



Fakultät Informatik

Softwaretechnik und Medieninformatik

Bachelorthesis

Evaluation verschiedener Container-Technologien

Corvin Schapöhler

Mat.-Nr. 751301

Sommersemester 2018

Firma: NovaTec GmbH, Leinfelden-Echterdingen

Erstprüfer: Prof. Dr.-Ing. Dipl.-Inform. Kai Warendorf

Zweitprüfer: Dipl.-Ing. Matthias Haeussler

Ehrenwörtliche Erklärung

Hiermit versichere ich, Corvin Schapöhler, dass ich die vorliegende Bachelorarbeit mit dem Titel „Evaluation verschiedener Container-Technologien“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Stellen der Arbeit, die im Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht in gleicher oder anderer Form als Prüfungsleistung vorgelegt worden.

Stuttgart, 30. Juni 2018

Ort, Datum

Corvin Schapöhler

Kurzfassung

Diese Thesis behandelt einen Vergleich verschiedener Container-Technologien, wobei Container-Runtimes im Mittelpunkt stehen. Es soll die Frage beantwortet werden, warum Docker der aktuelle Branchenprimus ist. Dabei wurden verschiedene Container-Runtimes wie Docker, runc, rkt oder LXD genutzt, um eine eigene serviceorientierte Anwendung bereitzustellen. Docker und rkt bieten für diesen Fall viele Tools und sind deutlich spezifischer für diese Aufgabe ausgelegt sind. Runtimes wie LXC / LXD dienen vor allem dazu, Infrastruktur zu bieten. Im weiteren Verlauf wurden die Themen Orchestrierung, Sicherheit und die Serverless-Technologie betrachtet, wobei ein Ausblick auf aktuelle Anwendungsfälle der Container Technologie gegeben wird.

Abstract

This thesis is about a comparison between different container technologies, where the focus is on container runtimes. For this, multiple runtimes like Docker, runc, rkt or LXD were used to deploy a serviceoriented application. Docker and rkt offer a lot of tools and are specifically designed for such tasks. Runtimes like LXC / LXD are designed to offer infrastructure. Furthermore the topics orchestration, security and the serverless technology were looked at, where a look in the future for use cases of the container technology is given.

Stichworte / Keywords: *Container, Docker, OCI, runc, Kubernetes, Serverless, Linux, rkt, LXD, gVisor*

Inhaltsverzeichnis

Ehrenwörtliche Erklärung	I
Kurzfassung	II
Abstract	II
1 Einleitung	1
1.1 Motivation	1
1.2 Aufbau der Arbeit	1
2 Grundlagen	3
2.1 Standards	4
2.2 Funktionsweise	6
2.3 Eigene Implementierung	11
3 Geschichte	17
3.1 Container-Runtimes	17
3.2 Weiterführende Technologien	21
4 Evaluation: Container-Runtimes	23
4.1 Vorgehen	23
4.2 Docker Stack	24
4.3 rkt	27
4.4 LXD / LXC	31
4.5 runC	33
4.6 VM basierte Runtimes	36
4.7 Fazit	38
5 Aktuelle Themen	42
5.1 Security	42
5.2 Orchestrierung	43
5.3 Serverless	45
<hr/>	
Evaluation verschiedener Container-Technologien	III

5.4 Fazit	46
Abbildungsverzeichnis	A
Tabellenverzeichnis	B
Listings	C
Akronyme	D
Quellen und weiterführende Literatur	E

1 Einleitung

1.1 Motivation

Die Welt wird immer stärker vernetzt. Durch den Drang, Anwendungen für viele Nutzer zugänglich zu machen, besteht der Bedarf an Cloud-Diensten wie Amazon Web Services, Microsoft Azure oder IBM Bluemix. Eine dabei immer wieder auftretende Schwierigkeit ist es, die Skalierbarkeit der Services zu gewährleisten. Selbst wenn viele Nutzer gleichzeitig auf einen Service zugreifen, darf dieser nicht unter der Last zusammenbrechen.

Bis vor einigen Jahren wurde diese Skalierbarkeit durch Virtuelle Maschinen (VMs) gewährleistet. Doch neben großem Konfigurationsaufwand haben VMs auch einen großen Footprint und sind für viele Anwendungen zu ineffizient. Eine Lösung für dieses Problem stellen Container dar.

Diese Arbeit gibt einen Einblick in das Thema Container-Virtualisierung und beantwortet die Fragen, wie sich Docker als führende Technologie durchsetzen konnte, wie sich andere Technologien im Vergleich zu Docker schlagen und was die Zukunft in Form von Serverless-Technologien mit Bezug zu Containern bereithält.

1.2 Aufbau der Arbeit

Zu Beginn der Arbeit werden benötigte Grundlagen der Technologie erläutert. Dabei werden bestehende Container-Standards betrachtet und alle benötigten Kernel-Funktionen erklärt, die in Container-Runtimes Verwendung finden. Um einen besseren Einblick in die Technologie zu geben wird gezeigt, wie man mit Bash-Befehlen ohne Container-Runtime einen Prozess von einem Host-OS

isoliert. Dabei wird darauf eingegangen, wie eine eigene Dateihierarchie isoliert werden kann, wie Namespaces dabei helfen, Funktionen des Linux-Kernels zu virtualisieren und wie der isolierte Prozess sicherer ausgeführt werden kann.

Im Anschluss wird die Frage beantwortet, wie Docker die populärste Container-Technologie wurde. Dazu wird die Geschichte betrachtet und Probleme einzelner Technologien aufgezeigt. Zudem wird gezeigt, wie Innovation durch die Vereinfachung von Schnittstellen entstehen kann.

Das folgende Kapitel vergleicht die aktuellen Container-Angebote Docker, rkt, LXD und runc miteinander und grenzt ab, wo welche Runtime Vorteile bietet, warum Docker nicht in jedem Fall die beste Lösung ist und welche Stärken und Schwächen andere Runtimes haben. Zudem wird ein Einblick in andere Ansätze der Isolierung mittels Container gegeben, indem die Runtimes Kata Containers und gVisor vorgestellt werden.

Das abschließende Kapitel behandelt die aktuellen Themen Sicherheit, Orchestrierung und Serverless-Technologien, um aufzuzeigen, welche Bereiche bei der weiteren Arbeit an Containern betrachtet werden können. Dabei wird auf die Orchestrierungsplattform Kubernetes, die Sicherheitslücke Dirty COW und aktuelle Function-as-a-Service (FaaS) Lösungen eingegangen.

2 Grundlagen

Container werden häufig als leichtgewichtige VMs beschrieben. Dies ist allerdings nicht richtig. Wie in Abbildung 1 zu erkennen, virtualisieren Container kein vollständiges Betriebssystem (OS), sondern lediglich das benötigte Dateisystem. Dabei wird der Kernel des Hosts nicht virtualisiert, sondern mitverwendet. Dies macht Container deutlich leichtgewichtiger als VMs, isoliert allerdings weniger umfangreich als diese.

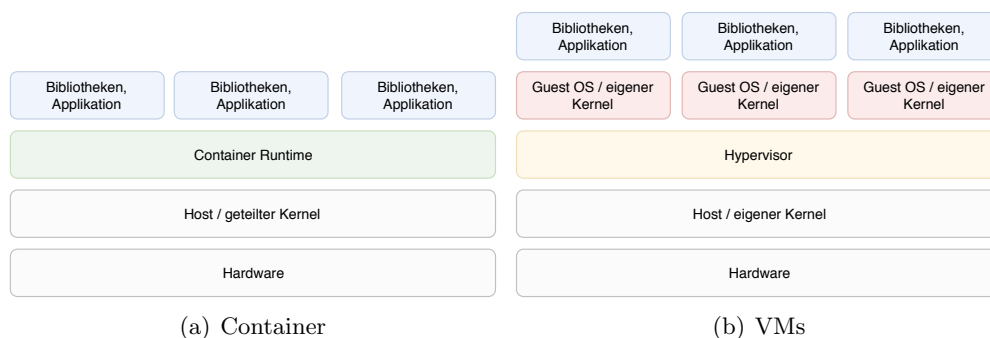


Abbildung 1: Container Isolation im Vergleich zu VMs

Dieses Kapitel behandelt alle benötigten Grundlagen, die zur Isolation eines Prozesses benötigt werden. Es werden vorhandene Standards wie die Open Container Initiative (OCI) und benötigte Systemcalls wie Change Root (chroot) näher erläutert. Zudem wird beschrieben, wie die Isolation, die Container bieten, durch Systemmittel des Linux-Kernels selber erreicht wird.

2.1 Standards

Durch die immer größere Verwendung von Containern und die Verbreitung verschiedener Container-Runtimes ist die Standardisierung eine wichtige Aufgabe. Folgend werden Standardisierungsprojekte aufgezählt, die bestehenden Spezifikationen erläutert und aktuelle Aufgaben der Projekte näher betrachtet.

2.1.1 App Container

App Container (appc) ist ein Standard, der viele Aspekte innerhalb der Container-Landschaft behandelt. Dabei lag die Hauptaufgabe darin, eine Laufzeitumgebung, wie auch das Image-Format und die Verbreitung von Images zu standardisieren. Seit 2016 wird das Projekt nicht mehr aktiv weiterentwickelt, da mit der Gründung der OCI ein größeres Standardisierungsprojekt entstand. Bestandteile der appc wurden von der OCI übernommen und dienen als Vorlage für die Spezifikation dieser.

2.1.2 Open Container Initiative

Die OCI ist eine Initiative, die seit 2015 unter der Linux Foundation agiert. Das Ziel der OCI ist es, einen offenen Standard für Container zu schaffen, sodass die Wahl der Container-Runtime nicht mehr zu Inkompatibilität führt. Dabei liegt der Fokus auf eine einfache, schlanke Implementierung (Open Container Initiative, 2018).

Die OCI arbeitet aktuell an drei Spezifikationen. Die runtime-spec standardisiert die Laufzeitumgebung von Containern. Dabei wird festgelegt, welche Konfiguration, Prinzipien und Schnittstellen Laufzeitumgebungen stellen müssen. Um die Umsetzung der runtime-spec zu fördern, stellt die OCI eine beispielhafte Implementierung durch runC. Das zweite Projekt der OCI ist die image-spec. Dieses versucht einen Standard für Images zu definieren. Dabei plant die OCI nicht, vorhandene Image-Formate zu ersetzen, sondern auf diesen aufzubauen und sie zu erweitern (Open Container Initiative, 2018).

Wie in Tabelle 1 zu sehen, wurden einige Konzepte des appc-Projekts in die

OCI übernommen. Vor allem die Image-Spezifikation wurde durch die Mitarbeit ehemaliger appc-Maintainer gefördert. Seit Mai 2018 publiziert die OCI auch eine Spezifikation zur Verbreitung von Images, die auf der Docker Registry HTTP API basiert. Themen, die außerhalb des Scopes der OCI liegen, werden teilweise in die Cloud Native Computing Foundation (CNCF) aufgenommen (Polvi, 2015).

	Standard		Container Runtime	
	OCI	appc	Docker	rkt
Container Image	✗	✓	OCI image-spec	App Container Image
Image Verbreitung	✓	✓	Docker Registry	appc Discovery Spec
Lokales Speicherformat	✓	✗	keine Spezifikation	keine Spezifikation
Runtime	✓	✓	runC	appc runtime Spec

Tabelle 1: Standards OCI und AppC im Vergleich (Polvi, 2015)

2.1.3 Cloud Native Computing Foundation

Die CNCF beschäftigt sich im Gegensatz zur OCI nicht nur mit Containern, sondern der kompletten Cloud-Native-Landschaft (CNCF, 2018). Projekte wie Kubernetes (K8s) und Prometheus werden durch die CNCF weiterentwickelt und publiziert. Da der Cloud-Native Entwicklungsprozess von Containern getragen wird, spielen Technologien wie containerd und rkt eine entscheidende Rolle für die CNCF und sind ein großer Teil der Cloud-Native-Landscape, wie in Abbildung 2 gezeigt. Neben Container-Runtimes beinhaltet die CNCF auch Projekte zur Orchestrierung von Containern. Außerdem Spezifikationen wie TUF, die das updaten von Software spezifiziert (CNCF, 2017).

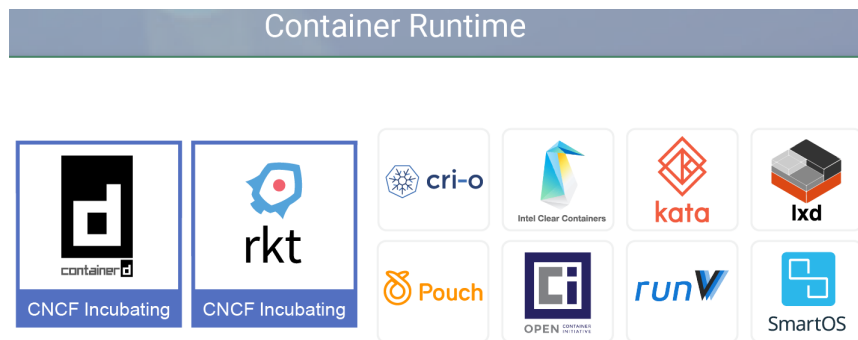


Abbildung 2: CNCF Container Runtime Landschaft (CNCF, 2018)

2.2 Funktionsweise

Container isolieren einzelne Prozesse durch verschiedene Kernel-Technologien, die im Folgenden erklärt werden.

2.2.1 Change Root

Chroot ist ein Unix Systemaufruf, der es erlaubt, einen Prozess in einem anderen Wurzelverzeichnis auszuführen (McGrath, 2017). Daraus folgt, dass der Prozess in einer eigenen Verzeichnisstruktur arbeitet und keine Dateien des Host-OS ändern kann. Chroot erlaubt somit die Isolierung des Dateisystems, die Container nutzen.

2.2.2 Control Groups

Control groups (cgroups) dienen dazu, Systemressourcen für einzelne Prozesse zu limitieren. Im OS sind cgroups als Dateihierarchie repräsentiert. Das gesamte cgroup-Dateisystem ist unter `/sys/fs/cgroup/` zu finden. Cgroups stellen zur Steuerung verschiedene Controller zur Verfügung, wie in Tabelle 2 zu sehen.

Controller	Ressource
io	Zugriff und Nutzung von Block Geräten wie Festplatten
memory	Monitoring und Beschränken des Arbeitsspeichers
pids	Limitierung der Anzahl an Unterprozessen
perf_event	Erlaubt Performance Monitoring der Prozesse
rdma	Zugriffe über Remote Direct Memory Access limitieren oder sperren
cpu	CPU-Zyklen und maximale CPU-Bandwidth

Tabelle 2: Cgroups-Controller und deren Verwendung (S. H. M. Kerrisk, 2018)

2.2.3 Namespaces

Namespaces abstrahieren einzelne Bereiche des OS. Sie werden genutzt, um globale Ressourcen zu virtualisieren. Ein Namespace kapselt dabei einzelne Ressourcen (siehe Tabelle 3). Veränderungen an diesen sind für alle Prozesse innerhalb desselben Namespaces sichtbar, allerdings außerhalb dieses unsichtbar (Biederman, 2017).

Namespace	Ressource
Cgroup	Cgroup-Dateisystem
IPC	System V IPC, POSIX Nachrichten
Network	Netzwerk Geräte, Stacks, Ports, ...
Mount	Mount Punkte
PID	Prozess IDs
User	Nutzer und Gruppen IDs
UTS	Hostnamen und Domänennamen

Tabelle 3: Linux Namespaces und verbundene Ressourcen (Biederman, 2017)

2.2.4 Mounting

Durch die Isolation eines Prozesses und der Bedingung, das Container unveränderlich sein sollen, stellt sich die Frage, wie Containern dynamische Inhalte aus dem Host-System zur Verfügung gestellt werden. Dies ist vor allem wichtig, wenn bei Veränderung der Umgebung nicht die Container neu gestartet werden sollen. Sollte zum Beispiel eine neue Datei durch einen Webserver zur Verfügung gestellt werden, möchte man nicht den Container neu starten. Die Lösung dieses Problems ist der Unix-Systembefehl `mount`.

Mit diesem Befehl wird eine beliebige Dateihierarchie an eine andere Stelle des Dateibaums angeheftet. Durch dieses vorgehen kann man Ordner vom Host-System für das isolierte Dateisystem des Containers zugänglich machen. Dabei ist zu beachten, dass es sich bei dem gemounteten Ordner nicht um einen symbolischen Link handelt (siehe Abbildung 3). Diese könnten durch den Aufruf von `chroot` nicht mehr aufgelöst werden.

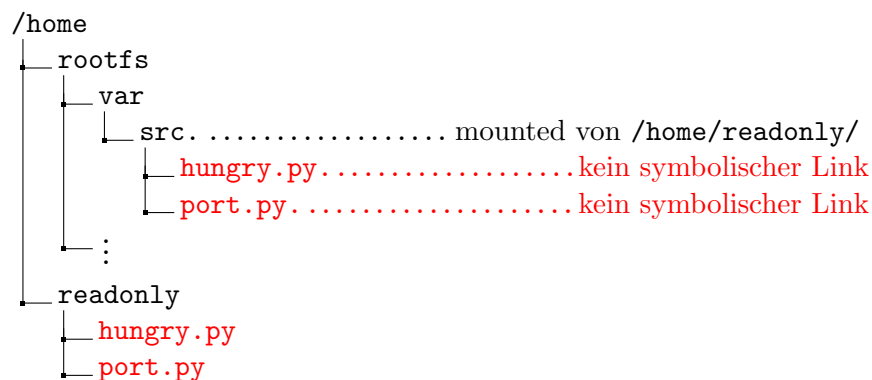


Abbildung 3: Auszug aus Dateisystem mit gemounteten Dateien

2.2.5 Netzwerk

Einen weiteren Aspekt, den Container vom Host-OS isolieren, ist das Netzwerk. Dabei kommen virtuelle Ethernet-Adapter zum Einsatz. Diese erlauben es, ein unabhängiges Netzwerk zu erzeugen. Ein Adapter der virtuellen Ethernet-Verbindung wird dabei der Process Identifier (PID) des Containers zugewiesen, das andere dem Host, wie in Abbildung 4 gezeigt. Zusätzlich wird der Network-Namespace genutzt um eine vollständige Isolation des Netzwerks zu erhalten.

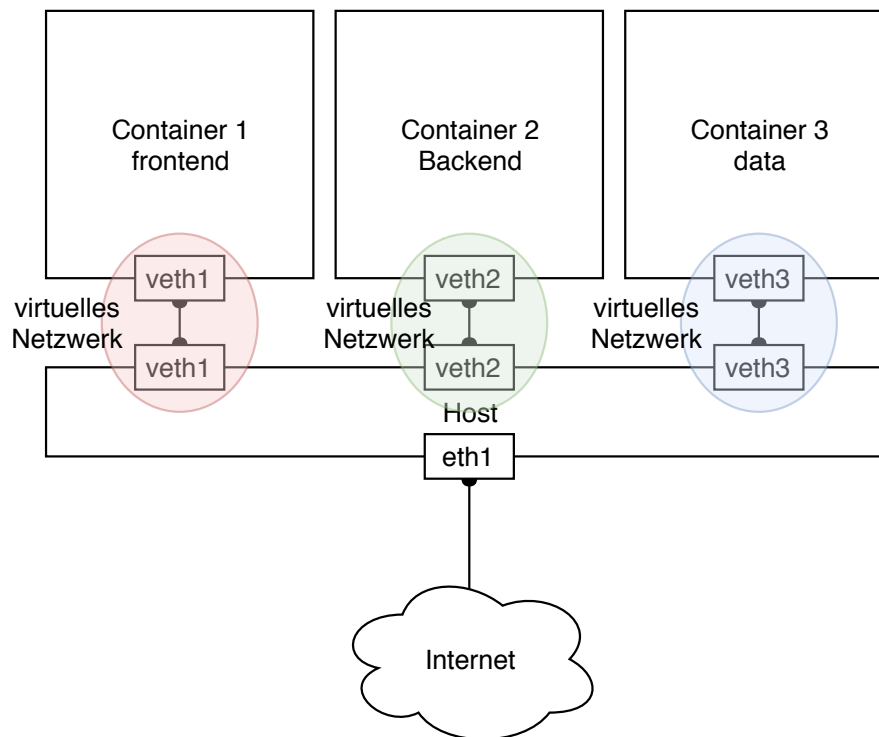


Abbildung 4: Netzwerk Topologie durch Trennung des Namespaces

2.2.6 Sicherheit

"Docker is about running random code downloaded from the Internet and running it as root"

—Dan Walsh (Red Hat)

Container haben ein großes Problem. Manche genannten Kernel-Features müssen als Nutzer **root** ausgeführt werden. Dadurch haben die gestarteten Prozesse häufig Berechtigungen, die es erlauben würden, aus der Isolierung des Containers auszubrechen. Um dies zu verhindern, können verschiedene Sicherheitskonzepte verwendet werden.

Das leichteste dieser Konzepte sind Capabilities. Jedem Prozess, sowie jeder

Datei kann eine Liste an Capabilities zugeordnet oder genommen werden (siehe Tabelle 4). Dabei können einzelnen Dateien beispielsweise die Rechte genommen werden, auf Port 80 zu hören. Auch viele Systemaufrufe können über Capabilities gewährt oder verwehrt werden.

Ein anderes Konzept ist die Implementation eines *Mandatory Access Control*-Systems wie SELinux oder AppArmor. Diese Implementationen sind granularer als Capabilities, allerdings mit einem höheren Konfigurationsaufwand verbunden.

Capability	Systemaufruf	Erklärung
CAP_CHOWN	chown	Ändern des Owners einer Datei
CAP_KILL	kill	Beenden eines Prozesses
CAP_CHROOT	chroot	Wechseln des Root Directories
CAP_NET_BIND_SERVICE		Binden eines Prozesses auf Ports <1024
CAP_SYS_TIME	stime, settimeofday	Setzen der Systemzeit
CAP_SYS_ADMIN	mount, umount, setns, pipe, syslog, ...	Alles, was in keine andere Kategorie passt

Tabelle 4: Einige Capabilities (M. Kerrisk, 2018)

2.2.7 Container unter Windows

Viele Kernel-Features des Linux-Kernels erlauben eine Isolation und wurden teilweise spezifisch für diese entwickelt (Biederman, 2017). Seit 2016 können spezifisch Docker-Container auch unter Windows genutzt werden. Dabei trennt Microsoft Container in zwei verschiedene Isolationen auf.

Windows Server Container 2016 sind ein nativer Windows Ansatz zur Isolation

eines Prozesses. Dabei werden, wie bereits unter Linux Systemen, verschiedene Kernel-Technologien verwendet, um einen einzelnen Prozess zu isolieren. Der andere Ansatz sind Hyper-V Container. Diese nutzen minimale VM Instanzen zur Isolation eines Containers. Dabei ist der größte Unterschied, das Hyper-V Container eine minimale Instanz eines Windows Kernels nutzen, um auf diesem einzelne Isolationen zu erstellen, während Windows Server Container einen gemeinsamen Kernel nutzen.

2.3 Eigene Implementierung

Um die in Abschnitt 2.2 erläuterten Kernel-Features näher zu beleuchten wird folgend gezeigt, wie ein Prozess isoliert von einem Host-System ausgeführt werden kann. Dabei wird darauf eingegangen, wie ein tarball mit dem Tool buildroot erstellt werden kann. Dieser lässt sich folgend in Container-Runtimes wie rkt oder Docker importieren. Außerdem wird ein Prozess innerhalb des komprimierten Dateisystems mithilfe der Kernel-Funktionen aus Abschnitt 2.2 isoliert. Das Ergebnis ist ein Dateisystem innerhalb eines Host-OS, indem eine Instanz der Python-Runtime isoliert und ohne Root-Berechtigungen ausgeführt wird.

2.3.1 Erstellen eines tarballs

Bei einem tarball handelt es sich um ein komprimiertes Dateisystem. Buildroot ist ein Unix-Tool, welches zur Erstellung von minimalistischen Linux-Distributionen für Embedded-Systems entworfen wurde. Es erlaubt aber auch, nur ein Dateisystem zu erzeugen, ohne Kernel oder Init-System. Dies ist entscheidend, da bei Containern der bestehende Kernel des Host-OS mitbenutzt wird (*siehe Abbildung 1*). Ein Init-System wird in Containern nicht benötigt, da ihre Funktion das initialisieren neuer Prozesse ist. Container isolieren allerdings nur einen Prozess. Diese Features von buildroot erlauben es, einen tarball zu erzeugen, der als Dateisystem von Containern genutzt werden kann.

Zudem erlaubt es buildroot, einzelne Bibliotheken, wie zum Beispiel die Python-Runtime, beim Build-Prozess der Distribution zu integrieren. Nach dem Einstellen der benötigten Bibliotheken und dem Deaktivieren der für Container unnötigen Features erzeugt Buildroot ein Makefile. Durch das Starten des

Build-Prozesses mit dem Befehl `make` wird die gewünschte Linux-Distribution erstellt. Am Ende dieses Prozesses liegt im Ordner `/buildroot/out/images/` das gewünschte Dateisystem `rootfs.tar` (siehe Abbildung 5).

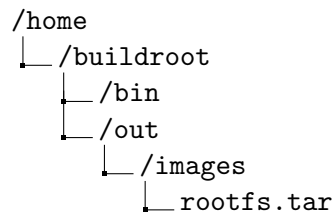


Abbildung 5: Dateisystem nach erfolgreichem Bauprozess mit buildroot

2.3.2 Isolieren des tarballs

In Abschnitt 2.3.1 wurde ein Dateisystem mit der Python-Runtime erstellt. Dieses muss nun isoliert, ein Pythonprogramm in das Dateisystem gemounted und ausgeführt werden.

Der erstellte tarball wird durch die in Listing 1 gezeigten Bash-Befehle entpackt. Durch den Aufruf aus Listing 1 entsteht die in Abbildung 6 zu sehende Dateistruktur.

```
cd /home
mkdir rootfs
cp rootfs.tar rootfs/
cd rootfs
sudo tar xvf rootfs.tar
sudo rm rootfs.tar
```

Listing 1: Entpacken des buildroot tarballs nach `/home/rootfs`

Um einen Prozess mit dem Wurzelverzeichnis `/home/rootfs/` auszuführen, ist lediglich der Aufruf `sudo chroot rootfs /usr/bin/python3.6 -m http.server` nötig.

Durch diesen wird ein Webserver auf Adresse `http://0.0.0.0:8000` ausgeführt, der alle ihm zugänglichen Dateien zum Download bereitstellt. Beim Aufruf

```
/home/rootfs/  
└─ usr  
    └─ bin  
        ├── python -> python3.6. .... nicht aus Hostsystem  
        └─ python3.6
```

Abbildung 6: Dateibaum nach entpacken der `rootfs.tar`

fen dieser Adresse erkennt man, dass der Webserver nur Zugriff auf die in `/home/rootfs/` liegenden Dateien hat, wie in Abbildung 7 gezeigt.

Directory listing for /

- [bin/](#)
- [dev/](#)
- [etc/](#)
- [lib/](#)
- [lib64/](#)
- [linuxrc/](#)
- [media/](#)
- [mnt/](#)
- [opt/](#)
- [proc/](#)
- [root/](#)
- [run/](#)
- [sbin/](#)
- [sys/](#)
- [tmp/](#)
- [usr/](#)
- [var/](#)

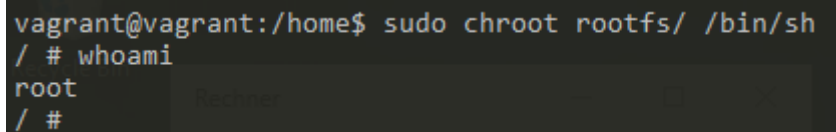
Abbildung 7: Python Webserver mit festgesetztem Root-Verzeichnis

Um weiterhin Zugriff auf dynamische Inhalte aus dem Host-System zu haben, kann man, wie in Abschnitt 2.2.4 erklärt, entsprechende Verzeichnisse in das neue `rootfs` des Prozesses mounten (siehe Listing 2).

Durch die Dateitrennung und den Aufruf von `chroot` tritt allerdings ein großes Problem auf. Der Python-Webserver wird mit erhöhten Rechten ausgeführt, da diese für den Aufruf von `chroot` benötigt werden (siehe Abbildung 8).

```
nsenter --mount=/proc/<PID isolierter Prozess>/ns/mnt \  
mount --bind -o ro $PWD/readonly $PWD/rootfs/var/src
```

Listing 2: Mounten von Verzeichnis `/readonly/` zu `/rootfs/var/src/`



```
vagrant@vagrant:/home$ sudo chroot rootfs/ /bin/sh  
/ # whoami  
root  
/ #
```

Abbildung 8: Root-Eskalation durch Aufruf von `sudo chroot`

Dies führt zu vielen Problemen. Ein Prozess, der nur auf diese Weise isoliert wird, könnte beispielsweise Prozesse auf dem Hostsystem mit `kill <pid>` beenden. Die Lösung dieses Problems sind die in Abschnitt 2.2.3 beschriebenen namespaces.

Um alle Prozesse des Hostsystems vor dem Container zu verstecken, muss der PID-Namespace des Container-Prozesses neu gemounted werden, wie in Listing 3 beschrieben.

```
sudo unshare -p --mount-proc=$PWD/rootfs/proc -f chroot  
↪ rootfs /bin/sh
```

Listing 3: Remount des PID-Namespaces und Chroot einer Shell

Beim Aufruf von „`ps aux`“ wird nur noch der Prozess `/bin/sh` angezeigt, der die PID 1 bekommen hat. Dieses Vorgehen löst allerdings nicht die Ursache des Problems. Der gestartete Prozess läuft auch weiterhin unter dem Nutzer `root`. Ein auf diese Weise isolierter Prozess kann zum Beispiel auf Port 80 hören. Um diese Berechtigungen zu entfernen werden die in Abschnitt 2.2.6 angesprochenen Capabilities verwendet.

Durch den Aufruf von `capsh --drop=cap_net_bind_service --chroot=rootfs/ --` hat die in `rootfs` gestartete `/bin/bash` nicht mehr die Möglichkeit, auf niedrigere Ports wie Port 80 zu hören.

Um vollständige Isolation des Containers zu erreichen, müssen Systemressourcen, wie Arbeitsspeicher oder CPU-Zyklen, limitiert werden. Dazu dienen die in Abschnitt 2.2.2 beschriebenen cgroups.

Um eine cgroup zu erstellen, muss ein Ordner unterhalb der cgroup Dateihierarchie erstellt werden. Folgend kann man einen Prozess einer cgroup zuweisen, indem man die PID des Prozesses in die Datei `/sys/fs/cgroup/CONTROLLER/CGROUPNAME/tasks` schreibt (siehe Listing 4).

```
mkdir /sys/fs/cgroup/memory/container  
echo $ContainerPID > /sys/fs/cgroup/memory/container/tasks
```

Listing 4: Erzeugen einer memory cgroup namens container

CGroups dienen dazu, Systemressourcen zu limitieren. Dazu können Limits in entsprechende Pseudo-Filehandles geschrieben werden. Um z.B. die Arbeitsspeichermenge auf $\sim 100\text{Mb}$ zu limitieren, können die in Listing 5 gezeigten Bash-Befehle ausgeführt werden.

```
echo "0" > /sys/fs/cgroup/memory/container/memory.swappiness  
echo "100000000" >  
↪ /sys/fs/cgroup/memory/container/memory.limit_in_bytes
```

Listing 5: Limitieren des Arbeitsspeichers und Memory-Swap deaktivieren

Um zu testen, ob die Zuweisung funktioniert und durch den isolierten Prozess maximal $\sim 100\text{Mb}$ Arbeitsspeicher belegt werden können, kann das Python Programm aus Listing 6 ausgeführt werden. Abbildung 9 zeigt die Ausgabe des Prozesses, der sich in der cgroup container befindet.

Fazit

Durch das beschriebene Vorgehen kann ein beliebiger Prozess auch ohne Container-Runtime isoliert werden. Diese simplifizieren das Vorgehen aber erheblich. So kann mit Ergebnis das selbe Verhalten durch die Commandozeile `docker run -d python:alpine3.6` erreicht werden

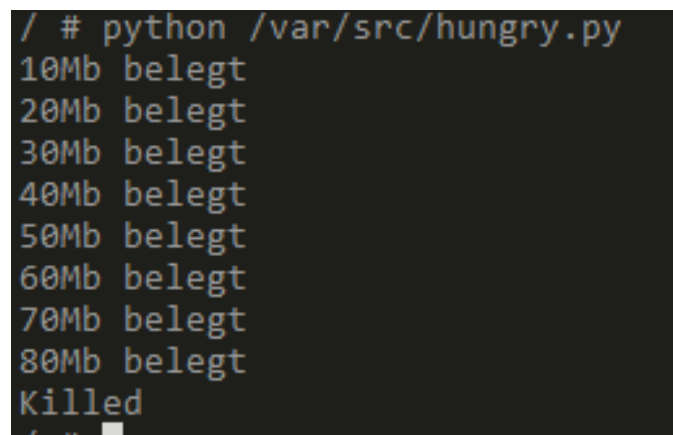
```
#hungry.py - Eating up memory in 10Mb blocks
import time

TEN_MEGABYTE = 10000000

f = open("/dev/urandom", "rb")
data = bytearray()
i = 0

while True:
    data.extend(f.read(TEN_MEGABYTE))
    i += 1
    print("%dMb belegt" % (i*10,))
    time.sleep(1)
```

Listing 6: Python Programm hungry.py um Arbeitsspeicher zu verbrauchen



```
/ # python /var/src/hungry.py
10Mb belegt
20Mb belegt
30Mb belegt
40Mb belegt
50Mb belegt
60Mb belegt
70Mb belegt
80Mb belegt
Killed
```

Abbildung 9: Ausgabe des Pythonprogramms hungry.py

3 Geschichte

Um die Frage zu beantworten, wie Docker die führende Container-Technologie wurde, wird im Folgenden die Geschichte dieser betrachtet. Dabei liegt der Schwerpunkt auf Probleme und Lösungen der Technologien.

3.1 Container-Runtimes

Wie in Tabelle 6 zu sehen, sind Container keine neue Erfindung. Bereits 2008 wurde die erste volle Implementierung einer Container-Runtime mit LXC veröffentlicht. Im folgenden Abschnitt wird ein Blick auf Container-Runtimes in der Vergangenheit geworfen und die Frage geklärt, wie Docker die erfolgreichste Runtime wurde.

3.1.1 Vor LXC: Isolation mit Kernel-Patches

Bereits 1979 wurde mit chroot die erste Möglichkeit der Isolation veröffentlicht. Diese kam mit dem Betriebssystem Unix V7 und erlaubte es erstmals, verschiedene Prozesse in unterschiedliche Dateisystemen zu trennen. Der Systemaufruf wurde 1982 in BSD hinzugefügt (Osnat, 2018). 20 Jahre später rücken isolierbare Prozesse durch FreeBSD Jails in den Mittelpunkt. Diese erweitern das Chroot-Konzept, indem nicht nur das Dateisystem vom Host-System getrennt wird, sondern auch Hostnamen, IP-Adressen und die Nutzerverwaltung (The FreeBSD Documentation Project, 2018).

Neben FreeBSD Jails wurde bereits 2001 Linux VServer veröffentlicht, eine Software, die ähnlich wie Jails eine Isolation des Dateisystems und auch der Netzwerkadresse erlaubt. Entgegen der aktuell noch weiterentwickelten Jails

war der letzte stabile Release von VServer 2008 (Pötzl, 2011). Der größte Nachteil des VServers waren Kernel-Patches, die benötigt wurden, um die Isolierung zu gewährleisten.

In den folgenden Jahren wurden immer mehr Lösungen zur Isolation von Prozessen veröffentlicht. Darunter Oracles Solaris Containers, das auf Zonen im Betriebssystem setzt und Open VZ, welches wie VServer, einen gepatchten Linux-Kernel benötigt, aber auch Ressourcen isolieren kann. Der größte Nachteil, den alle diese Technologien haben, ist die unzureichende und komplizierte Virtualisierung einzelner Prozesse. Zudem müssen Funktionen nachgepatched werden, die zur Isolation notwendig sind. Dies führte 2006 dazu, dass Entwickler von Google eine bessere Lösung entwickelten, Process Containers. Diese erlaubten ohne Patches eine einfache Verwendung und Isolation einzelner Ressourcen. Im Jahr 2007 wurden Process Container unter dem Namen cgroups in den Linux-Kernel merged und liefern seitdem eines der Fundamente für aktuelle Container-Technologien.

3.1.2 LXC: Erste Schritte in Container-Runtimes

Mit Linux Containers (LXC) kam 2008 die erste vollwertige Implementation einer Container-Runtime, die alleine mit dem nativen Linux-Kernel funktioniert (siehe Tabelle 5). Damit löste LXC eins der größten Probleme der vorherigen Lösungen, da kein gepachter Kernel benötigt wird.

Feature	Verwendung
Namespaces	Trennung unterschiedlicher Systemkomponenten zur Virtualisierung
Apparmor und SELinux	Mandatory Access Control
Seccomp	Sicherheitsprofile, sperrt Systemaufrufe des isolierten Prozesses
chroot und pivot_root	Isolation des Dateisystems und Nutzernamespace-Mapping
Capabilities	Entfernen von einzelnen Systemrechten
CGroups	Verwalten der Systemressourcen

Tabelle 5: Von LXC genutzte Kernel-Features (Graber, 2018)

Ein weiterer Vorteil, den LXC gegenüber älteren Technologien besitzt, ist die Einfachheit in der Benutzung. Im Gegensatz zu Linux VServern liefert LXC eine Konfigurationsdatei, vorgefertigte Seccomp oder SELinux Profile und ein Command Line Interface (CLI). Diese erlauben es, durch einfache Bash-Kommandos Prozesse in Containern zu isolieren. Dieser Vorteil ist allerdings auch ein großer Nachteil an LXC. LXC ist sehr systemnah und lässt sich durch die verwendeten Kernel-Features nur auf Linux-Systemen nutzen. Zudem ist die Konfiguration eines Containers weitreichender und komplexer als bei aktuellen Container-Runtimes.

3.1.3 CF Warden: Innovation durch Vereinfachung

2011 wurde die erste kommerziellen Container-Runtime mit Cloud Foundry (CF) Warden veröffentlicht. Diese basierte zu Beginn auf LXC und erweiterte diese mit eigenen Funktionen, wie einer REST API zur Steuerung und Verwaltung eines Container Clusters. Dabei setzt CF Warden auf eine Client-Server-Architektur (siehe Abbildung 10).

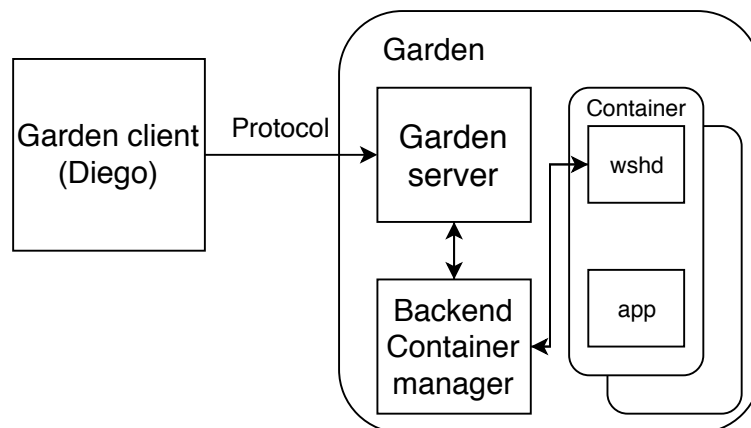


Abbildung 10: Client-Server-Architektur von CF Garden und Warden (Fedzkovich, 2016)

Im Gegensatz zu LXC ist CF Warden nicht mehr an Linux gebunden, sondern entkoppelt. Durch die Verwendung einer Bibliothek ist es möglich, Warden auch unter Windows Systemen zu verwenden. Warden kann als Container-Runtime für CF-Cluster genutzt werden, ist dort allerdings durch Garden, einer

Neuimplementation in der Sprache Go, ersetzt worden.

3.1.4 Let me contain that for you (LMCTFY) und Go: Googles Einfluss auf Container

CF Warden wird nahezu ausschließlich innerhalb eines CF-Clusters genutzt. Eine lokale Nutzung ist dabei nicht der Schwerpunkt. Für den Entwicklungsprozess und die Nutzung außerhalb einer CF Umgebung hat Google 2013 LMCTFY veröffentlicht. Dieser Service bildet Googles Container-Stack ab. LMCTFY wird nicht aktiv weiterentwickelt, die Kernkonzepte wurden allerdings in libcontainer übernommen und von Docker weiter entwickelt.

Eine weitere wichtige Entwicklung von Google, neben cgroups und LMCTFY, ist die Sprache Go. Diese findet in allen modernen Container-Runtimes Verwendung und ist eine der wichtigsten Entwicklungen der letzten 10 Jahre für Container. Dabei sind die Geschwindigkeit und Flexibilität von Go die wichtigsten Aspekte. Kompilierte Go-Programme laufen auf nahezu allen Linux-Systemen. Dabei sind die ausführbaren Binaries durch statisches Binden unabhängig von installierten Bibliotheken auf dem System. Zudem ist Go performanter als andere Sprachen.

3.1.5 Docker: Ecosystem, Runtime und Software-as-a-Service (SaaS) in einem

Im Jahr 2013 kam der größte Durchbruch für die Container-Technologie. Mit dem Release der Software Docker explodierten Container in Popularität und Nutzung. Wie auch CF Warden setzte Docker zu Beginn auf LXC. Der größte Unterschied zu bestehenden Container-Runtimes ist aber das Angebot als Ecosystem. Docker erlaubt es, Images aus dem Internet zu nutzen und bietet mit Docker Hub eine Plattform, die es sehr einfach ermöglicht, neue SaaS-Angebote für Kunden zugänglich zu machen. Zudem bietet Docker eine deutlich vereinfachte Form der Konfiguration mit Dockerfiles an. Durch die einfache Nutzung mittels CLI und der Möglichkeit, seine Services als Image bereitzustellen, gelang es Docker, unangefochten die meistgenutzte Container-Software zu werden.

3.2 Weiterführende Technologien

Anhand der Geschichte der Container-Technologien erkennt man, dass die heutige Innovation auf dem Gebiet der Isolation und Virtualisierung stark durch Container angetrieben wird. Vor allem Docker und Google, aber auch Microsoft oder Amazon treiben die Technologie an. Dabei wird immer ein Fokus auf die einfachere Nutzung, die Automatisierbarkeit einzelner Prozesse, sowie dem Bereitstellen oder kompilieren einer Anwendung gelegt.

Tabelle 6 gibt einen Rückblick auf die Geschichte der Container-Technologie. Dabei wird der zeitliche Hergang einzelner Funktionen und Runtimes in Bezug gestellt und aktuelle Themen aufgezeigt.

Benötigte Technologien	
1979	• Unix V7 mit chroot
1998	• SELinux
	• AppArmor
1999	• Linux Capabilites
2000	• FreeBSD Jails
2001	• Linux VServer
2002	• Linux namespaces
2004	• Solaris Container
2005	• Open VZ
2006	• Google Process Container
2007	• Process Container in Linux Kernel als cgroups
2012	• Erste stabile Version der Sprache Go
Container-Runtimes	
2008	• LXC
2011	• CF Warden
2013	• LMCTFY
	• CF Graden, Umstieg auf Go
2014	• Appc Spezifikation Release
	• Release rkt
2015	• Release LXD
	• Release runC
2016	• Windows Containers
	• CF Guardian Release, Support für runC
2017	• Release containerd v1.0.0
	• Release cri-o
Entwicklung Container-Ecosystem	
2011	• Initialer Release CF
2013	• Release Docker, erstes Container Ecosystem
2014	• Entwicklung K8s startet
2015	• Gründung CNCF und OCI
	• Docker Swarm
2016	• "Dirty Cow" → Container Sicherheit
	• Apache Mesos v1.0.0 Release
2017	• Übernahme rkt und containerd in CNCF
	• Release OCI runtime-spec und image-spec

Tabelle 6: Timeline Container-Technologien (Osnat, 2018)

4 Evaluation: Container-Runtimes

Container-Runtimes sind das Herz eines jeden Container-Angebots. Sie instan-
ziieren übergebene Prozesse in isolierten Containern. In diesem Kapitel werden
verschiedene Runtimes miteinander verglichen und veranschaulicht, wie andere
Runtimes neben Docker für spezielle Anforderungen besser geeignet sind.

4.1 Vorgehen

Um verschiedene Runtimes zu vergleichen wurde eine eigene Anwendung mit
drei Microservices implementiert. Dabei wurden, wie in Abbildung 11 zu se-
hen, verschiedene Technologien verwendet, um zu prüfen, wie die getesteten
Container-Runtimes mit diesen umgehen. Diese wurde im Folgenden mit ver-
schiedenen Runtimes lokal bereitgestellt.

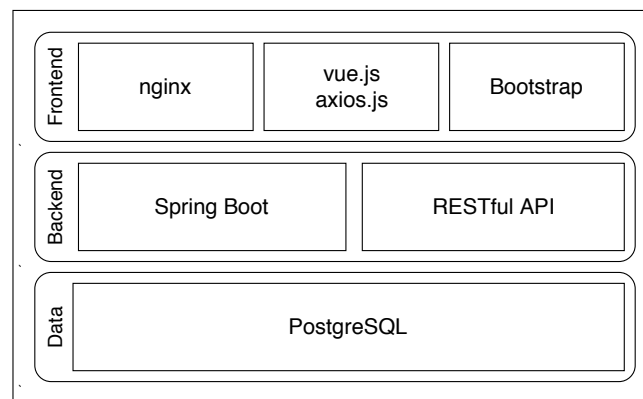


Abbildung 11: Beispielhafte Darstellung einer Microservice-Architektur

4.2 Docker Stack

Docker ist der de facto Standard unter den Container-Technologien und bietet eine vollständige Plattform zur Verwaltung und Orchestrierung von Container (Docker Swarm), der Verbreitung von Images (Docker Hub bzw. Docker Store) und der Verwaltung des Container-Lifecycles (Docker CLI) an.

Dabei kommen innerhalb des Docker Stacks die Runtimes **runC** und **containerd** zum Einsatz. Durch die in Abbildung 12 gezeigte Abkapselung der Runtime kann Docker auf jede beliebige OCI-konforme Runtime aufbauen.

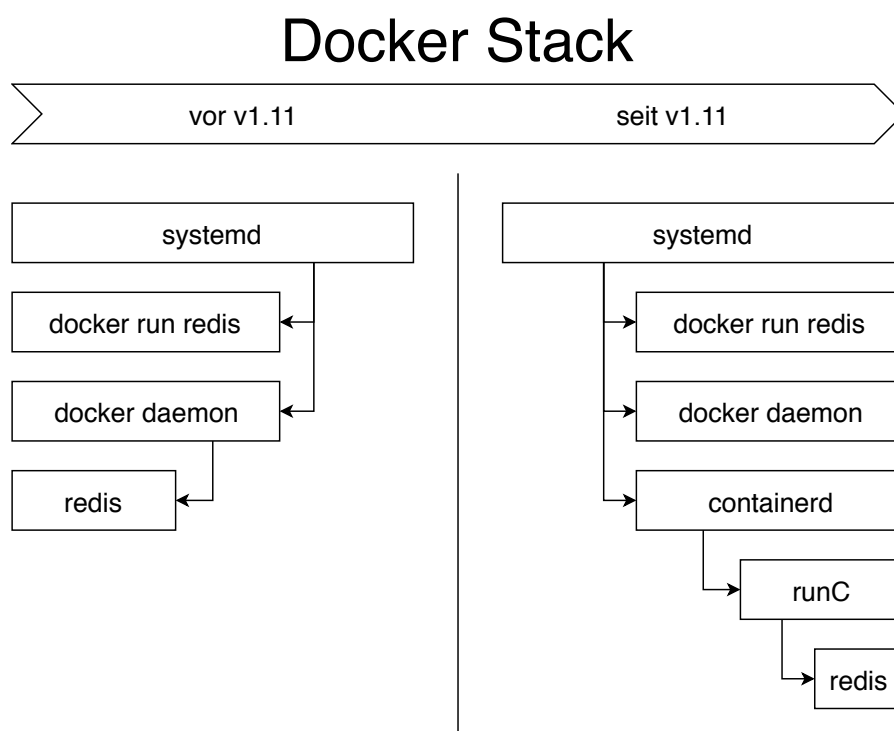


Abbildung 12: Docker Stack (Galeiras, 2017)

Der dadurch gewonnene Vorteil der Kompatibilität und Plattformunabhängigkeit bietet allerdings auch Nachteile. Falls einer der vielen Komponenten im Container-Stack einen Bug aufweist, ist das Debugging der Anwendung deutlich komplexer und Fehler können langsamer gefunden werden. Zudem benötigt

der Docker-Daemon privilegierte Berechtigungen um die in Abschnitt 2.2 beschriebenen Konzepte zu nutzen. Da der Daemon auch für den Download und Bau-Prozess der Images zuständig ist, werden alle Images in Docker im Kontext des Users `root` erstellt.

4.2.1 Vorgehensweise

Um den beispielhaften Microservice aus Abbildung 11 in Containern zu isolieren, wurde zuerst für jeden Service ein Image erstellt und diese dann in Containern gestartet. Um dieses Vorgehen zu erleichtern, wurde danach das Tool Docker Compose genutzt. Dieses erlaubt es, alle Container in einer Datei zu spezifizieren. Folgend werden beide Wege genauer beschrieben und Vor- bzw. Nachteile dieser Vorgehensweisen aufgezeigt.

Docker verwendet zur Beschreibung eines Images das Dockerfile. Dieses verwendet spezifische Keywords, um Docker beim Bau eines Images zu steuern (siehe Listing 7).

```
FROM nginx:1.13-alpine
VOLUME /tmp
ADD ./index.html /usr/share/nginx/html/index.html
EXPOSE 80
ENTRYPOINT ["nginx","-g","daemon off"]
```

Listing 7: Beispiel für ein Dockerfile

Dabei wird deklariert, von welchen Baseimage das neue Image erzeugt werden soll (**FROM**), welche Änderungen an diesem Image vorgenommen werden müssen (**VOLUME**, **ADD**, **EXPOSE**) und welchen Prozess der Container isolieren soll (**ENTRYPOINT**). Dieses Vorgehen erlaubt es einfach, neue Container auf Basis anderer Images zu erzeugen und diese für Dritte zu Verfügung zu stellen. Zu diesem Zweck bietet Docker den Docker Hub an, der als Standardrepository im Docker Stack verwendet wird. Dieser hostet mittlerweile mehr als 100 offizielle und über 100.000 Images, die aus der wachsenden Docker Community kommen. Um die in Abbildung 11 gezeigte Anwendung zu containern benötigt man somit drei Dockerfiles. Nach dem Anlegen der Dockerfiles kann mit dem Befehl `docker build -t <name of image>:<tag>`

<path to dockerfile> das entsprechende Image erstellt werden. Diese Images können mit `docker run <name of image>:<tag>` isoliert werden.

Vereinfacht wird dieser Prozess mit dem Tool Docker Compose. Dieses erlaubt es in einer YAML Ain't Markup Language (YAML)-Datei alle benötigten Container Images zu spezifizieren und mit dem Befehl `docker-compose up` zu starten (siehe Listing 8).

```
version: '3'
services:
  data:
    image: library/postgres
    environment:
      POSTGRES_USER: docker
      POSTGRES_PASSWORD: docker
      POSTGRES_DB: todos
  back:
    build:
      context: ./backend
    ports:
      - "8080:8080"
    environment:
      POSTGRES_PORT: 5432
      POSTGRES_IP: data
  front:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./frontend:/usr/share/nginx/html
```

Listing 8: docker-compose.yaml für Microservices

Der größte Unterschied liegt dabei in der Art und Weise, wie Container innerhalb des Compose-Clusters angesprochen werden können. Jeder Container bekommt zu seiner IP einen DNS Eintrag mit dem Namen des Service im `docker-compose.yml`. Dadurch lassen sich die Services einfacher miteinander verknüpfen, da die IP des Containers nicht bekannt sein muss.

4.2.2 Bewertung

Docker erlaubt es die Anwendung einfach in Containern bereitzustellen. Dafür ist vor allem die große Auswahl bereits bestehender Images verantwortlich, aber auch die einfache Nutzung durch Docker Compose. Das Format für Dockerfiles ist zwar einfach, aber gerade für Unixsysteme ungewöhnlich und nur mit Dokumentation nutzbar. Die Vorteile der einfachen Nutzung kommen allerdings zu einem Preis: Sicherheit. Images auf dem Dockerhub sind nicht verifizierbar, können somit schädliche Software und Sicherheitsprobleme mit sich bringen. Zudem baut Docker jedes Image unter dem Nutzer root, wodurch potentielle Sicherheitsrisiken, wie falsch konfigurierte AppArmor Profile, nicht beim Bau-Prozess auffallen. Viele dieser Probleme kommen durch die in Abbildung 12 gezeigte Architektur, bei der die Docker-CLI nur ein Client ist, der den Docker Daemon steuert. Diese Herangehensweise erlaubt keine Integration mit bereits vorhandenen Linux Tools wie `systemd` oder `upstart`.

Ein weiterer Nachteil gegenüber anderen Runtimes ist die Pflicht eines Image-Repositories. Um Docker Images zu teilen und zu verbreiten wird ein solches zwanghaft benötigt und muss somit gehostet, gewartet und konfiguriert werden. Um diese Aufgaben zu erleichtern bieten Google, Amazon, CoreOS und weitere gehostete Container Image-Repository an.

4.3 rkt

CoreOS veröffentlicht mit rkt den aktuell größten Konkurrenten zu Docker. Dieser setzt, wie in Abbildung 13 zu sehen, auf ein deutlich vereinfachtes, linuxartiges Prozessmodell. Zudem ist rkt auf die Integration mit anderen Linux Tools wie `systemd` konzipiert, mit dem sich der Container überwachen und steuern lässt.

4.3.1 Vorgehensweise

Im Gegensatz zu Docker bietet rkt die Möglichkeit verschiedene Imageformate zu nutzen um Container zu erstellen. Neben dem Dockerformat können auch OCI-konforme Bundles und App Container Images (ACIs) ausgeführt werden.

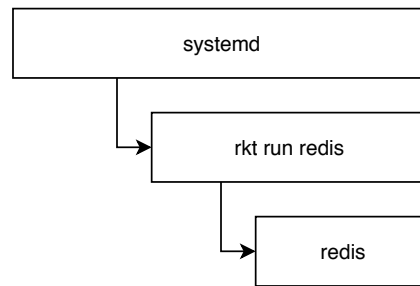


Abbildung 13: rkt Prozess Modell (Galeiras, 2017)

Um den beispielhaften Microservice aus Abbildung 11 bereitzustellen, wurde für jeden Service ein ACI erstellt. Zu diesem Zweck wird das Tool `acbuild` benötigt, welches ähnlich der Syntax eines Dockerfiles, einzelne Befehle zur Spezifizierung des ACIs nutzt (siehe Listing 9).

```
acbuild --debug begin
acbuild --debug set-name example.com/nginx
acbuild --debug dep add quay.io/coreos/alpine-sh
acbuild --debug run apk update
acbuild --debug run apk add nginx
acbuild --debug port add http tcp 80
acbuild --debug mount add html /usr/share/nginx/html
acbuild --debug set-exec -- /usr/sbin/nginx -g "daemon off;"
acbuild --debug write --overwrite
↪ nginx-latest-linux-amd64.aci
```

Listing 9: Bash Script um ACI mit `acbuild` zu erstellen (Crequey, 2018)

Ein Container mit dem daraus resultierenden ACI kann mit dem Befehl `systemd-run rkt run --insecure-options=image nginx-latest-linux-amd64.aci` gestartet werden. Dabei fallen folgende Unterschiede zur Herangehensweise mit Docker auf:

1. `systemd-run`

Dieses Prefix wird benötigt um einen rkt Container im Hintergrund auszuführen, vergleichbar mit `docker run -d <image>`. Da rkt kein Daemon nutzt (siehe Abbildung 13), kann es mit bekannten und verbreit-

teten Linux Tools genutzt werden kann. Um den Container-Prozess zu überwachen und zu steuern wird das Initssystem **systemd** verwendet.

2. `--insecure-options=image`

rkt verlangt bei jedem Image, dass es von einer vertrauenswürdigen Quelle gebaut wurde. Dazu nimmt rkt an, dass jedes Image digital signiert ist. Da dies bei dem angelegten ACI nicht der Fall ist, kann dieses Verhalten mit der gegebenen Option ausgeschaltet werden.

Um Container miteinander zu verknüpfen, nutzt rkt das von der CNCF publizierte Container Network Interface (CNI). Dieses erlaubt es, mittels einer JSON-Konfiguration neue Routen für den Traffic zum Container zu spezifizieren (siehe Listing 10). Dadurch kann man, ähnlich wie bei Docker Compose, einzelnen Containern DNS Namen zuweisen und somit ohne das Wissen der IP Container verknüpfen.

```
{
  "cniVersion": "0.2.0",
  "name": "mynet",
  "type": "bridge",
  "bridge": "cni0",
  "isGateway": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.22.0.0/16",
    "routes": [{ "dst": "0.0.0.0/0" }]
  }
}
```

Listing 10: Beispielhafte CNI-Konfiguration

4.3.2 Bewertung

In vielen Punkten gleichen sich rkt und Docker. Bei beiden steht das Bereitstellen einer Anwendung im Mittelpunkt. Doch gerade was das Thema Sicherheit und die Linuxähnlichkeit angeht treffen unterschiedliche Ansätze aufeinander. CoreOS bewegt sich näher an dem für Linuxsysteme typischen Aufbau und

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="ac-discovery"
      ↪ content="example.com/hello
      ↪ https://example.com/images/
      ↪ {name}-{version}-{os}-{arch}.{ext}">
    <meta name="ac-discovery-pubkeys"
      ↪ content="example.com/hello
      ↪ https://example.com/pubkeys.gpg">
  </head>
</html>
```

Listing 11: index.html mit Metatags um ACIs bereitzustellen

bietet mit rkt Integrationen zu weitverbreiteten Tools, wie systemd, an. Zudem benötigt rkt keine privilegierten Berechtigungen und arbeitet bei richtiger Konfiguration vollständig im Nutzerkontext. Durch dieses Vorgehen ist eine Privilege-Escalation weniger wahrscheinlich. rkt lässt sich zudem granularer steuern, da kein Daemon genutzt wird. Diese Vorteile kommen allerdings zum Preis von mehr Konfigurationsaufwand. Wenn man Listing 7 und Listing 9 vergleicht wird man feststellen, dass das Erstellen von Images mit **acbuild** komplexer und aufwändiger ist. Zudem wird neben rkt das Tool **acbuild** benötigt, da rkt selber keine Images erstellen kann, sondern lediglich eine Runtime bietet. Weiterreichend ist die Konfiguration des Netzwerks mittels CNI umfangreicher aber auch komplexer.

Ein großer Vorteil gegenüber Docker ist die Art und Weise, wie man Images teilen und verbreiten kann. Für ACIs ist kein private gehostetes Imagerepository notwendig. Es reicht lediglich ein Webserver und einige Metatags in der index.html (siehe Listing 11). Dadurch entfällt der Konfigurationsaufwand, der mit dem Hosten eines Repositories anfällt.

4.4 LXD / LXC

2008 kam mit LXC die erste vollwertige Implementierung einer Container-Runtime auf den Markt. Diese erlaubte es, ohne Veränderung des Kernels, Prozesse zu isolieren. 2015 wurde mit LXD eine Erweiterung von LXC veröffentlicht, die zu LXC eine RESTful API anbietet. Im Gegensatz zu Docker und rkt ist LXD dazu gedacht, komplette Betriebssysteme in Containern bereitzustellen und nicht einzelne Applikationen. Dadurch sieht sich LXD nicht als direkte Konkurrenz zu Docker, sondern als komplementäre Technologie, um mehr Sicherheit und Virtualisierung zu bieten (Canonical Ltd., 2018) (siehe Abbildung 14).

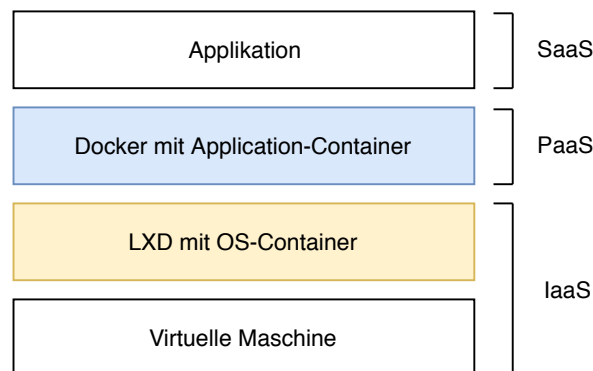


Abbildung 14: Docker Container in LXD Container

4.4.1 Vorgehensweise

Auch wenn LXD nicht hauptsächlich dafür gedacht ist, Applikationen und Services bereitzustellen, ist es möglich, diese mit LXD zu isolieren. Im Gegensatz zu Docker oder rkt können mit LXD mehrere Prozesse in einem Container isoliert werden. Um den in Abbildung 11 gezeigten Service mit LXD bereitzustellen, muss ein Baseimage ausgewählt, die benötigten Bibliotheken, Tools und Laufzeitumgebungen installiert und die erstellten Executables in das Dateisystem des Containers kopiert werden. Dazu kann man die mit LXD mitgelieferte CLI `lxc` nutzen (siehe Listing 12).

```
lxc launch ubuntu:16.04 todos
lxc file push /service/frontend/index.html
  ↪ /usr/share/nginx/html
lxc file push
  ↪ /service/backend/target/todos-backend-0.0.1-SNAPSHOT.jar
  ↪ /home/app.jar
lxc exec todos -- /bin/bash
# install and configure environment tools
# execute postgres, app.jar and nginx
exit
```

Listing 12: Shellbefehle um LXD Container zu starten

Durch dieses Vorgehen ist die Anwendung vollständig isoliert vom Hostsystem. Allerdings ist sie von außen nicht aufrufbar, da LXD keine Änderungen am Netzwerk des Hostsystems vornimmt. Um einen Container von außen zugänglich zu machen, muss zusätzlich eine EthernetBridge auf dem Hostsystem konfiguriert und mit dem Container verknüpft werden.

4.4.2 Bewertung

Wie man an Listing 12 erkennt, erwartet LXD Wissen über die Bedienung von Linuxsystemen. Vor allem die Konfiguration des Containernetzwerks ist komplexer als bei Docker oder rkt. Da LXD für die Isolation von Betriebssystemen gedacht ist, gibt es keine Images für Runtimes oder SaaS-Angebote, sondern Images für einzelne Betriebssysteme wie Alpine Linux oder Debian. Der Fokus liegt dabei allerdings auf Ubuntu, das wie LXD auch von Canonical angeboten wird. Die Inter-Process-Communication (IPC) gestaltet sich bei LXD deutlich leichter, da mehrere Prozesse in einem Container isoliert werden können. Allerdings ist das Anbinden des Container an das Netzwerk des Hosts komplexer als bei Application Containern. LXD bietet hierfür keine automatische Konfiguration, wodurch Linux Know-How benötigt wird. Zudem hat das Isolieren der Services in einzelne Container den Vorteil, das gezielt die Aspekte der Anwendung skaliert werden können, die unter Last stehen.

Der größte Vorteil von LXD gegenüber Docker oder rkt ist die REST-API. Diese

erlaubt es, ohne Zugriff auf das Hostsystem Container zu steuern, Checkpoints zu erstellen und die Last eines Containers zu überwachen. Für diesen Zweck wird bei Docker ein Third-Party Orchestrierungstool, wie z.B. K8s benötigt.

4.5 runC

Wie in Abschnitt 2.1 beschrieben ist das Interesse für einen Standard im Container-Umfeld so groß wie nie. Aus diesem Grund haben sich die meisten Firmen in der OCI zusammengeschlossen und mit **runC** eine Implementierung des definierten Standards veröffentlicht. Diese findet bereits innerhalb Dockers (siehe Abbildung 12), wie auch bei K8s in Form von cri-o Anwendung.

4.5.1 Vorgehensweise

Im Gegensatz zu Docker, rkt oder LXD nutzt runC keine Images um einen Container zu spezifizieren, sondern ein Bundle bestehend aus 2 Dateien:

- `rootfs.tar`
Ein tarball, der als Wurzelverzeichnis des Containers dient
- `config.json`
Konfiguration im JSON Format, die beschreibt, wie der Prozess innerhalb des `rootfs` ausgeführt werden soll.

Wie ein `rootfs.tar`-Archiv erstellt werden kann wurde bereit in Abschnitt 2.3.1 beschrieben. Eine weitere Möglichkeit ist das exportieren aus Docker (siehe Listing 13)

```
docker export \$(docker create -v  
↪ /service/frontend:/usr/share/nginx/html nginx) | tar -C  
↪ rootfs -xvf -
```

Listing 13: Exportieren eines `rootfs` aus Docker Container

Um einen Container mit runC zu starten wird zusätzlich eine `config.json` benötigt.

Diese kann mit `runC spec` erzeugt werden. Die dadurch erstellte Datei beinhaltet eine vorgegebene Spezifikation, die in Listing 14 auszugsweise zu sehen ist. Beim Start eines Containers mit der gegebenen Konfiguration wird eine isolierte Shell gestartet. Um einen anderen Prozess, beispielsweise `nginx`, zu starten, muss ein entsprechendes `rootfs` erstellt und in der Konfiguration die Schlüssel `{"args": ["sh"]}` und `{"process": {"terminal": true,}}` geändert werden. Dadurch ist es auch möglich, entkoppelte Container zu starten, vergleichbar mit dem Docker-CLI-Argument `-d`.

```
{
  "ociVersion": "1.0.0",
  "process": {
    "terminal": true,
  },
  "user": {
    "uid": 0,
    "gid": 0
  },
  "args": [
    "sh"
  ],
  "env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/
    ↪ sbin:/usr/bin:/sbin:/bin",
    "TERM=xterm"
  ],
  "cwd": "/",
  "capabilities": {
    "bounding": [
      "CAP_AUDIT_WRITE",
    ],
  },
  "root": {
    "path": "rootfs",
    "readonly": true
  },
  "hostname": "runC",
  "mounts": [
    {
      "destination": "/proc",
      "type": "proc",
      "source": "proc"
    },
  ],
}
```

Listing 14: Auszug aus Standardspezifikation durch den Aufruf von `runC spec`

4.5.2 Bewertung

Die OCI bietet mit runC eine standardisierte Container-Runtime, die eine einfache API besitzt und von den meisten Cloud-Providern unterstützt wird. Dabei ist vor allem die Steuerung und Konfiguration durch die erstellte `config.json` einfach und verständlich. Zudem bietet runC die Möglichkeit, Container ohne root-Berechtigungen zu starten. Diese rootless Container sind deutlich sicherer, da sie die Isolation nicht durch Systemcalls des Linux-Kernels umgehen können.

Neben den Vorteilen sind die Nachteile an runC als standalone Container-Runtime allerdings groß. So werden für jeden Container zwei Dateien benötigt, die nicht von runC verwaltet werden. Außerdem bietet runC keine Repositories für bestehende Images an. Ein weiterer großer Nachteil ist das erforderliche Wissen über Linux Kernel-Funktionen wie Capabilities oder Namespaces. Im Gegensatz zu rkt unterstützt runC auch keine automatische Prüfung einer Signatur (Galeiras, 2017).

4.6 VM basierte Runtimes

Die Isolation durch Container kann umgangen und somit Zugriff auf alle weiteren Container bzw. Prozesse einer VM, bis hin zu der gesamten Infrastruktur, erhalten werden. Um potentiell unsicheren Code Dritter auf eigenen Servern auszuführen wurden vor Containern VMs genutzt. Diese sind allerdings, wie in Kapitel 2 erläutert, schwergewichtiger und ineffizient. Dieses Problem haben Firmen wie Intel, HyperHQ und Google erkannt und Lösungsvorschläge implementiert.

4.6.1 Kata Containers

Kata Containers ist eine Initiative von Intel und HyperHQ, die die Projekte Intel Clear Containers und runV zusammenlegt. Kata Containers vereint dabei die Performance von Containern und die Sicherheit von VMs. Um dies zu erreichen werden einzelne Container in einer eigenen VM gestartet und somit der Kernel für jeden Container isoliert (siehe Abbildung 15). Die gestarteten

VMs sind dabei auf das Minimum an Funktionalität reduziert und sind somit deutlich leichtgewichtiger als vollwertige VMs.

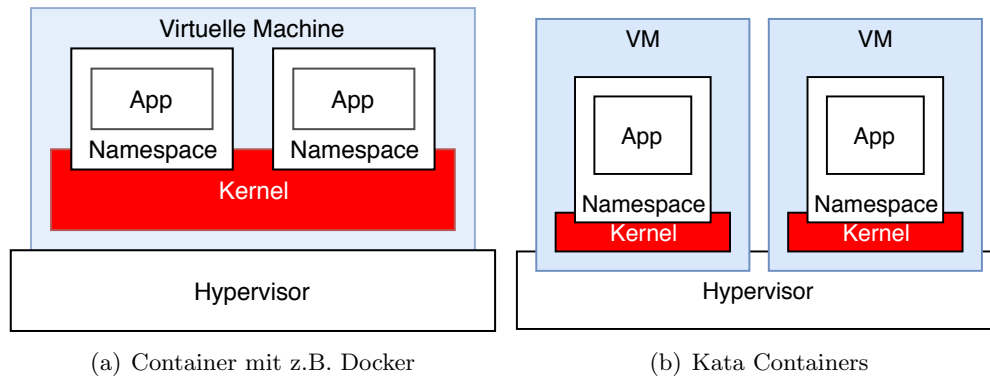


Abbildung 15: Kata Container im Vergleich zu Docker Container

Vorteile dieser Herangehensweise ist die gegebene Sicherheit innerhalb eines Containers. Diese ist durch die extra Isolation mittels einer VM vergleichbar mit selbiger und somit deutlich besser als bei Containern. Der größte Nachteil ist die langsamere Startzeit und der größere Footprint, da ein Extra Agent benötigt wird, um Kata Containers zu starten.

4.6.2 gVisor

Im Gegensatz zu Kata Containers wählt Google mit gVisor, welches Anfang Mai 2018 als Open Source Projekt veröffentlicht wurde, einen anderen Ansatz (Lacasse, 2018). Bei gVisor wird eine userspaced Implementation der meisten Systemaufrufe des Linux-Kernels gestellt. Diese werden alternativ zum Kernel des Hostsystems aufgerufen (siehe Abbildung 16). Durch diese Vorgehensweise ist es gVisor möglich, sicherer als Container und schneller als VMs zu sein.

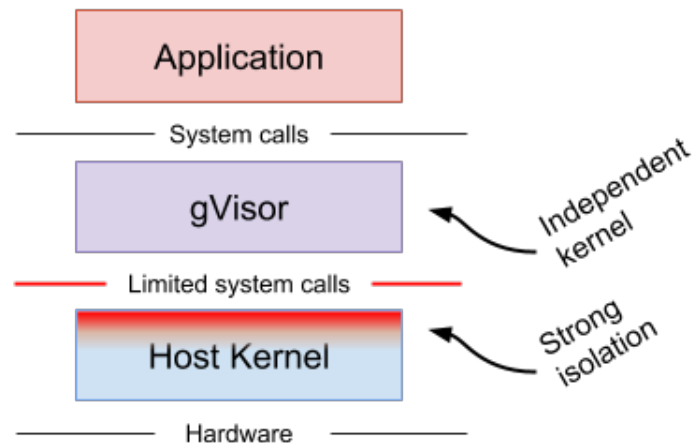


Abbildung 16: gVisor blockt Systemaufrufe zum Kernel ab (Lacasse, 2018)

4.6.3 Bewertung

Sicherheit spielt aktuell in der Cloud eine große Rolle. Angriffe wie Dirty COW im Jahr 2016 haben aufgezeigt, wie unsicher Container gegen gewissen Angriffe sein können (Oester, 2016). Technologien wie gVisor oder Kata Containers helfen bei der Isolation durch und bleiben dabei weiterhin skalierbar, allerdings auf Kosten der Performance.

4.7 Fazit

Die Auswahl an verschiedenen Container-Runtimes ist groß und wächst zunehmend. Dabei legen viele Runtimes seit 2015 Wert darauf, den von der OCI spezifizierten Standard zu erfüllen. Dadurch ist es zunehmend einfacher möglich, eine alternative Container-Runtime neben Docker zu wählen. Dabei bieten rkt, LXD oder gVisor verschiedene Vor- und Nachteile gegenüber Docker. Folgend werden die untersuchten Runtimes gegenübergestellt und eine Empfehlung für den in Abschnitt 4.1 geschilderten Sachverhalt gegeben. Dabei werden vor allem die folgenden Kriterien betrachtet:

- Sicherheit
- Einfachheit der Nutzung
- Bereitstellen von Images
- Konfigurationsaufwand

4.7.1 Sicherheit

Wie in Abbildung 17 zu sehen sind die in Abschnitt 4.6 erklärten VM basierten Ansätze deutlich sicherer als andere Runtimes. Sollte es gewünscht sein, potentiell gefährliche Software von unbekannten oder nicht vertrauenswürdigen Anbietern auszuführen, sollte definitiv auf eine Lösung wie Kata Containers oder gVisor gesetzt werden.

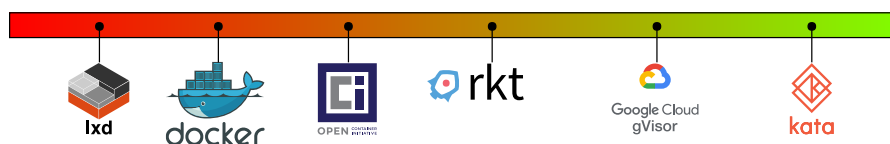


Abbildung 17: Gegenüberstellung der betrachteten Runtimes auf Sicherheit

4.7.2 Einfachheit der Nutzung

Abbildung 18 zeigt die Stärke Dockers gegenüber anderen Anbietern. Durch die große Community, die Anzahl der vorhandenen Images auf der SaaS-Plattform Docker Hub und die gut dokumentierte API ist Docker die am einfachsten nutzbare Container-Runtime. Die OCI-konformen Runtimes runC, Kata Containers und gVisor befinden sich etwa auf dem selben Level, da sie alle die selbe, standardisierte Vorgehensweise nutzen. Zum Zeitpunkt der Arbeit konnten allerdings nicht alle Services mit gVisor isoliert werden, da nicht alle benötigten Systemcalls implementiert waren. Im Zusammenspiel mit Docker hatten all diese Runtimes das selbe Level an Komfort wie Docker und konnten die gleichen Images nutzen. Dies liegt an der in Abbildung 12 gezeigten Architektur und der Austauschbarkeit von runC in dieser.

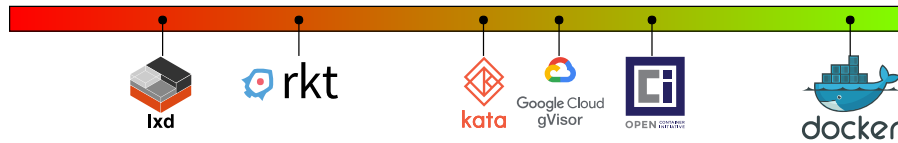


Abbildung 18: Gegenüberstellung der betrachteten Runtimes auf die Einfachheit der Nutzung

4.7.3 Verbreiten von Images

Abbildung 19 vergleicht das Bereitstellen innerhalb der einzelnen Runtimes. Das Bereitstellen von Images ist im Containerumfeld essentiell, um schnell Änderungen auf Cloud-Providern bereitzustellen. Die Container-Runtime rkt bietet dafür die einfache Möglichkeit der Meta-Discovery, mit der es einfach fällt, Images weiterzureichen. Da der OCI Standard nur wenig zur image-distribution spec aussagt ist es bislang nur schwer Möglich, ein Repository anzubieten, dass es einem ermöglichen würde, OCI Bundles zu teilen. Dazu kommt bislang der Docker Hub und Tools zur Konvertierung der Docker Images zum Einsatz. LXD bietet ein Repository für verschiedene Container-Images an, allerdings ist die Auswahl der Images sehr reduziert und es gibt keine Möglichkeit, ein bereits bestehendes, verändertes Images erneut zu nutzen, wie es mit Dockerfiles der Fall ist.

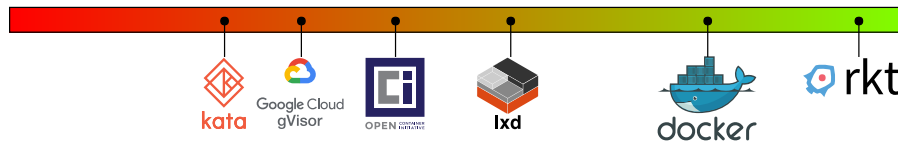


Abbildung 19: Gegenüberstellung der betrachteten Runtimes auf die Möglichkeit, Images zu teilen und zu verbreiten

4.7.4 Konfigurationsaufwand

Der Konfigurationsaufwand für das Betreiben der betrachteten Runtimes ist erheblich unterschiedlich (siehe Abbildung 20). Die Installation von Docker und rkt sind dank gestellter Scripte der Hersteller deutlich leichter als es bei Kata

Containers der Fall ist. Bei dieser Runtime muss zusätzlich zur Installation der CLI auch der benötigte Agent installiert und konfiguriert werden. Neben der Installation ist die Konfiguration des Containers nennenswert. Während Docker mit dem Dockerfile eine einfache Syntax zur Spezifikation eines Containers besitzt, sind bei anderen Runtimes, wie z.B. LXD, Linuxkenntnisse notwendig.

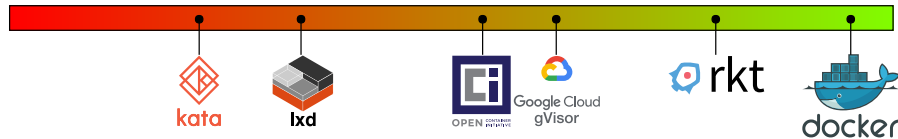


Abbildung 20: Gegenüberstellung der betrachteten Runtimes auf den benötigten Konfigurationsaufwand

4.7.5 Empfehlung

Je nach Einsatzzweck können unterschiedliche Runtimes Vorteile bieten. So sollten sicherheitsrelevante Anwendungen oder Services in sichereren Runtimes, wie z.B. in Kata Containers oder gVisor, isoliert werden. Falls man eine gute Allround-Lösung für das Bereitstellen einer service-orientierten Anwendung benötigt und keine sicherheitsrelevanten Daten nutzt, ist Docker die vermutlich beste Lösung. Die große Anzahl bereits bestehender Images und die aktive Community sind hierbei die Hauptgründe.

5 Aktuelle Themen

Durch den enormen Anstieg der Containernutzung im Cloud-Umfeld werden zunehmend Probleme ersichtlich, die teilweise der in Abschnitt 2.2 beschriebenen Funktionsweise geschuldet sind. Neben der Mitnutzung des Kernels und den damit verbundenen Risiken durch Bugs und sicherheitsrelevante Probleme ist auch die Verwaltung und Organisation der Container-Architekturen zunehmend schwieriger. Abseits dieser Probleme entstehen aber auch neue Innovationen rund um Container, z.B. die Serverless-Technologien. Hierbei werden einzelne Module auf Cloud-Plattformen wie AWS ausgeführt, ohne das Entwickler etwas von Containern und Konfigurationen wissen müssen.

Im folgenden Kapitel sollen aktuelle Probleme erläutert und potentielle Lösungen aufgezeigt werden. Zudem wird am Beispiel der Serverless-Technologie erklärt, wie Container aktuell Anwendung finden.

5.1 Security

”Container sind keine Sandbox”(Lacasse, 2018). Sandboxes sind ein Mechanismus, um laufende Programme zu separieren, damit Schwachstellen in einem Programm keine Auswirkungen auf andere Programme haben. Dies ist bei Containern nur bedingt der Fall. Schwachstellen in einzelnen Prozessen, die innerhalb eines Containers isoliert sind, betreffen weitestgehend nur den isolierten Prozess. Sicherheitslücken im Linux-Kernel verbreiten sich allerdings vom Host in jeden Container und werden nicht isoliert.

Ein Beispiel für solch eine Lücke, die genutzt werden konnte, um administrative Rechte innerhalb eines Containers und somit Zugriff auf den übergelegenen Host zu erhalten, ist Dirty COW. Dabei handelte es sich um eine Schwachstelle im Linux-Kernel, die innerhalb der Hauptspeicherverwaltung zu einer

Privilege-Escalation führte (Oester, 2016). Noch heute sind einige Systeme, die für Container als Host dienen, von dieser Schwachstelle betroffen.

Viele dieser Probleme lassen sich aufgrund der in Abschnitt 2.2 erklärten Funktionsweise von Containern nur schwer umgehen. Durch die größere Nutzung fallen allerdings immer mehr solcher Fehler im Linux-Kernel auf und können behoben werden. Um eine bessere Sicherheit zu gewähren, kommen mittlerweile vermehrt die in Abschnitt 4.6 beschriebenen VM-basierten Container-Runtimes zum Einsatz.

5.2 Orchestrierung

Google setzte bereits 2014 stark auf Container. So lief innerhalb Googles Cloud-Plattform alles auf Containerbasis und Google startete 2 Milliarden Container jede Woche (Beda, 2014). Diese massive Zahl an Containern führte zur dringenden Not einer Software, die all diese Container verwaltet, startet, stoppt und überwacht. Dies führte 2014 zum Release der Software Kubernetes (K8s), einer Orchestrierungsplattform für containerisierte Anwendungen. K8s erlaubt es, mittels einer YAML-Konfiguration, wie in Listing 15, ein Cluster an Containern zu spezifizieren, skalieren und bereitzustellen. Dabei verwendet Kubernetes das Konzept Pods, welche die kleinste Einheit bildet. Zu Beginn setzte K8s ausschließlich auf Docker Container, mittlerweile wird dies allerdings durch das Container Runtime Interface (CRI) vermieden. So können auch rkt, LXD oder beliebige andere Container-Runtimes in Clustern verwendet werden (siehe Abbildung 21).

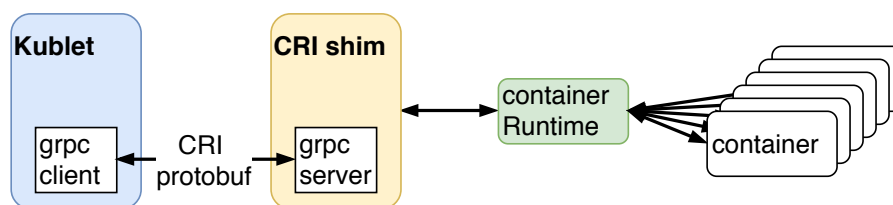


Abbildung 21: Container Runtime Interface Aufbau (The Kubernetes Authors, 2018)


```
apiVersion: apps/v1 # for versions before 1.9.0 use
↳ apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the
↳ template
  template: # create pods using pod definition in this
↳ template
    metadata:
      # unlike pod-nginx.yaml, the name is not included in
↳ the meta data as a unique name is
      # generated from the deployment name
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

Listing 15: Kubernetes YAML-Konfiguration für nginx Cluster (The Kubernetes Authors, 2018)

5.3 Serverless

Ein aktueller Anwendungsfall für Container ist die Serverless-Technologie. Egal ob Amazon Lambdas oder andere FaaS Anbieter, im Hintergrund dieser Technologie stehen immer Container, die genutzt werden, um einzelne minimale, zustandslose Funktionen auszuführen. Dabei muss sich ein Entwickler nicht um Container kümmern, diese werden vom FaaS-Provider aufgesetzt und orchestriert. Dieses Vorgehen hat Vorteile wie auch Nachteile für Entwickler und DevOps, die in Tabelle 7 gegenüber gestellt werden.

Vorteile	Nachteile
Kostenreduktion, da der FaaS-Provider nur die aktive Rechenzeit berechnet	Größere Applikationen, vor allem Monolithen, können die Vorteile von FaaS nicht nutzen
Weniger Unterhalt, da die Konfiguration sowie die Verwaltung der Container.Plattform dem Provider überlassen wird	Abgabe der Kontrolle über einzelne Container, die bei Serverless-Ansätzen nicht mehr gesteuert werden können
Reduktion des Overheads, da die Applikation die Serverless Funktion behandelt wie einen externen Service. Keine Instanziierung notwendig	Debugging, Monitoring und Testen der Anwendung im FaaS-Umfeld ist komplexer und teilweise unmöglich

Tabelle 7: Vor- und Nachteile der Serverless-Technologie (Churchman, 2017)

5.4 Fazit

Die Container-Technologie entwickelt sich rasant weiter. Seien es neue Virtualisierungsansätze wie gVisor, Standardisierungen wie runC oder neue Anwendungsfälle wie Amazon Lambda. All diese Entwicklungen sorgen für einen rasanten Anstieg der Technologie. Firmen wie Google haben diesen Trend bereits 2013 ins Rollen gebracht. Der Anstieg der Popularität sorgt zunehmend für die Not an Tools im Container-Umfeld. Mit Kubernetes, Docker Swarm oder Cloud-Foundry sind diese Tools mittlerweile ausgereift. Eine Sättigung der Technologie ist nicht abzusehen, der Trend geht eher zu einer größeren Nutzung (Kaczorowski, 2018). All diese Gründe machen es für Firmen wie Amazon, Google oder Microsoft unausweichlich, sich mit der Technologie und Vertretern der Branche auseinanderzusetzen. Dabei dürfen neben Docker, dem größten und bekanntesten Anbieter einer Container-Lösung, auch andere Vertreter wie CoreOS mit rkt nicht außer Acht gelassen werden.

Abbildungsverzeichnis

1	Container Isolation im Vergleich zu VMs	3
2	CNCF Container Runtime Landschaft (CNCF, 2018)	6
3	Auszug aus Dateisystem mit gemounteten Dateien	8
4	Netzwerk Topologie durch Trennung des Namespaces	9
5	Dateisystem nach erfolgreichem Bauprozess mit buildroot	12
6	Dateibaum nach entpacken der <code>rootfs.tar</code>	13
7	Python Webserver mit festgesetztem Root-Verzeichnis	13
8	Root-Eskalation durch Aufruf von <code>sudo chroot</code>	14
9	Ausgabe des Pythonprogramms <code>hungry.py</code>	16
10	Client-Server-Architektur von CF Garden und Warden (Fedzkovich, 2016)	19
11	Beispielhafte Darstellung einer Microservice-Architektur	23
12	Docker Stack (Galeiras, 2017)	24
13	rkt Prozess Modell (Galeiras, 2017)	28
14	Docker Container in LXD Container	31
15	Kata Container im Vergleich zu Docker Container	37
16	gVisor blockt Systemaufrufe zum Kernel ab (Lacasse, 2018)	38
17	Gegenüberstellung der betrachteten Runtimes auf Sicherheit	39
18	Gegenüberstellung der betrachteten Runtimes auf die Einfachheit der Nutzung	40
19	Gegenüberstellung der betrachteten Runtimes auf die Möglichkeit, Images zu teilen und zu verbreiten	40
20	Gegenüberstellung der betrachteten Runtimes auf den benötigten Konfigurationsaufwand	41
21	Container Runtime Interface Aufbau (The Kubernetes Authors, 2018)	43

Tabellenverzeichnis

1	Standards OCI und AppC im Vergleich (Polvi, 2015)	5
2	Cgroups-Controller und deren Verwendung (S. H. M. Kerrisk, 2018)	7
3	Linux Namespaces und verbundene Ressourcen (Biederman, 2017)	7
4	Einige Capabilities (M. Kerrisk, 2018)	10
5	Von LXC genutzte Kernel-Features (Graber, 2018)	18
6	Timeline Container-Technologien (Osnat, 2018)	22
7	Vor- und Nachteile der Serverless-Technologie (Churchman, 2017)	45

Listings

1	Entpacken des buildroot tarballs nach /home/rootfs	12
2	Mounten von Verzeichnis /readonly/ zu /rootfs/var/src/	14
3	Remount des PID-Namespaces und Chroot einer Shell	14
4	Erzeugen einer memory cgroup namens container	15
5	Limitieren des Arbeitsspeichers und Memory-Swap deaktivieren	15
6	Python Programm hungry.py um Arbeitsspeicher zu verbrauchen	16
7	Beispiel für ein Dockerfile	25
8	docker-compose.yaml für Microservices	26
9	Bash Script um ACI mit acbuild zu erstellen (Crequy, 2018)	28
10	Beispielhafte CNI-Konfiguration	29
11	index.html mit Metatags um ACIs bereitzustellen	30
12	Shellbefehle um LXD Container zu starten	32
13	Exportieren eines rootfs aus Docker Container	33
14	Auszug aus Standardspezifikation durch den Aufruf von runC spec	35
15	Kubernetes YAML-Konfiguration für nginx Cluster (The Kubernetes Authors, 2018)	44

Akronyme

ACI App Container Image.

appc App Container.

CF Cloud Foundry.

cgroup control group.

chroot Change Root.

CLI Command Line Interface.

CNCF Cloud Native Computing Foundation.

CNI Container Network Interface.

CRI Container Runtime Interface.

FaaS Function-as-a-Service.

IPC Inter-Process-Communication.

K8s Kubernetes.

LMCTFY Let me contain that for you.

LXC Linux Containers.

OCI Open Container Initiative.

OS Betriebssystem.

PID Process Identifier.

SaaS Software-as-a-Service.

VM Virtuelle Maschine.

YAML YAML Ain't Markup Language.

Quellen und weiterführende Literatur

- ARAVENA, Ricardo, 2018. What's Up With All The Container Runtimes. In: *What's Up With All The Container Runtimes. KubeCon Europe 2018* [online] [besucht am 2018-05-03]. Abgerufen unter: <http://sched.co/Dqtw>.
- BEDA, Joe, 2014. Containers at Scale. In: *Containers at Scale. GlueCon 2014* [online] [besucht am 2018-04-11]. Abgerufen unter: <https://speakerdeck.com/jbeda/containers-at-scale>.
- BIEDERMAN, Michael Kerrisk; Eric W., 2017. *namespaces(7) - Linux Manual Page*.
- CANONICAL LTD., 2018. *The LXD container hypervisor* [online] [besucht am 2018-06-06]. Abgerufen unter: <https://www.ubuntu.com/containers/lxd>.
- CHIANG, Eric, 2017. *Containers from Scratch* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://ericchiang.github.io/post/containers-from-scratch/>.
- CHURCHMAN, Michael, 2017. *Containers vs Serverless Computing* [online] [besucht am 2018-06-27]. Abgerufen unter: <https://rancher.com/containers-vs-serverless-computing/>.
- CNCF, 2017. *Cloud Native Computing Foundation* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://www.cncf.io/>.
- CNCF, 2018. *CNCF Cloud Native Interactive Landscape* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://landscape.cncf.io/>.
- CORE OS, 2017. *rkt 1.29.0 Documentation* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://coreos.com/rkt/docs/latest/>.
- CREQUY, Simone Gotti; Luca Bruno; Iago López Galeiras; Derek Gonyeo; Alban, 2018. *App Container* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://github.com/appc>.

- DOCKER INC., 2018. *Docker Documentation* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://docs.docker.com/>.
- FEDZKOVICH, Victoria, 2016. *Cloud Foundry's Garden: Back Ends, Container Security, and Debugging* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://www.altoros.com/blog/cloud-foundry-garden-back-ends-container-security-and-debugging-oss-cf/>.
- GALEIRAS, Iago López, 2017. *rkt vs other projects* [online] [besucht am 2018-05-15]. Abgerufen unter: <https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html>.
- GRABER, Daniel Lezcano; Christian Brauner; Serge Halryn; Stéphane, 2018. *lxc(7) - Linux manual page* [online] [besucht am 2018-04-09]. Abgerufen unter: <https://linuxcontainers.org/lxc/manpages/man7/lxc.7.html>.
- HARRINGTON, Brian "Readbeard", 2015. *Building minimal Containers: Getting Weird with Containers* [online] [besucht am 2018-06-28]. Abgerufen unter: https://github.com/brianredbeard/minimal_containers.
- HEO, Tejun, 2015. *Control Group v2*.
- KACZOROWSKI, Maya, 2018. *Exploring Container Security: detect and manage an attack* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://www.youtube.com/watch?v=go9Gadhy-R4>. Technischer Bericht. Google Cloud Platform.
- KERRISK, Michael, 2018. *capabilities(7) - Linux manual page*.
- KERRISK, Serge Halryn; Michael, 2018. *cgroups(7) - Linux Manual Page*.
- KNULST, Cornell, 2016. *Deep dive into Windows Server Containers and Docker* [online] [besucht am 2018-06-28]. Abgerufen unter: <http://blog.xebia.com/deep-dive-into-windows-server-containers-and-docker-part-1-why-should-we-care/>.
- LACASSE, Nicolas, 2018. *Open-sourcing gVisor, a sandboxed container runtime* [online] [besucht am 2018-05-15]. Abgerufen unter: <https://cloudplatform.googleblog.com/2018/05/Open-sourcing-gVisor-a-sandboxed-container-runtime.html>.
- MAS, Raphaël Hertzog; Roland, 2015. *The Debian Administrator's Handbook, Debian Jessie from Discovery to Mastery*. Freexian. ISBN 9791091414043.
- MATTHIAS, Karl; KANE, Sean P., 2015. *Docker: Up & Running: Shipping Reliable Containers in Production*. O'Reilly Media. ISBN 978-1-491-91757-2.

- MCGRATH, Roland, 2017. *chroot(1) - Linux Manual Page*.
- OESTER, Phil, 2016. *Dirty COW (CVE-2016-5195)* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://dirtycow.ninja/>.
- OPEN CONTAINER INITIATIVE, 2018. *Open Container Initiative* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://www.opencontainers.org/>.
- OSNAT, Rani, 2018. *A Brief History of Containers: From the 1970s to 2017* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016>.
- PETAZZONI, Jérôme, 2013. *Create Lightweight Containers with Buildroot* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://blog.docker.com/2013/06/create-light-weight-docker-containers-buildroot/>.
- POLVI, Alex, 2015. *Making Sense of Container Standards and Foundations: OCI, CNCF, appc and rkt* [online] [besucht am 2018-06-28]. Abgerufen unter: <https://coreos.com/blog/making-sense-of-standards.html>.
- PÖTZL, Herbert, 2011. *Paper - Linux-VServer*.
- ROUMELIOTIS, Rachel, 2016. The Open Container Essentials Video Collection. In: *The Open Container Essentials Video Collection. O'Reilly Open-Source Conference*. ISBN 9781491968253.
- RUSSINOVICH, Mark, 2015. *Containers in Windows Server, Hyper-V and Azure* [online]. Hrsg. von MECHANIC, Microsoft [besucht am 2018-06-28]. Abgerufen unter: https://www.youtube.com/watch?v=YoA_MMlGPRc.
- SARAI, Aleksa, 2018. *Rootless Containers* [online] [besucht am 2018-05-24]. Abgerufen unter: <https://rootlesscontaine.rs/>.
- THE FREEBSD DOCUMENTATION PROJECT, 2018. *FreeBSD Handbook* [online] [besucht am 2018-04-09]. Abgerufen unter: https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html.
- THE KUBERNETES AUTHORS, 2018. *Kubernetes Documentation* [online] [besucht am 2018-06-18]. Abgerufen unter: <https://kubernetes.io/docs/home>.
- ZAK, Karel, 2015. *mount(8) - Linx Manual Page*.