Überblick über Containervirtualisierung im Rahmen von Docker und der Open Container Initiative

Steven Solleder
Fakultät Informatik
Hochschule für angewandte Wissenschaften Hof
Hof, Deutschland
steven.solleder@hof-university.de

***Zusammenfassung*—Zuerst wird die Geschichte von Docker und der Open Container Initiative (OCI) betrachtet. Daraufhin wird erläutert, wie Docker technisch umgesetzt ist und was seine grundlegenden Funktionen sind. Danach wird Docker mit anderen Konzepten verglichen. Anschließend werden ausgewählte Einsatzzwecke beleuchtet. Zum Schluss wird noch ein kleines Fazit getroffen. Dabei kann gesagt werden, dass bei der Verwendung von Docker viele Abläufe neu gedacht und überarbeitet werden müssten, die Gewinne an Effizienz und Sicherheit jedoch groß sein könnten.**

***Index Terms*—Containervirtualisierung, Open Container Initiative (OCI), Docker, Images, Container, Volumes, Bind Mounts, Netzwerke, runC, containerd, dockerd, Compose, Contexts, Swarm**

I. Einleitung

A. Motivation

Früher wurde noch ein großer Teil der Daten lokal gespeichert und Software direkt auf einzelnen Rechnern ausgeführt. Mittlerweile ist Trend, Software auf Servern zu hosten und Daten auf Servern zu verwalten. Damit dies möglich ist, benötigen Unternehmen eine entsprechende Infrastruktur.

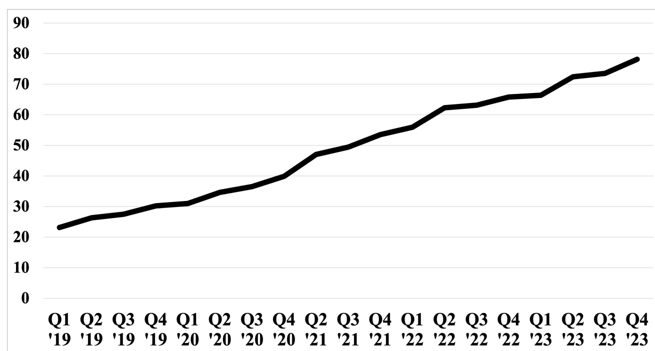

Abbildung 1. Ausgaben im Bereich Cloud-Infrastruktur-Services (IaaS/PaaS) weltweit (X-Achse: Zeit; Y-Achse: Ausgaben in Milliarden US-Dollar) [1]

Die gezeigte Grafik belegt diesen Trend sehr deutlich, indem diese zeigt, wie in den letzten fünf Jahren Quartal für Quartal mehr Geld für Cloud-Infrastruktur investiert wurde. Dabei stellt sich für jedes Unternehmen, welches auf Serverinfrastruktur Software hostet und Daten speichert, die Frage, wie teilweise mehrere Hundert Applikationen insbesondere im Hinblick auf Skalierbarkeit, Parallelisierbarkeit, Geschwindigkeit, Vermeidung von Kompatibilitätsproblemen, Verhinderung von komplizierten Abhängigkeitsbäumen, aber auch andere erstrebenswerte Ziele gemanagt werden können. Eine immer weiter verbreitete Lösung dafür ist die Nutzung von Containervirtualisierung, da diese im Gegensatz zu anderen Virtualisierungsmethoden zahlreiche Vorteile besitzt. [2]

B. Grundlegendes Konzept

Ein Container ist eine abgeschlossene Einheit, welche direkt auf einem Rechner ausgeführt wird. Was in ihm passiert, wird lediglich durch das ihm zugrunde liegende Image bestimmt. Ein Image beinhaltet alles zur Ausführung Erforderliche wie z. B. Quellcode, Laufzeitumgebung oder Bibliotheken. Images sind teilweise über öffentliche Verzeichnisse herunterladbar. Images selbst können wiederum auf anderen Images basieren. Damit mehrere Container - z. B. einer zur Ausführung des Backends und ein weiterer für die dem Backend zugrunde liegende Datenbank - untereinander kommunizieren können, schafft Containervirtualisierung eigene Netzwerke. Insofern es die Hardware zulässt, können beliebig viele auch identische Container gleichzeitig ausgeführt werden.

C. Ziel der Arbeit

Da die Software für Containervirtualisierung von Docker und der daraus entstandenen Open Container Initiative (OCI) unter allen Anbietern dieser Art von Software den größten Marktanteil besitzt und die OCI der vielversprechendste Containervirtualisierungsstandard ist, wird im Rahmen dieser Arbeit die OCI anhand von Docker näher beleuchtet. [3] Ziel der Arbeit ist es, zuerst die Geschichte hinter Docker und der daraus entstandenen Open Container Initiative näher zu betrachten. Daraufhin wird näher auf die Funktionen und Arbeitsweise von Docker eingegangen. Da der Großteil des Cloud-Computings mit Linux realisiert wird, wird sich auf die Linuxvariante konzentriert. Anschließend wird das hinter Docker liegende Konzept mit anderen Virtualisierungskonzepten verglichen. Kurz vor dem Ende sollen noch ausgewählte Einsatzzwecke von Docker vorgestellt werden. Zuletzt wird ein Fazit über Docker getroffen.

II. Geschichte

Nachdem ein grober Überblick über das Paper gegeben wurde, soll im Folgenden kurz die Geschichte der Open Container Initiative und Docker skizziert werden.

A. Gründung und Ziele der Docker Inc.

Ursprünglich wurde die Software für die Container-Software Docker als Projekt vom amerikanischen Unternehmen dotCloud entwickelt und erstmalig im Jahr 2013 am 15. März vom Gründer und Geschäftsführer des Unternehmens Solomon Hykes auf der Python Developers Conference in Santa Clara vorgestellt. Aufgrund der hohen Popularität des Projektes wurde dieses wenig später als Open Source auf der bekannten Plattform GitHub veröffentlicht, damit jeder es ausprobieren kann und auch andere Entwickler Verbesserungen hinzufügen können. Dabei wurde Docker als überwiegend positiv bewertet und als maßgeblicher Beitrag betrachtet, die Entwicklung und Distribution von Software zu verbessern. Um den Fokus auf Docker zu setzen, wurde zum einen das ganze Unternehmen noch im gleichen Jahr, in welchem die Docker-Software erschien, in Docker Inc. umbenannt. [4] Zum anderen wurde der sich im Besitz befindliche Platform-as-a-Service-Dienst dotCloud des gleichnamigen Unternehmens alsbald an die Firma Cloudcontrol verkauft. [8] Das Geschäftsmodell von Docker Inc. besteht dabei nicht darin, Lizenzen für Docker zu verkaufen. Stattdessen besteht es darin, kostenpflichtige Zusatzfunktionen für das Container-Register Docker Hub anzubieten und Abonnements für die Docker Desktop Software zu vertreiben. [6] Mitte 2014 bildete sich dann eine Vereinigung aus verschiedenen großen Technikfirmen, darunter Google, IBM, Microsoft, aber auch Docker selbst, um im Rahmen des „Kubernetes“-Projektes, welches von Google gestartet wurde, Docker Container auf jeglicher Infrastruktur laufen lassen zu können. [5]

B. Weiterentwicklung durch die Open Container Initiative der Linux Foundation

Um die Popularität von Docker weiter steigen zu lassen, wodurch auch langfristig das eigene Geschäftsmodell gefördert wird, entschied sich Docker Inc. mit einigen anderen führenden Unternehmen aus der Container-Branche wie zum Beispiel CoreOS im Jahr 2015, das Open Container Project ins Leben zu rufen. Dieses wurde später in Open Container Initiative umbenannt und der Linux Foundation unterstellt. Das Ziel der OCI ist das Pflegen und Fördern eines einheitlichen Container-Formats. Dies verhindert, dass andere inkompatible Container-Formate entstehen. Ebenso wird dafür Sorge getragen, dass der Container-Standard an keine bestimmte Software oder einen bestimmten Anbieter gebunden ist. Zur Förderung der Open Container Initiative stiftete Docker Inc. die Spezifikation und den Code seines Image-Formates. Ebenso spendete es seine Container-Runtime. Beide stellen die erste Implementierung der Image Specification (image-spec) bzw. Runtime Specification (runtime-spec) dar. Die Image Specification und Runtime Specification gehören beide zu den Open Container Initiative Specifications. [7] [9]

III. Aktueller Stand der Entwicklung

Nach der Betrachtung der Geschichte sollen in diesem zentralen Kapitel das Konzept und dessen Umsetzung näher angesehen werden.

A. Grundlegende Elemente

1) Images: Ein Image ist eine Art Vorlage, auf Basis derer Container erstellt werden. Man könnte das Image auch als Ausführungsdatei bezeichnen und den Container als Prozess. Ein Image kann nach seiner Fertigung nicht mehr verändert werden. Es bestimmt die grundlegende Funktionalität des Containers. Das Image kann sowohl ein allgemeineres Programm wie zum Beispiel ein Betriebssystem, eine Laufzeitumgebung oder eine Datenbank sein, es kann jedoch auch eine konkrete fertige Applikation sein. Um ein Image zu erstellen, muss man ein Dockerfile erstellen. In diesem wird mithilfe von 18 verschiedenen Befehlen beschrieben, wie ein neues Image erstellt werden soll. [18] [19]

2) Container: Ein Container ist ein Prozess. Das darin ausgeführte Programm wird alleine durch das Image bestimmt. Dieser Prozess ist von anderen Prozessen und anderen Containern durch verschiedene Mechanismen weitestgehend abgeschirmt. Durch die Verwendung von Volumes bzw. Bind Mounts, Netzwerke und weitere Elemente, welche allesamt explizit festgelegt und zugewiesen werden müssen, kann die Ausführung weiter beeinflusst werden. [18] [20]

3) Volumes und Bind Mounts: Zur Speicherung von Daten über den Lebenszyklus eines Containers hinaus, aber auch zum Bereitstellen von Daten für einen Container können sogenannte Volumes oder Bind Mounts verwendet werden. Beide haben jeweils ihre eigenen Vorzüge.

Volumes sind Speicher, welche vollständig von Docker verwaltet werden und von anderen Speichern abgeschirmt sind. Dementsprechend haben sie keine Nebeneffekte, je nachdem auf welchem Betriebssystem bzw. auf welchem Dateisystem sie sich befinden. Auch können sie ohne Weiteres einfach von mehreren Containern gleichzeitig genutzt werden, ohne dass es zu unbeabsichtigten Fehlern wie zum Beispiel dem Sperren von Dateien durch das Dateisystem kommt. Ebenso lassen sich Volumes vollständig durch die Docker CLI steuern. Dazu gehören insbesondere das Erstellen, Inspizieren und Löschen. Auch ist es möglich, Volume Treiber zu verwenden, welche den Volumes zusätzliche Funktionalitäten verleihen, ohne den Zugriff von Docker auf die Volumens zu beeinträchtigen. Ein wesentlicher Nachteil auf der anderen Seite ist die starke Abschirmung vom Betriebssystem. Ist nichts weiteres explizit festgelegt, erhält ein Container standardmäßig lediglich ein für ihn erstelltes anonymes Volume. [22] [23]

Bind Mounts sind Ordner im Dateisystem, auf welche Container Zugriff haben. Dadurch lässt sich das Innere von Bind Mounts über die üblichen Wege sehr einfach betrachten. In Folge dessen ist es auch wesentlich einfacher, Bind Mounts in Docker-unabhängige Flows einzubinden. Auf der anderen Seite fehlen ihnen die Vorteile, welche Volumes besitzen. [24]

Möchte man beispielsweise eine JAR-Datei innerhalb eines Containers ausführen, ist es zum einen möglich, ein eigens

dafür erstelltes Image zu verwenden. Zum anderen ist es möglich, nur ein Image für die Laufzeitumgebung, also ein Image, welches das Java Runtime Environment beinhaltet, zu verwenden und über ein Volume bzw. über ein Bind-Mount die JAR-Datei in den Container zu bringen.

4) Netzwerke: Die Erstellung spezifischer Netzwerke erlaubt es, die Kommunikation ausgewählter Container untereinander spezifisch zu kontrollieren. Dabei liegt jedem erstellten Netzwerk ein eigener Netzwerktreiber zugrunde, der die genauere Topologie bestimmt. Standardmäßig wird für ein neu erstelltes Netzwerk der Bridge-Netzwerktreiber verwendet. Auch hier ist es wieder möglich, neben den mitgelieferten Treibern eigene bzw. von anderen entwickelte Netzwerktreiber zu verwenden. So ist es zum Beispiel möglich, ein Netzwerk, welches den Bridgetreiber nutzt, zu erstellen und ausgewählte Container diesem hinzuzufügen, sodass ausschließlich diese Container untereinander kommunizieren können. Um von außen auf einen oder mehrere Container über das Netzwerk zugreifen zu können, müssen bei dem entsprechenden Container zusätzlich Portfreigaben getätigt werden. Standardmäßig wird ein Container einem Standard Bridge Netzwerk zugewiesen. [25] [26] [21]

Ein realistisches Beispiel, welches alle gerade genannten Grundelemente sinnvoll nutzt, wäre die folgende KI-Anwendung. Diese bietet verschiedene Endpunkte. An jeden Endpunkt können Bilder geschickt werden. Je nachdem, an welchen Endpunkt ein Bild geschickt wurde, werden andere Objekte auf dem Bild erkannt und deren Koordinaten im Bild zurückgeschickt. Um die Performance auswerten zu können, wird jeder Erkennungsvorgang geloggt. Ebenso soll es einfach möglich sein, das der KI zugrunde liegende Modell auszutauschen, um schnell verbesserte Modelle verwenden zu können.

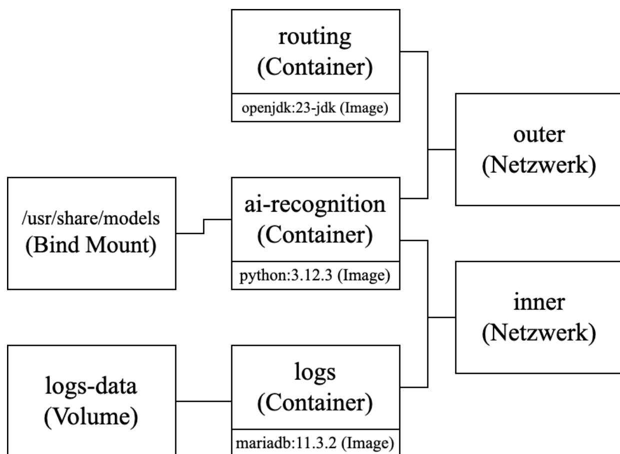

Abbildung 2. Struktur eines fiktiven Docker-Projektes

Um diese kleine KI-Anwendung umzusetzen, werden grundlegend drei Images verwendet, auf Basis derer dann Container gestartet werden. Eines, um innerhalb eines Java Runtime Environment eines Java Development Kits eine entsprechende JAR-Datei auszuführen, um die Anfragen zu routen („routing“). Ein weiteres, um Python-Code auszuführen, welches die Objekterkennung durchführt („ai-recognition“). Dieses bekommt mithilfe eines Bind Mounts Zugriff auf den Pfad „usr/share/models“, in welchem die zur Erkennung genutzten Modelle liegen. Durch das Bind Mount können die Modelle einfach ausgetauscht werden. Das Letzte dient schließlich dem Ausführen der Datenbank zum Loggen („logs“). Dieses erhält Zugriff auf ein Volume („logs-data“), damit andere Prozesse nicht ohne weiteres darauf zugreifen können und das Logging sichergestellt ist. Um zu erreichen, dass erhaltene Bilder nur von „ai-recognition“ genutzt werden, werden einzig „routing“ und „ai-recognition “ dem gleichen dafür geschaffenen Netzwerk („outer“) zugewiesen. Damit wirklich nur „ai-recognition“ loggen kann, werden ausschließlich „ai-recognition“ und „logs“ dem gleichen und speziell dafür erstellten Netzwerk („inner“) zugewiesen. Um sich bei der Abbildung auf das Wesentliche zu fokussieren, wird auf die Bind Mounts zum Zugriff der Container „routing“ bzw. „ai-recognition“ auf die JAR-Datei bzw. py-Datei verzichtet.

B. Technische Umsetzung der grundlegenden Elemente

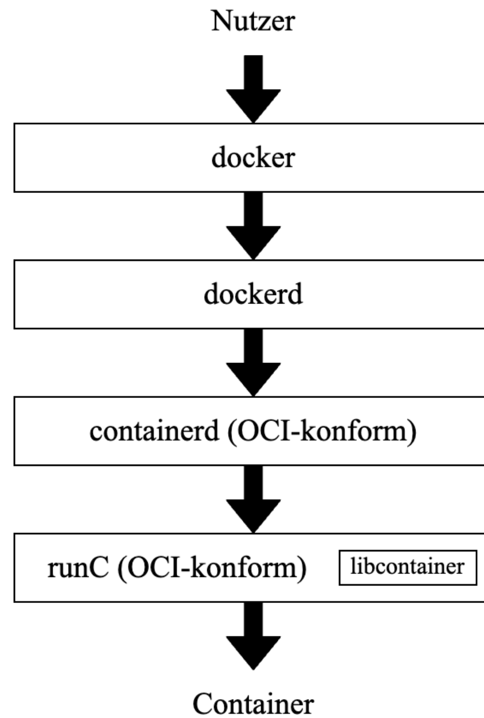

Abbildung 3. Docker Schichten

Docker besteht aus mehreren Schichten. Jede Schicht hat ihre eigenen Aufgaben:

1) runC: Wie am Anfang der Arbeit geschrieben, wird sich auf Docker im Rahmen von Linux beschränkt, weshalb die folgenden Ausführungen dementsprechend nur auf die Ausführung unter Linux zutreffen. Betriebssysteme, welche Docker nicht nativ unterstützen, sind zur Nutzung von Docker dazu gezwungen, eine virtuelle Maschine zu verwenden. Dies ist zum Beispiel bei dem Desktop-Betriebssystem macOS von

Apple der Fall. runC ist die Anwendung, welche die Container tatsächlich ausführt. Dabei setzt runC die OCI Runtime Specifications um. [9] Dazu nutzt runC die in der Programmiersprache Go geschriebene Bibliothek libcontainer. Eine wesentliche Eigenschaft der Ausführung ist, dass diese nativ auf dem Rechner stattfindet. Hierfür wird das zu containerisierende Programm erst einmal als ganz normaler Prozess gestartet. Um jeden Container zum einen vom Betriebssystem und seinen Prozessen und zum anderen von den anderen Containern abzuschirmen, nutzt libcontainer ausgewählte Technologien, welche den Kontext des Prozesses ändern. Zu diesen Technologien zählen an vorderster Stelle die Namespaces, die Control Groups (cgroups), das Union File System und das Setzen des Root directorys:

Für insgesamt acht Typen von Ressourcen können nahezu beliebig viele Namespaces erstellt werden. Zu diesen Ressourcen zählen Cgroup, IPC, Network, Mount, PID, Time, User und UTS. So wären beispielsweise die folgenden Namespaces mit den fiktiven Namen „namespace-ipc-0“, „namespace-ipc-1“ und „namespace-user-0“ denkbar. Innerhalb eines Namespaces eines bestimmten Ressourcentyps werden die Ressourcen vollständig unabhängig von anderen Namespaces des gleichen Ressourcentyps verwaltet. So können beispielhaft zur gleichen Zeit in zwei verschiedenen Namespaces des Ressourcentyps User jeweils ein Nutzer mit der User-ID 2 existieren. Jeder Prozess kann einem oder mehreren Namespaces zugewiesen werden. Standardmäßig gibt es für jeden der acht Ressourcentypen einen Namespace, wobei jeder neue Prozess, welcher ohne weitere Angaben erstellt wird, diesen acht Namespaces zugewiesen wird. Durch Namespaces ist es möglich, völlig neue autarke Systeme im Gesamtsystem zu schaffen, sodass Prozesse ohne Beeinflussung durch Nutzer bzw. durch das Gesamtsystem laufen können. [10] So ist es libcontainer möglich, den Container unabhängig vom konkreten Computer und dessen Einrichtung in der immer gleichen Umgebung zu platzieren, sodass der Container sich immer gleich verhält.

Die Control Groups erlauben es, Gruppen an Prozessen zu erstellen. Für diese kann genau bestimmt werden, welche Ressourcen bzgl. Leistung und in welcher Höhe diese zur Verfügung stehen. Zu diesen Ressourcen zählen insbesondere CPU-Zeit, Systemspeicher und Netzwerkbandbreite. Ebenso können die von einer Control Group genutzten Leistungsressourcen beobachtet werden. Eine weitere für Docker wichtige Eigenschaft ist, dass die Control Groups nicht zum Start des Betriebssystems feststehen müssen, sondern diese zur Laufzeit dynamisch erstellt, geändert und zerstört werden können. [11] Im Falle von libcontainer kann somit genau gesteuert werden, welcher Container welche Ressourcen in welcher Höhe erhält.

Ein Union File System ermöglicht es, Ordner von verschiedensten Orten, darunter sowohl vom gleichen Computer als auch von anderen über das Netzwerk erreichbaren Computern, in einem einzigen Dateisystem erreichbar und nutzbar zu machen. [12] Dadurch ist es für libcontainer möglich, mehrere Container auf Basis des gleichen Images zu erstellen, ohne dass die Daten des Images (zum Beispiel das Java Runtime Enviroment) für jeden neuen Container kopiert werden müssen.

Durch das Setzen des Root Directorys ist es möglich, dass ein privilegierter Prozess einen anderen Prozess erstellt und dessen Wurzelverzeichnis festlegt. Dies geschieht mithilfe des Syscalls chroot. Dem Prozess ist es dadurch verwehrt, insofern jener nicht selbst erhöhte Rechte besitzt, dieses Wurzelverzeichnis zu verlassen. Dies führt dazu, dass der sichtbare Bereich des Dateisystems für den Prozess entsprechend eingeschränkt wird. [13] Dadurch ist libocontainer befähigt einzuschränken, auf welche Dateien ein Container konkret Zugriff hat. [14]

2) containerd: containerd ist, wie der Name suggeriert, ein Programm, welches die containterübergreifende Kernfunktionen anbietet. Dazu zählen das Herunterladen und Managen von Images aus einem Register wie zum Beispiel Docker Hub. Ebenso gehört dazu das Definieren, Bearbeiten und Löschen von Netzwerk-Interfaces und anderer grundlegender Elemente, wobei die Realisierung dieser ausdrücklich runC obliegt. Auch gehört das Steuern und Verwalten von Containern über den ganzen Lifecycle hinweg mittels runC dazu. [15]

3) dockerd: Wie containerd ist auch dockerd ein Daemon-Programm. dockerd kann als eine Art Wrapper für containerd betrachtet werden. Auch wenn es zu containerd Alternativen gibt, kann dockerd ausschließlich containerd nutzen. Auf der einen Seite unterscheidet sich die Schnittstelle zwar von containerd, auf der anderen Seite ist die Funktionalität von dockerd im Vergleich zu containerd ziemlich ähnlich, wobei dockerd dennoch eine Übermenge von containerd ist. [16]

4) docker: docker schließlich ist die Software, welche für den Nutzer - in der Regel der Entwickler - gedacht ist, um alles, was Docker zu bieten hat, über die Kommandozeile mittels dockerd zu nutzen und zu steuern. Dazu stellt sie nutzerfreundliche und einfache Befehle zu Verfügung. [17]

C. Auf grundlegenden Elementen aufbauende Funktionen

Auf grundlegenden Elementen aufbauende Funktionen sind unabhängig von den Open Container Initiative Specifications und dienen dazu, die möglichen Anwendungszwecke zu erweitern.

1) Compose: Üblicherweise werden zu Ausführung von Aktionen, wie zum Beispiel dem Erstellen von Volumes oder dem Starten von Containern, entweder einzelne Befehle mithilfe des Docker CLI Programms „docker“ ausgeführt oder Nachrichten an den Docker Daemon „dockerd“ geschickt. Dies macht es insbesondere aufwendig, kompliziertere Setups aus speziell konfigurierten Containern, Volumes, Netzwerken usw. zu erstellen und mit anderen Menschen zu teilen. Compose macht es möglich, Setups in deklarativer Form innerhalb einer Datei, welche das YAML-Format nutzt, festzuhalten. Die Compose-Datei kann dann mithilfe des Docker CLI Programms ausgeführt werden. Dies bedeutet konkret, dass die in der Compose-Datei definierten Elemente erstellt und gestartet werden. Im Folgenden findet sich ein Beispiel für solch eine YAML-Datei, um sich ein Bild verschaffen zu können, wie diese aussieht. [27] [28]

```
services:
  routing:
    image: openjdk:23-jdk
    restart: always
    depends_on:
      - ai-recognition
    volumes:
      - ./routing.jar:/app/routing.jar
    networks:
      - outer
    ports:
      - "443:443"
    command: java -jar /app/routing.jar

  ai-recognition:
    image: python:3.12.3
    restart: always
    depends_on:
      - logs
    volumes:
      - /usr/share/models:./models
      - ./ai-recognition.py:
            /app/ai-recognition.py
    networks:
      - outer
      - inner
    environment:
      DATABASE_URL: mariadb://db:3306/logs
      DATABASE_USERNAME: logs
      DATABASE_PASSWORD: 1234
    command: python /app/ai-recognition.py

  logs:
    image: mariadb:11.3.2
    restart: always
    volumes:
      - logs-data:/var/lib/mysql
    networks:
      - inner
    ports:
      - "3306:3306/tcp"
    environment:
      MYSQL_DATABASE: logs
      MYSQL_USER: logs
      MYSQL_PASSWORD: 1234
      MARIADB_ROOT_PASSWORD: 12345678

volumes:
  logs-data:

networks:
  outer:
  inner:
```

Listing 1. Docker Compose Datei für das Beispiel aus Abbildung 2

2) Contexts: Mithilfe von Contexten ist es möglich, vom gleichen Computer aus verschiedenen Docker-Umgebungen zu steuern. Alle Informationen, um Zugriff zu einer Docker-Umgebung zu erhalten, werden innerhalb eines Contexts gespeichert. Dazu gehören konkret ein Name, Informationen zum Endpunkt des Docker Daemon und Infos zur TLS-Verschlüsselung. Dabei wird jeder Context in Form einer JSON-Datei im Homeverzeichnis des entsprechenden Nutzers gespeichert. Es können beliebig viele Contexte gespeichert werden. Mithilfe von Contexten ist es somit zum Beispiel möglich, auf dem lokalen Rechner ein Softwareprojekt zu entwickeln und dieses anschließend vom gleichen Computer aus auf einem anderen Server, auf welchem auch Docker ausgeführt wird, zu deployen. [30]

3) Swarm: Swarm ist eine in Docker eingebaute Funktion, um Container und weitere grundlegende Elemente in einem Swarm laufen zu lassen. Ein Swarm ist ein Cluster aus mehreren miteinander verbundenen Computern, wobei auf jedem dieser Computer Docker läuft. Dabei bietet er die Möglichkeit, die einzelnen Container je nach Wunsch des Administrators zu skalieren. Stürzen einer oder mehrere Container ab, kümmert sich Swarm darum, dass diese automatisch wieder hergestellt werden. Auch hostet Swarm einen DNS-Server, über welchen jeder Container die anderen Container über ihren Namen finden kann. Ebenso kümmert sich Swarm darum, dass die Kommunikation zwischen den Computern des Clusters mithilfe von TLS verschlüsselt wird. Durch diese genannten Funktionen, aber auch durch weitere, welche Swarm zu bieten hat, ist es leichter möglich, eine Anwendung skalierbar, sicher, ressourceneffizient und ausfallsicher zu hosten. [29]

IV. Vergleich mit anderen Konzepten

Um Docker nicht nur für sich kennenzulernen, sondern auch im Kontext der gesamten Informationstechnik-Landschaft, soll Docker in diesem Kapitel hinsichtlich Leistung, Portabilität, Effizienz und Isolierung mit ähnlichen Konzepten verglichen werden.

A. Eigenständige Rechner

Mit eigenständigen Rechnern sind physikalische autarke Rechner gemeint, welche im Elektronikfachhandel üblicherweise erworben werden können. Ein Rechner stellt in der Regel im Vergleich zu einem Container wesentlich mehr Leistung zu Verfügung, da er uneingeschränkt auf die eigene Hardware zugreifen kann. Der Rechner kann im Gegensatz zu einem Container nur physisch transportiert werden. Soll der Rechner dupliziert werden, muss zuerst ein Abbild der Festplatte gemacht werden und ein zweiter ähnlicher Rechner beschafft werden, auf welchem das Abbild wiederhergestellt werden kann. Dies benötigt viel Zeit und Speicher. Die Portabilität ist dadurch definitiv wesentlich eingeschränkter als bei einem Container. Befindet sich der Rechner unter dauerhafter Volllast, ist die Effizienz höher als bei einem Container. Schwankt die Last hingegen stark, können die Ressourcen des Rechners nicht dauerhaft ausgelastet werden, was letztendlich zu einer Ressourcenverschwendung und damit

zu einer wesentlich niedrigeren Effizienz führt, als hätte man die Last je nach Bedarf über mehrere gleichartige Container verteilt. Der Rechner ist genauso gut isoliert wie ein Container. Zugriff auf andere Ressourcen müssen dem Rechner physisch gegeben werden.

B. Virtuelle Maschinen

Virtuelle Maschinen sind ähnlich zu eigenständigen Rechnern mit dem Unterschied, dass sie auf einem eigenständigen Rechner simuliert werden. Eine virtuelle Maschine hat eine feste zugewiesene Menge an Leistung in Form von zur Verfügung stehenden CPU Kernen, Arbeitsspeicher usw.. Dies ist ähnlich zu Containern. Um virtuelle Maschinen auf einen anderen Rechner zu portieren, müssen die entsprechende Virtualisierungssoftware, die Festplatte der virtuellen Maschine, welche in der Regel eine Datei ist, und die Konfiguration der virtuellen Maschine auf den anderen Rechner verschoben bzw. kopiert werden. Die Portierbarkeit ist also gegeben, jedoch nicht so gut, wie es bei einem Container der Fall ist. Was die Effizienz angeht, verhält es sich wie bei dem eigenständigen Rechner. Jedoch können leichter viele kleine virtuelle Maschinen erstellt werden, sodass die Last genauer verteilt werden kann. Das Starten einer virtuellen Maschine benötigt mehrere Sekunden. Die Effizienz ist somit gut, aber nicht auf dem Niveau eines Containers. Die Isolierung entspricht der eines Containers, da der virtuellen Maschine die Ressourcen zielgenau zugewiesen werden können. Dabei ist die Konfiguration in der Regel aufwendiger bzw. komplizierter.

C. Sandboxing

Eine halbwegs anerkannte Definition einer Sandbox lässt sich schwer finden. Im Allgemeinen wird durch eine Sandbox versucht, ein Programm in seiner Ausführung von anderen Prozessen und dem System abzuschirmen. Genauere Eigenschaften der Sandbox hängen von der verwendeten Sandboxing-Software ab. So wird zum Beispiel in Android für jede Applikation ein eigener Nutzer erstellt, welcher diese dann ausführt, sodass die App nur die Rechte des Nutzers hat. Je nach dem, für welche Ressourcen der Nutzer welche Art von Rechten hat, kann die Anwendung mehr oder weniger tun. [31] Der Chromium Browser erzeugt zur Ausführung des JavaScript-Codes einer jeden Website einen eigenen Prozess. [32] So lässt sich letzten Endes sagen, dass ein allgemeiner Vergleich zwischen Sandboxing und Docker Container nicht getroffen werden kann, da Docker Container selbst nur eine spezielle Form des Sandboxings sind.

D. Prozessisolierung

Mit Prozessisolierung ist gemeint, dass ein normaler Prozess ohne spezielle Konfiguration zur Ausführung eines Programms verwendet wird. Welche Eigenschaften ein Prozess genau hat, richtet sich nach dem verwendeten Kernel, dementsprechend wird von den typischen Eigenschaften eines Prozesses ausgegangen. Ein Prozess muss sich seine CPU-Zeit mit anderen Prozessen teilen und hat je nach Anzahl und Auslastung der anderen Prozesse mehr oder weniger Leistung. Dabei ist es möglich, dem Prozess eine höhere Priorität zu geben. Einem Container hingegen kann genauer eine bestimmte Leistung zugewiesen werden. Portiert werden kann ein Prozess ähnlich wie ein Container, nämlich indem die Ausführungsdatei und die entsprechenden genutzten Daten verschoben bzw. kopiert werden. Zwar wäre es durchaus denkbar, mehrere Prozesse einer Anwendung zur effizienten Lastenverteilung zu erstellen, jedoch ist dies nicht immer möglich, insbesondere wenn die Prozesse die gleiche Ressource, zum Beispiel den gleichen Socket nutzen möchten. Obwohl Prozesse aufgrund weniger Einschränkungen für sich genommen effizienter sind als Container, sind sie wegen des schwierigen Nutzens für Lastenverteilung insgesamt weniger effizient als Container. Eine Isolierung ist nur insoweit gegeben, dass jeder Prozess seinen eigenen Bereich im Arbeitsspeicher erhält und nur in diesem schreiben kann. Andere Ressourcen wie Dateien oder Netzwerk teilen sich alle Prozesse. Die Isolierung ist dadurch schlechter als bei einem Container.

V. Ausgewählte Einsatzzwecke

Zuletzt sollen noch konkrete praktische Einsatzzwecke vorgestellt werden, um ein grobes Gefühl davon zu vermitteln, zu welchen vielfältigen Zwecken Containervirtualisierung genutzt werden kann.

A. Umsetzen einer Microservice-Architektur

Ein relativ neuer Trend in der Informatik bzw. der Softwareentwicklung ist die Nutzung von sogenannten Microservices. Diese sind das Resultat der Weiterentwicklung anerkannter Softwarearchitektur-Prinzipien wie zum Beispiel dem Single Responsiblity Prinzip und eine Folge des Trends, immer mehr Software in der Cloud anzubieten. Dabei sind Microservices kleine Programme, welche meist zwischen 100 und 1000 Zeilen Code benötigen. Ein solches läuft in seiner ganz eigenen Instanz der entsprechenden Laufzeitumgebung. Ebenso hat das Programm eine eigene Datenbank, insofern dieses einen persistenten Speicher benötigt. Wie sich aus dieser Beschreibung entnehmen lässt, bieten sich Container ideal zur Umsetzung von Microservices an. Dabei läuft die Laufzeitumgebung und ggf. die Datenbank eines Microservices in einem eigenen Container. Mithilfe von selbst erstellten Images lassen sich die Microservices zusätzlich versionieren. Die Datenbank speichert ihre Daten in einem Volume. Über Netzwerke und mithilfe von Swarm lassen sich diese ideal orchestrieren.

B. Bereitstellen einer einheitlichen Entwicklungsumgebung

Mit Entwicklungsumgebung sind alle Tools gemeint, welche für das tatsächliche Entwickeln von Software benötigt werden. Dazu zählen insbesondere ein Texteditor, eine Shell und ein Debugger. Ebenso gehören dazu auch jene Programme, welche genutzt werden, um die Software tatsächlich auszuführen, darunter der entsprechende Compiler bzw. Interpreter und die Laufzeitumgebung. Die meisten dieser Programme haben zahlreiche Einstellungsmöglichkeiten. Das gleiche gilt für das Betriebssystem, auf welchem diese Tools ausgeführt werden. Damit Entwickler beim Auftreten von Fehlern bei der

Softwareentwicklung nicht in Betracht ziehen müssen, dass diese durch ihr individuelles System verursacht werden, ist es sinnvoll, jegliche Entwickler Tools in Docker auszulagern und darüber zu nutzen. Ein weiterer Grund, welcher für dieses Vorgehen spricht, besteht darin, dass neue Softwareentwickler, welche zum Projekt hinzustoßen, direkt eine lauffähige Entwicklungsumgebung haben. Auch ist durch dieses Vorgehen die Entwicklungsumgebung wesentlich portabler und kann sehr einfach auf anderen Computern verwendet werden. Durch die Verwendung von Docker geht dabei kaum Performance verloren, sodass die Entwicklung nicht weiter eingeschränkt wird.

C. Testen von Anwendungen

Um zu Testen, ob eine Anwendung so funktioniert wie spezifiziert, benötigt es neben der Anwendung selbst noch automatische Tests, welche selbst wieder ein eigenes Programm darstellen. Ebenso werden häufig Mock-Objekte, wie zum Beispiel fiktive RESTful-Webservices benötigt, welche stets die gleichen Daten liefern. Um eine aussagekräftige Testdurchführung zu gewährleisten, muss sichergestellt werden, dass alle drei gerade genannten Teile frei von externen Störungen und damit deterministisch sind. Mithilfe von Docker wird die systemunabhängige Ausführung gesichert. Zusätzlich kann dabei z. B. durch Volumes, separate Netzwerke und Nichtnutzung von Portfreigaben garantiert werden, dass andere Prozesse die Anwendung, Tests und Mock-Objekte nicht von außen verändern.

D. Ausführen von Altsystemen

Altsysteme sind Programme, welche zwar noch in Gebrauch sind, aber nicht mehr aktiv weiterentwickelt werden. Dies bedeutet häufig, dass auch keine Schritte unternommen werden, die dem Programm zugrunde liegenden Bibliotheken, Laufzeitumgebung oder ähnliches zu aktualisieren. Sollen nun verschiedene Programme auf dem gleichen Computer ausgeführt werden, welche die gleiche Runtime bzw. die gleichen Bibliotheken nutzen, wobei sich diese lediglich in ihrer Version unterscheiden, resultieren daraus verschiedene Probleme. Zum einen ist es nicht immer möglich, verschiedene Laufzeitumgebungen bzw. Bibliotheken in verschiedenen Versionen auf dem gleichen Rechner zu installieren. Zum anderen ist nicht immer klar definiert, welche Programme welche konkrete Version einer Bibliothek bzw. Runtime verwenden. Dieses Problem lässt sich mit Docker recht einfach beheben. Dazu muss für jedes Programm ein eigenes Dockerfile geschrieben werden, welches beschreibt welches Programm mit welcher Laufzeit und mit welchen Bibliotheken in welchen Versionen genutzt werden soll. Daraus lässt sich dann ein Image erstellen, welches alles beinhaltet, was zur Ausführung nötig ist. Dieses kann dann vollständig unabhängig von anderen installierten Programmen in Form von Containern ausgeführt werden. So können beliebig viele Altsysteme ohne Konflikte gleichzeitig ausgeführt werden.

VI. Fazit

Es lässt sich urteilen, dass die Open Container Initiative und ihre Implementierung Docker samt dessen Zusatzfunktionen wie zum Beispiel Compose mit Kosten und Unsicherheiten für die Softwareentwicklung und -verteilung einherkommt, jedoch eine spannende Technologie mit vielen Potenzialen ist.
So muss auf der einen Seite gesagt werden, dass die Technologie noch recht neu ist und deshalb betrachtet werden muss, ob sich diese langfristig etabliert, durch eine alternative Lösung ersetzt wird oder gar nicht hält. Auch muss sich das Wissen zur Nutzung erst einmal angeeignet werden. Des Weiteren muss ein gewisser Aufwand betrieben werden, bestehende Abläufe mithilfe von Containern und Co. zu verbessern.
Auf der anderen Seite ist die Nutzung durch wenige Grundelemente einfach verständlich. Es wird eine weitestgehend ressourcenschonende Isolierung von Programmen und Steuerung dessen, was diese dürfen, ermöglicht. Dadurch können zahlreiche Abläufe und Schritte in den verschiedensten Szenarien stark vereinfacht, beschleunigt und sicherer gemacht werden, was somit die gesamte Softwareentwicklung an sich voranbringt.

Literatur

[1] Canalys. “Ausgaben im Bereich Cloud Computing weltweit für den Zeitraum vom 1. Quartal 2019 bis zum 4. Quartal 2023”. Statista. https://de.statista.com/statistik/daten/studie/1455690/umfrage/ausgaben-im-bereich-cloud-computing-weltweit/ (abgerufen 20.04.2024).
[2] Turbonomic. “State of containerization in organizations worldwide in 2021 and 2022”. Statista. https://www.statista.com/statistics/1223916/it-container-use-organizations/ (abgerufen 20.04.2024).
[3] Lionel Sujay Vailshery. “Leading containerization technologies market share worldwide in 2023”. Statista. https://www.statista.com/statistics/1256245/containerization-technologies-software-market-share/ (abgerufen 20.04.2024).
[4] Ben Golub. “DOTCLOUD, INC. IS BECOMING DOCKER, INC.”. Docker Blog. http://blog.docker.com/2013/10/dotcloud-is-becoming-docker-inc (abgerufen am 22.04.2024 über die Wayback Machine)
[5] Dipl. -Ing. Thomas Drilling, Ulrike Ostler. “Docker-Inspektion, Teil 1: Die Docker-Entstehung und-Einsatzgebiete”. Datacenter Insider. https://www.datacenter-insider.de/die-docker-entstehung-und-einsatzgebiete-a-750402/ (abgerufen am 22.04.2024)
[6] Sacra. “Docker-Inspektion, Teil 1: Die Docker-Entstehung und-Einsatzgebiete”. Sacra. https://sacra.com/c/docker/ (abgerufen am 22.04.2024)
[7] Redaktion ComputerWeekly.de. “Open Container Initiative”. ComputerWeekly. https://www.computerweekly.com/de/definition/Open-Container-Initiative (abgerufen am 22.04.2024)
[8] S. Kane, K. Matthias, “Praxiswissen Docker: Grundlagen und Best Practices für das Deployen von Software mit Containern”, Heidelberg dpunkt.verlag, 2024.
[9] Open Container Initiative. “OCI Certified”. Open Container Initiative. https://opencontainers.org/community/certified/#oci-certified-runtime-runtime-specification (abgerufen am 24.04.2024)
[10] Various Contributors. “namespaces(7) — Linux manual page”. man7. https://man7.org/linux/man-pages/man7/namespaces.7.html (abgerufen am 24.04.2024)
[11] Red Hat. “Chapter 1. Introduction to Control Groups (Cgroups)”. Red Hat. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01/ (abgerufen am 24.04.2024)
[12] Goldwyn Rodrigues. “Unifying filesystems with union mounts”. LWN.net. https://lwn.net/Articles/312641/ (abgerufen am 24.04.2024)
[13] Various Contributors. “chroot(2) — Linux manual page”. man7. https://man7.org/linux/man-pages/man2/chroot.2.html (abgerufen am 24.04.2024)

[14] John Skilbeck. “Docker: What’s Under the Hood?”. codementor. https://www.codementor.io/blog/docker-technology-5x1kilcbow (abgerufen am 24.04.2024)

[15] containerd. “containerd”. containerd. https://containerd.io/ (abgerufen am 24.04.2024)

[16] docs.docker. “dockerd”. docs.docker. https://docs.docker.com/reference/cli/dockerd/ (abgerufen am 24.04.2024)

[17] Vineet Kumar. “The differences between Docker, containerd, CRI-O and runc”. Medium. https://vineetcic.medium.com/the-differences-between-docker-containerd-cri-o-and-runc-a93ae4c9fdac (abgerufen am 24.04.2024)

[18] docker.docs. “Docker overview”. docker.docs. https://docs.docker.com/get-started/overview/ (abgerufen am 26.04.2024)

[19] docker.docs. “Dockerfile reference”. docker.docs. https://docs.docker.com/reference/dockerfile/ (abgerufen am 26.04.2024)

[20] docker.docs. “Running containers”. docker.docs. https://docs.docker.com/engine/reference/run/ (abgerufen am 26.04.2024)

[21] docker.docs. “Bridge network driver”. docker.docs. https://docs.docker.com/network/drivers/bridge/ (abgerufen am 26.04.2024)

[22] docker.docs. “Volumes”. docker.docs. https://docs.docker.com/storage/volumes/ (abgerufen am 26.04.2024)

[23] docker.docs. “docker volume”. docker.docs. https://docs.docker.com/reference/cli/docker/volume/ (abgerufen am 26.04.2024)

[24] docker.docs. “Bind mounts”. docker.docs. https://docs.docker.com/storage/bind-mounts/ (abgerufen am 26.04.2024)

[25] docker.docs. “Networking overview”. docker.docs. https://docs.docker.com/network/ (abgerufen am 28.04.2024)

[26] docker.docs. “Network drivers overview”. docker.docs. https://docs.docker.com/network/drivers/ (abgerufen am 28.04.2024)

[27] docker.docs. “Docker Compose overview”. docker.docs. https://docs.docker.com/compose/ (abgerufen am 28.04.2024)

[28] docker.docs. “Compose FAQs”. docker.docs. https://docs.docker.com/compose/faq/ (abgerufen am 28.04.2024)

[29] docker.docs. “Swarm mode overview”. docker.docs. https://docs.docker.com/engine/swarm/ (abgerufen am 28.04.2024)

[30] docker.docs. “Docker contexts”. docker.docs. https://docs.docker.com/engine/context/working-with-contexts/ (abgerufen am 28.04.2024)

[31] Contributors of the Android Open Source Project. “Application Sandbox”. Android Open Source Project. https://source.android.com/docs/security/app-sandbox (abgerufen am 30.04.2024)

[32] Contributors of The Chromium Projects. “Site Isolation”. The Chromium Projects. https://www.chromium.org/Home/chromium-security/site-isolation/ (abgerufen am 30.04.2024)