HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

# BACHELORARBEIT

Herr
**Manuel Lucas Hesse**

## Containerisierung einer Webanwendung, automatische Integration und Bereitstellung ihrer Softwareänderungen

**2022**

Fakultät: Angewandte Computer- und Biowissenschaften

# BACHELORARBEIT

# Containerisierung einer Webanwendung, automatische Integration und Bereitstellung ihrer Softwareänderungen


Autor:
**Herr Manuel Lucas Hesse**

Studiengang:
**Angewandte Informatik**

Seminargruppe:
**IF19ws-B**

Erstprüfer:
**Prof. Dr. habil. Matthias Vodel**

Zweitprüfer:
**Dr. Rico Beier-Grunwald**

Faculty: Applied Computer Science and Bioscience

**BACHELOR THESIS**

# Containerization of a web application, continuous integration, and deployment of its software changes

author:
**Mr. Manuel Lucas Hesse**

course of studies:
**Applied Computer Science**

seminar group:
**IF19ws-B**

first examiner:
**Prof. Dr. habil. Matthias Vodel**

second examiner:
**Dr. Rico Beier-Grunwald**

**Bibliografische Angaben**

Hesse, Manuel Lucas:

Containerisierung einer Webanwendung, automatische Integration und Bereitstellung ihrer Softwareänderungen – 2022. 9,40,5S.

Hochschule Mittweida, University of Applied Sciences,
Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2022

**Referat**

Containerisierung einer Webanwendung und Umsetzung von Continuous Integration und Continuous Delivery zum Erreichen von kürzeren Entwicklungszeiten.

# Inhaltsverzeichnis

**Inhaltsverzeichnis** ..... I

**Abbildungsverzeichnis** ..... III

**Quellcodeverzeichnis** ..... IV

**Abkürzungsverzeichnis** ..... V

**1 Einleitung** ..... 1

1.1 Motivation ..... 1
1.2 Zielsetzung ..... 2
1.3 Aufbau der Arbeit ..... 2

**2 Problembeschreibung und Lösungskonzeption** ..... 3

2.1 Beschreibung des aktuellen Entwicklungsprozesses ..... 3
2.2 Beschreibung des angestrebten Entwicklungsprozesses ..... 5
2.3 Bewertung von Lösungsmöglichkeiten ..... 7

**3 Stand der Technik** ..... 10

3.1 Webanwendung ..... 10
3.2 Containerisierung ..... 11
3.2.1 Docker Compose ..... 14
3.2.2 Kubernetes ..... 15
3.3 DevOps anhand CI / CD ..... 18

**4 Praktische Umsetzung** ..... 20

4.1 Containerisierung der Django Anwendung ..... 20
4.2 Entwicklungsumgebung ..... 22
4.3 Produktiv- und Testumgebung ..... 24
4.4 Continuous Integration Continuous Delivery ..... 29

**5 Ressourcenbetrachtung** ..... 33

5.1 Projekt- und Teamdarlegung ..... 33
5.2 Grundaufwand ..... 33

5.3 Aufwandsanalyse CI/CD .......... 34

**6 Ergebnisdarlegung .......... 37**

**7 Fazit .......... 39**

**Anlagen .......... I**

Anlage I: Containerkonstellation mit Docker Compose .......... II

Anlage II: Container Builddefinition als Dockerfile .......... III

Anlage III: Azure DevOps Pipelinedefinition in yaml .......... IV

**Literaturverzeichnis .......... VII**

**Selbstständigkeitserklärung .......... IX**

# Abbildungsverzeichnis

Abbildung 1 Aktuelle architekturelle Umsetzung ... 4

Abbildung 2 Aktueller Entwicklungsprozess ... 5

Abbildung 3 Zukünftige architekturelle Umsetzung ... 6

Abbildung 4 Vereinfachter zukünftiger Entwicklungsablauf ... 7

Abbildung 5 Architekturelle Darstellung des Einsatzes von Software in verschiedenen Deployment Ansätzen ... 12

Abbildung 6 Image Layer Konzept ... 13

Abbildung 7 Netzwerkdiagramm Ingress + Services ... 17

Abbildung 8 Laufzeit CI Pipeline ... 34

Abbildung 9 Aufwandsdiagramm mit Einführungsaufwand für DevOps Prozess ... 35

Abbildung 10 Aufwandsdiagramm in Abhängigkeit zur Entwickleranzahl ... 36

Abbildung 11 Finaler Entwicklungsprozess nach Inbetriebnahme der CI/CD ... 38

# Quellcodeverzeichnis

Codebeispiel 1 Base Image ... 21

Codebeispiel 2 Dockerfile RUN Command ... 21

Codebeispiel 3 Dockerfile Arguments ... 21

Codebeispiel 4 Dockerfile Entrypoint ... 22

Codebeispiel 5 Compose Container Definition ... 22

Codebeispiel 6 Compose Definition Hauptanwendung ... 23

Codebeispiel 7 Compose Build Anweisung ... 23

Codebeispiel 8 Compose Restart Anweisung ... 24

Codebeispiel 9 Compose Laufzeitüberschreibung des Entrypoints ... 24

Codebeispiel 10 Kubernetes Manifest Deployment Webanwendung ... 25

Codebeispiel 11 Kubernetes Manifest Service ... 26

Codebeispiel 12 Kubernetes Manifest Ingress ... 27

Codebeispiel 13 Helm Abschnitt Values in Template ... 28

Codebeispiel 14 Helm Value Deklaration ... 28

Codebeispiel 15 Helm aufgelöstes Manifest ... 28

Codebeispiel 16 Helm Verzweigung am Beispiel Service ... 28

Codebeispiel 17 Helminternes taggen ... 29

Codebeispiel 18 Azure DevOps Pipeline: Value Dateien erstellen aus Template ... 31

Codebeispiel 19 Azure DevOps Pipeline: GitOps Repository Update ... 31

# Abkürzungsverzeichnis

| | |
|---|---|
| **K8s** | Kubernetes |
| **CI** | Continuous Integration |
| **CD** | Continuous Delivery/Deployment |
| **PV** | Persistent Volume |
| **PVC** | Persistent Volume Claim |
| **ACR** | Azure Container Registry |

# 1 Einleitung

„Nicht die Großen werden die Kleinen fressen, sondern die Schnellen die Langsamen.“ (Zitat Heinz-Peter Halek)

## 1.1 Motivation

Das oben genannte Zitat ist vor allem in der Entwicklung von Webanwendungen von Bedeutung. Dort geht es für ein Unternehmen darum, seine Anwendung so schnell wie möglich den Anforderungen des Kunden gerecht zu gestalten. Dabei können gerade bei Ladezeiten Sekunden darüber entscheiden, ob der Kunde sich für das Unternehmen oder einen Konkurrenten entscheidet. Dies ist vor allem für einen industriellen Großkonzern wie die Harting Technology Group wichtig. Es handelt sich hierbei um einen deutschen Weltmarktführer für die Herstellung von Steckverbindern. Das Unternehmen generierte im Geschäftsjahr 2022 über eine Milliarde Euro Umsatz. Gerade bei solch einem Unternehmen können längere Ausfälle von Unternehmenswebseiten zu Verlusten in Millionenhöhe führen. Deswegen ist es von großer Bedeutung für jedes Unternehmen mit Onlinepräsenz, Ausfälle zu vermeiden und die Qualität sowie Aktualität deren Webpräsenz zu sichern. Bedingung dafür ist, dass Änderungen an der Webanwendung ausschließlich vorgenommen werden, wenn diese auf Qualität und Funktionalität überprüft wurden. An dieser Stelle greift die Arbeit in den Entwicklungsprozess ein. Durch das Erstellen eines Prozesses, welcher Qualitätsprüfung, Sicherheitsprüfung, und Bereitstellung der Software automatisiert, soll die Zeit zwischen Entwicklung und Inbetriebnahme einer Änderung minimiert werden. Dies soll dazu führen, dass das Entwicklerteam der Webanwendung, welches zum aktuellen Zeitpunkt aus fünf Entwicklern besteht, schnell und zuverlässig Kundenwünsche und Anforderungen an die Website umsetzen kann. Im Unternehmen wird aktuell noch manuell getestet und ausgerollt. Daraus folgt, dass mehrere Mitarbeiter Arbeitszeit mit dem Testen der Änderung verbringen, welche sie sonst dafür einsetzen könnten neue Features zu entwickeln. Auch werden Änderungen periodisch ausgerollt und nicht direkt zum Zeitpunkt der Fertigstellung. Da es dabei nicht eindeutig nachzuvollziehen ist, welche Änderung welches Problem verursacht, muss mit erhöhtem Debuggingaufwand gerechnet werden. Eine weitere bedeutende Quelle für Mehraufwand sind aktuell bestehende Abweichungen zwischen Entwicklungs- und Produktivumfeld. Daraus entstehende Fehler resultieren in vermeidbaren Produktivitätsverlusten. Ein weiterer zentraler Punkt der Arbeit stellt daher die Vereinheitlichung der Entwicklungs- und

Produktivumfelder dar. Im Zuge dessen sollen die Ressourceneinsparungen gemessen und ausgewertet werden.

Im Zentrum der Arbeit steht die folgende Frage. Welche Arbeitsaufwandseinsparungen lassen sich bei der Entwicklung einer Webanwendung durch deren Containerisierung und unter Anwendung von Continuous Integration und Continuous Deployment erzielen?

## 1.2 Zielsetzung

Die Arbeit soll ergründen, wie der Entwicklungsablauf automatisiert werden kann und welche Ressourcen- und Zeiteinsparungen aus dieser Automatisierung folgen. Dabei gilt es zu evaluieren, wie ein geeigneter Entwicklungsprozess aussieht und welche Tools genutzt werden können, um diese Einsparungen zu realisieren. Im besten Fall soll die Anwendung ohne manuellen Aufwand nach jeder Änderung auf Qualität und Sicherheit geprüft und danach direkt ausgerollt werden. Dadurch soll es ermöglicht werden, einer Änderung die daraus resultierenden Fehler eindeutig zuzuordnen. Des Weiteren soll eine Lösung gefunden werden, die Fehler auf Grund von unterschiedlichen Entwicklungsumgebungen beheben kann. Somit soll ebenfalls Entwicklungszeit gewonnen werden.

## 1.3 Aufbau der Arbeit

Die Arbeit ist in drei größere Abschnitte unterteilt. Der erste Abschnitt befasst sich mit dem zu Grunde liegenden Problem. Im Zuge dessen wird auf die aktuelle sowie die angestrebte Umsetzung eingegangen. Des Weiteren werden Lösungsmöglichkeiten des Problems abgewogen. In Kapitel 3 werden die theoretischen Grundbausteine für die Umsetzung gelegt. Dabei wird auf Webanwendungen, Containerisierung, Orchestrierung von Containern und Continuous Integration und Continuous Deployment eingegangen. Den zweiten großen Abschnitt stellt die praktische Umsetzung dar. Dabei wird dargelegt, wie die Webanwendung containerisiert und wie der Einsatz in der Entwicklungsumgebung und Produktivumgebung umgesetzt wird. Außerdem wird in diesem Kapitel auf das automatische Erstellen und Ausrollen der containerisierten Webanwendung eingegangen. Der dritte Abschnitt befasst sich mit dem Auswerten der Umsetzung. Dazu wird in Kapitel 5 der Aufwand nach der Umsetzung mit dem vor der Umsetzung verglichen. In Kapitel 6 werden die Ergebnisse der Arbeit dargelegt und ausgewertet. Abschließend wird die Forschungsfrage beantwortet und es wird ein Fazit für andere Projekte gezogen.

# 2 Problembeschreibung und Lösungskonzeption

Dieses Kapitel gibt einen Einblick in die aktuelle sowie die angestrebte Umsetzung des Softwareentwicklungsprozesses. Des Weiteren wird evaluiert welche Tools und welche Prozessgestaltungen sich am ehesten eignen, um das Ziel der Ressourcen- und Zeiteinsparung zu erreichen.

## 2.1 Beschreibung des aktuellen Entwicklungsprozesses

Um nun ein besseres Verständnis dafür zu bekommen, inwiefern die Umsetzung zur Zeit von dem idealen Ablauf abweicht, lohnt es sich, die aktuellen Entwicklungsprozesse und die umgesetzte Architektur genauer zu betrachten.

Die Entwicklung findet zurzeit in drei verschiedenen Umgebungen statt. Diese sind aufgeteilt in eine Entwicklungsumgebung „*DEV*“, eine Testumgebung „*TST*“ und eine Produktivumgebung „*PRD*“. „*DEV*“ ist dabei lokal auf dem Rechner des Entwicklers angesiedelt. Als erster Schritt wird das Projekt aus dem Git Repository geklont, entwickelt und ausgeführt. Alle notwendigen Datenbankenbackends, die beschrieben werden, werden lokal als SQLite Datenbanken umgesetzt, welche nach dem Ausführen des Projektes verworfen werden. Fertige Entwicklungsstände werden im Produktivumfeld ausgerollt. Dieses besteht aus einem Windows Server, der Zugriff auf die Produktivdatenbanken hat. Bei der Testumgebung handelt es sich um eine Kopie des Produktivumfeldes mit Zugriff auf Testdatenbanken, welche wiederum eine Kopie der Produktivdatenbanken darstellen. Die Umgebungen unterscheiden sich dabei dadurch, dass die Anwendung im Entwicklungsumfeld auf der Maschine des Entwicklers läuft, während sie im Produktivumfeld auf einem dedizierten Windows Server ausgeführt wird. In Abbildung 1 ist zu erkennen, dass die Anwendung wie beschrieben direkt auf den Maschinen installiert ist. Eine Isolierung erfolgt nicht.

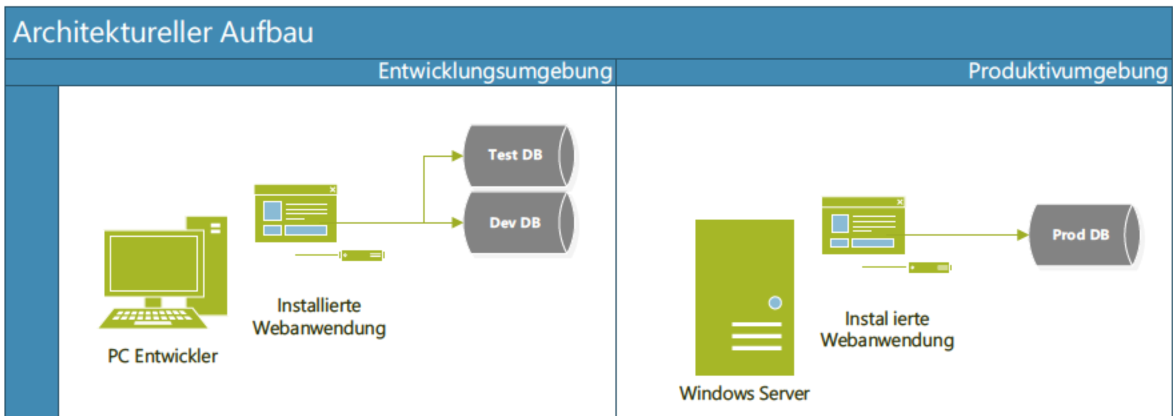



**Abbildung 1 - Aktuelle architekturelle Umsetzung**

Der aktuell umgesetzte Entwicklungsablauf der Webanwendung ist in Abbildung 4 dargestellt. Der Entwickler clont sich die aktuelle Version des Repository und erstellt einen neuen Feature Branch auf dessen Basis. In diesem neuen Branch kann der Entwickler nun seine Änderungen verfassen. Sobald er mit seinem Code zufrieden ist, führt er lokal die Datenbankenmigrationen für die Entwicklungsumgebung durch. Sind diese erfolgreich, können die automatischen Tests durchgeführt werden. Wenn es dabei zu keinen Fehlern kam, wird die Änderung noch einmal manuell auf gewünschtes Verhalten getestet. Nachdem so die Funktionalität gewährleistet ist, kann das fertige Feature in das Git Repository gepusht werden. Dort wird dann eine Jenkins Pipeline angestoßen, welche die fertige Version noch einmal per automatischen Tests im Testumfeld auf Funktionalität überprüft. Laufen auch diese Tests fehlerfrei durch, wird ein Pull Request für den Master Branch erstellt. Dieser muss dann von einem Teammitglied, welches nicht der Verfasser der Änderung ist, abgesegnet werden. Der Master Branch wird dann erneut per Jenkins Pipeline gebaut und automatisch mit den hinterlegten Produktivdatenbanken getestet. Anschließend wird die Änderung auch manuell mit den Produktivdatenbanken getestet. Besteht die Änderung auch diesen finalen Test, kann sie zu geeigneter Zeit auf dem Produktivserver installiert werden. Sollte auch nur einer dieser Schritte schief gehen, muss der Entwickler seine Version anpassen und den Vorgang erneut ab dem Testen im „*DEV*“-Umfeld durchführen.

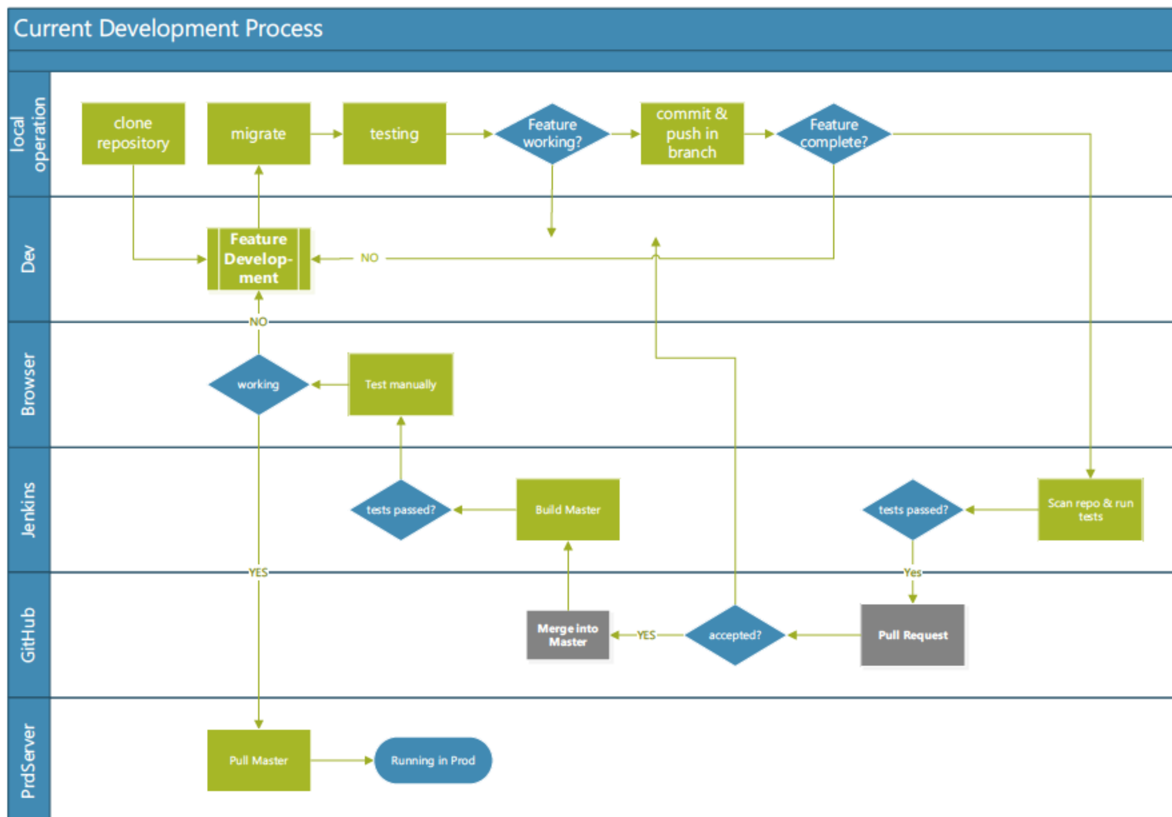



**Abbildung 2 - Aktueller Entwicklungsprozess**

## 2.2 Beschreibung des angestrebten Entwicklungsprozesses

Der gewünschte Zustand nach Umsetzung lässt sich wieder in einen architekturellen Wunschzustand und einen praktischen Prozess des Entwicklungsablaufes unterteilen. Die Änderungen am Entwicklungsprozess sind dabei auf die neue Architektur und deren Vorteile zurückzuführen.

Architekturell soll von den Entwicklungsumgebungen nicht abgewichen werden. Lediglich die Umsetzung dieser soll sich verändern. Es soll allerdings vom Einsatz mittels eines Windows Server abgesehen werden, da dieser in der Entwicklungsumgebung nicht eingesetzt wird. Das fertige Produktivumfeld soll somit eine Umgebung bieten, welche die Anwendung in derselben Form ausführt wie der Entwickler auf seiner Maschine. Die hinterlegten Datenbanken sollen in der angestrebten Architektur bestehen bleiben. In Abbildung 3 wird der hier beschriebene architekturelle Aufbau dargestellt. Dabei ist zu erkennen, dass die Installation isoliert durch Containerisierung umgesetzt werden soll.

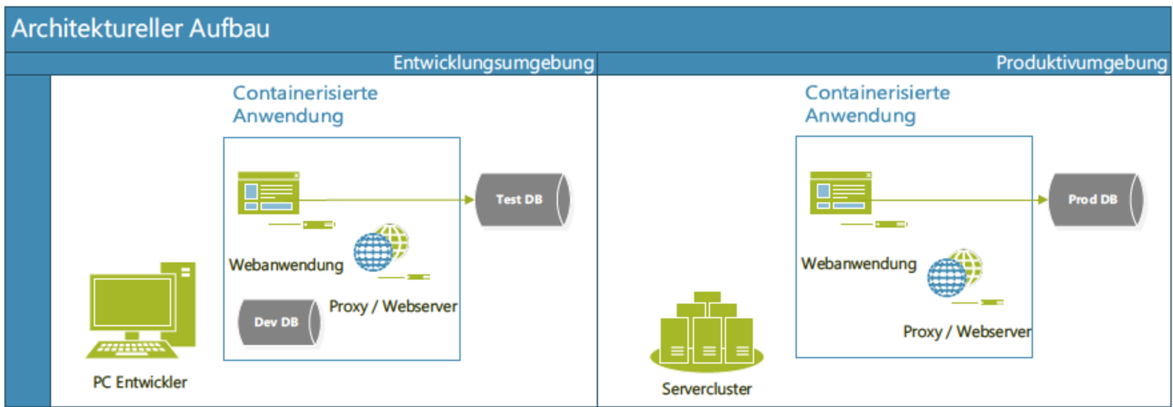



**Abbildung 3 - Zukünftige architekturelle Umsetzung**

Der Entwicklungsprozess soll sich nach der Umsetzung signifikant vom aktuellen Stand unterscheiden. In Abbildung 4 ist der Ablauf vereinfacht dargestellt. Als ersten Schritt clont der Entwickler das Git Repository mit der aktuellen Version der Anwendung und erstellt einen Feature Branch für seine Änderungen. Die Entwicklung des neuen Features soll containerisiert mit aktivem Debugging des Codes in diesem Branch möglich sein. Nachdem eine Version fertiggestellt wurde, muss diese per automatischen Tests auf Qualität geprüft werden. Sollten dabei keine Fehler aufgetreten sein, kann ein Pull Request erstellt werden, welcher die Software in den Master Branch bringen soll. Dazu ist anzumerken, dass per Continuous Integration Tool eine Build Pipeline an den Master angehängt werden muss, welche alle gestellten Pull Requests baut, auf Qualität prüft und im Testumfeld testet. Die Pipeline gilt somit als Voraussetzung für die Integration der Änderung in den Master. Ist der PR funktionsfähig, kann er von einem weiterem, dem Team angehörigen Entwickler abgesegnet und somit abgeschlossen werden.

Die Anwendung könnte, bei ausreichend automatisierten Tests, direkt per CD Tool ins Produktivumfeld ausgerollt werden. Das heißt, es müssten keine manuellen Tests mehr durchgeführt werden, da diese automatisiert und als Teil des PR abgearbeitet wurden. Sollten allerdings weiterhin manuelle Tests notwendig sein, muss die Version als Teil einer weiteren Pipeline in das Testumfeld ausgerollt werden. Dies muss vor dem Aufgeben des Pull Request durchgeführt werden. Unabhängig von der Umsetzung soll die Version nach Absegnen des PR direkt in die Produktivumgebung ausgerollt werden.

Am Ende der Umsetzung soll dann ein Ablauf stehen, welcher es dem Entwickler ermöglicht, seine Änderung schneller und garantiert auf Qualität überprüft in das Produktivumfeld einzubringen.

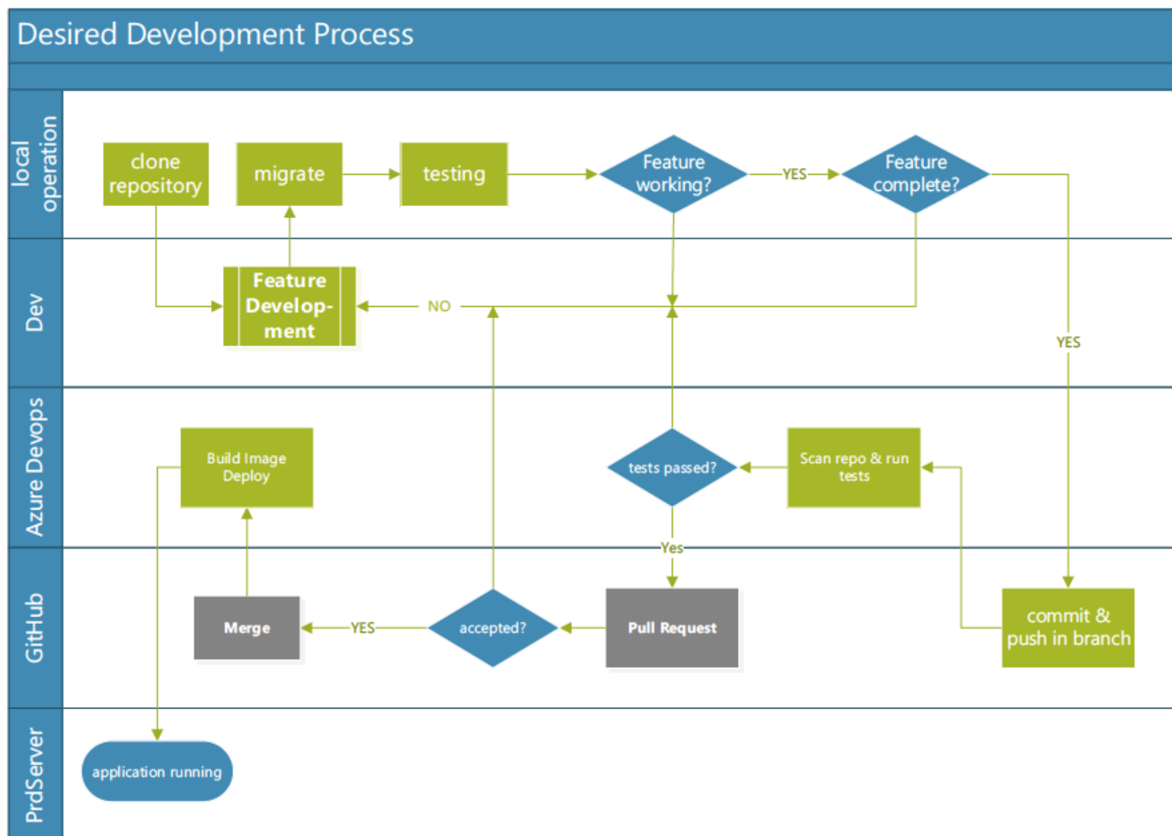



**Abbildung 4 - Vereinfachter zukünftiger Entwicklungsablauf**

## 2.3 Bewertung von Lösungsmöglichkeiten

Architekturell soll eine Vereinheitlichung der Einsatzumgebungen erzielt werden. Dies soll zu weniger Fehlern aufgrund von unterschiedlichen Abhängigkeiten auf den jeweiligen Maschinen führen. Dazu ist es notwendig, die verwendeten Technologien auf der Maschine des Entwicklers so zu gestalten, dass diese sich möglichst wenig vom Umfeld, in dem die Anwendung ausgerollt wird, unterscheidet. Dies kann auf konventionellem Wege durch das Verwenden von virtuellen Umgebungen geschehen. Solche Umgebungen lassen sich durch wenige Zeilen Code in den Entwicklungsablauf einbringen. Sie gelten damit als leicht zu implementierende Möglichkeit. Virtuelle Umgebungen können allerdings an Nutzen verlieren, wenn der Entwickler diese für mehrere Projekte verwendet. Das führt dazu, dass unbewusst unterschiedliche Abhängigkeiten in die Entwicklungsumgebung eingeschleust werden können. Somit kann diese Umsetzung Fehler nicht verlässlich verhindern. Eine weitere Möglichkeit ist das Verwenden von Virtualisierungstechniken. Diese ermöglichen es, eine Umgebung zu erstellen, welche sich im Produktivumfeld nicht vom Entwicklungsumfeld unterscheidet. Das eliminiert Fehler, welche auf Grund der Differenz der Umgebungen entstehen, komplett.

Zur Vereinheitlichung können virtuelle Maschinen mit vorinstalliertem Source Code oder Container genutzt werden. Während virtuelle Maschinen standardmäßig aufgrund ihrer Abgrenzung vom Host Betriebssystem als sicherer gelten, sind die Vorteile von Containern überwältigend. Die schnellere Startgeschwindigkeit und Cloud-Integrationsmöglichkeiten im Zusammenhang mit Orchestrationstools machen Container zur bevorzugten Virtualisierungs- und Isolationstechnologie im Entwicklungsumfeld des Unternehmens. Auf die genannten Vorteile wird in Kapitel 3.2 noch einmal näher eingegangen.

Der Einsatz von Virtualisierungstechniken ist allerdings mit einem größeren Mehraufwand verbunden, da die Anwendung dazu erst virtualisiert werden muss. Des Weiteren muss die Anwendung während des Entwicklungsprozesses in einer virtualisierten Form erstellt werden. Dies erschafft eine zusätzliche Verzögerung bei jeder Versionserstellung. Hierbei muss evaluiert werden, wie groß dieser Mehraufwand ist und in welchem Verhältnis er zum aktuell benötigten Zeitaufwand steht.

Das zweite große Ziel ist die Automatisierung des Testverfahrens zur Gewährleistung der Qualität. Dazu kann man verschiedene Frameworks oder Ansätze verwenden. Es ist beispielsweise möglich, jede Änderung manuell auf Qualität zu überprüfen. Es ist auch möglich, die Änderungen automatisch vor jedem Commit oder auf einem Testsystem, nachdem diese auf dem Sourcecode Repository landet zu testen. Generell hat das manuelle Testen immer den Nachteil, dass die Ausführung der Tests nicht garantiert ist. Menschliche Fehler können bei manuellen Tests nicht ausgeschlossen werden. Das Testen zu jedem Commit über beispielsweise Precommits hat den Vorteil, dass Änderungen in der Regel automatisch auf Qualität geprüft werden, bevor überhaupt gepusht werden kann. Dies lässt sich allerdings lokal umgehen. Das heißt, es besteht immer die Möglichkeit, diese automatischen Tests lokal zu deaktivieren und somit eine ungetestete Version hochzuladen. (Bass 2018)

Um dies zu verhindern, kann man Continuous Integration Tools verwenden. Diese Tools sind in der Lage, auf Änderungen im Git Repository zu reagieren und diese auf Qualität zu testen. Das Nutzen eines solchen Tools hat den Vorteil, dass der normale Entwickler nicht berechtigt sein sollte Änderungen am Testvorgang vorzunehmen. Somit ist garantiert, dass jede neue Version auf Qualität getestet wird. Ein Tool, welches sich gerade auf Grund der Microsoft Landschaft im Unternehmen besonders eignet, ist an dieser Stelle Azure DevOps. Alternativen dazu wären GitHub Actions oder Jenkins. Das CI Tool soll die Anwendung im besten Fall direkt in Form eines Containerimage bauen. (Makam 2020; Meyer 2015)

Das automatische Bauen hat den Vorteil, dass das Image dann im selben Zug auf Schwachstellen überprüft werden kann. Dazu muss das Image analysiert werden und ein Vulnerability Scanner sucht in diesem nach Schwachstellen. Der letzte Schritt ist dann, das Image an einen Ort zu verschieben von dem aus es zum Ausrollen aufbewahrt werden soll. Hierzu eignen sich Container Registries. Da das Unternehmen an dieser Stelle wieder von der Microsoft Integration profitiert, lohnt es sich auch hier wieder zum Microsoft eigenen Tool Azure Container Registry zu greifen. Alternativen wären an dieser Stelle Amazon ECR oder Google Registry. (Rossberg 2019)

Sobald das Container Image im Container Registry liegt, kann man es lokal ausrollen. Da es allerdings notwendig ist, neben der Anwendung weitere Komponenten einzusetzen, benötigt man ein Werkzeug, welches mehrere Docker Container ausrollen und sich um deren Verfügbarkeit kümmern kann. Dazu eignet sich Kubernetes im Produktivumfeld am besten. Auf diese Aussage und warum sich nicht für ein anderes Tool wie Docker Compose entschieden wurde, wird in Kapitel 3 näher eingegangen. Zusammen mit einem Werkzeug, welches den Container ausrollt, steht ein vollautomatisierter Ablauf, der jede Änderung auf Qualität überprüft und eine neue, isolierte Version der Software erstellt und ausrollt.

# 3 Stand der Technik

In diesem Kapitel werden die theoretischen Grundbausteine für die Umsetzung gelegt. Um der Umsetzung in Kapitel 4 folgen zu können, wird auf die Bereiche Containerisierung von Anwendungen, Orchestrierung von Containern und Gestaltung von Entwicklungsabläufen mit Hilfe von DevOps eingegangen. Außerdem werden Grundlagen über Webanwendungen vermittelt.

## 3.1 Webanwendung

Um den Nutzen der Arbeit einordnen zu können, lohnt es sich, über Kenntnisse im Bereich der Webanwendungen zu verfügen. Dies ist notwendig, um zu verstehen, warum einige Prozesse angewandt wurden und warum diese von Vorteil sind. Eine Webanwendung ist eine Software, welche auf einem Server installiert ist und Anfragen auf Basis des Hypertext Transfer Protokoll (http) beantwortet. Ziel ist es verteilten Clients eine Website zu präsentieren mit denen diese dann interagieren können. Eine Website ist ein Hypertext Dokument, welches über einen Browser angezeigt werden kann und meist eine grafische Darstellung von Informationen ist. Einer Webanwendung ist in der Regel eine Datenbank zum Speichern von persistenten Daten angebunden. Dies ermöglicht zum Beispiel das Realisieren von Benutzerkonten, da Nutzername und Passwort für einen Nutzer gespeichert werden müssen, um beim Authentifizierungsversuch überprüft werden zu können. Auch ist es so möglich, spezielle Anfragen an den Server zu stellen, welche dann mit bestimmten Daten aus der Datenbank beantwortet werden. Ein Beispiel könnte eine Webseite sein, die einen Kalender anzeigt. Der Client würde eine Anfrage an den Server stellen, die spezielle Kalenderseite anzuzeigen und der Webserver würde in der Datenbank nach anstehenden Terminen suchen und diese dem Nutzer präsentieren. HTTP bietet dem Client auch die Möglichkeit Informationen an den Server zu übermitteln. So ist es in diesem Beispiel etwa möglich, dem Server einen neuen Termin zu übermitteln. Der Server legt den Termin dann in der Datenbank an und sobald der Client den Kalender erneut anfragt, wird er den angelegten Termin präsentiert bekommen. Der Server, der die Anwendung behaust, hat dabei in der klassischen Umsetzung lediglich eine Version der Anwendung installiert. Um ein Update der Software auf den Server zu spielen, ist es demnach geläufig, dass Änderungen gebündelt in Major Versionen installiert werden müssen, da die Anwendung beim Installieren zeitweise nicht zur Verfügung steht. Grund dafür in Major Versionen zu arbeiten ist, dass das direkte Installieren jeder Änderung zu einer hohen Downtime führen würde. Wird das Patchmanagement nun allerdings in dieser Form umgesetzt, können

Fehler eventuell nicht auf konkrete Änderungen zurückgeführt werden. Dies äußert sich wiederum in erhöhtem Aufwand für Debugging, da Fehler auf mehrere Ursachen zurückzuführen sind. All dies sind Probleme, die der moderne DevOps Ansatz (beschrieben in 3.3) versucht zu umgehen. Dazu ist es allerdings notwendig, die Anwendung in ein Format zu bringen, welches sich leicht, zuverlässig, reproduzierbar und überwachbar ausrollen lässt. Dazu eignet sich Containerisierung, wie sie im nächsten Kapitel beschrieben ist. (GOURLEY 2002)

## 3.2 Containerisierung

Wie im vorherigen Kapitel beschrieben, würde das Ausrollen von Softwareupdates in der klassischen Betrachtung geschehen, indem die aktuelle Version der Anwendung direkt auf einen dedizierten Server installiert würde. Jedoch haben sich im Laufe der Zeit alternative Bereitstellungskonzepte entwickelt, welche Probleme der klassischen Direktinstallation umgehen und einige zum Teil große Vorteile bieten.

Moderne Ansätze verfolgen das Prinzip der Isolierung und Paketierung von Anwendungen. Die Software wird dazu zusammen mit all ihren erforderlichen Komponenten, Abhängigkeiten, Informationen über Laufzeitumgebung und Systemwerkzeugen als Softwarebündel verpackt. Dieses ausführbare Bündel wird Container genannt. Im Gegensatz zu virtuellen Maschinen werden diese direkt auf dem Betriebssystem des Hosts ausgeführt und es wird kein weiteres Betriebssystem zwischen Anwendung und Host eingesetzt. Auch führt die Tatsache, dass alle zur Ausführung notwendigen Bestandteile der Anwendung im Container vorhanden sind dazu, dass der Container Plattformunabhängig ausgeführt werden kann. So kann ein und derselbe Container auf einer Windows, Linux oder MacOS Maschine sowie in der Cloud oder sogar in einem weiteren Container laufen. Es ist dabei nicht nur gesichert, dass der Container ausführbar ist, sondern er wird auch auf jeder Plattform gleich ausgeführt. ( Joy 2015; Anderson 2015, 104)

Ein weiterer Vorteil liegt in der Größe von Containern. Wie bereits beschrieben, nutzen Container die Bestandteile des Hostbetriebssystem. Dadurch sinkt die Größe von mehreren Gigabyte für alle Bestandteile des OS bei einer VM auf oft nur wenige Megabyte. Auch eignen sich Container für ein schnelles Bereitstellen von Anwendungen gut, da ein Container in der Regel in wenigen Sekunden erstellt und gestartet ist. Im Vergleich dazu brauchen virtuelle Maschinen oft Minuten, um alle Komponenten des Betriebssystems zu starten und dann die Anwendung auszuführen. Mit Containern ist das Einspielen von

Updates ebenfalls wesentlich einfacher. In der klassischen Umsetzung würde die neue Version der Software oder Updates ihrer Abhängigkeiten direkt auf das Produktionssystem installiert. Dabei kann es zu Komplikationen mit alten Abhängigkeiten oder Rückständen vorheriger Installationsstände kommen. Es folgt eine architekturelle Darstellung des Einsatzes von Software in den Zeitabschnitten des klassischen Einsatzes, des virtualisierten Einsatzes und des containerisierten Einsatzes. (Joy 2015)

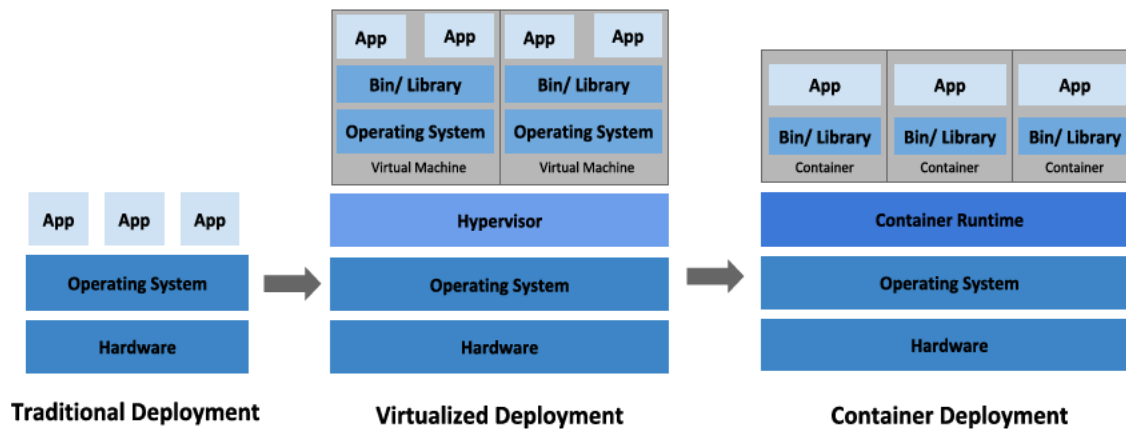



**Abbildung 5 - Architekturelle Darstellung des Einsatzes von Software in verschiedenen Deployment Ansätzen (Kubernetes Dokumentation 2022)**

Bei Umsetzung mit Hilfe von Containerisierung würde lediglich ein neues Container Image der Applikation und auf dessen Basis ein neuer Container erstellt, welcher die alte Version ersetzt. Der alte Container wird dabei gelöscht und es bleiben keine Rückstände der ersetzten Version. Daraus folgt eine geringe bis sogar keine Zeitspanne, in der die Anwendung nicht zur Verfügung steht, sowie ein rückstandsloses Update Management. (Joy 2015)

Eine der bekanntesten Technologie zur Realisierung von Containerisierung ist Docker. Dabei handelt es sich um ein Tool, welches das Erstellen, Verteilen, Einsetzen und Verwalten von containerisierten Anwendungen ermöglicht. Dazu kommt Docker auf allen gängigen Plattformen gebündelt als Docker Desktop. Dieses Bündel beinhaltet den Docker Daemon *dockerd,* welcher die erstellten Dockerobjekte wie Container, Images, Netzwerke und Volumen verwaltet, sowie das Clientprogramm *docker,* welches ein Frontend für den Docker Daemon darstellt und auszuführende Befehle an *dockerd* weiterleitet. (Kubernetes Dokumentation 2022; Docker Dokumentation 2022)

Grundlage für den Einsatz eines Containers liefern Container Images. Dabei handelt es sich um ein read-only Abbild eines Anwendungspaketes. Das Image ist für gewöhnlich in

mehrere Layer unterteilt. Ein Layer beschreibt den aktuellen Status des Abbildes und Schritte, welche ausgeführt werden müssen, um einen gewollten Zustand zu erreichen. So ist der Ausgangspunkt eines Containerimages immer ein Basisimage und Schritte, die ausgeführt werden müssen, um den gewünschten Zustand zu erreichen. Die Definition dieser Arbeitsschritte erfolgt in einem sogenannten Dockerfile. Beim Build des Images wird das Dockerfile gelesen und abgearbeitet. Es werden dann Arbeitsschritte ausgeführt und das Ergebnis jedes Schrittes als Layer gespeichert. (Bernstein 2014, S. 82–83; Bhat 2022)

Ein Beispiel für ein Imagekonzept könnte wie folgt aussehen. Die Basis des Images ist ein tar archiviertes Linux root Dateisystem einer beliebigen Distribution. Der erste Layer ist die Anweisung, ein Programm wie Git zu installieren. Git ist ein Tool zur Versionsverwaltung in der Softwareentwicklung. Danach folgt die Anweisung, ein Repository zu clonen und die Anwendung zu kompilieren. Nun würde der erste Layer aus dem Linux Dateisystem bestehen, der zweite Layer aus dem Linux Dateisystem mit installiertem Git, sowie der hinterlegten Anweisung Git zu installieren, der dritte Layer würde dann das geclonete Repository beinhalten und der letzte erstellte Layer würde ebenfalls den kompilierten Source Code beinhalten. In der folgenden Abbildung ist ein weiteres Beispiel für ein Layerkonzept dargestellt.

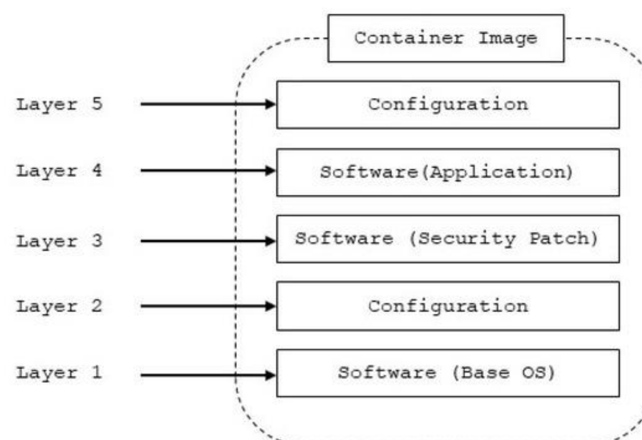



**Abbildung 6 - Image Layer Konzept (Ayres et al. 2021)**

Diese Architektur hat den Vorteil, dass beim erneuten Bauen des Images unveränderte Layer als Cache verwendet werden können. Das kann zu signifikanten Einsparungen in der Dauer des Builds führen. Der Cache kann nicht genutzt werden, wenn es bei den Instruktionen des jeweiligen Layers zu Änderungen kam oder benötigte Dateien dynamisch installiert werden müssen. Ein Container ist nun ein read-write Layer, der auf dem Image des Containers aufbaut und Konfigurationsoptionen beinhaltet, welche zum Start

übergeben wurden. Einem Container können Netzwerke und Volumes zur Speicherverwaltung angebunden werden. Volumes sind besonders dann wichtig, wenn Daten nach Beendigung des Containers gespeichert werden sollen. Dies ist wichtig, da alle Daten, welche nicht in einen angebundenen persistenten Speicher transferiert wurden, nach der Vernichtung des Containers verloren gehen. Netzwerke sind wichtig, um Container miteinander kommunizieren zu lassen, denn diese sind standardmäßig voneinander isoliert. Diese Isolation wird durch das Betriebssystem unter Verwendung von Namespaces erreicht und bietet einen soliden Schutz vor dem Übergriff eines Containers auf einen anderen. Insgesamt bietet Docker ein bequemes Umfeld für das Entwickeln und den Einsatz von Containern im Produktivumfeld. Das hat zur Folge, dass sich Docker gut für das Ausrollen von Anwendungen eignet und damit eine wichtige Rolle im modernen Software Deployment einnimmt. (Huang, Wu, Jiang, Jin 2019)

### 3.2.1 Docker Compose

Um nun einen Docker Container zu starten, würde sich mit dem Docker Daemon auf dem Host System verbunden. Dann wird aus einem Containerimage ein Container erstellt, welcher dann ausgeführt wird. Sobald die auszuführende Anwendung nun aber mehrere Container mit verschiedenen Laufzeitumgebungen benötigt, wird das manuelle Ausführen der Container unübersichtlich, fehleranfällig und aufwendig. Eine Möglichkeit dieses Problem zu lösen wäre, sich ein Skript zu schreiben, welches alle Container erstellt und ausführt. Dieser Ansatz würde das Problem zwar lösen, jedoch würde das gegen die Philosophie der Portierbarkeit von Containern gehen, da eine spezielle Scriptsprache genutzt werden muss, welche möglicherweise nicht auf allen Zielsystemen vorinstalliert ist. Ein weiterer Ansatz ist die Nutzung von Docker Compose, was standardmäßig mit der Docker Engine in Form des Befehles *„docker compose“* geliefert wird. Docker Compose nutzt eine yaml Datei in der beschrieben ist, welche Container mit welchen Einstellungen gestartet werden sollen und startet diese mit dem Befehl *„docker compose up“.* Auch bietet Docker Compose die Möglichkeit, alle in der *docker-compose.yaml* definierten Container wieder zu löschen. Dies ist recht einfach mit dem *„docker compose down“* Befehl möglich.

Docker Compose eignet sich besonders für den Einsatz auf dem lokalen Gerät des Entwicklers. Die Funktionalität, das lokale Dateisystem in den Container einzubinden, ermöglicht lokales Debugging. So ist es möglich, den Sourcecode der Anwendung im Container auszuführen und beispielsweise über eine IDE auf dem Host anzupassen und zu debuggen. Wenn die zu entwickelnde Anwendung nun in der Lage ist, Änderungen am Quellcode zu erkennen und diese anzuwenden, ist es möglich, aktiv Änderungen

einzubringen, ohne den Container neu starten oder bauen zu müssen. Aus der Tatsache, dass so auch beim Entwickeln der Anwendung Container verwendet werden, folgt eine große Ähnlichkeit der Entwicklungsumgebung zur Produktionsumgebung. Das wiederum verhindert unerwartete Fehler durch Abweichungen in der Umsetzung oder Entwicklungsumgebung. Auch handelt es sich bei Docker Compose um eine sehr Docker nahe Umsetzung. Daher ist es möglich, eine *docker-compose.yaml* mit Docker Grundkenntnissen zu schreiben. Das macht Compose wiederum zu einem einsteigerfreundlichen Tool. Entwickler können also ohne enormen Einarbeitungsaufwand vom klassischen Installieren der Anwendung auf Ausrollen in Containerform umsteigen.

Für den Einsatz im Produktivumfeld eignet sich Docker Compose hingegen weniger. Ein Grund hierfür ist das Fehlen einer Multihost Deployment Funktion, da Compose für den Einsatz auf einer Maschine konzipiert ist. Dementsprechend benötigt es ein lokales Ausführen des Docker Compose Befehls auf jedem zu bespielenden Host, um Container zu verwalten. Dies kann mit steigender Anzahl an Servern, auf denen die Anwendung bereitgestellt werden soll, sehr schnell zu einem Kraftakt werden. Des Weiteren fehlen Möglichkeiten der Hochverfügbarkeit. Compose ist nicht für das Verwalten von mehreren Kopien einer Anwendung für den Fall des Absturzes einer Instanz konzipiert. Daraus folgt, dass es beim Absturz einer Anwendung zu Downtime kommen kann, was wiederum vermieden werden muss. Insgesamt folgt, dass sich Compose gut für den Einsatz im Entwicklungsumfeld, jedoch schlecht für den Einsatz im Produktionsumfeld eignet.

### 3.2.2 Kubernetes

Im Gegensatz zu Docker Compose handelt es sich bei Kubernetes (k8s) um eine Container Orchestrationssoftware. Das heißt, k8s übernimmt nicht nur das Starten, Stoppen und Verwalten von Containern, sondern auch weitere weitreichende Funktionen für das Ausführen auf einem oder mehreren Zielsystemen. So bietet Kubernetes die Möglichkeit, eine containerisierte Anwendung auf mehrere Hosts so zu verteilen, dass Ressourcen optimal genutzt werden. Dies kann ebenfalls redundant geschehen, also in mehreren Kopien. Das hat zur Folge, dass beim unvorhergesehenen Ausfall eines Hosts die Integrität der Anwendung gewährleistet ist, indem eine Kopie auf beispielsweise mindestens einem weiteren Host läuft. Kubernetes überprüft ebenfalls konstant den Status der Applikation und startet diese neu, falls ein Fehler auftreten sollte. Auch ist es möglich, Rollouts von neuen Versionen vorzunehmen und diese bei Fehlern per Rollback rückgängig zu machen. Kubernetes würde dazu die neue Version der Applikation ausrollen und die alte Version

erst nachdem die neue vollständig funktionsfähig ist, terminieren. (Kubernetes Dokumentation 2022)

Um den Aufbau und Einsatz von Kubernetes zu verstehen, lohnt es sich, einen Blick auf die wesentlichen Komponenten und Arbeitsweisen dieser Software zu werfen. Im Zentrum stehen immer die eingesetzten Container. Kubernetes ist Containerengine unabhängig entwickelt worden und unterstützt somit fast jede Art von Container. Die Container werden auf einem Host (Node) ausgeführt, auf dem neben der Containerengine auch Kubelet und kubeproxy installiert sein müssen. Kubelet ist eine Agent Software, die sicherstellt, dass alle dem Node zugeordneten Container ordnungsgemäß laufen. Kubeproxy ist eine Netzwerkkomponente, welche erstellte Netzwerkregeln durchsetzt und die Kommunikation mit anderen Containern ermöglicht. (Kubernetes Dokumentation 2022)

Kubernetes setzt Container allerdings nicht direkt ein, sondern tut dies in Form von sogenannten Pods. Ein Pod besteht aus einem oder mehreren Containern, welche sich Speicher- und Netzwerkkomponenten teilen. Somit bieten Pods eine Abgrenzung zwischen Containern oder Containergruppen. Realisiert wird das in der Praxis über Abgrenzungsmittel des Betriebssystems wie Namespaces oder cgroups. K8s unterstützt ebenfalls die Möglichkeit, einen Pod mehrfach auszuführen, um die Anwendung zu skalieren oder Redundanz zu ermöglichen. Dazu wird eine Komponente namens Replicaset verwendet, welche sich darum kümmert, dass zu jeder Zeit eine bestimmte Anzahl an identischen Pods läuft. Es verwaltet somit Kopien von Pods. Kombiniert und um deklarative Updates erweitert, formen diese Komponenten das K8s Konzept des Deployments. In einem Deployment wird ein gewünschter Zustand für eine Anwendung angegeben. So würde im Deployment angeben, welche Container in welcher Version und Konstellation wie oft redundant eingesetzt werden sollen. Dann, wenn eine neue Version des Containers verfügbar ist, könnte das Deployment angepasst werden, indem das hinterlegte Containerimage auf die neuste Version geändert wird. K8s kümmert sich dann selbstständig darum, dass ein neues Replicaset mit der neuen Containerversion ausgerollt wird. Dabei kann bestimmt werden, wie das Ausrollen geschehen soll. Beispielsweise ist es möglich, einen Pod nach dem anderen gegen die neue Version zu tauschen, es ist aber auch möglich, sofort das ganze Replicaset auszutauschen. Über sogenannte Quotas kann dabei definiert werden, in wie vielen Kopien die Anwendung zu jeder Zeit mindestens lauffähig im Cluster erreichbar sein muss. (Kubernetes Dokumentation 2022)

Des Weiteren besitzt Kubernetes auch Komponenten zur Netzwerkverwaltung und -gestaltung. Eine dieser Komponenten sind die Services. Diese sind dazu da, Pods der Außenwelt zur Verfügung zu stellen. Services bestehen aus einem Selektor, welcher

deklariert, für welche Pods der Service gilt und einem Port, auf dem Anfragen beantwortet werden. Im Hintergrund werden Anfragen an diesen Port durch einen Kubernetes verwalteten Proxy an die im Selektor definierten Pods umgeleitet. Dabei kann definiert werden, ob der Proxy auf eine IP im Cluster, einen Domainnamen oder einen Port an jedem Node des Clusters hören soll. Somit ist es möglich, automatisch bei der Erstellung der Anwendung eine dynamische Möglichkeit zu definieren, die Anwendung zu erreichen. Alternativ ist es auch möglich, manuell eine Portweiterleitungsregel per Kubernetes einzurichten. Diese ist allerdings nur temporär und bedarf, dass der Anwender den identifizierenden Namen des Pods kennt, den er erreichen will. Mit dem fertigen Service ist es zwar möglich die Anwendung zu erreichen, jedoch bieten Services allein noch nicht die Möglichkeit, bequem Anfragen auf Basis der aufzulösenden DNS Adresse an den zugeordneten Service weiterzuleiten. Dazu kommt ein Ingress zum Einsatz. Ein Ingress ist eine Definition von Regeln, welche umgesetzt werden, um das Backend, in diesem Fall die Services der Anwendung, zu erreichen. Beispiele für solche Ingressregeln sind das Bereitstellen von Namen, unter welchen die Anwendung von außen erreichbar ist, das Auflösen von SSL Verbindungen sowie Loadbalancing von Anfragen. Dies ist hilfreich, um Anfragen an „https://appname.host" sowie „https://Host/Appname" oder „https://host/Name-ÄhnlichAppname" aufzulösen und diese an die richtigen Anwendungen weiterzuleiten. Um den Aufbau der Netzwerkkomponenten einfacher zu verstehen, folgt ein Diagramm über die Architektur einer eingesetzten Anwendung mit Erreichbarkeit über Services und Namensauflösung über einen Ingress. (Kubernetes Dokumentation 2022; Minna, Blaise, Rebecchi 2021)

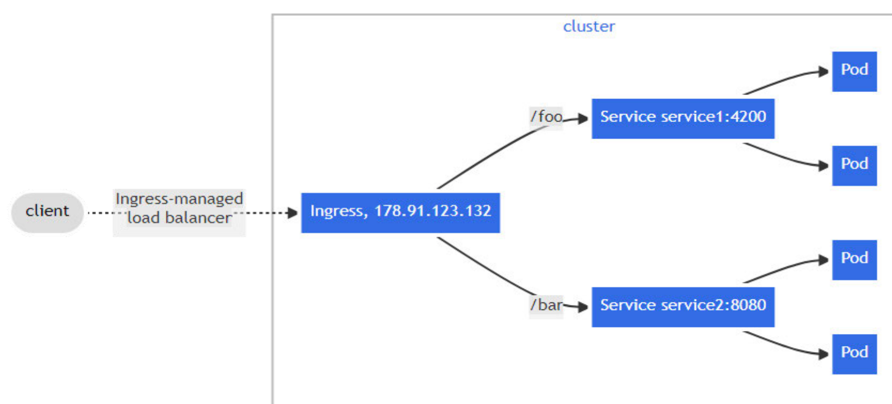



**Abbildung 7 - Netzwerkdiagramm Ingress + Services (Kubernetes Dokumentation 2022)**

Kubernetes bietet des Weiteren die Möglichkeit, Speicher zu verwalten. Im Gegensatz zu Docker werden von Kubernetes jedoch wesentlich mehr Speicherkonfigurationen unterstützt. Ein wichtiges Konzept sind hierbei Persistent Volumes (PV) und deren

Persistent Volume Claims (PVC). Persistent Volumes sind administrierte Speicherabschnitte, welche innerhalb des Clusters definiert sind und deren Größe bekannt ist. Sie sind allerdings vom Lebenszyklus von Pods unabhängig. Dies ermöglicht es Informationen zu speichern, obwohl der Container, der diese erhoben hat, bereits terminiert wurde. PVs unterstützen viele verschiedene Speicherarten wie NFS, SCSI, iSCSI oder auch das Mounten eines Verzeichnisses von einem Node. Ein Persistent Volume Claim ist nun die Anfrage eines Pods, Speicher des PV zu nutzen. Es definiert, wie viel des Speichers genutzt werden soll und wie dieser eingebunden wird. Dabei gibt es die Möglichkeit, Speicher read-only, read-write oder read-write-once zu verwenden. Weitere für diese Arbeit weniger relevante Kubernetes Objekte sind Namespaces, welche Kubernetes Objekte voneinander isolieren, Network Policies, welche den Verkehr zwischen Namespaces einschränken, Jobs, die zeitlich wiederkehrende Aufgaben abarbeiten und auch Policies, die das Beschränken von Anfragen ermöglichen oder Rechenleistung beschränken können. Dies sind jedoch bei weitem nicht alle Komponenten, sondern lediglich die am häufigsten gebrauchten. (Kubernetes Dokumentation 2022)

Kubernetes ist ein mächtiges Tool, welches sich gerade im DevOps Bereich für das Verwalten von containerisierten Anwendungen, die in kurzen Intervallen ausgerollt werden müssen, eignet. Die Möglichkeit, Anwendungen redundant in kontrollierten Rollouts einzusetzen, ist dabei wichtig, um die Prinzipien des Continuous Deployments zu realisieren.

## 3.3 DevOps anhand CI / CD

Continuous Integration und Continuous Deployment sind feste Bestandteile der modernen DevOps Kultur und tragen maßgeblich zur Gestaltung von modernen Softwareentwicklungsprozessen bei. DevOps ist ein Kunstbegriff, welcher die Wörter Development und Operations zusammenführt und genau das passiert auch in der Praxis. DevOps setzt auf das Verschmelzen von Entwicklung und operativer Seite der Softwareentwicklung und versucht damit, die klassischen Übergänge zwischen diesen Gebieten zu eliminieren. Ziel ist es, die Qualität von Softwareänderungen zu verbessern und diese schneller zu testen und auszurollen. Continuous Integration spielt dabei vor allem für die Testgeschwindigkeit und -qualität eine bedeutende Rolle. Grund dafür ist, dass so viele Schritte des Softwaretestens wie Unit- und Integrationstests automatisiert werden und die Tests als Voraussetzung für die Implementation der Änderung gelten. Dies garantiert, bei ausreichender Testabdeckung, dass zu jeder Zeit eine qualitativ hochwertige Version der Software eingesetzt ist. Des Weiteren ermöglicht es Continuous Integration,

Sicherheitstests als weiteres Kriterium für die Integration der Softwareänderung zu definieren. Dies bringt den Vorteil, dass Entwickler weniger, bis keine Schwachstellen durch alte Abhängigkeiten in die Anwendung einschleusen können. Auch bietet CI eine direkte Rückmeldung über die Qualität der Softwareänderung. Dies geschieht über die eingesetzte CI Software, welche den Entwickler direkt bei einem Fehlschlagen des Ablaufes über die Natur des Fehlschlags unterrichtet. Am Ende des Integrationsablauf steht eine einsetzbare Version der Anwendung, welche der Qualität der eingesetzten Tests entspricht und je nach eingesetzter Sicherheitsscans bestenfalls schwachstellenlos ist. Entgegen der Continuous Integration, welche im Entwicklungsprozess von der eingeschleusten Änderung bis zur fertig einsetzbaren Version einzuordnen ist, geschieht Continuous Deployment erst, nachdem die fertige Version erstellt ist. Es wird versucht, das Ausrollen der Anwendung zu automatisieren und zu garantieren, dass stets eine funktionierende Version der Software im Einsatz ist. Dies wird durch Verwendung von Hochverfügbarkeitstools in Kombination mit Containerisierung erreicht. Das hat das Potenzial, Downtime komplett zu eliminieren. Die höchste Kunst beim Gestalten eines Softwareentwicklungszyklus ist, CI und CD zu verschmelzen und einen Ablauf zu gestalten, welcher eine Änderung automatisch auf Qualität testet, deren Sicherheit garantiert und diese dann direkt im Produktionsumfeld ausrollt. (Shahin et al. 2017, S. 3910–3911)

# 4 Praktische Umsetzung

In diesem Kapitel wird die praktische Umsetzung dargelegt. Es wird darauf eingegangen, welche Schritte nötig waren, um die Anwendung zu containerisieren, wie sie im Entwicklungs- und Produktivumfeld eingesetzt wurde und wie Continuous Integration und Deployment organisiert wurden.

## 4.1 Containerisierung der Django Anwendung

Der erste praktische Schritt auf dem Weg zum automatischen Build und Deployment der Anwendung ist das Containerisieren jener Anwendung. Dies wurde mit Hilfe der Containersoftware Docker umgesetzt. Es musste also ein Container erstellt werden, welcher die Django Webanwendung und alle nötigen Libraries zur Ausführung beinhaltet. Für eine einfache Django Anwendung wäre es möglich, ein vorgefertigtes Container Image zu verwenden. Da die hier zu containerisierende Python Webanwendung jedoch auf Grund ihrer umfangreichen Funktionalitäten eine ganze Reihe an selbst entwickelten und frei verfügbaren Abhängigkeiten zur Ausführung benötigt, ist es notwendig, ein Containerimage selbst zu erstellen. Dies ist mit Hilfe eines Dockerfiles möglich. Ein Dockerfile besteht meist aus einem Base Image, welches in der Regel ein Tar Archiv einer Linux Distribution ist und weiteren Schritten, die ausgeführt werden müssen, um den gewünschten Stand des Betriebssystems zu erreichen. Für die praktische Umsetzung gilt es nun also, ein geeignetes Base Image zu finden. Da es sich um eine Django Anwendung handelt, welche wiederum ein Python Framework ist, ist es notwendig, dass Python installiert ist. Python bietet bereits ein Docker Image mit laufender Python Installation auf Basis verschiedener Linux Distributionen an, jedoch lohnt es sich auch, eventuelle Alternativen zu betrachten. Eine Alternative ist das manuelle Installieren von Python auf einer fertigen Linux Distribution. Das hat den Vorteil, dass eine aktuelle Version der Linux Distribution mit einer älteren Version von Python gepaart werden kann und somit viele Libraries des Betriebssystems auf einem neueren Stand sind. Das äußert sich wiederum in einem sichereren Base Image. Der unmittelbar daraus folgende Nachteil ist allerdings, dass die Installation mehr Schritte beinhaltet und dadurch länger und fehleranfälliger wird. In der hier durchgeführten Umsetzung wurde sich aus Performancegründen und vor dem Hintergrund, dass es sich vorerst um einen Prototyp handelt, für die Variante mit vorgefertigtem Python Base Image entschieden. Dabei wurde ein Python:3.10-bullseye Image verwendet. Dies besteht aus der zum aktuellen Zeitpunkt (12/2022) neusten stable Version der Debian Linux Distribution und einer Python Version 3.10 Installation. Für Debian wurde sich auf Grund

der Verfügbarkeit mehrere benötigter Libraries entschieden. Python 3.10 wurde gewählt, da dies die aktuelle getestete Version ist, unter der die Django Anwendung läuft. Im Dockerfile wird das Base Image wie folgt hinterlegt.

```
FROM python:3.10-bullseye
```

**Codebeispiel 1 Base Image**

Als nächsten Schritt ist es notwendig alle Systemlibraries und Utilityprogramme zu installieren, welche für den Betrieb des Webframeworks notwendig sind. Dazu zählen in diesem Fall Treiber für die Kommunikation mit einem MSSQL Backends und Libraries zur Nutzung von Verschlüsselungs- und Authentifizierungsprotokollen. Da einige dieser Programme nur in proprietären Anwendungsarchiven zu finden sind, gilt es, diese den apt-get Sources hinzuzufügen. In dieser Datei sind unter Debian Linux die Anwendungsarchive des Paketmanagers hinterlegt. Das Hashicorps vault cli Tool oder der MSSQL Treiber sind zum Beispiel Programme, die noch installiert werden müssen. Auch wird krb5-user zur Kerberos Authentifizierung in diesem Schritt aufs System gebracht. Hier ist zu erwähnen, dass alle Befehle, die auf dem Image ausgeführt werden sollen, per RUN Befehl durchgeführt werden. Der entsprechende Eintrag für die Installation der Tools sieht wie folgt aus.

```
RUN apt -y update
RUN ACCEPT_EULA=Y apt install -y lsb-release gpg vault krb5-user
```

**Codebeispiel 2 Dockerfile RUN Command**

Da ein Großteil der benötigten Python Libraries Eigenentwicklungen sind, welche sich auf einem nicht öffentlichen PyPi Server, auf der firmeneigenen Azure DevOps Instanz befinden, ist es nötig, diesen in der Python Paketmanager Konfigurationsdatei pip.conf zu hinterlegen. Dazu ist ein Authentifizierungstoken notwendig, welcher aus Sicherheitsgründen zur Laufzeit übergeben werden muss und weiter oben im Dockerfile über ein Argument definiert wird.

```
ARG token
```

**Codebeispiel 3 Dockerfile Arguments**

Als Nächstes wird der Paketmanager pip auf den aktuellen Stand gebracht. Ab dieser Stelle wird der Sourcecode der Anwendung im Container benötigt, da in diesem die Requirements der Webanwendung zu finden sind. Es werden also zuerst alle globalen Requirements in der globalen requirements.txt installiert und nachfolgend wird durch jede Unteranwendung in der Anwendung gegangen und deren spezielle Requirements installiert. Nachdem dies

geschehen ist, kann die pip.conf mit dem Accesstoken gelöscht werden und der Container ist in der Lage, das Projekt auszuführen. Um die Ausführung zu vereinfachen, wird dem Container noch ein Startskript beigefügt, welches die Anwendung migriert und in der korrekten Umgebung ausführt. Dies geschieht im Dockerfile mit dem CMD Eintrag.

```
CMD ["/bin/bash", "/Entrypoint.sh"]
```

**Codebeispiel 4 Dockerfile Entrypoint**

Nachdem all diese Schritte erfolgt sind, ist der Container komplett und sollte in der Lage sein, das Projekt in jeder Umgebung auszuführen, da er sich im Wesentlichen nicht von der produktiv eingesetzten, direkt installierten Version unterscheidet. Als Anlage I ist eine gekürzte Version des eingesetzten Dockerfiles im Anhang zu finden.

## 4.2 Entwicklungsumgebung

Der zweite große Schritt in der Umsetzung bezieht sich auf den Prozess, die Webanwendung lokal so einzusetzen, dass aktiv an ihr entwickelt werden kann. Das bedeutet, alle zu beschreibenden Datenbanken müssen lokal bereitgestellt werden und es muss möglich sein, die Anwendung aktiv zu debuggen. Es wird somit ein weiterer Container für das Datenbankenbackend benötigt und der Container der Anwendung muss so angepasst werden, dass der Sourcecode der Anwendung nicht in den Container kopiert, sondern dynamisch eingebunden bzw. gemountet wird. Des Weiteren werden zwei Kopien des Anwendungscontainers benötigt, um notwendige Aufgaben wie das Caching einer Unteranwendung und das Bereitstellen einer djangoeigenen Background Task Funktion zu übernehmen. Daraus folgt, dass es insgesamt vier Container gibt, welche gestartet werden müssen, um die Anwendung in der Entwicklungsumgebung zum Laufen zu bekommen. In Kapitel 3.2.1 wurde bereits beschrieben, warum sich Docker Compose für die Bereitstellung dieser Container eignet, deshalb wird in diesem Kapitel nur auf die Umsetzung eingegangen. Auch wird erwähnt, wie die Umsetzung unter Kubernetes aussehen würde. Als Erstes muss es für jeden zu startenden Container per Docker Compose einen Eintrag in der *docker-compose.yml* geben, der wie folgt aussieht.

```
services:
  serviceName:
    image: containerImageName
    container_name: RuntimeContainerName
```

**Codebeispiel 5 Compose Container Definition**

Darin wird beschrieben, welche Container eingesetzt werden sollen, auf welchem Image diese basieren und wie diese zur Laufzeit heißen sollen. Damit ist die Grundlage gelegt und die Container werden beim Ausführen von *„docker Compose up“* gestartet. Es ist jedoch nötig, weitere Eigenschaften zu modellieren, um die Anwendung lauffähig zu bekommen. So werden z.B. für die Hauptanwendung ein Portmapping von Hostport zu Containerport, sowie das Setzen einer bestimmten Umgebungsvariable benötigt. Diese Eigenschaften können in der gleichen *docker-compose.yml* hinterlegt werden und Compose kümmert sich zur Laufzeit selbst darum, dass Variablen gesetzt, Ports weitergeleitet und Volumen erstellt werden. Der fertige Abschnitt für die Hauptanwendung würde ähnlich dem folgenden Beispiel aussehen.

```
DjangoHauptanwendung:
    image: webanwendungsImage
    container_name: webanwendung
    environment:
      - SPEZIELLE_LAUFZEIT_ENV_VARIABLE=http://10.0.0.123:45678
    volumes:
      - ./sourcecodeverzeichnis_host:/sourcecodeverzeichnis_ct
    ports:
      - "8080:80"
```

**Codebeispiel 6 Compose Definition Hauptanwendung**

Eine Anmerkung hierzu ist, dass ein Volumen angelegt wird, welches aus einem auf dem Host liegenden Verzeichnis besteht. Dies führt dazu, dass alle in diesem Verzeichnis liegenden Daten ebenfalls im Container vorhanden sind. Das wiederum ermöglicht, dass Änderungen am Sourcecode direkt an der Anwendung geschehen. Es handelt sich dabei allerdings um ein Sicherheitsrisiko, da der Container Änderungen am Hostsystem vornehmen kann. Viele Privilege Escalation Angriffe basieren auf Schreibzugriff auf das Hostsystem, daher gilt es diesen Ansatz nur zur lokalen Entwicklung zu verfolgen und unter keinen Umständen auf einem Produktivsystem. Des Weiteren ist zu bedenken, dass das benutzte Image in diesem Fall bereits vor dem Ausführen mit Docker Compose gebaut werden muss. Um das Bauen des Containers als Schritt beim Ausführen von Docker Compose zu übernehmen, ist es nötig einen Build Abschnitt in dem jeweiligen Service anzulegen, welcher ein Dockerfile referenziert. Der entsprechende Abschnitt im Code folgt.

```
build:
    dockerfile: ./webanwendung.dockerfile
```

**Codebeispiel 7 Compose Build Anweisung**

Auch ist es notwendig, das Neustartverhalten der Container zu definieren. So ist es zum Beispiel wichtig, dass die Hauptanwendung beim Auftreten eines Fehlers neugestartet wird. Dies geschieht über das Definieren einer Restart Policy in den Eigenschaften des Service.

```
restart: always|onError
```

**Codebeispiel 8 Compose Restart Anweisung**

Für die Backgroundtasks ist es außerdem notwendig, vor der Ausführung der Python Anwendung, eine Kerberos Authentifizierung durchzuführen. Dies ist möglich, indem dem Container in der *docker-compose.yml* Datei ein spezifischer Startbefehl zugewiesen wird, welcher ausgeführt wird, wenn der Container gestartet wird. Er ersetzt das Entrypoint Skript, welches dem Container beigefügt wurde und ermöglicht so, das gleiche Base Image mit anderen auszuführenden Befehlen zu verwenden. Im folgenden Beispiel wird für das Ausführen des Python Skriptes ein spezielles Kerberos Ticket benötigt. Dies ist hier zum Beispiel notwendig, da sich das Webprojekt als Domänennutzer authentifizieren muss, um in eine Active Directory eingebundene Datenbank schreiben zu können.

```
command: /bin/bash -
c "/usr/bin/kinit databaseuser@COMPANY.DOMAIN; python /webapp/manag
e.py process_tasks"
```

**Codebeispiel 9 Compose Laufzeitüberschreibung des Entrypoints**

Eine vollständige *docker-compose.yml* wie sie im Projekt eingesetzt wurde, ist im Anhang als Anlage II zu finden.

Alternativ zur Umsetzung mit Compose ist es auch möglich, die Anwendung lokal per Kubernetes auszuführen. Im nächsten Kapitel wird eine detaillierte Beschreibung über das Deployment einer containerisierten Anwendung per Kubernetes gegeben. Die Umsetzung im Entwicklungsumfeld unterscheidet sich jedoch in geringem Maß zu der im Produktivumfeld. Daher wird an den Stellen, an denen es im nächsten Abschnitt zu strukturellen Abweichungen zwischen den Umgebungen kommt, darauf hingewiesen, welche Änderungen vorgenommen werden müssten, um den Einsatz im Entwicklungsumfeld zu ermöglichen.

Resultat ist eine funktionsfähige Version der Webanwendung, welche containerisiert auf dem lokalen Gerät des Entwicklers läuft. Diese ist von dort aus ebenfalls problemlos zu debuggen.

## 4.3 Produktiv- und Testumgebung

Für den Einsatz in der Test- und Produktivumgebung wurde sich für ein Deployment mittels helmverwalteten Kubernetes Manifesten entschieden. Ein Manifest ist eine von Kubernetes gelesene Datei, welche mindestens ein Kubernetes Objekt beinhaltet. Helm bietet die

Möglichkeit, Manifest Versionen zu verwalten. Dies ist notwendig, um beispielsweise das Modellieren eines wegfallenden Kubernetes Objektes zu ermöglichen.

Ein Beispiel dafür ist ein Manifest, welches aus drei verschiedenen Pods besteht. Wird nun ein Pod aus dem Manifest entfernt, so würde Kubernetes beim nächsten Rollout lediglich Änderungen an den beiden übrig gebliebenen Pods vornehmen. Der dritte Pod würde unberührt gelassen. Bei dem Versuch, die Pods mit dem Manifest zu löschen, würde der Pod ebenfalls nicht berücksichtigt und er würde zurückbleiben.

Helm kümmert sich darum, dass es immer einen Verweis auf alle Kubernetes Objekte gibt, die in einer Version eingesetzt wurden, und kümmert sich auch um das Aufräumen dieser.

Zur Umsetzung im Produktivumfeld lässt sich sagen, dass es grundsätzlich möglich ist, die Container über eine Pod Definition zu realisieren. Dies würde es relativ einfach gestalten, das Mounten eines lokalen Verzeichnisses umzusetzen, jedoch bieten Deployments die Funktionalität, redundante Kopien der Anwendung kontrolliert einzusetzen. Dabei wird deklarativ eine gewisse Anzahl an Replicas mit einer spezifischen Version der Anwendung ausgerollt. Dies führt zum Wegfallen der Notwendigkeit, Kopien der Anwendung manuell über mehreren Pod Definitionen zu verwalten. Auch ist es damit möglich, eine neue Version des unterliegenden Containers auszurollen. Eine Beispieldeklaration für das Deployment sieht wie folgt aus.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
  labels:
    app.kubernetes.io/name: webapp
spec:
  replicas: 1
    spec:
      containers:
      - name: webanwendung
        securityContext:
            {}
        image: "MyDockerRepository/MyWebApp:1.2.3"
        imagePullPolicy: IfNotPresent
        env:
          - name: SECRET_VAR
            value: 1234
        ports:
          - containerPort: 8080
```

**Codebeispiel 10 Kubernetes Manifest Deployment Webanwendung**

Hier würde ein Deployment angelegt, welches einen Pod ausrollt. Dieser wiederum erstellt einen Container auf Basis von Version 1.2.3 des Docker Images MyWebApp und populiert

diesen mit der Umgebungsvariable SECRET_VAR mit dem Wert 1234 und mapt Port 8080 am Pod an diesen Container. Zwar erhalten alle Pods von Kubernetes eine eindeutige IP Adresse und können sich damit untereinander erreichen, jedoch sind sie damit nicht bequem von außen erreichbar. Um dies zu erreichen ist es notwendig, einen Service anzulegen. Das Codebeispiel 11 zeigt hierbei, wie ein solcher Service deklariert wird.

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
  labels:
    app.kubernetes.io/name: webapp
spec:
  type: ClusterIP
  ports:
    - port: 8080
      targetPort: 8080
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: webapp
```

**Codebeispiel 11 Kubernetes Manifest Service**

In diesem Beispiel würde ein Service angelegt werden, welcher den Namen webapp-service trägt und dafür zuständig ist, alle Pods mit dem Namen webapp unter Port 8080 erreichbar zu machen. Dazu wird eine ClusterIP verwendet. Das bedeutet, es wird im Cluster eine IP Adresse verwendet, welche als Proxy fungiert und Anfragen an alle hinterlegten Pods umleitet.

Um die Pods auch erreichen zu können, ohne die hinterlegte ClusterIP zu kennen, wurde ein weiteres Kubernetes Objekt namens Ingress benutzt. Ingresse sind jedoch nicht nur dazu da, Services extern erreichbar zu machen, sondern sie eignen sich auch zur Umsetzung von Namensauflösung und Loadbalancing. Im nachfolgenden Beispiel wurde ein Ingress Controller auf Basis von NGINX erstellt, welcher Anfragen an den Ziel Host auflöst und an den webapp Service weiterleitet. Dieser wiederum referenziert alle Pods, welche die erstellte Anwendung behausen. Diese Funktion wird vor allem dann an Bedeutung gewinnen, wenn die Webanwendung nicht mehr monolithisch gestaltet wird, sondern nach den Grundsätzen der Microservice-Architektur in ihre Einzelteile zerlegt wird. Dann ist es nämlich notwendig oder möglich, einen Service für jede Unteranwendung zu erstellen und dann per Namensauflösung Traffic zu loadbalancen und an die jeweiligen Anwendungen weiterzuleiten. Gerade im Zusammenhang mit Kubernetes Autoscaling Funktion wird das attraktiv, denn dann ist es möglich, automatisch Teile der Anwendung, welche stärker genutzt werden, horizontal zu skalieren, während weniger genutzte

Unteranwendungen weniger Rechenleistung zugewiesen bekommen. Das Codebeispiel 12 zeigt ein Beispiel für einen Ingress Controller.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webapp-ingress
  labels:
    app.kubernetes.io/name: webapp
spec:
  ingressClassName: nginx
  rules:
    - host: localhost
      http:
        paths:
          - path: "/"
            pathType: ImplementationSpecific
            backend:
              service:
                name: webapp-service
                port:
                  number: 8000
```

**Codebeispiel 12 Kubernetes Manifest Ingress**

Die oben genannten Beispiele sind allesamt fertige Kubernetes Manifeste. Da es in der Praxisanwendung jedoch vorteilhaft ist, die Manifeste dynamisch nach Umgebung zu erstellen, ist es notwendig, jene aus einem Template mit variablen Segmenten zu erstellen. Beispielsweise muss das Template in der Testumgebung möglicherweise andere Netzwerkpolicies umsetzen als im Produktivbereich. Dafür könnten mehrere Manifeste für die jeweiligen Einsatzgebiete geschrieben werden. Dies würde jedoch dazu führen, dass Änderungen an einem Element zur Folge haben, dass in mehreren Manifesten dieselbe Änderung umgesetzt werden muss. Dies würde zu einer erhöhten Fehleranfälligkeit und Mehraufwand führen. Helm bietet mit sogenannten Helm Charts die Möglichkeit, Manifeste dynamisch zu erstellen. Ein Helm Chart ist eine Sammlung an Dateien, welche zusammen ein Projekt bilden. Grundsätzlich gibt es zwei Komponenten, welche zusammen die fertigen Kubernetes Manifeste erstellen. Das sind die Templates und Value Dateien. Templates sind um Variablen und Entscheidungsanweisungen erweiterte Kubernetes Manifest Dateien, welche zur Laufzeit interpretiert und aufgelöst werden und somit dynamisch Kubernetes Manifeste erzeugen. Es folgt ein Abschnitt aus einem Template für das Erstellen eines Service. In Codebeispiel 13 ist der spec Abschnitt des zukünftigen Manifests, welcher den eingesetzten Service beschreibt, zu sehen.

```
spec:
  type: {{ .Values.mainApp.serviceType }}
  ports:
    - port: {{ .Values.mainApp.Port }}
```

**Codebeispiel 13 Helm Abschnitt Values in Template**

Nun muss es einen mainApp Abschnitt in der Values Datei geben, welcher eine serviceType und eine port Variable beinhaltet. Die Praxisumsetzung folgt in Codebeispiel 14.

```
mainApp:
  serviceType: NodePort
  port: 5678
```

**Codebeispiel 14 Helm Value Deklaration**

Nachdem Helm das Template interpretiert und die Variablen aufgelöst hat, würde das ausgegebene Manifest wie folgt aussehen.

```
spec:
  type: NodePort
  ports:
    - port: 5678
```

**Codebeispiel 15 Helm aufgelöstes Manifest**

Das dynamische Erstellen von Abschnitten in Kubernetes Manifesten ist über If-Anweisungen möglich. Dies wurde genutzt, um Komponenten abhängig von der Umgebung zu erstellen. Beispielsweise war es notwendig, in der Entwicklungsumgebung einen Service für den Debugging Port anzulegen, welcher im Test- und Produktivumfeld nicht benötigt wird. Im Template würde dazu evaluiert, ob eine Umgebungsvariable in der Values Datei auf „*DEV*“ gesetzt ist. Ist dies der Fall, wird dem Manifest ein weiterer Serviceabschnitt hinzugefügt. Dieses dynamische Erstellen wird in Codebeispiel 16 dargestellt.

```
{{ if eq .Values.env "dev" }}
---
apiVersion: v1
kind: Service
metadata:
  …
spec:
  …
{{- end }}
```

**Codebeispiel 16 Helm Verzweigung am Beispiel Service**

Beim Ausführen würde die Umgebungsvariable dann auf den gewollten Wert gesetzt und Helm würde das Chart dynamisch für diese Umgebung erstellen. Helm kümmert sich außerdem um die Versionierung der eingesetzten Komponenten und darum, diese identifizieren zu können. Dies geschieht durch sogenannte Labels. Ein Label ist ein Key

Value Paar, welches dem Kubernetes Objekt zugewiesen wird. Über diese Labels ist es möglich zu definieren, dass ein Objekt von Helm eingesetzt wurde und in welcher Version dies geschehen ist. Daher sind beim Einsatz alle Objekte wie in Codebeispiel 17 getaggt.

```
metadata:
  name: webApp
  labels:
    helm.sh/chart: webApp-0.1.0
    app.kubernetes.io/name: webApp
    app.kubernetes.io/instance: webApp
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
```

**Codebeispiel 17 Helminternes taggen**

Helm kann somit beim Löschen der Anwendung alle Objekte mit der ausgerollten Version finden und terminieren, auch wenn eine Komponente in der nächsten Version nicht mehr vorhanden ist.

Mit diesen Mitteln ist es möglich, eine containerisierte Anwendung zuverlässig, hochskalierbar, sicher und einfach bereitzustellen. Im nächsten Kapitel wird unter anderem darauf eingegangen, wie die Kubernetes Manifeste dynamisch zu jeder Softwareänderung erstellt und zusammen mit ihren hinterlegten Containern ausgerollt werden.

## 4.4 Continuous Integration Continuous Delivery

Die Umsetzung der Continuous Integration wurde in der Praxis mit Azure DevOps realisiert. Azure DevOps ist ein mächtiges Tool von Microsoft, welches eine Vielzahl an Komponenten für das Erstellen von Continuous Integration und Continuous Deployment Abläufen bietet. Dazu zählen eine eigene Versionsverwaltung auf Basis von Git, sogenannte Pipelines, welche CI/CD Abläufe modellieren oder auch Artefakte, die wiederum dazu genutzt werden, um fertige Versionen der erstellten Software zum Download bereitzustellen. In diesem Projekt spielen dabei lediglich die Pipelines eine wichtige Rolle. Das Git Repository und die Artefakte werden zwar genutzt, sind aber nicht von großer Bedeutung bei der Umsetzung. Die Pipelines werden unterschieden in Build- und Release Pipelines. Build Pipelines realisieren Continuous Integration typische Aufgaben, während Release Pipelines für das Ausrollen der gebauten Software eingesetzt werden. Dementsprechend sind in der Umsetzung die Schritte der Qualitäts-, Formatierungs- und Sicherheitsüberprüfung, sowie der Schritt des Bauens als Build Pipeline umgesetzt. Pipelines sind unterteilt in Stages. Eine Stage realisiert eine Gruppe an zusammengehörigen Aufgaben oder Tasks.

Die erste Stage beinhaltet in dieser Umsetzung die automatischen Tests am Python Sourcecode. Der erste Task dieser Stage beinhaltet das Installieren aller notwendigen Tools, welche zur Überprüfung der Formatierung, des Lintings oder der Qualität des Codes notwendig sind. Des Weiteren gehört ein Task, welcher Formatierung mit Hilfe des Black Modules überprüft, zur Stage. Der nächste Abschnitt wird mit Hilfe des pylint Modules der Sourcecode gelintet. Als letzter Schritt dieser Stage wird der Python Code mit Hilfe des Xenon Modules auf Code Qualität überprüft. Der fehlerfreie Durchlauf dieser Tests ist Bedingung für das Ausführen der zweiten Stage, welche das Erstellen des Docker Images beinhaltet. Dabei wird ein Docker Task erstellt, der das in Kapitel 4.1 angelegte Dockerfile auf dem Pipeline ausführenden Agent baut. Das daraus entstehende Docker Image wird im nächsten Task mittels der Software Trivy auf Schwachstellen überprüft. Sobald die Sicherheit des Images gewährleistet ist, wird das tar archivierte Container Image in das sogenannte Artifact Staging Directory exportiert. Das ist ein Verzeichnis, welches genutzt wird, um Artefakte zu behausen, welche später einer Release Pipeline oder anderen Diensten zur Verfügung gestellt werden soll. Realisiert wird der Export über einen Docker Task, welcher das tar Archiv erstellt und in das Zielverzeichnis kopiert. Als letzter Schritt der eigentlichen Build Stage wird das Container Image in Form eines Artefakts an den Build angehängt. Dies ermöglicht es, das erstellte Container Image zu jedem Buildvorgang manuell über Azure DevOps herunterzuladen, anschließend lokal zu testen sowie die Möglichkeit, das Containerimage als Teil einer dem Build folgenden Release Pipeline zu nutzen. In der dritten Stage wird nun das erstellte Container Image in das Firmeneigene Azure Container Registry geladen, um dies zu archivieren. Dazu muss sich in einem Schritt beim ACR authentifiziert werden. Bevor dies allerdings geschieht, wird dem Image noch der latest Tag verpasst, um die neuste Version des Containers bequem identifizieren zu können. Auch dafür wird der existierende Docker Task genutzt. Dieser Task wird dann im Zusammenhang mit einer Service Connection verwendet, um das Image im ACR zu publishen. Service Connections sind ein Feature von Azure DevOps, welches die Authentifizierung von der Pipeline auf den Server verlagert, um Servicezugriffe einfach administrieren zu können. Dies ermöglicht, dass sich der Server um die Verwaltung der Zugangsdaten für den Servicezugriff kümmert und die Pipeline diese nicht hinterlegt haben muss. Über Service Connection ist es ebenfalls möglich zuzuweisen, welche Pipelines einen Service nutzen dürfen und welche nicht. Hat sich die Pipeline nun beim ACR authentifiziert, kann es das tar archivierte Containerimage publishen. Die letzte Stage widmet sich nun dem Ausrollen des Images. Dazu ist zu wissen, dass entsprechend der Implementation aus Kapitel 4.3 ein Git Repository angelegt ist, welches das Helm Chart beinhaltet, was das Deployment übernimmt. In diesem Chart muss die Version des

ausgerollten Images hinterlegt werden, um ein Deployment nachvollziehbar zu gestalten. Das Konzept, das auszurollende Helm Chart in Git zu verwalten, folgt dem „single source of truth“ Ansatz und wird in der Praxis GitOps genannt. Dementsprechend muss im GitOps Repository das hinterlegte Image angepasst werden. Um Änderungen an dem Git Repository vorzunehmen ist es dabei notwendig, dass der Build Account, welcher die Pipeline ausführt, das Recht besitzt, in das GitOps Repository zu commiten. Der erste Schritt dieser Stage ist nun also das Authentifizieren mit Git via einer weiteren Service Connection. Dann folgt das Clonen des GitOps Repository. Nun kann das vorgefertigte Template für die *values.yaml,* um die einzusetzende Image Version erweitert, als eine neue Datei gespeichert und dem Commit hinzugefügt werden. Dieser Prozess wird über ein Skript realisiert. Dieses Skript ersetzt den Image Namen, das Image Tag und das hinterlegte Container Registry in der Datei. Diese Datei wird später direkt per Helm ausgerollt. Diese heißt dann je nach einzusetzender Umgebung *values_tst.yaml* oder *values_prd.yaml*. Die Definition dieses Tasks ist in Codebeispiel 18 dargelegt. (Beetz und Harrer 2022, S. 73; Satapathi, Mishra 2021)

```
- task: Bash@3
          condition: succeeded()
          displayName: Update helm tst
          inputs:
            targetType: 'inline'
            script: |
              sed   's/CONTAINER_REGISTRY/$(containerRegistry)/g'   \
                  values.yaml > values_tst.yaml
              sed -i 's/IMAGE_NAME/$(imageName)/g' values_tst.yaml
              sed -i 's/IMAGE_TAG/$(tag)/g' values_tst.yaml
```

**Codebeispiel 18 Azure DevOps Pipeline: Value Dateien erstellen aus Template**

Im Letzten Schritt wird die Values Datei dann dem Commit hinzugefügt und unter dem Namen CI-Pipeline ins GitOps Repository gepusht. Der Task dazu ist in Codebeispiel 19 dargestellt.

```
        - task: Bash@3
          displayName: Git commit and push
          inputs:
            targetType: 'inline'
            script: |
              git     config     --global     user.email     "ci-
pipeline@Unternehmen.com"
              git config --global user.name "CI-Pipeline"
              git checkout -b main
              git commit -a -m "$(tag)"
              git status
              git push --force origin main
```

**Codebeispiel 19 Azure DevOps Pipeline: GitOps Repository Update**

Als Anlage III ist die komplette Pipelinedefinition, wie sie im Projekt verwendet wurde, mit allen Stages und Tasks zu finden.

Damit ist der Continuous Integration Ablauf abgeschlossen und es folgt der Ablauf des Continuous Deployment. Im Deployment Schritt wird lediglich das fertige Helm Chart im Produktiv- oder Testumfeld ausgerollt. Dies geschieht aktuell manuell über das Ausführen des „helm install“ Befehls auf dem Zielsystem. Idealerweise würde dies jedoch über ein CD Tool wie ArgoCD oder FluxCD geschehen. Diese Tools fragen das GitOps Repository ab, um den gewünschten Zustand des Kubernetes Clusters zu ermitteln. Sobald eine Änderung an diesem Repository vorgenommen wird, beispielsweise in Form einer neuen Image Version, kümmert sich das CD Tool darum, die Änderungen automatisch im Cluster des jeweiligen Umfelds auszurollen. Dabei bietet ArgoCD ebenfalls eine GUI, welche den gewünschten Zustand des Kubernetes Clusters und eventuelle Abweichungen dazu grafisch darstellt. Dieser Ansatz ermöglicht es, eine neue Version des Container Images innerhalb von Sekunden in das Produktions- oder Testumfeld auszurollen. Sollte es dabei zu Fehlern kommen, können diese Tools den aktuellen Stand per Rollback zu einer älteren, funktionierenden Version migrieren. Auch ist es möglich, neue Versionen nur zu einem bestimmten Prozentsatz auszurollen. So kann ein Rollout erstellt werden, welches 60% der eintreffenden Anfragen auf die alte und 40% auf die neue Version umleitet. Somit könnte die neue Version im Produktivumfeld getestet und bei Fehlern auf die alte Version umgeleitet werden. Im Laufe der Zeit könnte dann mehr Traffic auf die neue Version verwiesen werden. (Ramadoni, Utami, Fatta 2021)

# 5 Ressourcenbetrachtung

In diesem Kapitel wird darauf eingegangen, inwiefern die Arbeit einen Einfluss auf den zu erwartenden Arbeitsaufwand hat. Hierzu wird evaluiert, wie viel Arbeitsstunden pro Woche für das Debugging und Zuordnen von Fehlern genutzt wird. Danach wird analysiert, welchen Einfluss das Verwenden von Continuous Integration und Continuous Deployment auf diesen Aufwand hat. Abschließend wird darauf eingegangen, ob und inwiefern eine steigende Entwicklerzahl Einfluss auf diesen Aufwand hat.

## 5.1 Projekt- und Teamdarlegung

Für das Verständnis des Entwicklungsaufwandes ist ein Überblick über das Entwicklerteam und die konkreten Arbeitsschritte notwendig. Es handelt sich bei der entwickelten Anwendung um eine Python Webanwendung, welche ohne Zuarbeit von externen Entwicklern erstellt wird. Die Entwicklung der Webanwendung wurde 2020 ins Leben gerufen. Anfangs waren an der Entwicklung drei Softwareentwickler beschäftigt. Im Kalenderjahr 2021 hatte das Team seine größte Mitarbeiterzahl mit sieben Entwicklern in Vollzeitanstellung. Zum Zeitpunkt der Erstellung der Arbeit hat das Team fünf Entwickler in Vollzeitanstellung sowie einen Projektleiter, welcher allerdings Aufgaben im Projektmanagement übernimmt und somit bei Berechnungen des Entwicklungsaufwandes nicht berücksichtigt wird. Pro Woche bringt jeder Entwickler durchschnittlich zwei Änderungen in die Codebasis ein. Somit kommt es wöchentlich zu rund zehn Änderungen an der Anwendung. Änderungen werden periodisch ausgerollt. Das heißt es gibt einen Stichtag, an dem alle fertiggestellten Änderungen ausgerollt werden. Danach gibt es eine Periode, in der alle entstandenen Fehler behoben werden. Dabei kommt es zu einem Mehraufwand, da die Fehler meist schwer einer konkreten Änderung zuzuordnen sind. Dieser Mehraufwand stellt den Grundaufwand vor Umsetzung des CI/CD Ablaufes dar und wird im folgenden Kapitel als Grundaufwand betitelt.

## 5.2 Grundaufwand

Um den Grundaufwand ohne die Verwendung von CI/CD zu ermitteln, wurde eine Beobachtung des aktuellen Entwicklungsprozesses vorgenommen. Dies geschah in Zusammenarbeit mit dem projektleitenden Entwickler. Es wurden Gruppengespräche mit den projektbeteiligten Entwicklern durchgeführt, welche ergaben, dass im Schnitt zwei

Arbeitsstunden pro Woche für das Debugging von Fehlern, auf Grund unterschiedlicher Entwicklungsumgebungen und das Zuordnen von Fehlern zu den jeweiligen Änderungen einzuplanen sind. Diese Erkenntnis hat sich über die Lebensdauer des Projektes für eine Anzahl von drei bis sieben Teammitgliedern bestätigt.

## 5.3 Aufwandsanalyse CI/CD

Continuous Integration in Zusammenarbeit mit Containerisierung hat zur Folge, dass Fehler auf Grund von unterschiedlichen Entwicklungsumgebungen fast vollständig ausgeschlossen werden können. Dies wird in Kapitel 3.2 belegt. Des Weiteren hat das Arbeiten nach CI/CD den Vorteil, dass Änderungen direkt ins Produktionsumfeld ausgerollt werden. Das hat zur Folge, dass Fehler einer Änderung in der Regel eindeutig zugeordnet werden können. Ergründet wird dies in Kapitel 3.4. Es ist allerdings notwendig, den Container zu bauen, zu testen und auszurollen. Hierzu wird ein Mehraufwand in Form der Laufzeit der CI/CD Pipeline in den Entwicklungsprozess eingebracht. Dieser Mehraufwand bemisst sich allerdings auf lediglich 15 Minuten pro Änderung. Um dies zu belegen, wurde der Continuous Integration Ablauf entsprechend Kapitel 4 modelliert und ausgeführt. Die Laufzeit der jeweiligen Abschnitte dieses Ablaufs ist in Abbildung 8 dargestellt.

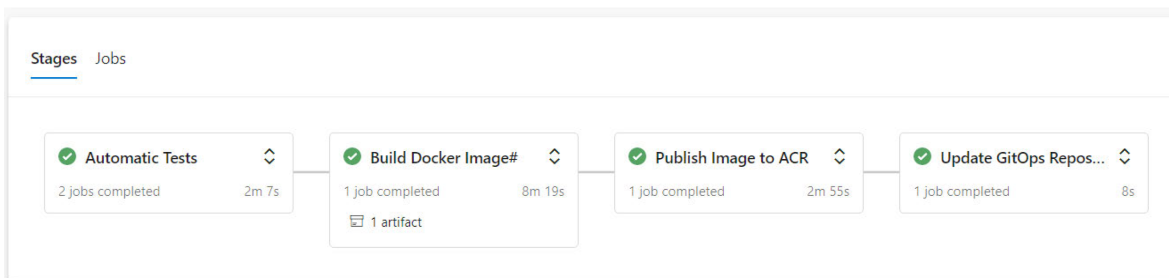



**Abbildung 8 - Laufzeit CI Pipeline**

Kumuliert erhält man einen zeitlichen Mehraufwand von 13,29 Minuten. Dies beinhaltet jedoch allein das Erstellen des Containerimages. Während des „Proof of Concept“ zum automatischen Ausrollen der Anwendung wurde festgestellt, dass es einer weiteren Minute Wartezeit bedarf, bis der Container dann auch im Produktivumfeld ausgerollt ist. Somit kann man mit ungefähr 15 Minuten rechnen, welche zwischen Änderungserstellung und Einsatz der Änderung vergehen. Es lässt sich an dieser Stelle diskutieren, ob diese Zeit tatsächlich als Mehraufwand zu interpretieren ist, da der Entwickler sich in dieser Zeit mit anderen Problemen auseinandersetzen kann. Wenn ein Entwickler an mehreren Problemen arbeitet, kann die Laufzeit der Pipeline vernachlässigt werden. In den folgenden Berechnungen wird immer davon ausgegangen, dass der Entwicklungsablauf nicht

parallelisiert werden kann und es durch das Ausführen der Pipeline zu einem Mehraufwand kommt. Es wird jedoch bei den Berechnungen erwähnt, welche Einsparungen möglich sind, falls dies nicht der Fall sein sollte.

Zwar bietet CI einen entscheidenden Geschwindigkeitsvorteil, jedoch ist die Erstellung eines solchen Ablaufes auch mit großem Aufwand verbunden. In diesem Projekt wurde über neun Wochen jeweils 40 Wochenarbeitsstunden nur in das Ausarbeiten und Erstellen dieses Ablaufes investiert. Am Ende der Arbeit steht lediglich ein erster funktionierender Prototyp. Erfahrungsgemäß ist weitere Verbesserungsarbeit und Fehlerbehebung notwendig, bevor der Einsatz des Continuous Integration Ablaufs in einer signifikanten Aufwandseinsparung resultiert. In absehbarer Zukunft sollte allerdings ein Großteil der Fehler gefunden sein. Wenn man also davon ausgeht, dass nach finaler Einführung lediglich eine Wochenstunde damit verbracht wird, die Pipeline zu warten, kann man, bei fünf aktiven Entwicklern, mit jeweils zwei größeren Änderungen pro Woche, mit aufgerundet drei Stunden Mehraufwand rechnen. In Kombination mit einer Stunde Wartungsaufwand für die Pipeline kommt man auf 4 Wochenstunden Zusatzaufwand. Das folgende Diagramm zeigt den Arbeitsaufwand ohne den Einsatz von CI/CD sowie den nötigen Einführungsaufwand, um nach DevOps Prinzipien arbeiten zu können.

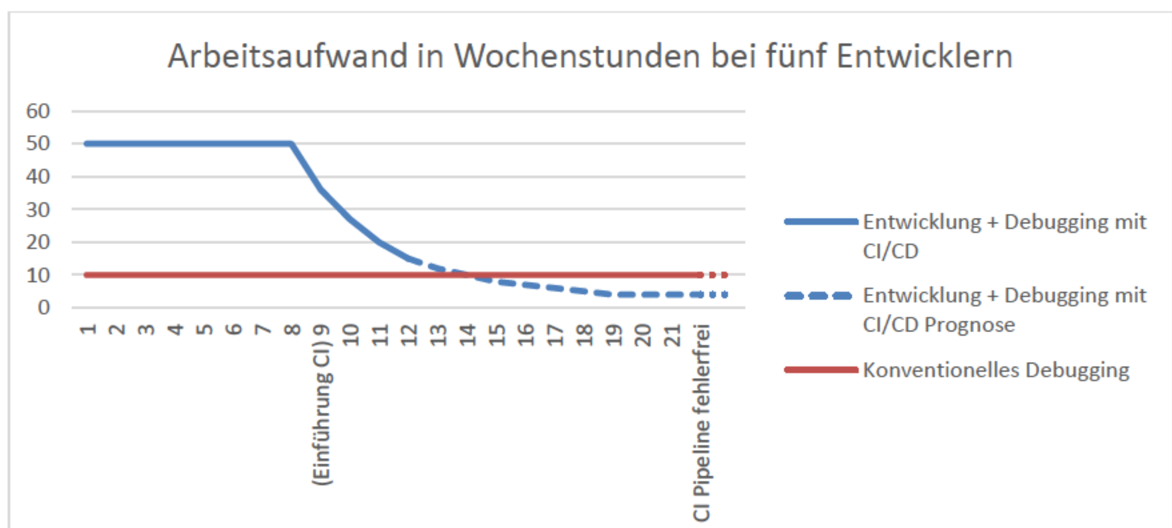



**Abbildung 9 – Aufwandsdiagramm mit Einführungsaufwand für DevOps Prozess**

Bis Woche zwölf spiegelt es den tatsächlichen Arbeitsaufwand wider. Ab Woche 13 wird auf Grund des abnehmenden Trends die Annahme getroffen, dass innerhalb von zehn Wochen alle groben Fehler beseitigt worden und ab diesem Zeitpunkt bei voller Effizienz nach CI/CD gearbeitet werden kann. An dieser Stelle lässt sich anmerken, dass es bei fünf Entwicklern nach ungefähr einem Jahr zu einer quantitativen Einsparung an

Arbeitsaufwand kommt. Rechnet man damit, dass die Laufzeit der Pipeline für andere produktive Tätigkeiten genutzt werden kann, wird eine quantitative Aufwandseinsparung bereits nach neun Monaten erreicht.

Unter der Prämisse, dass jedes Teammitglied zwei Änderungen pro Woche einbringt, die Aufwände entsprechend der bestehenden Beobachtung linear steigen und pauschal eine Stunde für die Wartung der Pipeline eingeplant wird, kann man folgendes Diagramm über den Aufwand bei steigender Entwicklerzahl, erstellen.

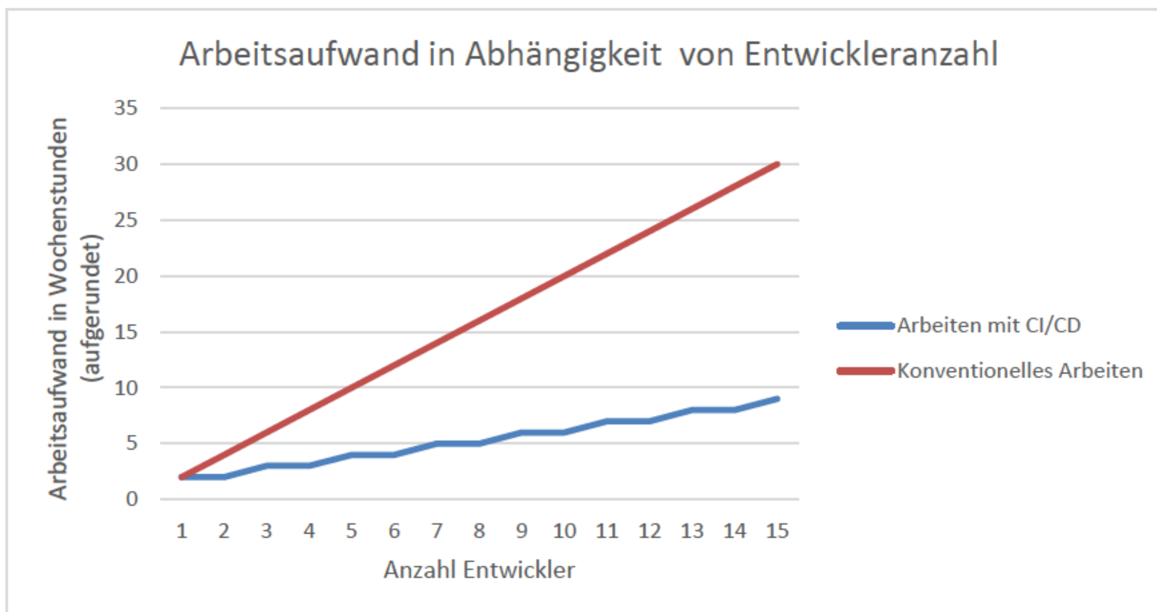



**Abbildung 10 - Aufwandsdiagramm in Abhängigkeit zur Entwickleranzahl**

Hierbei ist zu erkennen, dass der Aufwand der alten Umsetzung schneller ansteigt als die Umsetzung mittels CI/CD. Daraus lässt sich schließen, dass mit steigender Entwicklerzahl steigende Effizienzgewinne zu verzeichnen sind. Schon sobald zwei Mitarbeiter an der Webanwendung arbeiten, kann der DevOps Ansatz somit eine fünfzig prozentige Verringerung an Aufwand bieten. Bei der aktuellen Teamkonstellation im Unternehmen mit fünf Entwicklern sind bei der beschriebenen Umsetzung Aufwandseinsparungen von 60 % möglich. Rechnet man damit, dass die Laufzeit der Pipeline für andere wertschöpfende Tätigkeiten eingesetzt wird, kann man bei fünf Entwicklern sogar mit einer Einsparung von 90 % rechnen. Bezogen auf Diagramm 7 lässt sich sagen, dass der Anfangsaufwand mit steigender Entwicklerzahl relativiert wird. Dies führt dazu, dass sich die Umsetzung nach CI/CD gerade für größere Teams lohnt. Bei kleineren Entwicklerteams bis drei Entwickler ist allein der quantitative Nutzen nicht mit dem Einführungsaufwand gleichzusetzen. Kleinere Entwicklerteams sollten CI/CD daher nur in Erwägung ziehen, wenn das schnelle Ausrollen der Anwendung eine Notwendigkeit darstellt.

# 6 Ergebnisdarlegung

Es wurde ergründet, dass sich eine Umsetzung nach Grundsätzen der Continuous Integration und des Continuous Delivery als wertschöpfend für den Entwicklungsprozess ausspricht. Daraufhin wurde erarbeitet, dass die Containerisierung der Anwendung als notwendig anzusehen ist. Es wurden Containerisierungstools abgewogen, aus denen Docker als am praktikabelsten hervortrat. Mit Hilfe eines Dockerfiles wurde ein Container auf Basis der Webanwendung erstellt, welcher ausgerollt werden kann. Nachdem die Anwendung containerisiert wurde, konnte sich darauf vertieft werden, wie der Ablauf der Continuous Integration für die Anwendung aussehen soll. Dabei wurde eine Azure DevOps Pipeline aufgesetzt, welche die Anwendung auf Qualität überprüft. Dabei wird mit den Python Modulen *„black“, „pylint“* und *„xenon“* jeweils auf Formatierung, statische Fehler und Komplexität untersucht. Des Weiteren übernimmt ADO das automatische Erstellen eines Docker Containers der jeweiligen Version. Nach dem Erstellen des Containerimages wird dieses auf Sicherheit überprüft. Dazu wurde das Tool *„Trivy“* eingesetzt. Es wurde ebenfalls ein automatischer Upload des fertigen Containerimage in ein Containerregistry eingerichtet. Auch ein GitOps Repository, welches den aktuellen Status der Anwendung beinhaltet, wurde erstellt. Dieses ist notwendig für das automatische Deployment der Anwendung.

Der Einsatz der Anwendung wurde per Kubernetes in Kombination mit Helm Charts realisiert. Dabei wurde der Anwendungskontext mit allen benötigten Zusatzcontainern definiert. Im Zuge dessen wurden geteilter Speicher zwischen Containern per Kubernetes Volumes realisiert. Die Anwendung wurde mit Hilfe von Kubernetes Services per Netzwerkverbindung erreichbar modelliert. Per Kubernetes Loadbalancer wurde die Möglichkeit geschaffen, die Anwendung per Hostname aufzurufen. Mit Hilfe von Helm Chart wurde das Anwendungsdeployment dann für die jeweiligen Entwicklungsumgebungen variabel gestaltet.

Lediglich das automatische Ausrollen der Anwendung konnte nur in einer Testumgebung realisiert werden. Grund dafür war, dass zum Ausrollen ein produktives Kubernetes Cluster notwendig ist. Da dieses Cluster zur Zeit der Abgabe nicht produktiv war, konnte das Ausrollen per CD Tool nur als „Proof of Concept“ in einem lokalen Kubernetes Cluster durchgeführt werden. Die finale Umsetzung steht somit noch aus. Der finale Ablauf erfolgt vollkommen automatisiert und wird im nachfolgenden Prozessdiagramm dargestellt.

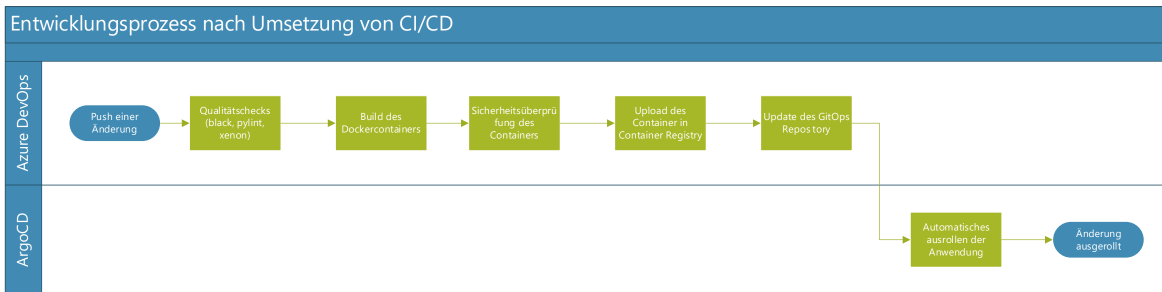



**Abbildung 11 - Finaler Entwicklungsprozess nach Inbetriebnahme der CI/CD**

Des Weiteren wurde der aktuelle Entwicklungsprozess evaluiert und im Zuge dessen festgestellt, dass zwei Stunden Arbeitszeit wöchentlich pro Entwickler anfallen, um Fehler auf Grund von verschiedenen Umgebungen der Entwicklungsumfelder zu debuggen. Dazu wurde gezeigt, dass das Containerisieren der Anwendung solche Fehler eliminiert. In den zwei Wochenstunden ist außerdem das Zuordnen von Fehlern im Produktivumfeld zu der jeweiligen Änderung, die ihn verursacht hat, enthalten. Dies ist eine unmittelbare Folge des gesammelten Ausrollens von Änderungen. Es wurde ergründet, dass CI/CD Änderungen einzeln ausrollt, was zur Folge hat, dass ein Fehler im Normalfall direkt der Änderung zugeordnet werden kann, die ihn verursacht hat. Es wurde gezeigt, dass dieser Ansatz zwar einen Mehraufwand von 15 Minuten pro Änderung in den Entwicklungsprozess einbringt, dafür allerdings zwei Wochenstunden pro Entwickler frei gibt.

# 7 Fazit

Zusammenfassend lässt sich sagen, dass die Arbeit dem Unternehmen einen Prozess erstellt hat, welcher die eingesetzte Webanwendung containerisiert. Die Containerisierung wurde so umgesetzt, dass diese bei jeder Versionsänderung an der Software bestehen bleibt und nicht nur einen Proof of Concept darstellt. Der dabei entstehende Container wird auf Qualität und Sicherheit überprüft und kann manuell hochverfügbar in jeder Entwicklungsumgebung ausgerollt werden. Es wurden außerdem die daraus entstehenden Aufwandseinsparungen evaluiert.

Es wurde gezeigt, dass mit CI/CD Debuggingaufwände auf Grund unterschiedlicher Entwicklungsumgebungen, sowie Fehlerzuordnungsaufwände durch Ausrollen von Änderungssammlungen entfallen. Lediglich das Ausführen dieser automatisierten Abläufe bringt einen Mehraufwand von 15 Minuten pro Änderung mit sich. Geht man davon aus, dass während des automatischen Ablaufes andere wertschöpfende Tätigkeiten durchgeführt werden können, so kann man diesen Mehraufwand vernachlässigen. Daraus folgt, dass bei durchschnittlichen Fertigstellungsfrequenzen von zwei Änderungen pro Woche bei fünf Entwicklern rund 75 % der oben genannten Aufwände eingespart werden können. Nutzen die Entwickler die Laufzeit der Pipeline für andere Änderungen, kann bei einer Wochenstunde Wartungsaufwand für die Pipeline, mit 90 % Einsparung gerechnet werden. Dies führt dazu, dass der Einführungsaufwand der Pipeline in unter einem Jahr ausgeglichen ist.

Rückblickend gilt es, die Berechnung des Produktivitätsmehrwertes mit Vorsicht zu genießen. Diese baut auf einem Wert auf, welcher nicht wissenschaftlich überprüft wurde, sondern lediglich ein Erfahrungswert darstellt. Es könnte als zukünftiger Schritt evaluiert werden, inwiefern der Aufwand bei steigender Entwicklerzahl tatsächlich von diesem Wert abweicht. Auch der Wert von zehn Wochen zum Beseitigen der groben Fehler, bei der Einführung der CI/CD Pipeline, ist ein spekulativer Wert. Dieser hat sich zwar bei anderen Projekten bewährt, jedoch könnte der tatsächliche Aufwand stark von der Prognose abweichen.

Weitere Verbesserungen sind in der Umsetzung des Continuous Integration Ablauf möglich. Das automatische Erstellen des Docker Containers zum Beispiel findet in der hiesigen Umsetzung lediglich als ein Single-Stage Build statt. An dieser Stelle gilt es, die Vorteile von Multi-Stage Builds zu ergründen, um den Mehraufwand beim Erstellen des Containers, durch Anwendung von effizientem Caching zu verringern. Auch der Einsatz von Bash

Skripten als Teil der Azure Pipeline sollte neu evaluiert werden und möglicherweise durch einen dafür geeigneteren Job ersetzt werden.

Für zukünftige Arbeiten eignet sich unter anderem eine Untersuchung der architekturellen Umsetzung. Es könnte evaluiert werden, inwiefern die Anwendung von einer Microservice-Architektur profitieren könnte.

Durch diese Arbeit wurde gezeigt, dass die Containerisierung im Zusammenspiel mit Continuous Integration und Continuous Deployment bei der Harting Technology Group zu einer Aufwandseinsparung von neun Wochenstunden, in einem Entwicklerteam mit fünf Entwicklern, geführt hat. Bei größeren Entwicklerteams mit frequenten Änderungen kann es zu größeren Einsparungen kommen. Bei Projekten mit drei oder weniger Entwicklern oder geringen Änderungszahlen pro Woche, können Continuous Integration und Continuous Deployment zu Einführungsaufwänden führen, welche um ein Vielfaches größer sind als der erzielte Nutzen.

# Anlagen

Anlagenverzeichnis …………………………………………………………………... A-I

Anlage I: Container Konstellation mit Docker Compose …………………………. A-II

Anlage II: Container Builddefinition als Dockerfile ………………………………… A-III

Anlage III: Azure DevOps Pipelinedefinition in Yaml ……………………………… A-IV

# Anlage I: Containerkonstellation mit Docker Compose

```
version: "3.9"
services:
  ssp:
    image: webapp:latest
    container_name: app
    env_file:
      - containerisation/.env
    environment:
      - VAULT_ADDR=http://x.x.x.x:portx
    build:
      context: .
      dockerfile: ./containerisation/webapp_container/dockerfile
    volumes:
      - ./webapp:/webapp
    ports:
      - "8000:8000"
  background_tasks_python:
    container_name: background_tasks_python
    image: webapp:latest
    env_file:
      - containerisation/.env
    environment:
      - VAULT_ADDR=http://x.x.x.x:portx
    volumes:
      - ./webapp:/webapp
    command: /bin/bash -c "vault kv get -
field=password kv/srvc | /usr/bin/kinit
srvc@DOMAIN; python /webapp/manage.py process_tasks --settings=dev"
    restart: always
  background_cache_calls:
    container_name: background_cache_calls
    image: webapp:latest
    env_file:
      - containerisation/.env
    environment:
      - VAULT_ADDR=http://x.x.x.x:portx
    volumes:
      - ./webapp:/webapp
    command: /bin/bash -c "python /webapp/cache_calls.py"
    restart: always
  mssql_default:
    container_name:  mssql_default
    image: mcr.microsoft.com/mssql/server:2019-latest
    environment:
      - ACCEPT_EULA=Y
      - SA_PASSWORD=xxxxxxxxxxxxx
networks:
  default:
    external:
      name: dbnet
```

# Anlage II: Container Builddefinition als Dockerfile

```
FROM python:3.10-bullseye

# Azure DevOps Token for accessing Python Repos
ARG token

# Creating Directories to copy webapp and install stuff
RUN mkdir -p /ssp/cachedb
RUN mkdir install

# Copying files needed for execution or install
COPY requirements.txt /install
COPY containerisation/webapp_container/krb5.conf /etc/
COPY webapp/ /webapp/

COPY containerisation/webapp_container/Entrypoint.sh /

# Installation of neccessary libraries and helper programs
RUN apt -y update
RUN ACCEPT_EULA=Y apt install -y lsb-release gpg ca-certificates
RUN curl https://packages.microsoft.com/keys/microsoft.asc | apt-key
add -
RUN . /etc/os-release; curl
https://packages.microsoft.com/config/debian/$VERSION_ID/prod.list
> /etc/apt/sources.list.d/mssql-release.list
RUN wget -O- https://apt.releases.hashicorp.com/gpg | gpg --dearmor
| tee /usr/share/keyrings/hashicorp-archive-keyring.gpg >/dev/null
RUN echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-
keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs)
main" | tee /etc/apt/sources.list.d/hashicorp.list
RUN apt update
RUN ACCEPT_EULA=Y apt install -y libsasl2-dev libldap2-dev vault
krb5-user unixodbc-dev msodbcsql17

# Pip config for installation of libraries
RUN mkdir -p ~/.pip/
RUN touch ~/.pip/pip.conf
RUN echo "[global]\nextra-index-url =
https://AzureOrganisation:${token}@pkgs.dev.azure.com/AzureOrganisa
tion/packaging/AzureOrganisation/pypi/simple" > ~/.pip/pip.conf

# Installation of needed python libraries
RUN pip install --upgrade pip
RUN pip install -r /install/requirements.txt
RUN find /webapp -type f -name "requirements.txt" -exec echo {} \; -
exec pip install -r {} \;
RUN pip install debugpy

# cleanup
RUN rm -rf /install ~/.pip/pip.conf

# predefined Entrypoint
CMD ["/bin/bash", "/Entrypoint.sh"]
```

# Anlage III: Azure DevOps Pipelinedefinition in yaml

```yaml
trigger:
- main
pool:
  vmImage: ubuntu-latest
variables:
  imageName: "webapp
  DOCKER_BUILDKIT: 1
  containerRegistry: ctreg
  isMain: $[eq(variables['Build.SourceBranch'], 'refs/heads/main')]
  tag: $(Build.SourceVersion)
stages:
  - stage: automaticTests
    displayName: Automatic Tests
    jobs:
      - job: Formatting
        steps:
        - task: PipAuthenticate@1
          inputs:
            artifactFeeds: 'azure'
            onlyAddExtraIndex: true
        - task: Bash@3
          displayName: Prepare environment
          inputs:
            targetType: 'inline'
            script: |
              sudo apt install -y libkrb5-dev libsasl2-dev libldap2-dev
              pip install -r requirements.txt
        - task: Bash@3
          displayName: Formater
          inputs:
            targetType: 'inline'
            script: |
              pip install black==22.3.0
              python -m black --check ./webapp/ -l 79
        - task: Bash@3
          displayName: Linter
          inputs:
            targetType: 'inline'
            script: |
              pip install pylint==2.15.3
              # python -m pylint -j 0 ./webapp
      - job: Quality
        steps:
        - task: Bash@3
          displayName: QualityChecks
          inputs:
            targetType: 'inline'
            script: |
              pip install xenon==0.7.0
              python -m xenon ./webapp/ --max-absolute C --max-modules C
```

```yaml
- stage: build
    displayName: Build Docker Image#
    jobs:
      - job: BuildImage
        steps:
        - task: Docker@2
          displayName: BuildImage
          inputs:
            command: build
            repository: $(imageName)
            tags: $(tag)
            Dockerfile: containerisation/ssp_container/dockerfile
            buildContext: .
            arguments: --build-arg token=$(token)
        - task: Bash@3
          displayName: SecurityScans
          inputs:
            targetType: 'inline'
            script: |
              ./trivy image --exit-code 0 $(imageName):$(tag)
        - task: Docker@2
          displayName: SaveImage
          inputs:
            command: 'save'
            arguments: '$(imageName) -o $(Build.ArtifactStagingDirect
                    tory)/container.tar'
        - task: PublishBuildArtifacts@1
          displayName: PublishArtifact
          inputs:
            PathtoPublish: '$(Build.ArtifactStagingDirectory)'
            ArtifactName: '$(imageName)'
            publishLocation: 'Container'
  - stage: publish
    displayName: Publish Image to ACR
    jobs:
      - job: PublishImage
        steps:
        - task: Docker@2
          displayName: Login to ACR
          inputs:
            command: login
            containerRegistry: $(containerRegistry)
        - task: Docker@2
          displayName: Tag Image SourceVersion
          inputs:
            command: 'tag'
            arguments: '$(imageName):$(tag) $(containerRegistry).azur
                    ecr.io/$(imageName):$(tag)'
        - task: Docker@2
          displayName: Tag Image Latest
          inputs:
            command: 'tag'
            arguments: '$(imageName):$(tag) $(containerRegistry).azu
                    recr.io/$(imageName):latest'
        - task: Docker@2
          displayName: Push Image to ACR
          inputs:
            command: push
            repository: $(imageName)
            tags: |
              $(tag)
              latest
        - task: Docker@2
          displayName: Logout of ACR
          inputs:
            command: logout
```

```yaml
- stage: gitops
  displayName: Update GitOps Repository
  jobs:
    - job: UpdateManifest
      steps:
      - checkout: git://webapp/webapp-GitOps
        persistCredentials: true
      - task: Bash@3
        condition: and(succeeded(), eq(variables.isMain, 'false'))
        displayName: Update helm tst
        inputs:
          targetType: 'inline'
          script: |
           sed 's/CONTAINER_REGISTRY/$(containerRegistry)/g' values.yaml
           > values_tst.yaml
            sed -i 's/IMAGE_NAME/$(imageName)/g' values_tst.yaml
            sed -i 's/IMAGE_TAG/$(tag)/g' values_tst.yaml
            cat values_tst.yaml
      - task: Bash@3
        displayName: Git commit and push
        inputs:
          targetType: 'inline'
          script: |
            git config --global user.email "ci-pipeline@unternehmen.com"
            git config --global user.name "CI-Pipeline"
            git checkout -b main
            git add --all
            git commit -a -m "$(tag)"
            git status
            git push --force origin main
```

# Literaturverzeichnis

Anderson, Charles (2015): Docker [Software engineering]. In: IEEE Softw. 32 (3), 102-c3. DOI: 10.1109/MS.2015.62.

Ayres, Nicholas; Deka, Lipika; Paluszczyszyn, Daniel (2021): Continuous Automotive Software Updates through Container Image Layers. In: Electronics 10 (6), S. 739. DOI: 10.3390/electronics10060739.

Bass L., "The Software Architect and DevOps," in IEEE Software, vol. 35, no. 1, S. 8-10, January/February 2018, DOI: 10.1109/MS.2017.4541051.

Beetz, Florian; Harrer, Simon (2022): GitOps: The Evolution of DevOps? In: *IEEE Softw.* 39 (4), S. 70–75. DOI: 10.1109/MS.2021.3119106.

Bernstein, David (2014): Containers and Cloud: From LXC to Docker to Kubernetes. In: *IEEE Cloud Comput.* 1 (3), S. 81–84. DOI: 10.1109/MCC.2014.51.

Bhat, S. (2022). Understanding the Dockerfile. In: Practical Docker with Python. Apress, Berkeley, CA. DOI: 10.1007/978-1-4842-7815-4_4

Docker Dokumentation (2022): get-started. Online verfügbar unter https://docs.docker.com/get-started/, zuletzt aktualisiert am 2022, zuletzt geprüft am 02.12.2022.

GOURLEY, David (2002): HTTP: the definitive guide.: O'Reilly Media, Inc.

Huang Z., S. Wu, S. Jiang and H. Jin, "FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container," in 2019 35th Symposium on Mass Storage Systems and Technologies (MSST), Santa Clara, CA, USA, 2019,S. 28-37, doi: 10.1109/MSST.2019.00-18.

Joy, Ann Mary (2015): Performance comparison between Linux containers and virtual machines. In: 2015 International Conference on Advances in Computer Engineering and Applications. 2015 International Conference on Advances in Computer Engineering and Applications (ICACEA). Ghaziabad, India, 19.03.2015 - 20.03.2015: IEEE, S. 342–346.

Kubernetes Dokumentation (2022): Online verfügbar unter https://kubernetes.io/docs, zuletzt aktualisiert am 2022, zuletzt geprüft am 01.12.2022.

Makam, Vasanth K. (2020), "Continuous Integration on Cloud Versus on Premise: A Review of Integration Tools." In: Advances in Computing 10.1: S. 10-14. DOI: 10.5923/j.ac.20201001.02

M. Meyer, "Continuous Integration and Its Tools," in IEEE Software, vol. 31, no. 3, S. 14-16, May-June 2014, doi: 10.1109/MS.2014.58.

F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran and F. Massacci, "Understanding the Security Implications of Kubernetes Networking" in IEEE Security & Privacy, vol. 19, no. 05, pp. 46-56, 2021.DOI: 10.1109/MSEC.2021.3094726

Ramadoni, E. Utami and H. A. Fatta, "Analysis on the Use of Declarative and Pull-based Deployment Models on GitOps Using Argo CD," 2021 4th International Conference on Information and Communications Technology (ICOIACT), Yogyakarta, Indonesia, 2021, S. 186-191, DOI: 10.1109/ICOIACT53268.2021.9563984.

Rossberg, J. (2019). An Overview of Azure DevOps. In: Agile Project Management with Azure DevOps. Apress, Berkeley, CA., S.54-56, DOI: 10.1007/978-1-4842-4483-8_2

Ruan, B., Huang, H., Wu, S., Jin, H. (2016). A Performance Study of Containers in Cloud Environment. In: Wang, G., Han, Y., Martínez Pérez, G. (eds) Advances in Services Computing. APSCC 2016. Lecture Notes in Computer Science, vol 10065. Springer, S. DOI: 10.1007/978-3-319-49178-3_27

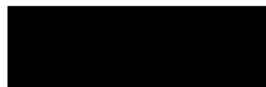
Satapathi, A., Mishra, A. (2021). Deploying Your Azure Functions Using a CI/CD Pipeline with Azure DevOps. In: Hands-on Azure Functions with C#. Apress, Berkeley, CA. DOI:10.1007/978-1-4842-7122-3_15

Shahin, Mojtaba; Ali Babar, Muhammad; Zhu, Liming (2017): Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. In: *IEEE Access* 5, S. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als die angegebenen verwendet habe.

Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Waldheim, 23.12.2022 (Ort, Datum)  (Unterschrift)