OPUS-HSO

Repositorium der Hochschule Offenburg
https://opus.hs-offenburg.de

Gaia-X4KI - TP2.2 Container Report

Janis Keuper

Zitiervorschlag im APA Stil:

Keuper, J. (2021). *Gaia-X4KI - TP2.2 Container Report*.

Abstract

Tech Report

Nutzungsbedingungen

Dieses Dokument wird unter diesen Bedinungen zur Verfügung gestellt:
Urheberrechtlich geschützt
Für weitere Informationen siehe:
https://rightsstatements.org/page/InC/1.0/

Kontakt

Hochschule Offenburg | Bibliothek
Badstraße 24
77652 Offenburg
Telefon: (0781) 205-240
E-Mail: bibliothek@hs-offenburg.de
www.hs-offenburg.de/bibliothek

TP2|AP2.2: Container Report

AP 2.2 Team Members

December 17, 2021

Contents

1 Introduction **1**
1.1 Versioning of this document 1
1.2 Introduction to Containerization 1

2 Container-Solutions Overview **3**
2.1 Docker Container/ContainerD 8
2.2 Kata Containers 8
2.3 Singularity (Apptainer) 8
2.4 Enroot 9

3 Use Case Requirements **9**
3.1 Container Payload Example: OpenVINO 9
3.2 Container Payload Example: CARLA 10

4 Container Registry **11**

5 Security **11**
5.1 Known Vulnerabilities 12
5.2 Basic Security Features 12

6 Summary **13**

Document History

- December 17, 2021: Release V1

1 Introduction

1.1 Versioning of this document

- V1 [current state]:
  - Gathering technical details and capabilities of available Container-Solutions
  - Gathering goals and requirements of all use cases
  - Recommend 2+x Candidates
- V2 [Q1 22]: Use Case Requirements
  - Present V1 to use cases
  - Sync with TP2.1 (Schedulers)
  - Gather feedback on technical requirements
  - Match requirements with candidate list
- V3 [Q2 22]:
  - Execute and document Infrastructure setup

1.2 Introduction to Containerization

This document describes the current state of the art regarding container technologies with the goal to select and implement the "best" solution for the *GAIA-X 4KI* use cases.

With the increasing use of software applications all over the world, it becomes an important part to make these applications executable on different hardware systems – data centers, private computers or in the cloud. Each of these systems may utilize a specific operating system and a maybe unique policy of software and libraries that is available for the users.

During the development of a new tool, developers have mostly the freedom to install every library they need in every version they need. Even though this is not "new" and dates back already to the first software tools, it becomes nowadays a more and more challenging part. In particular, if a software utilizes libraries and tools that are located in the broader field of *artificial intelligence*. In order to be able to run your application on different hardware setups without the necessity to completely rewrite your entire application, one can make use of so-called *"containers"*.

Containers make it possible to isolate an application or service in such a way that it is separated from the rest of the operating system. It therefore includes the software and applications that are needed for the application to work. In order to reduce the actual overhead, containers make use of the kernel of the host system. This has direct consequences for e.g. acceleration units like GPUs that need special drivers to function properly.

The idea of isolation applications can be dated back to the introduction of *chroot* [1], introduced in *UNIX V7* in 1979 (part of *BSD* since 1982). The biggest developments took place at the beginning of the 21st century. The first container realizations were *FreeBSD Jails* [2] (2000), *Solaris Containers* [3] (2004), and LXC [4] (2008). The first Docker [5] release – based on LXC – is from 2013. Since then, usage of containers increased drastically as the way to create and run them became easier and easier, and can be found in nearly every area – industry, science and private users.

With the increasing focus on using containers for scientific applications and services it became important, that in data centers and high performance computing (HPC) clusters one has to take

care of some special conditions and restrictions in those systems. Some of these restrictions are for example the fact that a user cannot gain root privileges or run any application that would make it possible for the user to gain more privileges than without the application. Other restrictions are for example that most HPC clusters run "headless" which means that they cannot run – without some effort – applications that need a running X-server. Also, due to the fact that such systems use a so-called distributed file systems – like *BeeGFS* [6] – require that there is no chance that a user could manipulate the file system with the use of a container. This increases the security aspect for a container solution in the HPC world even more. Besides these restrictions, HPC clusters heavily utilize batch-schedulers and orchestrators to schedule jobs on their system. Therefore, it is essential that a container solution in an HPC ecosystem has to work with these schedulers. All of these restrictions led to the first release of Singularity [7] in 2015. For more details on scheduler software and which type of container software they support, we recommend having a look at the scheduler overview document from TP2|AP2.1 which is available as result E.2.1.1 on the *Gaia-X 4KI* confluence website.

Besides HPC clusters and data centers containers are as well heavily used in cloud systems. For example *Google* schedules – according to their own numbers – millions of containers in their cloud every week. The use of these containers is in cloud systems quite diverse and ranges from running microservices to providing an environment for development. Besides the fact that cloud providers make use of the more or less same hardware that is used in HPC data centers, they use it in a different way. This makes it there possible to utilize container solutions like *Docker* easier, even though they can have restrictions like being headless as well. This is mostly due to the fact that the resources you get in a cloud are already isolated from the rest of the cloud – physically and via firewall restrictions – providing you more freedom to drastically change for example the OS of the nodes you get. Another point that is important to mention is that storage you get is either isolated as well or is associated to the resources you get. This does not mean that the security requirements of container solutions are lower, but the focus is different.

It is worth to mention here that containers are not the only way to isolate an application from the rest of the host system. Some other notable ways of doing this includes *"Sandboxes"* and *"Virtual Machines"*. The scope of a *sandbox* is more to just isolate an application from other parts of the system in such a way that most of the OS is not visible anymore, but it utilizes all binaries and libraries installed in the OS. The other part of the isolation spectrum are *virtual machines*. They use the hardware of the host but are fully separated from the OS. Even though they provide a better isolation like containers or sandboxes, they are quite heavy in resource usage and can be worse in performance.

2 Container-Solutions Overview

This chapter provides a table with the most common container applications.

TAB. 1: Container Applications Overview.

Name	**Website**	**License**	**Packages**	**Rootless**	**GPU Support**	**OCI Compliant**	**special Features**	**Comment**
Docker	website	Apache-2.0 License (GitHub)	deb, rpm, source, various (see here)	no	yes (NVIDIA Docker)	yes	–	Widely used in companies. One of that standard tools to containerize micro-services. Often used as synonym for containers in general.
Charliecloud	docs	Apache-2.0 License (docs)	source, rpm	yes	yes (NVIDIA Docker)	no, converts images via docker to docker-archives	–	actively maintained
Udocker	github	Apache License 2.0	pip, source	yes	yes (via NVIDIA Docker)	extracts existing Docker images	–	mimics docker capabilities without relying on docker libs

TAB. 1: Container Applications Overview. *(continued from previous page)*

Name	**Website**	**License**	**Packages**	**Rootless**	**GPU Support**	**OCI Compliant**	**special Features**	**Comment**
Singularity (Apptainer)	website	BSD 3-Clause	deb, rpm, source	yes	yes	yes (pulls images from registries and converts them)	user in- & outside the same, all networks directly available (ethernet & IB), build in GPU integration (if needed)	Interesting for data centers that do not allow `root`. Project aims to new name *"Apptainer"* (no 1.0 release so far).
Enroot	github	Apache-2.0 License (GitHub)	deb, rpm, source	yes ("enhanced unprivi-leged chroot(1)")	yes	yes (converting existing OCI images to usable file system images)	built-in GPU support with libnvidia-container, standalone (no daemon), also the same as Singularity ("minimized isolation"), IB support	developed by Nvidia.

TAB. 1: Container Applications Overview. *(continued from previous page)*

Name	**Website**	**License**	**Packages**	**Rootless**	**GPU Support**	**OCI Compliant**	**special Features**	**Comment**
LXC	website, github	GNU LGPLv2.1+, 2-clause BSD, GNU GPLv2 (license)	deb	yes	yes	yes (converting from OCI image to LXC usable image)	"Full" integration in the Linux ecosystem. Image can run in username space (rootless) and on system level with root privileges	utilizes features like namespaces, cgroups, kernel capabilities etc. to create a lightweight image in between a chroot and a VM
Shifter	website, github	custom	source (build as "standalone" for CRAY and plugins for slurm)	yes	yes	able to convert OCI images	native scheduler integration via plugins	works "best" when integrated in a batch scheduler, e.g. slurm, usage without a scheduler is not recommended
Podman	website, docs	Apache-2.0 License (GitHub)	rpm, source	yes (daemonless)	yes (rootfull/rootless) source	yes	by design for rootless user, supports Pods	Red Hat counterpart to docker, referred as a docker alias

TAB. 1: Container Applications Overview. *(continued from previous page)*

Name	**Website**	**License**	**Packages**	**Rootless**	**GPU Support**	**OCI Compliant**	**special Features**	**Comment**
Containerd	website, github	Apache-2.0 License (GitHub)	binaries, tar, source	yes	yes	yes	–	low level container run time
Firecracker	github,	Apache version 2.0	source	N/A	planned	yes	–	Developed at Amazon – used in Amazon Fargate and Amazon Lambda. VM level isolation
Kata Containers	website, github	Apache-2.0 License	source	N/A	yes	yes	–	Container runtime for Kubernetes that provides VM-level isolation (can use Firecracker instead of QEMU)
FreeBSD Jails	website	BSD-2-Clause License	–	–	–	–	–	needs a BSD based OS on the underlying host

TAB. 1: Container Applications Overview. *(continued from previous page)*

Name	**Website**	**License**	**Packages**	**Rootless**	**GPU Support**	**OCI Compliant**	**special Features**	**Comment**
Userdocker	github	MIT License (GitHub)	pip, source	yes	yes (NVIDIA Docker)	yes	–	Docker Wrapper to restrict command-line access to users, Beta, *unmaintained since 07/2019*
AppC	github	–	–	–	–	–	–	*discontinued*
rkt	github	–	–	–	–	–	–	*discontinued*

2.1 Docker Container/ContainerD

The Docker Container Engine was release in 2013 and simplified the containerization of applications. It uses existing concept of the Linux kernel like cgroups and namespaces and bundles them in a CLI tool. Docker brings tools like *build* to create a container image from a *Dockerfile*, a text file with the description of the container. The Docker tool made it possible to share the images with other people and reuse them.

Due to its early position in the market of container tools, Docker influenced how containers work and what makes a container. This was reflected when *Docker* donated their container runtime *runC* and the *Base Container Format* to the Open Container Initiative (OCI) [8], which became the basis of the OCI Specification.

Dockers container runtime is *ContainerD*, which was also spun out by Docker and graduated in the Cloud Native Computing Foundation in 2019 [9]. *ContainerD* is used as an abstraction layer between operating system and and userspace, shown in figure 1, and makes it possible to port and reuse the same interface on different operating systems. The container engine is *runC*, but it can be exchanged or extended by other shims. Examples can be *Kata Containers* (see section 2.2), *runhcs*, a fork of *runC* for Windows [10] and *Firecracker*, a micro-virtual-machine created by Amazon [11].

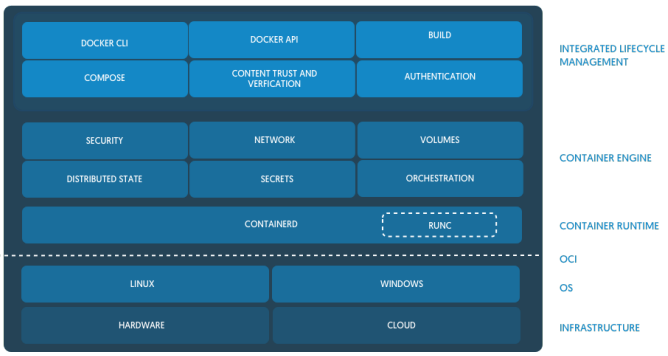

FIG. 1: *Diagram of the Docker stack using containerd and runC [12].*

2.2 Kata Containers

Kata Containers is an open source project and community working to build a standard implementation of lightweight Virtual Machines (VMs) that feel and perform like containers, but provide the workload isolation and security advantages of VMs.

Kata Containers can be used as the runtime instead of Docker in Kubernetes. Kata Containers can also utilize Firecracker microVMs for isolation (as an alternative to QEMU).

Kata Containers are VMs that run OCI containers over a thin custom Linux Kernel.

2.3 Singularity (Apptainer)

Like other container solutions *"Singularity"* [7] follows the idea of creating and running packaged software – either pieces or multiple tools – in such a way that the resulting "container" is portable, secure and creates independent of the used hardware the same results.[13]

Singularity had its focus originally on scientific environments of high performance computing (HPC) clusters. With this in mind, *Singularity* follows a sightly different approach compared to

other container solutions. Therefore, *Singularity* has to take care of different user *and* environment needs. Maybe the biggest difference to mention comes from the HPC systems – a user on an HPC cluster cannot use a container that requires root privileges to start. The most demanding needs on the users side are direct access to different networks – Ethernet or Infiniband – direct usage of accelerator cards – GPUs or FPGAs – and fully utilize the provided distributed file system – e.g. *BeeGFS* [6]. Following these demands, a singularity container – or to be more precise a *Singularity* "image" – does not require any root privileges to start and has therefore no extra user management. The image is started by a normal user and inside the container he has the same privileges as outside and *cannot* gain any additional privileges.

In order to realize all the demands *Singularity* follows a rather strict security policy.[14] This policy leads to some limitations – e.g. it is not easily possible to run a daemon process inside a singularity container – but provides on the other side to interesting features – e.g. direct GPU, MPI and GPI support or high level of isolation and security layers. Besides the already mentioned security features *Singularity* adds as well the ability to sign containers with a private/public key as well as the feature to fully encrypt the containers.[7]

It is worth to mention that *Singularity* provides the possibility to convert already existing Docker containers into the *Singularity* format, so that they can also be started with the help of *Singularity*.

With the end of 2021 *Singularity* will be renamed to *"Apptainer"* which so far seems to be not more than a re-branding without any changes regarding license or features.[15]

2.4 Enroot

"Enroot" [16] follows like *Singularity* the approach to provide secure, rootless and highly scalable containers with scientific and/or HPC systems in mind. As being developed by NVIDIA, *Enroot* directly utilizes NVIDIA's GPU libraries that are also used in the older NVIDIA Docker images.

In contrast to Singularity, the actual amount of isolation is less restrictive and can be seen more like an extended `chroot` setup.[16] It shall be mentioned that this less restrictive security concept provides a greater flexibility regarding usage, batch scheduler integration, speedup and scalability. With on the other side the price of having fewer isolation capabilities and an even further focus on scientific applications than *Singularity*.

Another "limitation" of *Enroot* is that one cannot use it with GPUs that are older than the "Fermi" architecture which was released in 2010, as well as NVIDIA drivers older that 361.93 released 2016.[17] Even though it is quite unlikely that such old hardware is still used widely it is a limitation to be aware of them.

3 Use Case Requirements

In the first month of the project, the first requirements for the infrastructure of the different use cases were collected. Currently, only basic requirements like reading and writing to S3, parameterization of a container and GPU support are present.

In the following sections, two different container payloads are presented, these examples show what kind of tasks will be performed on the infrastructure.

3.1 Container Payload Example: OpenVINO

OpenVINO™ is an open-source toolkit for optimizing and deploying AI inference.[18] Its component Model Optimizer facilitates the transition between the training and deployment environment, performs static model analysis, and adjusts deep learning models for optimal execution

on end-point target devices. The second part is the Inference Engine, which is a set of C++ libraries with C and Python bindings providing a common API to deliver inference solutions on the platform of your choice. The Inference Engine could be used within Docker containers.[19] There is also the OpenVINO Model Server [20], which is a scalable, high-performance solution for serving machine learning models optimized for Intel architectures. The server provides an inference service via gRPC or REST API - making it easy to deploy new algorithms and AI experiments using the same architecture as TensorFlow* Serving for any models trained in a framework that is supported by OpenVINO.

3.2 Container Payload Example: CARLA

CARLA [21] is an open-source simulator for autonomous driving and broadly used within both academic and industrial research.

The application is based on the Unreal Engine and provides a client-server architecture. The simulation (environment, vehicles, pedestrians), the physics and the rendering is executed by the server. A client (or multiple) could attach to the server. Vehicles and sensors (e.g. cameras) could be spawned via that interface, data could be received (e.g. sensor data like simulated LIDAR or rendered camera images) and both the ego vehicle and the simulation in general could be controlled. This consists of e.g. global settings like weather or lighting conditions and other traffic participants.

CARLA provides a Python API for the client implementation [22] as well as a CARLA ROS bridge [23], which facilitates the usage of CARLA with autonomous driving stacks based on-top the Robot Operating System versions 1 and 2 (ROS1 and ROS2).

For the deployment of CARLA server and client(s), there are some constraints or dependencies to be considered. The first on is network latency, the second is bandwidth. The CARLA client / server setup could be run in several configurations regarding asynchronous or synchronous mode and time stepping [24]. Especially in the asynchronous mode, the round-trip latency from sensor data produced by the server and sent to the client, and the control information sent back has to stay in certain limits to allow proper closed loop control. The bandwidth highly depends on the amount and type of sensors simulated by the server. Cameras are most critical in regards of network bandwidth as they are transmitted uncompressed due to latency and compute reasons and a single full HD (1920 x 1080) with 30 FPS produces 1.4 Gbit/s when the simulation is executed in real-time.

CARLA provides pre-built images on Docker Hub [25]. Prerequisites and usage are described in the online documentation [26]. The default image has a size of 15-20 GB, but adding additional maps could increase this significantly.

The CARLA server asks for 16+GB RAM and 8+ vCPUs. The hardware needs for the client and the autonomous driving stack could not be quantified here as they are highly dependent on the chosen solution, but typical numbers are the same or even higher requirements for compute.

CARLA requires 3D hardware acceleration, either via Vulcan or OpenGL Interface. It strongly recommends (and only documents and tests) the usage of mid to high level NVIDIA GPUs ($\geq$ RTX 2070), especially in a containerized environment. This setup requires a reasonable new installation of Docker, the official proprietary NVIDIA GPU Drivers and the NVIDIA Container Toolkit [27]. Usage of alternative GPUs like Intel Alchemist (aka DG2) has to be evaluated.

4 Container Registry

One innovation of *Docker* was not only to put applications into containers, but also to upload them into a public registry. This enabled the sharing and reuse since it is possible to take an existing container, build a new layer of modifications on top of it and push it again.

A registry is similar to a package repository, it stores the images in a tree-like structure and lets the creator upload multiple versions of a container that can contain security fixes and updates. The user specifies the name and version of the image that is downloaded. Most images are either base images that contain a single feature or an application that comes bundled with all dependencies in the container.

There are many different hosted container registries. First was the *Docker* registry, which is the default choice of the *Docker* tool. Every cloud service provider offers a registry as a service, like Amazon Web Services, Google Cloud and Azure. RedHat joined the market with a mix of hosted and self-hosted registry with their *Quai.io* offer [28].

The alternative to a hosted registry is to host it on premise. This has multiple advantages:

- It can be used as a pull through cache to reduce download volume and time. It also helps to stay below the rate limits that Docker implemented for their free registry service in 2020 [29].
- A local registry is better for confidentiality. While it is never appropriate to store sensible data or secrets in a container, there can be business logic or proprietary tools inside the container, which should not be uploaded to a public registry.
- Run an air-gapped registry for deployments that can't be connected to the internet due to security concerns.

Docker provides a minimal Docker container for running a self hosted registry [30]. Harbor is more complete alternative with lots of features like a web interface and security and vulnerability scanning for container images[31].

5 Security

The popularity of containers has led to a significant change in the provision of software and applications. An application is no longer provided as a monolithic VM, but several containers represent the end application from many small components. A frequently described example from the literature [32] is a Wordpress application, which consists of Wordpress and a database. Enriched with the business logic, it forms an executable end application.

Compared to the historical approach, containers offer advantages such as: faster product updates, easier rolebacks, better utilization of the infrastructure, and easier scaling up and down of the application. However, you don't just buy benefits with new technology. Many new hurdles needed to be addressed when using container technologies. In the following, an incomplete list of Vulnerabilities is introduced. Also, a minor introduction for some security features is provided. A more detailed list and approaches to avoid/mitigate known vulnerabilities is provided by the Institute of Standards and Technology (NIST) in its Application Container Security Guide [33]. The German Federal Office for Information Security provides the German counterpart, SYS.1.6 Containerization [34].

5.1 Known Vulnerabilities

In respect to Container management there are many threats which needs to be faced, to have a secured and trusted System. Some known vulnerabilities and configuration mistakes are described below. In addition, many discovered Vulnerabilities related to containers can be found in the CVE-Mitre Database [35].

- **Root Daemon**: In the early days of Docker, as well as today, container management in Docker takes place through the Docker daemon. However, this can run in the context of the root user and can therefore only be accessed by root and the specific group *docker*. Unprivileged users who are added to the docker group are able to change the host system to theirs through the docker daemon and accordingly make themselves a privileged user and thus corrupt the system. In no way should a container software with a root daemon be given to a simple user.
- **Container excavation**: Due to the lower isolation between the host system and the container than in the case of virtual machines, there is a further attack surface on the system. An attacker can thus succeed in breaking out of a container. In combination with a mounted root user, this offers attackers a gateway into the system. An unprivileged user can also examine and explore the system to a certain extent and, if necessary, further exploit vulnerabilities.
- **Insecure & Malicious Container**: Another vulnerability in the container environment also concerns the containers themselves. These can contain outdated software, for example, which introduces an initial security vulnerability, such as an outdated OpenSSL library. Containers from untrusted sources are another security scenario. For example, in one incident [36] on Docker Hub, a container was deployed under an inconspicuous name, which was prepared with third-party software for cryptocurrency mining.

5.2 Basic Security Features

Many approaches are known for container managers to avoid security vulnerabilities. For this, a Linux system (kernel) offers many possibilities to influence the management of containers from the ground up. However, it should be noted that many functions do not actively provide protection until they have been properly configured. Two of these mechanisms and their use are described in more detail below.

- **Namespaces** [37] provide the functionality to define different resources (e.g. process IDs, user IDs) in namespaces. Thus, for example a software-sided separation between applications, within and outside a namespace, is realized. For a resource, which is inside a namespace, it has the appearance as if it were the only process and is thereby isolated from processes on the host system.
- **CGroups** [38] is a Linux kernel function for the exact allocation of resources to processes or process groups. Thus, for example, it can be achieved that a process is only available a specified share of the CPU or RAM. In the context of container management in such a way for the respective container a limit of used hardware (CPU, RAM, HDD, etc.) can be realized. Among other things, this enables better resource utilization, since several containers are not in competition with the available hardware. Furthermore, it offers protection against overload of the system, so that it always remains accessible for maintenance and management purposes.

- **Container Scanner** [39] are external tools that provide protection against vulnerabilities from containers. The tools offered look at different aspects of an image or container. *"Harbor"* [40] is primarily a container registry to persistently store and manage container images. In addition, it also provides functions to statically scan stored images. The *"Falco Project"* [41] monitors containers at runtime and, depending on the configuration, informs about rule violations that have occurred.

6 Summary

Based on the broad evaluation of existing container tools in V1, we suggest the following most promising candidates (order as listed before) for V2.

- Cloud-focused Container Solutions
  - Docker | Podman
  - Kata
- HPC-focused Container Solutions
  - Singularity
  - Enroot

This shortlist does not only consider the container technologies that are mostly used, but also takes care of the different demonstrators – cloud *and* HPC based – that will be used in *Gaia-X 4KI*. This would give us the possibility to flexibly utilize each of the demonstrators and increase the number of compute providers in the future as well. In addition, we would get the ability to smoothly convert existing containers from one solution to another one, which will give us in the long run to also provide suggestions for future users which container solution would fit better to their needs or the available resources.

References

[1] *chroot – Linux manual page*. URL: http://man7.org/linux/man-pages/man2/chroot.2.html. (accessed: 09/12/2021).

[2] *jail – BSD manual page*. URL: https://www.freebsd.org/cgi/man.cgi?query=jail&format=html. (accessed: 09/12/2021).

[3] *Oracle Solaris Containers*. URL: https://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html. (accessed: 09/12/2021).

[4] *Linux Containers*. URL: https://linuxcontainers.org. (accessed: 09/12/2021).

[5] *Docker website*. URL: www.docker.com. (accessed: 09/12/2021).

[6] *Beegfs website*. URL: https://www.beegfs.io/. (accessed: 09/12/2021).

[7] *Singularity/Apptainer website*. URL: https://apptainer.org/. (accessed: 06/12/2021).

[8] *Introducing runC: a lightweight universal container runtime*. URL: https://www.docker.com/blog/runc/. (accessed: 08/12/2021).

[9] *ContainerD Website*. URL: https://containerd.io/. (accessed: 10/12/2021).

[10] *Container platform tools on Windows*. URL: https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/containerd. (accessed: 10/12/2021).

[11] *Firecracker Website*. URL: https://firecracker-microvm.github.io/. (accessed: 10/12/2021).

[12] *What is containerd?* URL: https://www.docker.com/blog/what-is-containerd-runtime/. (accessed: 10/12/2021).

[13] *Singularity/Apptainer Documentation*. URL: https://apptainer.org/user-docs/master/introduction.html. (accessed: 06/12/2021).

[14] *Singularity/Apptainer Security Documentation*. URL: https://apptainer.org/user-docs/master/security.html#security. (accessed: 06/12/2021).

[15] *Singularity/Apptainer re-branding annoncement*. URL: https://apptainer.org/news/community-announcement-20211130. (accessed: 06/12/2021).

[16] *enroot website*. URL: https://github.com/NVIDIA/enroot. (accessed: 06/12/2021).

[17] *enroot requirements*. URL: https://github.com/NVIDIA/enroot/blob/master/doc/requirements.md. (accessed: 06/12/2021).

[18] *OpenVINO documentation*. URL: https://docs.openvino.ai/. (accessed: 09/12/2021).

[19] *OpenVINO Docker usage documentation*. URL: https://docs.openvino.ai/latest/openvino_docs_install_guides_installing_openvino_docker_linux.html. (accessed: 09/12/2021).

[20] *OpenVINO Model Server documentation*. URL: https://docs.openvino.ai/latest/openvino_docs_ovms.html. (accessed: 09/12/2021).

[21] *CARLA website*. URL: https://carla.org/. (accessed: 06/12/2021).

[22] *Carla Python API reference*. URL: https://carla.readthedocs.io/en/0.9.13/python_api/. (accessed: 07/12/2021).

[23] *Carla ROS bridge*. URL: https://carla.readthedocs.io/projects/ros-bridge/en/latest/. (accessed: 07/12/2021).

[24] *Carla time synchronicity*. URL: https://carla.readthedocs.io/en/0.9.13/adv_synchrony_timestep/. (accessed: 07/12/2021).

[25] *Carla Docker hub website*. URL: https://hub.docker.com/r/carlasim/carla. (accessed: 06/12/2021).

[26] *Carla Docker usage documentation*. URL: https://carla.readthedocs.io/en/0.9.13/build_docker/. (accessed: 06/12/2021).

[27] *NVIDIA Container Toolkit documentation*. URL: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/overview.html. (accessed: 07/12/2021).

[28] *An introduction to the Quay container registry*. URL: https://www.redhat.com/sysadmin/introduction-quay. (accessed: 13/12/2021).

[29] *Scaling Docker to Serve Millions More Developers: Network Egress*. URL: https://www.docker.com/blog/scaling-docker-to-serve-millions-more-developers-network-egress/. (accessed: 13/12/2021).

[30] *Docker Registry Container*. URL: https://hub.docker.com/_/registry. (accessed: 13/12/2021).

[31] *Harbor Website*. URL: https://goharbor.io/. (accessed: 13/12/2021).

[32] Daniel Stender. *Cloud-Infrastrukturen*. Rheinwerk Computing, 2020. ISBN: 978-3-8362-6948-3.

[33] *Application Container Security Guide*. URL: https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-190.pdf. (accessed: 06/12/2021).

[34] *SYS.1.6 Containerisierung*. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Drafts/Final_Draft/SYS_1_6_Containerisierung_Edition_2022.pdf;jsessionid=B161E86D4D4CE1D8418380066D33009A.internet462?__blob=publicationFile&v=2. (accessed: 06/12/2021).

[35] *CVE*. URL: https://cve.mitre.org/index.html. (accessed: 09/12/2021).

[36] *Malicious Docker Hub Container Images Used for Cryptocurrency Mining*. URL: https://www.trendmicro.com/vinfo/fr/security/news/virtualization-and-cloud/malicious-docker-hub-container-images-cryptocurrency-mining. (accessed: 06/12/2021).

[37] *namespace(7) — Linux manual page*. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html. (accessed: 07/12/2021).

[38] *cgroups(7) — Linux manual page*. URL: https://man7.org/linux/man-pages/man7/cgroups.7.html. (accessed: 07/12/2021).

[39] *10 Container-Sicherheitsscanner zum Auffinden von Sicherheitslücken*. URL: https://geekflare.com/de/container-security-scanners/#anchor-docker-scan. (accessed: 07/12/2021).

[40] *Vulnerability Scanning*. URL: https://goharbor.io/docs/2.0.0/administration/vulnerability-scanning/. (accessed: 07/12/2021).

[41] *The Falco Project*. URL: https://falco.org/docs/. (accessed: 07/12/2021).