Engineer

Container runtime

the program that actually starts a container

What it is

A container runtime is the software between a container image on disk and an isolated process in the kernel. The phrase covers two layers, and people often mean different ones: the low-level runtime, also called the OCI runtime, and the high-level runtime, more often called a container engine.

The low-level runtime does one narrow job. It reads a bundle, which the Open Container Initiative's runtime specification defines as a directory holding a `config.json` file and the container's root filesystem, asks the kernel for the namespaces, cgroups and mounts that the configuration describes, and starts the process inside them. The specification names the operations every runtime must support, namely state, create, start, kill and delete, and leaves the command line to each implementation. Version 1.0 was published in July 2017. runc, maintained by the OCI itself, calls itself "a low level tool not designed with an end user in mind". crun does the same job in C with a smaller memory footprint and is the default Podman's documentation names. youki is a third, written in Rust.

The engine sits above it. containerd describes itself as a daemon that manages the complete container lifecycle of its host: image transfer and storage, container execution and supervision, and the storage and network attachments around them. CRI-O does a similar job built for Kubernetes alone. An engine pulls and unpacks the image, writes the bundle, hands it to the low-level runtime, and keeps watching the process after that runtime has exited. Docker Engine sits one step higher still: its documentation says it uses containerd for the container lifecycle, and that containerd uses runc by default. Podman takes the other route and keeps no long-running daemon. Each command prepares the bundle itself, calls crun or runc, and leaves a small monitor called conmon in charge of the running container.

Kubernetes adds one more piece of vocabulary. The kubelet on each node talks to the engine over the Container Runtime Interface, CRI, a gRPC protocol that containerd and CRI-O both implement. Docker Engine predates CRI, so for years Kubernetes carried an adapter for it called dockershim, and removed it in v1.24.

Use the concept in a decision

Put “What is a container runtime?” to work in a real decision. We can help relate the concept to the system you are building or buying. Packaged to behave identically on a laptop, on a test server and in production, and to scale up when a busy week arrives.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Ran 20 Docker apps on Google Cloud and AWS — 20 containerised applications deployed and watched

Ran 20 Docker apps on Google Cloud and AWS Get in touch →

Why it matters

Choosing a container runtime is really choosing which layer to change, and the two layers buy different things.

The engine decides how containers live from day to day: who pulls and stores images, who restarts a crashed container, where its logs go, which API other tools speak, and who owns the process tree. Swapping Docker Engine for Podman on a server, or for containerd on a Kubernetes node, changes all of that and leaves the containers themselves alone, because the image format and the bundle are the same OCI standards underneath. The Kubernetes documentation on the dockershim removal says it plainly: images produced by `docker build` work with all CRI implementations.

The low-level runtime decides how isolated the process is. runc and crun start an ordinary process on the host’s kernel, which is fast and fully compatible, and means every container on the machine shares that one kernel. gVisor’s runtime, runsc, puts an application kernel written in Go between the container and the host and answers the container’s system calls itself; its own documentation names the price as reduced application compatibility and higher overhead per system call. Kata Containers boots each workload inside a lightweight virtual machine with a dedicated kernel, using hardware virtualization as a second layer of defence, and needs a hypervisor on each node that runs it.

Kubernetes lets the second choice be made per pod rather than per cluster. A RuntimeClass names a handler configured in the engine, and a pod that sets `runtimeClassName` runs under that handler, so an untrusted workload can sit in gVisor or Kata while everything else stays on runc. The documentation frames this as a balance of performance against security.

What it looks like in practice

In everyday speech the phrase drifts towards the engine layer, and the work on this site uses it in that sense.

Twenty Docker-packaged applications spread over two clouds and a few container runtimes had to deploy consistently and be diagnosable fast when something broke. Docker packaged them, troubleshooting went through Podman, and orchestration debugging went through Kubernetes. Working across engines like that leans on the layering above: the engines differ, and the OCI image and runtime standards beneath them are shared.

On a newer host the choice was made at the engine layer. The container plane runs on Podman and Quadlet in place of Docker, because the default engine publishes container ports above the firewall rules the hardening role installs, and because it would add a second supervisor beside systemd. What the choice changed is who starts each container and who watches it afterwards: containers became services that start the way services start, under the supervisor the host already trusted.

The same reasoning carries over to a 512 MB host for the company's public services, where memory settled every argument. Its container plane is Quadlet units under systemd rather than a daemon with its own supervisor, a choice made on a machine where about 464 MB of memory is usable.

Where it goes wrong

Runtime trouble tends to surface at the seams between layers, where each side assumes the other is set up the way it expects.

Node tooling goes blind after a cluster upgrade. From Kubernetes v1.24 the kubelet talks to containerd or CRI-O directly, so containers the cluster schedules never pass through Docker Engine. `docker ps` on the node lists only what Docker

itself started, and monitoring agents, startup scripts and privileged pods that call the Docker CLI or mount `docker.sock` lose their view. The workloads keep running. It is the tools around them that need moving to `crictl` or the engine's own API.

The kubelet and the runtime manage cgroups two different ways. Both must use the same cgroup driver, `cgroupfs` or `systemd`, and the kubelet defaults to `cgroupfs`. On a host where systemd is the init system, that leaves two cgroup managers with two views of the same resources, and the Kubernetes documentation warns that such nodes can become unstable under resource pressure. The fix is the systemd driver on both sides, which is also the recommended driver with cgroup v2. For containerd 2.x with runc, the runtime side looks like this:

```
# /etc/containerd/config.toml, containerd 2.x
[plugins.'io.containerd.cri.v1.runtime'.containerd.runtimes.runc.options]
  SystemdCgroup = true
```

The kubelet side is `cgroupDriver: systemd` in its configuration, and containerd needs a restart afterwards. containerd 1.x spells the section differently, and the Kubernetes page gives both forms. Changing the driver on a node that has already joined a cluster is delicate: pods created under one driver can hit errors when they are recreated under the other, so the documented advice is to replace or reinstall the node with the new configuration.

A pod moved to a sandboxed runtime stops at startup. A RuntimeClass whose handler the node's engine lacks sends the pod straight to the Failed phase, with the reason in an event. Past that, the workload has to suit the sandbox. gVisor answers system calls in its own kernel, so software that leans on less common kernel features may behave differently, and its documentation names system-call-heavy workloads as a poor fit. Kata needs hardware virtualization on the node, which on a cloud virtual machine means nested virtualization. A `scheduling` node selector on the RuntimeClass keeps such pods on the nodes that can run them.

An image imported by hand stays invisible to the kubelet. containerd keeps image names and metadata per namespace. Its `ctr` client works in the namespace `default` unless told otherwise, while the CRI plugin the kubelet talks to works in `k8s.io`. An image loaded with a plain `ctr images import` therefore

lands where the kubelet never looks, and the pod goes back to the registry or stops with an image pull error. Importing with `ctr -n k8s.io images import`, and checking with `crictl images`, which sees what the kubelet sees, avoids the gap.

Further reading

- OCI Runtime Specification — the standard every low-level runtime implements: the bundle, `config.json` and the lifecycle operations.
- runc — the OCI's own runtime, with notes on how higher-level tools drive it.
- Container Runtimes — cgroup drivers, CRI versions and the containerd and CRI-O settings Kubernetes expects.
- Container Runtime Interface (CRI) — the gRPC protocol between the kubelet and the engine.
- Runtime Class — picking a different runtime handler per pod, and keeping those pods on nodes that support it.
- Alternative container runtimes — how Docker Engine hands containers to gVisor, Kata Containers, youki or Wasmtime.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a container runtime?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/container-runtime/