Wie kann ein automatisiertes Deployment von Docker Containern in der Cloud aussehen?

Sebastian Gaiser
Informatik-6
Matrikelnummer: 88256

***Zusammenfassung*—Um in der agilen Software Entwicklung Updates schneller ausliefern zu können, wird der Prozess des Deployments oft automatisiert. In diesem Paper wird das automatisierte Deployment von Software in der Cloud beschrieben. Dabei werden Docker Container verwendet, die nach einer Code Änderung in der Versionsverwaltung Git automatisch erstellt, getestet und in eine Container Registry gepusht werden. Je nach Konfiguration kann der erstellte Container auch sofort ausgeliefert und in der Cloud deployt werden. Dadurch können Updates von Applikationen schneller ausgeliefert werden und es kann agiler auf Veränderungen reagiert werden.**

***Index Terms*—Docker, CI, CD, Jenkins, Pipeline, Cloud**

I. Einleitung

In den letzten Jahren hat sich die Softwareentwicklung erheblich verändert. Immer mehr Unternehmen nutzen agile Methoden wie Scrum oder Kanban um schneller auf Änderungswünsche an Software eingehen zu können. Deswegen soll oft auch der Prozess des Deployments zum Beispiel bei einem Update eines Software Produktes zügiger und risikofreier ablaufen. Außerdem erlangt die Cloud eine immer größer werdende Bedeutung und Beliebtheit, da sie eine flexiblere und dynamischere Bereitstellung von Software ermöglicht.

Im Rahmen dieses Papers soll die Frage beantwortet werden, wie ein automatisiertes Deployment von Docker Containern in einer Cloud aussehen kann. Hierfür ist das Kennenlernen von kostenfreien Tools wichtig, welche ein einfaches und schnelles Deployment in der Cloud ermöglichen. Daher werden im Verlauf des Papers diese Tools und Grundkonzepte erläutert und beispielhaft angewendet.

II. Docker

Eines der wichtigsten Tools hierbei ist Docker. Docker bietet eine Möglichkeit Software plattformunabhängig, dynamisch und skalierbar zu deployen, da Docker Container standardisiert sind. Somit sind sie optimal geeignet, um auch in der Cloud laufen zu können.

Bei Docker handelt es sich um eine Open Source Containervirtualisierung, welche 2013 veröffentlicht wurde und auf *Linux Containers* (LXC) zurück geht. Mittlerweile ist Docker unabhängig von LXC und wird seit 2015 von der *OCI* (Open Container Initiative) [1] weiterentwickelt. Docker verwendet den Linux Kernel des darunter liegenden unixoiden Betriebssystem und dessen dazugehörigen Funktionen wie *Cgroups* und *namespaces* zur Isolation von Prozessen.

Dabei beschreibt ein Container im Normalfall genau einen Prozess, wie beispielsweise das fortwährende Ausführen von Binärdateien für eine Webapplikation, wie zum Beispiel *App 1* in Abbildung 1.

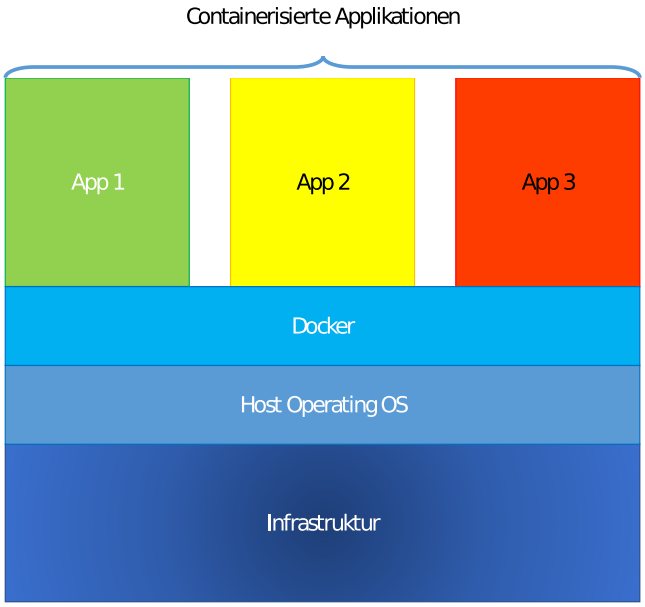

Abbildung 1. Aufbau Docker, [2]

Dadurch ergibt sich eine bessere Prozess Isolation, sowie eine effizientere Auslastung der Infrastruktur des Hosts. Herkömmliche Virtualisierungen wie z.B. VMware ESX(i) oder Microsoft Hyper-V führen meist mehre-

re virtuelle Maschinen, mit mehr als einem einzelnen Prozess parallel aus, wie nachfolgend in Abbildung 2 ersichtlich ist. Der Hypervisor muss dadurch meist mehrere, unterschiedliche Betriebssysteme der virtuellen Maschinen zur selben Zeit ausführen. Da die Prozesse bei Docker jedoch isoliert ablaufen, ist hier nur ein Betriebssystem nötig, was wiederum dazu führt, dass sich die Prozesse nicht so leicht gegenseitig blockieren können.

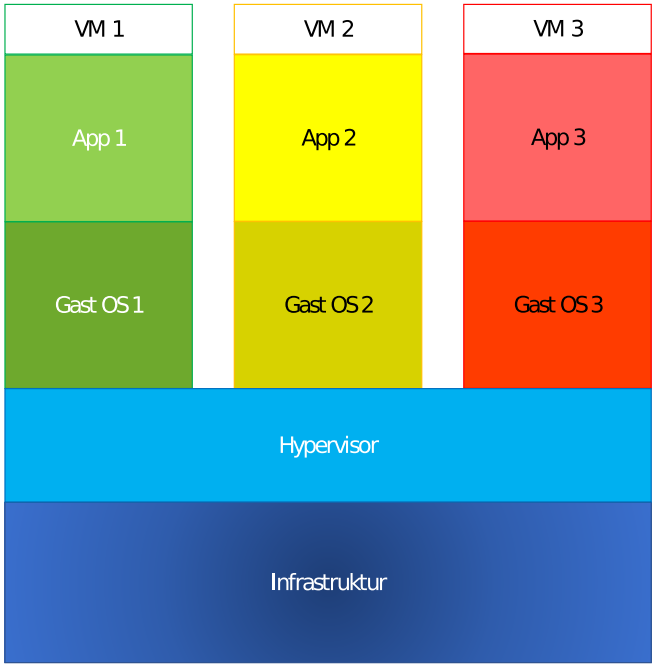

Abbildung 2. Aufbau herkömmliche Virtualisierung, [2]

Der modulare Ansatz mit Docker Containern eignet sich besonders für, die immer beliebter werdenden Microservices, die im Gegensatz zu monolithischen Software Architekturen dynamischer und unabhängiger gestaltet werden können.

Gerade die Vorteile der Modularität sind für ein automatisiertes Deployment sehr hilfreich. Außerdem ist es mehr oder weniger Grundvoraussetzung, da Docker Container in wenigen Sekunden bereitgestellt und auch ohne einen Reboot auf einen anderen darunterliegenden Host verschoben werden können.

A. *Dockerfile und Docker Build*

Um einen Docker Container zu erstellen, wird ein Docker Image benötigt, welches mittels eines Dockerfiles erstellt wird. Ein Dockerfile besteht aus einzelnen Layern (`FROM`, `RUN`, `COPY`, ...), die zwischengespeichert werden können und dadurch die Build Zeit verringern. Die Layer werden am Ende des Build Prozesses in einem einzelnen Image vereint.

```
FROM node:10

RUN mkdir -p /app
WORKDIR /app
COPY src /app/src
COPY package*.json ./
RUN npm install
ENV PORT 3000
EXPOSE ${PORT}

CMD [ "node", "src/server.js" ]
```

Listing 1. Beispiel Dockerfile

Einfache Dockerfiles basieren auf einem Basis Image, welches in Listing 1 `node` mit der Version `10` ist. Danach werden einzelne Schritte, wie das Anlegen des Verzeichnisses `/app` mittels des ersten `RUN` Befehls (vgl Listing 1) oder das Kopieren (`COPY`) (vgl Listing 1) von Dateien und Verzeichnissen in den Container ausgeführt. Um nun einen Build eines Docker Images anzustoßen, wird auf der Kommandozeile folgender Befehl ausgeführt:

```
docker build <aktuelles Verzeichnis>
```

Listing 2. Kommando docker build

Somit ist nach einem erfolgreichen Ablauf ein Docker Image mit einer ID erstellt, jedoch ohne Namen und Version.

B. *Docker Image Tagging und Registry Push*

Um daraus ein wiederverwendbares Docker Image zu erstellen, wird das zuvor erstellte Image mit einem Namen und einer Version versehen.
Dies geschieht mittels dem Befehl `tag`:

```
docker tag <Image ID aus docker build>
    ↪ <Name>:<Version>
```

Listing 3. Kommando docker tag

Nach dem erfolgreichen Benennen des Images, kann es nun in eine für Docker Images bereitgestellte Registry gepusht werden. Dadurch können sich andere Nutzer das Image wieder herunterladen, um es selbst zu nutzen. Alternativ kann auch ein automatisiertes System das Image auf eine bestehende Infrastruktur deployen. Dazu wird der Befehl `push` aufgerufen:

```
docker push <Name des Images>:<Version>
```

Listing 4. Kommando docker push

Somit wurde ein lauffähiges Docker Image erstellt und in einer Registry zur Verfügung gestellt.

C. Erzeugen eines neuen Docker Containers

Um nun einen laufenden Container mit dem zuvor gebauten Image zu erhalten, benutzen wir folgenden `run` Befehl:

```
docker run -d -p
    ↪ <hostPort>:<containerPort> --name
    ↪ <Container Name> <Name des
    ↪ Images>:<Tag bzw Version>
```

Listing 5. Kommando docker run

Dadurch wird der Container mit einem eigenen Namen und dem zuvor erzeugten Image erstellt. Außerdem wird der Prozess automatisch in den Hintergrund verlagert (`-d`) und die Ports des Containers auf den Host weitergeleitet (`-p`).
Um eine Liste aller laufenden Container zu erhalten, kann der `ps` Befehl genutzt werden:

```
docker ps
```

Listing 6. Kommando docker ps

Wenn der Container erfolgreich gestartet ist, wird er in der Liste aufgeführt und ist somit betriebsbereit.

Die zuvor beschriebenen Schritte, wie die Erstellung eines Docker Images und das Erstellen eines Docker Containers, können auch automatisiert werden. In der Industrie werden Docker Container oft im Zusammenhang mit Cloud Computing genannt.

III. Cloud Computing

Cloud Computing, oder kurz Cloud, bedeutet, dass bei Bedarf jederzeit über das Internet Ressourcen bereitgestellt und genutzt werden können. Dies geschieht über einen geteilten Ressourcenpool (zum Beispiel Netze, Server, Speichersysteme, Anwendungen und Dienste), welche durch Konfigurationen an die eigenen Wünsche angepasst werden können. Durch diese Konfigurationen können die Ressourcen besonders schnell und mit wenig Serviceprovider-Interaktion zur Verfügung gestellt werden.

A. Arten von Cloud Computing

Zunächst muss sich jedoch für eine Art von Cloud Computing entschieden werden, da es unterschiedliche Ausführungen gibt. Hauptsächlich unterscheiden sie sich in der Art der Bereitstellung und der Architektur, mit den dazugehörigen Vor- und Nachteilen. Zudem unterscheiden sich die Angebote je nach Anbieter bzw. Betreiber.

1) Public Cloud: Bei einer *Public Cloud* werden entsprechende Dienste und Services von einem Anbieter der Allgemeinheit zur Verfügung gestellt. [3]

2) Private Cloud: Eine *Private Cloud* wird dagegen von nur einer Organisation genutzt. Dabei kann Sie von der Organisation selbst oder von Dritten verwaltet und dadurch im eigenen Rechenzentrum oder im Rechenzentrum des Anbieters betrieben werden. [3]

3) Community Cloud: Bei einer *Community Cloud* werden die Services einer ganzen Interessensgruppe geteilt. Jedoch kann auch diese von der eigenen Institution oder von einem Anbieter bereitgestellt und betrieben werden.

4) Hybrid Cloud: Eine *Hybrid Cloud* ist eine Mischform aus der Public Cloud und der Private Cloud. Beide sind über technische Mittel zur gemeinsamen Nutzung von Daten und Anwendungen verbunden. In einer Hybrid Cloud können somit Daten und Anwendungen zwischen der Public und Private Cloud verschoben bzw. geteilt werden. Somit kann sich ein Unternehmen beispielsweise für einen kurzen Zeitraum zusätzliche Rechenressourcen anmieten, wodurch die Flexibilität erhöht wird. Dies trägt wiederum zur Optimierung der vorhanden Infrastruktur, Sicherheit und Compliance bei. [4]

B. Arten von Cloud Computing-Diensten

Cloud Computing-Dienste lassen sich außerdem in drei grundlegende Kategorien untergliedern: *Infrastructure-as-a-Service* (IaaS), *Platform-as-a-Service* (PaaS) und *Software-as-a-Service* (SaaS).

1) IaaS: Bei Infrastructure-as-a-Service handelt es sich um die Bereitstellung einer ganzen Infrastruktur. Das bedeutet, es werden dem Nutzer Rechenleistung, Datenspeicher oder ganze Netze als Service bereitgestellt. Der Kunde mietet solche virtualisierten und standardisierten Services an, um seine eigenen Dienste darauf betreiben zu können. [3]

2) PaaS: Bei Platform-as-a-Service wird dem Kunden lediglich eine Plattform mit standardisierten Schnittstellen zur Verfügung gestellt. Auf diese kann der Kunde mit seinen Diensten zugreifen und die darunter liegende Infrastruktur nutzen. Außerdem kann die Plattform Dienste zur Verfügung stellen, wie zum Beispiel Mandantenfähigkeit, Skalierbarkeit, Zugriffskontrolle und Datenbankzugriffe.

Der Unterschied zu IaaS besteht darin, dass der Kunde keinen Zugriff auf darunterliegenden Schichten hat, wie auf Betriebssystem und Hardware. Jedoch kann er auf der Plattform seine eigenen Anwendungen mit den vom Provider vorgegeben Tools betreiben. [3]

3) SaaS: Software-as-a-Service beschreibt alle Softwareangebote, die in der Cloud bereitgestellt werden können. Sämtliche eigenständige Applikationen können hier als Beispiel angesehen werden. [3]

C. Erweiterung der Cloud Computing-Dienste

Cloud Computing-Dienste lassen sich vielfältig um andere Varianten erweitern. Für das hier erläuterte Thema wäre das optimale Modell *Containers-as-a-Service* (CaaS).
Bei CaaS stellt ein Anbieter eine Plattform zur Verfügung, auf welcher Container zur Verfügung gestellt und verwaltet werden können. Der Fokus von CaaS liegt hauptsächlich auf dem Entwickeln von skalierbaren Applikationen, welche dann im Container betrieben werden können.
Im Vergleich mit anderen Cloud Computing-Diensten befindet sich CaaS zwischen den beiden Arten IaaS und PaaS. Da hier Teile der Infrastruktur durch den Container bereitgestellt werden, jedoch keine komplette Plattform zur Verfügung steht, in der nur noch der kompilierte Quellcode ausgeführt wird.
Einige populäre Anbieter bzw. Tools für CaaS sind unter anderem Amazon EC2 Container Service (ECS), Microsoft Azure Container Service (ACS), Rancher oder Docker Platform.
Für eine private Cloud (vgl. III-A2) kann man außerdem das Docker eigene Produkt *Docker Swarm* nutzen. Dabei werden einzelne Hosts zu einem Cluster zusammengeschlossen, auf dem dann die Container dynamisch verteilt bzw. orchestriert werden können.

IV. Automatisierung

Um nun eine Software bzw. einen Service automatisiert in der Cloud bereitstellen zu können ist ein Build- und Deployment-Prozess nötig.

Wichtige Schlagwörter hierbei sind die Methoden:

- *Continuos Integration* (CI)
- *Continuos Delivery* (CD)
- *Continuos Deployment* (CD)

Die Methoden der Automatisierung bauen aufeinander auf und sind daher stark voneinander abhängig. Sie lassen sich für ein automatisiertes Deployment in einer Pipeline abbilden.

A. Continuos Integration

In Continuos Integration führen Software Entwickler ihre Codeänderungen in einem gemeinsamen *Branch* zusammen. Im Anschluss wird der eingecheckte Code automatisch mit Hilfe von Tests überprüft (vgl. Abbildung 3). [5]

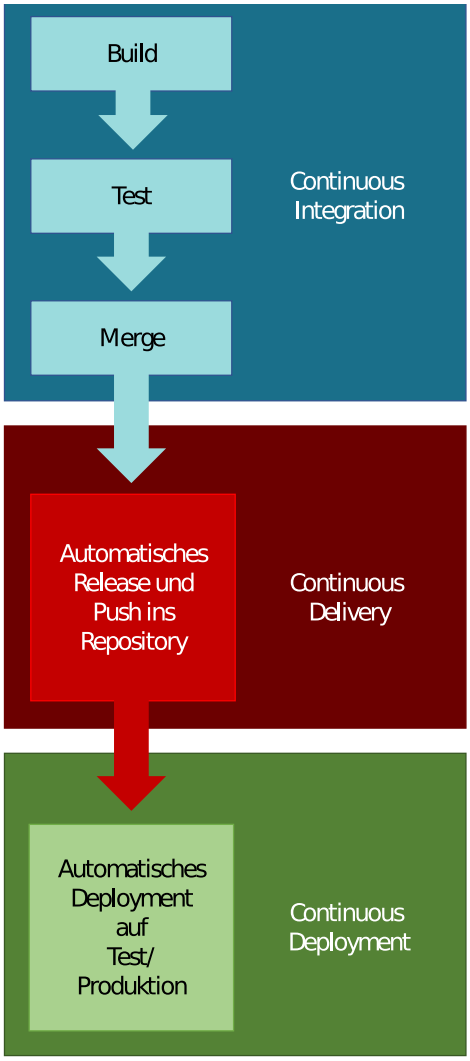

Abbildung 3. Ablauf Continuos Integration und Continuos Delivery und Continuos Deployment, [5]

Oft werden Unit- und Integrationstests verwendet, welche die Funktionstüchtigkeit des bestehenden Codes prüfen und sicherstellen, dass sich keine Fehler im Code eingeschlichen haben. Ein gängiges Mittel ist, dass alle Tests erfolgreich sein müssen um seine Änderungen in den Release Branch mergen zu können. Die Code Qualität kann darüber hinaus durch Systeme zur statischen Analyse und Bewertung der technischen Qualität überwacht werden, zum Beispiel von *Sonarqube*. [6] Diese können so genutzt werden, dass sie einen Merge in den Release Branch verhindern

können. Sind alle Test erfolgreich, wird der Code mit dem bereits zuvor freigegeben Code zusammengeführt (vgl. Abbildung 3).

Mit der Zusammenführung des Codes ist CI erfolgreich abgeschlossen und kann durch Continuos Delivery erweitert werden.

B. Continuos Delivery

Bei Continuos Delivery werden zunächst Funktions- bzw. Akzeptanztests ausgeführt. Sind diese erfolgreich, wird der Code zu einem Paket zusammengeführt (zum Beispiel in einem Docker Container) und anschließend an ein Repository oder eine (Docker-) Registry gepusht (vgl. Abbildung 3). Das Paket ist nun bereit um auf einer Umgebung, zum Beispiel auf der Testumgebung bereitgestellt und freigegeben zu werden. [5]

C. Continuos Deployment

Um Continuos Integration und Continuos Delivery zu komplettieren wird Continuos Deployment genutzt.

Hierbei gängige Praxis ist, dass der zuvor gepackte Code auf einer Testumgebung bereitgestellt und vom Auftraggeber getestet wird. Nach erfolgreichen Tests wird das Artefakt freigegeben und kann nun auf der Produktivumgebung, mit der in der Pipeline spezifizierten Ausführung, deployt werden (vgl. Abbildung 3).
Für das Freigeben des Deployments wird oft das *Vier-Augen-Prinzip* verwendet, zum Beispiel durch einen Entwickler und einen fachlichen Ansprechpartner.

Mit dem Live Gang des Artefaktes ist der gesamte Build- und Deployment-Prozess automatisiert durchgeführt worden.

Ein Vorgehen nach vollständigem CI/CD erleichtert die Integration von neuem Code, da sich riesige Releases durch viele Kleine verhindern lassen und diese daher weniger risikoreich sind. Außerdem wird die Code Qualität besser, da Fehler bei der Implementierung durch die automatisierten Tests schneller festgestellt werden können und daher umgehend behoben werden können. Die Erhöhung der Code Qualität wirkt sich wiederum positiv auf die Anwendungsstabilität aus.

V. Durchführung

Für die Automatisierung ist die Verwendung von einer Versionsverwaltung wie zum Beispiel *Git* unerlässlich, da hier der Source Code der einzelnen Entwickler zusammengeführt und versioniert werden kann.

In Kombination mit Git wird auch oft das Tool Jenkins verwendet. Hierbei handelt es sich um ein Open Source Software-System zur kontinuierlichen Integration und Automatisierung von Applikationen. Neben Jenkins gibt es weitere Tools zur Automatisierung des CI/CD-Prozesses. Die meisten verwenden, wie auch Jenkins, sogenannte Pipelines.

A. Pipeline

Eine Pipeline ist ein benutzerdefiniertes Modell eines CI/CD-Prozesses, welches iterativ abgearbeitet wird und Befehle beinhaltet, die einem Skript entsprechen. Der Code der Pipeline definiert den gesamten Build-Prozess, dieser umfasst die Phasen für das Erstellen bzw. Bauen einer Anwendung, das Testen und die anschließende Bereitstellung. Die Pipeline wird dabei im Optimalfall automatisch mittels eines sogenannten *Hooks* von der Versionsverwaltung ausgelöst. Ein Hook bemerkt eine Veränderung des Code Standes in der Versionsverwaltung und reagiert je nach Konfiguration zum Beispiel mit dem Anstoßen einer Pipeline.

Eine Pipeline kann dabei beispielsweise das Erstellen eines Docker Images von einer Applikation enthalten (vgl. Listing 7).

```
pipeline {
  agent any
      options {
          buildDiscarder(logRotator(
    ↪ numToKeepStr: '10',
    ↪ artifactNumToKeepStr: '10'))
          disableConcurrentBuilds()
          skipStagesAfterUnstable()
      }
  environment {
    REGISTRY_REPOSITORY = 'XXXXXXX/XXXXXX'
    REGISTRY_CREDENTIALS = 'dockerhub'
    dockerImage = ''
  }
  stages {
    stage('Cloning Git') {
      steps{
        checkout scm
      }
    }
    stage('Building Image') {
      steps{
        script {
          dockerImage = docker.build
    ↪ REGISTRY_REPOSITORY +
    ↪ ":$BUILD_NUMBER"
```

```
        }
      }
    }
    stage('Pushing Image') {
      steps{
        script {
          docker.withRegistry( '',
↪ REGISTRY_CREDENTIALS ) {
            dockerImage.push()
          }
        }
      }
    }
    stage('Cleanup') {
      steps{
        sh "docker rmi
↪ $REGISTRY_REPOSITORY:$BUILD_NUMBER"
      }
    }
  }
}
```

Listing 7. Beispiel Jenkins-Pipeline für das Erstellen und Pushen eines Docker Containers

B. Pipeline-as-a-Code

Jenkins benutzt seit der Version 2 das *Pipeline-as-a-Code* (PaaC) Prinzip. Dabei werden die Pipelines nicht über das User Interface konfiguriert, sondern als Code neben dem eigentlichen Quellcode hinterlegt. Jenkins verwendet dafür, die eigens entwickelte Sprache, *Groovy-DSL* (Domain Specific Language). Diese bietet sehr große Möglichkeiten, da sie per Plugins und eigenen Code erweitert werden können. [7]

Eine Jenkins Pipeline (as-a-Code) besteht aus einzelnen Abschnitten. Ein einzelner dieser Abschnitte wird *stage* genannt und unterteilt damit die Pipeline in kleinere Pakete, welche mit Bedingungen versehen werden können. Eine *stage* hat einen eindeutigen Namen, mit welchem sich die Schritte über das Logfile von Jenkins nachvollziehen lassen. Jede *stage* wird von einem *agent* ausgeführt, zum Beispiel ein anderer Docker Container. Zudem beinhaltet sie *steps*, welche Befehle auf der Kommandozeile (*sh*) oder ganze Skripte (*script*) sein können. Jede einzelne *stage* wird in der *stages* Sektion mit allen Anderen zusammengefasst. [8]

Die Pipeline (Listing 7) besteht aus vier Abschnitten (*stage*), *Cloning Git*, *Building Image*, *Pushing Image* und *Cleanup* und nutzt unter anderem das Docker Plugin für Jenkins.

- Cloning Git: Klont das im Jenkins spezifizierte Git Repository und führt im Hintergrund folgenden Befehl aus:

```
git clone <Repository>
```

- Building Image: Erstellt das Docker Image und führt die Befehle *docker build* (vgl. Listing 2) und *docker tag* (vgl. Listing 3) als *script* aus.
- Pushing Image: Pusht das Docker Image in eine Registry und führt den Befehl *docker push* (vgl. Listing 4) als *script* aus.
- Cleanup: Löscht das zuvor erstellte Docker Image im Hintergrund mit dem folgenden *sh* Befehl:

```
docker rmi <Name des
↪ Images>:<Version>
```

Durch die oben genannten Abschnitte wurden die Methoden Continuos Integration (vgl. IV-A) und Continuos Delivery (vgl. IV-B) erfolgreich durchgeführt (vgl. Abbildung 3).

C. Deployment

Um nun ein Deployment nach Continuos Deployment (vgl. IV-C) auf einem Server in der Cloud vorzunehmen, werden weitere Schritte benötigt, welche in der gleichen oder einer weiteren Pipeline ablaufen können. In diesem Beispiel wird eine eigene Pipeline angelegt, welche mit Hilfe eines Konfigurations-Management Tools das Deployment durchführt. Dafür empfiehlt sich hierbei der Einsatz von Ansible.

Ansible ist ein Konfigurations-Management Tool, welches zur Verteilung von Software genutzt werden kann. Dieses nimmt dann auf den Host-Maschinen Konfigurationen vor und führt dort Anweisungen aus. Standardmäßig verbindet sich Ansible dabei per SSH, jedoch sind auch andere Übertragungsmöglichkeiten nutzbar, wie beispielsweise der Zugriff auf den Docker Socket über die Docker API über welche Docker Container gesteuert werden. [9]

Ansible führt dort dann sogenannte Playbooks aus, welche die spezifizierten Konfigurationen durchführen. Als Grundvoraussetzung für die Ausführung von Ansible ist die Installation von Ansible selbst. Auf den Host-Maschinen auf denen lediglich die Playbooks angewendet werden, ist Python die einzige Voraussetzung. Dadurch muss auf den Host-Maschinen also keine spezielle Software installiert sein und es ist daher kaum Installationsaufwand nötig.

```
---
- hosts: all
  become: true
  vars:
    container_name: example
```

```
  container_image: ubuntu:latest
  container_command: sleep 1h
    container_restart_policy : no
  docker_network_name: default

tasks:
  - name: Pull Docker Image
    docker_image:
      name: "{{ container_image }}"
      source: pull

  - name: Stop Container
    docker_container:
      name: "{{ container_name }}"
      state: stopped

  - name: Remove Container
    docker_container:
      name: "{{ container_name }}"
      keep_volumes: true
      state: absent

  - name: Create Container
    docker_container:
      name: "{{ container_name }}"
      image: "{{ container_image }}"
      command: "{{ container_command
↪ }}"
        container_restart_policy :
↪ "{{restart_policy}}"
      networks:
        - name:
↪ "{{docker_network_name}}"
      state: started
```

Listing 8. Beispiel Ansible Playbook

Ein Playbook kann auf unterschiedlichen Gruppen oder Hosts über folgenden Befehl ausgeführt werden:

```
ansible-playbook <Playbook>
```

In diesem Beispiel wird das Playbook auf allen Hosts ausgeführt (vgl. Listing 8 - *hosts*). Nachfolgend werden einige Standard Variablen deklariert (vgl. Listing 8 - *vars*), welche jedoch vor dem Ausführen überschrieben werden können. Dies kann zum Beispiel beim Ausführen auf der Kommandozeile über die zusätzliche Option $--extra-vars$ erfolgen.

Das Playbook (Listing 8) besteht aus vier Schritten (vgl. Listing 8 - *tasks*), mit den Namen *Pull Docker Image*, *Stop Container*, *Remove Container* und *Create Container*, die jeweils genau ein Ansible Modul nutzen, welches die entsprechenden Operationen ausführt.

- Pull Docker Image: Nutzt das Ansible Modul *docker_image*, um das spezifizierte Docker Image aus dem Docker Hub (Registry) auf den Server herunterzuladen und führt im Hintergrund folgenden Befehl aus:

```
docker pull <Name des Images>:<Version>
```

- Stop Container: Nutzt das Ansible Modul *docker_container*, um den aktuell laufenden Container zu stoppen und führt im Hintergrund folgenden Befehl aus:

```
docker stop <Name des Containers>
```

- Remove Container: Nutzt das Ansible Modul *docker_container*, um den gestoppten Container zu löschen und führt im Hintergrund folgenden Befehl aus:

```
docker rm <Name des Containers>
```

- Create Container: Nutzt das Ansible Modul *docker_container*, um den neuen Container zu starten und führt im Hintergrund den Befehl *docker run* (vgl. Listing 5) aus.

In diesem Beispiel wurde gezeigt wie Docker Container auf Host-Maschinen mit Ansible unter Verwendung der Methode Continuos Deployment (vgl. IV-C) deployt werden können.

VI. Ausblick

Die Ausführung des beschriebenen automatisierten Deployments ermöglicht ein einfaches und beispielhaftes Bereitstellen von einzelnen Docker Containern.
Jedoch bietet Docker in Kombination mit dem Orchestrierungs-Tool Docker Swarm nur eingeschränkte Funktionen der Orchestrierung von Services. Daher empfiehlt sich für ein flexibleres und dynamischeres Deployment die Verwendung von *Kubernetes*. Kubernetes ist ein Open Source Container Orchestrierungs-Tool, welches initial von Google entwickelt wurde und bietet Unterstützung für vielfältige Container Technologien. Für die Verwendung ist jedoch zusätzlich ein Kubernetes Cluster notwendig, welcher von vielen Cloud Providern, wie zum Beispiel *Amazon Web Services* mit *AWS EKS*, zur Verfügung gestellt werden kann. [10]
Um Kubernetes Ressourcen vernünftig verwalten und versionieren zu können, empfiehlt sich der Einsatz von *Helm*. Helm ist ein Paket Manager für Kubernetes und ermöglicht durch Templates das Bereitstellen von Kubernetes Ressourcen. Die Templates verwenden hierzu die Programmiersprache *Go* bzw.

das Paket *templates*, welche auch *Go Templates* genannt werden. Die Templates werden in sogenannten Charts zusammengefasst, welche mit unterschiedlichen Variablen befüllt werden können. [11]

Zuletzt kann man Jenkins durch die extra für Kubernetes entwickelte Version *Jenkins X* ersetzen. Jenkins X bietet dadurch eine optimale Integration von Kubernetes Ressourcen bzw. die Verwendung von Helm Charts. Außerdem erleichtert Jenkins X die Entwicklung von Software, durch einen eingebauten Staging Prozess zum Beispiel von *Entwicklung* über *Test* bis in *Produktion*. [12]

VII. Fazit

Das Paper beantwortet die Frage, wie kann ein automatisiertes Deployment von Docker Containern in einer Cloud aussehen? Zu diesem Zweck wurden die verwendeten Tools und Grundkonzepte erläutert. Die Ergebnisse zeigen, dass sich mit Docker, Jenkins und Ansible ein einfaches und automatisiertes Deployment sowohl in der privaten, öffentlichen oder Hybrid Cloud durchführen lässt.
Dabei sind besonders die Pipelines wichtig, die die Methoden Continuos Integration, Continuos Delivery und den Continuos Deployment Prozess abbilden. Dadurch werden die manuellen Schritte für ein Software Update auf ein Minimum reduziert.
Ein automatisiertes Deployment wurde mit gängigen Tools durchgeführt, dass jedoch durch weitere Tools wie Kubernetes, Helm und Jenkins X noch optimiert werden kann.

Literatur

[1] OCI. *Open Container Initiative*. URL: https://opencontainers.org/ (besucht am 04. 06. 2020).

[2] Docker. *What is a Container?* URL: https://www.docker.com/resources/what-container/ (besucht am 19. 06. 2020).

[3] Bundesamt für Sicherheit in der Informationstechnik. *Cloud Computing Grundlagen*. URL: https://www.bsi.bund.de/DE/Themen/DigitaleGesellschaft/CloudComputing/Grundlagen/Grundlagen_node.html (besucht am 05. 06. 2020).

[4] Microsoft. *Was ist Cloud Computing?* URL: https://azure.microsoft.com/de-de/overview/what-is-cloud-computing/ (besucht am 05. 06. 2020).

[5] Red Hat. *Was versteht man unter CI/CD?* URL: https://www.redhat.com/de/topics/devops/what-is-ci-cd (besucht am 04. 06. 2020).

[6] Sonarqube. *Code Quality and Security*. URL: https://www.sonarqube.org/ (besucht am 27. 06. 2020).

[7] Philip Stroh Hendrik Brinkmann. *Der Praxischeck: Pipeline as Code mit Jenkins 2*. URL: https://jaxenter.de/pipeline-jenkins-2-61568 (besucht am 05. 06. 2020).

[8] Jenkins. *Pipeline Syntax*. URL: https://www.jenkins.io/doc/book/pipeline/syntax/ (besucht am 12. 06. 2020).

[9] Red Hat Ansible. *How Ansible Works*. URL: https://www.ansible.com/overview/how-ansible-works/ (besucht am 27. 06. 2020).

[10] Kubernetes. *Production-Grade Container Orchestration*. URL: https://kubernetes.io/ (besucht am 27. 06. 2020).

[11] Helm. *The package manager for Kubernetes*. URL: https://helm.sh/ (besucht am 27. 06. 2020).

[12] Jenkins X. *Accelerate Your Continuous Delivery on Kubernetes*. URL: https://jenkins-x.io/ (besucht am 27. 06. 2020).