

VMs can take up a lot of system resources. Each VM runs not just a full copy of an operating system, but a virtual copy of all the hardware that the operating system needs to run.

**Benefits of VMs**

All OS resources available to apps

Established management tools

Established security tools

Better known security controls

**Container:**

With containers, instead of virtualizing the underlying computer like a virtual machine (VM), just the OS is virtualized.

Sharing OS resources such as libraries significantly reduces the need to reproduce the operating system code, and means that a server can run multiple workloads with a single operating system installation.

Containers are thus exceptionally light.

Containers are implemented using [Linux namespaces](https://en.wikipedia.org/wiki/Linux_namespaces) and [cgroups](https://en.wikipedia.org/wiki/Cgroups).

* ***Namespaces*** let you virtualize system resources, like the file system or networking, for each container.
* ***Cgroups*** provide a way to limit the amount of resources like CPU and memory that each container can use.

**Docker:**

Docker was released in 2013 and solved many of the problems that developers had running containers end-to-end. It had all these things:

* A container image format
* A method for building container images (Dockerfile/docker build)
* A way to manage container images (docker images, docker rm, etc.)
* A way to manage instances of containers (docker ps, docker rm, etc.)
* A way to share container images (docker push/pull)
* A way to run containers (docker run)

At the time, Docker was a ***monolithic*** system. However, none of these features were really dependent on each other. Each of these could be implemented in smaller and more focused tools that could be used together.

They then broke out their code for running containers as a tool and library called [runc](https://github.com/opencontainers/runc) and donated it to OCI as a reference implementation of the [OCI runtime specification](https://github.com/opencontainers/runtime-spec).

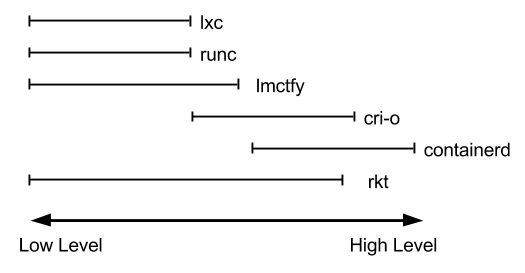
OCI spec just includes following three:

1. **Download the image**
2. **Unpack the image into a "bundle". This flattens the layers into a single filesystem.**
3. **Run the container from the bundle**

**Container Runtimes:**

Each runtime is built for different situations and implements different features.

Some, like containerd and cri-o, actually use runc to run the container but implement image management and APIs on top. You can think of these features -- which include image transport, image management, image unpacking, and APIs -- as high-level features as compared to runc's low-level implementation.



At the lowest level, container runtimes are responsible for setting up namespaces and cgroups for containers, and then running commands inside those namespaces and cgroups. ***Low-level runtimes*** support using these operating system features.

Developers need APIs and features around image formats, image management, and sharing images. These features are provided by ***high-level runtimes.***

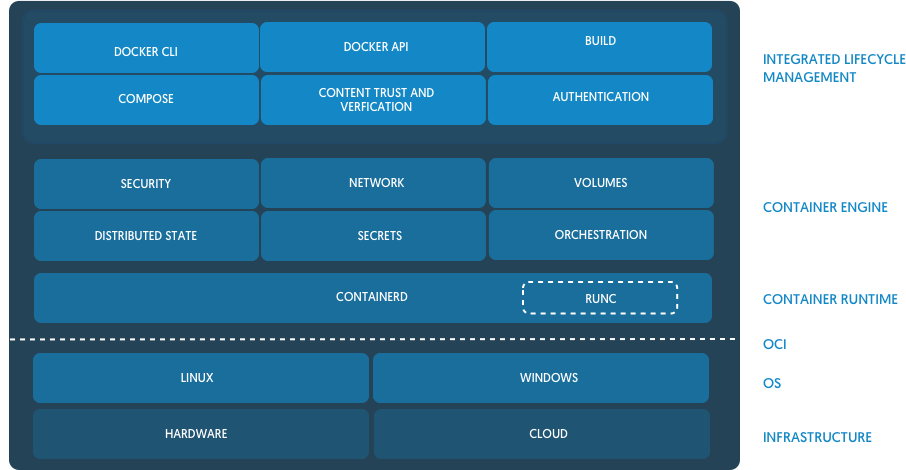
**Container Runtimes -- CONTAINERD**

Docker decided to split the monolith into separate parts, some of which other projects can even build on — that’s how [containerd happened](https://blog.docker.com/2016/04/docker-containerd-integration/).

Containerd is a daemon that acts as API for container runtimes and OS. When using containerd, you no longer work with syscalls, instead you work with higher-level entities like snapshot and container — the rest is abstracted away.

Abstraction layer between your management code and the syscalls to run a container.  That is where containerd lives.  It provides a client layer of types that platforms can build on top of without ever having to drop down to the kernel level

containerd manages the complete container lifecycle of its host system, from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond.



Containerd is for container platform that wants to abstract away syscalls or OS specific functionality to run containers. It uses **runc** for interaction with underlying OS.

networking are out of scope for containerd.

**Container Runtimes** – **CRI-O**

They do not have to modify the kubernetes code base for each new container runtime because they created an API to define calls that it would make into container runtimes, thus the Container Runtime Interface (***CRI***) was born. Interaction between kubernetes and a given runtime is performed through the CRI API

CRI-O leverages all of the OCI standards:

* Runs containers using the OCI Runtime tools defaulting to runc.
* Managing container images following the OCI image specification.
* Uses the OCI-Runtime-tools for generating the OCI Runtime Specification
* CNI for setting up the container networking.
* containers/image for pulling container images from container registries like docker.io

CRI-O defaults to running containers with runc, exactly the same as Docker does today: running containers with runc.

CRI-O is a specific container runtime dedicated to Kubernetes work loads.

There’s no big Docker daemon in CRI-O, although it uses runC under the covers which Docker created and maintains.

Some interesting [benchmarks](https://gist.github.com/kunalkushwaha/66629a90e0f8f5cc5dc512ef1c346f2f) point to Containerd being faster than CRI-O.