Comparison of Operating System Level Virtualisation Frameworks on Linux

Seminar Operating Systems and Software Engineering
360.037

TU Wien

Stefan Lendl BsC
O.Univ.Prof. Dipl.-Ing. Dr.techn. Dr.h.c. Siegfried Selberherr
Univ.Ass. Dipl.-Ing. Dr.techn. Josef Weinbub, BSc

February 2018

In the last decade Operating System Level Virtualisation or more commonly known as containers have become a very important technology for Cloud computing. Especially the flexibility and ease-of-use of containers lead to a rapid adoption in the industry. The recent standardization of containers within the industry allowed the development of further specialized solutions and the integration into powerful orchestration software which typically used VMs.

The leading containerization software Docker is just one of many available solutions. This paper presents and compares key characteristics of the most common container solutions on Linux: Docker, OpenVZ, LXC/LXD, rkt, runC and Kata Containers.

Contents

1 Introduction

This section presents the various types of virtualization used for sharing physical resources; presents the open container standards; introduces the features of the Linux kernel which are used to implement Containers and references recent scientific work on this topic.

Typse of Virtualisation

Historically every service ran on its own physical server which lead to inefficient use of resources when the service did not utilize the full processing power of the server. In order to optimize the hardware utilization multiple services shall be operated 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 web 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 virtualisation** isolates the service by executing it inside its own virtual machine (VM) which fully emulates the entire hardware.
- **Operating System Level Virtualisation** 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 significantly isolation between the processes and does not offer additional protection against security threats. Installing different versions of libraries is typically difficult. 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 filesystem level and effortless installation of different library versions on the same system. There are several ways to break out of such a jail [1] and it is commonly no considered secure.

The development of full system virtualization revolutionized the hosting industry and allowed the evolution of Cloud computing. The execution in individual virtual machines offers the maximum isolation between the processes. 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 overall the resource is utilized more inefficiently. The hypervisor which manages the virtual machines can segment the hardware resources and apply resource quotas to the individual machines to adapt the share of resources to various needs.

In the last decade OS-level virtualization or containers as described in [2] 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 much fast start up time. CITE?

One of the main reasons for containers is their lightweight design and simple deployment mechanisms through so orchestration software that manages containers on various physical locations. Due to the lightweight design quickly adding or removing containers or moving them from one location to another is a common operation in Cloud computing. 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 [3]. A former Docker developer claims that containers are safe with the appropriate settings [4]. Although configuring these settings is difficult especially for general purpose.

The recent development of what the Author calls 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.

Open Container Standardization

In order to standardize the different implementations of containers several companies collaborate. The Open Container Initiative (OCI) [5] was formed under the Linux Foundation as an effort to define such standards. The Standardization is also activly driven by the Cloud Native Computing Foundation (CNCF) [6]. These are the most important standards [7]:

- the Image Specification [8] which defines the content of container images
- the Runtime Specification (CRI) [9] describes the “configuration, execution environment, and lifecycle of a container”
- the Container Network Interface (CNI) [10] 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 strongly influenced by the growing container infrastructure.

namespaces as extensively described in [11] defines what a process can see by creating a different view on the system. For example a process inside a container may have process ID (PID) 1 which 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 [12] define which resources a process can use. 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 [13] a process can transition to a secure state where it can call only a limited subset of kernel system calls. The white-listing of system calls are done with filters that also consider the arguments of the system call.

Mandatory access control (MAC) allows fine-grained permissions that are much more powerful than traditional permissions based on file permissions. 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 used AppArmor are used in conjunction with containers.

2 Recent Work

According to researchers in [14] OS-level virtualization imposes a small overhead compared to native execution and offers better or equal performance than virtual machines. The impact on CPU and memory performance is small while the I/O and network performance is degraded due to the more complex network stack. Researchers in [15] measured the performance overhead introduced by Docker. With a CPU overhead was shown to be 5% to 10% and the I/O overhead ranges from 10% to 30%.

Researches in [16] compared Docker and rkt in Kubernetes and showed that Docker has a better CPU performance and rkt a better disk write performance. Researchers in **???** analyzed the impact of running Docker on top of KVM and showed a small overhead compared to execution directly in the VM. On ARM architecture virtual machine show a better performance for I/O tasks due to the caching mechanisms in the hypervisor [17].

Several approaches to increase the container security are proposed in [18],[19],[20]. They all focus on additional security layers to enforce a stronger segmentation between containers and the host.

3 Container Frameworks

This section presents various container frameworks for Linux in chronological order of release are all under active development. Other containerization solutions exist for Linux, FreeBSD, Solaris and Windows but these are not investigated in this paper.

OpenVZ

OpenVZ [21] was one of the first container solutions on Linux. It was released 2005 as part of Virtuozzo [22] and is still under development although mainly use inside Virtuozzo. The full feature set of OpenVZ is only available with a separate kernel. Many of its features were merged upstream in the Linux kernel significantly benefited the development of Linux container technology.

OpenVZ containers are so-called system container 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 [23] which allows a container to stop during execution. This is a useful feature for live migration of containers and applications with slow startup time can be stored in the already started process. This is useful 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 [24] was released in 2008 and the first implementation of container management on Linux without requiring any patches. It is mainly based around the already mentioned concepts of cgroups and namespaces.

LXC offers stateful snapstops and live migration by using the same technology as OpenVZ's checkpointing.

The Linux Container Daemon (LXD) [25] adds a layer on top of LXC. It is developed by Canonical and first released in (October 2015). In contrast to Docker, the focus is not

on containerising a single application but having a full operating system 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 [26]. 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 it exposes a secure REST API to allow management to be done through — for example — cloud orchestration software like Canonical's own Juju [27], OpenStack [28] or Kubernetes [29]. LXD does not offer any network or storage management leaving that to the orchestration on top of it. On the other hand it offers life migration of containers to another host which is essential of dynamic load distribution in a cloud infrastructure.

Docker

With the release of Docker 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 previously complex task of container management.

Docker is not a standalone container runtime but consists of many software components with different purpose. 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 [30].

Initially Docker used LXC as its container runtime and later developed its own runtime libcontainer which was later extracted and donated to the Open Container Initiative (OCI) and is now known as runC [31].

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 as shown in Figure 3.1. containerd uses runC to execute the container.

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. This type of container is called an application container. 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.

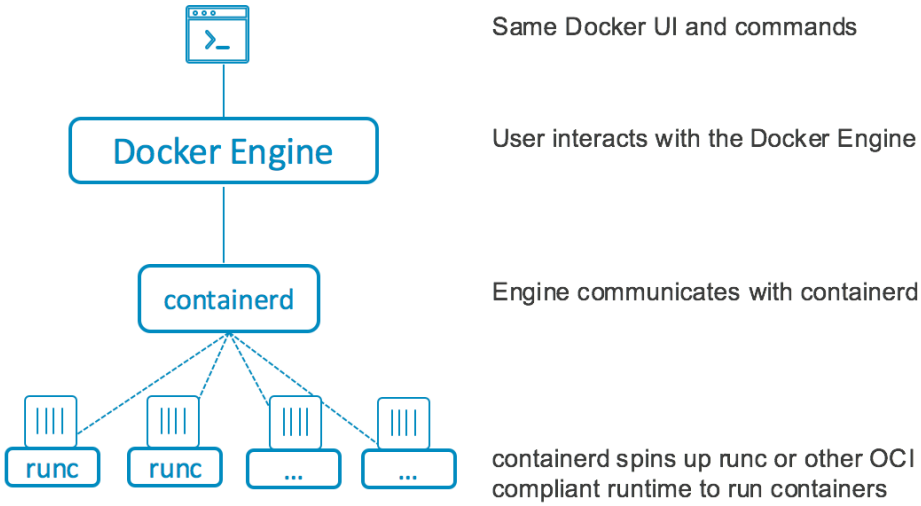

Figure 3.1: The Docker Engine architecture [32]

Docker containers and images use a layered approach where each layer contains a change set to the layer underneath [33]. As shown in Figure 3.2 an image consists of various read-only layers and the execution of current container is done in an additional read-write layer on top of the image layers. With this layered concept if an image is executed multiple concurrent or sequential times the images layers are only stored on the system once. The same is true for image sub-layers where for example multiple layers can inherit a base layer provided by Ubuntu.

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 a stored as a new image that can again be started up at that state. Furthermore, directories from outside the container can be mapped inside the container to store consistent changes outside the container.

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 [34] which allow networking interfaces to act as *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; *macvlan* where each container gets a unique mac address on the host's network and Docker even

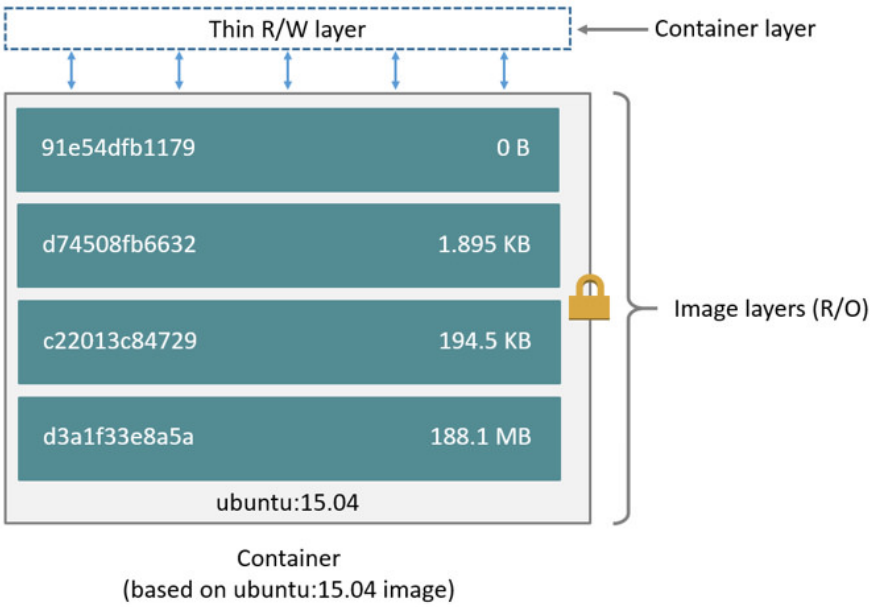

Figure 3.2: Docker Container Layers

allows third-party networking plugins to be used.

runC

runC [31] 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 [35] which is a lightweight and simple container runtime developed by Red Hat as a minimal container runtime for Kubernetes which was released in October 2017. "CRI-O is designed to be simpler than other solutions, following the Unix philosophy of doing one thing and doing it well, with reusable components." [36]. The goal for CRI-O is to have maximum compatibility with Kubernetes and provide a stable interface.

Another example is Garden [37], which is the containerization layer in the CloudFoundry orchestration software [38]. The Garden project moved from LXC to runC as their container back-end on Linux.

rkt

rkt [39] (pronounced "rocket") is an open-source container engine developed by CoreOS project. rkt took security into consideration from the early design. rkt offers encryption and signing of images [40]. In addition to rkt's native image format it can execute Docker images.

Same as Docker rkt focuses on application containers where the ideally only a single process is executed inside a container. rkt does not have a centralized daemon for configuration but integrates in systemd to start containers directly under the Linux init process. This eliminates the single point-of-failure of having a centralized daemon like Docker. Figure 3.3 shows the execution hierarchy of rkt compared to Docker.

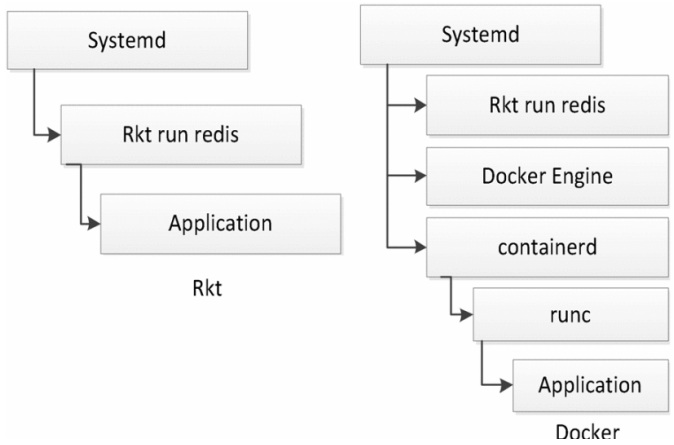

Figure 3.3: Execution hierarchy Docker vs. rkt [41]

rkt developed and uses the App Container standard appC [42] which was later included in the OCI standards. The project compares in [43] the appC standards to the OCI standard as follows: "*The App Container Image format (ACI) is maps more or less directly to the OCI Image Format Specification, with the exception of signing and dependencies. The App Container Executor (ACE) specification is related conceptually to the OCI Runtime Specification, with the notable distinctions that the latter does not support pods and generally operates at a lower level of specification. App Container Image Discovery does not yet have an equivalent specification in the OCI project.*"

appC uses the concept of pods as the basic unit of execution. "*A pod is a grouping of one or more app images (ACIs), with some additional metadata optionally applied to the pod as a whole.*"[44] This allows configuration of — for example — resource constraints and networking for multiple containers from a single point. The concept of pods in rkt is identical to the concept of pods in Kubernetes [45] which allows a good integration and rkt became the first non-Docker runtime to be supported in Kubernetes [46].

The rkt execution workflow as shown in Figure 3.4 describes three stages. After the

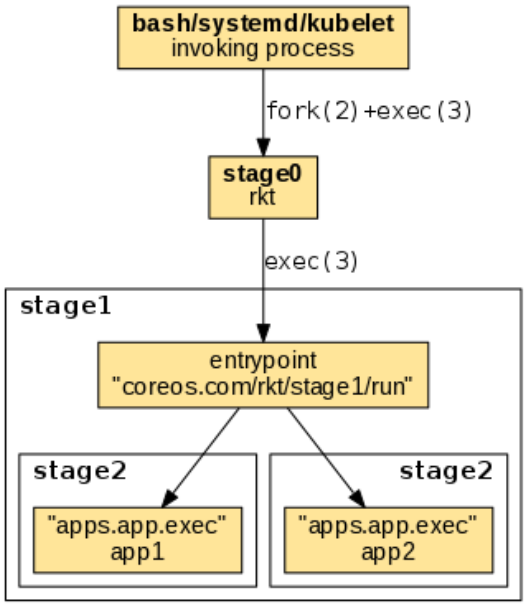

Figure 3.4: rkt execution workflow [47]

pod is started from various sources like the command line, systemd or Kubernetes the container runtime is set up in stage 1. Different execution environments are available in rkt. The default options are:

- **systemd-nspawnd** the default behavior that 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 container is launched.

rkt is currently changing its internal architecture to be compliant to the OCI standard [45] and is currently developing the integration of runC as a stage 2 execution environment. The current status rkt's OCI transition can be seen at [48].

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

rkt uses systemd-journald for logging which integrates directly in 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 [49] is an OCI-compliant hybrid VM container runtime that executes containers within virtual machines released in 2018.

Kata Container originated from merging of the two projects Hyper runV [50] and Intel Clear Containers [51] which focused on securing the runtime environment that containers execute in by executing the container inside a Kernel Virtual Machine (KVM) runtime [52].

Kata containers is developed under the OpenStack foundation and provide a virtualized isolation layer to make container execution more secure [53]. The project is supported by several companies to accelerate development and hardware support.

Kata Containers can be integrated into Docker and containerd and into Kubernetes with Kata containers OCI-compatable runtime or directly into the CRI layer with Frakti [54]. Figure 3.5 shows the different integration options of Kata containers into Kubernetes.[1] Figure 3.6 shows how Kata containers can be executed in Kubernetes under CRI-O alongside other OCI compliant containers.

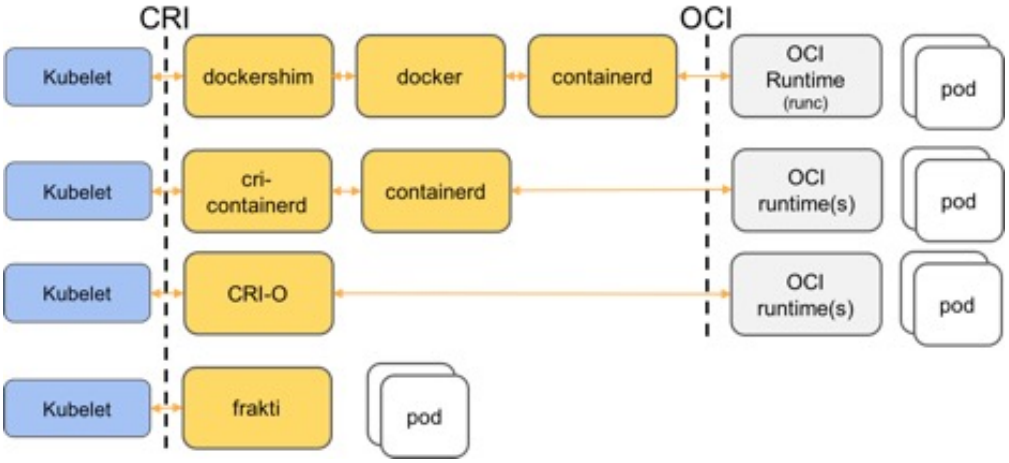

Figure 3.5: Kata Container intergration into Kubernets [55]

Kata Containers uses a minimal version of QEMU for virtualization. For faster start-up time of the VM container other virtualization engines have been added. Noticeably, Firecracker MicroVM [56] is available with Kata Containers as of version 1.5[2] [57]. 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.

[1] This is a good representation of the current development trend towards open standardization compliance on all layers.

[2] released Jan 22. 2019

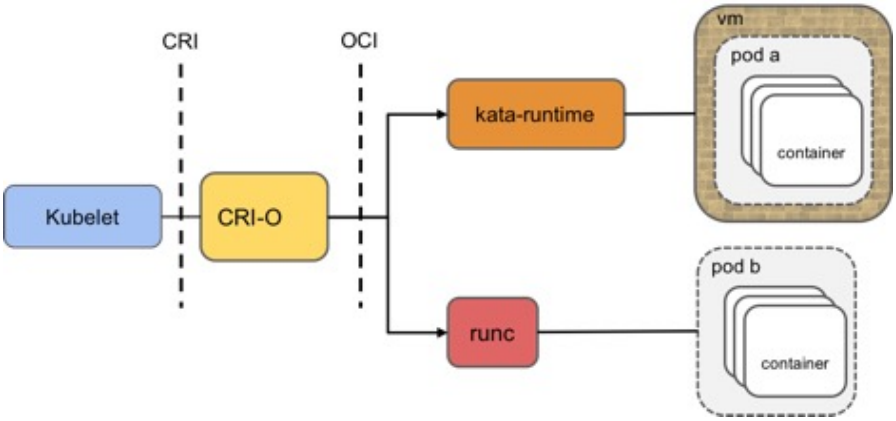

Figure 3.6: Kata Container intergration into Kubernets with CRI-O [55]

Other Frameworks

There are many other solutions available and the following few are mentioned only for reference and an interested reader might continue there.

- systemd-nspawnd [58] is part of systemd and allows the creation of a container directly from systemd
- gVisor [59] is an attempt to secure containers by providing a user space kernel abstraction that processes the system calls used by the container process.
- Nabla containers [60] tries to secure containers by restricting the kernel system calls with a strict seccomp profile that allows only 7 system calls.
- Singularity [61] is a container framework intended for scientific computing where operations are executed on many notes in a high performance cluster.

4 Comparison

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

Table 4.1: 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
LXD	2016	Kernel	System	yes	yes	yes	no
Docker	2014	Kernel	App	yes	yes	yes	yes
runC	2016[1]	Kernel	App	no	yes	yes	yes
rkt	2016	Kernel	App	no	no	yes	partial
Kata Containers	2018	VM	App	no	yes	yes	yes

Table 4.1 illustrates some characteristics of the presented container frameworks. It shows the release year of the framework and the underlying isolation strategy of the framework where 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 the daemon for LXC. All framework but rkt support live migration while the container solutions use CRIU and Kata containers the snapshotting feature of the underlying virtualization engine. All container frameworks all 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 focus on OCI compliance from the start.

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

Table 4.2: 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 on top of the container. 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 with shim layers available in Kubernetes and OpenStack. CloudFoundy supports Docker and integrated runC into there containerization engine. Apache Mesos [62] offers support for Docker alongside their own containerization implementation.

[2]Also has its own containerization engine.
[3]On top of Virtuozzo
[4]Inside OpenVZ containers

5 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 is build modular to facility integration into orchestrator software and allows the underlaying execution runtime to be replaces 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 and the standardized standalone runtime runC is directly integrated into other solutions that need to have full low-level control of the container.

This makes Docker for simple use the best choice when directly interacting with the containers and runC the best options when containers need to be integrated into other solutions with which the user interacts with.

Most container in the Cloud are executed inside a VM due to security concerns of the operators. This fact and its strong focus on integrability makes Kata Containers – which is currently still under heavy development – a very promising alternative to eliminating this layer of virtualization.

References

[1] "Breaking out of a chroot() padded cell." https://web.archive.org/web/20160127150916/http://www.bpfh.net/simes/computing/chroot-break.html, Jan-2016.

[2] 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. AINA 2008. 22nd International Conference on*, 2008, pp. 861–868.

[3] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A Measurement Study on Linux Container Security: Attacks and Countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 418–429.

[4] "Containers from user space [LWN.Net]." https://lwn.net/Articles/745820/.

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

[6] "Cloud Native Computing Foundation," *Cloud Native Computing Foundation*. https://www.cncf.io/.

[7] "Demystifying container runtimes." https://lwn.net/Articles/741897/.

[8] "OCI Image Format." https://github.com/opencontainers/image-spec, Jan-2019.

[9] "OCI Runtime Specification." https://github.com/opencontainers/runtime-spec, Jan-2019.

[10] "Container Network Interface: Networking for Linux containers." https://github.com/containernetworking/cni, Jan-2019.

[11] "Namespaces article index [LWN.Net]." https://lwn.net/Articles/766124/.

[12] "Understanding the new control groups API." https://lwn.net/Articles/679786/.

[13] "A seccomp overview." https://lwn.net/Articles/656307/, Sep-2015.

[14] 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 (ISPASS), 2015 IEEE International Symposium On*, 2015, pp. 171–172.

[15] E. Casalicchio and V. Perciballi, *Measuring Docker Performance: What a Mess !* 2017.

[16] X. Xie, P. Wang, and Q. Wang, "The Performance Comparison of Native and Containers for the Cloud," in *2018 International Conference on Smart Grid and Electrical*

Automation (ICSGEA), 2018, pp. 378–381.

[17] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing," in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, 2015, pp. 1–8.

[18] X. M. Aguilera, C. Otero, M. Ridley, and D. Elliott, "Managed Containers: A Framework for Resilient Containerized Mission Critical Systems," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 946–949.

[19] A. R. Manu, J. K. Patel, S. Akhtar, V. K. Agrawal, and K. N. B. S. Murthy, "Docker container security via heuristics-based multilateral security-conceptual and pragmatic study," in *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, 2016, pp. 1–14.

[20] "Measuring container security." https://lwn.net/Articles/773976/.

[21] "Open source container-based virtualization for Linux." *OpenVz*. https://openvz.org/.

[22] "Hyperconverged Infrastructure Software Provider." https://www.virtuozzo.com/.

[23] "CRIU." https://criu.org/Main_Page.

[24] "Linux Containers - LXC - Introduction." https://linuxcontainers.org/lxc/introduction/.

[25] "Linux Containers - LXD - Introduction." https://linuxcontainers.org/lxd/.

[26] "Systemd." https://freedesktop.org/wiki/Software/systemd/.

[27] "Jujucharms | Juju." https://jujucharms.com/.

[28] "OpenStack: Build the future of Open Infrastructure." https://www.openstack.org/.

[29] "Kubernetes: Production-Grade Container Orchestration." https://kubernetes.io/.

[30] P. Eder, "An infrastructure agnostic application deployment framework for the Internet of things," PhD thesis, Wien, 2017.

[31] "runC: CLI tool for spawning and running containers according to the OCI specification: Opencontainers/runc." https://github.com/opencontainers/runc, Jan-2019.

[32] "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/, Apr-2016.

[33] "About storage drivers," *Docker Documentation*. https://docs.docker.com/storage/storagedriver/, Jan-2019.

[34] "Docker Networking Documentation," *Docker Documentation*. https://docs.docker.com/network/, Jan-2019.

[35] “Cri-o.” https://cri-o.io/.

[36] “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.

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

[38] “Cloud Foundry: Open Source Cloud Application Platform,” *Cloud Foundry*. https://www.cloudfoundry.org/.

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

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

[41] X.-L. Xie, P. Wang, and Q. Wang, “The performance analysis of Docker and rkt based on Kubernetes,” in *2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, 2017, pp. 2137–2141.

[42] “App Container Specification and Tooling.” https://github.com/appc/spec/blob/master/SPEC.md, Jan-2019.

[43] “App Container Specification and Tooling. README.” https://github.com/appc/spec/blob/master/README.md, Jan-2019.

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

[45] “What Kubernetes users should know about the rkt container engine | CoreOS.” https://coreos.com/blog/rkt-and-kubernetes.html.

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

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

[48] “CoreOS rkt OCI native support | current project state.” https://github.com/rkt/rkt/projects/4, Jan-2019.

[49] “Kata Containers - The speed of containers, the security of VMs.” https://katacontainers.io/.

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

[51] “Intel Clear Containers.” https://github.com/clearcontainers/runtime, Jan-2019.

[52] “Intel, Hyper.Sh merge tech into Kata Containers,” *ICT Monitor Worldwide*, Dec. 2017.

[53] "OpenStack Boosts Container Security With Kata Containers 1.0," *ICT Monitor Worldwide*, May 2018.

[54] "The hypervisor-based container runtime for Kubernetes." https://github.com/kubernetes/frakti, Jan-2019.

[55] "Kata Containers - Why Kata Containers doesn't replace Kubernetes: A Kata Containers explainer." https://katacontainers.io/posts/why-kata-containers-doesnt-replace-kubernetes/.

[56] "Firecracker MicroVM." https://firecracker-microvm.github.io/.

[57] "Kata Containers documentation | Initial release of Kata Containers with Firecracker support." https://github.com/kata-containers/documentation/wiki/Initial-release-of-Kata-Containers-with-Firecracker-support, Jan-2019.

[58] "Systemd-nspawn." https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html.

[59] "Container Runtime Sandbox. Contribute to google/gvisor development by creating an account on GitHub." https://github.com/google/gvisor, Jan-2019.

[60] "Nabla containers: A new approach to container isolation," *Nabla Containers*. https://nabla-containers.github.io/.

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

[62] "Apache Mesos," *Apache Mesos*. http://mesos.apache.org/.