts devpts ...
# ...
```

The first line is the rootfs — an OverlayFS mount with a lowerdir chain corresponding to the image's layers and an upperdir corresponding to the snapshotter's active snapshot. The rest are the standard runtime mounts: `/proc`, `/dev` as tmpfs, `/dev/pts`, `/dev/shm` as tmpfs, `/dev/mqueue`, `/sys`, and a read-only `cgroup2` bind.

```
docker stop demo
```

The order runc applies these is fixed because `pivot_root` has constraints:

1. Make the new mount namespace's root mount private to prevent propagation back to the host.
2. Mount the rootfs (OverlayFS or whatever the snapshotter returns).
3. Mount the special filesystems described in `config.json`'s `mounts` array.
4. Create device nodes (bind from host) and link standard ones (`/dev/stdin` → `/proc/self/fd/0`, etc.).
5. Apply `linux.maskedPaths` (bind `/dev/null` over them).
6. Apply `linux.readonlyPaths` (remount as read-only).

7. `pivot_root(2)` to swap the prepared rootfs in as `/`.
8. Unmount and remove the old root.
9. Set propagation on `/` per `linux.rootfsPropagation`.

The runc source for this lives in `libcontainer/rootfs_linux.go`; the OCI spec describes the requirements in `runtime-linux.md`.

`pivot_root` By Hand

To see what `pivot_root(2)` is doing without runc in the way:

```
sudo unshare --mount -- bash
# Inside the new mount namespace.

mkdir -p /tmp/newroot/{old,bin,proc,sys,dev}
mount -t tmpfs tmpfs /tmp/newroot       # placeholder rootfs
cp /bin/busybox /tmp/newroot/bin/        # adjust if no busybox; use bash etc.

# Make / private so pivot_root constraints are met.
mount --make-rprivate /

# Bind-mount the rootfs onto itself; pivot_root requires the new and
# old roots to be on different mounts.
mount --bind /tmp/newroot /tmp/newroot

cd /tmp/newroot
pivot_root . old

# After this, `.` is the new /; `/old` is the old root.
exec /bin/busybox sh
ls /old
# usr  etc  var  ... (the host's old root)

umount -l /old
ls /
# bin  old  proc  sys  dev  (now without /old)
```

This is exactly what runc does, with one important addition: runc unmounts old root *before* exec'ing the user process so the container cannot escape via the old root. The example above leaves it mounted for inspection.

OCI Mounts In Practice

The `mounts` array in `config.json` is what the runtime applies in step 3 above. Each entry has `destination`, `type`, `source`, and `options`. The conventional default set:

Destination	Type	Source	Notes
`/proc`	`proc`	`proc`	Reflects PID namespace. Required for `ps`, `/proc/self/`.
`/dev`	`tmpfs`	`tmpfs`	`mode=755`.
`/dev/pts`	`devpts`	`devpts`	`newinstance,ptmxmode=0666`.
`/dev/shm`	`tmpfs`	`shm`	`mode=1777,size=65536k`.
`/dev/mqueue`	`mqueue`	`mqueue`	Matches IPC namespace.
`/sys`	`sysfs`	`sysfs`	Often `ro` for non-privileged.
`/sys/fs/cgroup`	`cgroup2`	`cgroup`	Read-only bind, view limited by cgroup namespace.

Bind mounts (`type: bind` , `source: <host-path>`) are how host paths show up inside containers — Docker volumes, Kubernetes hostPath, and secret mounts all use them. The `options` field controls propagation: `rprivate` (default) does not propagate; `rslave` accepts host changes; `rshared` propagates both ways. Kubernetes' `mountPropagation: HostToContainer` corresponds to `rslave` ; `Bidirectional` corresponds to `rshared` .

To see a real container's mount config:

```
docker run --rm -d --name demo -v /tmp:/host-tmp alpine:3.20 sleep 600
PID=$(docker inspect -f '{{.State.Pid}}' demo)
sudo cat /proc/$PID/mountinfo | grep host-tmp
# .../tmp /host-tmp rw,relatime - ext4 /dev/...
docker stop demo
```

Where This Goes

The next chapter covers the security controls that compose with the filesystem to form the full container boundary — capabilities, seccomp, MAC, masked paths. The masked paths in particular layer on top of the mount setup described here: they are bind mounts performed after the OCI mount table is in place.

Sources And Further Reading

- OCI Image Specification: https://github.com/opencontainers/image-spec
- OCI image layout: https://github.com/opencontainers/image-spec/blob/main/image-layout.md
- OCI image layer format: https://github.com/opencontainers/image-spec/blob/main/layer.md
- containerd content flow: https://github.com/containerd/containerd/blob/main/docs/content-flow.md
- containerd snapshotter docs: https://containerd.io/docs/2.2/snapshotters/readme/
- Linux OverlayFS: https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html
- `pivot_root(2)` : https://man7.org/linux/man-pages/man2/pivot_root.2.html
- `mount(2)` : https://man7.org/linux/man-pages/man2/mount.2.html
- runc rootfs setup: https://github.com/opencontainers/runc/blob/main/libcontainer/rootfs_linux.go

Chapter 7: Security Boundaries

A container's security boundary is the assembled effect of several Linux subsystems — capabilities, seccomp, AppArmor or SELinux, user namespaces, `noNewPrivileges`, masked paths, and the device cgroup BPF program — each tunable independently.

> ***Safety:*** *most examples need root. The seccomp and capabilities demonstrations are harmless on a VM. The MAC examples assume the relevant LSM is already loaded and configured by the distribution. Use a disposable Linux VM. Examples were checked on Ubuntu 24.04 (AppArmor) and a Fedora 40 VM (SELinux).*

What The Controls Are For

Three threat classes shape what the controls are for:

1. **Container → host escape** — a process inside the container gaining privileges or visibility on the host.
2. **Container → container interference** — one container reading another's data, signaling its processes, or starving its resources.
3. **Container → external resource abuse** — a container performing actions outside its intended scope.

Capabilities and seccomp limit (1) and (3). MAC (AppArmor, SELinux) covers (1) and (2). User namespaces strengthen (1) at the cost of operational complexity. Cgroups address resource starvation in (2). No single control covers everything; the next sections cover each one, then the chapter returns to what `--privileged` actually loosens.

Linux Capabilities

Capabilities split the historical "root or not" model into ~40 individually grantable units. Setting capabilities is the runtime's job; reading them back is yours.

```
# What capabilities does the current shell have?
grep ^Cap /proc/self/status
# CapInh: 0000000000000000
# CapPrm: 0000000000000000
# CapEff: 0000000000000000
# CapBnd: 000001ffffffffff
# CapAmb: 0000000000000000
```

Five hex bitmaps, one per set: Inheritable, Permitted, Effective, Bounding, Ambient. The bounding set is the upper bound the process can ever hold; for an unprivileged shell it is "all caps known to this kernel" because a setuid root binary could push more in. For a runc-launched container the bounding set is restricted to the OCI `process.capabilities.bounding` list.

Decode the bitmaps with `capsh`:

```
capsh --decode=000001ffffffffff
# 0x000001ffffffffff=cap_chown,dac_override,...,cap_checkpoint_restore
```

To see the difference inside a container:

```
docker run --rm alpine:3.20 sh -c 'grep ^Cap /proc/self/status'
# CapInh: 0000000000000000
# CapPrm: 00000000a80425fb
# CapEff: 00000000a80425fb
# CapBnd: 00000000a80425fb
# CapAmb: 0000000000000000

docker run --rm alpine:3.20 sh -c '
  apk add -q libcap
  capsh --decode=$(grep ^CapBnd /proc/self/status | cut -f2)
'
# 0x00000000a80425fb = cap_chown,dac_override,fowner,fsetid,kill,setgid,
# setuid,setpcap,net_bind_service,sys_chroot,mknod,audit_write,setfcap
```

Thirteen capabilities — the conventional default container set. Notably absent: `CAP_SYS_ADMIN`, `CAP_NET_ADMIN`, `CAP_SYS_PTRACE`, `CAP_SYS_TIME`, `CAP_NET_RAW`, `CAP_SYS_MODULE`. The container's "root" cannot configure interfaces, load kernel modules, or set the clock.

Compare with `--privileged`:

```
docker run --rm --privileged alpine:3.20 sh -c 'grep ^CapBnd /proc/self/status'
# CapBnd: 000001ffffffffff   <- everything
```

`--privileged` clears the bounding set, drops the seccomp profile, removes the AppArmor / SELinux profile, and gives the device cgroup a wildcard rule. It is the easiest way to make a container "work," and it removes most of what made it a container.

File Capabilities

Capabilities can also live on executables as the `security.capability` xattr:

```
sudo apt-get install -y libcap2-bin
getcap -r /usr/bin /usr/sbin 2>/dev/null | head
# /usr/bin/ping cap_net_raw=ep
# /usr/bin/newuidmap cap_setuid+ep
# /usr/bin/newgidmap cap_setgid+ep
```

When `ping` is exec'd, its file capabilities become permitted+effective on the new process. This is how a non-root user can `ping` despite needing `CAP_NET_RAW` — the binary brings the capability with it. Inside a container with `noNewPrivileges` set, file capabilities are ignored on `execve`; setuid bits and `security.capability` xattrs both stop working as escalation vectors.

`noNewPrivileges`

A one-bit `prctl(2)` that, once set, prevents the process from gaining privileges via `execve`:

```
sudo apt-get install -y libcap2-bin

# A non-privileged shell. ping works because of file capabilities.
ping -c1 127.0.0.1 > /dev/null && echo "ping ok"
# ping ok

# Set no_new_privs, then exec ping. File capabilities are ignored.
exec setpriv --no-new-privs ping -c1 127.0.0.1
# ping: socktype: SOCK_RAW
```

`setpriv --no-new-privs` is the user-space wrapper around `prctl(PR_SET_NO_NEW_PRIVS, 1)`. Once set, the bit cannot be cleared. OCI containers default to it.

Seccomp

Seccomp filters syscalls. To watch it work, compile a tiny program that sets a filter and tries something it has just blocked. The shell-friendliest path is `prctl` via `setpriv` for the no-new-privs bit and a small Python program for the seccomp filter.

A C example using `libseccomp`:

```
sudo apt-get install -y libseccomp-dev gcc

cat > /tmp/seccomp-demo.c <<'EOF'
#include <seccomp.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_ALLOW);
    seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EPERM), SCMP_SYS(uname), 0);
    seccomp_load(ctx);
    seccomp_release(ctx);

    char buf[1024];
    if (gethostname(buf, sizeof buf) < 0) {
        perror("gethostname");
    } else {
        printf("hostname: %s\n", buf);
    }

    struct utsname u;
    if (uname(&u) < 0) {
        perror("uname");
    } else {
        printf("uname: %s\n", u.sysname);
    }
    return 0;
}
EOF

gcc /tmp/seccomp-demo.c -lseccomp -o /tmp/seccomp-demo
/tmp/seccomp-demo
# hostname: <something>
# uname: Operation not permitted
```

`gethostname(2)` is allowed, `uname(2)` is forced to return `EPERM`. The OCI spec's `linux.seccomp` field describes the same filter as JSON; runc compiles it to BPF before exec. Docker and containerd ship a default profile that blocks ~50 syscalls including `kexec_load`, `keyctl`, `add_key`, `init_module`, `mount`, `umount2`, `swapon`, `clock_settime`, and `reboot`. The full profile is at `containerd/contrib/seccomp/seccomp_default.go` (or `moby/profiles/seccomp/default.json` for the Docker copy).

To inspect the filter on a running container:

```
docker run --rm -d --name demo alpine:3.20 sleep 600
PID=$(docker inspect -f '{{.State.Pid}}' demo)
grep -E 'Seccomp|^Sec' /proc/$PID/status
# Seccomp:        2
# Seccomp_filters: 1
docker stop demo
```

`Seccomp: 2` is `SECCOMP_MODE_FILTER`. `Seccomp_filters: 1` is the number of attached BPF programs.

AppArmor (Ubuntu/Debian)

AppArmor is path-based MAC. Containers run inside a profile that the kernel enforces alongside DAC and capabilities.

```
# Confirm AppArmor is enabled.
sudo aa-status | head
# apparmor module is loaded.
# 70 profiles are loaded.

# Find the profile a running container is using.
docker run --rm -d --name demo alpine:3.20 sleep 600
PID=$(docker inspect -f '{{.State.Pid}}' demo)
sudo cat /proc/$PID/attr/current
# docker-default (enforce)
docker stop demo
```

Inside `docker-default`, writes to `/proc/sys`, `/proc/sysrq-trigger`, `/sys/kernel`, and most of `/sys` are denied. Mount operations are blocked except for the ones the runtime itself sets up before the profile attaches.

Try writing to a kernel parameter from inside a default-profile container:

```
docker run --rm alpine:3.20 sh -c 'echo 1 > /proc/sys/kernel/sysrq'
# sh: can't create /proc/sys/kernel/sysrq: Permission denied
```

Without AppArmor, this would be allowed if the container had `CAP_SYS_ADMIN` (it doesn't by default), or if the kernel parameter happened to be writable for non-root (it isn't here).

Per-pod AppArmor profiles in Kubernetes use `securityContext.appArmorProfile` (since 1.30, GA). The named profile must already be loaded on the node.

SELinux (RHEL/Fedora)

SELinux is label-based MAC. Every process and every file has a security context: `user:role:type:level`.

```
# On a Fedora/RHEL host with SELinux in enforcing mode:
getenforce
# Enforcing

# Process contexts.
ps -eZ | head
# system_u:system_r:init_t:s0   1 ?  init
# ...

# Container process context.
podman run --rm -d --name demo registry.access.redhat.com/ubi9/ubi-minimal sleep 600
PID=$(podman inspect -f '{{.State.Pid}}' demo)
sudo cat /proc/$PID/attr/current
# system_u:system_r:container_t:s0:c123,c456
podman stop demo
```

The process type `container_t` is the policy bucket for normal containers. The `:s0:c123,c456` suffix is **MCS** (Multi-Category Security): each container gets a unique pair of categories, and the policy permits access only when the categories of subject and object match. Two containers running as the same `container_t` cannot read each other's files because their MCS labels differ.

To see the file labels under a container's rootfs:

```
sudo ls -lZ /var/lib/containers/storage/overlay/<id>/diff/etc/ | head
# system_u:object_r:container_file_t:s0:c123,c456 ...
```

`container_file_t` is the policy bucket for container-managed files. The MCS pair matches the process's, which is what makes the access decision come out "allowed."

When SELinux denies an action that DAC and capabilities would allow, the audit log records it:

```
sudo ausearch -m AVC -ts recent | tail
# type=AVC msg=audit(...): avc:  denied  { read } for  pid=...
#   scontext=...:container_t:s0:c123,c456
#   tcontext=...:container_file_t:s0:c789,c012
#   tclass=file
```

An AVC denial line tells you which container (the categories), which kind of object (the type), and which permission the policy was missing (the action).

Distros ship either AppArmor or SELinux, not both. The kernel's LSM framework can stack additional minor LSMs (Yama, Lockdown, BPF-LSM) alongside, but the path-vs-label MAC layer is one or the other.

User Namespaces As A Security Boundary

User namespaces let a process hold root-equivalent capabilities inside the namespace without holding any host-level privilege. The chapter on namespaces showed how to create one; here is what the security implication looks like.

```
# Inside a user namespace where I am "root":
unshare --user --map-root-user -- bash -c '
  # Try to mount the host /proc.
  mount -t proc proc /mnt 2>&1
  # mount: /mnt: permission denied. (only privileged user can mount)

  # But create a new mount namespace - mount in there works:
  unshare --mount -- bash -c "mount -t tmpfs tmpfs /mnt && echo mounted"
  # mounted
'
```

The `CAP_SYS_ADMIN` the inner shell holds applies *to resources owned by its user namespace*. Mount namespaces created from inside that user namespace are owned by it; the host's mount namespace is not. A compromise that escalates to root inside such a container lands at an unprivileged host UID instead of host root — the threat-model improvement rootless containers exist for.

Kubernetes has `spec.hostUsers: false` (beta in 1.30, on track for GA) to give every pod its own user namespace. Inside, the pod's processes appear to run as their requested UIDs; outside, those UIDs map to a high-numbered host UID range (e.g. 100000-165535). A compromise that escalates to "root" inside the pod gets host UID 100000, which on the host is unprivileged.

Caveats:

- The rootfs has to be `chown`-ed to match the namespace's mapping, or **idmap mounted** so the kernel performs the translation at access time.
- File capabilities, ACLs, and `setuid` binaries on shared filesystems still run as the namespace-mapped UID, which is usually fine but occasionally surprising.

Masked And Read-Only Paths

`/proc` and `/sys` aggregate host-wide information that the PID and mount namespaces do not isolate. Two OCI fields plug the leaks:

- `linux.maskedPaths` — runc implements this by bind-mounting `/dev/null` over files and an empty `tmpfs` over directories. The OCI spec requires the path be inaccessible, not the specific mechanism.
- `linux.readonlyPaths` — remount the path read-only.

Default masked set in most runtimes:

```
/proc/asound
/proc/acpi
/proc/kcore
/proc/keys
/proc/latency_stats
/proc/timer_list
/proc/timer_stats
/proc/sched_debug
/proc/scsi
/sys/firmware
/sys/devices/virtual/powercap
```

Default read-only:

```
/proc/bus
/proc/fs
/proc/irq
/proc/sys
/proc/sysrq-trigger
```

To verify:

```
docker run --rm alpine:3.20 sh -c 'cat /proc/kcore' 2>&1 | head
# (no output -- /dev/null is mounted over it)
echo "exit code: $?"

docker run --rm alpine:3.20 sh -c 'echo 1 > /proc/sysrq-trigger' 2>&1
# sh: can't create /proc/sysrq-trigger: Read-only file system
```

`/proc/kcore` is masked because it would otherwise let a process with `CAP_SYS_RAWIO` read kernel memory. `/proc/sysrq-trigger` is read-only because writes to it can crash, reboot, or sync the host.

Each entry was added in response to a public disclosure — for example, `/proc/sched_debug` was masked after it was shown to leak kernel pointers. Any new `/proc` or `/sys` interface that exposes host state is a candidate.

Device Access (cgroup v2 + eBPF)

cgroup v2 has no `devices.allow` file. Device policy is enforced by an eBPF program of type `BPF_PROG_TYPE_CGROUP_DEVICE` attached to the cgroup. runc compiles the OCI `linux.resources.devices` list into BPF and attaches it.

```
docker run --rm -d --name demo alpine:3.20 sleep 600
CGROUP=$(docker inspect -f '{{.HostConfig.CgroupParent}}/docker-{{.Id}}.scope' demo)

sudo bpftool cgroup tree /sys/fs/cgroup$CGROUP 2>/dev/null || \
  sudo bpftool cgroup tree | grep -A1 "$(docker inspect -f '{{.Id}}' demo | head -c12)"
# /sys/fs/cgroup/.../docker-<id>.scope
# ID  AttachType      AttachFlags     Name
# X   cgroup_device                   sd_devices
docker stop demo
```

Try to access a device that is not in the allow list:

```
docker run --rm alpine:3.20 sh -c '
  cat /dev/null > /dev/null  # allowed
  echo "null ok"
  cat /dev/sda 2>&1 | head -1  # not allowed
'
# null ok
# cat: /dev/sda: Operation not permitted
```

The kernel returns `EPERM` because the BPF program denies the open. Privileged containers attach a BPF program that allows everything (`a *:* rwm`).

Putting It Together

A non-privileged container's actual boundary, in the order it is built:

1. **Namespaces** create separate views.
2. **Mount setup** with masked and read-only paths closes `/proc` and `/sys` leaks.
3. **Capability bounding set** strips kitchen-sink privileges.
4. `noNewPrivileges` prevents privilege gain across exec.
5. **Seccomp** filters dangerous syscalls.
6. **AppArmor or SELinux** denies operations that DAC and capabilities would allow.
7. **Cgroup device BPF** restricts which devices work.
8. **Cgroups** bound resource consumption.
9. **User namespace mapping** (when enabled) makes container root unprivileged on the host.

Each layer is independently configurable. `--privileged` clears most of these layers at once, which is why it is rarely the right answer when a container does not work.

Where This Goes

Part 3 picks up the OCI runtime side: how an OCI bundle is laid out, what `config.json` looks like in detail, and how runc translates the spec into the kernel state we have just spent four chapters cataloguing.

Sources And Further Reading

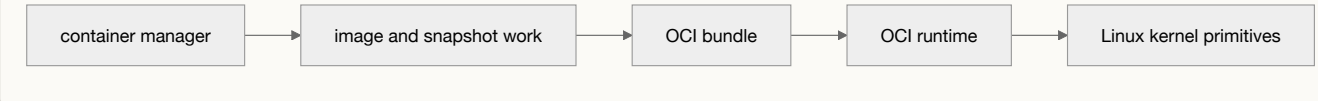
- `capabilities(7)`: https://man7.org/linux/man-pages/man7/capabilities.7.html
- `seccomp(2)`: https://man7.org/linux/man-pages/man2/seccomp.2.html
- Linux seccomp_filter docs: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html
- AppArmor: https://gitlab.com/apparmor/apparmor/-/wikis/home
- SELinux project: https://github.com/SELinuxProject
- container-selinux: https://github.com/containers/container-selinux
- `prctl(2)` `PR_SET_NO_NEW_PRIVS`: https://man7.org/linux/man-pages/man2/prctl.2.html
- OCI runtime spec, capabilities: https://github.com/opencontainers/runtime-spec/blob/main/config.md#capabilities
- OCI runtime spec, seccomp: https://github.com/opencontainers/runtime-spec/blob/main/config-linux.md#seccomp
- Default Docker seccomp profile: https://github.com/moby/moby/blob/master/profiles/seccomp/default.json
- containerd seccomp default: https://github.com/containerd/containerd/blob/main/contrib/seccomp/seccomp_default.go
- BPF cgroup device program: https://docs.kernel.org/bpf/prog_cgroup_device.html

PART III — OCI AND RUNC

Chapter 8: OCI Runtime Bundles

An OCI runtime bundle is the local handoff between a higher-level manager and a low-level runtime: a directory containing `config.json` and the root filesystem the runtime will use as `/`.

By the time runc sees a bundle, containerd or another manager has already done the image work. Registry resolution, content download, layer unpacking, snapshot preparation, and mount calculation happen above the OCI runtime boundary. The runtime receives the result as local filesystem state plus a JSON spec.



containerd, CRI-O, Docker, and other managers prepare filesystem and metadata state in their own way, then hand a common bundle to runc, crun, youki, runsc, Kata, or another compatible implementation.

Bundle Layout

The OCI runtime spec defines a bundle as a directory with `config.json` at the root. The root filesystem is referenced by `root.path`; the spec does not require the rootfs directory to have one universal name, though `rootfs` is conventional.

The practical shape is:

```
bundle/
  config.json
  rootfs/
    bin/
    etc/
    proc/
    ...
```

`config.json` is the contract. It describes the process, root filesystem, mounts, namespaces, cgroups, hooks, annotations, and platform-specific settings. The root filesystem is the tree that should become `/` inside the container after the runtime sets up the mount namespace and switches root.

The top-level Go type generated for the runtime spec is a useful map of that contract:

```
type Spec struct {
    Version string `json:"ociVersion"`
    Process *Process `json:"process,omitempty"`
    Root *Root `json:"root,omitempty"`
    Mounts []Mount `json:"mounts,omitempty"`
    Hooks *Hooks `json:"hooks,omitempty" platform:"linux,solaris,zos"`
    Annotations map[string]string `json:"annotations,omitempty"`
    Linux *Linux `json:"linux,omitempty" platform:"linux"`
}
```

The OCI runtime spec is the authoritative document. Most implementations vendor the `runtime-spec/specs-go` types, so configs that round-trip through one tool tend to round-trip through the others.

Process And Root

The `process` object describes the program to execute. It includes `args`, `env`, `cwd`, user and group settings, capabilities, rlimits, terminal settings, `noNewPrivileges`, AppArmor and SELinux labels, scheduler fields, I/O priority, and CPU affinity. A minimal one looks like this:

```
"process": {
  "args": ["/bin/sh", "-c", "echo hello"],
  "env": ["PATH=/usr/bin"],
  "cwd": "/",
  "noNewPrivileges": true
}
```

The `root` object names the container root filesystem. `root.path` is interpreted relative to the bundle unless it is absolute. `root.readonly` asks the runtime to make the root filesystem read-only after setup; individual mounts still carry their own options.

```
"root": { "path": "rootfs", "readonly": false }
```

The runtime creates the namespace and mount state that make `process` and `root` true.

Mounts

The `mounts` array is ordered. That matters because later mounts can cover earlier paths. Each mount has a destination, type, source, and options:

```
"mounts": [
  { "destination": "/proc", "type": "proc", "source": "proc" },
  { "destination": "/data", "type": "bind", "source": "/var/data",
    "options": ["rbind", "ro"] },
  { "destination": "/tmp",  "type": "tmpfs", "source": "tmpfs",
    "options": ["mode=1777", "size=64m"] }
]
```

The three entries share one JSON shape and produce three very different kernel calls: a `proc` mount, a host-path bind, and a fresh tmpfs. A bind mount source can be absolute or relative to the bundle.

The spec defines the desired mount table without prescribing how the runtime achieves it. A runtime may use classic `mount(2)`, the file-descriptor mount API (`fsopen(2)`, `fsmount(2)`, `move_mount(2)`), idmapped mounts via `mount_setattr(2)`, or bind-mount fallbacks, depending on kernel support and user-namespace mode.

Linux Namespaces

The Linux-specific `namespaces` list names namespace types such as mount, PID, network, UTS, IPC, user, cgroup, and time. For each namespace type, the meaning of `path` is the key:

OCI namespace entry	Runtime behavior
Type present with no `path`	Create a new namespace of that type.
Type present with `path`	Join the namespace at that path, usually via `setns(2)`.
Type absent	Inherit the runtime's namespace of that type.

Duplicate namespace types are invalid.

`linux.namespaces` does not isolate anything by itself. It instructs the runtime which namespaces to create or join when it builds the container process — the kernel calls from Part II happen at that step, not at JSON-decode time.

Cgroups And Resources

`linux.cgroupsPath` describes where the container should live in the cgroup hierarchy. `linux.resources` describes controller settings: CPU, memory, block I/O, pids, hugepages, RDMA, device rules, and cgroup v2 `unified` settings not otherwise modeled by the spec.

The spec does not mandate one cgroup manager. A runtime can write cgroupfs directly, use systemd, or combine approaches depending on host configuration. For cgroup v2, delegation rules matter: ownership and writable files are constrained by the kernel's delegation model.

If a limit is expressed correctly in `config.json` but not visible in `/sys/fs/cgroup`, the failure is in the runtime's cgroup application path or the host manager interaction, not in the bundle format itself.

Hooks

Hooks run at defined points around container setup, each in a specific namespace context.

The current runtime spec defines:

- `createRuntime` - runs in the runtime namespace after environment creation and before `pivot_root` or an equivalent root switch.
- `createContainer` - runs in the container namespace after namespace setup and before `pivot_root` or equivalent.
- `startContainer` - runs in the container namespace before the user command executes.
- `poststart` - runs in the runtime namespace after the user process starts.
- `poststop` - runs in the runtime namespace after the container is deleted.

A hook entry is a process invocation, nothing more:

```
"hooks": {
  "createRuntime": [
    { "path": "/usr/local/bin/setup-net",
      "args": ["setup-net", "--bridge", "br0"],
      "timeout": 5 }
  ]
}
```

`prestart` is deprecated but still appears in implementations for compatibility. Hooks are a common place for device injection and runtime extensions: `nvidia-container-runtime` is a `createRuntime` hook that injects GPU device nodes and library bind mounts before runc switches root. Each hook is a single exec at a defined lifecycle point — no plugin discovery, no shared state.

Runtime State

The OCI runtime also has a state JSON format. `state` reports fields such as `ociVersion`, `id`, `status`, `pid`, `bundle`, and `annotations`. The core statuses are `creating`, `created`, `running`, and `stopped`.

The lifecycle verbs use that state model:

- `create` prepares the container environment and leaves the user-specified program not yet executed.
- `start` runs the user-specified program.
- `state` reports status.
- `kill` sends a signal.
- `delete` removes runtime state after the container is stopped.

The `create` / `start` split exists so a caller can do work between setup and execution: attach IO, pass file descriptors, run hooks, or coordinate external namespace setup.

What The Spec Leaves Open

The OCI runtime spec defines the bundle and lifecycle, not a particular process tree. runc, crun, youki, gVisor, and Kata can all keep an OCI-facing shape while making different implementation choices.

Those choices include:

- whether the runtime uses `pivot_root(2)`, `chroot(2)`, `MS_MOVE`, or newer mount strategies for root switching;
- whether cgroups are applied through cgroupfs or systemd;
- how the runtime orders `clone(2)`, `clone3(2)`, `unshare(2)`, and `setns(2)`;
- how mounts are realized on older kernels, in user namespaces, or with idmapped mount support;
- whether the isolation boundary is direct host Linux primitives, a userspace kernel, or a VM.

Chapter 9 follows runc's choices through this list.

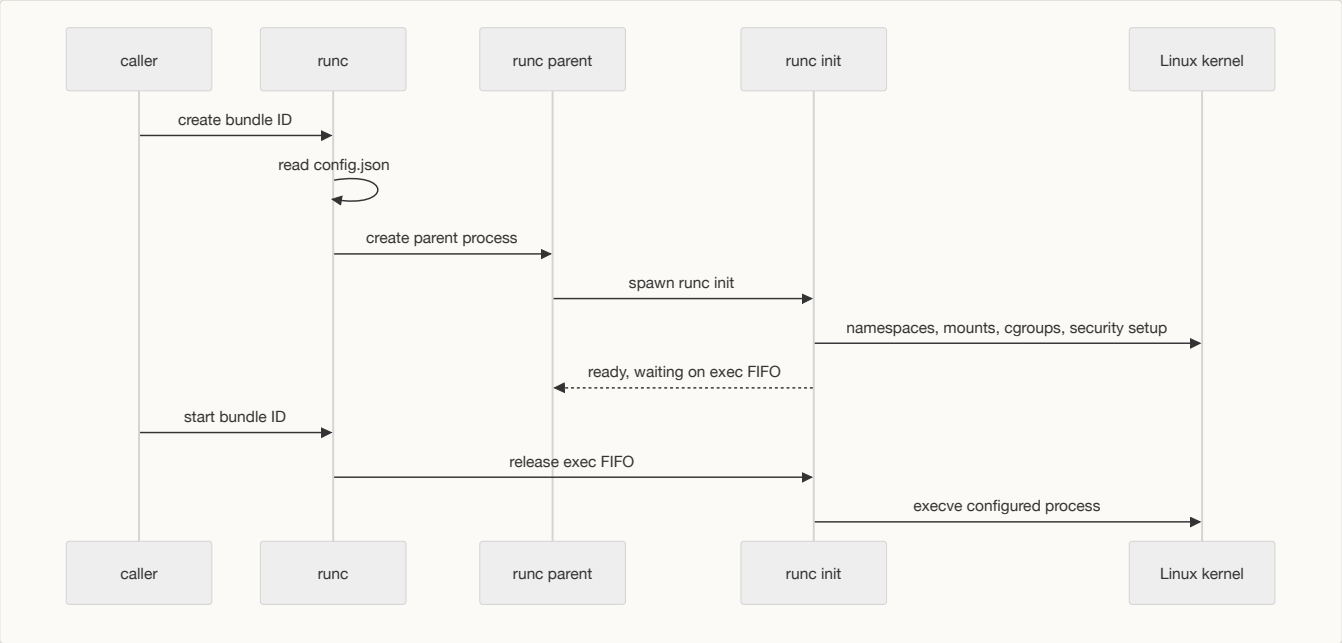
Sources And Further Reading

- OCI runtime bundle: https://github.com/opencontainers/runtime-spec/blob/6999a89a76a0329f440d5740497bedb9dd431297/bundle.md
- OCI runtime lifecycle: https://github.com/opencontainers/runtime-spec/blob/6999a89a76a0329f440d5740497bedb9dd431297/runtime.md
- OCI config: https://github.com/opencontainers/runtime-spec/blob/6999a89a76a0329f440d5740497bedb9dd431297/config.md
- OCI Linux config: https://github.com/opencontainers/runtime-spec/blob/6999a89a76a0329f440d5740497bedb9dd431297/config-linux.md
- OCI Go structs: https://github.com/opencontainers/runtime-spec/blob/6999a89a76a0329f440d5740497bedb9dd431297/specs-go/config.go
- `mount(2)`: https://man7.org/linux/man-pages/man2/mount.2.html
- `mount_setattr(2)`: https://man7.org/linux/man-pages/man2/mount_setattr.2.html
- `clone(2)`: https://man7.org/linux/man-pages/man2/clone.2.html
- `setns(2)`: https://man7.org/linux/man-pages/man2/setns.2.html
- `pivot_root(2)`: https://man7.org/linux/man-pages/man2/pivot_root.2.html

Chapter 9: runc Lifecycle

runc is containerd's default Linux runtime and the most common implementation of the OCI runtime contract. It takes the bundle from Chapter 8, turns `config.json` into libcontainer configuration, creates the requested Linux environment, and eventually calls `execve` for the configured process.

runc coordinates a parent process, an init process, namespace entry, cgroup placement, mount setup, hooks, seccomp, capabilities, labels, IO, state files, and cleanup. It is one implementation of the OCI runtime spec; crun, youki, runsc, and Kata are others.



`create` builds processes, namespaces, mounts, and cgroups but stops at the FIFO gate before `execve`; `start` releases the gate.

CLI Verbs

runc exposes the OCI lifecycle directly:

- `run` creates and starts a container in one command.
- `create` creates the container environment and leaves the configured program gated.
- `start` starts a container in the `created` state.
- `state` emits OCI state JSON.
- `kill` sends a signal.
- `delete` tears down a stopped or forcibly killed container.

The `run` command is convenient for humans. Container managers such as containerd use the two-phase `create` / `start` shape because it gives them a setup point between environment creation and process execution.

Reading The Bundle

runc starts by entering the bundle directory, opening `config.json`, decoding it into the OCI Go `Spec`, validating the process section, and converting the spec into libcontainer configuration.

The handoff from JSON to libcontainer is small:

```
if err = json.NewDecoder(cf).Decode(&spec); err != nil {
    return nil, err
}
return spec, validateProcessSpec(spec.Process)
```

From there runc translates the spec into libcontainer's model: namespaces, mounts, cgroups, devices, process settings, hooks, and runtime flags such as systemd cgroup mode, rootless mode, and no-pivot behavior.

The Parent And The Gate

For an init container, runc creates an exec FIFO before starting the container path:

```
if process.Init {
    if err := c.createExecFifo(); err != nil {
        return err
    }
}
```

The parent process then starts a cloned copy of `/proc/self/exe` running `runc init`. That init path receives pipes, bootstrap data, the init config, logging descriptors, and namespace setup instructions. runc marks unrelated file descriptors close-on-exec before starting `runc init`, a hardening detail added because leaked descriptors have produced container escapes in the past (CVE-2024-21626).

The FIFO is the gate. During `runc create`, the init process prepares the environment and waits. During `runc start`, runc releases that gate so the init path can continue to the final user process.

That gate is why `created` and `running` are different states: in `created`, namespaces, mounts, and cgroups exist and the init process is parked; the user-specified `execve` has not happened yet.

Namespace Entry

runc includes C namespace-entry code because namespace operations are sensitive to process and thread state. `setns(2)`, PID namespace creation, and clone ordering do not fit cleanly into an already-running multithreaded Go runtime.

The parent starts `runc init`, the C `nsenter` code handles low-level namespace entry and clone staging, and the Go init code reads `_LIBCONTAINER_*` environment variables and init config from pipes. From there the init path chooses standard init for a new container or setns init for `runc exec`.

crun and youki use different internal structures to satisfy the same OCI contract.

Root Filesystem Setup

The standard Linux init path prepares networking and routes, initializes labeling state, then prepares the root filesystem:

```
if err := setupNetwork(l.config); err != nil {
    return err
}
err := prepareRootfs(l.pipe, l.config)
```

`prepareRootfs` is where the bundle's root and mount declarations become a mount table inside the container's mount namespace. runc opens the rootfs, iterates configured mounts, creates device nodes when needed, sets up `/dev/ptmx` and `/dev` symlinks, runs parent-side hooks at the correct point, and switches root:

```
for _, m := range config.Mounts {
    if err := setupAndMountToRootfs(pipe, config, mountConfig, m); err != nil {
        return err
    }
}
err = pivotRoot(rootFd)
```

The full code path adds hardening around `/proc`, `/sys`, user-namespace device behavior, and read-only remounts. The mount list in `config.json` becomes kernel mount state, then `pivot_root(2)`, `MS_MOVE`, or `chroot(2)` swaps the prepared tree in as `/`.

Setup Order

runc's parent and init processes synchronize because setup order matters. The parent can apply cgroups before children escape placement, move configured network interfaces after it knows the child PID, run `prestart` and `createRuntime` hooks from the parent side, and pass file descriptors to the child. The child prepares the rootfs, runs container-side hooks, applies user and group settings, labels, capabilities, `noNewPrivileges`, seccomp, scheduler settings, I/O priority, and cwd checks close to the final exec.

That order is security-sensitive. A seccomp filter installed too early can block setup calls. A capability dropped too late gives more privilege to setup code than intended. A cwd outside the container root can become a host filesystem exposure.

Start, State, Kill, Delete

`start` only operates on a created container. It releases the exec FIFO so the init path can call `execve` for the configured program. `state` reports the OCI state fields from runc's stored state and live process information.

`kill` has more policy than a raw `kill(2)`. runc has special handling for `SIGKILL`, stopped and running states, cgroup process killing, and cases where the container does not have a private PID namespace. `delete --force` kills before teardown and must handle processes that may remain in the cgroup after the init process exits, especially when PID namespaces are shared.

A runtime owns the lifecycle state and teardown rules, not just the start path: `delete --force` reaps stragglers in the cgroup, and shared PID namespaces require killing the whole tree.

Other Runtime Answers

runc is the book's main implementation path, but the OCI contract allows different answers:

Runtime	What stays the same	What changes
crun	OCI bundle and lifecycle	C implementation and `libcrun` library-oriented design
youki	OCI bundle and lifecycle	Rust implementation and Rust abstractions for process, rootfs, cgroups, seccomp
gVisor `runsc`	OCI-facing command shape	Workload syscalls go through gVisor's userspace Sentry
Kata Containers	containerd/OCI-facing manager boundary	Workload runs inside a lightweight VM and guest agent

Part IV moves back up the stack to containerd, where image content, snapshots, container metadata, tasks, shims, and CRI all meet before the runtime ever receives a bundle.

Sources And Further Reading

- runc repository at checked commit: https://github.com/opencontainers/runc/tree/eb7eaf19b6eec5d1143b257057899e4a7b738c81
- runc CLI commands: https://github.com/opencontainers/runc/tree/eb7eaf19b6eec5d1143b257057899e4a7b738c81

- runc `config.json` loading: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/spec.go
- runc libcontainer setup: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/libcontainer/container_linux.go
- runc parent process sync: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/libcontainer/process_linux.go
- runc init dispatch: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/libcontainer/init_linux.go
- runc standard init: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/libcontainer/standard_init_linu
- runc rootfs setup: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/libcontainer/rootfs_linux.go
- runc nsenter C source: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/libcontainer/nsenter/nsexec.c
- OCI runtime lifecycle: https://github.com/opencontainers/runtime-spec/blob/6999a89a76a0329f440d5740497bedb9dd431297/runtime.md

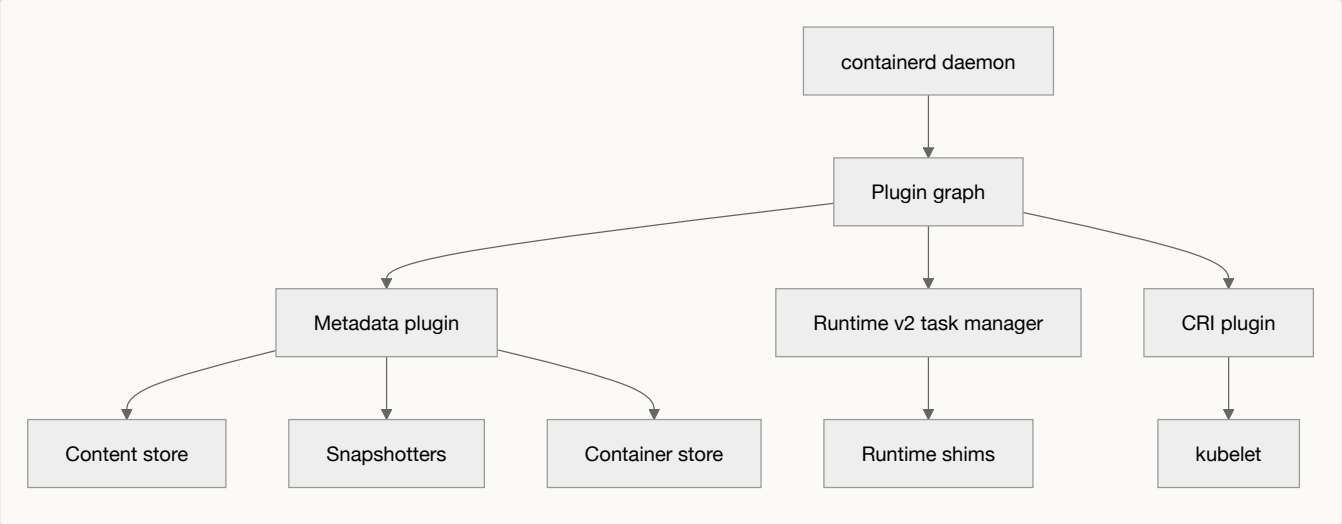
PART IV — CONTAINERD

Chapter 10: containerd Architecture

containerd is a daemon that hosts a plugin graph: content store, snapshotters, metadata, runtime v2, CRI, transfer, events, and services are all plugins, and clients reach each of them through a gRPC API.

To do that job it has to keep several kinds of state coordinated without collapsing them into one object: image references and descriptors, immutable content blobs, unpacked filesystem snapshots, persistent container metadata, live tasks, runtime shims, leases that protect work from garbage collection, and events that report what changed.

Those nouns carry weight for the rest of Part IV. A containerd *container* is a metadata object. A *task* is live execution. A *snapshot* is filesystem state managed by a snapshotter. *Content* is digest-addressed bytes, not a mounted root filesystem.



The Daemon As Plugin Host

The containerd daemon loads each configured plugin at startup, passing in a context with the root and state directories, daemon addresses, and the results of plugins it depends on. The startup loop in `cmd/containerd/server/server.go` is three lines of substance:

```
result := p.Init(initContext)
instance, err := result.Instance()
s.plugins = append(s.plugins, result)
```

Required plugins are tracked, so a missing core dependency fails startup instead of producing a half-wired daemon. Content, snapshotters, metadata, runtime v2, CRI, transfer, events, and services all show up as nodes in this graph.

The metadata plugin is the best anchor. It depends on content, events, and snapshots; at startup it opens `meta.db`, collects registered snapshotters, and builds a metadata database over the content store and snapshotter map. Images, containers, snapshots, and content can then be reasoned about together without being forced into one physical backend.

The metadata database stores names, records, labels, relationships, and namespace-scoped state. The content store stores blobs. Snapshotters own filesystem snapshot state. On disk under `/var/lib/containerd/`, the three stores live side by side:

```
io.containerd.metadata.v1.bolt/meta.db
io.containerd.content.v1.content/blobs/sha256/
io.containerd.snapshotter.v1.overlayfs/
```

Namespaces Are Metadata Partitions

containerd namespaces are not Linux namespaces. They do not call `unshare(2)` or `clone(2)` and they do not create process, mount, or network namespaces. They partition containerd's own metadata so one daemon can serve multiple consumers without mixing images, containers, leases, and snapshots.

CRI uses the `k8s.io` namespace. `ctr` defaults to `default`. Other clients pick their own. A single daemon can hold Kubernetes-managed objects and `ctr`-created objects at the same time, with each client seeing only the namespace it asks for.

A Linux PID namespace changes what processes can see. A containerd namespace changes where records are stored and looked up inside containerd. The two travel together by convention but not by mechanism: a process can run with no new Linux namespace and still belong to a containerd namespace, and a process can run inside many Linux namespaces while its metadata lives under `k8s.io`.

The Smart Client Model

containerd deliberately leaves higher-level work to clients. The plugin documentation calls this a smart-client model: if a step does not need to live in the daemon, the client does it before asking a service to do anything.

`ctr`, `nerdctl`, Docker, and CRI each resolve image names, choose snapshotters, build an OCI spec, attach labels, select runtime options, and pick a namespace before calling containerd. The same daemon serves all of them; it never sees how they differ.

The CRI plugin is a special case in that it runs inside the daemon, but it follows the same model: it translates Kubernetes CRI calls into containerd service calls and runtime choices. CRI is not a second runtime below containerd.

Core Services

The useful axis for learning containerd services is lifecycle responsibility:

Service	Responsibility
Content	Store immutable blobs and active ingestions by digest.
Images	Map names to OCI descriptor targets and keep image metadata.
Snapshots	Create active, view, and committed filesystem snapshots.
Containers	Store persistent container metadata: spec, image, runtime, snapshot key, labels, extensions.
Tasks	Manage live execution through runtime plugins and shims.
Leases	Protect content, snapshots, and metadata while work is in progress.
Events	Publish daemon and runtime lifecycle events.

The rows compose: image metadata points at content; unpacking content creates snapshots; a container record points at an image and a snapshot key; a task uses the container record to start live execution through runtime v2; a lease holds intermediate objects together while a pull or unpack is in flight; events tell observers what happened after requests return.

Keeping those responsibilities separate is what lets a single daemon serve several modes at once. An image can be pulled but not unpacked. A container can exist with no running task. A task can exit while the container metadata stays. A snapshot can outlive the image name that originally produced it, because a container or lease still references the chain.

Runtime v2 In The Architecture

Runtime v2 is the boundary between containerd's task service and runtime-specific process supervision. containerd starts or reconnects to a shim and talks to that shim over the runtime v2 task API. The common Linux path is `io.containerd.runc.v2`, implemented by the `containerd-shim-runc-v2` binary, which in turn drives `runc`.

The daemon never embeds the details of any specific runtime. The shim owns the runtime-specific work — invoking the OCI runtime, tracking exits, handling IO, publishing task events. That is why containerd can be restarted while existing tasks keep running under their shims, and why a different runtime can be slotted in by configuration alone.

The same boundary is why containerd has both synchronous calls and asynchronous events. A client calls `Start` and gets a response immediately, but the task's later exit arrives as an event. Runtime v2 defines task create, start, exit, delete, pause, resume, checkpoint, OOM, and exec events with ordering guarantees. A caller that only reads request responses misses task exits, OOM events, and any state change that happens after the synchronous call returns.

Where CRI Fits

The CRI plugin is a gRPC plugin inside containerd. It registers Kubernetes `RuntimeService` and `ImageService` servers on containerd's gRPC server and maps kubelet requests onto containerd services.

Kubelet gets a Kubernetes-shaped API and never has to know about containerd's image store, snapshotters, task service, leases, event monitor, or runtime v2 shims. It asks for a pod sandbox or a container start. The CRI plugin turns that into namespace-scoped containerd operations under `k8s.io`.

Four contracts stack on top of each other in the runtime path:

1. CRI sits between kubelet and containerd.
2. containerd services sit inside the daemon's own API surface.
3. Runtime v2 sits between containerd and the shim.
4. The OCI runtime spec sits between the shim and a runtime such as `runc`.

Each layer has its own vocabulary, and a word from one layer rarely means the same thing in another. The CRI runtime vs OCI runtime distinction from chapter 1 is the standing example.

Where This Goes

The rest of Part IV walks one object at a time across this graph. Chapter 11 takes image bytes from a reference into the content store and out through a snapshotter. Chapter 12 turns a container record into a task and a shim. Chapter 13 enters from the kubelet side and lands in the same containerd services.

Sources And Further Reading

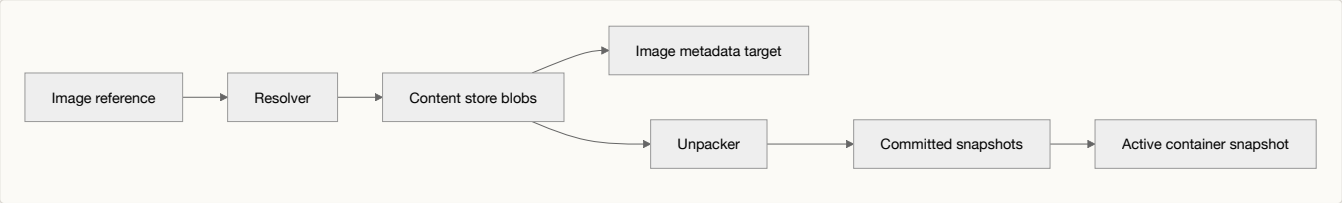
- containerd `v2.3.0` source: https://github.com/containerd/containerd/tree/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a
- Daemon plugin initialization: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/cmd/containerd/server/server.
- Metadata plugin: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/plugins/metadata/plugin.go
- Metadata DB: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/metadata/db.go
- Plugin docs: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/docs/PLUGINS.md
- Feature docs: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/docs/features.md
- Runtime v2 docs: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/docs/runtime-v2.md

Chapter 11: Images, Content, And Snapshots

An image pull in containerd is four operations on four different things: name resolution, content download, image-metadata recording, and (optionally) layer unpacking through a snapshotter.

The four things are an **image reference**, the registry-style name a user types; **content blobs**, the digest-addressed bytes that make up manifests, configs, and layers; **image metadata**, which maps a name to an OCI descriptor target; and **snapshots**, which are filesystem states produced by applying layers in order.

The reason to keep them apart is that containerd can be in any of the partial states between them. Image metadata can exist without all the content being present. Content can exist without a mounted root filesystem. A snapshot chain can exist without a running process. A running task's writable layer is a separate object from the committed image layers it sits on top of.



The Content Store Is For Bytes

The content store is digest-addressed storage for immutable blobs: manifests, indexes, image configs, and compressed layers.

The interface in `core/content/content.go` is small enough to fit on one screen:

```
type Store interface {
    Manager
    Provider
    Ingester
}
```

`Provider` reads committed content by descriptor. `Ingester` handles active writes. `Manager` exposes metadata operations such as labels. An ingestion is invisible to readers until it is committed; after commit, callers can read the blob by digest and inspect or update its labels.

That commit boundary is the reason image pulls need leases. While a pull is running, containerd carries temporary blobs, partially traversed descriptors, and metadata that the garbage collector would happily delete if it ran in the gap. The pull path wraps the whole operation in a lease so the collector and the pull cannot race.

Image Metadata Is A Name To A Descriptor

The image store is not the image bytes. It maps a name to an OCI descriptor target — usually an image index or a manifest — and the content store holds whatever the descriptor points to.

The helpers around `images.Image` make the split visible. They resolve the image config, manifests, rootfs diff IDs, and size by walking the descriptor graph through a content provider. The image record is the named root of that graph; the content provider supplies the bytes at every node.

That split is why retagging an image is a metadata-only operation while fetching a missing layer is a content operation. It is also why deleting an image name does not delete the underlying blobs. The bytes might still be reachable from another image, a container, or a lease, and only the garbage collector — after looking at the whole reference graph — gets to decide.

Pulling An Image

A pull starts with a reference and a resolver, not with a download. The resolver locates the registry content. The fetcher walks the descriptor graph, downloading what it does not already have. The image service records a final image object at the end. If the caller asked for unpack, an unpacker is hooked into the traversal so layers are applied to the selected snapshotter as content arrives.

In containerd `v2.3.0` the path through `Client.Pull` does six things, in order:

1. wrap the operation in `WithLease`;
2. resolve the reference to a descriptor;
3. fetch content by walking the descriptor graph;
4. if `WithPullUnpack` is set, run the unpacker as part of the walk;
5. wait for unpack completion before creating the final image object;
6. reject Docker schema 1 manifests, which are no longer accepted as of containerd 2.1.

A pull is therefore a descriptor walk with content ingestion, optional unpack, metadata creation, and lease protection.

Snapshotters Are Filesystem State Machines

A snapshotter is a small state machine over filesystem snapshots, with three states — active, view, committed — and five operations:

- `Prepare` creates an active writable snapshot over a parent.
- `View` creates a read-only view.
- `Commit` records the changes in an active snapshot as a committed snapshot.
- `Mounts` returns mount instructions for a snapshot.
- `Remove` deletes active or committed state when no dependency holds it.

Image layers become committed snapshots; the container gets a new active writable snapshot on top of the committed chain at start time. That active snapshot is the container's writable layer. The image's committed layers stay immutable underneath it.

The default Linux backend is overlayfs, but the interface also has native, btrfs, zfs, blockfile, devmapper, Windows, and remote/lazy backends. Everything above this interface — containers, tasks, CRI — talks to "a snapshotter" without caring which one.

Unpack Connects Content To Snapshots

Unpacking is where image config and layer descriptors meet filesystem state. The unpacker reads the image config, lines layer descriptors up against the rootfs diff IDs, computes chain IDs, applies layers, and commits snapshots under stable names.

The core prepare/apply/commit loop in `core/unpack/unpacker.go` is three calls per layer:

```
mounts, err = sn.Prepare(ctx, key, parent, opts...)
diff, err := a.Apply(ctx, desc, mounts, ...)
err = sn.Commit(ctx, chainID, key, opts...)
```

Three identifiers travel together and must not be confused. The **descriptor digest** names the compressed blob in the content store. The **diff ID** names the uncompressed filesystem change after the layer is applied. The **chain ID** is a digest over the sequence of diff IDs up to and including the current layer; it becomes the snapshot key for the committed layer chain.

That is also why unpack waits for the image config. The config's `rootfs.diff_ids` is the source of truth for the uncompressed changes containerd expects. After applying a layer, containerd recomputes the diff ID and checks it against the config; only on a match does it commit the snapshot.

Garbage Collection References

Unpack also writes the labels that let the garbage collector cross from content into snapshots. After verifying a layer, unpack stamps the layer's content blob with a label for its uncompressed diff ID. After committing the chain, it stamps the image config with a snapshot GC reference label for the selected snapshotter, pointing at the final chain ID.

Without those labels, the collector cannot bridge the two stores. The content store and the snapshotter are separate systems with separate garbage; the labels are the edges that turn them into one reference graph. An image config points at a final chain ID, that chain ID depends on earlier committed snapshots, and the collector preserves the whole subgraph as long as anything reachable still references it.

Leases sit on top of that. A pull, unpack, or container-creation flow takes a lease to protect the in-progress set of content, metadata, and snapshots from collection until it has assembled a consistent result. Without leases, the collector would race the pull and delete in-progress objects.

From Image Chain To Container Rootfs

Committed snapshots for the image chain are not a running container; a client (or the CRI path) calls `Prepare` with the chain as parent, and task creation asks the snapshotter for mounts and passes them to runtime v2.

The full path from a typed reference to a mounted rootfs is seven steps:

1. Resolve an image reference.
2. Fetch descriptors and blobs into the content store.
3. Record image metadata pointing at the descriptor target.
4. Unpack layers into committed snapshots.
5. Prepare an active snapshot for the container.
6. Pass snapshot mounts to task creation.
7. Let the shim and runtime mount the root filesystem for execution.

No single object in that list is "the image" in every sense. The name, the bytes, the metadata, the committed chain, and the active rootfs are five different objects with five different lifetimes.

Where This Goes

A missing blob is a content problem. A missing unpacked root is a snapshot problem. A wrong tag is image metadata. A writable layer that will not mount is an active snapshot problem. A process that never starts is in the task and shim path — which is where chapter 12 picks up, with a container record that already knows which snapshot it owns and a task request that turns that record into live execution.

Sources And Further Reading

- containerd `v2.3.0` source: https://github.com/containerd/containerd/tree/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a
- Content flow docs: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/docs/content-flow.md
- Content interfaces: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/content/content.go
- Image store and helpers: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/images/image.go
- Snapshotter interface: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/snapshots/snapshotter.go
- Pull path: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/client/pull.go

- Unpacker: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/unpack/unpacker.go
- Snapshotter docs: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/docs/snapshotters/README.n

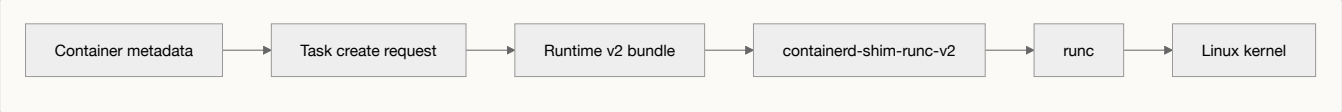
Chapter 12: Containers, Tasks, And Shims

In containerd, a container is not a running process. A container is persistent metadata; a task is live execution; a shim is the process boundary that lets containerd supervise that task without becoming the workload's direct parent.

That single distinction is why `ctr container ls` can list a container while `ctr task ls` shows nothing for it, and why `ctr task rm` leaves the container record in place. A container can exist with no task. A task can exit while the container record remains. Deleting a task is not the same operation as deleting the container metadata, and starting a task is not the same operation as creating the container record.

Word	containerd meaning	Concrete state
Container	Persistent metadata	ID, OCI spec, image, snapshot key, runtime, labels, extensions, sandbox ID
Task	Live execution	PID, status, IO, exec processes, runtime operations
Shim	Runtime supervisor	Task service endpoint, runtime calls, IO, exit handling, event publishing

The common Linux path is:



Container Metadata

A container record is the durable intent containerd needs later: runtime choice, image name, OCI spec, snapshotter, snapshot key, labels, extensions, and an optional sandbox ID. It is not an init process and it is not a cgroup.

The client-side `Container` interface reinforces the split. The metadata operations — `Info`, `Spec`, `Image`, `Update`, `Delete` — all stay on the metadata side; only `NewTask` crosses over into live execution. That is why `ctr container create` can return success while nothing is running on the host: the record exists, the process does not.

Task Creation

`Container.NewTask` turns a container record into a task service request. It creates or attaches IO, resolves snapshot mounts if the container has a snapshot key, carries runtime options across, and sends a `CreateTaskRequest` to the task service.

Strip the supporting code in `client/container.go` away and the call is two lines:

```
request := &tasks.CreateTaskRequest{ContainerID: c.id}
response, err := c.client.TaskService().Create(ctx, request)
```

The surrounding code builds IO, reads the container spec, asks the snapshotter for mounts, and fills runtime options before the request goes out — but a container record does not become a task until a client calls `NewTask`.

Starting is a separate boundary again. The client `Task` interface offers `Start`, `Kill`, `Pause`, `Resume`, `Exec`, `Pids`, `Checkpoint`, `Update`, `Metrics`, `Spec`, `Wait`, and `Delete`. `Create` prepares runtime state. `Start` runs the process. The split lets a runtime build a container's disk and IO state, optionally take a checkpoint, and only then begin execution.

Runtime v2 Bundles

Before a shim starts, the runtime v2 task manager builds a bundle directory on disk. A bundle is not an image layer and not part of the content store; it is per-task runtime state, scoped to one container.

`NewBundle` validates the task ID, creates the namespace-scoped state and work directories, creates a `rootfs` directory inside the bundle, links the work directory back in, and writes `config.json` when the request carries an OCI spec. The snapshot mounts from chapter 11 are activated into this same path.

That bundle is the handoff point between containerd metadata and a real OCI runtime. The shim is given enough information to ask runc to create the container without ever consulting the containerd database.

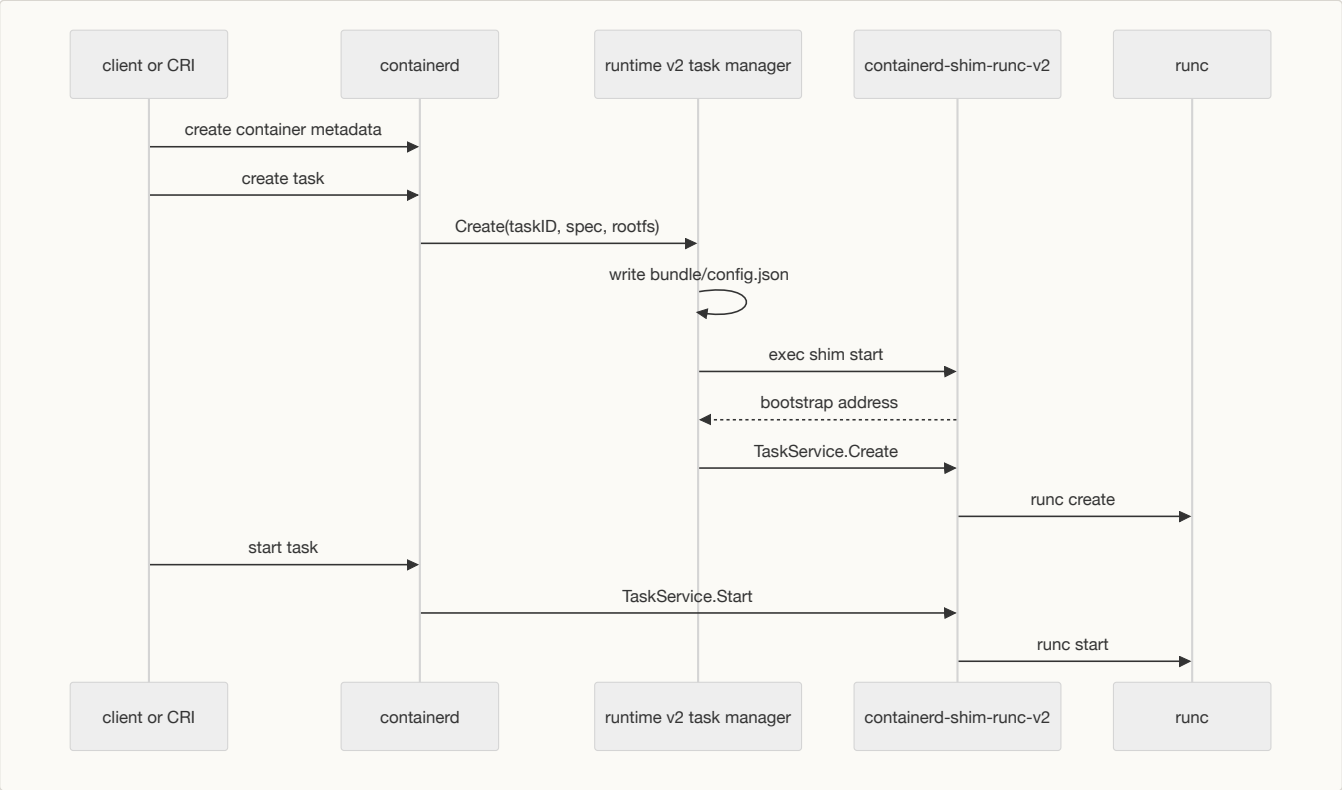
Starting Or Reconnecting To A Shim

The runtime v2 task manager creates the bundle, activates mounts, hands off to the shim manager to start or reconnect to a shim, validates the shim's runtime features, and calls the shim task client's `Create`.

Runtime names look like Java package paths: `io.containerd.runc.v2` is the canonical Linux/runc handler. The shim manager translates that to the binary name `containerd-shim-runc-v2`, finds it on `PATH`, executes it with the `start` action, parses the bootstrap address out of the response, connects over ttrpc (or gRPC, for shims that opt in), and persists the bootstrap data so containerd can reconnect to the same shim later.

This is the daemon-restart survival case from chapter 3, made concrete: the shim keeps supervising the workload across a containerd restart and reattaches when the daemon comes back.

The startup sequence looks like this:



Shim Grouping

"One shim per container" is a useful first approximation and not a rule. containerd 2.3 ships shim API version 3, and when a task belongs to a sandbox whose endpoint is reachable, the shim manager connects to the existing sandbox shim instead of starting another binary. If the endpoint is missing or its API version is older than the sandbox protocol, it falls back to launching a fresh shim.

The CRI sandbox path is the most visible place this matters: every container in a pod can share a single sandbox shim. What stays constant is that the shim is the runtime supervision boundary; how many shims there are depends on the runtime and sandbox configuration.

The runc Shim

`containerd-shim-runc-v2` is the runtime v2 task service for the standard Linux runc path. Its service object tracks containers and processes, watches OOM events, reaps exits, and publishes lifecycle events back to containerd.

On `Create`, the shim runs its runc container creation path: it converts the task request's mounts into process mounts, mounts the rootfs into the bundle's `rootfs` directory, writes runtime options, constructs the init process, and prepares a `runc create` invocation that carries the runtime root, bundle path, containerd namespace, runc binary name, and `systemd-cgroup` setting.

On `Start`, the shim calls the container's `Start` method and publishes a task-start or exec-start event. When the process exits, the shim picks the exit up through its wait handling and publishes a task-exit event. containerd never had to be the workload's direct parent to learn it died.

The snapshotter produced mounts in chapter 11; the task manager dropped them into the bundle path; the runc shim mounted the rootfs and handed runc an OCI bundle to create and start.

Events And Waiting

Task lifecycle is not only request and response. Callers can `Wait`, subscribe to events, or inspect status; runtime v2 shims publish task create, start, exit, delete, pause, resume, OOM, and exec events with defined ordering.

A `Start` can succeed and the process can exit a millisecond later. A client that only records the response from `Start` has observed the request, not the lifecycle. CRI, Docker, and `nerdctl` all rely on the wait and event paths to keep their own state honest.

Where This Goes

Chapter 13 stacks Kubernetes on top of all of that. Kubelet does not call runc — it calls CRI, and the CRI plugin builds sandboxes and containers in containerd that ride the same task-and-shim machinery this chapter just walked through.

Sources And Further Reading

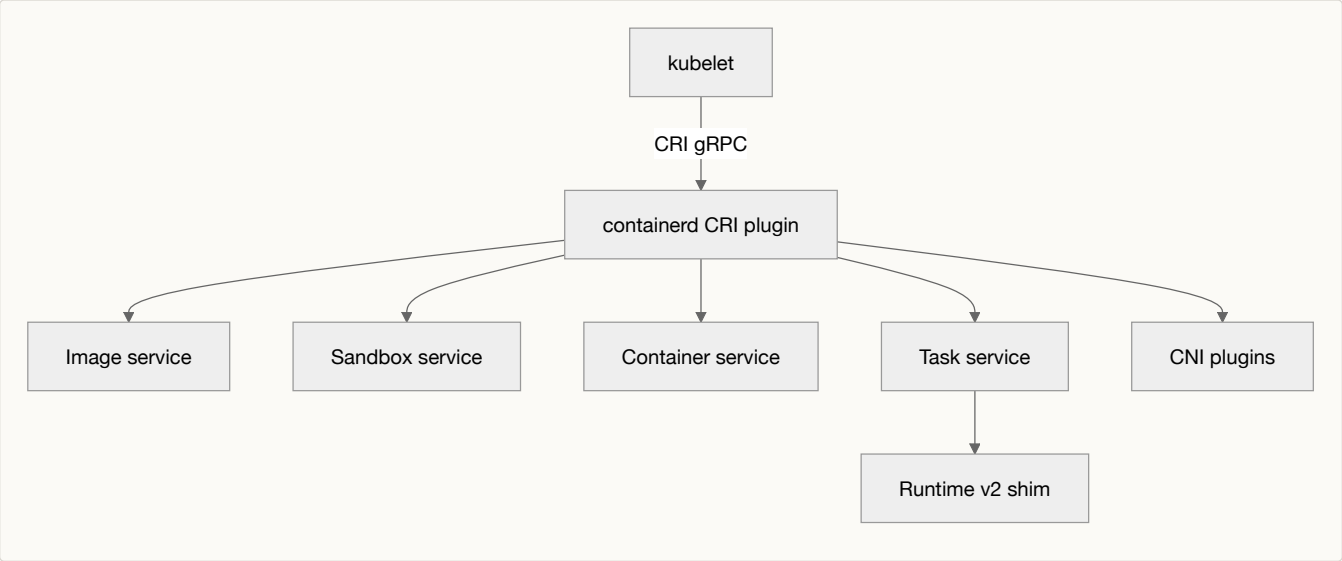
- Runtime v2 docs: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/docs/runtime-v2.md
- Container metadata API: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/containers/containers.go
- Client container code: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/client/container.go
- Client task code: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/client/task.go
- Runtime v2 task manager: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/runtime/v2/task_manage
- Shim manager: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/runtime/v2/shim_manag
- Bundle handling: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/runtime/v2/bundle.go
- runc shim task service: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/cmd/containerd-shim-runc-v2/task/service.go

- runc shim container path:
  https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/cmd/containerd-shim-runc-v2/runc/container.go

Chapter 13: CRI And Kubernetes

Kubelet does not talk to `runc`. It talks to the Container Runtime Interface, and in containerd, CRI is an in-daemon plugin that translates Kubernetes runtime and image RPCs into containerd service calls.

Kubernetes has pods, pod sandboxes, runtime handlers, image pulls, container creates, exec, attach, port-forward, status, stats, and logs. containerd has namespaces, image metadata, content, snapshotters, containers, tasks, shims, leases, events, and runtime options.



Registration

The CRI plugin registers itself as a containerd gRPC plugin and pulls in a long dependency list — runtime, image, sandbox, NRI, event, service, lease, transfer, sandbox-store, and warning plugins. During init it builds an in-memory containerd client pinned to the `k8s.io` namespace, constructs the CRI service, and registers Kubernetes runtime and image servers on containerd's existing gRPC server.

The registration call in `plugins/cri/cri.go` is two lines:

```
runtime.RegisterRuntimeServiceServer(s, instrumented)
runtime.RegisterImageServiceServer(s, instrumented)
```

Everything kubelet ever sees of containerd flows through those two servers.

The `k8s.io` namespace is the second half of the boundary. Kubernetes-managed images, containers, sandboxes, leases, and snapshots all live in that one metadata partition. It does not isolate Linux processes — it keeps containerd records from colliding with whatever a developer creates by hand through `ctr` in the `default` namespace.

CRI Service State

CRI keeps a Kubernetes-shaped index over containerd state — sandbox stores, container name indexes, CNI state, stats — so kubelet can answer its own questions without round-tripping through containerd. containerd remains the source of truth for images, snapshots, and tasks; CRI keeps Kubernetes bookkeeping next to it. That is why CRI source can look larger than expected: it is preserving Kubernetes semantics on top of containerd objects that have their own.

The CRI API Shape

Kubernetes splits CRI into a runtime service and an image service. The runtime service covers pod sandbox operations, container operations, exec, attach, port-forward, status, stats, checkpointing, and runtime config. The image service covers image listing, status, pulls, removals, and filesystem usage.

The shape is visible in kubelet's own behavior. An image pull is a CRI image-service call. A workload container create is a runtime-service call. Starting that container is another runtime-service call. Inside containerd, those three calls land on the same image, snapshot, container, task, and shim machinery from chapters 11 and 12.

The same split is why `CreateContainer` does not start the workload. CRI inherits the OCI and containerd convention of keeping create and start as separate lifecycle steps; the API never had a "run" verb to begin with.

Pod Sandboxes

The first runtime object Kubernetes asks for is a pod sandbox. The pause container is the most visible piece; the sandbox is the metadata, network namespace, CNI result, runtime endpoint, labels, monitor state, and (on the default Linux path) the pause container itself.

`RunPodSandbox` does the Kubernetes-facing setup. In containerd `v2.3.0`, the source path runs through thirteen steps:

1. generate and reserve a sandbox name;
2. create a lease;
3. resolve the runtime handler;
4. store sandbox metadata;
5. create a network namespace unless host networking is requested;
6. run CNI setup;
7. create sandbox metadata through the sandbox service;
8. ensure the pause image exists;
9. start the sandbox;
0. save the sandbox endpoint, labels, and spec;
1. run NRI hooks;
2. mark the sandbox ready;
3. store the sandbox and start an exit monitor.

CNI gets its own chapter in Part V; for now it is enough that CRI runs CNI as part of `RunPodSandbox` and turns the result into containerd service calls.

The default pod-sandbox controller still creates a real containerd container and task for the sandbox image. It builds a sandbox container spec, prepares a snapshot, creates a container with runtime options, creates a task with null IO, waits on it, starts it, and records the PID. That sequence is the bridge between a Kubernetes pod sandbox and the container/task/shim model from chapter 12.

Image Pulls Through CRI

CRI `PullImage` starts as a Kubernetes image request, not a raw containerd pull. The CRI code normalizes the reference, picks a snapshotter from the pod sandbox or runtime handler context, and then drops into either the local client pull path or the transfer service.

On the local pull path, CRI passes a set of options that all show up again later:

- `WithPullUnpack`, so the image is unpacked into the chosen snapshotter;
- the snapshotter selection itself;

- labels for indexing and GC;
- download concurrency and rate limits;
- unpack-duplication suppression;
- optional layer-discard behavior.

That handover is why runtime handlers can affect image pulls. If a handler points at a non-default snapshotter, the image service needs the choice at pull time, because unpack has to land in the same filesystem backend the workload will eventually mount. Pulling for one snapshotter and starting on another is a common operational mistake and a hard one to debug after the fact.

Workload Container Creation

`CreateContainer` takes a pod sandbox ID and a container config. It checks the sandbox exists, resolves the already-pulled image, generates CRI metadata, builds an OCI spec using the sandbox's PID and network namespace state, prepares a new writable snapshot, attaches runtime and sandbox metadata, creates the containerd container, records CRI container state, and emits a container-created event.

It does not start the workload. `CreateContainer` prepares metadata, spec, snapshot state, and CRI bookkeeping; the user's process does not exist until `StartContainer` runs.

The snapshot step is where chapter 11's image work meets the workload. The image has already been pulled and unpacked into committed snapshots. `CreateContainer` calls `Prepare` for an active writable snapshot on top of that chain, and that active snapshot is what task creation will eventually hand to the shim as the root filesystem.

Workload Start

`StartContainer` is the live-execution step. It verifies the sandbox is ready, creates loggers and IO, carries the sandbox endpoint into task options when one exists, creates a containerd task, waits on it in the background, runs NRI start hooks, starts the task, records the PID and start time, launches an exit monitor, and emits a container-started event.

The single call crosses every layer Part IV has introduced:

1. kubelet calls CRI `StartContainer`;
2. CRI looks up its container and sandbox state;
3. CRI asks containerd to create a task for the container;
4. containerd runtime v2 builds the bundle and dials the shim;
5. the shim asks the OCI runtime to create and start the process;
6. CRI records the PID, monitors exit, and reports status back to kubelet.

kubelet never has to know how `io.containerd.runc.v2` becomes `containerd-shim-runc-v2`, where the bundle is written, or how snapshots are mounted into the rootfs.

Runtime Handlers

A runtime handler is the Kubernetes-facing name for a configured slice of containerd runtime behavior. Inside containerd it selects the runtime type, runtime options, snapshotter, sandboxer, runtime binary path, and IO mode, and it carries snapshotter information into the image service so pulls land in the right place.

Kubernetes should be able to ask for a runtime class — default, a VM-backed runtime such as Kata, an alternative such as crun — without constructing containerd runtime options by hand. The handler is the indirection that makes RuntimeClass a first-class Kubernetes object instead of an opaque config string.

The same handler is where most operational mistakes surface. A handler that names one snapshotter while the image was pulled and unpacked into another will fail at container start as a filesystem problem, not a configuration one. A handler that selects a sandbox-aware runtime pulls shim grouping and sandbox endpoints into task startup. Every field on a runtime handler changes a

concrete containerd behavior: snapshotter selection, runtime options, shim grouping for sandbox-aware runtimes.

Where This Goes

The stack reads cleanly in both directions. From kubelet down: CRI request → containerd services → runtime v2 shim → OCI runtime → kernel. From the kernel up: process and namespaces → OCI runtime → shim → containerd task → CRI container → Kubernetes pod. Part V picks up the pod's network namespace and shows how it gets wired into the host.

Sources And Further Reading

- Kubernetes CRI docs: https://kubernetes.io/docs/concepts/containers/cri/
- CRI API proto `v0.36.0` : https://github.com/kubernetes/cri-api/blob/v0.36.0/pkg/apis/runtime/v1/api.proto
- containerd CRI architecture docs: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/docs/cri/architecture.md
- CRI plugin: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/plugins/cri/cri.go
- CRI runtime service: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/service.go
- CRI sandbox run path: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/sandbox_r
- Pod sandbox controller: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/podsandbo
- CRI image pull path: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/images/ima
- CRI container create path: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/container_
- CRI container start path: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/container_

PART V — NETWORKING

Chapter 14: Network Namespaces And Virtual Ethernet

A process can have the same filesystem view as another process and still be on a different network. On Linux, that separation is a network namespace: a separate network stack with its own interfaces, addresses, routes, firewall state, sysctls, procfs networking views, port numbers, and abstract UNIX domain socket namespace.

That scope matters because container networking is not one mechanism. A namespace gives the process its own network stack. A veth pair connects that stack to a peer device in another namespace. A bridge, route table, overlay, cloud route, eBPF datapath, or direct device assignment decides where packets go after that.

What A Network Namespace Holds

A network namespace owns network devices, IPv4 and IPv6 protocol stacks, routing tables, firewall rules, `/proc/net`, `/sys/class/net`, `/proc/sys/net`, port numbers, and the abstract UNIX domain socket namespace. Four ordinary Linux objects compose that state:

Object	Kernel-facing meaning	Container relevance
Link	A network interface, backed by `struct net_device`.	A veth end, bridge, loopback device, and physical NIC are all links.
Address	An IPv4 or IPv6 address associated with an interface index.	A pod IP is an address on an interface in the pod namespace.
Route	A rule for selecting an output device, table, and optional next hop.	A namespace needs routes before traffic can leave its local link.
Neighbor	A mapping from protocol address to link-layer address.	Ethernet delivery still needs ARP or neighbor discovery after route lookup.

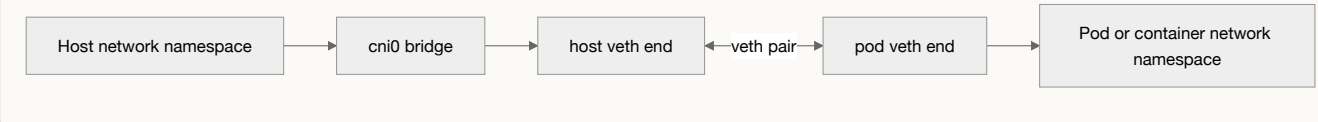
The kernel object behind an interface is `struct net_device`. It is larger than the fields a user normally sees, but the common inspection vocabulary is visible in the struct: MTU, flags, interface index, name, protocol-specific state, and hardware address.

```
unsigned int mtu;
unsigned int flags;
int ifindex;
char name[IFNAMSIZ];
struct in_device __rcu *ip_ptr;
const unsigned char *dev_addr;
```

Modern Linux network configuration usually travels over rtnetlink. The message names line up with the same objects:

```
RTM_NEWLINK,
RTM_NEWADDR = 20,
RTM_NEWROUTE = 24,
```

Three reminders before plugins enter the picture: an IP address belongs to an interface, not a process; a route picks an output path but does not deliver a frame; on Ethernet, the neighbor subsystem still resolves the link-layer destination.



This is one CNI layout — a local bridge with veth pairs into the host namespace; chapter 16 covers the Kubernetes pod model, and chapter 15 covers the CNI ADD/DEL contract.

Network Namespace State

In the kernel, a network namespace is represented by `struct net`. A few fields are enough to show what that namespace owns:

```
struct user_namespace *user_ns;
struct list_head dev_base_head;
struct proc_dir_entry *proc_net;
```

The namespace has an owning user namespace, a device list, and procfs state.

Creating a new network namespace is tied to `CLONE_NEWNET`. In the kernel path that copies network namespace state for a new task, no flag means "keep the old namespace." The flag means "allocate and initialize a new `struct net`."

```
if (!(flags & CLONE_NEWNET))
    return get_net(old_net);
net = net_alloc();
rv = setup_net(net);
```

That is the boundary runtimes and tools cross when they use `clone(2)`, `unshare(2)`, or `setns(2)` with network namespace flags.

Device Lifetime

A physical network device belongs to one network namespace at a time. Moving a physical NIC into a container namespace is possible, but it takes the device away from the host namespace while it is there. When the last process in a network namespace exits, physical devices move back to the initial network namespace.

Virtual Ethernet devices behave differently. A veth pair is destroyed when the namespace that owns it is freed. That makes veth useful for container lifetimes: a runtime or plugin can create a pair, move one end into the target namespace, keep the other end on the host, and let cleanup remove the virtual link when the namespace is gone or when the plugin deletes the attachment.

The `veth(4)` model is direct: packets transmitted on one end are received on the other.

veth Pairs

The veth driver encodes the pair relationship as peer pointers. During creation it registers both devices and stores the peer pointer in each direction:

```
rcu_assign_pointer(priv->peer, peer);
rcu_assign_pointer(priv->peer, dev);
```

Transmit begins by looking up that peer:

```
rcv = rcu_dereference(priv->peer);
```

One end of the veth can be named `eth0` inside a pod namespace. The other end can have a host-generated name and live in the host namespace. The kernel treats them as two Ethernet devices joined back to back. What happens outside the host end depends on the surrounding network implementation.

The host end might be attached to a Linux bridge. It might be routed directly. It might be consumed by an eBPF datapath. It might be part of an overlay system.

Bridges

A Linux bridge is a layer-2 forwarding device. In the local bridge pattern, the bridge sits in the host network namespace, host-side veth ends are enslaved to it, and the bridge can also hold a gateway address for the container subnet. Packets between containers on the same bridge can be forwarded at layer 2. Packets leaving that subnet need routing, forwarding, and sometimes masquerade.

The bridge interface code shows the relationship between an enslaved device and the bridge. When a device is added as a bridge port, the kernel registers a receive handler on the device and marks it as a bridge port:

```
err = netdev_rx_handler_register(dev, br_get_rx_handler(dev), p);
dev->priv_flags |= IFF_BRIDGE_PORT;
```

From inside the namespace, the bridge is invisible: the process sees only the pod-side veth, addresses, routes, and resolver configuration.

Routes, NAT, And Firewall State

Network namespaces have their own route tables and firewall state. A simple bridge setup usually needs an address inside the container namespace, a default route pointing at the bridge gateway, forwarding on the host, and a route or NAT rule for traffic beyond the local subnet.

Local bridge demos commonly masquerade outbound traffic because the external network does not know how to route back to the container subnet. Kubernetes makes a stronger cluster-level promise: pods can communicate with pods on other nodes without NAT at the Kubernetes layer. A plugin may still use NAT for selected features, such as host ports or private egress, but pod-to-pod reachability is the contract.

The CNI bridge plugin exposes the local-bridge choice as configuration. When `ipMasq` is enabled, the plugin installs masquerade for the configured networks.

DNS Is Separate

A network namespace does not configure DNS. Processes still read resolver configuration from `/etc/resolv.conf`, and orchestrators arrange `/etc/hosts`, hostnames, and search domains.

In Kubernetes, pod DNS belongs above the Linux namespace boundary. The pod gets a network namespace and IP address, but kubelet and the runtime arrange files such as `/etc/resolv.conf`, while the cluster DNS add-on supplies service and pod records according to Kubernetes DNS rules. A CNI result can include DNS fields, but that is not the same thing as the cluster's DNS policy.

Chapter 15 picks up the CNI contract — the spec that tells a plugin which namespace to modify and how to report what it did.

Sources And Further Reading

- Linux `netdevice(7)`: https://man7.org/linux/man-pages/man7/netdevice.7.html
- Linux `rtnetlink(7)`: https://man7.org/linux/man-pages/man7/rtnetlink.7.html
- Linux `ip-route(8)`: https://man7.org/linux/man-pages/man8/ip-route.8.html
- Linux `arp(7)`: https://man7.org/linux/man-pages/man7/arp.7.html
- Linux `packet(7)`: https://man7.org/linux/man-pages/man7/packet.7.html
- Linux `network_namespaces(7)`: https://man7.org/linux/man-pages/man7/network_namespaces.7.html
- Linux `veth(4)`: https://man7.org/linux/man-pages/man4/veth.4.html
- Linux `resolv.conf(5)`: https://man7.org/linux/man-pages/man5/resolv.conf.5.html
- Linux kernel `struct net_device`: https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/include/linux/netdevice.h

- Linux kernel rtnetlink:
  https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/net/core/rtnetlink.c
- Linux kernel IPv4 routing:
  https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/net/ipv4/route.c
- Linux kernel neighbor subsystem:
  https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/net/core/neighbour.c
- Linux kernel `struct net`:
  https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/include/net/net_namespace.h
- Linux kernel network namespace setup:
  https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/net/core/net_namespace.c
- Linux kernel veth driver:
  https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/drivers/net/veth.c
- Linux bridge interface code:
  https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/net/bridge/br_if.c
- Kubernetes networking model: https://kubernetes.io/docs/concepts/services-networking/
- CNI bridge plugin: https://github.com/containernetworking/plugins/blob/v1.9.1/plugins/main/bridge/bridge.go

Chapter 15: CNI

CNI is the contract a container runtime uses to call into network implementation code: it specifies how the runtime invokes a plugin binary and how the plugin returns its result, not what the plugin does.

A CNI runtime executes plugin binaries. It passes a container ID, a network namespace path, an interface name, optional runtime arguments, and JSON configuration. Plugins mutate the target network attachment and return JSON results. The data path might be a Linux bridge, routes, an overlay, eBPF, cloud networking, device plugins, or a chain of smaller plugins.

Library tags and spec version are separate. Source links in this chapter use `containernetworking/cni` `v1.3.0` and `containernetworking/plugins` `v1.9.1`; the CNI spec they implement is version `1.1.0`.

Configuration

CNI configuration is JSON. A single network configuration names one plugin. A configuration list names a sequence of plugins and lets each plugin receive the previous result.

The list-level fields are:

- `cniVersion` names the CNI spec version used for the configuration.
- `cniVersions` can advertise supported versions.
- `name` is the network name.
- `disableCheck` and `disableGC` disable those maintenance operations for a list.
- `plugins` is the ordered plugin chain.

Inside each plugin object, `type` names the binary to execute. Other fields belong to the plugin. Common examples are `ipMasq` for bridge masquerade behavior, `ipam` for address management delegation, `dns` for DNS data in the result path, and `capabilities` for runtime-provided values such as port mappings.

The spec uses the word "container" broadly. It means the network isolation domain being attached to a network. On Linux that is usually a network namespace path, but the CNI protocol is not restricted to Linux namespace internals.

Invocation

The runtime side executes a plugin binary from `CNI_PATH`, passes invocation data through environment variables, sends the configuration JSON on stdin, and reads a JSON result from stdout. Failures are structured CNI errors rather than free-form terminal output.

In libcni, the runtime arguments become environment variables like these:

```
"CNI_COMMAND="+args.Command,
"CNI_NETNS="+args.NetNS,
"CNI_IFNAME="+args.IfName,
```

The plugin execution path then runs the binary and decodes the returned bytes into a versioned CNI result:

```
stdoutBytes, err := exec.ExecPlugin(ctx, pluginPath, netconf, args.AsEnv())
return create.Create(resultVersion, fixedBytes)
```

That is why a CNI plugin does not have to be linked into containerd, kubelet, or any runtime. It has to be executable, discoverable, and able to speak environment variables plus JSON.

Operations

The core operations are verbs sent in `CNI_COMMAND`.

Operation	Purpose
`ADD`	Attach the container to the network or apply a plugin's change.
`DEL`	Remove the attachment or undo the plugin's change.
`CHECK`	Verify the expected attachment still exists.
`STATUS`	Report plugin or network availability.
`VERSION`	Report supported CNI versions.
`GC`	Clean stale attachments known to the runtime and plugin.

`ADD` is the easy path to understand because it creates visible state. `DEL` is just as important because interfaces, routes, firewall rules, and IPAM allocations outlive the process that asked for them unless something removes them. `CHECK`, `STATUS`, and `GC` are about drift and maintenance: what exists now, whether the plugin can operate, and which stale attachments can be collected.

Chaining

Plugin chaining is the reason a configuration list is more than a list of independent commands. On `ADD`, libcni walks the list in order and passes each result into the next plugin:

```
result, err = c.addNetwork(ctx, list.Name, list.CNIVersion, net, result, rt)
```

On `DEL`, libcni walks the list in reverse order:

```
for i := len(list.Plugins) - 1; i >= 0; i-- {
    net := list.Plugins[i]
```

The order matches ownership. If one plugin creates an interface and a later plugin installs port-forwarding rules for that interface, deletion should remove the port-forwarding rules before the underlying interface disappears. From CNI spec `1.1.0` onward, libcni passes cached results into deletion, so a cleanup call has more context than just a container ID.

The `GC` path is separate from normal deletion. libcni can read cached attachments, compare them with the runtime's valid attachment set, delete stale attachments, and issue plugin `GC` operations for CNI version `1.1.0` and later.

Bridge And IPAM

The bridge plugin is a concrete implementation of the local pattern from the previous chapter. Its `ADD` path can create or reuse a Linux bridge, create a veth pair, move one end into the target namespace, run IPAM, configure addresses and routes, enable forwarding, and install masquerade rules when configured.

The source makes both halves visible:

```
hostInterface, containerInterface, err := setupVeth(...)
r, err := ipam.ExecAdd(n.IPAM.Type, args.StdinData)
```

`setupVeth` handles the link between the host namespace and the target namespace. `ipam.ExecAdd` delegates address allocation to another plugin named in the `ipam` block. In a typical bridge configuration, the main plugin wires the device and the IPAM plugin decides which IP address and route data to return.

The `host-local` IPAM plugin stores allocations on local disk and returns addresses and routes from configured ranges:

```
ipConf, err := allocator.Get(args.ContainerID, args.IfName, requestedIP)
```

That local-disk detail is why IPAM cleanup cannot be treated as optional. If `DEL` does not release an allocation, later containers can run out of addresses even after every visible process has exited.

Chained Plugin Examples

The `portmap` plugin shows what a chained plugin looks like. It does not create the pod interface. It expects a previous plugin to have produced a result, reads runtime-supplied port mappings, and installs host port forwarding rules through an iptables or nftables backend.

Its guard is blunt:

```
if netConf.PrevResult == nil {
    return fmt.Errorf("must be called as chained plugin")
}
```

CNI plugins are not equal peers that can run in any order. Some create base network state. Some allocate addresses. Some decorate or enforce behavior around a previous attachment.

containerd's CNI Wrapper

containerd `v2.3.0` depends on `github.com/containerd/go-cni` `v1.1.13`, `github.com/containernetworking/cni` `v1.3.0`, and `github.com/containernetworking/plugins` `v1.9.1`. The `go-cni` package wraps libcni behind a smaller interface shaped for containerd: `Setup`, `SetupSerially`, `Remove`, `Check`, `Load`, `Status`, and `GetConfig`.

The namespace attach path maps a containerd namespace object into a libcni network-list add:

```
r, err := n.cni.AddNetworkList(ctx, n.config, ns.config(n.ifName))
```

containerd CRI knows the pod sandbox ID and the network namespace path; CNI knows how to run the configured plugin chain against that path; the plugin chain owns the host network mutations.

What CNI Does Not Promise

CNI does not promise that every pod can reach every other pod. Kubernetes defines that model, and plugins implement it with different data paths. CNI does not make plugin execution safe to run casually on a developer host; real calls can create interfaces, alter routes, change firewall rules, and write IPAM state.

CNI is a process protocol: given a namespace path and a JSON config, the runtime runs a plugin chain and reads back a versioned result. The next chapter shows how kubelet and containerd prepare the namespace path for a Kubernetes pod.

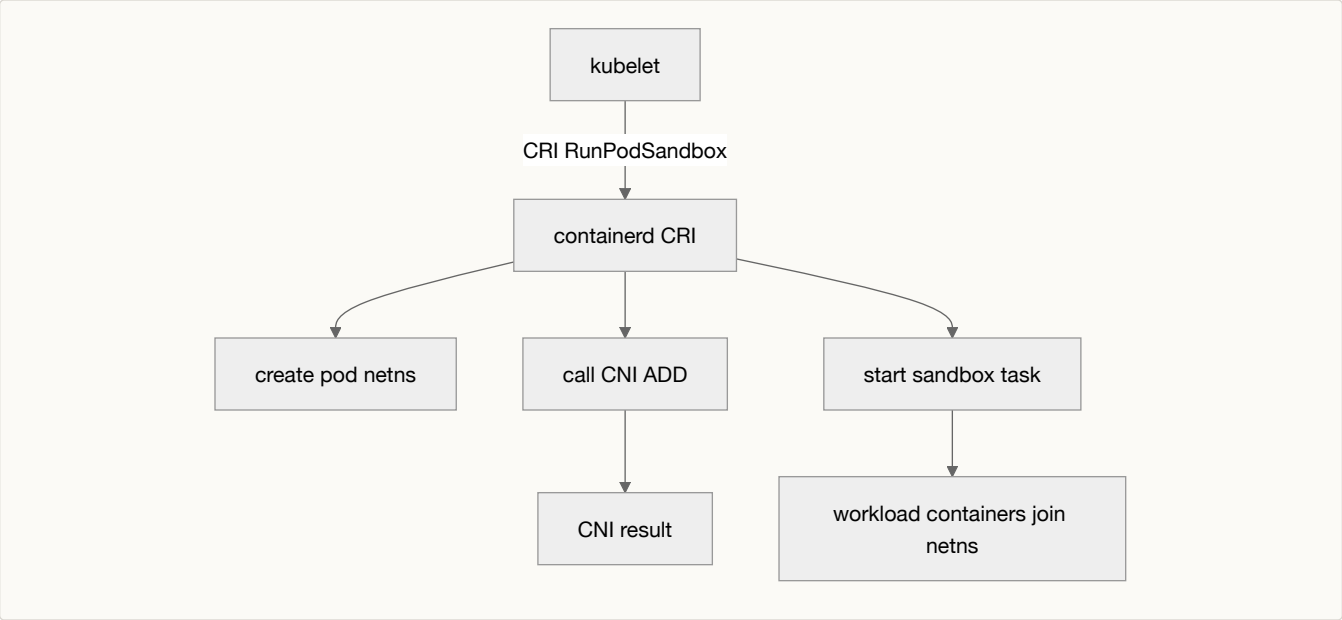
Sources And Further Reading

- CNI specification: https://github.com/containernetworking/cni/blob/v1.3.0/SPEC.md
- CNI docs site: https://www.cni.dev/docs/spec/
- libcni API: https://github.com/containernetworking/cni/blob/v1.3.0/libcni/api.go
- libcni invoke args: https://github.com/containernetworking/cni/blob/v1.3.0/pkg/invoke/args.go
- libcni plugin execution: https://github.com/containernetworking/cni/blob/v1.3.0/pkg/invoke/exec.go
- CNI bridge plugin: https://github.com/containernetworking/plugins/blob/v1.9.1/plugins/main/bridge/bridge.go
- host-local IPAM plugin: https://github.com/containernetworking/plugins/blob/v1.9.1/plugins/ipam/host-local/main.go
- portmap plugin: https://github.com/containernetworking/plugins/blob/v1.9.1/plugins/meta/portmap/main.go
- containerd `go.mod`: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/go.mod
- containerd `go-cni`: https://github.com/containerd/go-cni/tree/v1.1.13

Chapter 16: Pod Networking Model

Kubernetes promises pod networking, not "a CNI network." Each pod gets an IP address. Containers in the same pod share the pod's network namespace and port space. Pods can communicate with pods on other nodes without NAT at the Kubernetes layer, and node agents can communicate with pods on their node.

A plugin can satisfy that model with bridges and routes, overlays, eBPF, cloud provider routes, direct device attachment, or a mix of those techniques. In the containerd path, kubelet does not run the plugin itself. kubelet calls CRI, containerd CRI creates or receives the pod sandbox network namespace, and containerd invokes CNI for that namespace.



The pod IP belongs to the shared pod network namespace. It is not assigned independently to each workload container.

The Kubernetes Contract

The Kubernetes networking model has three details that shape runtime behavior. Pods on a node can communicate with all pods on all nodes without NAT at the Kubernetes layer. Agents on a node, such as kubelet or system daemons, can communicate with pods on that node. Containers in one pod share the pod IP and port space because they share one network namespace.

NetworkPolicy sits beside that model rather than inside CNI core. Kubernetes defines the policy API, but enforcement is plugin-specific. A bridge-only local setup, an eBPF plugin, and a cloud CNI can all attach namespaces through CNI while providing very different policy behavior.

RunPodSandbox Creates The Namespace

In containerd `v2.3.0`, the CRI path handles networking inside `RunPodSandbox`. If the pod requests host networking, CRI skips pod network namespace creation. Otherwise it creates a namespace mount, stores the path in sandbox metadata, and passes that path into network setup.

The source path is:

```
sandbox.NetNS, err = netns.NewNetNS(netnsMountDir)
sandbox.NetNSPath = sandbox.NetNS.GetPath()
```

When pod-level user namespaces are enabled, containerd has to create the network namespace after entering the user namespace context. The Linux helper path uses `CLONE_NEWNET` and then mounts the namespace from the helper process PID:

```
syscall.CLONE_NEWNET,
netns.NewNetNSFromPID(netnsMountDir, uint32(pid))
```

User namespaces change ownership and capability rules around namespace creation, so the runtime has to create the network namespace inside the right user-namespace context before CNI runs against it.

CRI Calls CNI

After namespace creation, containerd CRI calls `setupPodNetwork`. That function chooses the CNI plugin set for the runtime handler, prepares Kubernetes labels and runtime capabilities, calls the selected network plugin, and stores the result on the sandbox:

```
result, err = netPlugin.Setup(ctx, id, path, opts...)
sandbox.CNIResult = result
```

The `id` is the sandbox ID. The `path` is the pod network namespace path. The options carry Kubernetes metadata and runtime data that plugins may need. Labels include pod namespace, pod name, pod UID, and the sandbox container ID:

```
"K8S_POD_NAMESPACE": config.GetMetadata().GetNamespace(),
"K8S_POD_INFRA_CONTAINER_ID": id,
```

Capabilities can include annotations, port mappings, bandwidth, DNS, and cgroup path. A plugin can ignore unsupported fields, but a chained plugin such as `portmap` depends on runtime-provided capability data to install host-port rules.

containerd can also select CNI configuration by runtime handler. During Linux service initialization, it builds `go-cni` instances with plugin configuration directories and binary directories:

```
cni.WithPluginConfDir(dir),
cni.WithPluginDir(c.config.NetworkPluginBinDirs)
```

That gives operators a way to pair a Kubernetes runtime class or handler with a particular network configuration.

The Sandbox Still Starts

CNI setup does not replace the pod sandbox task. After networking is configured, the default pod sandbox controller still prepares a sandbox container from the pause image. It builds an OCI spec, prepares a snapshot, creates a containerd container, creates a task with null IO, starts it, records the PID, and marks the sandbox ready.

The sandbox process is the stable namespace anchor for the pod: the network namespace was created before CNI ran, and the sandbox task keeps the pod's namespaces alive for workload containers that start later.

The pause process is one piece of the sandbox. The rest is CRI metadata, the network namespace path, the CNI result, labels, runtime endpoint data, and monitor state, all of which containerd needs to answer kubelet later.

Workload Containers Join The Namespace

When kubelet asks CRI to create a workload container, the request names a pod sandbox. containerd looks up that sandbox, reads the sandbox PID and network namespace path, and builds the workload's OCI spec so it joins the pod namespaces.

The spec option that does the work is `WithPodNamespaces`. For the network namespace, it writes an OCI Linux namespace entry that points at the sandbox process namespace:

```
oci.WithLinuxNamespace(runtimespec.LinuxNamespace{
    Type: runtimespec.NetworkNamespace,
    Path: GetNetworkNamespace(sandboxPid),
})
```

The same option joins IPC and UTS namespaces and handles pod-level user namespace settings. The workload process does not get a new network stack: it joins the pod sandbox's network namespace, sees the pod interfaces and routes, and shares the port space with the other containers in the pod.

That is why two containers in one pod can collide on a TCP port even when they have different root filesystems and processes. Their network namespace is the same object.

DNS And Pod Files

Pod DNS is configured around the network setup, not by it. Kubernetes defines DNS behavior for Services and Pods, and containerd mounts sandbox `/etc/hosts`, hostname, and `/etc/resolv.conf` files into workload containers when those files exist or are delegated to a sandbox controller.

CNI can return DNS fields, and plugins can participate in resolver configuration. But the resolver file a process reads, the search domains Kubernetes chooses, and the records served by the cluster DNS add-on are not created by `CLONE_NEWNET`. A working pod network needs both sides: an attached namespace and the pod files that make names resolve the way Kubernetes promises.

Cleanup

Teardown follows the ownership chain in reverse. CRI owns sandbox lifecycle state. CNI `DEL` removes the network attachment. libcni deletes chained plugins in reverse order, which lets decorators such as port mapping clean their rules before the base interface disappears. The runtime can then remove the sandbox network namespace after network teardown has run.

Cleanup is where partial failures matter. A failed `ADD` can leave an IPAM allocation, a host-side veth, or a firewall rule. A failed `DEL` can make the next pod fail for reasons that look unrelated.

Where This Goes

Part VI turns these relationships into experiments inside a disposable VM.

Sources And Further Reading

- Kubernetes networking model: https://kubernetes.io/docs/concepts/services-networking/
- Kubernetes Pods: https://kubernetes.io/docs/concepts/workloads/pods/
- Kubernetes DNS for Services and Pods: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
- containerd CRI sandbox run path: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/sandbox_r
- containerd CRI Linux sandbox network namespace helper: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/sandbox_r
- containerd CRI CNI initialization: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/service_lin
- containerd pod sandbox controller: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/podsandbo
- containerd CRI container create path: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/server/container_
- containerd CRI pod namespace spec opts: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/internal/cri/opts/spec_opts.go

- containerd `go-cni` : https://github.com/containerd/go-cni/tree/v1.1.13

PART VI — EXPERIMENTS

Chapter 17: Lab Safety And Shape

A command that creates a namespace, writes a cgroup file, adds a veth pair, or starts `containerd-shim-runc-v2` mutates host state. Part VI runs every mutating experiment inside a disposable Linux VM.

Mounts can leak through shared propagation. cgroup files can affect real processes. A veth pair and route can change packet flow. CNI plugins can write IPAM state and firewall rules. runc and containerd create runtime directories, cgroups, namespaces, mounts, and processes.

Safety Classes

The experiments use three safety classes.

Class	Meaning	Examples
Inspect-only	Reads state without changing it.	`readlink /proc/self/ns/pid`, `findmnt`, `ctr -n lab containers ls`.
User-namespace scoped	Uses a user namespace to reduce host privilege.	`unshare --user --map-root-user` labs.
VM-only mutation	Changes kernel or runtime state.	Mounts, cgroups, veth, bridges, CNI, runc, containerd tasks.

Root inside a user namespace is not root on the host: distribution policy, kernel configuration, subordinate ID mappings, and capability rules all affect what works. Lab steps that depend on predictable behavior call out a VM.

The Lab Shape

Every mutating experiment follows the same shape:

1. State the question.
2. Name the scope.
3. Inspect the starting state.
4. Make the smallest change.
5. Inspect the changed state.
6. Clean up.
7. Verify cleanup.

An experiment that creates a namespace but never looks at `/proc/<pid>/ns` teaches an incantation. An experiment that starts a containerd task but never reads `pgrep containerd-shim-runc-v2` hides the runtime boundary.

Namespace Tools

`unshare(1)` creates new namespaces for a program. `nsenter(1)` enters namespaces that already exist. In util-linux, both tools map command-line choices to kernel namespace flags.

`unshare` carries the namespace table:

```
{ .type = CLONE_NEWNET,   .name = "ns/net"  },
{ .type = CLONE_NEWPID,   .name = "ns/pid_for_children" },
{ .type = CLONE_NEWNS,    .name = "ns/mnt"  },
```

`nsenter` carries the same idea for target namespace files:

```
{ .nstype = CLONE_NEWNET,   .name = "ns/net",  .fd = -1 },
{ .nstype = CLONE_NEWPID,   .name = "ns/pid",  .fd = -1 },
{ .nstype = CLONE_NEWNS,    .name = "ns/mnt",  .fd = -1 },
```

The header comment states the purpose:

```
 * nsenter(1) - command-line interface for setns(2)
```

`unshare(1)` and `nsenter(1)` are lab handles for the same `setns(2)` and `clone(2)` flags that runtimes use.

Persistent References

A namespace can outlive the process that first created it if something still holds a reference. A bind mount of a namespace file is one way to keep that reference. That is useful when an experiment needs two shells to inspect one namespace, but it also creates a cleanup obligation.

The cleanup step should not stop at "exit the shell." It should verify that namespace bind mounts, processes, links, routes, cgroup directories, CNI cache entries, runc state, and containerd tasks are gone.

For Part VI, the shortest useful cleanup rule is:

```
If the setup step names an object, the cleanup step names the same object.
```

Every lab object in Part VI carries a `cdb-` prefix: `cdb-net-a`, `cdb-br0`, `cdb-runc`, `cdb-task`, `cdb-lab.slice`. The prefix is what lets one `find`, `grep`, or `ip link` filter the lab's residue out of the host's existing state.

The First Check

Every lab VM should start with one inspect-only check that records the environment. The exact commands can vary by distribution, but the state being checked does not:

```
uname -a
readlink /proc/self/ns/{mnt,pid,net,user,cgroup}
findmnt -no TARGET,FSTYPE,OPTIONS /sys/fs/cgroup
ip -br link
```

That gives the reader the kernel, current namespace identities, cgroup mount type, and starting link list. If a later cleanup step fails, the lab has a baseline to compare against.

Sources And Further Reading

- `unshare(1)`: https://man7.org/linux/man-pages/man1/unshare.1.html
- `nsenter(1)`: https://man7.org/linux/man-pages/man1/nsenter.1.html
- `setns(2)`: https://man7.org/linux/man-pages/man2/setns.2.html
- `/proc/<pid>/ns`: https://man7.org/linux/man-pages/man5/proc_pid_ns.5.html
- util-linux `unshare.c`: https://github.com/util-linux/util-linux/blob/2308d4c07f74d3149d9bb127afb85ce617ecad88/sys-utils/unshare.c
- util-linux `nsenter.c`: https://github.com/util-linux/util-linux/blob/2308d4c07f74d3149d9bb127afb85ce617ecad88/sys-utils/nsenter.c

Chapter 18: Linux Primitive Experiments

These labs look at kernel objects directly: namespace links, mount table rows, cgroup files, process IDs — before any runtime appears.

These commands are sketches for a disposable Linux VM. Do not run the mutating sections on a normal workstation.

Namespace Membership

Question: what changes when a process enters new namespaces?

Scope: VM-only mutation, with an optional user-namespace variant later.

Start by recording the current namespace links:

```
readlink /proc/self/ns/{mnt,pid,uts,ipc,net,user,cgroup}
```

Then run a shell in new PID, mount, UTS, and IPC namespaces:

```
sudo unshare --fork --pid --mount --mount-proc --uts --ipc bash
```

Inside that shell, inspect the same links:

```
echo "inside pid: $$"
readlink /proc/self/ns/{mnt,pid,uts,ipc,net,user,cgroup}
ps -ef
exit
```

The PID namespace needs `--fork` because the new PID namespace applies to children. `--mount-proc` mounts a procfs view for that PID namespace, which is why `ps` becomes meaningful inside the lab shell. The network and user namespace links should not change in this exact command; the point is to see that namespaces are independent choices.

Cleanup is the shell exit. Verification is another read of the original shell's namespace links.

Entering A Target Namespace

Question: how does another process enter an existing namespace?

Scope: VM-only mutation.

In one shell, keep a namespaced process alive:

```
sudo unshare --fork --pid --mount --mount-proc --uts bash -c 'hostname cdb-lab; sleep 300'
```

In another shell, find that process and inspect its namespace links:

```
pid=$(pgrep -f "sleep 300" | head -n 1)
sudo readlink /proc/"$pid"/ns/{pid,mnt,uts}
sudo nsenter --target "$pid" --pid --mount --uts hostname
```

`nsenter(1)` is the user-space wrapper for `setns(2)`. The test is whether `/proc/$pid/ns/uts` matches the link the parent shell read; the `hostname` output is only a sanity check.

Cleanup the sleeping process:

```
sudo kill "$pid"
```

Mount Namespace

Question: does a mount namespace give a process its own mount table?

Scope: VM-only mutation.

Run a shell with a new mount namespace, then make propagation private before creating test mounts:

```
sudo unshare --mount bash
mount --make-rprivate /
mkdir -p /tmp/cdb-mnt/source /tmp/cdb-mnt/target
touch /tmp/cdb-mnt/source/inside-source
mount --bind /tmp/cdb-mnt/source /tmp/cdb-mnt/target
findmnt /tmp/cdb-mnt/target
grep /tmp/cdb-mnt /proc/self/mountinfo
umount /tmp/cdb-mnt/target
exit
```

The bind mount exists in the new mount namespace. `mount --make-rprivate /` is there because shared mount propagation can make mount experiments surprise the host.

Verify cleanup from the original shell:

```
findmnt /tmp/cdb-mnt/target || true
rm -rf /tmp/cdb-mnt
```

Root Filesystem Boundary

Question: what does a root filesystem need before a process can run inside it?

Scope: VM-only mutation.

A directory is not automatically runnable: a dynamic binary needs its loader and shared libraries, `/proc` does not exist unless mounted, device nodes do not appear unless created or mounted, and a shell does not exist unless it is in the tree.

Whichever rootfs the lab uses — a static `busybox` tarball, an exported image, an extracted layer — the inspection checkpoints are the same:

```
find rootfs -maxdepth 2 -type f -o -type l | sort
file rootfs/bin/sh
ldd rootfs/bin/sh || true
```

`rootfs/` is a directory tree, not an image or a snapshot; it can become `/` for a process once the mount namespace and root switch are set up. `pivot_root(2)` is the runtime's call (chapter 20 uses it inside an OCI bundle); `chroot(2)` is enough to demonstrate pathname resolution but does not produce container filesystem setup.

cgroup v2

Question: how does cgroup v2 attach a process to resource-control files?

Scope: VM-only mutation. On a systemd VM, prefer a delegated subtree. Do not write arbitrary cgroup files under host-managed services.

Start with inspection:

```
findmnt -no TARGET,FSTYPE,OPTIONS /sys/fs/cgroup
cat /sys/fs/cgroup/cgroup.controllers
cat /proc/self/cgroup
```

The mutating lab should create a named lab cgroup only under a subtree the lab owns. The smallest useful controller experiment is `pids.max`, because it can be observed without a benchmark:

```
sudo mkdir /sys/fs/cgroup/cdb-lab
echo $$ | sudo tee /sys/fs/cgroup/cdb-lab/cgroup.procs
cat /sys/fs/cgroup/cdb-lab/cgroup.procs
echo 20 | sudo tee /sys/fs/cgroup/cdb-lab/pids.max
cat /sys/fs/cgroup/cdb-lab/pids.current
```

Cleanup requires moving the shell back out before removing the cgroup:

```
echo $$ | sudo tee /sys/fs/cgroup/cgroup.procs
sudo rmdir /sys/fs/cgroup/cdb-lab
```

If `rmdir` fails, something still belongs to the cgroup or the hierarchy does not allow the lab to remove it.

Sources And Further Reading

- `namespaces(7)`: https://man7.org/linux/man-pages/man7/namespaces.7.html
- `unshare(1)`: https://man7.org/linux/man-pages/man1/unshare.1.html
- `nsenter(1)`: https://man7.org/linux/man-pages/man1/nsenter.1.html
- `mount_namespaces(7)`: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html
- `mount(2)`: https://man7.org/linux/man-pages/man2/mount.2.html
- `pivot_root(2)`: https://man7.org/linux/man-pages/man2/pivot_root.2.html
- Linux cgroup v2 docs: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html
- Linux shared subtree docs: https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/Documentation/filesystems/sharedsub

Chapter 19: Networking And CNI Experiments

These labs build the Linux pieces by hand — namespaces, veth, bridges — before invoking a CNI plugin against the result.

Every command in this chapter is for a disposable Linux VM. These labs create network namespaces, veth devices, bridges, routes, and possibly firewall or CNI state.

Manual Network Namespace

Question: what does a new network namespace contain before a plugin touches it?

Scope: VM-only mutation.

Create a namespace and inspect its starting state:

```
sudo ip netns add cdb-a
sudo ip -n cdb-a -br link
sudo ip -n cdb-a route
```

Bring up loopback explicitly:

```
sudo ip -n cdb-a link set lo up
sudo ip -n cdb-a -br link
```

A new network namespace has its own link list and route table. Loopback exists, but it is not useful until it is up.

Cleanup:

```
sudo ip netns del cdb-a
ip netns list
```

veth Pair

Question: how does a namespace get a link to the outside?

Scope: VM-only mutation.

Create a namespace, create a veth pair, move one end into the namespace, and assign addresses:

```
sudo ip netns add cdb-a
sudo ip link add cdb-host type veth peer name cdb-eth0
sudo ip link set cdb-eth0 netns cdb-a
sudo ip addr add 10.200.0.1/24 dev cdb-host
sudo ip link set cdb-host up
sudo ip -n cdb-a addr add 10.200.0.2/24 dev cdb-eth0
sudo ip -n cdb-a link set cdb-eth0 up
sudo ip -n cdb-a link set lo up
```

Inspect both sides:

```
ip -br addr show cdb-host
sudo ip -n cdb-a -br addr
sudo ip -n cdb-a route
ping -c 2 10.200.0.2
sudo ip netns exec cdb-a ping -c 2 10.200.0.1
```

The veth driver stores peer pointers in both directions:

```
rcu_assign_pointer(priv->peer, peer);
rcu_assign_pointer(priv->peer, dev);
```

One kernel network device transmits to its peer.

Cleanup:

```
sudo ip netns del cdb-a
sudo ip link del cdb-host 2>/dev/null || true
```

Deleting the namespace should remove the namespace-owned veth end. The explicit `ip link del` is there so cleanup is idempotent if the host end still exists.

Bridge

Question: what changes when the host-side veth is attached to a bridge?

Scope: VM-only mutation.

Create a bridge and attach a veth host end:

```
sudo ip netns add cdb-a
sudo ip link add cdb-br0 type bridge
sudo ip addr add 10.201.0.1/24 dev cdb-br0
sudo ip link set cdb-br0 up

sudo ip link add cdb-vetha type veth peer name cdb-eth0
sudo ip link set cdb-vetha master cdb-br0
sudo ip link set cdb-vetha up
sudo ip link set cdb-eth0 netns cdb-a
sudo ip -n cdb-a addr add 10.201.0.2/24 dev cdb-eth0
sudo ip -n cdb-a link set cdb-eth0 up
sudo ip -n cdb-a link set lo up
```

Inspect the bridge relationship:

```
bridge link show
ip -br addr show cdb-br0
sudo ip -n cdb-a route
sudo ip netns exec cdb-a ping -c 2 10.201.0.1
```

Forwarding and NAT can collide with the VM's default network setup, so this lab stops at bridge attachment.

Cleanup:

```
sudo ip netns del cdb-a
sudo ip link del cdb-br0
```

CNI With cnitool

Question: what does CNI add above manual namespace wiring?

Scope: VM-only mutation.

`cnitool` runs a CNI configuration against an existing network namespace; the namespace is created by the lab first, then the plugin chain mutates it.

Use a local config and plugin directory, not the host defaults:

```
sudo mkdir -p /tmp/cdb-cni/net.d /tmp/cdb-cni/bin
sudo ip netns add cdb-cni
```

Whichever plugin the lab uses (a small `bridge` or `ptp` config), the runtime variables passed to `cnitool` are the same:

```
sudo NETCONFPATH=/tmp/cdb-cni/net.d CNI_PATH=/tmp/cdb-cni/bin cnitool add cdb-net /var/run/netns/cdb-cni
sudo NETCONFPATH=/tmp/cdb-cni/net.d CNI_PATH=/tmp/cdb-cni/bin cnitool check cdb-net /var/run/netns/cdb-cni
sudo NETCONFPATH=/tmp/cdb-cni/net.d CNI_PATH=/tmp/cdb-cni/bin cnitool del cdb-net /var/run/netns/cdb-cni
```

The experiment should inspect three things after `ADD`: the target namespace links and routes, the host-side link or bridge state, and the IPAM/cache files the plugin wrote. After `DEL`, inspect the same three places again.

Cleanup:

```
sudo ip netns del cdb-cni 2>/dev/null || true
sudo rm -rf /tmp/cdb-cni
```

DNS Boundary

Question: why can a namespace have working packets but broken names?

Scope: inspect-only or VM-only, depending on how the namespace was created.

DNS is not created by `CLONE_NEWNET`. Resolver behavior comes from files and orchestration policy. In a VM namespace lab, inspect route reachability separately from resolver configuration:

```
sudo ip -n cdb-a route
sudo ip netns exec cdb-a cat /etc/resolv.conf
```

If a lab later adds a custom resolver file, it should say exactly how the process sees that file: bind mount, chroot/rootfs content, runtime-generated pod file, or host file inherited by `ip netns exec`.

Sources And Further Reading

- `network_namespaces(7)`: https://man7.org/linux/man-pages/man7/network_namespaces.7.html
- `ip-netns(8)`: https://man7.org/linux/man-pages/man8/ip-netns.8.html
- `ip-link(8)`: https://man7.org/linux/man-pages/man8/ip-link.8.html
- `ip-address(8)`: https://man7.org/linux/man-pages/man8/ip-address.8.html
- `ip-route(8)`: https://man7.org/linux/man-pages/man8/ip-route.8.html
- `veth(4)`: https://man7.org/linux/man-pages/man4/veth.4.html
- Linux veth driver: https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/drivers/net/veth.c
- Linux bridge interface source: https://github.com/torvalds/linux/blob/57b8e2d666a31fa201432d58f5fe3469a0dd83ba/net/bridge/br_if.c
- CNI `cnitool`: https://www.cni.dev/docs/cnitool/
- CNI specification: https://github.com/containernetworking/cni/blob/v1.3.0/SPEC.md

Chapter 20: runc And containerd Experiments

The final lab group connects the raw Linux experiments to the runtime stack. runc consumes an OCI bundle and drives the kernel setup. containerd prepares image, snapshot, container, and task state, then talks to a runtime v2 shim.

Run these only in a disposable Linux VM. runc and containerd create real processes, mounts, cgroups, runtime directories, snapshots, and shim processes.

OCI Bundle

Question: what does runc need before it can create a container?

Scope: VM-only mutation.

runc expects an OCI bundle: a directory with `config.json` and a root filesystem. The README's basic flow is still the right mental model: populate `rootfs/`, generate a starter spec with `runc spec`, edit the spec, then run lifecycle commands.

The `spec` command source generates a starter spec and writes it as `config.json`:

```
spec := specconv.Example()
data, err := json.MarshalIndent(spec, "", "\t")
return os.WriteFile(specConfig, data, 0o666)
```

The lab should inspect these fields before running anything:

```
find bundle -maxdepth 2 -type f -o -type d | sort
sed -n '1,160p' bundle/config.json
grep -n '"path"\|"args"\|"namespaces"\|"mounts"' bundle/config.json
```

The rootfs source is a choice with consequences. A prepared static rootfs (a busybox tarball, for example) keeps the lab self-contained. Exporting an image with `docker create` and `docker export` matches the runc README but pulls Docker into the dependency list. Pick one and document it in the lab notes; a floating choice makes cleanup unreliable.

runc Lifecycle

Question: what is the difference between create and start?

Scope: VM-only mutation.

The runc commands name the split:

```
Name:  "run",
Usage: "create and run a container",
```

```
Name:  "create",
Usage: "create a container",
```

`runc run` is convenience. It creates, starts, waits, and cleans up depending on flags. The lifecycle lab should use `create`, `state`, `start`, `kill`, and `delete` so the state transitions are visible:

```
cd bundle
sudo runc create cdb-runc
sudo runc state cdb-runc
sudo runc start cdb-runc
sudo runc state cdb-runc
pid=$(sudo runc state cdb-runc | sed -n 's/.*"pid": *\([0-9][0-9]*\).*/\1/p')
sudo readlink /proc/"$pid"/ns/{mnt,pid,net}
sudo cat /proc/"$pid"/cgroup
sudo runc kill cdb-runc KILL
sudo runc delete cdb-runc
```

The process needs to stay alive long enough to inspect it, so set `process.args` in `config.json` to `["sleep", "300"]` and `process.terminal` to `false` before running `runc create`.

Cleanup verification:

```
sudo runc state cdb-runc || true   # nonzero = clean
```

containerd Task

Question: where does the shim appear when containerd starts a task?

Scope: VM-only mutation.

containerd's own docs say `ctr` is for debugging containerd. That is why this lab uses `ctr` instead of a friendlier container CLI.

Use a dedicated containerd namespace:

```
sudo ctr namespaces create cdb-lab 2>/dev/null || true
sudo ctr -n cdb-lab images pull docker.io/library/busybox:latest
sudo ctr -n cdb-lab containers create docker.io/library/busybox:latest cdb-task sleep 300
sudo ctr -n cdb-lab containers ls
sudo ctr -n cdb-lab tasks start -d cdb-task
sudo ctr -n cdb-lab tasks ls
```

Now inspect the runtime boundary:

```
ps -ef | grep -E 'containerd-shim-runc-v2|sleep 300' | grep -v grep
sudo ctr -n cdb-lab tasks ps cdb-task
```

The runtime v2 docs make the ownership clear:

```
containerd, the daemon, does not directly launch containers.
```

The shim invokes the OCI runtime, usually `runc`, and holds the task's control socket while containerd is restart-cycled.

Cleanup:

```
sudo ctr -n cdb-lab tasks kill cdb-task
sudo ctr -n cdb-lab tasks delete cdb-task
sudo ctr -n cdb-lab containers delete cdb-task
sudo ctr namespaces remove cdb-lab
```

If task deletion fails, inspect `ctr -n cdb-lab tasks ls` before forcing anything.

Bundle From containerd

Question: how does containerd's generated runtime bundle relate to the hand-built runc bundle?

Scope: VM-only mutation, mostly inspection after a task exists.

After creating a task, inspect containerd's runtime state directory for the lab namespace and task. The exact path depends on the containerd build and configuration, but the object to find is stable: a runtime v2 bundle containing an OCI `config.json` for the task.

The inspection target should include:

```
sudo find /run/containerd -path '*cdb-lab*' -o -path '*cdb-task*'
```

Once the bundle is found, compare its `config.json` to the runc bundle from the earlier lab. The generated spec should show where image config, snapshot mounts, runtime options, namespaces, cgroups, and task arguments ended up.

Events And Logs

Question: what can containerd tell us while the task is running?

Scope: inspect-only after the VM-only task setup.

`ctr events` can show lifecycle events while another shell creates and starts a task:

```
sudo ctr -n cdb-lab events
```

Run after the task lab; the events stream is empty without an active task. Debug logs and `strace` follow the same rule.

Sources And Further Reading

- runc README: https://github.com/opencontainers/runc
- runc `spec` command: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/spec.go
- runc `run` command: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/run.go
- runc `create` command: https://github.com/opencontainers/runc/blob/eb7eaf19b6eec5d1143b257057899e4a7b738c81/create.go
- OCI Runtime Specification bundle docs: https://github.com/opencontainers/runtime-spec/blob/6999a89a76a0329f440d5740497bedb9dd431297/bundle.md
- OCI runtime lifecycle: https://github.com/opencontainers/runtime-spec/blob/6999a89a76a0329f440d5740497bedb9dd431297/runtime.md
- containerd getting started: https://containerd.io/docs/getting-started/
- containerd runtime v2 docs: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/docs/runtime-v2.md
- containerd runtime v2 bundle handling: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/core/runtime/v2/bundle.go
- containerd runc v2 shim task service: https://github.com/containerd/containerd/blob/2976f38ccbfcda5ef1364d63d60b0a304e4bf94a/cmd/containerd-shim-runc-v2/task/service.go