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 Krise 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 dieser Stelle noch bei mein Bruder Grewan und meine Freundin Irem für die Korrektur Lesung herzlich bedanken.

Hochschule Worms
University of Applied Sciences

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Programmcodeverzeichnis

Abkürzungsverzeichnis

VM	Virtual Machine
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 Machine 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 werden verschiedene Hardwarekomponenten der 3 Containerlösungen auf ihre Leistungsfähigkeiten überprüft und in Grafiken dargestellt und bewertet.

In Kapitel 5 erfolgt die Zusammenfassung und schlussendlich im letzten Kapitel 6 der Ausblick auf die Zukunft.

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).

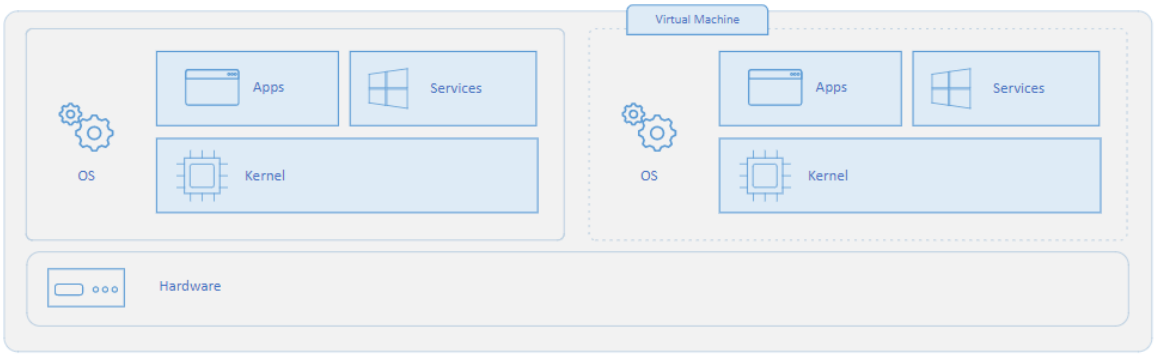
Damit ist ein Container im Prinzip nichts anderes als eine Software, die Code und alle seine Abhängigkeiten zusammenbringt, damit die Anwendung schneller und zuverlässiger ausgeführt werden kann. Ein Container wird mithilfe einer Image-Datei aufgebaut, diese beinhaltet alle erforderlichen Anwendungen 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).

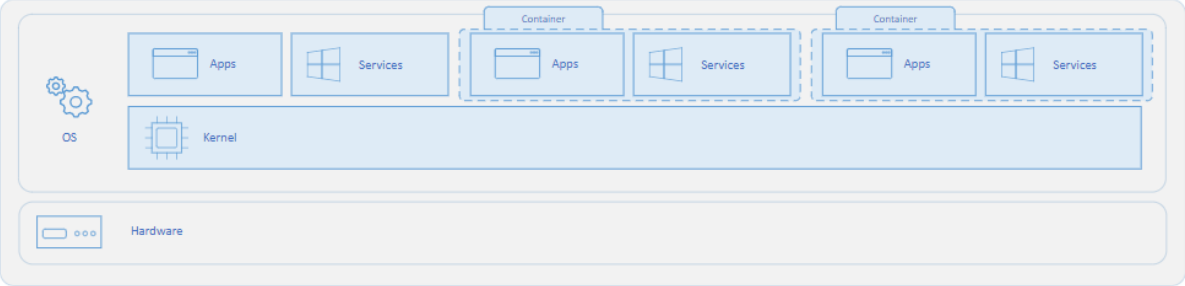


Quelle: (Docs Microsoft Inc. 2020)

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).

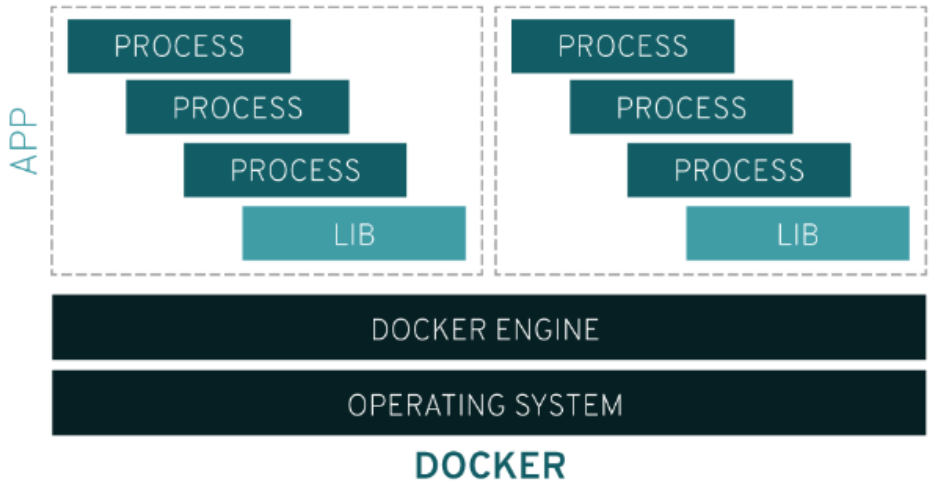


Abbildung 3:Architektur von Docker Container

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

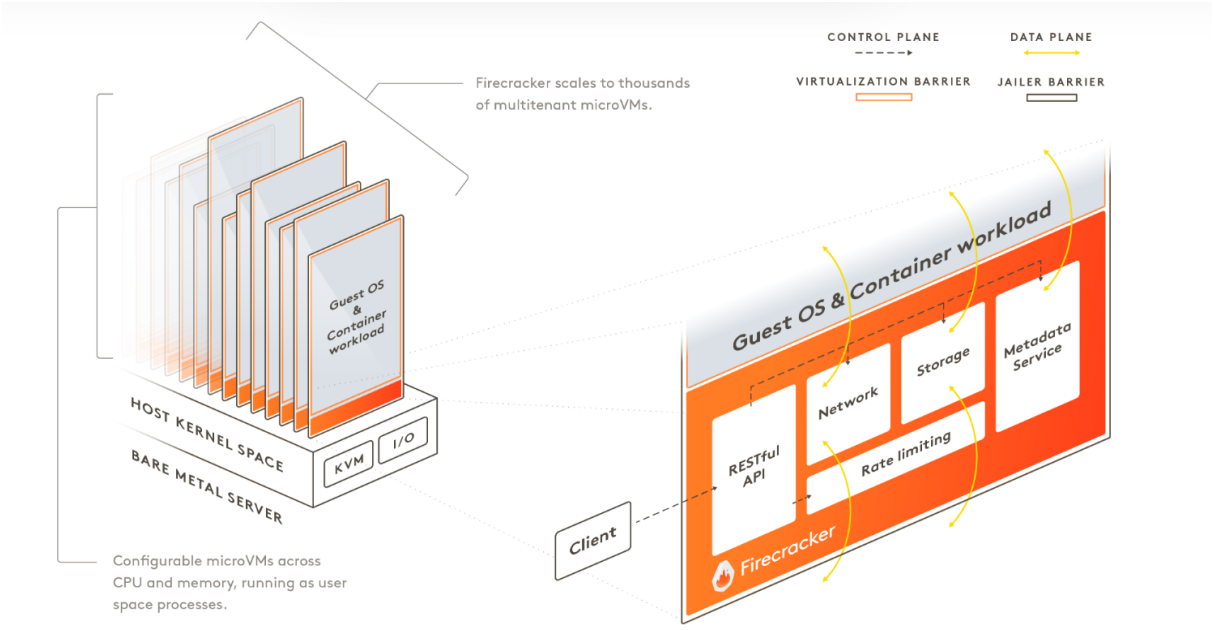


Abbildung 4: Firecracker in Funktionsweise

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).

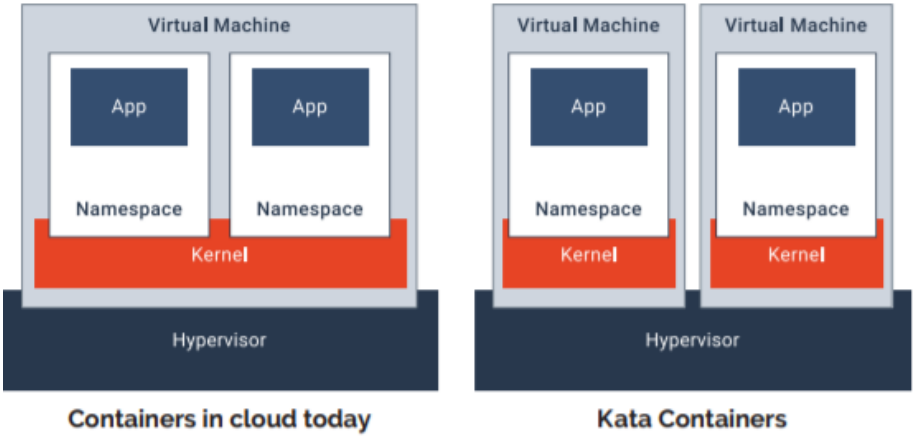


Abbildung 6: Unterschied zwischen Containern in Cloud und Kata Containern

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

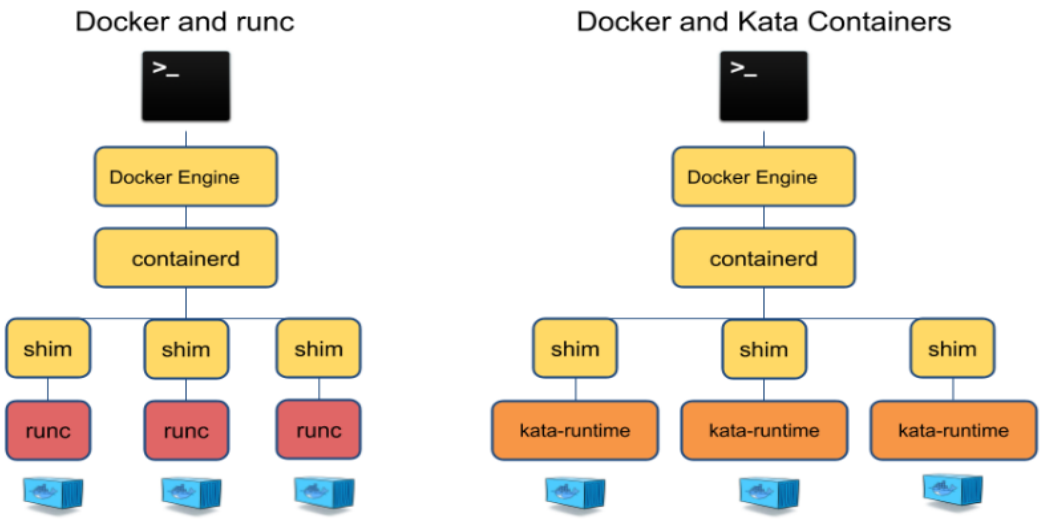


Abbildung 7: Docker und runc im vergleich mit Docker und Kata Containers

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 ist somit auch startbereit, um ein Container zu starten. Im nächsten Schritt wird ein *Dockerfile* erstellt und konfiguriert, 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: Rootfs Image Datei installieren

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 detailliertere Informationen in diesem Kapitel nicht näher beschrieben.

```
$ 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 zu-sehen 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 Code 1: Repository Herunterladen

Zunächst wird mithilfe von *Goolang* das Kata Container *Repository* heruntergeladen. 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 Code 2: 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 Code 3: 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 Code 4: 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 Code 5: 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 Code 6: 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. Es werden beide *Repositories* Herunterladen und mit dem *Makefile* die beiden Funktionen auf jeweils ihre Ordner erstellt.

Nachdem diese beiden Funktionen erstellt wurden sind, müssen wir ein *Rootfs* wie auch bei *Firecracker* installieren, dafür wird zunächst das *Repository Osbuilder* benötigt, mit dieser Datei kann man den *Rootfs* Datei für das *Kata Container* aufbauen. Der *Osbuilder* lässt sich mit dem Befehl `go get -d -u github.com/kata-containers/osbuilder` Herunterladen. Im nächsten Schritt wird ein *Rootfs* Datei installiert.

```
$ export ROOTFS_DIR=${GOPATH}/src/github.com/kata-containers/osbuilder/rootfs-buil-
der/rootfs
$ sudo rm -rf ${ROOTFS_DIR}
$ cd $GOPATH/src/github.com/kata-containers/osbuilder/rootfs-builder
$ export distro=debian
$ script -fec 'sudo -E GOPATH=$GOPATH USE_DOCKER=true SECCOMP=no ./rootfs.sh ${dis-
tro}'
```

Quelle: (Github 2020h)

Kata Container Code 7: Rootfs Image Architektur aufbauen

Bei der Installation der *Rootfs* Datei kann man sich wie auch bei *Firecracker* zwischen einem *Image* oder *Initrd* entscheiden, da bei *Firecracker* eine Image Datei ausgewählt wurde, wird auch im Kata Container 7 Code der *Rootfs Image* installiert. Die *Image* Date wird Mithilfe des Docker *runc* installiert, somit ist die Installation des *Docker Container* für diesen Vorgang essential. Zunächst müssen Grundlegende Vorkehrungen gemacht werden, um die *Rootfs* Datei anschließend installieren zu können. Zunächst muss die *Rootfs* Datei erstellt werden und anschließend dann mit einem *Script* aufgebaut werden. Damit die *Rootfs* Datei erstellt werden kann müssen die ersten Drei Befehle im *Kata Container* Code 7 ausgeführt werden und anschließend wird der *Rootfs* dann mit dem letzten *Script* Befehl aufgebaut.

```
$ commit=$(git log --format=%h -1 HEAD)

$ date=$(date +%Y-%m-%d-%T.%N%z)

$ image="kata-containers-${date}-${commit}"

$ cd /usr/share/kata-containers

$ sudo install -o root -g root -m 0640 -D kata-containers.img "/usr/share/kata-con-
tainers/${image}"

$ (cd /usr/share/kata-containers && sudo ln -sf "$image" kata-containers.img)
```

Quelle: (Github 2020h)

Kata Container Code 8: Rootfs Image datei installieren

Nachdem die Image Datei aufgebaut wurde kann diese mit den Befehlen aus *Kata Container* Code 8 installiert werden.

```
go get -d -u github.com/kata-containers/packaging

cd $GOPATH/src/github.com/kata-containers/packaging/kernel

./build-kernel.sh setup

./build-kernel.sh install
```

Quelle: (Github 2020h)

Kata Container Code 9: Kernel installieren

Nachdem die Image Datei erstellt wurde, wird auch bei Kata Container ein Kernel benötigt, damit der Kata Container auch ausgeführt werden kann. Im Kata Container Code 9 wird zunächst der *Repository packaging* heruntergeladen. Im Ordner *packaging/kernel* kann man mit den letzten zwei Befehlen den *kernel* für das *Kata Containers* installieren.

```
go get -d github.com/qemu/qemu

mv ${GOPATH}/root/go/src/github.com kata-containers

cd ${GOPATH}/src/github.com/kata-containers/qemu/qemu

mkdir build

cd build

../configure

make -j $(nproc)
sudo -E make install
```

Quelle: (Github 2020h und Github 2020i)

Kata Container Code 10: Hypervisor installieren

Der *Hypervisor*, was im Kapitel 2.4.1.1 detailliert erklärt wurde, wird im Kata Container Code 10 installiert. Es wird zunächst der *Repository qemu* heruntergeladen und anschließen mit den letzten 3 Befehlen der Qemu Hypervisor Datei aufgebaut und installiert.

3.3.1 Kata Container starten

```
$ sudo mkdir -p /etc/systemd/system/docker.service.d/
$ cat <<EOF | sudo tee /etc/systemd/system/docker.service.d/kata-containers.conf
[Service]
ExecStart=
ExecStart=/usr/bin/dockerd -D --add-runtime kata-runtime=/usr/bin/kata-runtime --
default-runtime=kata-runtime
EOF
```

Quelle: (Github 2020j)

Kata Container Code 10: Docker Daemon einrichten

Damit der Kata Container über den Docker *Daemon* auch ordnungsgemäß läuft muss der *Kata Runtime* hinzugefügt werden. Im *Kata Container Code 10* wird der *Kata Runtime* mit dem Befehl `--add-runtime` hinzugefügt.

```
{
  "default-runtime": "kata-runtime",
  "runtimes": {
    "kata-runtime": {
      "path": "/usr/bin/kata-runtime"
    }
  }
}
```

Quelle: (Github 2020j)

Kata Container Code 11: Docker daemon.json Datei konfigurieren

Als letzte Schritt müssen die Zeilen im *Kata Container Code 10* in der Datei */etc/docker/daemon.json* hinzugefügt werden. Diese Änderungen finden nur dann statt, wenn der *Docker Daemon* auch mit den beiden Befehlen `systemctl daemon-realod` und *`systemctl restart docker`* aktualisiert wird. Nachdem der Daemon die einrichtung übernommen hat kann man mithilfe eines Docker Images ein Kata Container starten. Wie auch bei *Docker Container* und *Firecracker Containerd* wird auch bei *Kata Container* dieselbe Image was verwendet. Im Kapitel 3.1.1 wird die verwendete *Docker Image* detailliert erläutert.

Jetzt kann man mit dem einfachen Befehl `sudo docker run -ti --runtime kata-runtime debian-test sh` ein *Container* mit dem erstellten *Image Datei* ausführen und der *Kata Container* läuft mithilfe des *Docker Damons*.

4 Leistungsbewertung der Container

In den letzten Kapiteln wurde erläutert, wie die Containerlösungen funktionieren, welche Unterschiede sie haben und wie man diese drei unterschiedlichen Container installiert. Für die Leistungsbewertung wird bei allen drei Containerarten die gleiche Docker Image verwendet, um ein Container auszuführen. Diese Voraussetzung macht es uns möglich die Container in einer Äquivalenten Systemzustand zu bewerten. Für die Leistungsbewertung wird Benchmark als Haupt Werkzeug verwendet, um verschiedene Hardware Komponenten zu testen. Danach werden die einzelnen Startzeiten der Container und Network Perfomance auf ihre Leistungen bewertet. Die Testdurchläufe werden mithilfe von verschiedenen Shell Scripts durchgeführt und anschließend dann mit einer Bibliothek geplottet, die daraus resultierenden Grafiken mit den Ergebnissen analysiert. Anschließend werden diese Ergebnisse miteinander verglichen und ein Gesamtergebnis interpretiert.

Die ersten Beiden Teste beziehen sich auf CPU und RAM Leistung, um diese zu testen wird Benchmark verwendet. Mithilfe von Benchmark kann man die Perfomance von verschiedenen Systemen messen. Es gibt eine sogenannte Software namens *sysbench*, mit dieser Software kann man verschiedene Test durchführen, um Hardware Komponenten wie Beispielweise *CPU*, *RAM* oder *Festplatten* auf ihre Leistungsfähigkeiten zu messen (vgl. Webhosterwissen 2018).

In dieser Arbeit wird die Benchmark Testdurchläufe ein sehr wichtiger Bestandteil der Leistungsbewertung von Containern sein. Mithilfe des Tools werden die CPU und RAM Leistungen der 3 unterschiedlichen Containerarten gemessen und anschließend in einer Grafik miteinander verglichen.

Der dritte Test bezieht sich auf die Leistung der Startzeiten von Containern und der vierte und somit auch der letzte Test wird sich auf die Network Perfomance der Container fokussieren. Nachdem der verschiedene Test jeweils auf den Container durchgeführt wurde, werden diese Ergebnisse eingelesen und anschließend geplottet. Für das Einlesen und Plotten der Ergebnisse wird die Bibliotheken *matplotlib* von *Pyhton* verwendet. Mithilfe dieser Bibliothek kann man die Ergebnisse in Grafiken darstellen und die unterschiedlichen Ergebnisse der Leistungen miteinander vergleichen (siehe. Matplotlib 2020).

4.1 Leistungsbewertung von CPU

Der CPU Test wird bei allen drei Container innerhalb des Containers mit dem gleichen Docker Image durchgeführt. Für jede Containerart wird aus dem Docker Image ein Container gestartet und nacheinander ein CPU und RAM Test durchgeführt. Die Ergebnisse werden dann in einer Datei eingelesen und dann mithilfe des Python Bibliothek *matplotlib* in Grafiken dargestellt.

4.1.1 Primzahlen Test

```
#/bash/bin!
# CPU: Primzahlen-Test

i=1
while [ $i -le 25 ]
do
  sysbench --num-threads=1 --test=cpu --cpu-max-prime=20000 run | head -n15 | tail -
n1 >> CPU_Test_fuer_Docker_Container.csv
  sleep 30s
  i=`expr $i + 1`
done
```

Quelle: (Webhosterwissen 2018)

CPU Code 1: Benchmark CPU Shell Script

Im CPU Code 1 *Script* wird ein `sysbench` Test durchgeführt, dabei wird die Anzahl der *Cores* mit der Option `--num-threads=` festgelegt. Da in allen Container nur ein *Core* verwendet wird, kann man auch maximal nur ein Core festlegen. Der `--test=` legt fest welche Art von Test sein soll, in diesem Fall handelt es sich um ein CPU Test. Mit der Option `--cpu-max-prime=` legt man die Art des Testes fest, in diesem Fall handelt es sich um ein Primzahlen Test und der maximale Wert wurde auf 20.000 festgelegt. Dies bedeutet das der Container die Primzahlen zwischen 1 und 20.000 sucht und als Ergebnis wird in Zeile 15 ein *totale: time* Wert ausgegeben, dieser Wert sagt aus, wie lange der Container benötigt hat, um die Primzahlen zwischen 1 und 20.000 zu suchen. Der Wert 20.000 entspricht die Ausführung von insgesamt 321.238 Rechenoperationen, d.h. es werden beim Suchen der 20.000 Primzahlen 32.238 Rechenoperationen auf die CPU ausgeführt (vgl. Webhosterwissen 2018). Dieser Vorgang wird mithilfe der while Schleife 25-mal wiederholt und nach jedem durchlauf mit der `sleep` Befehl 30 Sekunden. Dadurch kann sich jeder der drei Container kurz erholen und genauere Ergebnisse liefern. Nach jedem Testdurchlauf wird das Ergebnis in ein CSV Datei hinterlegt, sodass man später diese Ergebnisse mithilfe von Python einlesen und plotten kann. Am Anhang A 4 kann man die Resultate der Primzahlen Test für Docker Container, Firecracker Containerd und Kata Container einsehen.

4.1.2 Ergebnisse von CPU Test

In den durchgeführten Testdurchläufen wurden wie schon erklärt die Sekunden ausgegeben, wie lange ein Container benötigt, um Primzahlen zwischen 1 und 20.000 zu suchen. Die CSV Datei beinhaltet die Ergebnisse der Testdurchläufe aber eingelesen kann diese von Python noch nicht, weil die Syntax des CSV Datei noch nicht korrekt ist. Die Korrekte Syntax der CSV Datei ist am Anhang A 5 zusehen. Nachdem die CSV Datei korrigiert wurde, kann diese wie am Anhang A 6 zusehen ist mit der Funktion `open` geöffnet werden. Danach werden die Zeilen der CSV Datei gelesen und anschließend mit der Funktion `plt.plot` geplottet. Als Ausgabe werden die Ergebnisse geplottet und in einer Grafik dargestellt.

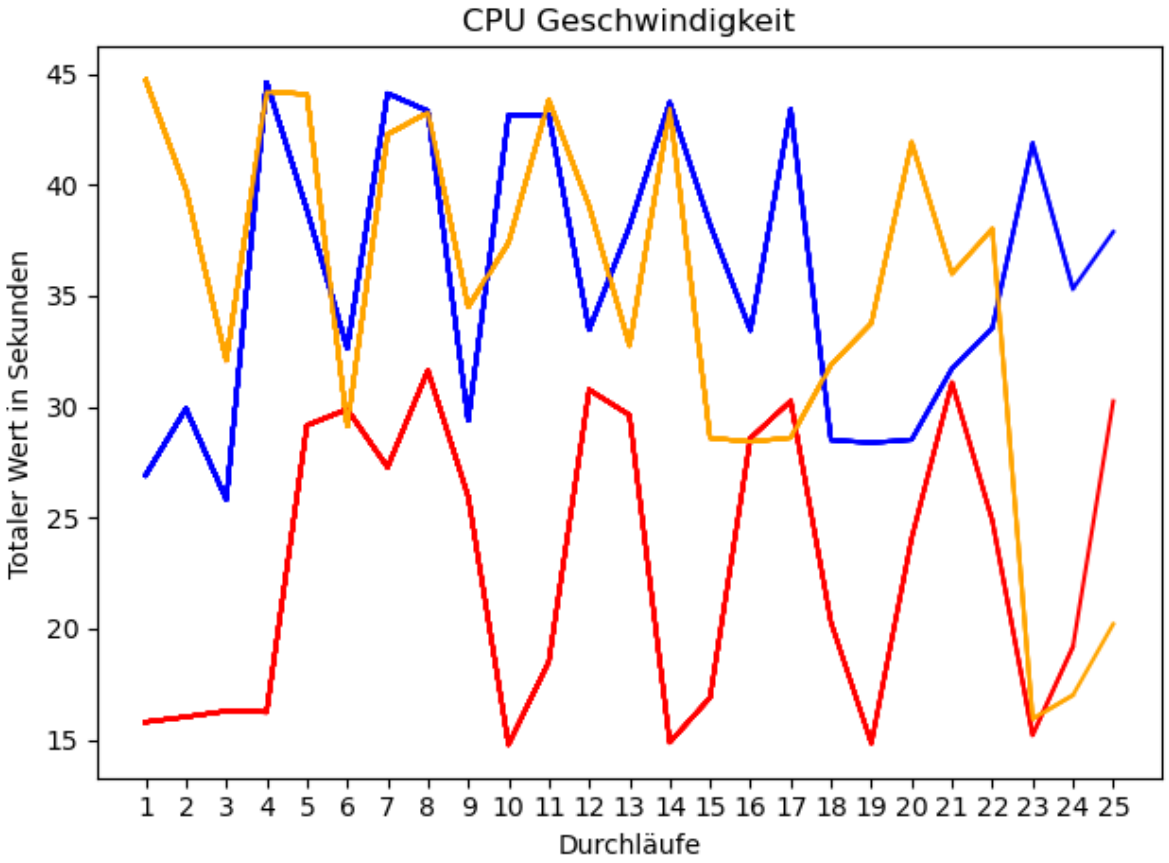


Abbildung 5: Totale Wert Ergebnisse

Abbildung 5 zeigt die Ergebnisse von Anhang A4. Diese Werte wurden Mithilfe der CSV Datei Am Anhang A5 mit Python und die Bibliothek *matplotlib* eingelesen und geplottet. Auf der x-Achse wird die Anzahl der Durchläufe aufgezeigt und auf der y-Achse der Totaler Wert in Sekunden Einheit. Es sind insgesamt drei Kurven zusehen, der Docker Container ist mit der Blauen Kurve gezeichnet, der Firecracker Containerd mit Rot und mit Orange der Kata Container.

Der Docker Container hat an der Stelle x=1 ~26,94s benötigt, der *Firecracker Containerd* ~15,80s und der Kata Container ~44,76s. Somit war der Docker Container, um ~11,14s langsamer als Firecracker Containerd aber um ~17,82s schneller als der Kata Container.

Der Tiefste Punkt erreicht der Docker Container bei x=3, dort beträgt der ~Wert 25,78s. Ab diesem Durchlauf kann man ganz genau erkennen das der Kata Container und Docker Container zwischen x=4 und x=14 die Werte sich überschneiden. Der Docker Container und Kata Container bewegen sich zwischen diesem Intervall immer zwischen ~44,40s und ~29,25s. Eine Starke Veränderung passiert bei Docker Container erst ab x=16 bis x=17, da steigt die Kurve von ~33,26s auf ~42.06s. Beim Kata Container bleibt der Wert Konstant bei ~28.36s. Einer der Tiefsten Stellen von Docker Container wird bei x=18 erreicht. Da fällt der Wert um ~28.45s, während der Kata Container Wert ab x=17 langsam aufsteigt. Die Kurve von Kata Container steigt von x=17 bis x=20 auf ~41.46s während der Docker Container auf einem Konstanten Wert von ~28.45s sich bewegt. Der größte Tiefpunkt erreicht der Kata Container ab x=23. Da fällt die Kurve von x=22 ~37,99s auf x=23 ~15,98s. Bei Docker Containert steigert sich der Wert von x=20 ~28,63s auf ~41.37s. Zum Schluss steigert sich der Wert von Docker Container von x=24 bis x=25 auf ~34,51s und beim Kata Container von x=23 bis x=25 auf ~20.07s.

Der Firecracker Containerd überschneidet im Gegensatz zu Docker Container und Kata Container kaum. Die Einzige Überschneidung findet ab dem Punkt x=6, x=16, x=17 und x=23 mit dem Kata Container. Eine Überschneidung von Docker Container und Firecracker Container ist nicht zusehen. Die Kurve bleibt von x=1 bis x=4 konstant auf ~16,00. Danach steigert sich der Wert von ~16,42s auf ~29,07s. Der Wert bleibt bis x=7 auf ~29,43s konstant und danach steigert sich der Wert ab x=8 auf ~31.48s Sekunden. Anschließend fällt die Kurve ab x=8 bis x=10 auf ~15,08s. Danach steigert sich wieder der Wert ab x=10 bis x=12 auf 30,59s. Danach stürzt wieder ab x=10 bis x=14 die Kurve auf 15,17s. Dieser Vorgang zieht sich kontinuierlich von x=10 bis x=25. Der tiefste Punkt der Kurve liegt bei ~14,91s und der höchste Wert bei ~30.77s.

Zusammenfassend kann man die folgenden Aussagen feststellen:

Der Docker Container kann die Primzahlen zwischen 1 und 20.000 am schnellsten in ~25,78s ausführen und am langsamsten innerhalb von ~44.49s. Die Werte von Docker Container Kata Container überschneiden sich in vielen Punkte wie man bei Abbildung 5 sehen kann, aber nicht ein einziges Mal überschneiden sich die Werte zwischen dem Docker Container und Firecracker

containert. Das zeigt nur wie weit sich Firecracker Containerd und Docker Container von der CPU-Leistung stark unterscheiden.

Der Firecracker Containerd kann die Primzahlen zwischen 1 und 20.000 am schnellsten in ~14,91s ausführen und am langsamsten innerhalb von ~30.77s. Die Kurve von Firecracker Containerd überschneidet sich wie schon erklärt mit keinem Punkt von Docker Containert aber in ein paar Punkte überschneidet sich diese mit dem Kata Container. Die Kurve von Firecracker hat wie auch auf der Abbildung sehr oft tiefe, aber auch hohe Kurven, zwischen einem Intervall zwischen 2-3 steigt die Kurve ganz nach oben und danach fällt sie wieder. Dieser Prozess passiert kontinuierlich ab x=10. Sehr überraschend sind ist, dass nicht mal Ansatzweiße Überschneidungen stattfinden. Dies bedeutet das der Firecracker Containerd wirklich in eine ganz andere Leistung als die andere beiden konkurriert. Was zumindest die CPU-Leistung betrifft.

Der Kata Container kann die Primzahlen zwischen 1 und 20.000 am schnellsten in ~ ~29.25s ausführen und am langsamsten innerhalb von 44,76s. Der Kata Container überschneidet sich in einen Punkt wie auch erwähnt mit dem Firecracker Containerd, aber viel mehr Überschneidungen finden mit dem Docker Container statt. Die Kurven überschneiden sich von x=3 bis x=22 mit dem Docker Container, d.h. die haben eine sehr ähnliche Kurven Struktur und unterschieden sich im Prinzip sehr minimal voneinander. Der Einzige unterschieden zwischen den beiden ist der schnellste und langsamste Wert. Kata Container hat ein sehr schlechter Start aber dafür verbessert Erst am Ende und überschneidet sich sogar mit dem Firecracker Containerd. Es kann sowohl schneller als auch langsamer als Docker Container sein und fast genau so schnell wie auch Firecracker Containerd. Der Kata Container ist von den drei Container was die Leistung zumindest angeht im Interessantesten. Es kann mit Docker Container mithalten mit Sicherheit mithalten und mit Firecracker Containerd gibt es wirklich Punkte, die auch eine eindeutige Überschneidung äußern. Es hat am Punkt x=23 fast die maximale Geschwindigkeit wie Firecracker Containerd erreicht.

Mit der Folgenden Formal soll der Mittelwert, der der Ergebnisse von Anhang A4 berechnet werden, dadurch kann man sehen, wie viel Zeit ein Container durchschnittlich für die suche von Primzahlen zwischen 1 und 20.000 benötigt.

$$\frac{1}{n}\sum_{i=1}^{n} x_i$$

Formel 1: Mittelwert von Primzahlen-Test

Die variable *n* ist der Wert 25, weil es insgesamt 25 Testdurchläufe ausgeführt und der Totale Wert entspricht *x*, das sind die Ergebnisse am Anhang A4. Diese Formel muss man jeweilig für die Ergebnisse von Docker Container, Firecracker Containerd und Kata Container anwenden, somit bekommt man für den Primzahlen-Test folgende Ergebnisse:

- Docker Container benötigt durchschnittlich 35.92 Sekunden, um die Primzahlen zwischen 1 und 20.000 zu suchen.
- Firecracker Containerd benötigt durchschnittlich 22.90, um die Primzahlen zwischen 1 und 20.000 zu suchen.
- Kata Container benötigt durchschnittlich, 34.83 Sekunden, um die Primzahlen zwischen 1 und 20.000 zu suchen.

Die CPU von Firecracker ist somit um 13,02 Sekunden schneller als der Docker Container und der Kata Container ist auch knapp mit 1,09 Sekunden schneller als der Docker Container. Somit ist die Leistung der CPU bei Firecracker mit großem Abstand besser als die von Docker Container oder Kata Container.

Was noch besonders interessant ist sind die Rechenoperationen pro Sekunde, die jeweils ein Container berechnen kann. Die Rechenoperationen können mithilfe der Ergebnisse von Anhang A4 berechnet werden.

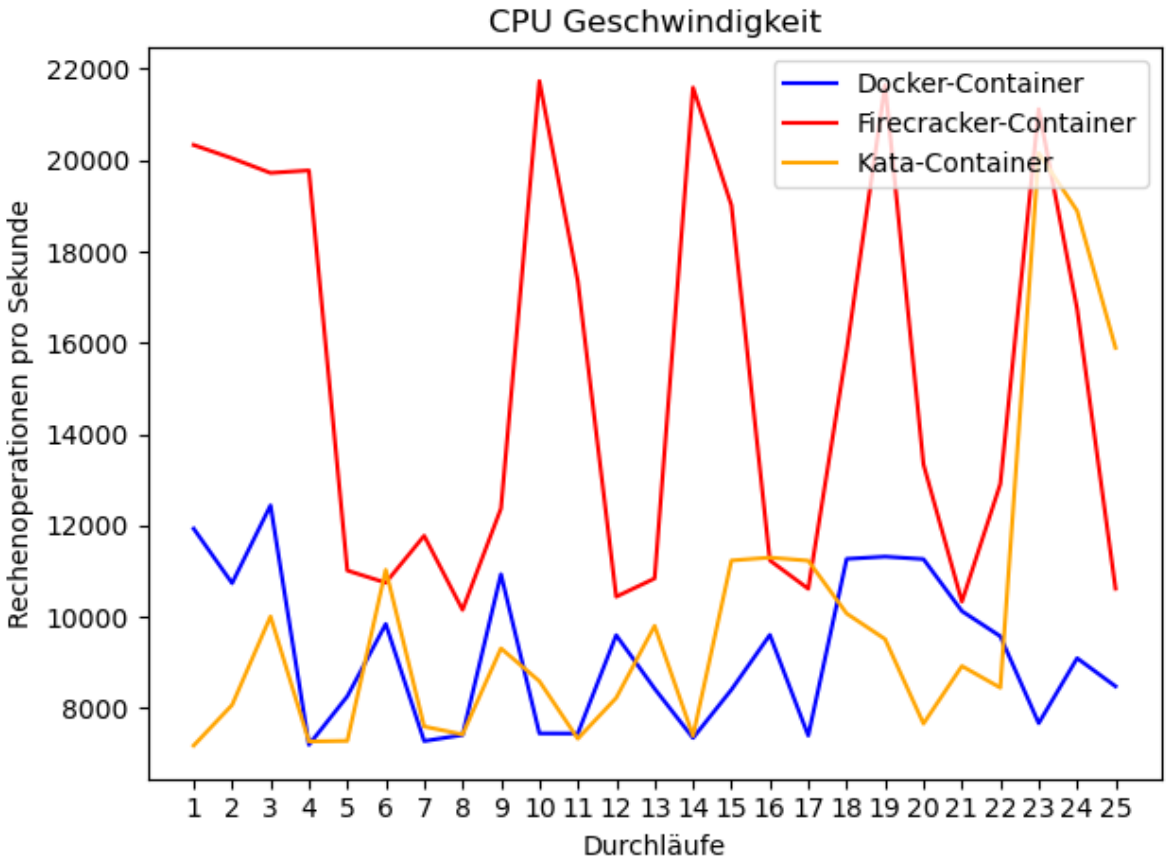


Abbildung 6: Rechenoperationen pro Sekunde Ergebnisse

Die Abbildung 6 zeigt die Ergebnisse der Rechenoperationen pro Sekunde. Diese Abbildung kann man genauso gleich Interpretieren wie die Abbildung 5 im Kapitel 4.2.1, die Kurven der beiden Abbildungen sind gleich zu interpretieren. Der einzige Unterschied liegt daran das mit den Daten am Anhang A5 nochmal eine Berechnung stattgefunden hat, um die Rechenoperationen pro Sekunde zu berechnen, die Berechnungen kann man Am Anhang A6 nachvollziehen. Der einzige Unterschied zwischen Abbildung 5 und 6 ist, dass die y=Achse nicht mehr der Totale Wert ist, sondern die Rechenoperationen pro Sekunde. Die Rechenoperationen werden mithilfe des Totalen Wertes berechnet und als Ergebnis entstand die Abbildung 5. Da die Kurven von Abbildung 5 und 6 gleich sind, und diese gleich zu interpretieren sind, wird die Interpretation weggelassen. Die Abbildung dient nur dazu, damit man den Leser vermittelt, dass die Rechenoperationen nicht durchgeführt sind, um nochmal eine CPU Test durchzuführen, sondern aus den Resultierenden Ergebnisse von Abbildung 5 die Rechenoperationen zu ermitteln. Die Berechnung der Rechenoperationen ändern nicht die CPU-Ergebnisse von Anhang A5 wie auch bei beide Abbildungen zusehen, sondern geben uns nur Auskunft wie viele Rechenoperationen pro Sekunde innerhalb der Ausführung des Testes

im Kapitel 4.1.1durchgeführt wurde. Die Ergebnisse zeigen die Rechenoperationen pro Sekunde für jeden Durchlauf, Interessant wäre, wenn man den durchschnittlichen Wert für jede Containerart herausberechnet. Wie diese zu berechnen sind, wird an der folgenden Formel nochmal näher erklärt.

$$\frac{\sum_{i=1}^{n} \frac{321.238}{x_i}}{25}$$

Formel 2: Rechenoperationen pro Sekunde

Für die variable n wird der Wert 25 gesetzt, weil der Testdurchlauf 25-mal durchgeführt wird und für x werden die Werte am Anhang A4 ersetzt. Sowohl für den Docker Container, Firecracker Containerd und Kata Container wird der Mittelwert wie auch am Anhang A6 zusehen separat durchgeführt. Mithilfe dieser Formel wird die Anzahl der Rechenoperationen indem fall 321.238 durch jeden Wert am Anhang A4 dividiert.

Beispielweise würde man die Rechenoperationen beim ersten Durchlauf folgendermaßen ausrechnen:

$$\frac{321.238}{26{,}9217} = 11.6\ Rechenoprationen\ pro\ Sekunde$$

Wir haben im Zähler den Basiswert 321.238 und im Nenner den Totalen Wert 26,9217 Sekunden wie auch am Anhang A4 für Docker Container zusehen. Das Ergebnis dieser beiden Werte sagt aus, wie viele Rechenoperationen ein Container pro Sekunden ausrechnen.

Diese Berechnung wird in der Formel 2 für jede Einzelne wird am Anhang A4 berechnet und anschließend durch die gesamt Durchläufe 25 dividiert, somit bekommt man den Mittelwert der Rechenoperationen pro Sekunde für alle drei Containerarten.

Beispielweise würde die Formale 2 für die Ergebnisse am Anhang A4 Docker Container folgendermaßen aussehen:

$$\frac{\sum_{i=1}^{n} \frac{321.238}{x_i}}{25} = \frac{\frac{321.238}{26{,}9217} + \frac{321.238}{29{,}9154} + \frac{321.238}{25{,}8181} \ldots + \frac{321.238}{x_n}}{25}$$

Diese Formel muss man jeweilig für die Ergebnisse von Docker Container, Firecracker Containerd und Kata Container anwenden, somit bekommt man für den Primzahlen-Test folgende Ergebnisse:

- Der Docker Container (Blau) kann durchschnittlich 9221.32 Rechenoperationen pro Sekunde ausführen.
- Der Firecracker Containerd (Rot) kann durchschnittlich 15252.32 Rechenoperationen pro Sekunde ausführen.
- Der Kata Container (Orange) kann durchschnittlich 9994.63 Rechenoperationen pro Sekunde ausführen.

Die CPU von Firecracker ist somit mit einer Rechenoperation von 15252.32 mit weitem, Leistungsfähiger als Docker Container oder Kata Container. Der Kata Container kann 773,31 Rechenoperationen pro Sekunde mehr ausführen als der Docker Container. Somit hat Docker Container schlechter als Firecracker und Kata Container abgeschnitten.

4.2 Leistungsbewertung von RAM

Die Leistungsbewertung von RAM wird auch wie bei dem Testverlauf von CPU innerhalb des Containers ausgeführt und als Tool wird auch hier von Benchmark das sysbench benutzt, um den RAM zu testen. Beim Ram Test geht darum, wie schnell ein Container MB pro Sekunde im Hauptspeicher abspeichern kann (vgl. Webhosterwissen 2018).

4.2.1 RAM Test

```
#/bash/bin!
# CPU: Primzahlen-Test

i=1
while [ $i -le 25 ]
do
 sysbench --num-threads=1 --test=memory --memory-block-size=1M --memory-total-size=100G run | head -n18 | tail -n1 >>
RAM_Test_fuer_Docker_Container.csv
 sleep 30s
 i=`expr $i + 1`
done
```

Quelle: (Webhosterwissen 2018)

RAM Code 1: Benchmark RAM Shell Script

Im CPU Code 1 *Script* wird ein `sysbench` wie auch beim Primzahlen Test im Kapitel 4.1.1 durchgeführt und Optionen wie die Anzahl der *Cores* mit der Option `--num-threads=` festgelegt. Da in allen Container nur ein *Core* verwendet wird, kann man auch maximal nur ein Core festlegen. Beim Primzahlen Test wurde für die Option `--test=` die CPU festgelegt, beim Rest Test wir nicht die CPU festgelegt, sondern memory wie auch im RAM Code 1 zu

sehen ist. Beim Memory Test werden GB in Blocken von je MB in dem RAM geschrieben, dadurch wird gezeigt, wie viel MB pro Sekunde in dem RAM geschrieben wird. Den MB Wert legt man wie oben schon im Code gesehen mit der Option --memory-block-size=. In dem Fall wurde der MB wert 1 festgelegt. Der GB wird mithilfe der Option --memory-total-size=. Hier wurde der Wert 100G festgelegt, d.h. im Test werden 100 GB in Blocken von je 1MB in dem Ram geschrieben. Die Durchläufe der Testdurchläufe mithilfe der while und sleep Option bezieht sich auch wie der CPU Test im Kapitel 4.1.1 auf 25 Durchläufe und 30 Sekunden Pause nach jedem Durchlauf. Die Ergebnisse werden wie am Anhang A7 zusehen in die CSV Datei hineingeschrieben, sodass wir diese Daten für das einlesen und plotten der Ergebnisse verwendet können. Diese müssen aber natürlich wie auch beim CPU Test in die richtige Syntax umgeschrieben werden, wie auch Anhang A8 zeigt.

4.2.2 Ergebnisse RAM Test

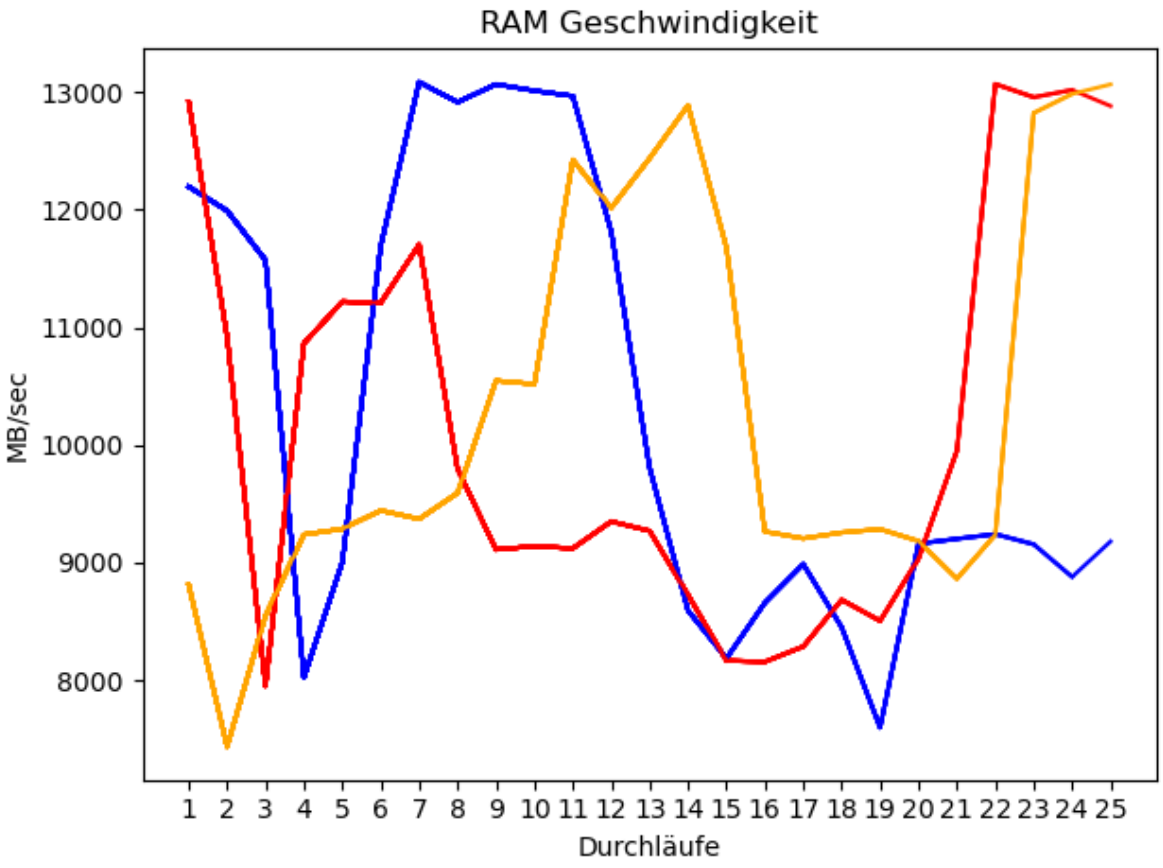


Abbildung 7: MB Pro Sekunde im Hauptspeicher speichern Ergebnisse

In der Abbildung 7 sind die Ergebnisse am Anhang A8 mithilfe des Python Programmes am Anhang A9 eingelesen und danach geplottet wurden. Als Resultat wurde die Grafik in der Abbildung 7 ausgegeben. An der Abbildung sieht man das die Ergebnisse von alle drei Containerarten sich sehr unterscheiden. Der Docker Container ist mit der Blauen Kurve gekennzeichnet, der Firecracker Containerd mit Rot und der Kata Container wie auch im Kapitel 4.1.2 mit Orange.

Im Durchlauf x=1 erreicht der Docker Container ein Wert von ~12.220 MB/sec. Der Firecracker erreicht ein Wert von ~12.940 MB und der Kata Container ~8.810 MB. Der Docker Container schreibt somit ~720 MB/sec langsamer als der Firecracker Container aber ist ~3.410 MB/sec schneller als der Kata Container

Die Überschneidungen der Kurven sind bei allen vorhanden, der Docker Container überschneidet sich sowohl mit Firecracker als auch mit Kata Container. Sowohl bei Firecracker als auch bei Kata Container ist die Überschneidung gegenseitig als auch mit Docker Container vorhanden. Jeder überschneidet sich mit jedem.

Die Kurve von Docker Container fällt von x=1 bis x=4 auf ~8.080 MB/sec. Von x=4 bis auf x=7 erreicht es sein höchster Punkt mit einem Wert von ~13.111 MB/sec. Danach wird der Tiefste Punkt von x=7 bis schlißelich x=19 mit ~7.660 MB/sec erreicht. Danach erhöht sich der Wert am Ende am Punkt x=25 auf ~9.180 MB/sec. Während des Prozesses überschneidet sich der Docker Container sowohl mit dem Firecracker und Kata Container. Mit Firecracker überschneidet sich der Docker Container in den Punkten x=1, x=4, x=6, x=14, x=15, x=18, x=20 und mit dem Kata Container x=4, x=5, x=12, x=20 und x=22.

Im Gegensatz zu Docker Container fällt die Kurve bei Firecracker schon ab x=3 auf ~7.950 MB/sec. Danach steigert sich die Kurve zwischen x=3 und x=7 auf ~1.660 MB/sec. Die Kurve fällt schritt für schritt immer runter bis x=16 ~8.170 MB/sec und danach steigert es sich ganz schnell zwischen x=16 und x=22 auf dem höchsten Wert ~1.3090 MB/sec. Ab x=22 fällt die Kurve ganz bis x=25 auf ~12.890 MB/sec. Wie auch bei Docker Container gibt es hier mehrere Überschneidungen sowohl mit Kata Container als auch mit Docker Container. Mit Kata Container gibt es Überschneidungen an den Punkten x=3, x=8, x=20 und x=24.

Der Kata Container erreicht den tiefsten Punkt bei x=2 ~7.440 MB/sec danach steigt die Kurve von x=2 bis x=14 auf 12.860 MB/sec. Dann fällt sie wieder am Punkt x=21 auf ~8860 MB/sec, und zwischen x=21 und x=25 steigt sie auf ~13.060 MB/sec. Der Kata Container hat wie auch bei Firecracker und Docker Container Überschneidungen mit beiden. Die Überschneidung mit

Firecracker findet an den Punkten x=3, x=8, x=20 und x=24. Der Kata Container überschneidet sich mit dem Docker Container an den Punkten x=4, x=5, x=12, x=20 und x=2.

Zusammenfassend kann man die folgenden Aussagen feststellen:

Der Docker Container kann am schnellsten ~13.110 MB/sec im Hauptspeicher speichern und am langsamsten innerhalb von ~7.610 MB/sec. Die Kurve von Docker Container hat, wie beim CPU nicht viele Punkte, an denen es fällt und wieder steigt. Es gibt insgesamt nur 3 Punkte, wo sich die ganz tief fällt und wieder steigt. Das wären die Punkte bei x=4, x=11 und x=19. Was interessant ist, dass der Docker Container sowohl sehr tiefe stellen aufzeigt wie am Punkt x=19 und x=4 zusehen aber auch hohe Werte beispielweise bei Punkten wie x=7, x=8, x=9, x=10 und x=11 aufweist. Man kann sagen das der Docker Container mal eine sehr gute Leistung aufweist aber sehr schlechte. Es hat nicht so viele Kurvige stellen wie bei Abbildung 5 aber dafür sehr tiefe und hohe Werte.

Der Firecracker Containerd kann am schnellsten ~13.090 MB/sec im Hauptspeicher speichern und am langsamsten innerhalb von ~7.980 MB/sec. Bei Firecracker ändert sich die Kurve minimal. Am Anfang des Durchlaufes x=3 erreicht es ein sehr Tiefe stelle, aber danach gibt es keine Momente, wo die Kurve richtig tief fällt, sie steigt und fällt in minimalen Schritten wie auch von x=3 bis x=15 zusehen. Danach nimmt die Leistung sehr stark zu und die Kurve steigert sich sehr schnell oben. Man kann sagen das die Kurve von Firecracker eindeutiger ist wie die von Docker Container, weil die Werte sich minimal Durchlauf zu Durchlauf ändern. Es gibt nur ein Punkt wo die Kurve sehr stark fällt aber ansonsten steigt oder fällt sie minimal. Am Ende gibt es sogar eine sehr starke Verbesserung und die Leistung von RAM steigt immens hoch.

Der Kata Container kann am schnellsten ~13.090 MB/sec im Hauptspeicher speichern und am langsamsten innerhalb von ~7.430 MB/sec. Kata Container hat bei den drei Container die tiefste stelle am punkt x=2 erreicht, aber danach verbessert sich die Kurve schritt für schritt bis sie ab x=14 wieder fällt und ab x=21 wieder aufsteigt und sich sogar mit der Leistung von Firecracker überschneidet. Die Kurve von fällt im Prinzip nur 2-mal und an den restlichen Punkten ist ganz klar eine Steigerung zu erkennen.

Wie auch bei dem CPU Test, ist hier der Mittelwert Interessant zu betrachten. Für die Berechnung des Mittelwertes wird die Summe mit der Option sum wie auch am Anhang A9 zu sehen ist, verwendet, um jeweils die Ergebnisse von Anhang A8 zu berechnen. Nachdem die

Summe berechnet wurde, wird der durchschnittliche wert, wie auch im Kapitel 4.1.2 mit der der folgende Formel berechnet:

$$\frac{1}{n}\sum_{i=1}^{n} x_i$$

Im Kapitel 4.1.2 wurde der Durchschnitt des Totalen Wertes mithilfe dieser Formel berechnet. Im RAM Test wird die Formel verwendet, um den durchschnittlich MB Wert pro Sekunde zu berechnen. Nachdem der Wert ermitteln wurde, weiß man wie viel MB/sec die jeweiligen Container im Hauptspeicher speichern können.

- Der Docker Container (Blau) kann durchschnittlich 10257,88 MB/sec im Hauptspeicher speichern.
- Der Firecracker Containerd (Rot) kann durchschnittlich 10160,86 MB/sec im Hauptspeicher speichern.
- Der Kata-Container (Orange) kann durchschnittlich 10296,45 MB/sec im Hauptspeicher speichern.

Die Ergebnisse der drei Containerlösungen unterscheiden sich zumindest nach den Daten minimal. Der Docker Container ist durchschnittlich 97,02 MB/sec langsamer als Firecracker und 38,57 MB/sec schneller als der Kata Container. Somit hat Firecracker die besseren Ergebnisse als Docker Container und Kata Container geliefert. Der Docker Container war mit minimal abstand besser als der Kata Container. Somit hat die Besten Ergebnisse der Firecracker geliefert und der Kata Container mit minimalem Abstand die schlechtesten und Docker Container war zwischen Firecracker und Kata Container aber wie schon erwähnt mit minimalem Abstand zu Kata Container.

4.3 Leistungsbewertung von Network-Perfomance

In diesem Test geht es darum die verschiedenen Container zu testen, wie lange sie Brauchen eine Webseite mit all samt der HTTP und CSS-Dateien herunterzuladen.

Für die Durchführung des Testes wird das folgende Bash Skript verwendet.

```
#/bash/bin!


i=1
while [ $i -le 25 ]
do
 wget -nv -r -k -E -l 8 http://192.168.1.109/ -o networktestganz
 cat networktestganz | head -n16 | tail -n1 >> networktestvoll
 rm -rf 192.168.1.109
 sleep 10s
 i=`expr $i + 1`
done
```

Quelle: (Webhosterwissen 2018)

RAM Code 1: Network-Performance Bash Script

Im RAM Code 1 wird mithilfe der while Schleife wie auch bei den anderen Skripts der Test 25-mal durchgeführt und nach jedem Durchlauf findet eine 10 Sekunden Pause statt. Detaillierter wird die while und sleep Option im Kapitel 4.1.1 erklärt. Was hier wichtig zu betrachten ist Befehl wget, cat und rm. Mithilfe von wget kann man von einer Domän eine ganze Webseite Herunterladen. In diesem Skript wird die Domän http://192.168.1.109/ heruntergeladen und die Dateien im Ordner 192.168.1.109 abgespeichert. Wie lange der Download gedauert hat wird in der Datei networktestganz abgespeichert und mit den Befehlen, cat, head und tail die wichtige Zeile in der Datei networktesvoll abgelegt. Aus dieser Datei wird dann die CSS-Datei wie auch am Anhang A10 zusehen ist in die richtige Forma gebracht und danach mit dem Programm am Anhang A11 eingelesen und geplottet. Der Test wird durchgeführt, um zu wissen wie viel MB/sec die jeweiligen Container benötigen, um die Webseite herunterzuladen.

4.3.1 Ergebnisse von Network-Performance Test

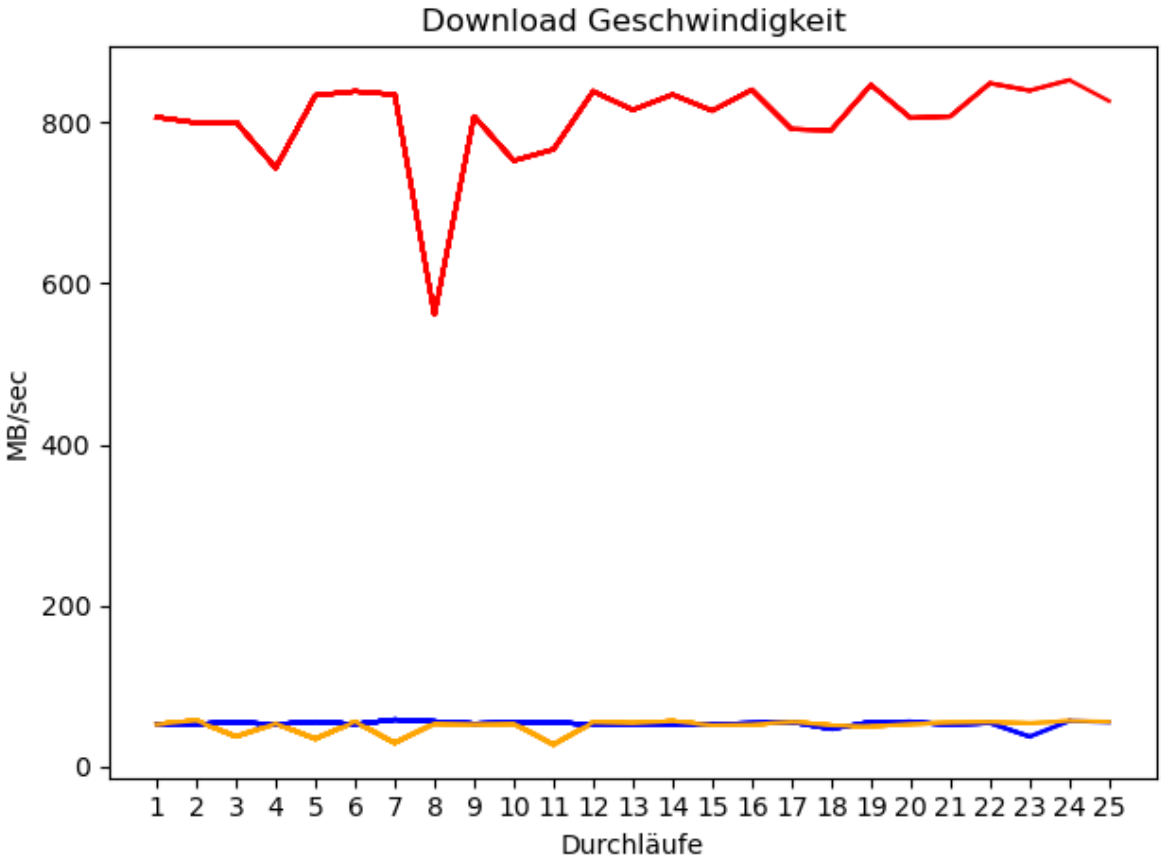


Abbildung 8: Network-Peromance Test Ergebnisse

Die Abbildung 8 zeigt die Ergebnisse von Network-Performance Test. Auf der x-asche sind auch wie die anderen Abbildungen von RAM und CPU Test die Anzahl der Testdurchläufe zusehen. Die y-asche zeigt die Geschwindigkeit von MB/sec, die jeweils die Container benötigt haben, um die Webseite herunterzuladen.
Der Docker Container und Kata Container benötigen im ersten Durchlauf genauso viel wie der Kata Container ~58 MB/sec. Dieser Wert bleibt konstant und ändert sich bis x=22 kaum. Ab x=22 fällt die Kurve minimal auf 32 MB/sec und danach ändert er sein Kurs wieder auf ~58 MB/sec. Eine Überschneidung findet bei Docker Container kontinuierlich von x=1 bis x=25 mit Kata Container statt. Die Überschneidung von Docker Container und Firecracker findet mit großem Abstand nicht statt.

Der Kata Container ähnelt sich wie man in Abbildung 8 sehen kann sehr. Der Start von Kata Container ist wie schon erwähnt wie bei Docker Container bei ~58 MB/sec. Die Kurve bleibt kontinuierlich wie bei Docker Container auf ~58 MB/sec. Ab und zu fällt die Kurve wie bei

x=3 oder x=11 zu sehen ist, aber danach steigt sie direkt wieder auf ~58 MB/sec. Eine Überschneidung findet auch hier nur mit Docker Container statt aber nicht mit Firecracker.

Der Firecracker startet im Gegensatz zu Docker Container und Kata Container bei ~811 MB/sec. Dieser Wert bleibt bis x=3 gleich und danach fällt sie auf ~747 MB/sec und steigt dann ab x=5 auf 828 MB/sec. Danach fällt die Kurve bei x=8 auf ~559 MB/sec. Dieser Wert steigt dann wieder ab x=8 auf 804 MB/sec und bleibt dann mit minimal Änderung bis x=25 konstant. Die Überschneidung zwischen Firecracker, Kata Container und Docker Container findet hier nicht statt.

Zusammenfassend kann man die folgenden Aussagen feststellen:

Der Docker Container und Kata Container sind bei der Geschwindigkeit des Network-Performance sich gegenwürdig und erzielen wie man bei Abbildung 8 sehen kann die fast die gleichen Ergebnisse. Der Höchste wert liegt bei Docker Container und Kata Container bei ~58 MB/sec. Der tiefste Wert erzielt der Kata Container ~29 MB/sec und Docker Container ~34 MB/sec.

Firecracker liefert mit großem Abstand die besten Leistungen was Network-Perfomance angeht. Auf der Abbildung 8 sieht man ganz deutlich das der Firecracker mit großem Abstand eine viel höhere Network-Perfomance anbietet als Docker oder Kata Container.

Der durchschnittlich wurde, wie auch im Kapitel 4.2.2 mit der gleichen Formel berechnet. Die Berechnung ist am Anhang A11 zusehen. Das Programm liefert folgende Ergebnisse:

- Der Docker-Container (Blau) Herunterladet 432k durchschnittlich in 53.55 MB pro Sekunde.
- Der Firecracker-Container (Rot) Herunterladet 432k durchschnittlich in 803.36 MB pro Sekunde.
- Der Kata-Container (Orange) Herunterladet 432k durchschnittlich in 50.78 MB pro Sekunde.

Am Ergebnis ist klar erkennbar, dass der Firecracker mit großem Abstand eine bessere Leistung abgeliefert hat als der Docker oder Kata Container. Der Docker Container ist mit 2,77 MB/sec schneller als der Kata Container und 749,81 langsamer als der Firecracker.

5 Leistungsbewertung von Startzeit

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.

GITHUB.com (2020i), Building, https://github.com/qemu/qemu, letzter Zugriff: 04.08.2020.

GITHUB.com (2020j), Install Docker for Kata Containers on Ubuntu, https://github.com/kata-containers/documentation/blob/master/install/docker/ubuntu-docker-install.md, letzter Zugriff: 04.08.2020.

WEBBHOSTERWISSEN.de (2018), Server-Benchmark mittels sysbench, 17. August, https://www.webhosterwissen.de/know-how/server/server-benchmark/, letzter Zugriff: 05.08.2020.

MATPLOTLIB.org (2018), documentation, https://matplotlib.org/users/index.html, letzter Zugriff: 05.08.2020.

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
```

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
```

Anhang A 4. Ergebnisse von Primzahlen für CPU Test

```
# Benchmark CPU Test: Ergebnisse für Docker Container
    total time:                          26.9217s
    total time:                          29.9154s
    total time:                          25.8181s
    total time:                          44.6201s
    total time:                          38.8803s
    total time:                          32.6317s
    total time:                          44.1203s
    total time:                          43.3279s
    total time:                          29.3796s
    total time:                          43.1258s
    total time:                          43.1427s
    total time:                          33.4642s
    total time:                          38.1140s
    total time:                          43.7146s
    total time:                          38.2142s
    total time:                          33.4431s
    total time:                          43.4092s
    total time:                          28.5042s
    total time:                          28.3797s
    total time:                          28.5179s
    total time:                          31.7246s
    total time:                          33.5470s
    total time:                          41.8722s
    total time:                          35.3179s
    total time:                          37.8857s
```

```
# Benchmark CPU Test: Ergebnisse für Firecracker Containerd
    total time:                          15.7989s
    total time:                          16.0287s
    total time:                          16.2886s
    total time:                          16.2437s
    total time:                          29.1637s
    total time:                          29.8874s
    total time:                          27.2782s
    total time:                          31.6355s
    total time:                          25.9607s
    total time:                          14.7812s
    total time:                          18.5145s
    total time:                          30.7640s
    total time:                          29.6306s
    total time:                          14.8777s
    total time:                          16.8969s
    total time:                          28.5868s
    total time:                          30.2606s
    total time:                          20.2853s
    total time:                          14.8412s
    total time:                          24.0811s
    total time:                          31.0873s
    total time:                          24.8814s
    total time:                          15.2115s
    total time:                          19.1876s
    total time:                          30.2509s
```

```
# Benchmark CPU Test: Ergebnisse für Kata Container
    total time:                          44.7216s
    total time:                          39.7975s
    total time:                          32.0916s
    total time:                          44.1692s
    total time:                          44.0947s
```

```
    total time:                          29.1214s
    total time:                          42.2696s
    total time:                          43.2580s
    total time:                          34.5016s
    total time:                          37.3852s
    total time:                          43.8078s
    total time:                          39.0521s
    total time:                          32.7664s
    total time:                          43.4334s
    total time:                          28.5933s
    total time:                          28.4297s
    total time:                          28.6037s
    total time:                          31.8891s
    total time:                          33.7666s
    total time:                          41.9213s
    total time:                          36.0006s
    total time:                          38.0205s
    total time:                          15.9383s
    total time:                          17.0055s
    total time:                          20.2145s
```

Anhang A 5. CSV Datei für CPU

```
1,totaltime:,44.7216,15.7989,26.9217
2,totaltime:,39.7975,16.0287,29.9154
3,totaltime:,32.0916,16.2886,25.8181
4,totaltime:,44.1692,16.2437,44.6201
5,totaltime:,44.0947,29.1637,38.8803
6,totaltime:,29.1214,29.8874,32.6317
7,totaltime:,42.2696,27.2782,44.1203
8,totaltime:,43.2580,31.6355,43.3279
9,totaltime:,34.5016,25.9607,29.3796
10,totaltime:,37.3852,14.7812,43.1258
11,totaltime:,43.8078,18.5145,43.1427
12,totaltime:,39.0521,30.7640,33.4642
13,totaltime:,32.7664,29.6306,38.1140
14,totaltime:,43.4334,14.8777,43.7146
15,totaltime:,28.5933,16.8969,38.2142
16,totaltime:,28.4297,28.5868,33.4431
17,totaltime:,28.6037,30.2606,43.4092
18,totaltime:,31.8891,20.2853,28.5042
19,totaltime:,33.7666,14.8412,28.3797
20,totaltime:,41.9213,24.0811,28.5179
21,totaltime:,36.0006,31.0873,31.7246
22,totaltime:,38.0205,24.8814,33.5470
23,totaltime:,15.9383,15.2115,41.8722
24,totaltime:,17.0055,19.1876,35.3179
25,totaltime:,20.2145,30.2509,37.8857
```

Anhang A 6. CPU Python Programm

```
# Autor: Vahel Hassan
# Abschlussarbeit: Leistungsbewertung von Container
# Benchmark Daten Plotten und berechnen
import matplotlib.pyplot as plt

dateihandler = open('cpu_plotten.csv')

inhalt = dateihandler.read()

zeilen = inhalt.split('\n')
```

```
tabelle = []

for zeile in range(len(zeilen)):
    spalten = zeilen[zeile].split(',')
    tabelle.append(spalten)
    tabelle[zeile][2:] = [float(zahl) for zahl in tabelle[zeile][2:]]

    durchlaufe = [zeile[0] for zeile in tabelle]
    kata_container = [zeile[2] for zeile in tabelle]
    firecracker_container = [zeile[3] for zeile in tabelle]
    docker_container = [zeile[4] for zeile in tabelle]

# ------------------------------------Ergebnisse von Benchmark-------------------------------------
------------
    plt.plot(durchlaufe, docker_container, color='blue')
    plt.plot(durchlaufe, firecracker_container, color='red')
    plt.plot(durchlaufe, kata_container, color='orange')

    plt.xlabel("Durchläufe")
    plt.ylabel("Totaler Wert in Sekunden")
    plt.title("CPU Geschwindigkeit")

plt.show()
#------------------------------------Auswertung Docker-Container-----------------------------------
-------------
#Durchschnittlich benötigte suche von Primzhalen
durschnitt_docker = sum(docker_container)
print("Der Docker-Container (Blau) benötigt Durchschnittlich %.2f Sekunden um die Primzahlen zwischen 1 und
20.000 zu suchen." % (durschnitt_docker/25), "\n-")

#------------------------------------Auswertung von Firecracker-Container--------------------------
----------------------
#Durchschnittlich benötigte suche von Primzhalen
durschnitt_firecracker = sum(firecracker_container)
print("Der Firecracker-Container (Rot) benötigt Durchschnittlich %.2f Sekunden um die Primzahlen zwischen 1
und 20.000 zu suchen." % (durschnitt_firecracker/25), "\n-")

#------------------------------------Auswertung von Kata-Container---------------------------------
---------------
#Durchschnittlich benötigte suche von Primzhalen
durschnitt_kata = sum(kata_container)
print("Der Kata-Container (Orange) benötigt Durchschnittlich %.2f Sekunden um die Primzahlen zwischen 1 und
20.000 zu suchen." % (durschnitt_kata/25), "\n-")
#------------------------------------Auswertung Docker-Container-----------------------------------
-------------
#TOTALER Wert
ergebnisse_docker = \
                    [321238/(docker_container[0]), 321238/(docker_container[1]), 321238/(docker_contai-
ner[2]),
                    321238/(docker_container[3]), 321238/(docker_container[4]), 321238/(docker_container[5]),
                    321238/(docker_container[6]), 321238/(docker_container[7]), 321238/(docker_container[8]),
                    321238/(docker_container[9]), 321238/(docker_container[10]), 321238/(docker_contai-
ner[11]),
                    321238/(docker_container[12]), 321238/(docker_container[13]), 321238/(docker_contai-
ner[14]),
                    321238/(docker_container[15]), 321238/(docker_container[16]), 321238/(docker_contai-
ner[17]),
                    321238/(docker_container[18]), 321238/(docker_container[19]), 321238/(docker_contai-
ner[20]),
                    321238/(docker_container[21]), 321238/(docker_container[22]), 321238/(docker_contai-
ner[23]),
                    321238/(docker_container[24])]

#Durchschnittliche Rechenoperationen pro Sekunde
durschnitt_docker = \
                    321238/(docker_container[0]) + 321238/(docker_container[1]) + 321238/(docker_contai-
ner[2]) + \
                    321238/(docker_container[3]) + 321238/(docker_container[4]) + 321238/(docker_contai-
ner[5]) + \
                    321238/(docker_container[6]) + 321238/(docker_container[7]) + 321238/(docker_contai-
ner[8]) + \
                    321238/(docker_container[9]) + 321238/(docker_container[10]) + 321238/(docker_contai-
ner[11]) + \
                    321238/(docker_container[12]) + 321238/(docker_container[13]) + 321238/(docker_contai-
ner[14]) + \
                    321238/(docker_container[15]) + 321238/(docker_container[16]) + 321238/(docker_contai-
ner[17]) + \
                    321238/(docker_container[18]) + 321238/(docker_container[19]) + 321238/(docker_contai-
ner[20]) + \
                    321238/(docker_container[21]) + 321238/(docker_container[22]) + 321238/(docker_contai-
ner[23]) + \
                    321238/(docker_container[24])

print("Der Docker-Container (Blau) kann Durchschnittlich %.2f Rechenoperationen pro Sekunde ausführen." %
(durschnitt_docker/25), "\n-")

#------------------------------------Auswertung von Firecracker-Container--------------------------
----------
```

```
----------------------
#TOTALER Wert
ergebnisse_firecracker = \
                  [321238/(firecracker_container[0]), 321238/(firecracker_container[1]), 321238/(firecra-
cker_container[2]),
                  321238/(firecracker_container[3]), 321238/(firecracker_container[4]), 321238/(firecra-
cker_container[5]),
                  321238/(firecracker_container[6]), 321238/(firecracker_container[7]), 321238/(firecra-
cker_container[8]),
                  321238/(firecracker_container[9]), 321238/(firecracker_container[10]), 321238/(firecra-
cker_container[11]),
                  321238/(firecracker_container[12]), 321238/(firecracker_container[13]), 321238/(firecra-
cker_container[14]),
                  321238/(firecracker_container[15]), 321238/(firecracker_container[16]), 321238/(firecra-
cker_container[17]),
                  321238/(firecracker_container[18]), 321238/(firecracker_container[19]), 321238/(firecra-
cker_container[20]),
                  321238/(firecracker_container[21]), 321238/(firecracker_container[22]), 321238/(firecra-
cker_container[23]),
                  321238/(firecracker_container[24])]

#Durchschnittliche Rechenoperationen pro Sekunde
durschnitt_firecracker = \
                  321238/(firecracker_container[0]) + 321238/(firecracker_container[1]) + 321238/(firecra-
cker_container[2]) + \
                  321238/(firecracker_container[3]) + 321238/(firecracker_container[4]) + 321238/(firecra-
cker_container[5]) + \
                  321238/(firecracker_container[6]) + 321238/(firecracker_container[7]) + 321238/(firecra-
cker_container[8]) + \
                  321238/(firecracker_container[9]) + 321238/(firecracker_container[10]) + 321238/(firecra-
cker_container[11]) + \
                  321238/(firecracker_container[12]) + 321238/(firecracker_container[13]) + 321238/(fire-
cracker_container[14]) + \
                  321238/(firecracker_container[15]) + 321238/(firecracker_container[16]) + 321238/(fire-
cracker_container[17]) + \
                  321238/(firecracker_container[18]) + 321238/(firecracker_container[19]) + 321238/(fire-
cracker_container[20]) + \
                  321238/(firecracker_container[21]) + 321238/(firecracker_container[22]) + 321238/(fire-
cracker_container[23]) + \
                  321238/(firecracker_container[24])

print("Der Firecracker-Container (Rot) kann Durchschnittlich %.2f Rechenoperationen pro Sekunde ausführen."
% (durschnitt_firecracker/25), "\n-")

#--------------------------------------Auswertung von Kata-Container-------------------------------------
---------------
#TOTALER Wert
ergebnisse_kata = \
                  [321238/(kata_container[0]), 321238/(kata_container[1]), 321238/(kata_container[2]),
                  321238/(kata_container[3]), 321238/(kata_container[4]), 321238/(kata_container[5]),
                  321238/(kata_container[6]), 321238/(kata_container[7]), 321238/(kata_container[8]),
                  321238/(kata_container[9]), 321238/(kata_container[10]), 321238/(kata_container[11]),
                  321238/(kata_container[12]), 321238/(kata_container[13]), 321238/(kata_container[14]),
                  321238/(kata_container[15]), 321238/(kata_container[16]), 321238/(kata_container[17]),
                  321238/(kata_container[18]), 321238/(kata_container[19]), 321238/(kata_container[20]),
                  321238/(kata_container[21]), 321238/(kata_container[22]), 321238/(kata_container[23]),
                  321238/(kata_container[24])]

#Durchschnittliche Rechenoperationen pro Sekunde
durschnitt_kata = \
                  321238/(kata_container[0]) + 321238/(kata_container[1]) + 321238/(kata_container[2]) + \
                  321238/(kata_container[3]) + 321238/(kata_container[4]) + 321238/(kata_container[5]) + \
                  321238/(kata_container[6]) + 321238/(kata_container[7]) + 321238/(kata_container[8]) + \
                  321238/(kata_container[9]) + 321238/(kata_container[10]) + 321238/(kata_container[11]) +
\
                  321238/(kata_container[12]) + 321238/(kata_container[13])+ 321238/(kata_container[14]) +
\
                  321238/(kata_container[15]) + 321238/(kata_container[16])+ 321238/(kata_container[17]) +
\
                  321238/(kata_container[18]) + 321238/(kata_container[19])+ 321238/(kata_container[20]) +
\
                  321238/(kata_container[21]) + 321238/(kata_container[22])+ 321238/(kata_container[23]) +
\
                  321238/(kata_container[24])

print("Der Kata-Container (Orange) kann Durchschnittlich %.2f Rechenoperationen pro Sekunde ausführen." %
(durschnitt_kata/25), "\n-")

plt.plot(durchlaufe, ergebnisse_docker, color='blue', label = 'Docker-Container')
plt.plot(durchlaufe, ergebnisse_firecracker, color='red', label = 'Firecracker-Container')
plt.plot(durchlaufe, ergebnisse_kata, color='orange', label =  'Kata-Container')

plt.legend(loc='upper right')
plt.xlabel("Durchläufe")
plt.ylabel("Rechenoperationen pro Sekunde")
plt.title("CPU Geschwindigkeit")
```

```
plt.show()
```

Anhang A 7. Ergebnisse von MB/sec im Hauptspeicher abspeichern für RAM Test

```
# Benchmark RAM Test: Ergebnisse für Docker Container
102400.00 MB transferred (12195.62 MB/sec)
102400.00 MB transferred (11991.45 MB/sec)
102400.00 MB transferred (11571.40 MB/sec)
102400.00 MB transferred (8024.11 MB/sec)
102400.00 MB transferred (9004.17 MB/sec)
102400.00 MB transferred (11693.18 MB/sec)
102400.00 MB transferred (13084.89 MB/sec)
102400.00 MB transferred (12913.66 MB/sec)
102400.00 MB transferred (13066.23 MB/sec)
102400.00 MB transferred (13011.20 MB/sec)
102400.00 MB transferred (12966.20 MB/sec)
102400.00 MB transferred (11815.09 MB/sec)
102400.00 MB transferred (9807.75 MB/sec)
102400.00 MB transferred (8593.12 MB/sec)
102400.00 MB transferred (8187.01 MB/sec)
102400.00 MB transferred (8659.22 MB/sec)
102400.00 MB transferred (8987.97 MB/sec)
102400.00 MB transferred (8451.08 MB/sec)
102400.00 MB transferred (7599.87 MB/sec)
102400.00 MB transferred (9161.15 MB/sec)
102400.00 MB transferred (9202.97 MB/sec)
102400.00 MB transferred (9242.92 MB/sec)
102400.00 MB transferred (9156.25 MB/sec)
102400.00 MB transferred (8879.65 MB/sec)
102400.00 MB transferred (9180.80 MB/sec)
```

```
# Benchmark RAM Test: Ergebnisse für Firecracker Containerd
102400.00 MB transferred (12918.40 MB/sec)
102400.00 MB transferred (10921.80 MB/sec)
102400.00 MB transferred (7953.22 MB/sec)
102400.00 MB transferred (10864.56 MB/sec)
102400.00 MB transferred (11219.07 MB/sec)
102400.00 MB transferred (11209.14 MB/sec)
102400.00 MB transferred (11703.00 MB/sec)
102400.00 MB transferred (9798.41 MB/sec)
102400.00 MB transferred (9111.94 MB/sec)
102400.00 MB transferred (9140.12 MB/sec)
102400.00 MB transferred (9122.03 MB/sec)
102400.00 MB transferred (9350.39 MB/sec)
102400.00 MB transferred (9270.56 MB/sec)
102400.00 MB transferred (8727.21 MB/sec)
102400.00 MB transferred (8167.83 MB/sec)
102400.00 MB transferred (8157.82 MB/sec)
102400.00 MB transferred (8287.49 MB/sec)
102400.00 MB transferred (8684.63 MB/sec)
102400.00 MB transferred (8505.59 MB/sec)
102400.00 MB transferred (9033.94 MB/sec)
102400.00 MB transferred (9951.28 MB/sec)
102400.00 MB transferred (13067.99 MB/sec)
102400.00 MB transferred (12955.79 MB/sec)
102400.00 MB transferred (13017.93 MB/sec)
102400.00 MB transferred (12881.44 MB/sec)
```

```
# Benchmark RAM Test: Ergebnisse für Kata Container
    total time:                          44.7216s
    total time:                          39.7975s
    total time:                          32.0916s
    total time:                          44.1692s
    total time:                          44.0947s
    total time:                          29.1214s
    total time:                          42.2696s
    total time:                          43.2580s
    total time:                          34.5016s
    total time:                          37.3852s
    total time:                          43.8078s
    total time:                          39.0521s
    total time:                          32.7664s
    total time:                          43.4334s
    total time:                          28.5933s
    total time:                          28.4297s
    total time:                          28.6037s
    total time:                          31.8891s
    total time:                          33.7666s
    total time:                          41.9213s
    total time:                          36.0006s
    total time:                          38.0205s
    total time:                          15.9383s
    total time:                          17.0055s
    total time:                          20.2145s
```

Anhang A 8. CSV Datei für RAM

```
1,8811.76,12918.40,12195.62
2,7437.72,10921.80,11991.45
3,8550.27,7953.22,11571.40
4,9239.54,10864.56,8024.11
5,9287.28,11219.07,9004.17
6,9443.67,11209.14,11693.18
7,9371.69,11703.00,13084.89
8,9592.20,9798.41,12913.66
9,10546.95,9111.94,13066.23
10,10518.15,9140.12,13011.20
11,12424.42,9122.03,12966.20
12,12015.21,9350.39,11815.09
13,12437.20,9270.56,9807.75
14,12887.03,8727.21,8593.12
15,11678.49,8167.83,8187.01
16,9264.62,8157.82,8659.22
17,9206.69,8287.49,8987.97
18,9256.39,8684.63,8451.08
19,9285.30,8505.59,7599.87
20,9184.25,9033.94,9161.15
21,8863.75,9951.28,9202.97
22,9237.74,13067.99,9242.92
23,12821.67,12955.79,9156.25
24,12984.14,13017.93,8879.65
25,13065.08,12881.44,9180.80
```

Anhang A 9. RAM Python Programm

```
# Autor: Vahel Hassan
# Abschlussarbeit: Leistungsbewertung von Container
# Benchmark Daten Plotten und berechnen
import matplotlib.pyplot as plt
```

```
dateihandler = open('ram_plotten.csv')

inhalt = dateihandler.read()

zeilen = inhalt.split('\n')

tabelle = []

for zeile in range(len(zeilen)):
    spalten = zeilen[zeile].split(',')
    tabelle.append(spalten)
    tabelle[zeile][1:] = [float(zahl) for zahl in tabelle[zeile][1:]]

    durchlaufe = [zeile[0] for zeile in tabelle]
    kata_container = [zeile[1] for zeile in tabelle]
    firecracker_container = [zeile[2] for zeile in tabelle]
    docker_container = [zeile[3] for zeile in tabelle]

# ---------------------------------------Ergebnisse von Benchmark-----------------------------------------
-----------
    plt.plot(durchlaufe, docker_container, color='blue')
    plt.plot(durchlaufe, firecracker_container, color='red')
    plt.plot(durchlaufe, kata_container, color='orange')

    plt.xlabel("Durchläufe")
    plt.ylabel("MB/sec")
    plt.title("RAM Geschwindigkeit")

plt.show()

#---------------------------------------Auswertung Docker-Container---------------------------------------
-------------
#Durchschnittliche Wert
durschnitt_docker = sum(docker_container)
print("Der Docker-Container (Blau) kann Durchschnittlich %.2f MB/sec im Hauptspeicher speichern." % (dur-
schnitt_docker/25), "\n-")

#---------------------------------------Auswertung von Firecracker-Container-----------------------------
-----------------------
#Durchschnittliche Wert
durschnitt_firecracker = sum(firecracker_container)
print("Der Firecracker-Container (Rot) kann Durchschnittlich %.2f MB/sec im Hauptspeicher speichern." %
(durschnitt_firecracker/25), "\n-")

#---------------------------------------Auswertung von Kata-Container-------------------------------------
---------------
#Durchschnittliche Wert
durschnitt_kata = sum(kata_container)

print("Der Kata-Container (Orange) kann Durchschnittlich %.2f MB/sec im Hauptspeicher speichern." % (dur-
schnitt_kata/25), "\n-")
```

Anhang A 10. Network Python Programm

```
1,52.8,806,53.2
2,57.5,799,53.3
3,37.6,800,55.1
4,53.8,743,52.8
5,34.8,833,56.1
6,56.1,838,52.9
7,29.7,834,58.6
8,53.8,561,56.3
9,53.0,807,53.7
10,53.7,752,54.8
11,27.7,766,55.7
12,55.3,838,52.0
13,54.1,815,53.2
14,56.7,834,53.2
15,51.7,814,53.1
16,52.6,840,54.2
17,55.9,791,55.7
18,51.5,790,46.9
19,50.3,846,55.8
20,53.3,805,56.2
```

```
21,54.7,807,51.7
22,55.6,848,54.4
23,54.0,839,37.5
24,57.2,852,57.0
25,56.0,826,55.4
```

Anhang A 11. Network Python Programm

```
# Autor: Vahel Hassan
# Abschlussarbeit: Leistungsbewertung von Container
# Benchmark Daten Plotten und berechnen
import matplotlib.pyplot as plt

dateihandler = open('startzeit_plotten.csv')

inhalt = dateihandler.read()

zeilen = inhalt.split('\n')

tabelle = []

for zeile in range(len(zeilen)):
    spalten = zeilen[zeile].split(',')
    tabelle.append(spalten)
    tabelle[zeile][1:] = [float(zahl) for zahl in tabelle[zeile][1:]]

    durchlaufe = [zeile[0] for zeile in tabelle]
    kata_container = [zeile[1] for zeile in tabelle]
    firecracker_container = [zeile[2] for zeile in tabelle]
    docker_container = [zeile[3] for zeile in tabelle]

    plt.plot(durchlaufe,docker_container, color='blue')
    plt.plot(durchlaufe, firecracker_container, color='red')
    plt.plot(durchlaufe, kata_container, color='orange')

    plt.xlabel("Durchläufe")
    plt.ylabel("MB/sec")
    plt.title("Download Geschwindigkeit")

plt.show()

#--------------------------------------Auswertung Docker-Container-------------------------------------
-------------
#Durchschnittliche Rechenoperationen pro Sekunde
durschnitt_docker = sum(docker_container)
print("Der Docker-Container (Blau) Downloadet 432k Durchschnittlich in %.2f MB pro Sekunde." % (dur-
schnitt_docker/25), "\n-")

#--------------------------------------Auswertung von Firecracker-Container----------------------------
----------------------
#Durchschnittliche Rechenoperationen pro Sekunde
durschnitt_firecracker = sum(firecracker_container)
print("Der Firecracker-Container (Rot) Downloadet 432k Durchschnittlich in %.2f MB pro Sekunde." % (dur-
schnitt_firecracker/25), "\n-")

#--------------------------------------Auswertung von Kata-Container-----------------------------------
---------------
#Durchschnittliche Rechenoperationen pro Sekunde
durschnitt_kata = sum(kata_container)

print("Der Kata-Container (Orange) Downloadet 432k Durchschnittlich in %.2f MB pro Sekunde." % (dur-
schnitt_kata/25), "\n-")
```

Anhang A 12. Startzeit Python Programm

```
1,026.625,029.180,028.475
2,023.902,027.296,026.240
3,024.689,023.060,025.649
4,027.086,021.903,025.377
5,025.397,024.759,023.456
6,022.446,024.848,021.562
7,022.142,024.790,022.825
```

```
8,026.361,032.097,031.314
9,028.785,025.022,031.028
10,032.139,022.172,029.043
11,032.056,026.633,026.831
12,029.034,029.333,020.997
13,023.105,041.037,029.810
14,030.098,035.005,036.508
15,034.728,032.314,038.679
16,032.933,025.767,034.933
17,038.824,022.689,036.217
18,044.079,030.655,039.560
19,036.488,039.071,029.985
20,023.684,037.298,018.406
21,018.607,036.084,023.264
22,026.534,033.763,034.068
23,034.586,033.857,038.775
24,034.341,028.603,036.880
25,031.839,022.619,031.933
```

Anhang A 13. Startzeit Python Programm

```
# Autor: Vahel Hassan
# Abschlussarbeit: Leistungsbewertung von Container
# Benchmark Daten Plotten und berechnen
import matplotlib.pyplot as plt

dateihandler = open('startzeit_plotten.csv')

inhalt = dateihandler.read()

zeilen = inhalt.split('\n')

tabelle = []

for zeile in range(len(zeilen)):
    spalten = zeilen[zeile].split(',')
    tabelle.append(spalten)
    tabelle[zeile][1:] = [float(zahl) for zahl in tabelle[zeile][1:]]

    durchlaufe = [zeile[0] for zeile in tabelle]
    kata_container = [zeile[1] for zeile in tabelle]
    firecracker_container = [zeile[2] for zeile in tabelle]
    docker_container = [zeile[3] for zeile in tabelle]

# ---------------------------------------Ergebnisse von Benchmark-----------------------------------------
------------
    plt.plot(durchlaufe, docker_container, color='blue')
    plt.plot(durchlaufe, firecracker_container, color='red')
    plt.plot(durchlaufe, kata_container, color='orange')

    plt.xlabel("Sekunden")
    plt.ylabel("Durchläufe")
    plt.title("Start Geschwindigkeit")

plt.show()

#---------------------------------------Auswertung Docker-Container---------------------------------------
--------------
#Durchschnittliche Rechenoperationen pro Sekunde
durschnitt_docker = sum(docker_container)
print("Der Docker-Container (Blau) startet und beendet den Sysbench Test Durchschnittlich innerhalb von
%.2f Sekunden." % (durschnitt_docker/25), "\n-")

#---------------------------------------Auswertung von Firecracker-Container-----------------------------
----------------------
#Durchschnittliche Rechenoperationen pro Sekunde
durschnitt_firecracker = sum(firecracker_container)
print("Der Firecracker-Container (Rot) startet und beendet den Sysbench Test Durchschnittlich innerhalb von
%.2f Sekunden." % (durschnitt_firecracker/25), "\n-")

#---------------------------------------Auswertung von Kata-Container------------------------------------
----------------
#Durchschnittliche Rechenoperationen pro Sekunde
durschnitt_kata = sum(kata_container)
```

```
print("Der Kata-Container (Orange) startet und beendet den Sysbench Test Durchschnittlich innerhalb von
%.2f Sekunden." % (durschnitt_kata/25), "\n-")
```