Comparison of Operating-System-Level Virtualization Frameworks for Linux

Seminar Operating Systems and Software Engineering
360.037

TU Wien

Stefan Lendl
00928895

under the supervision of
O.Univ.Prof. Dipl.-Ing. Dr.techn. Dr.h.c. Siegfried Selberherr
Univ.Ass. Dipl.-Ing. Dr.techn. Josef Weinbub, BSc

Abstract

In the last decade operating-system-level virtualization or more commonly known as *containers* have become a very important technological component in various solutions like cloud computing. Especially the flexibility and ease-of-use of containers in cloud computing environments lead to a rapid adoption in this industry and promoted its enhancement. The recent standardization of containers within the industry allowed the development of further specialized solutions.

This work presents and compares key characteristics of the most common container solutions for Linux: Docker, OpenVZ, LXC/LXD, rkt, runC and Kata Containers.

Contents

1 Introduction **4**
Open Container Standardization 5
Technical Background of Containers in Linux 5

2 Container Frameworks **7**
OpenVZ 7
LXC/LXD 7
Docker 8
runC 10
rkt 11
Kata Containers 12
Other Frameworks 14

3 Comparison **15**

4 Conclusion **17**

References **18**

1 Introduction

In order to optimize the hardware utilization multiple services operate on the same physical server, sharing the hardware resources.

Several ways of sharing hardware resources are available which significantly differ in terms of isolation:

- **Shared hosting** where services run alongside each other or even on the same physical server.
- **chroot** is an operating system (OS) kernel system call that changes the apparent root directory of a process to a subdirectory of the host.
- **Full system virtualization** isolates the service by executing it inside its own virtual machine (VM) which fully emulates the entire hardware.
- **Operating-system-level virtualization** or **Containers** are a set of OS kernel features to isolate individual processes.
- **Hybrid VM containers** is a new development of combining the benefits of isolation provided by VMs and the flexibility of containers.

The first approach of shared hosting does not offer any significant isolation between processes, which is an important measure to protect services against security violation and escalation in other services on the same host. Furthermore, installing different versions of applications or libraries in a host system is typically difficult because common package managers like DPKG [1] and RPM [2] do not allow the installation of multiple versions in the system.

When using the system call chroot to create a so called jail, the effected process can only access files within this subdirectory. This allows isolation between processes on the filesystem level and effortless installation of different library versions on the same system. Several ways to break out of such a jail exist [3] and it is therefore not considered secure.

The development of full system virtualization revolutionized the hosting industry and supported the evolution of cloud computing. The execution in individual virtual machines offers maximum isolation. Inside the VM another full OS is running with all its system processes and services. VMs add a significant CPU and network performance overhead but because multiple instances can run on a single physical machine the resource is utilized more efficiently. The hypervisor which manages the VMs can

segment the hardware resources and apply resource quotas to the individual VM to adapt the share of resources.

In the last decade OS-level virtualization or containers as described in [4] have become increasingly popular. The container accesses the same OS kernel as the host instead of emulating the hardware. The isolation is only done through the segmentation features of the kernel and containers have less performance overhead than VMs and near-native startup time [5].

OS-level virtualization imposes a small overhead compared to native execution and offers better or equal performance than virtual machines [6]. For example the CPU overhead introduced by Docker was measured to be 5% to 10% whereas the I/O overhead ranges from 10% to 30%.

One of the main reasons for containers is their lightweight design and simple deployment mechanisms through so-called orchestration software that manages multiple containers typically across multiple servers in multiple datacenters. The lightweight design of containers improves the performance of common operations like starting, stopping or migration from one server to another. The OS kernel is considered a single-point-of failure and vulnerabilities in the kernel may allow malicious access from a container to the host or other containers [7].

The development of so-called hybrid VM containers combines virtual machines with containers by executing a container runtime in an extremely lightweight VM engine. Through complying to the open container standards, these containers can be integrated into flexible container orchestration software like Kubernetes [8].

Open Container Standardization

The Open Container Initiative (OCI) [9] – formed under the Linux Foundation – and the Cloud Native Computing Foundation (CNCF) [10] define the following [11]:

- Image Specification [12] defines the content of container images.
- Runtime Specification (CRI) [13] describes the configuration, execution environment and lifecycle of a container.
- Container Network Interface (CNI) [14] specifies how to configure network interfaces inside containers, which was standardized by the CNCF.

Technical Background of Containers in Linux

The main mechanisms to implement container isolation are *namespaces* and control groups (*cgroups*). For security purposes secure computation (*seccomp*), capabilities and mandatory access control (MAC) mechanisms of the kernel are used. These mechanisms

have been implemented into the Linux kernel over the years which strongly influenced by the growing container infrastructure.

Namespaces define what a process can see by creating a different view on the system [15]. For example a process inside a container may see its process ID (PID) as 1 while it is mapped to a different PID outside of the container. Namespaces are for example also available for mount points, inter-process communication (IPC), network resources and users.

Cgroups define which resources a process can use [16]. With cgroups – amongst other things – the CPU or memory usage of a process can be limited. Other cgroups include device access, network and I/O throughput.

With seccomp a process can transition to a secure state where it can call only a limited subset of kernel system calls [17]. The white-listing of system calls are done with filters that also consider the arguments of the system call.

MAC allows fine-grained control of permissions that are much more powerful than the traditional file-based permissions model. These rules are used with containers to additionally restrict the access of processes inside the container. There are various implementations for MAC and most commonly SELinux and AppArmor are used in conjunction with containers.

2 Container Frameworks

This section presents various container frameworks for Linux in chronological order of their release which are all under active development. Containerization exist for Linux, FreeBSD, Solaris and Windows.

OpenVZ

OpenVZ [18] was one of the first container solutions on Linux. It was released 2005 as part of Virtuozzo [19] and is still under development although mainly used inside Virtuozzo. The full feature set of OpenVZ is only available with a custom kernel. Many of its features were merged upstream in the Linux kernel which significantly benefited the development of Linux container technology.

OpenVZ containers are so-called system containers, in contrast to application containers of Docker and similar solutions. System containers are closer to a full VM with a full init process, starting other system processes. An application container – like in Docker – typically starts only the actual process without any background processes running inside the container.

OpenVZ has a feature called checkpointing with CRIU [20], which allows a container to stop during execution. This is a useful feature for live migration of containerized applications with slow startup time. These containers can be stored in the already started state. This is useful – for example – if an additional instance of an application has to be started up dynamically.

LXC/LXD

After the Linux mainline kernel added cgroups in the year 2007, LXC [21] was released in 2008 which was the first implementation of container management for Linux without requiring any patches. It is mainly based around the concepts of cgroups and namespaces.

LXC offers stateful snapstots and live migration by using CRIU from the OpenVZ project.

The Linux Container Daemon (LXD) [22] adds a layer on top of LXC. It is developed by Canonical and was first released in October 2015. In contrast to Docker, the focus is not on containerizing a single application but having a full OS running inside.

LXD containers are system containers where the entry point for the container is not an application like in Docker but a full OS init system like systemd [23]. The focus of LXD is to have a full OS running inside. LXD aims to be used in a cloud environment as a replacement for VMs.

To manage the containers, LXD offers a secure application programming interface (API) to allow management – for example – through cloud orchestration software like Canonical's own Juju [24], OpenStack [25] or Kubernetes [26]. LXD does not offer any network or storage management leaving that to the orchestration on top of it. On the other hand it offers live migration of containers to another host which is essential for dynamic load distribution in a cloud infrastructure.

Docker

With the release of Docker [27] in 2013 the container mechanisms had its break-through and Docker is still the most popular container solution. One of the philosophies behind Docker's success was to hide complexity behind simplicity. This is especially true for the simplicity of the Command Line Interface (CLI) of Docker that significantly simplified initialisation and execution of the previously complex container management tasks.

Docker is not a standalone container runtime but consists of many software components with different purposes. Many of these components have been extracted in order to collaborate with other projects and support further development.

Docker uses a central daemon that handles the various running containerized processes and allows configuration by the user. Together with the configuration interface this is called the Docker Engine [28].

Initially Docker used LXC as its container runtime and later developed its own runtime (libcontainer) which was later extracted as runC.

Docker had frequent API changes which was difficult to integrate in other systems which favor a stable API. This lead to the development of containerd as a standalone daemon with stable API which was donated as well to the OCI. Later, Docker Engine also used containerd, which uses runC as shown in Figure 2.1.

The Docker architecture is designed around the concept that a single container only runs a single process. For example, a container runs the web server, another container runs the database and yet another container runs a caching proxy in front of the service. Many of these application containers *images* are available as ready-to-use images that can be downloaded from a *registry*. Once the images is started it is referred to as a container.

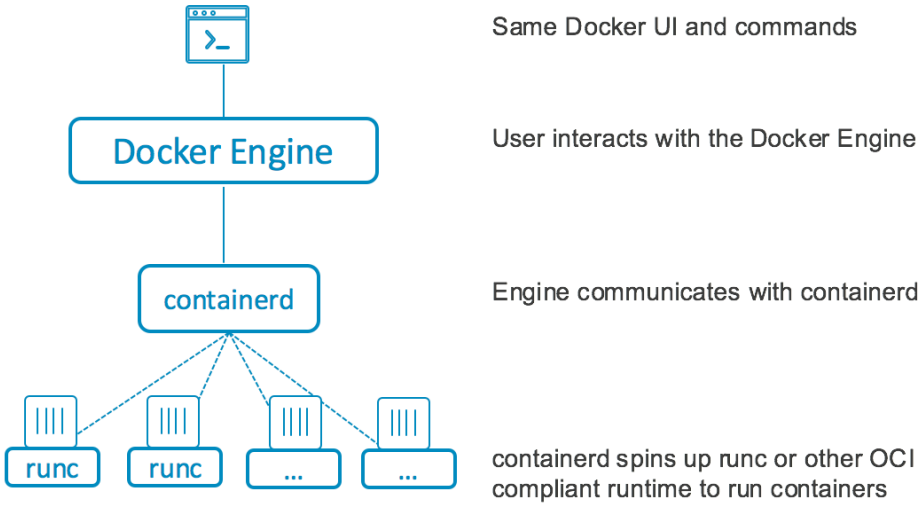

Figure 2.1: The Docker Engine architecture [29].

Docker containers and images use a layered approach where each layer only contains the differences to the layer underneath [30]. As shown in Figure 2.2, an image consists of various read-only layers and the execution of the current container is done in an additional read-write layer on top of the image layers. With this layered concept images can be used by multiple containers. The read-only layers are stored on the system only once and each executed container has an individual read-write layer. The same is true for image sub-layers where multiple layers can inherit the same a base layer.

In Docker the life-time of a container matches the life-time of the process that is executed inside. If the process terminates, the container terminates with it. If a Docker container is deleted all changes made inside the container are deleted as well. In order to store the changes in a container a snapshot of the current state can be taken and stored as a new image with a new layer that can again be started up. Furthermore, consistent changes like the content of a database can be mounted from outside the container without modifying the data-independent image.

Docker images can be built using a Dockerfile that starts off at a specified image and specifies instructions to be executed inside the container before the resulting image is saved. With this method images can be produced automatically at – for example – a new release of the application packaged inside the Docker image.

Docker supports multiple network drivers [31] which allow networking interfaces to act as a *bridge*, where the virtual Docker network is separated from the host network; *host*, where the container has direct access to the hosts network interface; *overlay*, where a virtual network is created between multiple hosts that run the Docker daemon and *macvlan*, where each container gets a unique address on the host's network. Docker also

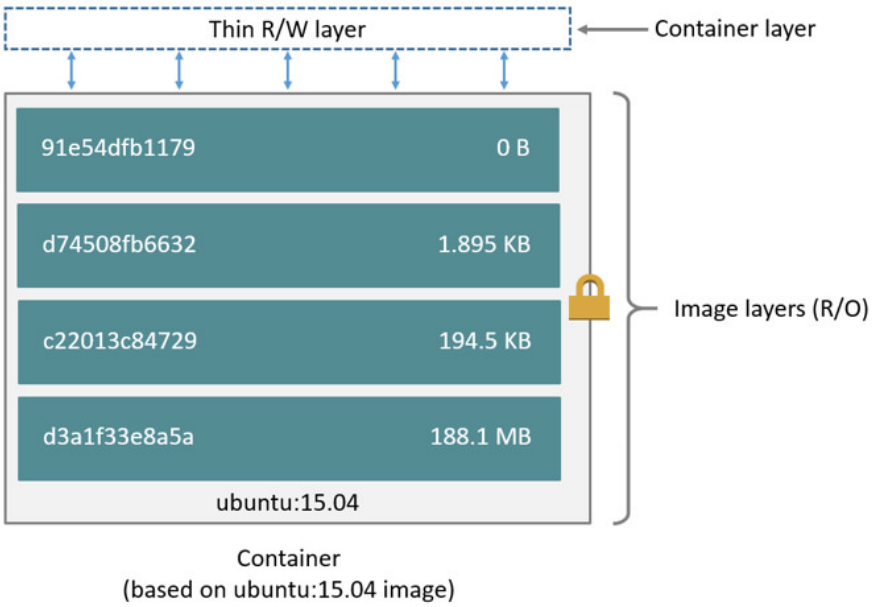

Figure 2.2: Docker container layers.

allows third-party networking plugins to be used.

runC

RunC [32] is a lightweight container runtime without a daemon, which originated as a software component of Docker and is now developed by the OCI. It gives the user full control over the execution of the container, which allowed the development of other solutions on top of containers that use runC as their container runtime.

An example of software on top of runC is CRI-O [33] – released in October 2017 – which is a lightweight and simple container runtime developed by Red Hat as a minimal container runtime for Kubernetes. CRI-O is designed to be simple by following the Unix philosophy of *doing one thing and doing it well* [34]. The goal for CRI-O is to have maximum compatibility with Kubernetes and provide a stable interface.

Another example is Garden [35], which is the containerization layer in the CloudFoundry orchestration software [36]. Garden started by using LXC and later moved to runC as their container runtime.

rkt

Rkt [37] (pronounced "rocket") is an open-source container engine developed by CoreOS project. Rkt offers encryption and signing of images [38]. In addition to rkt's native image format it can execute Docker images.

Rkt focuses on application containers, where ideally only a single process is executed inside a container. Rkt does not have a centralized daemon for configuration and systemd is used to start containers directly from the Linux init process. This eliminates the single point-of-failure of having a centralized daemon like Docker. Figure 2.3 shows the execution hierarchy of rkt compared to Docker.

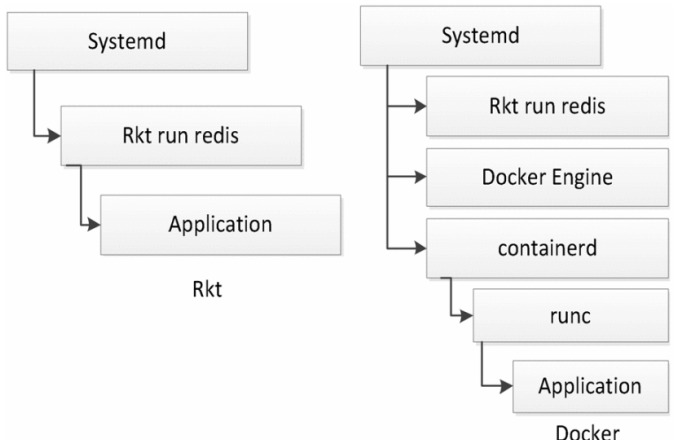

Figure 2.3: Execution hierarchy in Docker and rkt [39].

Rkt project uses the App Container standard (appC) [40] which was later included in the OCI standards. The appC standard is similar to the OCI standards with a few differences [41].

AppC uses a so-called *pod* as the basic unit of execution. A pod is a group of one or more services that is logically related [42]. The concept of pods in rkt is identical to the concept of pods in Kubernetes [43]. Rkt became the first non-Docker runtime to be supported by Kubernetes [44].

The rkt execution workflow as shown in Figure 2.4 describes three stages. After the pod is started from various sources – the command line, systemd or Kubernetes – the container runtime is set up in stage 1. Different execution environments for stage 1 are available in rkt. The default options are:

- **systemd-nspawnd** uses the cgroups and namespaces invoked from systemd.
- **fly** executes the process in a chroot without further isolation.
- **KVM** uses a fully virtualized environment.

Other options for stage 1 are available as plugins. In stage 2, the process inside the

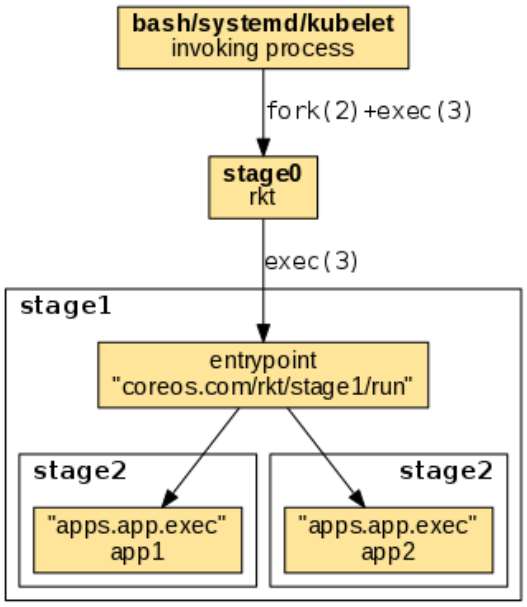

Figure 2.4: Rkt execution workflow [45].

container is launched.

Rkt is currently changing its internal architecture to be compliant to the OCI standard [43] and is currently developing the integration of runC as a stage 2 execution environment. The current status of rkt's OCI transition is described in [46].

To build rkt images scripts are available that are similar to Dockerfiles.

Rkt uses systemd-journald for logging which integrates directly with the logging infrastructure of the host system.

Although, rkt may arguably have a better architecture than other solutions, it seems that developers have stopped actively developing rkt when CoreOS was bought by Red Hat in Spring 2018. There is no official announcement but the last GitHub commit on rkt was in May 2018.

Kata Containers

Kata containers [47] – released in 2018 – is an OCI-compliant hybrid VM container runtime that executes containers within a VM.

Kata Container originated from merging of the two projects Hyper runV [48] and Intel Clear Containers [49], which focused on securing the runtime environment by executing the container inside the Kernel Virtual Machine (KVM) runtime [50].

Kata containers is developed under the OpenStack foundation and provides a virtualized isolation layer to make container execution more secure [51]. The project is supported by several companies to accelerate development and hardware support.

Kata Container integrates into Docker's containerd with their OCI-compatable runtime and it can be used with Kubernetes in different ways. Either through containerd, CRI-O or directly through the CRI layer with Frakti [52]. Figure 2.5 shows the different integration options of Kata containers into Kubernetes.[1] Figure 2.6 shows how Kata containers can be executed in Kubernetes under CRI-O alongside other OCI compliant containers.

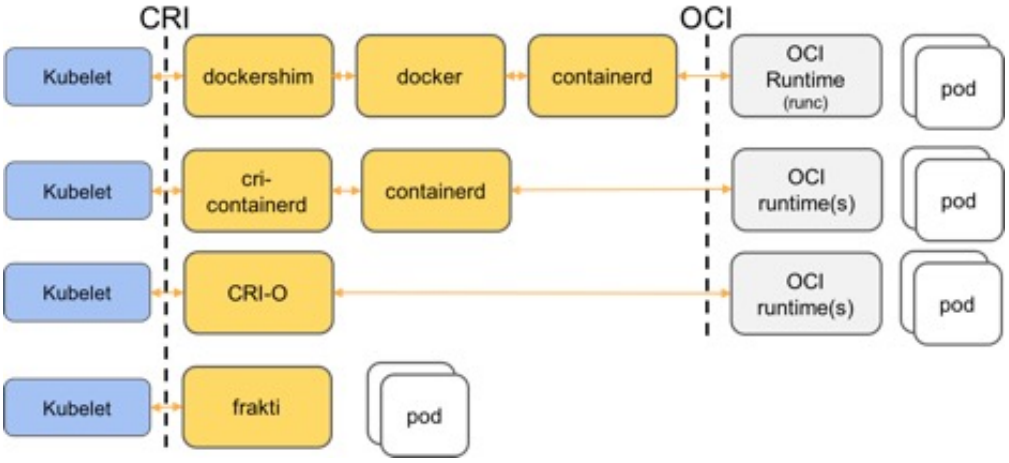

Figure 2.5: Kata Container intergration into Kubernets [53].

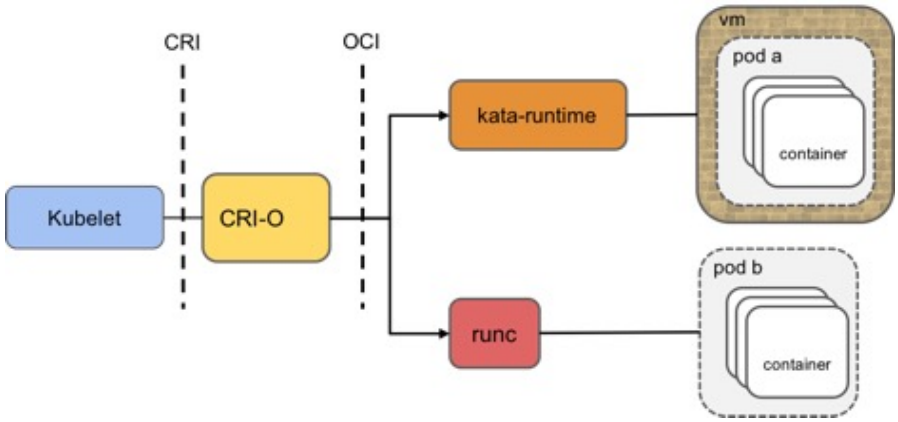

Figure 2.6: Kata Container intergration into Kubernets with CRI-O [53].

Kata Containers uses a minimal version of QEMU for virtualization. For faster startup time of the VM container other virtualization engines have been added. Noticeably,

[1]This shows the current development trend towards open standardization compliance on all layers.

Firecracker MicroVM [54] is available with Kata Containers as of version 1.5[2] [55]. Firecracker MicroVM is a minimalistic virtual machine developed by Amazon as a faster replacement for QEMU. Firecracker only emulates 4 devices for networking, block devices, serial console and a 1-button keyboard controller used only to stop the microVM. This minimalistic design enables a startup time of less than 125ms.

Other Frameworks

There are many other solutions available. Some noticeable projects are:

- systemd-nspawnd [56] is part of systemd and can manage containers directly from systemd.
- gVisor [57] attempts to secure containers by providing a user space kernel abstraction which processes the system calls used by the container process.
- Nabla containers [58] tries to secure containers by restricting the kernel system calls with a strict seccomp profile that allows only 7 system calls.
- Singularity [59] is a container framework intended for scientific computing where operations are executed on many servers in a high performance cluster.

[2]released Jan 22, 2019

3 Comparison

This section compares various characteristics of the previously introduced container frameworks and the availability of their frameworks in different orchestrator solutions.

Table 3.1: Comparison of container framework characteristics.

Framework	Release	Isolation	Sys/App container	Central daemon	Live migration	Resource quotas	OCI compliant
OpenVZ	2005	Kernel	System	no	yes	yes	no
LXC	2008	Kernel	System	no	yes	yes	no
Docker	2014	Kernel	App	yes	yes	yes	yes
LXD	2016	Kernel	System	yes	yes	yes	no
runC	2016[1]	Kernel	App	no	yes	yes	yes
rkt	2016	Kernel	App	no	no	yes	partial
Kata Containers	2018	VM	App	no	no	yes	yes

Table 3.1 presents some characteristics of the presented container frameworks. It shows the release year of the framework and the underlying isolation strategy. Traditional container frameworks offer isolation through Linux Kernel mechanisms and Kata Containers offers isolation through full system virtualization. Some containers are designed as system containers while other – newer frameworks, starting with Docker – are application containers. The only two container frameworks with a centralized daemon are Docker and LXD, whereas LXD is essentially a daemon for LXC. All frameworks but rkt support live migration while the container solutions use CRIU. Kata containers does not yet support a snapshotting feature. All container frameworks allow the specification of CPU, network, I/O and memory quotas and are therefore not listed separately. Not all frameworks are OCI compliant because OCI standardization focuses on application containers. Docker – which uses runC internally – is OCI compliant. Rkt does not fully comply to the OCI standard and Kata containers was developed after the standardization and is one of the first container frameworks that focuses on OCI compliance from the start.

[1] Stable version 1.0 has not yet been released.

Table 3.2: Availability of container framework in various orchestrators.

Orchestrator→ Container ↓	Kubernetes	OpenStack	CloudFoundry	Mesos[2]	Virtuozzo
OpenVZ	no[3]	no[3]	no	no	yes
LXC/LXD	yes	yes	no	no	no
Docker	yes	yes	yes	yes	yes[4]
runC	yes	yes	yes	no	no
rkt	yes	yes	no	no	no
Kata	yes	yes	no	no	no

An interesting distinction between container frameworks is their integration into orchestration software. OpenVZ was developed inside Virtuozzo with a separate Linux kernel and some of its features are not available upstream. It is therefore only available in Virtuozzo. On top of Virtuozzo it is possible to deploy Kubernetes and OpenStack and it is possible to run Docker inside OpenVZ which makes it available in Virtuozzo. All other frameworks are directly or through a shim[5] layers available in Kubernetes and OpenStack. CloudFoundy supports Docker and integrated runC into there containerization engine. Apache Mesos [61] offers support for Docker alongside their own containerization implementation.

[2]Mesos has its own containerization engine.
[3]On top of Virtuozzo.
[4]Inside OpenVZ containers.
[5]A shim is *"a small piece of software that is added to an existing system program or protocol in order to provide some enhancement."* [60]

4 Conclusion

Docker made containers popular by significantly simplifying container management through their easy-to-use CLI and API and large selection of readily available images. Docker's central daemon – containerd – has a modular design which allows integration into orchestrator software and allows the underlaying execution runtime to be replaced with another runtime. For example Kata containers – even with Firecracker MicroVM – can directly be used with the same Docker API.

Through the standardization, the industry has agreed on a single set of standards which was mainly pushed by Docker. The standardized stand-alone runtime runC is directly integrated into other solutions that need to have full low-level control of the container.

This makes Docker the best choice for simple use when directly interacting with the container runtime through the API. RunC is the best options when control over the runtime is required for full integration into other solutions.

Due to security concerns, most container runtimes in the cloud are executed inside a VM which eliminates the performance advantage of containers over VMs. Kata Containers optimizes this by controlling the virtualization layer and directly integrating into the hypervisor. By focusing on integrability into other solutions it allows fast adoption of the technology.

References

[1] "Debian Package Manager." https://wiki.debian.org/Teams/Dpkg/.

[2] "RPM Package Manager." https://rpm.org/.

[3] "Breaking Out of a chroot() Padded Cell." https://web.archive.org/web/20160127150916/http://www.bpfh.net/simes/computing/chroot-break.html.

[4] J. P. Walters, V. Chaudhary, M. Cha, S. Guercio Jr, and S. Gallo, "A Comparison of Virtualization Technologies for HPC," in *Advanced Information Networking and Applications*, 2008, pp. 861–868.

[5] J. S. Hale, L. Li, C. N. Richardson, and G. N. Wells, "Containers for Portable, Productive, and Performant Scientific Computing," *Computing in Science Engineering (CiSE)*, vol. 19, no. 6, pp. 40–50, Nov. 2017.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," in *Performance Analysis of Systems and Software*, 2015, pp. 171–172.

[7] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A Measurement Study on Linux Container Security: Attacks and Countermeasures," in *Computer Security Applications Conference*, 2018, pp. 418–429.

[8] "Updates in Container Isolation," *LWN.net*. https://lwn.net/Articles/754433/.

[9] "Open Containers Initiative." https://www.opencontainers.org/.

[10] "Cloud Native Computing Foundation." https://www.cncf.io/.

[11] "Demystifying Container Runtimes," *LWN.net*. https://lwn.net/Articles/741897/.

[12] "OCI Image Format," *GitHub*. https://github.com/opencontainers/image-spec.

[13] "OCI Runtime Specification," *GitHub*. https://github.com/opencontainers/runtime-spec.

[14] "Container Network Interface: Networking for Linux Containers," *GitHub*. https://github.com/containernetworking/cni.

[15] "Namespaces Article Index," *LWN.net*. https://lwn.net/Articles/766124/.

[16] "Understanding the New Control Groups Api," *LWN.net*. https://lwn.net/Articles/679786/.

[17] “A seccomp Overview,” *LWN.net*. https://lwn.net/Articles/656307/.

[18] “OpenVZ - Open Source Container-based Virtualization for Linux.” https://openvz.org/.

[19] “Hyperconverged Infrastructure Software Provider.” https://www.virtuozzo.com/.

[20] “CRIU.” https://criu.org/Main_Page.

[21] “Linux Containers - LXC - Introduction.” https://linuxcontainers.org/lxc/introduction/.

[22] “Linux Containers - LXD - Introduction.” https://linuxcontainers.org/lxd/.

[23] “Systemd,” *Freedesktop.org*. https://freedesktop.org/wiki/Software/systemd/.

[24] “Jujucharms | Juju.” https://jujucharms.com/.

[25] “Openstack: Build the Future of Open Infrastructure.” https://www.openstack.org/.

[26] “Kubernetes: Production-Grade Container Orchestration.” https://kubernetes.io/.

[27] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” *Linux Journal*, vol. 2014, no. 239, Mar. 2014.

[28] P. Eder, “An Infrastructure Agnostic Application Deployment Framework for the Internet of Things,” PhD thesis, TU Wien, 2017.

[29] “Docker 1.11: The First Runtime Built on Containerd and Based on Oci Technology,” *Docker Blog*. https://blog.docker.com/2016/04/docker-engine-1-11-runc/.

[30] “About Storage Drivers,” *Docker Documentation*. https://docs.docker.com/storage/storagedriver/.

[31] “Docker Networking Documentation,” *Docker Documentation*. https://docs.docker.com/network/.

[32] “runC: CLI Tool for Spawning and Running Containers According to the OCI Specification.” *GitHub*. https://github.com/opencontainers/runc.

[33] “CRI-O.” https://cri-o.io/.

[34] “KubeCon + CloudNativeCon North America 2017: CRI-O: All the Runtime Kubernetes Needs,...” https://kccncna17.sched.com/event/CU6T/cri-o-all-the-runtime-kubernetes-needs-and-nothing-more-mrunal-patel-red-hat.

[35] “Garden | Cloud Foundry.” https://docs.cloudfoundry.org/concepts/architecture/garden.html.

[36] “Cloud Foundry - Open Source Cloud Application Platform.” https://www.cloudfoundry.org/.

[37] “CoreOS rkt.” https://coreos.com/rkt/.

[38] “CoreOS rkt Image Signing and Verification Guide.” https://coreos.com/rkt/docs/latest/signing-and-verification-guide.html.

[39] X.-L. Xie, P. Wang, and Q. Wang, “The Performance Analysis of Docker and rkt based on Kubernetes,” in *International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery*, 2017, pp. 2137–2141.

[40] “App Container Specification and Tooling.” *GitHub*. https://github.com/appc/spec/blob/master/SPEC.md.

[41] “App Container Specification and Tooling. README,” *GitHub*. https://github.com/appc/spec/blob/master/README.md.

[42] “CoreOS rkt App Container Basics.” https://coreos.com/rkt/docs/latest/app-container.html.

[43] “What Kubernetes Users Should Know About the rkt Container Engine.” https://coreos.com/blog/rkt-and-kubernetes.html.

[44] “Rktnetes Brings rkt Container Engine to Kubernetes.” https://kubernetes.io/blog/2016/07/rktnetes-brings-rkt-container-engine-to-kubernetes/.

[45] “CoreOS rkt Architecture Documentation.” https://coreos.com/rkt/docs/latest/devel/architecture.html.

[46] “CoreOS rkt OCI Native Support | Current Project State.” *GitHub*. https://github.com/rkt/rkt/projects/4.

[47] “Kata Containers - the Speed of Containers, the Security of VMs.” https://katacontainers.io/.

[48] “Hyper - Make VM run like Container.” https://www.hypercontainer.io/.

[49] “Intel Clear Containers,” *GitHub*. https://github.com/clearcontainers/runtime.

[50] “Intel, Hyper.Sh Merge Tech into Kata Containers,” *ICT Monitor Worldwide*, Dec. 2017.

[51] “Openstack Boosts Container Security with Kata Containers 1.0,” *ICT Monitor Worldwide*, May 2018.

[52] “Frakti - The Hypervisor-based Container Runtime for Kubernetes,” *GitHub*. https://github.com/kubernetes/frakti.

[53] “Kata Containers - Why Kata Containers Doesn’t Replace Kubernetes.” https://katacontainers.io/posts/why-kata-containers-doesnt-replace-kubernetes/.

[54] “Firecracker MicroVM.” https://firecracker-microvm.github.io/.

[55] “Kata Containers Documentation | Initial Release of Kata Containers with Firecracker Support,” *GitHub*. https://github.com/kata-containers/documentation/wiki/Initial-release-of-Kata-Containers-with-Firecracker-support.

[56] "Systemd-nspawn," *Freedesktop.org.* https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html.

[57] "gVisor - Container Runtime Sandbox." *GitHub.* https://github.com/google/gvisor.

[58] "Nabla Containers - A New Approach to Container Isolation." https://nabla-containers.github.io/.

[59] "Singularity," *Sylabs.io.* https://www.sylabs.io/singularity/.

[60] A. Freedman, *The Computer Desktop Encyclopedia*, 1st ed. New York, NY, USA: American Management Assoc., Inc., 1996.

[61] "Apache Mesos." http://mesos.apache.org/.