Hochschule Worms, Fachbereich Informatik

Studiengang: Angewandte Informatik

Bachelorarbeit

Leistungsbewertung von VM-basierten Containerlösungen

Vahel Hassan

Abgabe der Arbeit: 19. August 2020

Betreut durch:

Prof. Dr. Zdravko Bozakov Hochschule Worms

Zweitgutachter/in: Prof. Dr. Herbert Thielen, Hochschule Worms

Hochschule Worms
University of Applied Sciences

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich meine **Bachelorarbeit** mit dem Titel

---

Leistungsbewertung von VM-basierten Containerlösungen

---

selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie nicht an anderer Stelle als Prüfungsarbeit vorgelegt habe.

Ort

Datum

Unterschrift

Hochschule Worms
University of Applied Sciences

Kurzfassung

Es gibt in der heutigen Zeit viele Lösungen zu Container-basierte Virtualisierung, viele Unternehmen versuchen verschiedene Containerlösungen auszuprobieren, um schneller, günstiger und flexibler seine Anwendungen darauf auszuführen. Im Gegensatz zu klassischen virtuellen Maschinen, die heute noch sehr oft in Unternehmen verwendet werden, sind Containerlösungen in den letzten Jahren sehr bekannt geworden. Einer der bekanntesten Faktoren, warum Container so populär geworden sind, ist Docker Container. Seitdem Docker Container veröffentlicht wurde, interessieren sich immer mehr Unternehmen für Containerlösungen, um leistungsfähiger zu arbeiten. Seitdem sind zwei neue Interessante Projekte entstanden, zum einen der Kata-Container und Firecracker Containerd. Die vorliegende Bachelorarbeit möchte einen Überblick über die 3 unterschiedlichen erwähnten Containerlösungen geben und die Leistung der zwei neuen Containerlösungen mit dem Docker Container vergleichen und bewerten. Es werden mehrere Testdurchläufe durchgeführt, um die einzelnen Container-Technologien zu testen und bewerten.

Hochschule Worms
University of Applied Sciences

Abstract

There are many solutions to container-based virtualization today, many companies try different container solutions to run their applications faster, cheaper and more flexible. In contrast to classic virtual machines, which are still very often used in companies today, container solutions have become very popular in recent years. One of the best known factors why containers have become so popular is Docker Container. Since Docker Container was released, more and more companies are interested in container solutions to work more efficiently. Since then, two new interesting projects have been created, the Kata Container and Firecracker Containerd. This bachelor thesis wants to give an overview of the 3 different mentioned container solutions and to compare and evaluate the performance of the two new container solutions with the Docker Container. Several test runs are carried out to test and evaluate the different container technologies.

Danksagung

Ich möchte mich bei Prof. Dr. Zdravko Bozakov herzlich bedanken, er hat trotz der Corona Phase sehr viel Zeit investiert um mir sowohl bei der Durchführung des Praktischen Teiles als auch bei der Verfassung der Arbeit mit Ratschlägen geholfen.

Ich möchte mich an letzte Stelle für die Korrektur Lesung an meinem Bruder Gerwen und meine Freundin herzlich dafür bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Programmcodeverzeichnis

Abkürzungsverzeichnis

VM	Virtual Machin
AWS	Amazone Webservices
CPU	Central Processing Unit
RAM	Random-Access Memory
App	Application
API	Application Programming Interface
CLI	Command-line reference
RESTful	Representational State Transfer full
mircoVM	Micro Virtual Machine
vCPU	Virtual Central Processing Unit
VMM	Virtual Maschin Manager
UnionFS	Union Filesystem
OSCP	Open Source Community Project
TAP	Terminal Access Point
Rootfs	Root Dateisystem
Initrd	Initial ramdisk
Cpio	Copy files to and from archives
Tmpfs	Temporary File System
PID	Process identifier
IPC	Interprocess communication
GPG	GNU Privacy Guard

1 Einleitung

Die Digitalisierung hat sich in den letzten Jahren stark verändert. Früher hat man Telefone für nur einen Zweck benutzt, um mit anderen zu kommunizieren, diese wurden durch sogenannte *Smarthone* und andere *smart Devices* ersetzt, weil sie nicht nur für ein Zweck dienen sondern für viele andere Funktionen auch, wie Fotos schießen, viele verschiedene *Apps* die wiederum unterschiedliche Funktionen anbieten. Diese Möglichkeiten gibt es seitdem das Internet so populär geworden ist. Die Digitalisierung hat sich in den letzten Jahren ständig weiterentwickelt und einer der großen Technologien über fast jedes Unternehmen in der jetzigen Zeit redet, sind die Virtualisierungstechniken. Immer mehr Unternehmen möchten sich in diesem Bereich weiter entwickeln, weil die immer mehr an Interesse gewinnt. Zu einer den weitverbreiteten Containern zählt der *Docker Container* (Witt et al. 2017). Der *Docker Container* wurde 2013 veröffentlicht und hat in diesen kurzen Jahren viele Firmen aufmerksam gemacht, sie haben ähnliche Vorteile wie herkömmliche VMs aber sind effizienter, und tragbarer (vgl. Docker Inc. 2020). 4 Jahre später wurde eine weitere Containerlösung veröffentlicht, dabei handelt es sich um den *Kata-Container*, der sich zurzeit noch in der Gründungsphase befindet, die Entwickler haben sich hierbei stark auf den *Workload-Isolations* und Sicherheitsvorteile von Containern fokussiert (vgl. Kata Container 2020a). Die neuste und interessanteste Container Veröffentlichung wurde von einer der größten Unternehmen der Welt *AWS* herausgebracht, hierbei handelt es sich um den *Firecracker-Containerd* der im Jahr 2018 von *AWS* selbst entwickelt wurde. Der *Firecracker* soll hohe *Workload-Isolation* bieten und gleichzeitig die Geschwindigkeit und Ressourceneffizient von Containern ermöglichen (vgl. Firecracker Microvm 2020). Diese Arbeit beschäftigt sich mit der Leistungsbewertung von Containerlösungen und deren Technologische Anwendungsarten.

1.1 Problemdarstellung

Viele Unternehmen die auf Schnelligkeit, Sicherheit und kostengünstige Virtualisierung Wert liegen ist es besonders wichtig wie die Leistungen der unterschiedlichen Container zu bewerten ist. Diese Arbeit befasst sich mit der Installation und Durchführung von verschiedenen Tests, um die Leistungen von Docker-Container mit den 2 neuartigen *Container Firecracker Containerd* und *Kata Container* zu vergleichen und bewerten. Zu der Durchführung werden noch wichtige Technologische Kenntnisse, die man benötigt erklärt.

1.2 Zielsetzung

Die Zielsetzung dieser Arbeit ist es dem Laien zu veranschaulichen wie sich die 3 verschiedenen Containerlösungen *Firecracker Containerd*, *Docker Container* und *Kata Container* sowohl von Technischen Aspekten als auch von der Perfomance unterscheiden. Es soll mithilfe von Grafiken veranschaulicht werden wie die *CPU Geschwindigkeit*, *die RAM Geschwindigkeit*, die *Network Perfomance* und *Startzeiten* sich voneinander unterschieden. Aber nicht nur die Unterscheidung dieser Aspekte soll verdeutlicht werden, sondern auch worin sich diese Containerlösungen von Technologische Aspekten unterscheiden. Welche Vorteile die verschiedenen Containerlösungen versprechen und mit welche Technologischen Wissen diese entwickelt wurden.

1.3 Aufgabenstellung

Die Aufgabe mit dem sich diese Arbeit beschäftigt ist die Leistungsbewertung der Container-Technologien: *Firecracker Containerd*, *Kata Container* und *Docker Container*. Zunächst werden die einzelnen Container-Technologien auf einem Linux Betriebssystem installiert, sodass das sie in einer ausführbaren Umgebung mit einer Netzwerkverbindung laufen. Nachdem diese installiert sind werden die 3 Container auf den gleichen Systemstand gebracht. Dabei wird für alle 3 Containerarten dieselbe *Docker Image* verwendet. Der *Docker Image* wird mithilfe eines *Dockerfiles* erzeugt und dann für den jeweiligen Container ausgeführt, sodass alle 3 Technologien die gleichen Systemvoraussetzungen erfüllen.

Nachdem die die Container alle auf den gleichen stand sind, werden verschiedene *shell-scripts* verwendet, um die verschiedenen Testdurchläufe auszuführen und die Ergebnisse in einer Datei abzulegen. Diese Ergebnisse werden mithilfe von *Python* eingelesen, und zum Schluss dann *geplottet*, sodass die Ergebnisse in Grafiken dargestellt werden können, um dies verschiedenen Ergebnisse zu Bewerten. Bestandteil der Arbeit ist es nicht die Gesamte Technologischen Hintergründe der Containerlösungen zu erklären, sondern die Leistungen der Container zu Bewertungen und wichtige Technische Prozesse, um Hintergrund zu verstehen. Es werden nur die dafür benötigten Technischen Hintergründe aufgeklärt, die zum Verständnis der Leistungsbewertung der Containerlösungen auch benötigt wird.

1.4 Aufbau der Arbeit

In Kapitel 2 werden die benötigten Theoretischen Grundladen dieser Arbeit behandelt. Zu Beginn wird erklärt was überhaupt unter Virtualisierung und Container zu verstehen ist. Danach werden die Unterschiede und Gemeinsamkeiten von einem Container und Virtuelle Maschine. Zum Schluss wird die Architekturen der drei Containerlösungen erklärt und wie sie funktionieren.

Im Kapitel 3 wird erklärt, wie man die unterschiedlichen Containerlösungen installiert und welche *Tools* man dafür benötigt.

Im Kapitel 4

2 Theoretische Grundlagen

Im Theoretischen Teil werden Grundwissen über allgemein Virtualisierung vermittelt. Es werden die wichtigsten Technischen Hintergründe dieser Containerlösungen erklärt, damit der Leser bei der Installation der Container die benötigten Theoretischen Grundlagen versteht.

2.1.1 Virtualisierung

Schone Ende 1960-Jahre wurde in einem Großrechner Virtualisierungstechniken verwendet, damit mehrere Benutzer auf einem System gleichzeitig arbeiten konnten. Dadurch hat man eine Menge Hardwarekosten erspart und das Interesse vieler Unternehmen enorm erhöht (Buhl, H. und Winter 2008).

Bei der Virtualisierung werden physischer *Hardwareressourcen*, *Softwareressourcen*, *Speicheressourcen* und *Netzwerkkomponenten* abstrahiert, um diese auf der virtuellen Ebene zur Verfügung zu stellen. Mithilfe dieser Bereitstellung soll der Verbrauch von IT-Ressourcen bei der Virtualisierung stark reduziert werden (vgl. IONOS 2020).

Es gibt eine sogenannte Software namens *Hypervisors*, diese Software ist für die Trennung der physischen Ressourcen der virtuellen Maschinen zuständig. Im Prinzip sind virtuelle Maschinen, Systeme, die Ressourcen benötigen, um zu laufen. Diese IT-Ressourcen werden vom *Hypervisor* auf die jeweiligen virtuellen Maschinen partitioniert, sodass mehrere virtuelle Maschinen gleichzeitig laufen können (vgl. redhat Inc. 2020a).

2.1.2 Containerbasierte Virtualisierung

Containerbasierte Virtualisierung gilt als Leichtgewichte Alternative zu herkömmlichen virtuellen Maschinen, weil bei der Erstellung und Ausführung viel weniger IT-Ressourcen benötigt wird. Bei Virtuellen Maschinen werden die Ressourcen vollständig vom Hypervisor abgebildete und Container bilden im Gegensatz zu virtuellen Maschinen nur ein Abbild der benötigten Betriebssysteme und deren Funktionen (vgl. Docker Inc. 2020).

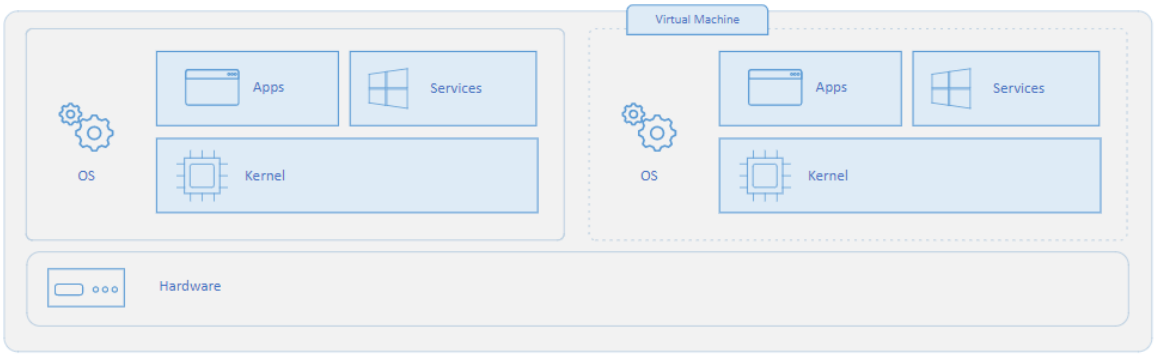
Damit ist ein Container im Prinzip nichts anderes als eine Software, die Code und alle seine Abhängigkeiten zusammenbringt, damit die Anwendung schneller und zuverlässiger ausgeführt werden kann. Ein Container wird mithilfe einer Image-Datei aufgebaut, diese beinhaltet alle erforderlichen Anwendungen wie Code, Systemtools und Systembibliotheken, sodass der Container eigenständig laufen kann (vgl. Docker Inc. 2020). Im Späteren Verlauf werden wir

nochmal detaillierter drauf eingehen was die Image-Datei genau beinhaltet und wie die Syntax zu verstehen ist.

Bei Containerbasierter Virtualisierung spielt es keine Rolle um welches Nutzen es sich handelt, es wird immer leicht und konsistent bereitgestellt (vgl. Cloud Google).

2.1.3 Vergleich von Containern und virtuellen Maschinen

Bei der Isolierung und Zuweisungen der *IT-Ressourcen* ähneln sich Container und virtuelle Maschinen. Woran sie sich unterscheiden, ist die Art der Virtualisierung. Bei einer virtuellen Maschine virtualisiert die Hardware das Betriebssystem und beim Container wird es vom Container selbst virtualisiert (vgl. Docker Inc. 2020).

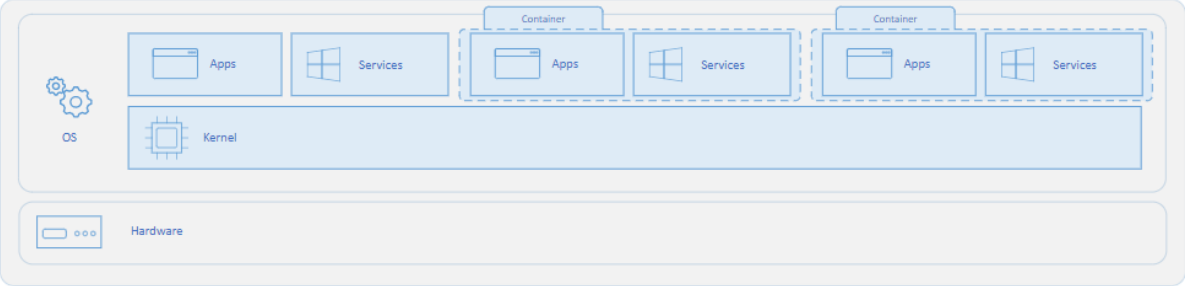


Quelle: (Docs Microsoft Inc. 2020)

Abbildung 1: Architektur von virtueller Maschine

Abbildung 1 zeigt das im Gegensatz zu einem Container eine Virtuelle Maschinen ein vollständiges Betriebssystem mit Samt seine Komponenten Abbildetet (vgl. Docs Microsoft 2019). Man unterscheidet bei der Virtualisierung 2 Arten, einmal die Aggregation von Ressourcen und das Aufsplitten von Ressourcen. Wenn man von aggregiert spricht wird eine virtuelle Umgebung auf mehrere Geräte abgebildet und wenn Ressourcen gesplittet werden, dann wird ein Gerät in mehreren virtuellen Umgebungen aufgeteilt. Diese beiden Faktoren werden dafür verwendet, um eine Optimale Nutzung der Ressourcen anzubieten (Berl et al. 2010). Beim Ausführen eine virtuelle Maschine wird nicht nur die Anwendung selbst, sondern auch die dafür benötigten Ressourcen benutzt, damit diese dann laufen kann. Dies verursacht ein enormer

Overhead und sehr große Abhängigkeit von notwendigen Bibliotheken, vollwertige Betriebssystem und andere mögliche Dienste, die zum Ausführen der Anwendung benötigt wird (vgl. crisp 2014).



Quelle: (Docs Microsofft Inc. 2020)

Abbildung 2:Architektur von Container

Abbildung 2 zeigt mehreren Containern, worauf eine Anwendung auf dem Hostbetriebssystem läuft. Der Container baut auf dem Kernel des Hostbetriebssystem auf und enthält verschiedene *Apps*, *APIs*, und verschiedene Prozesse für das Betriebssystem (vgl. Docs Microsoft 2019). Bei der Ausführung von Containern wird der Linux Kernel und seine Funktionen verwendet, um Prozesse voneinander zu isolieren, damit diese voneinander unabhängig laufen können (vgl. redhat Inc. 2020b). Mithilfe der Isolation kann der Zugriff von mehreren Containern auf dasselbe Kernel erfolgen. Es lässt sich dadurch bestimmten, wie viele Prozessoren, Ram und Bandbreite für jede Container bereitgestellt wird (vgl. crisp 2014).

In der Folgende Tabelle werden wichtige Unterschiede und Gemeinsamkeiten der beiden Virtualisierungsarten erklärt.

	Virtuelle Maschine	**Container**
Isolierung	Die virtuellen Maschinen werden voneinander und vom Hostbetriebssystem isoliert. Es bietet dadurch eine sehr hohe Sicherheitsgrenze an.	Die Container bieten eine vereinfachte Isolierung des Host-Systems und anderen Container, dennoch ist die Sicherheitsgrenze nicht so stark wie bei virtuellen Maschinen.
Betriebssystem	Ein vollständiges Betriebssystem wird ausgeführt mit Kernel. Somit werden mehr Systemressourcen wie *CPU*, *RAM* und Speicherplatz benötigt.	Bei Containern ist die Systemressourcen Verteilung viel flexibler als virtuelle Maschinen. Nur die benötigten Ressourcen, die eine Anwendung für ihre Dienste benötigt wird, auch verwendet.
Bereitstellung	Es können mehrere *VMs* erstellt und verwaltet werden.	Es können auch hier mehrere Container erstellt und verwaltet werden.
Netzwerk	Für jede virtuellen Maschine werden virtuelle Netzwerkadapter erzeugt.	Genauso wie bei virtuellen Maschinen werden virtuelle Netzwerkadapter für jede einzelne Container erzeugt.

Tabelle 1:Unterschied von Container und virtuelle Maschine

Quelle: (vgl. Docs Microsoft Inc. 2020)

2.2 Docker-Container

Docker Container ist ein *Open-Source Project*, somit können alle Anwender egal ob es Private Anwender oder Unternehmen sind, frei benutzen. *Docker Container* können verschiedene *Apps* und *APIs* beinhalten und ausgeführt werden. Beim Ausführen eines *Docker Containers* wird der Linux Kernel verwendet, um *IT-Ressourcen* wie Prozessor, *RAM* und Netzwerk

voneinander zu isolieren. Zudem lassen sich die Container mit der jeweiligen *App* vollständig voneinander isolieren, sodass sie unabhängig laufen können. Dabei werden in einen virtuellen Container sowohl Anwendungen als auch Bibliotheken bereitgestellt und auf mehrere verschiedene Linux Server ausgeführt. Dadurch wird die Portabilitätsgrad und Flexibilität stark erhöht (vgl. crisp 2014). Um den Technischen Hintergrund von Docker zu verstehen gibt es drei wichtige Bestandteile, die von Docker Container verwendet wird. Der erste wichtige Bestandteil ist der *Docker Host*, es handelt sich hierbei um die Laufzeitumgebungen, wo der *Docker Container* ausgeführt wird. Das Erstellen, Ausführen und Terminieren kann mithilfe des *Docker Clients* ausgeführt werden und sorgt zugleich dafür, dass ein Netzwerk definiert werden kann. Mit dem letzten Bestandteil, der *Docker Registry* können Images angelegt werden und mithilfe des Docker Client Container gestartet oder terminiert werden. Mit einem *Snapshot* wird ein Container gespeichert, sodass man den wiederverwenden kann (vgl. eos 2017). Da der Docker Image und daraus erstellte Docker Container kein vollständiges Betriebssystem enthält, ist ein Docker Image viel kleiner als eine virtuelle Maschine und ist dadurch beim Starten viel schneller als eine virtuelle Maschine (vgl. Entwickler 2019).

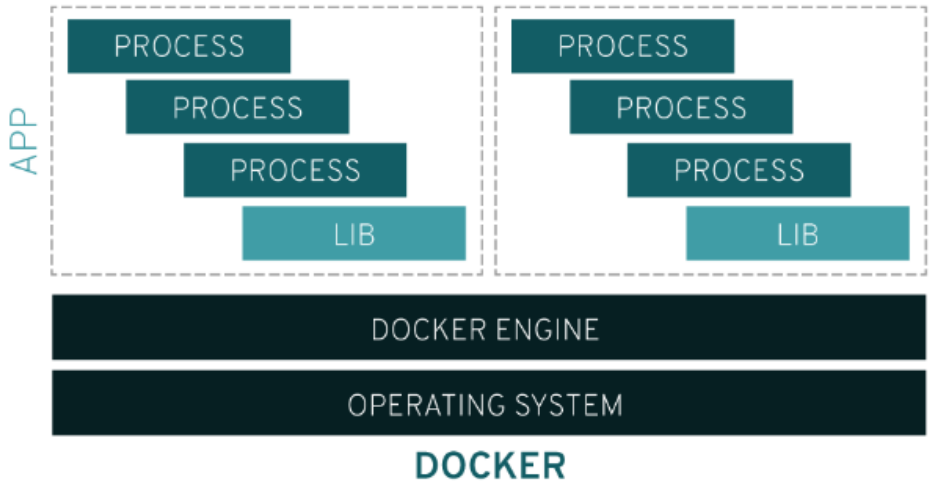


Abbildung 3:Architektur von Docker Container

Quelle: (redhat Inc. 2020)

Abbildung 3 zeigt die Architektur von *Docker Container*. Der *Docker Engine* ist dafür zuständig, dass ein Zugriff auf dem Kernel des Host-Betriebssysteme möglich ist, zusätzlich wird der *Docker Engine* benötigt, um ein *Docker Container* zu erstellen, zu starten oder zu stoppen. Dabei kontaktiert der *Docker Client* den *Docker Engine* und der Container kann erstellt, gestartet oder gestoppt werden (vgl. Entwickler 2019). Ein *Docker Container* wird

mithilfe von *Dockerfile* aufgebaut. Alle Abhängigkeiten lassen sich in einer *Docker Image* durch den *Dockerfile* abbilden (vgl. Entwickler 2019).

2.2.1 Docker CLI

Bei *Docker Container* gibt es das sogenannte Docker *CLI*. Es handelt sich hierbei um eine Dokumentation von Docker Befehlen, die man benötigt, um Beispielweise mithilfe einer Docker Image ein Container laufen zu lassen. In dieser Dokumentation werden sowohl Docker Befehle erklärt und wie sie in der Praxis angewendet werden. Ein besonderer wichtiger Befehl ist beispielsweise der Befehl `docker run`. Dieser Befehl kann in isolierten Docker durchgeführt werden, um aus einer existierende *Docker Image* ein Docker Container zu erstellen und auszuführen. Dabei wird dem *Docker Container* eine Container-ID und Docker Name zugewiesen, um diese dann später stoppen, löschen oder starten zu können. Für eine Internetverbindung wird in der Regel für jede Container Standardmäßig die Automatische Netzwerkbrücke aktiviert. Bei der Erstellung der Netzwerkbrücke wird vom Host-System zum Container eine Netzwerkbrücke aufgebaut, sodass man über das Host Netzwerk eine Internetverbindung herstellen kann. Natürlich hat man auch die Möglichkeit Benutzerdefinierte Einstellungen für Netzwerkverbindungen und andere Sicherheitsaspekte zu konfigurieren. (vgl. Docs. Docker 2020a).

In der Folgende Tabelle werden wichtige Docker *CLI* beschrieben die später für die Leistungsbewertung benötigt wird.

Docker Befehl	**Bedeutung**
docker attach	*Attach* gibt die Standard Eingabe, Ausgabe, und Fehler-Datenströme eines laufenden Containers.
docker build	Baut mithilfe des *Dockerfiles* eine Image Datei.
docker commit	Erstellt aus den Änderungen eines Containers eine neue Image Datei.
docker exec	Das ausführen eines Befehles innerhalb des Containers.
docker images	Die Docker *Images* auflisten.
docker kill	Eine oder mehrere laufende Container killen.

docker login	Einloggen in ein Docker Register.
docker logout	Ausloggen von einem Docker Register.
docker network	Verwalten des Netzwerkes.
docker pause	Pausieren von allem Prozesse innerhalb eines oder mehreren Containern.
docker ps	Die Docker Container auflisten.
docker pull	Ziehen eines Images oder *Repositorys* aus einem Register.
docker push	Hochziehen eines Images oder *Repositorys* in einem Register.
docker rename	Name eines Containers ändern.
docker restart	Neustarten von einem oder mehrere Container.
docker rm	Entfernen von einem oder mehrere Container.
docker rmi	Entfernen von einem oder mehrere *Images*.
docker run	Ausführen von Befehlen in einem neuen Container.
docker stop	Stoppen von einem oder mehreren laufenden Containern.

Tabelle 2: Docker CLI

Quelle: (vgl Docs. Docker 2020b)

2.2.2 Dockerfile

Das *Dockerfile* ist nicht anderes als eine Bauanleitung der benötigt wird, um den Docker Images aufzubauen. Der *Dockerfile* beinhaltet für den Aufbau der *Image* Datei verschiedene Linux Befehle, die dann beim Erstellen des *Docker Images* durchgeführt werden (vgl. Entwickler 2019). Das Format des *Dockerfiles* besteht aus `INSTRUSTION arguments.` Ein Kommentar kann mithilfe von `#` definiert werden. Diese Kommentarzeilen werden bevor der *Dockerfile* zu einem Image aufgebaut wird automatisch gelöscht (vgl. Docs. Docker 2020c).

In der Folgende Tabelle werden wichtige Anweisungen beschrieben die in einem *Dockerfile* benötigt wird, um daraus dann eine *Docker Image* aufzubauen.

INSTRUSTION	**arguments**
MAINTAINER	Autor des Images.
RUN	Ein *shell* Befehl kann dadurch ausgeführt werden.
CMD	Ein *CMD* Befehl kann nach dem Starten eines Docker Containers ausgeführt werden.
EXPOSE	Mithilfe von *EXPOSE* können Ports angegeben werden, auf den der Container dann hört.
ADD	Es ermöglicht komprimierte Dateien automatisch zu öffnen und zu entpacken.
COPY	Dadurch kann der Inhalt vom Host-Betriebssysteme in einem Container kopiert werden.
ENV	Dadurch können Umgebungsvariablen innerhalb des Containers gesetzt werden.
USER	Dadurch kann der *User* festgelegt werden unter welchem die Skripte dann ausgeführt werden können.
ENTRYPOINT	Dadurch kann festgelegt werden, welcher Befehl beim Starten des Containers ausgeführt werden soll

Tabelle 3: Dockerfile-Anweisungen

Quelle: (vgl. anecon, 2018)

2.3 Firecracker Containerd

Firecracker ist ein sehr neues Projekt die vor knapp 2 Jahren von AWS veröffentlicht wurden ist. Es handelt sich dabei um eine neue Virtualiersungstechnik die Open-Source angeboten wird. Die *Geschwindigkeit*, *Ressourceneffizient* und *Leistungen* der Container sollen mit VMs kombiniert werden und dadurch die *Sicherheit* und *Perfomance* der Container erhöht werden.

Firecracker benutzen sogenannten *MicroVMs*, sie bieten im Gegensatz zu *VMs* eine höhere Sicherheit und *Workload-Isolation* an. Zudem sollen *MicroVMs* gleichzeitig, wie normale Container eine hohe *Geschwindigkeit* und *Ressourceneffizienz* erzielen (vgl. Firecracker Microvm 2020). Außerdem wird eine reduzierte Speicherverbrauch und *Sandboxing*-Umgebung für jede *MicroVM* eingerichtet (vgl. Firecracker Microvm 2020). Dadurch werden Anwendungen nicht direkt auf dem Hostbetriebssystem ausgeführt, sondern nur in der eigenen Umgebung (vgl. Cloud Google). In dem nächsten Kapitel wird genauer auf die Funktionsweise des *Firecracker* eingegangen und bestimmte Komponente und Technische Hintergrundwissen aufgeklärt.

2.3.1 Firecracker funktionsweise

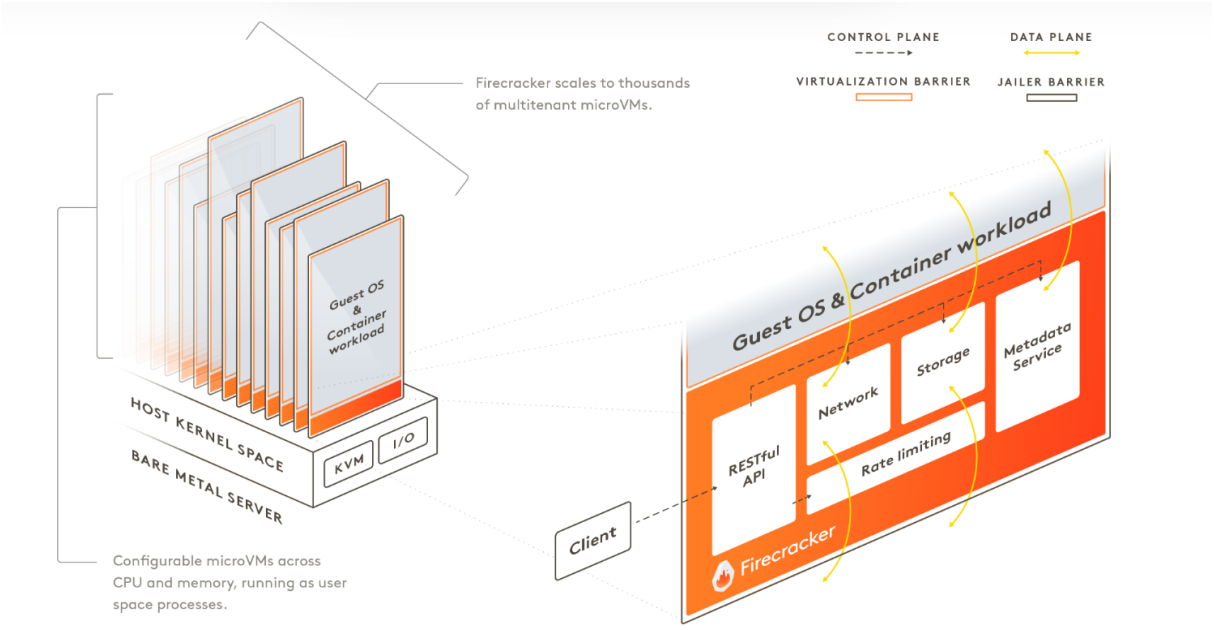


Abbildung 4: Firecracker in Funktionsweise

Quelle: (Firecracker Microvm 2020)

Abbildung 4 zeigt wie *Firecracker* funktionsweise arbeitet. Es wird die *KVM* verwendet, um ein *microVMs* zu erstellen, der erstellte *microVMs* wird im Benutzerbereich ausgeführt. Da ein *microVMs* beim Starten ein geringe Speicheraufwand benötigt, können mehrere bzw. hunderte oder Tausende *microVMs* erstellt und ausgeführt werden. Die erstellten *microVMs* werden in Containergruppen innerhalb einer virtuellen Maschine mit einer Barriere eingekapselt. Mithilfe dieser Funktionen können Workloads auf derselben Maschine ausgeführt werden, ohne sich über die *Sicherheit* und *Effizienz* Gedanken zu machen und beliebige Gastbetriebssystem gehostet werden (vgl. Firecracker MicroVm 2020). *Firecracker* soll einen *VMM* basierend auf dem

KVM implementieren und dadurch ein *RESTful API* anbieten, um *microVMs* zu erstellen und zu verwalten. Die *vCPU* kann unabhängig irrerer Anwendungsanforderungen in beliebigen Kombinationen mit *microVMs* erstellt werden (vgl. Firecracker Microvm 2020).

Um verschieden Netzwerke und Ressourcen, wie beispielsweise Speicher zu steuern gibt es sogenannte *Rate Limiting*, diese können über die *Firecracker API* erstellt und konfiguriert werden. Außerdem gibt es noch eine *Metadata Service*, der für den Austausch von Informationen zwischen dem Host-Betriebssystem und Gast-Betriebssystem zuständig ist. Der Metdatendienst kann genauso wie der *Rate Limiting* über die *Firecracker API* erstellt und konfiguriert werden. Um die Sicherheit von *microVM* zu erhöhen hat man ein *Supporting Programm* mit dem Namen *Jailer* implementiert, diese sorgt für eine Sicherheitsbarriere im Linux *User Space*. Diese Barriere soll eine doppelte Sicherheit gewähren, im Falle der Fälle, wenn die Virtualisierungbarriere durchbrochen wird (vgl. Firecracker Microvm 2020).

2.3.2 Firecracker Containerd Architektur

In diesem Kapitel werden die wichtigsten Komponenten von *Firecracker Containerd* erklärt, damit der Technische Hintergrund einigermaßen verständlich bzw. nachvollziehbar ist. Dazu gehören wichtige Komponente wie *Orchestrator*, *VMM*, *Agent*, *Snapshotter*, *V2 runtime* und der *Control Plugin*.

2.3.2.1 Orchestrator

Container Orchestration ermöglicht, dass verknüpfte mehrere Anwendungen in einem Container, sodass die Anwendungen in einer festgelegten weiße zusammenarbeiten können. So kann der Benutzer beispielweise mehrere Anwendungen gleichzeitig starten und stoppen (vgl. HPE 2020).

2.3.2.2 Agent

Der Agent wird innerhalb von *Firecracker microVM* benötigt, um eine Verbindung mit *runc* aufzubauen und dadurch dann ein Container zu erstellen. Er Kommuniziert außerhalb der *VM* mit dem *Runtime* über die *Vsock* (vgl. Github 2018a).

2.3.2.3 Virtual Maschine Manageröö

Die VMM soll die Ausführung von Containern mit einer Isolierung der virtuellen Maschine erleichtern. Dafür wurden 2 Schnittstellen in *Firecracker Containerd* implementiert, einmal der *Snapshotter* und zum anderen der *V2 runtime* (vgl. Github 2019c).

2.3.2.4 Snapshotter

Snapshotters ist für die Bereitstellung von *Layer Storage* und *UnionFS* für *Containerd* Container zuständig (vgl. Github 2019c).

2.3.2.5 V2 runtime

Der *V2 runtime* ist für die Ausführung von Container Prozessen zuständig, indem sie die benötigten Konfigurationen und Implementationen des Containers bereitstellt (vgl. Github 2019c).

2.3.2.6 Control Plugin

Der Control Plugin ist für die Implementierung von API und Verwaltung des Lebenszyklus von *Runtime* zuständig (vgl. Github 2019c).

2.4 Kata Container

Kata Container verhält sich wie ein herkömmlicher Container, mit dem Unterschiede, dass besonders viel Wert auf die Sicherheit und *Workload-Isolation* geachtet wird gleichzeitig sollen auch die Sicherheitsvorteile von *VMs* erfüllt. Es ist sozusagen eine Kombination aus den Vorteilen eines Containers und *VMs*. Zurzeit befindet sich der *Kata Container* auch wie *Firecracker* am Anfang ihrer Phase und ist deswegen noch in der Entwicklung. Zurzeit kann der *Kata-Container* im Linux Betriebssystem installiert werden. Es ist ein *Open Source Community* Projekt und kann auch kostenlos unter der Lizenz von *Apache 2.0* installiert werden und bei der Entwicklung mitgewirkt werden. Da es sich um ein OSCP Entwicklung handelt, ist Kata-Container drauf angewiesen das freiwillige bei diesem Projekt auch mitwirken (vgl. Kata Container 2020a).

Der Kata-Container Projekt wurde aus 2 verschiedenen Projekten vereinigt, einmal *der Intel Clear Container Project* und *Hyper runV Project*. Man hat die die Technologien dieser beiden

Projekte verschmolzen und daraus entstand dann das *Kata-Container Project* (vgl. Kata Container 2020a). Der *Intel Clear* Container Projekt hat Beispielweise dazu beigetragen, dass der dazu resultierende *Kata-Container* Bootzeiten von <100ms schafft dazu bietet es noch eine höhere Sicherheit an. Der *Hyper runV* hat bei Kata-Container auf die Unterstützung von Technologische-Agnostische fokussiert. Durch diese Zusammenführung bietet der Kata-Container eine Kompatible Leistungsfähige Technologie an (vgl. OpenStack Foundation 2017).

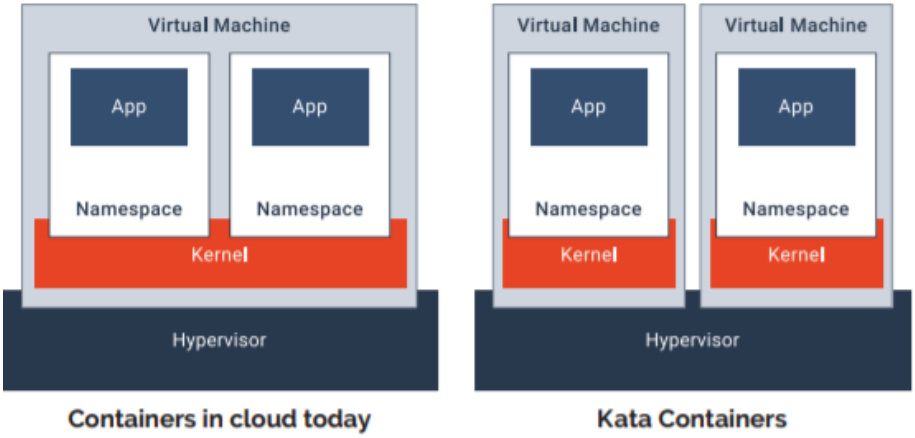


Abbildung 6: Unterschied zwischen Containern in Cloud und Kata Containern

Quelle: (OpenStack Foundation 2017)

Man sieht bei der Abbildung 6, das sowohl der Container in der *Cloud* als auch der *Kata Container* mit dem *Hypervisor* laufen. Der einzige Unterschied besteht darin, das bei *Kata Container* die verschiedenen *Apps* nicht auf derselben virtuellen Maschine ausgeführt werden sondern, jede *Apps* auf seine eigene virtuelle Maschine läuft, so sind die verschiedenen Anwendungen voneinander isoliert, bei normalen Containern laufen alle *Apps* auf der selben virtuelle Maschine. Diese Lösung soll die Sicherheit, Skalierbarkeit und Ressourcenauslastung von *Kata Container* im Gegenzug zu normalem Container enorm steigern. (vgl. OpenStack Foundation 2017).

2.4.1 Kata Container Architektur

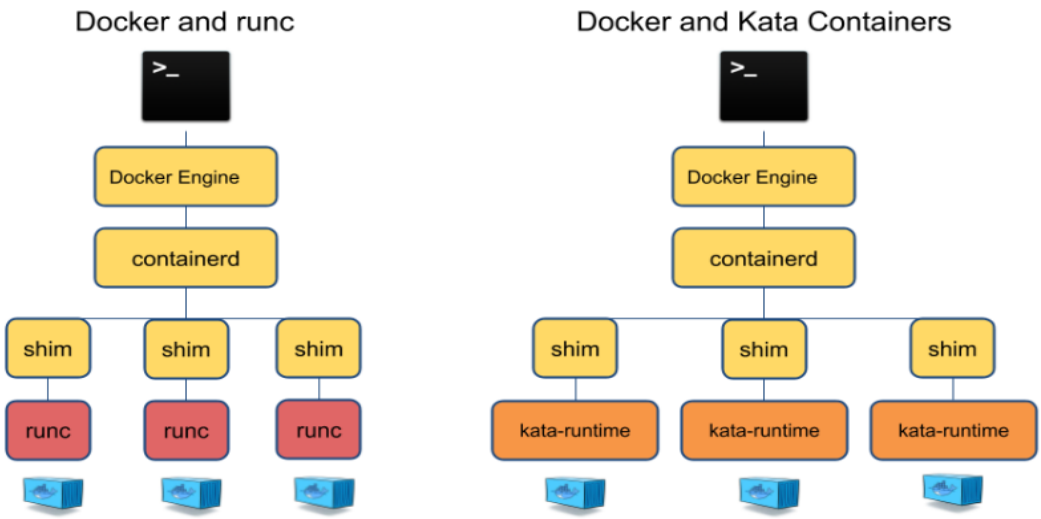


Abbildung 7: Docker und runc im vergleich mit Docker und Kata Containers

Quelle: (Github 2020d)

Die Abbildung 7 zeigt unterschiedliche Anwendungen von *Docker Container*. Die Linke Abbildung zeigt *Docker Container* mit *runc* und die rechte Abbildung zeigt das bei Kata Container stattdessen *kata-runtime* implementiert wurde. Die Kombination aus Docker Engine und *kata-runtime* sollte genauso einwandfrei laufen wie mit *runc*, der kata-runtime erstellt für jede Container ein *QEMU KVM* Maschine, der dann innerhalb des Kata Containers. Der *Kata Shim* ist für die Erstellung von *POD Sandbox* zuständig, diese wird für jede Container erstellt (vgl. Github 2020d). In den Nächsten Kapiteln wird der *Kata Shim* näher erläutert.

2.4.1.1 Guest Assests

Eine virtuelle Maschine wird mit eine minimale Gast-Kernel und Gast-Images, mithilfe von *Hypervisor* gestartet. Der Gast Kernel wird zum Hochfahren der virtuellen Maschine verwendet. Es ist auf minimalen Speicherbedarf ausgeprägt und stellt nur die Dienste, die auch für eine Container erforderlich sind zur Verfügung. Der Gast-Images unterstützt sowohl eine *Initird* als auch ein *Rootfs* Image. Bei der *Rootfs Image* handelt es sich um eine optimierte *Bootstrap System*, der für eine minimale Umgebung sorgt. Im Gegensetz zu dem *Rootfs Image* besteht der *Initird* aus einem komprimierten *Cpio-Archiv*, das als Linux-Startprozess verwendet wird.

Während des Startvorgangs, wird vom Kernel eine Speziale *Tmpfs* entpackt und dadurch dann ein Root Dateisystem erzeugt (vgl. Github 2020d).

2.4.1.2 Agent

Der Agent läuft in einem Gast als Prozess und ist für die Verwaltung von Containern zuständig. Zur wichtigen Ausführungseinheit für *Sandboxing* gehört der *Kata-Agent*. Diese definiert verschiedene *Name Spaces* wie Beispielsweise *PID* oder *IPC* (vgl. Github 2020d).

2.4.1.3 Runtime

Der *Kata-Runtime* nutzt das *Virtcontainers Project*, die eine Agnostische und Hardware-Virtualisierte Container Bibliotheken für Kata-Container bereitstellt (vgl. Github 2020d). Es gibt die Datei *configration.toml* die automatisch bei der Installation des *Kata Containers* erzeugt wird, in dieser Datei werden verschiede Pfade festgelegt. Beispielweise muss man den Pfad angeben, wo sich genau der *Hypervisor* oder die *Image* Datei befindet (vgl. Github 2020d).

2.4.1.4 Shim

(vgl. 2020d)

2.4.1.5 Proxy

(vgl. 2020d)

3 Installation

Ab diesem Abschnitt befassen wir uns nicht mehr mit den Theoretischen Dingen, sondern mit den Praktischen Teilen, um die Leistungen der Container zu bewerten.

In diesem Abschnitt wird erklärt, wie man die Unterschiedlichen Containerlösungen installiert.

3.1 Docker Container installieren

Zunächst erfolgt die Installation von Docker auf einem Ubuntu-Betriebssystem. Danach wird ein *Dockerfile* konfiguriert und daraus dann der *Docker Image* erzeugt. Nachdem ein Image erstellt wurde kann mithilfe des *Docker Images* ein Container ausgeführt. Der *Dockerfile* wird sowohl für *Docker Container*, *Firecracker Containerd* und *Kata Container* als Basis verwendet, um die Leistungen der Container auf eine äquivalente Umgebung durchzuführen und zu bewerten.

Wie schon in der Einführung angedeutet wird zunächst der *Docker Container* auf ein Ubuntu-Betriebssystem installiert. Die Anleitung wie man Docker installiert findet man auf der Offiziellen Docker Seite (siehe. Docs. Docker 2020d).

Zunächst befassen wir uns mit der Docker Installation.

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
```

Quelle: (Docs. Docker 2020d)

Docker 1 Code: Ältere Docker Versionen entfernen

Zuallererst werden ältere Docker Versionen, die schonmal installiert wurden, entfernt.

```
$ sudo apt-get update

$ sudo apt-get install \
      apt-transport-https \
      ca-certificates \
      curl \
      gnupg-agent \
   software-properties-common
```

Quelle: (Docs. Docker 2020d)

Docker 2 Code: Aktualisierung des Systems und Tools installieren

Bei diesem Vorgang wird das Ubuntu-Betriebssystem zunächst aktualisiert und danach die benötigten Tools für die Docker Installation installiert auf das System installiert.

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

$ sudo apt-key fingerprint 0EBFCD88

pub   rsa4096 2017-02-22 [SCEA]

      9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88

uid           [ unknown] Docker Release (CE deb) <docker@docker.com>

sub   rsa4096 2017-02-22 [S]
```

Quelle: (Docs. Docker 2020d)

Docker 3 Code: Das GPG Herunterladen und überprüfen

Im dritten Schritt wird der Docker *GPG* heruntergeladen und dann mit dem *Fingerprint* Befehl überprüft, dieser Vorgang wird wegen Sicherheitsgründen durchgeführt. Die `pup` Ausgabe muss mit dem folgenden Öffentliche Schlüssel von Docker `9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88` übereinstimmen.

```
$ sudo add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"
```

Quelle: (Docs. Docker 2020d)

Docker 4 Code: Repository von Docker hinzufügen

Hier wird der *Repository* von Docker auf unser Ubuntu System hinzugefügt. Je nach Linux System muss der Download link am Ende des Pfades verändert werden. Bei einem Debian System würde die Download Adresse folgendermaßen aussehen: `https://download.docker.com/linux/debian`.

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Quelle: (Docs. Docker 2020d)

Docker 5 Code: System Aktualisieren und Docker Engine installieren

Nachdem der *Repository* heruntergeladen wurde, wird das System noch einmal aktualisieren und anschließend der Docker Engine installiert, um beispielsweise Container starten oder stoppen zu können.

```
$ apt-cache madison docker-ce

        docker-ce | 5:18.09.1~3-0~ubuntu-xenial | https://download.docker.com/linux/ubuntu  xenial/stable amd64 Packages
        docker-ce | 5:18.09.0~3-0~ubuntu-xenial | https://download.docker.com/linux/ubuntu  xenial/stable amd64 Packages
        docker-ce | 18.06.1~ce~3-0~ubuntu       | https://download.docker.com/linux/ubuntu  xenial/stable amd64 Packages
        docker-ce | 18.06.0~ce~3-0~ubuntu       | https://download.docker.com/linux/ubuntu  xenial/stable amd64 Packages

$ sudo apt-get install docker-ce=<VERSION_STRING> docker-ce-cli=<VERSION_STRING>
containerd.io

$ sudo docker run hello-world
```

Quelle: (Docs. Docker 2020d)

Docker 6 Code: Docker Engine Version auswählen

Mit dem Ersten Befehl wird der *Docker Cache* aufgerufen, um eine *Docker Engine* Version für sein Linux System auszuwählen. In unserem Fall wird ein *Ubuntu 18.06* verwendet, somit wird der *Docker Engine* Version *18.06.0~ce~3-0~ubuntu* `ausgewählt.` Dies Versionsnummer wird mit den Platzhaltern des unteren Befehles ersetzt und durchgeführt. Nachdem der Docker Engine erfolgreich installiert wurde, kann man den *Daemon* mit dem Befehl `sudo service Docker start/status/stop`, starten, stoppen oder den Status abfragen. Wenn der *Daemon* keine Probleme aufweist kann man mit dem Befehl `sudo docker run hello-word`, den ersten *Docker Container* starten.

3.1.1 Dockerfile

Bis hierhin wurde der Docker erfolgreich installiert und somit auch startbereit. Im nächsten Schritt erstellen wir ein *Dockerfile* und konfiguriert diese, um ein *Docker Image* für die Leistungsbewertung der drei Containerlösungen aufzubauen.

Zunächst muss der *Dockerfile* mit dem Befehl `sudo touch Dockerfile` erzeugt werden. Nachdem der *Dockerfile* erzeugt wurde wird diese konfiguriert.

```
# Dockerfile für Firecracker-Container, Kata-Container und Docker-Container.
# Installation von: Debian System, wichtige Tools und Webserver Apache2.
```

```
from debian # Zeile:1

MAINTAINER Vahel Hassan <Vahel.Hassan@outlook.de> # Zeile:2
RUN apt-get update && apt-get install -y # Zeile:3
# Tools die wir später für unsere Leistungsbewertung benötigen
RUN apt-get install sysstat -y # Zeile:4
RUN apt-get install nicstat -y # Zeile:5
RUN apt-get install iftop -y # Zeile:6
RUN apt-get -y install nano # Zeile:7
RUN apt-get -y install wget # Zeile:8
# Install Benchmark
RUN echo "deb http://deb.debian.org/debian  stretch main" >> /etc/apt/sources.list # Zeile:9
RUN echo "deb-src http://deb.debian.org/debian  stretch main" >> /etc/apt/sources.list # Zeile:10
RUN apt-get update # Zeile:11
RUN apt-get install -y libmariadbclient18 # Zeile:12
RUN apt-get install -y sysbench # Zeile:13
# Install Apache2
RUN apt-get install apache2 -y # Zeile:14
RUN mkdir /run/lock # Zeile:15
RUN mkdir -p /var/www/ # Zeile:16
RUN chown -R $USER:$USER /var/www/ # Zeile:17
RUN chmod -R 755 /var/www/ # Zeile:18
RUN service apache2 restart # Zeile:19
```

Quelle: (vgl. Docs. Docker 2020d)

Dockerfile 1 Code: Dockerfile Konfiguration

Der *Dockerfile* beinhaltet die Bauanleitung für das Image, dieses Image wird für die Leistungsbewertung der 3 verschiedenen Containerlösungen verwendet.

In Zeile 1 wird die Anweisung *from* aufgerufen. Mithilfe dieser Anweisung wird das *Linux System Debian* auf unsere Image Datei installiert.

Danach wird in Zeile 2 der Autor und die dazugehörige E-Mail-Adresse mithilfe der Anweisung *MAINTAINER* definiert. Die E-Mail-Adresse wird benötigt, um den Autor bei Problemen oder Fragen kontaktieren zu können.

In Zeile 3 wird beim Aufbauen des Docker Images, der erstellte Debian System mit den Befehlen `update` aktualisiert und mit `install` können andere verschiedene Pakete auf das Debian System installiert sowohl später als auch beim Erstellen des *Docker Images*.

In den Zeilen 4-6 werden wichtige *Tools* mit der Anweisung `run` installiert, diese *Tools* werden eventuell für die Leistungsbewertung der Containerlösungen benötigt.

Von Zeile 9 -13 wird das *Tool Benchmark* installiert. Mit *Benchmark* hat man die Möglichkeit die Leistungsfähigkeit eines Systems zu überprüfen (vgl. Wiki Ubuntuusers 2020a). In der Arbeit wird *Benchmark* benötigt, um einzelne System Komponenten wie Beispielsweise *CPU* auf ihre Leistungen mit verschiedene Testdurchläufe zu bewerten.

Die Zeile 14-19 beschreibt wie Apache2 installiert wird. Bei Apache 2 handelt es sich um einer der meist verwendeten Webserver im Internet (vgl. Wiki Ubuntuusers 2020a). Apache2 wird später für die Bewertungen der Container Startseiten benötigt.

3.1.2 Docker starten

Nachdem der *Dockerfile* erstellt wurde kann man diese mit dem Befehl `docker build` aufbauen. Nachdem aufbauen wird ein *Docker Images* erstellt, die Erstellung des *Images* ermöglicht es mit dem Docker Befehl `run` ein Container auszuführen.

```
$ docker build -t debian-test .
$ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
debian-test                 latest              3417c1c607bc        5 seconds ago       352MB
```

Quelle: (vgl. Docs. Docker 2020d)

Dockerfile 2 Code: Dockerfile aufbauen und das erstellte Image auflisten

In diesem Code wird zunächst der *Dockerfile* aufgebaut. Mit `docker build` kann mit einem erstellen *Dockerfile* eine *Docker Image* aufgebaut werden. Mit dem ganzen Befehl wird ausgedrückt, dass der *Dockerfile* im aktuellen Pfad verwendet werden soll, um ein *Docker Image* mit dem Namen *debian-test* aufzubauen. Der nächste Befehl `docker images` listet alle unsere erstellten Images auf.

```
$ docker run -it -d debian-test
b78f6ce04028a3db8ab5c0a87d02388c3272bbc2e1e6d9725196703131a7ea16
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b78f6ce04028        debian-test         "bash"              7 seconds ago       Up 4 seconds                            bold_poincare
```

Quelle: (Docs. Docker 2020d)

Dockerfile 3 Code: Mit Docker Images ein Container ausführen

Jetzt wird mithilfe des *Docker Images*, die unser ganze Bauanleitung für den Container beinhaltet ein Container ausgeführt. Der Docker Befehl `docker run -it -d debian-test`, macht die Ausführung eines Containers im Hintergrund möglich. Nachdem ein Container ausgeführt wurde erhält es ein *Container ID*, diese *Container ID* wird verwendet, um beispielsweise mit dem Docker Befehl ein Container zu `stoppen` oder zu `starten`. Die Container IDs können mit dem Befehl `docker ps` aufgelistet werden. Wenn irgendwelche Änderungen innerhalb des erstellten Docker Container stattfinden, kann man mit dem *docker commit* Befehl eine neue Image Datei aus dem Docker Container erstellen. Im Dockerfile 3 Code wird aus dem Container *debian-test* eine neue *Docker Image* Datei erzeugt. Der Befehl beinhaltet die ID des Containers und Name des neuen Images.

Zusammenfassend ist die Installation von Docker Container hiermit erfolgreich gelungen, es wurde ein *Dockerfile* erstellt, daraus dann ein *Docker Image* aufgebaut und anschließend mithilfe dieses Images ein *Docker Container* ausgeführt.

3.2 Firecracker Containerd installieren

Für die Installation von Firecracker-Containerd wird von der offiziellen Firecracker GitHub Repositories verwendet (siehe. Github 2020e und Github 2020f). Für Firecracker wird auch Docker Container benötigt, weil dieselbe Docker Image für den Aufbau des Containerd benötigt wird. Wie die Image Datei erstellt wird, kann man in dem Kapitel 3.1.1 nochmal einsehen, dort wird es ausführlich erklärt.

```
#!/bin/bash
err=""; \
[ "$(uname) $(uname -m)" = "Linux x86_64" ] \
  || err="ERROR: your system is not Linux x86_64."; \
[ -r /dev/kvm ] && [ -w /dev/kvm ] \
  || err="$err\nERROR: /dev/kvm is innaccessible."; \
(( $(uname -r | cut -d. -f1)*1000 + $(uname -r | cut -d. -f2) >= 4014 )) \
  || err="$err\nERROR: your kernel version ($(uname -r)) is too old."; \
dmesg | grep -i "hypervisor detected" \
  && echo "WARNING: you are running in a virtual machine. Firecracker is not well
tested under nested virtualization."; \
[ -z "$err" ] && echo "Your system looks ready for Firecracker!" || echo -e "$err"
```

Quelle: (Github 2020f)

Firecracker 1 Code: Hardware überprüfen

Bevor die richtige Installation von Firecracker begonnen werden kann, muss die Hardware überprüft werden, ob die benötigten Voraussetzungen für die Virtualisierung von Firecracker

erfüllt. Als Ergebnis sollten folgende Zeilen im Terminal angezeigt werden: *Your system looks ready for Firecracker!*

```
$ git clone https://github.com/firecracker-microvm/firecracker-containerd.git
$ cd firecracker-containerd
$ sudo make install install-firecracker demo-network
```

Quelle: (Github Docker 2020e)

Firecracker 2 Code: Firecracker-Containerd Repositories Herunterladen

Zuallererst wird der *Firecracker Repository* von der GitHub Seite mit dem Befehl `git clone` heruntergeladen und anschließend und für den Container ein Demo Netzwerk installiert, um später eine Internetverbindung aufzubauen.

```
  - runtime/containerd-shim-aws-firecracker

  - firecracker-control/cmd/containerd/firecracker-containerd

  - firecracker-control/cmd/containerd/firecracker-ctr
```

Quelle: (Github 2020f)

Firecracker 3 Code: Installierte Binärdateien

Beim erfolgreichen Herunterladen des *Repository*s sollten die drei Binärdateien die im Firecracker 4 zusehen sind erstellt wurden sein. Diese 3 Binärdaten benötigen wir später, um *den firecracker-Containerd* auszuführen und mithilfe des *firecracker-ctr* ein Container zu starten.

```
$ curl -fsSL -o hello-vmlinux.bin https://s3.amazonaws.com/spec.ccfc.min/img/hello/kernel/hello-vmlinux.bin
```

Quelle: (Github 2020f)

Firecracker 4 Code: VM Linux Kernel installieren

Damit der *Firecracker Containerd* auch hochfahren kann benötigt es ein *KVM*. Der Linux Kernel wird von Amazone bereitgestellt und kann mit dem obigen Befehl heruntergeladen werden.

```
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2376
```

Quelle: (Unix Stackexchange 2019)

Firecracker 5 Code: Docker API-Socket aktivieren

Damit ein *Rootfs* im nächsten Schritt installiert werden kann muss der *Docker API-Socket* aktiviert werden. Zunächst wird die Datei *startup_options.conf* geöffnet und anschließend die Zeilen im *Firecracker* 5 Code hinzugefügt. Der *Docker Daemon* wird dann mit den Befehlen `sudo systemctl daemon-reload` und `sudo systemctl restart docker.service` Aktualisiert, dadurch werden die Änderungen übernommen und der *Docker API-Socket* aktiviert.

```
$ make image
$ sudo mkdir -p /var/lib/firecracker-containerd/runtime
$ sudo cp tools/image-builder/rootfs.img /var/lib/firecracker-
containerd/runtime/default-rootfs.img
```

Quelle: (Github 2020f)

Firecracker 6 Code: Image Datei

Nachdem der API-Socket aktiviert wurden ist muss das Image mit dem Befehl `make image` installiert. Danach wird ein Ordner mit dem Namen *runtime* erzeugt und die erstellte Image Datei reinkopiert.

```
disabled_plugins = ["cri"]
root = "/var/lib/firecracker-containerd/containerd"
state = "/run/firecracker-containerd"
[grpc]
  address = "/run/firecracker-containerd/containerd.sock"
[plugins]
  [plugins.devmapper]
    pool_name = "fc-dev-thinpool"
    base_image_size = "10GB"
    root_path = "/var/lib/firecracker-containerd/snapshotter/devmapper"
[debug]
  level = "debug"
```

Quelle: (Github 2020e)

Firecracker 7 Code: config.toml Konfiguration

Damit der *Firecracker* auch richtig ausgeführt werden kann benötigt es zu wichtige Speicherorte, diese müssen in der Datei *config.toml* angegeben werden. Im *root* Speicherort werden wichtige Ordner wie Beispielsweise *Runtime*, *Metadata* oder *Snapshotter* angelegt, um wichtige Informationen über einen erstellten Container zu speichern. Diese Informationen kann man wie eine Art Datenbank für erstellte Containernd betrachten. Im *state* Anweisung wird

beispielsweise der *containerd.sock* abgelegt, der Socket wird für das starten eines Images benötigt. Die *address* gibt den Pfad an, wo der *containerd.sock* abgelegt ist, damit diese dann auch verwendet werden kann. Zum Schluss werden *Plugins* für den *devmapper-snapshotter* angegeben, wie zum Beispiel der *Name*, die Größe des *Images* und wo diese Informationen abgelegt werden sollen.

Als nächstes wird ein *Thinpool* erstellt, diese wird für den *Snapshotter* benötigt. Wie diese zu konfigurieren ist, wird am Anhang A 1 anhand eines Shell-Skriptes genau gezeigt.

```
{
  "firecracker_binary_path": "/usr/local/bin/firecracker",
  "kernel_image_path": "/var/lib/firecracker-containerd/runtime/hello-vmlinux.bin",
  "kernel_args": "console=ttyS0 noapic reboot=k panic=1 pci=off nomodules ro sys-
temd.journald.forward_to_console systemd.unit=firecracker.target init=/sbin/over-
lay-init",
  "root_drive": "/var/lib/firecracker-containerd/runtime/default-rootfs.img",
  "cpu_template": "T2",
  "log_fifo": "fc-logs.fifo",
  "log_level": "Debug",
  "metrics_fifo": "fc-metrics.fifo",
  "shim_base_dir": "/root/go/src/github.com/firecracker-containerd/runtime/contai-
nerd-shim-aws-firecracker",
  "default_network_interfaces": [{
    "CNIConfig": {
      "NetworkName": "fcnet",
      "InterfaceName": "veth0"
    }
  }]
}
```

Quelle: (Github 2020f und Github 2020e)

Firecracker 8 Code: firecracker-runtime.json Konfiguration

In diese Datei werden wichtige Speicherorte angegeben, die wir davor installiert haben. Beispielweise haben wir im *Firecracker 4 Code* das Kernel für den Containernd installiert, der Speicherort des Kernels wird dann entsprechend in der *"kernel_image_path"* Zeile definiert. Dann erfolgen noch wichtige Pfade für den installierten Image die *im Firecracker Code 6* und *Firecracker Code 2* durchgeführt wurde. Damit später eine stabile Internetverbindung erfolgen kann wird ein *Interface Name* mit *Network Name* für den *Network Interface* definiert. Die nicht erwähnten Metriken und Einstellungen sind meistens standardisiert und müssen nicht verändert werden.

3.2.1 Firecracker starten

Zum Starten von Firecracker Containerd werden 2 Terminals benötigt.

```
$ sudo PATH=$PATH ~/go/src/github.com/firecracker-containerd/firecracker-
control/cmd/containerd/firecracker-containerd  \
  --config /etc/containerd/config.toml
```

Quelle: (Github 2020f)

Firecracker 9 Code: Terminal 1 starten

Im *Firecracker 2 Code* wurde die Binärdatei *firecracker-containerd* installiert, diese Datei wird im Terminal 1 benötigt, um die *Firecracker Containerd* Umgebung zu starten. Die Umgebung wird mithilfe der Einstellung von *config.toml* Datei die wir in *Firecracker Code 7* beschrieben hatten realisiert.

```
$ sudo firecracker-ctr --address /run/firecracker-containerd/containerd.sock \
  run \
  --snapshotter devmapper \
  --runtime aws.firecracker \
  --rm --tty --net-host \
  docker.io/library/busybox:latest busybox-test
```

Quelle: (Github 2020f)

Firecracker 10 Code: Terminal 2 starten

Während im Terminal 1 die Umgebung von *Firecracker Containerd* realisiert wurde kann im Terminal 2 ein oder mehrere Container mithilfe der Binärdatei *firecracker.ctr* gestartet werden. Diese Binärdatei wurde im *Firecracker Code 2* installiert. *Firecracker-ctr* stellt dem Benutzer wie Docker verschiedene Befehle zur Verfügung. Die *CTR* Befehle sind zu Docker Befehle fast Äquivalent zu verstehen. Beispielweise kann mit sowohl mit Docker als auch mit *CTR* den Befehl `run` verwendet werden, um ein Container mithilfe eines *Docker Images* zu starten. Beim Starten des Containers werden noch die folgenden Optionen in der Tabelle ausgeführt.

CTR Options	**Erklärung**
--snapshotter	Hier wird der Name des *Snapshots* angegeben.
--runtime	Hier wird der Laufzeitname angegeben.
--rm	Nachdem ausführen des Containers wird diese wieder gelöscht.
--tty	Der Container wird einem tty zugeordnet.

--net-host	Dadurch wird der Host-Netzwerk für den Container aktiviert.

Tabelle 4: CTR-Options

Quelle: (vgl. Systutorials)

Im *Firecracker Code 10* ist der Name des *Snappshots devmapper* und der Name unsere Laufzeitumgebung lautet *aws.firecracker*. Wenn der Container ausgeführt wird dann soll es wieder gelöscht werden, zudem soll der Container zu einem *tty* zugeordnet sein und das Host-Netzwerk aktiviert werden, um eine Internetverbindung aufzubauen. In der letzten Zeile wird der *Docker Image* aus *Dockerfile Code 1* verwendet, um den *Firecracker-Containerd* zu starten.

Nachdem dieser Vorgang erfolgreich durchgeführt wurde, wird man im Container eingeloggt.

Am Anhang A 2 wird gezeigt, wie man *Firecracker* ohne *Containerd* installiert und startet (siehe. Guthub 2020g).

3.3 Kata Container installieren

Für die Installation von Kata Container wird die offiziellen Kata Container GitHub Repository verwendet (siehe. Github 2020h). Wie bei der Installation von Firecracker Containerd müssen auch hier wichtige Tools installiert werden. Der Kata Container läuft mithilfe von Docker Container Umgebung, deswegen ist die Installation von Docker Container einer der wichtigen Voraussetzungen von Kata Container. Die genaue Installation von Docker Container wurde im Kapitel 3.1 erläutert, deswegen werden wir in diesem Kapitel nicht näher drauf eingehen.

```
$ sudo apt-get update
$ sudo apt-get -y upgrade
$ cd /tmp
$ wget https://dl.google.com/go/go1.13.3.linux-amd64.tar.gz
$ sudo tar -xvf go1.13.3.linux-amd64.tar.gz
$ sudo mv go /usr/local

$ export GOROOT=/usr/local/go
$ export GOPATH=~/go
$ export PATH=$GOPATH/bin:$GOROOT/bin:$PATH

$ go env // Pfade überprüfen
```

Kata Container Code 1: Goolang installieren

Goolang wird bei der Installation von *Kata Container* benötigt, um verschiedene *Repositories* herunterzuladen. Neben *Goolang* werden noch andere wichtige *Tools* wie am Anhang A 3 zusehen installiert.

```
$ go get -d -u github.com/kata-containers/runtime
$ cd $GOPATH/src/github.com/kata-containers/runtime
$ make && sudo -E PATH=$PATH make install
```

Quelle: (Github 2020h)

Kata Container 1 Code: Repository Herunterladen

Zunächst laden wir uns mithilfe von *Goolang* den Kata Container *Repository* herunter. Danach wird im Ordner *runtime* die *Makefile* Datei aufgebaut. Mit diesem Vorgang wird der *Repository* Architektur *realisiert* und der *Kata Container Repository* wurde erfolgreich installiert.

```
$ sudo kata-runtime kata-check
```

Quelle: (Github 2020h)

Kata Container 2 Code: Hardware überprüfen

Nachdem der *Kata Container Repository* installiert wurde, ist zu überprüfen ob auch das System die benötigten Hardware Ressourcen verfügt, um ein *Kata Container* auszuführen.

```
System is capable of running Kata Containers
System can currently create Kata Containers
```

Kata Container 3 Code: Ergebnisse nach der Hardware überprüfung

Als Resultat sollte im Terminal die Ausgabe von Kata Container 3 Code angezeigt werden.

Wie auch bei *Firecracker Containerd* benötigt auch der *Kata Container* ein installiertes Image oder *Initrd* Datei. Bevor die Installation einer der beiden erfolgen kann, müssen Vorkehrungen getroffen werden. Beim installieren des *Repositories* wurde auch eine Date mit dem Namen *configuration.toml* installiert. Diese Datei muss konfiguriert werden. Wie diese zu konfigurieren ist wird im nächsten Schritt erläutert.

```
[hypervisor.qemu]
path = "/usr/bin/qemu-system-x86_64"
kernel = "/usr/share/kata-containers/vmlinuz.container"
# initrd = "/usr/share/kata-containers/kata-containers-initrd.img"
image = "/usr/share/kata-containers/kata-containers.img"
machine_type = "pc"
```

Quelle: (Github 2020h)

Kata Container 4 Code: configuration.toml Konfiguration

In der Datei *configuration.toml* werden wichtige Speicherorte festgelegt, wie zum Beispiel der Pfad zum QEMU Datei oder das Kernel System und der Speicherort des verwendeten Images. In den nächsten schritten wird genaue auf die Installationen dieser Dateien eingegangen. Wichtig hier zu beachten ist, ob das *Image* oder *Initrd* für das Ausführen des Kata Container verwendet werden soll. Im Kapitel 3.2 hatte man auch die Auswahl zwischen diesen beiden Dateien. Da die Image Datei bei *Firecracker Containerd* ausgewählt wurde, wird auch hier die Image Datei für *Kata Container* verwendet, deswegen wird die Zeile `initrd = "/usr/share/kata-containers/kata-containers-initrd.img"` mit einem # ausgeklammert.

```
$ sudo mkdir -p /etc/kata-containers/
$ sudo install -o root -g root -m 0640 /usr/share/defaults/kata-containers/configu-
ration.toml /etc/kata-containers
$ sudo sed -i -e 's/^# *\(enable_debug\).*=.*$/\1 = true/g' /etc/kata-contai-
ners/configuration.toml
$ sudo sed -i -e 's/^kernel_params = "\(.*\)"/kernel_params = "\1 agent.log=debug
initcall_debug"/g' /etc/kata-containers/configuration.toml
```

Quelle: (Github 2020h)

Kata Container 5 Code: Debug für Kata Shim aktivieren

Mit den Befehlen im Kata Container 5 Code wird in der Datei *configuration.toml* der *Debug* für *Kata Shim* aktiviert. In Kapitel 2.4.1.4 wird die Funktion von Kata Shim ausführlich erläutert. Jetzt muss in der Datei *config.toml* die Zeilen *[debug]*und *level = "debug"* hinzugefügt werden. Die Bedeutung und Konfiguration von *config.toml* wurde im Kapitel 3.2 ausführlich erklärt. Somit sollte der *Debug* für Kata-Shim aktiviert sein.

```
#!/bin/bash
# Kata Shim installieren
go get -d -u github.com/kata-containers/shim
cd $GOPATH/src/github.com/kata-containers/shim
make && sudo make install

# Kata Proxy installieren
go get -d -u github.com/kata-containers/proxy
cd $GOPATH/src/github.com/kata-containers/proxy
make && sudo make install
```

Quelle: (Github 2020h)

Kata Container 6 Code: Proxy und Shim installieren

Nachdem der Kata Shim aktiviert wurde, wird der Kata Shim und Kata Proxy installiert. Im Kapitel 2.4.1.5 wird die Funktion von Kata Proxy ausführlich erklärt.

4 Leistungsbewertung

5 Zusammenfassung

6 Ausblick

Literaturverzeichnis

Witt, M., Jansen, C., Breuer, S., Beier, S., Krefitng, D. (2017), Artefakterkennung über eine cloud-basierte Plattform, 19. September, https://link.springer.com/article/10.1007/s11818-017-0138-0#citeas, letzter Zugriff: 22.07.2020.

DOCKER.de Inc. (2020), What is a Container?, https://www.docker.com/resources/what-container, letzter Zugriff: 23.07.2020.

KATACONTAINERS.io (2020a), An overview of the Kata Containers project, https://katacontainers.io/learn/, letzter Zugriff: 23.07.2020.

AWS.AMAZONE.com (2020), Jetzt neu: Firecracker, ein neues Virtualisierungstechnologie- und Open-Source-Projekt zur Ausführung von Mehrmandanten-Container-Workloads, 26. November, https://aws.amazon.com/de/about-aws/whats-new/2018/11/firecracker-lightweight-virtualization-for-serverless-computing/, letzter Zugriff 23.07.2020.

FIRECRACKER-MICROVM.GITHUB.io (2018), Firecracker is an open source virtualization technology that is purpose-built for creating and managing secure, multi-tenant container and function-based services, https://firecracker-microvm.github.io/, letzter Zugriff: 23.07.2020.

Buhl, H., Winter, R. (2008), Vollvirtualisierung – Beitrag der Wirtschaftsinformatik zu einer Vision, 13. Dezember, https://link.springer.com/article/10.1007/s11576-008-0129-7, letzter Zugriff: 23.07.2020.

IONOS.de (2020), Konzepte der Virtualisierung im Überblick, 24. März, https://www.ionos.de/digitalguide/server/konfiguration/virtualisierung/, letzter Zugriff: 24.07.2020.

REDHAT.com Inc. (2020a), Was ist Virtualisierung?, https://www.redhat.com/de/topics/virtualization/what-is-virtualization, letzter Zugriff: 24.07.2020.

REDHAT.com Inc. (2020b), Docker – Funktionsweise, Vorteile, Einschränkungen, https://www.redhat.com/de/topics/containers/what-is-docker, letzter Zugriff: 24.07.2020.

CLOUD.GOOGLE.com, CONTAINER BEI GOOGLE,

https://cloud.google.com/containers?hl=de, letzter Zugriff: 24.07.2020.

DOCS.MICROSOFT.com Inc. (2019), Container im Vergleich zu virtuellen Computern, 21. Oktober, https://docs.microsoft.com/de-de/virtualization/windowscontainers/about/containers-vs-vm, letzter Zugriff, 24.07.2020.

Berl. A., Fischer A., Meer H. (2010), Virtualisierung im Future Internet, 16. Februar, https://link.springer.com/article/10.1007/s00287-010-0420-z, letzter Zugriff: 24.07.2020.

DOCS.DOCKER.com (2020a), Docker run reference, https://docs.docker.com/engine/reference/run/, letzter Zugriff: 24.07.2020.

DOCS.DOCKER.com (2020b), The base command for the Docker CLI, https://docs.docker.com/engine/reference/commandline/docker/, letzter Zugriff: 24.07.2020.

DOCS.DOCKER.com (2020c), Build an image from a Dockerfile, https://docs.docker.com/engine/reference/commandline/build/, letzter Zugriff: 24.07.2020.

EOSGMBH.de (2017), Was sind eigentlich Container und Docker?, 13. September, https://www.eosgmbh.de/container-und-docker, letzter Zugriff: 24. 07.2020.

CRISP-RESEARCH.com (2014), Docker Container: Die Zukunft moderner Applikationen und Multi-Cloud Deployments?, 31. Juli, https://www.crisp-research.com/docker-container-die-zukunft-moderner-applikationen-und-multi-cloud-deployments/, letzter Zugriff: 24.07.2020.

ENTWICKLER.de (2019), Docker: Einstieg in die Welt der Container, 19. Februar,, https://entwickler.de/online/windowsdeveloper/docker-grundlagen-dotnet-container-579859289.html, letzter Zugriff: 24.07.2020.

ANECON.com (2018), Docker Basics – Befehle und Life Hacks, 26. Juni, http://www.anecon.com/blog/docker-basics-befehle-und-life-hacks/, letzter Zugriff: 25.07.2020.

GITHUB.com (2018a), agent, 19. November, https://github.com/firecracker-microvm/firecracker-containerd/blob/master/docs/agent.md, letzter Zugriff: 26.07.2020

Lius L., Brown M. (2017), Containerd Brings More Container Runtime Options for Kubernetes, 02. November, https://kubernetes.io/blog/2017/11/containerd-container-runtime-options-kubernetes/, letzter Zugriff: 26.07.2020.

GITHUB.com (2020a), runc, 23. Juli, https://github.com/opencontainers/runc, letzter Zugriff: 26.07.2020.

MAN7.org (2020), vsock(7) – Linux manual page, 2. September, https://man7.org/linux/man-pages/man7/vsock.7.html, letzter Zugriff: 26.07.2020.

GITHUB.com (2019c), firecracker-containerd architecture, 04. Juni, https://github.com/firecracker-microvm/firecracker-containerd/blob/master/docs/architecture.md, letzter Zugriff: 26.07.2020

Floyd B., Dr. Bergler A. (2017), Was ist Storage?, 19. Oktober, https://www.it-business.de/was-ist-storage-a-663183/, letzter Zugriff 26.07.2020.

LINUX-MAGAZIN.de (2005), Überlagerte Dateisysteme in der Praxis, https://www.linux-magazin.de/ausgaben/2005/10/hochstapler/, letzter Zugriff: 26.07.2020.

GITHUB.com (2020c), Firecracker Design, 01. Juli, https://github.com/firecracker-microvm/firecracker/blob/master/docs/design.md#host-integration, letzter Zugriff: 26.07.2020.

HPE.com (2020), WAS IST CONTAINER ORCHESTRATION?, https://www.hpe.com/de/de/what-is/container-orchestration.html, letzter Zugriff: 26.07.2020.

GITHUB.com (2020d), Kata Containers Architecture, 20. Juli, https://github.com/kata-containers/documentation/blob/master/design/architecture.md, letzte Zugriff: 26.07.2020.

OpenStack Foundation (2017), The speed of containers, the security of VMs, https://katacontainers.io/collateral/kata-containers-1pager.pdf, letzter Zugriff: 26.07.2020.

Rouse M. (2020), Agnostisch, https://whatis.techtarget.com/de/definition/Agnostisch, letzter Zugriff: 26.07.2020.

DOCS.DOCKER.com SPI Inc. (2020d), Build an image from a Dockerfile, https://docs.docker.com/get-docker/, letzter Zugriff: 30.07.2020.

PACKAGES.DEBIAN.org SPI Inc. (2020a), Paket: sysstat (11.0.1-1), https://packages.debian.org/de/jessie/sysstat, letzter Zugriff: 30.07.2020.

PACKAGES.DEBIAN.org (2020b), Paket: nicstat (1.95-1 und andere), https://packages.debian.org/de/sid/nicstat, letzter Zugriff: 30.07.2020.

PACKAGES.DEBIAN.org SPI Inc. (2020c), Paket: iftop (1.0~pre4-7 und andere), https://packages.debian.org/de/sid/iftop, letzter Zugriff: 30.07.2020.

WIKI.UBUNTUUSERS.de (2020a), wget, https://wiki.ubuntuusers.de/wget/, letzter Zugriff: 01.08.2020.

WIKI.UBUNTUUSERS.de (2020b), Benchmarks, https://wiki.ubuntuusers.de/Benchmarks/, letzter Zugriff: 01.08.2020.

WIKI.UBUNTUUSERS.de (2020c), Apache 2.4, https://wiki.ubuntuusers.de/Apache_2.4/, letzter Zugriff: 01.08.2020.

GITHUB.com (2020e), Quickstart with firecracker-containerd, https://github.com/firecracker-microvm/firecracker-containerd/blob/master/docs/quickstart.md, letzter Zugriff: 01.08.2020.

GITHUB.com (2020f), Getting started with Firecracker and containerd, https://github.com/firecracker-microvm/firecracker-containerd/blob/master/docs/getting-started.md, letzter Zugriff: 01.08.2020.

GITHUB.com (2020g), Getting Started with Firecracker, https://github.com/firecracker-microvm/firecracker/blob/master/docs/getting-started.md, letzter Zugriff: 01.08.2020.

DATACENTER-INSIDER.de (2019), Was ist ein Pod in der IT?, 02. Juli, https://www.datacenter-insider.de/was-ist-ein-pod-in-der-it-a-841195/#:~:text=Kubernetes%3A%20Der%20Software%2DPod&text=Hier%20ist%20ein%20Pod%20laut,sich%20Storage%20und%20Netzwerk%20teilen%E2%80%9C.&text=Die%20Container%20eines%20Kubernetes%2DPods,Umgebung%20und%20unter%20gemeinsamer%20Orchestrierung., letzter Zugriff: 01.08.2020.

SYSTUTORIALS.com, ctr (1) – Linux Man Pages,

https://www.systutorials.com/docs/linux/man/1-ctr/, letzter Zugriff 03.08.2020.

GITHUB.com (2020h), documentation, https://github.com/kata-containers/documentation/blob/master/Developer-Guide.md, letzter Zugriff: 03.08.2020.

UNIX.STACKXCHANGE.com (2019), docker.service - How to edit systemd service file?,

https://unix.stackexchange.com/questions/542343/docker-service-how-to-edit-systemd-service-file, letzter Zugriff 01.08.2020.

Glossar

Sandbox Sandbox ist eine Isolierte Umgebungsbereich, die von anderen Bereichen abgeschottet ist. Dadurch lassen sich beispielsweise Konflikte zwischen Betriebssysteme und *Software* durch die isolierte Umgebung vermeiden (vgl. Vogel 2018).

Runc *Runc* ist ein *CLI Tool* mit denen man Container *Spawnen* ausführen kann (vgl. Github 2020b).

Runtime Der Container *Runtime* ist eine Software, die für das Ausführen und Verwalteneines Containers auf einem Knoten zuständig ist (Lou und Brown 2017).

VSock Der *Vsock* ist für die Erleichterung der Kommunikation zwischen einer virtuellen Maschine und Host-Betriebssystem zuständig (vgl. Man7 2020).

Storage Der *Storage* ist eine Art Speicherlösung, um Daten temporär bzw. dauerhaft aufzubewahren (vgl. Floyd und Dr. Bergler 2017).

UnionsFS Mithilfe von *UnionFS* lassen sich für Schnittstellen mehrere gestapelte Dateisysteme bereitstellen (vgl. Linux Magazin 2005).

Agnostisch „Unter Agnostisch versteht man, dass etwas soweit verallgemeinert wurde, dass es auch unter verschiedene Umgebungen eingesetzt werden kann“ (Rouse M. 2020).

QEMU

POD „eine Gruppe von einem oder mehr Containern, die sich Storage und Netzwerk teilen“ (Datacenter Insider 2019).

PID

IPC

GPG

Sysstat	Mit dem *Tool* sysstat können verschiedene Systemleistungen überwacht werden (vgl. Packages Debian SPI Inc. 2020a).
Nicstat	Das Netzwerk kann durch den Befehl nicstat überwacht werden (vgl. Packages Debian SPI Inc. 2020b).
Iftop	Mit dem Tool iftop kann das Netzwerkverkehr überwacht werden, um Schluss zu folgern wie die aktuelle Bandbreitnutzung erfolgt (vgl. Packages Debian SPI Inc. 2020c).
Wget	Mit wget ist es innerhalb eines Terminals möglich HTTP oder HTTPS Server Dateien Herunterzuladen (vgl. Wiki Ubuntuusers 2020a).

Anhang A 1. Thinkpool erstellen

```
#!/bin/bash
set -ex

DATA_DIR=/var/lib/firecracker-containerd/snapshotter/devmapper
POOL_NAME=fc-dev-thinpool

mkdir -p ${DATA_DIR}

# Create data file
sudo touch "${DATA_DIR}/data"
sudo truncate -s 100G "${DATA_DIR}/data"

# Create metadata file
sudo touch "${DATA_DIR}/meta"
sudo truncate -s 10G "${DATA_DIR}/metadata"

# Allocate loop devices
DATA_DEV=$(sudo losetup --find --show "${DATA_DIR}/data")
META_DEV=$(sudo losetup --find --show "${DATA_DIR}/metadata")

# Define thin-pool parameters.
# See https://www.kernel.org/doc/Documentation/device-mapper/thin-provisio-
ning.txt for details.
SECTOR_SIZE=512
DATA_SIZE="$(sudo blockdev --getsize64 -q ${DATA_DEV})"
LENGTH_IN_SECTORS=$(echo $((${DATA_SIZE}/${SECTOR_SIZE})))
DATA_BLOCK_SIZE=128
LOW_WATER_MARK=32768

THINP_TABLE="0 ${LENGTH_IN_SECTORS} thin-pool ${META_DEV} ${DATA_DEV}
${DATA_BLOCK_SIZE} ${LOW_WATER_MARK} 1 skip_block_zeroing"
echo "${THINP_TABLE}"

if ! $(dmsetup reload "${POOL_NAME}" --table "${THINP_TABLE}"); then
dmsetup create "${POOL_NAME}" --table "${THINP_TABLE}"
fi
```

Anhang A 2. Firecracker ohne Containerd installieren

```
#!/bin/bash
#Firecracker installieren
sudo setfacl -m u:${USER}:rw /dev/kvm
curl -Lo firecracker https://github.com/firecracker-microvm/firecracker/relea-
ses/download/v0.16.0/firecracker-v0.16.0

chmod +x firecracker

sudo mv firecracker /usr/local/bin/firecracker

# Rootfs Kernel für Firecracker installieren
curl -fsSL -o hello-vmlinux.bin https://s3.amazo-
naws.com/spec.ccfc.min/img/hello/kernel/hello-vmlinux.bin

curl -fsSL -o hello-rootfs.ext4 https://s3.amazo-
naws.com/spec.ccfc.min/img/hello/fsfiles/hello-rootfs.ext4

#Netzwerkverbindung konfigurieren
#!/bin/bash
sudo ip tuntap add tap0 mode tap
sudo ip addr add 172.20.0.1/24 dev tap0
sudo ip link set tap0 up

DEVICE_NAME=enp0s3

sudo sh -c "echo 1 > /proc/sys/net/ipv4/ip_forward"
sudo iptables -t nat -A POSTROUTING -o enp0s3 -j MASQUERADE
sudo iptables -A FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A FORWARD -i tap0 -o enp0s3 -j ACCEPT
sudo ip addr add 172.20.0.1/24 dev tap0

# Firecracker in Terminal 1 starten
#!/bin/bash

rm -f /tmp/firecracker.socket

firecracker \
        --api-sock /tmp/firecracker.socket \


#Firecracker in Terminal 2
#!/bin/bash

curl --unix-socket /tmp/firecracker.socket -i \
-X PUT 'http://localhost/boot-source' \
-H 'Accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
"kernel_image_path": "./hello-vmlinux.bin",
"boot_args": "console=ttyS0 reboot=k panic=1 pci=off"
}'

curl --unix-socket /tmp/firecracker.socket -i \
-X PUT 'http://localhost/drives/rootfs' \
-H 'Accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
"drive_id": "rootfs",
"path_on_host": "./hello-rootfs.ext4",
"is_root_device": true,
"is_read_only": false
```

```
}'
curl --unix-socket /tmp/firecracker.socket -i \
-X PUT 'http://localhost/machine-config' \
-H 'Accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
"vcpu_count": 1,
"mem_size_mib": 512
}'

curl --unix-socket /tmp/firecracker.socket -i \
  -X PUT 'http://localhost/network-interfaces/eth0' \
  -H 'Accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
      "iface_id": "eth0",
      "guest_mac": "a2:96:04:dc:75:a1",
      "host_dev_name": "tap0"
    }'

curl --unix-socket /tmp/firecracker.socket -i \
-X PUT 'http://localhost/actions' \
-H 'Accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
"action_type": "InstanceStart"
}'

#Im Gast System MAC-Adresse erfassen
#!/bin/bash
ifconfig eth0 up && ip addr add dev eth0 172.20.0.2/24
ip route add default via 172.20.0.1 && echo "nameserver 8.8.8.8" > /etc/resolv.conf

#Internet Verbindung testen
ping google.de
```

Anhang A 3. Wichtige Tools für die Installation von Kata Container

```
#Wichtige Tools für die Installation von Kata Container
#!/bin/bash

sudo apt-get update && pte-get upgrade
sudo apt-get install gcc
sudo apt install curl
sudo apt-get install flex
sudo apt-get install -y bison
sudo apt-get install libelf-dev
sudo apt-get install -y python3-simpleeval
sudo apt-get install python3
sudo apt-get install -y pkg-config
sudo apt-get install libglib2.0-dev
sudo apt-get install libpixman-1-dev
```