

TH KÖLN UND FACHHOCHSCHULE DORTMUND

Technology
Arts Sciences
TH Köln

Fachhochschule
Dortmund
University of Applied Sciences and Arts

VERBUNDSTUDIENGANG WIRTSCHAFTSINFORMATIK

ABSCHLUSSARBEIT
ZUR ERLANGUNG
DES MASTERGRADES
MASTER OF SCIENCE


„VERGLEICH VON CI/CD UND GITOPS IM DEVOPS-KONTEXT“


„COMPARISON OF CI/CD AND GITOPS IN DEVOPS“

Erstprüferin: Prof.Dr. Heide Faeskorn-Woyke

Zweitprüfer: Prof.Dr. Frank Victor

von cand. Artem Lajko

Matr.-Nr.: 

Wohnort: 

E-Mail: 

vorgelegt am: Dortmund, 29. September 2022

Kurzfassung

Die Digitalisierung hat Auswirkung auf die Entwicklung von Produkten. Die Teams werden agiler, die Kunden haben als Eigentümer eines Produktes Mitspracherecht und bestimmen gemeinsam mit den Entwicklern, welche Features als Nächstes aus dem vordefinierten Backlog in dem nächsten Sprint als Inkrement eingebaut und ausgeliefert werden. Durch die enge Zusammenarbeit werden die Release-Zyklen eines Produktes kürzer und es werden mehr Features als in der Vergangenheit bei nicht agilen Methoden deployt. Durch die kurzen Release-Zyklen werden viele unterschiedlichen Versionen ausgerollt. Dies hat zur Folge, dass Fehler, die vorher durch das Testen nicht entdeckt worden sind, auf einer produktiven Umgebung ausgerollt werden. Hat der Fehler eine starke Auswirkung auf die Benutzung der Anwendung, dann besteht die Möglichkeit, die Änderung manuell durchzuführen. Diese muss beim nächsten Release eingebaut werden. Wenn die Release-Zyklen länger werden, birgt es die Gefahr, dass viele manuelle Änderungen in der Zeit entstehen. Der ausgerollte Zustand entspricht nach einer bestimmten Zeit nicht mehr dem eingeecheckten Zustand im Git-Repository. An der Stelle bietet der GitOps-Ansatz eine mögliche Abhilfe, um den ausgerollten Ist-Zustand durch regelmäßiges Pullen so nah wie möglich an dem im Repository eingeecheckten Soll-Zustand zu halten.

Abstract

Digitalization has an impact on the development of products. The teams are becoming more agile, the customers, as owners of a product, have a say and determine together with the developers which features will be integrated and delivered next from the predefined backlog in the next sprint as an increment. Through close collaboration, the release cycles of a product become shorter and more features are rolled out than in the past with non-agile methods. Due to the short release cycles, many versions are rolled out. As a result, bugs that were previously undetected by testing are rolled out on a production environment. If the bug has a strong impact on the use of the application, then there is an option to make the change manually. This must be incorporated in the next release. If the release cycles become longer, there is a risk that many manual changes have been created in the time. After a certain time, the rolled out state does not correspond to the checked in state in the Git repository. At this point, the GitOps approach offers a possible remedy to keep the rolled-out actual state as close as possible to the target state checked in to the repository through regular rolling.

Inhaltsverzeichnis

1	Einleitung	8
1.1	Motivation	8
1.2	Zielsetzung	9
1.3	Aufbau der Arbeit	10
2	Grundlagen	11
2.1	Cloud-Computing	11
2.1.1	Cloud-Services	13
2.2	Virtuelle Maschinen vs Container	14
2.2.1	Virtuelle Maschinen	15
2.2.2	Container	15
2.2.3	Fazit	17
2.3	Architektonische Entwürfe von Software-Entwicklung	17
2.3.1	Traditionelles Anwendungsdesign	17
2.3.2	Serviceorientierte-Architektur	18
2.3.3	Microservices	19
2.3.4	Cloud-Native-Design	20
2.3.5	Cloud-Native-Design mit Kubernetes	20
2.4	Orchestrierung	21
2.5	Agile Methoden	24
2.6	DevOps	26
2.7	Site Reliability Engineering	28
2.8	DevOps vs SRE	29
2.9	Das Spotify Modell	30
2.10	Netzwerk	32
2.10.1	DNS	32
2.10.2	Domain	33
2.10.3	Root-CA, Intermediate und Server-Cert	34
2.10.4	TLS/SSL	35
2.10.5	Public-Key-Infrastruktur (PKI)	36
2.10.6	Zertifikatsaussteller (Cert-Issuer)	37
2.10.7	Letsencrypt	37
2.10.8	Container-Netzwerke	39
3	Kubernetes und Azure-Kubernetes-Service	40
3.1	Kubernetes im Überblick	40
3.2	Kubernetes Architektur	41
3.2.1	Nodes: Control-Planes und Worker-Nodes	41
3.2.2	Control-Plane-Komponente: API-Server	42
3.2.3	Control-Plane-Komponente: Controller Manager	43
3.2.4	Control-Plane-Komponente: Scheduler	44
3.2.5	Control-Plane-Komponente: Key-Value-Store etcd	45
3.2.6	Control-Plane- und Workernode-Komponente: kubelet	46
3.2.7	Control-Plane- und Workernode-Komponente: Container-Enginge	47

3.2.8	Control-Plane- und Workernode-Komponente: kube-proxy	47
3.2.9	Networking in Kubernetes und CNI	47
3.2.10	Control-Plane- und Workernode-Komponente: kube-dns	48
3.3	Managed Kubernetes - Azure-Kubernetes-Service	49
3.3.1	Cloud-Controller-Manager	50
3.3.2	Architektur	50
3.4	Stateless vs Statefull	51
3.5	Operatoren	52
4	Continuous Integration, Continuous Delivery und GitOps	53
4.1	Continuous Integration, Continuous Delivery und Continuous Deployment	53
4.2	GitOps	55
4.3	DevOps-Tools	57
4.3.1	Git	57
4.3.2	Flux	57
4.3.3	Helm	60
4.3.4	Kustomization	61
4.3.5	Sealed-Secrets-Operator	63
4.3.6	externalDNS	63
4.3.7	Azure DevOps Pipelines	63
4.3.8	Terraform	64
5	Design und Implementierung	66
5.1	Anforderungen: CI/CD und GitOps	66
5.2	Bootstrappig	67
5.2.1	Flussdiagramm: Bootstrapping „Azure-Backend“	67
5.2.2	Flussdiagramm: Bootstrapping „Azure-DevOps“	69
5.2.3	Flussdiagramm: Bootstrapping „Kubernetes und Flux-Stack“	69
5.3	Architekturen	71
5.4	Anwendungsfall: Deployment CD vs Flux	76
5.5	Anwendungsfall: manuelle Änderung – Skalierung, Deployment löschen und Umgebungsvariable manipulieren	77
5.5.1	Manuelle Änderung: Skalierung	77
5.5.2	Manuelle Änderung: Deployment löschen	80
5.5.3	Manuelle Änderung: Umgebungsvariable im Container manipulieren	82
6	Ergebnisse	85
6.1	Anwendungsfall: Deployment CD vs Flux	85
6.2	Manuelle Änderung: Skalierung	86
6.3	Manuelle Änderung: Deployment löschen	87
6.4	Manuelle Änderung: Umgebungsvariable im Container manipulieren	89
7	Fazit und Ausblick	90
	Literaturverzeichnis	92
	A Erklärung	95
	B Disclaimer	96

Abbildungsverzeichnis

2.1	Einordnung und Vergleich zwischen IaaS, PaaS und SaaS	14
2.2	Einordnung von CaaS zwischen IaaS und PaaS	15
2.3	Vergleich: VMs vs Container	16
2.4	Visualisierung einer Three-Tier-Architecture	18
2.5	Service-Oriented-Architecture	19
2.6	Container-Orchestrierungs-Schichten	23
2.7	SCRUM	26
2.8	Das Spotify-Modell	32
2.9	Root-CA-Chain-Trust	35
2.10	Sequenzdiagramm: Let´s Encrypt ACME-Challenge	38
3.1	Kubernetes-Kernkomponenten	41
3.2	Kubernetes-Kernkomponenten	45
3.3	Zusammenspiel Kubernetes-Networking, Kube-DNS und Kube-Proxy	49
3.4	Kommunikation zwischen <i>Pod</i> in unterschiedlichen Namespaces	49
3.5	Azure Kubernetes Service Kernkonzept	51
4.1	CI/CD	54
4.2	Flux Architektur	59
4.3	Helm-Diagramm Beispiel Baumansicht	61
4.4	Kustomize Base und Overlay Beispiel Baumansicht	62
4.5	Vergleich kustomization.yaml und flux-kustomization.yaml	62
5.1	Tabelle: Azure Anforderungen	66
5.2	Tabelle: Azure DevOps Anforderungen	66
5.3	Flussdiagramm: Initiales Bootstrapping „Azure-Backend“	68
5.4	Flussdiagramm: Bootstrapping „Azure-DevOps“	69
5.5	Flussdiagramm: Bootstrapping „Kubernetes und Flux-Stack“	70
5.6	Abstrahierte Gesamtarchitektur	72
5.7	Flux Referenzen Architektur	75
5.8	Zustandsdiagramm: Flux-Flow	76
5.9	Deployment CD vs Flux	77
6.1	Deployment CD vs Flux Dateibaum	85
6.2	Deployment CD vs Flux Dateibaum	88

Abkürzungsverzeichnis

CPU	Central-Processing-Unit
SaaS	Software-as-a-Service
FaaS	Function-as-a-Service
PaaS	Platform-as-a-Service
IaaS	Infrastructure-as-a-Service
DBaaS	Database-as-a-Service
CaaS	Communication-as-a-Service
SSD	Solid-State-Drive
DBaaS	Database-as-a-Service
CaaS	Container-as-a-Service
GCP	Google-Cloud-Platform
CI/CD	Continuous-Integration and Continuous-Delivery
SRE	Site-Reliability-Engineering
NIST	National Institute of Standards and Technology
SOA	Serviceorientierte-Architektur
CSP	Cloud-Service-Provider
MVC	Model-View-Controller
BIOS	Basic-Input-Output-System
UEFI	Unified-Extensible-Firmware-Interface
API	Application-Programming-Interface
ROI	Return-On-Investment
CAMS	Culture-Automation-Measurement-Sharing
SLI	Service-Level-Indicators
SLA	Service-Level-Agreements
SLO	Service Level Objectives
MVP	Minimum-Viable-Product
PKI	Public-Key-Infrastructure
SSL	Secure-Socket-Layer
TLS	Transport-Layer-Security
CSR	Certificate-Signing-Request
DNS	Domain-Name-System
TLD	Top-Level-Domain
SLD	Second-Level-Domain
DN	Distinguished-Name
ISRG	Internet-Security-Research-Group

AKS Azure-Kubernetes-Service
HDD Hard-Disk-Drive
SSD Solid-State-Drive
CNCF Cloud-Native-Computing-Foundation
GKE Google-Kubernetes-Engine
GCP Google-Cloud-Platform
CLI Command-Line-Interface
CRI Container-Runtimes-Interface
CRD Custom-Ressource-Definitions
gRPC general-purpose-Remote-Procedure-Calls
CIDR Classless Inter-Domain-Routing
NFS Network-File-Share
NAT Network-Address-Translation
CNI Container-Network-Interface
CI Continuous-Integratio
CDE Continuous-Delivery
CD Continuous-Deploymen
CT Continuous-Testing
RBAC Role-Based-Access-Control
CR Custom-Resources
CRD Custom-Resource-Definition
AAD Azure-Active-Directory
IAM Identity-and-Access-Management
PAT Personal-Access-Token
UML Unified-Modeling-Language
VNET Virtual-Network

1 | Einleitung

1.1 Motivation

Durch die Digitalisierung (Bendel, 2021) verändert sich nicht nur die Geschwindigkeit in den IT-Abteilungen durch Einführung von neuen Konzepten und *Tools*. Die Digitalisierung hat auch Auswirkung auf die Entwicklung von Produkten. Die Teams werden agiler, die Kunden (Heath (2021), S. 6-8) haben als Eigentümer eines Produktes Mitspracherecht und bestimmen gemeinsam mit den Entwicklern, welche Features als Nächstes aus dem vordefinierten Backlog in dem nächsten Sprint als Inkrement eingebaut und ausgeliefert werden. Durch die enge Zusammenarbeit werden die Release-Zyklen eines Produktes kürzer und es werden mehr Features als in der Vergangenheit bei nicht agilen Methoden ausgerollt. Durch die kurzen Release-Zyklen werden viele unterschiedliche Versionen ausgerollt. Um den Überblick über die gemachten Änderung im Zusammenhang mit einer Version zu behalten, erfolgt die Versionierung durch die Vergabe von *Tags*. Die neuen Features werden nach ausgiebigem Testen (KitelyTech, 2021) der Entwickler, sowie Fachabteilungen und Fachanwendern ausführliche getestet, bevor diese in einer produktiven Umgebung ausgerollt werden. In der Regel gibt es dafür unterschiedliche Stages wie zum Beispiel *Development*, *Integration* und *Production*. In der *Development-Stage* erfolgt die hauptsächliche Entwicklung und deswegen stellt die *Stage* eine instabile Umgebung dar. Auf der *Integration-Stage* (Böhlke, 2021) werden die stabilen Entwicklungsfortschritte aus der *Development-Stage* übernommen. Die *Stage* wird dann in der Regel den Fachabteilungen und Fachanwendern zum Testen zur Verfügung gestellt. Trotz ausführlicher Tests lässt sich die Abwesenheit von Fehlern nicht ausschließen. Dies hat zur Folge, dass Fehler, die vorher durch das Testen nicht entdeckt worden sind, auf einer produktiven Umgebung ausgerollt werden. Der Fehler wird dann durch einen Anwender meistens in Form eines Tickets an den angegebenen Support gemeldet. Das Support-Team ordnet das Ticket im Kontext ein und leitet es an die Entwicklungsabteilung weiter. Wenn der Fehler eine starke Auswirkung auf die Benutzung der Anwendung hat, dann besteht die Möglichkeit, die Änderung manuell in Zusammenarbeit mit dem *Operation-Team* nach Absprache mit dem Kunden durchzuführen. Die manuelle Änderung muss dann von Entwicklern in das nächste Release (Heath (2021), S. 49) eingebaut werden, damit die manuelle Änderung bei dem nächsten Release nicht überschrieben wird. Dies liegt in der Verantwortung der Entwicklungsabteilung. Um einen sicheren Betrieb einer produktiven Umgebung zu gewährleisten, erfolgt eine strikte Trennung zwischen dem Infrastruktur- und Entwickler-Team. Die Entwickler haben keinen schreibenden Zugriff auf die produktiven Infrastrukturkomponenten. Damit soll sichergestellt werden, dass alle Änderungen nur über Releases und Versionierung erfolgen, um die Nachvollziehbarkeit zu gewährleisten. Jetzt kann ein weiteres Release vom Entwickler-Team gebaut werden, welches die fehlende Konfiguration ausrollt. Dies stellt erstmal für die Entwickler allgemein kein Problem dar. Dazu wird ein Zwischen-Release in Absprache mit dem *Product-Owner* (Heath (2021), S. 32-33) gebaut und mit einer speziellen *Minor-Version* ausgerollt. Die spezielle *Minor-Version* wie zum Beispiel "1.7.3", ist notwendig, da in der Sprintplanung für einen Sprint bereits eine Version für das nächste Release feststeht. Wenn es sich jedoch um ein produktives System handelt, dann kann eine Änderung nicht ohne *Downtime* ausgerollt werden. Das *Deployment* rollt in der Regel nicht nur eine einzelne Konfiguration aus, sondern baut eine Anwendung neu, testet alle Abhängigkeiten und gibt diese anschließend zum *Deployment* frei. Das *Deployment* eines Releases (Böhlke, 2021) erfolgt über eine konfigurierte *Pipeline*, die durch eine Änderung auf einer bestimmten *Stage* angestoßen (push) wird. Wenn die Release-Zyklen län-

ger werden, birgt es die Gefahr, dass viele manuelle Änderungen in der Zeit entstanden sind. Der ausgerollte Zustand entspricht nach einer bestimmten Zeit nicht dem eingeeckten Zustand im *Git-Repository* (Heath (2021), S. 114-115). Das Delta zwischen dem ausgerollten realen Ist-Zustand und dem eingeeckten Quellcode im *Git-Repository* Soll-Zustand wird immer größer.

Bei der Erfassung des beschriebenen Problems handelt es sich um einen nicht üblichen täglichen Zustand in der Entwicklung. Es wurde dabei nicht berücksichtigt, dass durch die Fluktuation (Nollau, 2018) in der IT-Branche, eine Rotation im Team stattfindet. Des Weiteren kommen noch weitere Aspekte, die sich gut oder weniger gut planen lassen wie Krankheit oder Urlaub der Mitarbeiter. Im Idealfall sind alle manuellen Änderungen, die angefallen sind, in einem Backlog als *To-Dos* transparent für das gesamte Team hinterlegt. In der Realität bestehen durch die unterschiedlichen Arbeitsweisen und das unterschiedliche Verantwortungsgefühl der Mitarbeiter eine Abweichung zum Idealfall. Der Idealfall kann nicht als Standard für die Zusammenarbeit im Team gesetzt werden. An der Stelle bietet der GitOps-Ansatz (Billy et al. (2021), S. 3-6) eine mögliche Abhilfe, um den ausgerollten Ist-Zustand durch regelmäßiges *pullen* so nah wie möglich an dem im *Repository* eingeeckten Soll-Zustand zu halten. Dies wird durch eine Synchronisierung des *Git-Repositorys*, die in einem fest definierten Intervall erfolgt, realisiert.

1.2 Zielsetzung

Zum Abschluss der Masterthesis soll der traditionelle Continuous-Integration and Continuous-Delivery (CI/CD) Ansatz (*Push*) und der GitOps-Ansatz (*Pull*) im DevOps Kontext gegenübergestellt und verglichen werden. Dabei werden die notwendigen Grundlagen, die Einordnung und die Entwicklung des GitOps-Ansatzes schwerpunktmäßig auf Kubernetes¹ betrachtet, da die Idee und Weiterentwicklung im Kubernetes-Umfeld entstanden ist. Die Arbeit soll das Verständnis für die Bedeutung von der Entwicklung von CI/CD und GitOps durch die Zusammenarbeit ehemals getrennter Gruppen wie Development und Operation, welche die DevOps-Kultur darstellen, erhöht werden. Bei der schriftlichen Ausarbeitung liegt der Fokus auf Development und Operation. Die Sicherheit, die DevOps, um *DevSecOps* erweitert, wird nur im Rahmen der Notwendigkeit betrachtet. Für den Vergleich wird eine Kubernetes-Distribution von Microsoft als Infrastruktur verwendet, um die Implementierung sowohl von CI/CD als auch von GitOps zu ermöglichen. Diesbezüglich erfolgt eine Anforderungsanalyse für die notwendigen Komponenten, eine architektonische Designentscheidung und anschließend die Implementierung. Die Arbeit beschreibt nicht *Best-Practices* für das Verwenden einer CI/CD- oder GitOps-Lösung. Durch die schriftlichen Ausarbeitungen sollen Tendenzen ausgearbeitet werden, für welche Teams und welche Projekte welche Lösung sinnvoll sind. Dies basiert auf den Ergebnissen, die im Rahmen der Arbeit entstanden sind.

Schritte der Arbeit:

- Erarbeitung der notwendigen Grundlagen
- Vertiefung der Themen Kubernetes, CI/CD und GitOps
- Design von unterschiedlichen Architekturen
- Implementierung einer Azure-Kubernetes-Service-Infrastruktur

¹<https://kubernetes.io>, letzter Zugriff 25.06.2022

- Implementierung einer CI/CD-Lösung und einer GitOps-Lösung
- Vergleich der unterschiedlichen Lösungsansätze
- Ausrollen einer Beispiel-Anwendung mit CI/CD und GitOps zu Demo-Zwecken

1.3 Aufbau der Arbeit

Die vorliegende Arbeit wird in sieben Kapitel gegliedert. Zuerst wird die Struktur sowie das zu lösende Problem vorgestellt. In Kapitel 2 werden Grundlagen, die für das Verständnis der Arbeit notwendig sind, erarbeitet und erläutert. Außerdem beschäftigt sich das Kapitel mit der Einordnung von CI/CD und GitOps im DevOps-Kontext. Es findet außerdem in dem Kapitel eine Unterscheidung und Einordnung zwischen agilen Methoden, Site-Reliability-Engineering (SRE) und DevOps statt. Kapitel 3 liefert einen tieferen Einblick in die Orchestrierungsplattform Kubernetes und deren Distribution *Managed Kubernetes* von Microsoft. Das Kapitel 4 beschäftigt sich ausführlicher mit der Erläuterung von CI/CD und GitOps. Darüber hinaus erfolgt in dem Kapitel der theoretische Vergleich bzw. die Differenzierung sowie die Untersuchung der Gemeinsamkeiten der beiden zuvor beschriebenen Methoden. Das Kapitel 5 beschäftigt sich mit dem Designentwurf von Architekturen und der Implementierung einer Infrastruktur, welche unter anderem ein Azure Kubernetes Service beinhaltet. Des Weiteren erfolgt in dem Kapitel die Beschreibung der Implementierung einer CI/CD- und GitOps-Lösung auf der Basis, der zuvor ausgerollten Infrastruktur. Es erfolgt außerdem eine Anforderungsanalyse für die Implementierung der beiden Lösungen. Anschließend werden in Kapitel 6 die Ergebnisse aus der bereits beschriebenen Implementierung in Kapitel 5, vorgestellt. Abschließend erfolgt in Kapitel 7 eine Zusammenfassung. Zum Schluss wird ein Fazit gezogen, für welche Einsatzfälle, welche der beiden Methoden in Frage kommt. Basierend auf den während der schriftlichen Ausarbeitung gewonnenen Erkenntnissen wird auch ein möglicher Ausblick über die Auswirkungen dieser beiden Ansätze im DevOps-Bereich und ihre zukünftigen Beiträge zu der weiteren Entwicklung von Cloud-Nativen-Infrastrukturen gegeben.

2 | Grundlagen

Das Kapitel beschreibt die notwendigen Grundlagen, die als Fundament für das Verständnis der Arbeit, sowie der darauffolgenden Kapitel dienen. Bei den Grundlagen wird unter anderem der Unterschied von *Cloud-Computing* zu anderen Computing-Techniken erläutert. Des Weiteren beschreibt das Kapitel die Entwicklung von einer klassischen N-Tier-Architektur zu einer Serviceorientierte Architekturen basierend auf *Microservices*, die unter anderem auch durch das *Cloud-Computing* vorangetrieben worden sind.

2.1 Cloud-Computing

Die Datenwolke (**engl.** Cloud-Computing) (Gai & Li, 2012) stellt eine IT-Infrastruktur, welche durch die Entwicklung des Internets und die Entwicklung der Web-Anwendungen vorangetrieben wurde. Das Konzept und der bekannte Begriff *Cloud-Computing* wurde erstmals im Oktober 2007 durch Google und IBM öffentlich vorgestellt. Im Jahr 2011 veröffentlichte das National Institute of Standards and Technology (NIST) eine Definition, die auf Akzeptanz bei der Öffentlichkeit stieß. Das Potenzial von *Cloud-Computing* (Varghese et al., 2019) wurde von Professor John McCarthy bereits vor mehr als 50 Jahren erkannt. Im Jahre 1961 sagte McCarthy bei MIT Centennial, dass *Computing* eines Tages als öffentliches Versorgungsunternehmen organisiert werden, so wie die Telefonanlage ein öffentliches Versorgungsunternehmen ist. Es ist anders gekommen als vorhergesagt. Zwar hat sich *Computing* durchgesetzt, aber nicht als öffentliches Versorgungsunternehmen. Die Beschreibung eines öffentlichen Versorgungsunternehmens wird bei *Utility-Computing*¹ durch das Management von IT-Services von einem technologieorientierten zu einem geschäftsorientierten Ansatz, gereicht. Dabei bekommt der Kunde von einem Serviceanbieter Rechenkapazität und die gesamte Infrastruktur inkl. dessen Management zur Verfügung gestellt. Der Kunde zahlt dabei nach dem Prinzip „*Pay-Per-Use*“ nur das, was er tatsächlich verbraucht. Eine Cloud ((Manvi & Shyam, 2021), S. 1-2) ist eine Dienstleistung und wird definiert als ein Raum über eine Netzwerkinfrastruktur, in dem Computerressourcen wie Computerhardware, Speicher, Storage, Datenbanken, Netzwerke, Betriebssysteme und sogar ganze Softwareanwendungen sofort und auf Abruf verfügbar sind. Es stimmt zwar, dass *Cloud-Computing* nicht viele neue Technologien beinhaltet, aber es stellt dennoch eine neue Art der IT-Verwaltung dar. So lassen sich beispielsweise Skalierbarkeit und Kosteneinsparungen in höchstem Maße durch *Cloud-Computing* erzielen.

Das *Cloud-Computing* ((Manvi & Shyam, 2021), S. 2-3) wird oft mit Serviceorientierte-Architektur (SOA) (**engl.** *Service-Oriented-Architecture*), *Grid Computing*¹, *Utility Computing* und *Cluster Computing* verglichen. Somit werden *Cloud-Computing* und *SOA* unabhängig voneinander verfolgt. Die Plattform- und Speicherdienste des *Cloud-Computing-Providers* wie Microsoft bieten einen Mehrwert für die Bemühungen um *SOA*. Mit Technologien wie *Grid Computing* können Rechenressourcen als Dienstprogramm verteilt über Netzwerke bereitgestellt werden. Bei *Cloud-Computing* hingegen erfolgt eine bedarfsgerechte Bereitstellung von Ressourcen und damit geht *Cloud-Computing* noch einen Schritt weiter. Es entfällt somit die Notwendigkeit einer Überversorgung, um den Anforderungen mehrerer Kunden gerecht zu werden. Beim *Utility-Computing* werden nur tatsächlich genutzte Ressourcen

¹<https://de.cloudflight.io/presse/was-ist-utility-computing-154/>, letzter Zugriff 21.06.2022

abgerechnet, ähnlich wie bei der Versorgung von öffentlichen Versorgungsleistungen von Strom oder Gas, bei welchen nur der tatsächliche Verbrauch in Rechnung gestellt wird.

Durch das Clustern ((Manvi & Shyam, 2021), S. 2-3) ergibt sich eine kostengünstigere Form der Verarbeitung von Anwendungen, die im Parallelbetrieb laufen. Um einen Cloud-Service zu nutzen, greifen Endnutzer über das Internet auf Cloud-Dienste wie Rechenressourcen in Form einer Virtuellen-Maschine zu. Der Benutzer benötigt für den Zugriff ein Konto beim Cloud-Service-Provider (CSP) für Sicherheits- und Abrechnungszwecke. Die benötigten Ressourcen werden von den Nutzern angegeben. Der CSP stellt die Ressourcen in Form von virtuellen Maschinen direkt den Benutzerkonten zur Verfügung. Dieses Angebot ermöglicht den Nutzern mehr Flexibilität bei der Erstellung ihrer eigenen Anwendungen auf der Grundlage von remote gehosteten Ressourcen. Die Nutzer mieten im wesentlichen Betriebssysteme, CPU, Arbeitsspeicher, Speicher und Netzwerkressourcen vom CSP, um die Effizienz und Skalierbarkeit der Arbeitslasten zu verbessern und an die eigenen Anwendungen bei Bedarf anzupassen. Somit beinhaltet *Cloud-Computing*, sowohl Grundzüge von *Grid-Computing* durch die Möglichkeit, mehrere Ressourcen über ein Netzwerk zu clustern, als auch Grundzüge von *Utility-Computing* durch Angebote wie „*Pay-As-You-Go*“² Ressourcen wie Virtuelle-Maschinen, bei denen auf Stunden-Basis nur die Rechenzeit abgerechnet wird, die auch tatsächlich verbraucht worden sind.

Die Tabelle 2.1 enthält eine Zusammenfassung der Merkmale zu jeder der beschriebenen Computing-Techniken.

Computing-Techniken	Merkmale
Cloud-Computing	- Kosteneffizient - nahezu unbegrenzte Speicherung - Sicherung und Wiederherstellung - einfache Bereitstellung
SOA	- Lose Kopplung und verteilte Verarbeitung
Grid-Computing	- Effiziente Nutzung ungenutzter Ressourcen - modular - Parallelität kann erreicht werden - überschaubare Komplexität
Cluster-Computing	- Verbesserte Netzwerktechnologie und Rechenleistung - reduzierte Kosten - Verfügbarkeit und Skalierbarkeit

Tabelle 2.1: Vergleich von verschiedenen Computer-Techniken und deren Merkmalen, Bildquelle: eigene Darstellung in Anlehnung an ((Manvi & Shyam, 2021), S. 2)

Bei *Cloud-Computing* (Microsoft, 2022d), ((Manvi & Shyam, 2021), S. 5) gibt es drei Arten von Cloud-Technologien. Diese werden im Folgenden beschrieben:

- **Public-Cloud:** Ressourcen befinden sich im Besitz externer *Cloud-Service-Provider* wie Microsoft Azure und werden von diesen ausgeführt. Die Bereitstellung von Ressourcen wie Server und Speicher erfolgt über das Internet. Die Infrastrukturkomponenten sind Eigentum des Cloudanbieters. Es ist ein *Cloud-Hosting*, bei dem die Cloud-Dienste über ein Netz bereitgestellt

¹<https://azure.microsoft.com/de-de/overview/what-is-grid-computing/>, letzter Zugriff 21.06.2022

²<https://azure.microsoft.com/en-us/pricing/purchase-options/pay-as-you-go/>, letzter Zugriff 21.06.2022

werden, das für die öffentliche Nutzung offen ist. Dieses Modell ist eine echte Darstellung des *Cloud-Hostings*, bei dem der Dienstanbieter Dienste und Infrastruktur für verschiedene Kunden bereitstellt.

- **Private-Cloud:** Cloud-Computing Ressourcen werden exklusiv von einem einzigen Unternehmen genutzt. Meistens physische lokale Dienste im Rechenzentren des Unternehmens. Infrastrukturkomponenten werden in einem privaten Netzwerk verwaltet. *Private-Cloud* wird in einer Cloud-basierten sicheren Umgebung implementiert, die durch eine Firewall geschützt ist. Die Firewall steht unter der Kontrolle der IT-Abteilung des jeweiligen Unternehmens. Die *Private-Cloud* erlaubt den Zugriff nur autorisierten Benutzern und gibt der Organisation eine größere und direkte Kontrolle über ihre Daten.
- **Hybrid-Cloud:** stellt eine Kombination aus *Public-* und *Private-Clouds* dar und nutzt dabei die Vorteile der beiden Arten aus. Daten und Anwendungen bewegen sich zwischen privaten und öffentlichen Clouds.

Die später in der Arbeit verwendeten Ressourcen sind *Cloud-Services* und deswegen liegt der Schwerpunkt der Arbeit bei *Cloud-Computing* auf *Public-Cloud*. Die meisten *Cloud-Computing-Services* lassen sich in vier grundlegende Kategorien unterteilen. Diese werden im nächsten Abschnitt genauer beschrieben.

2.1.1 Cloud-Services

Ein Cloud-Bereitstellungsmodell ((Manvi & Shyam, 2021), S. 2) stellt eine spezifische, vorgefertigte Kombination von IT-Ressourcen, die von einem Cloud-Anbieter angeboten wird, dar. Es gibt drei gängige Cloud-Bereitstellungsmodelle, die sich weitestgehend etabliert und formalisiert haben. Diese werden im Folgenden genauer erläutert ((Manvi & Shyam, 2021), S. 2), (Microsoft, 2022d)).

- **Infrastructure-as-a-Service (IaaS):** stellt die einfachste Kategorie von *Cloud-Computing* dar. Die Fähigkeit, die dem Kunden zur Verfügung gestellt wird, ist die Bereitstellung von Speicher-, Netzwerk- und anderen grundlegenden Rechenressourcen. Es werden IT-Infrastruktur-Komponenten, wie Server, virtuelle Computer, Speicher, Netzwerke und Betriebssysteme eines Cloudanbieters in Anspruch genommen. Die Gebühren für die Dienste werden nutzungsbasiert entrichtet. Ein Beispiel dafür ist, die Ausführung einer CPU-/Speicher-intensiven Anwendung auf der Microsoft Azure *IaaS* Cloud.
- **Platform-as-a-Service (PaaS):** stellt eine spezielle Form von *IaaS* zur Verfügung. Anstatt Infrastrukturkomponenten wird bei *PaaS* eine bedarfsgesteuerte Umgebung, z.B. für die Entwicklung von einer Softwareanwendung zur Verfügung gestellt. Die Entwicklung der Anwendung erfolgt ohne die Berücksichtigung der zugrunde liegenden Infrastruktur aus Servern, Speichern, Netzwerkkomponenten und einer spezifischen Laufzeitumgebung. Ein Beispiel dafür ist, die Erstellung und Bereitstellung einer Anwendung auf einem *App-Service*¹ von Microsoft Azure.
- **Software-as-a-Service (Software-as-a-Service (SaaS)):** ist wiederum eine spezielle Form von *PaaS* bei der keine Plattform, sondern eine Software über das Internet zur Verfügung gestellt wird. Es werden bei SaaS Softwareanwendungen und zugrunde liegende Infrastrukturen von *Cloud-Service Providern* gehostet und verwaltet. Ein Beispiel dafür ist das Öffnen von Word-Dateien mit *Outlook-Web-Apps* ohne Installation von MS Office Client auf dem Endgerät.

¹<https://azure.microsoft.com/de-de/services/app-service/>, letzter Zugriff 21.06.2022

Die Schichten bauen aufeinander ((Marvi & Shyam, 2021), S. 4) auf. Die SaaS-Schicht benötigt im Unterbau eine Plattform, auf der die Anwendung laufen kann. Die PaaS-Schicht benötigt wiederum einen Server, auf dem die Plattform ausgeführt werden kann und ist somit abhängig von der IaaS-Schicht. Deswegen umfasst der Begriff Cloud-Computing alle Schichten und wird oft als Synonym für verschiedene Arten von Clouddiensten verwendet. Es gibt weitere Services wie zum Beispiel Database-as-a-Service (DBaaS) (z.B. *Azure Database for MySQL*¹) oder Container-as-a-Service (CaaS) (z.B. *Voice over IP*) die auf den darunter liegenden Schichten basieren.

Die Abbildung 2.1 verdeutlicht die Verantwortlichkeiten des Kunden und die des Cloud-Service-Providers beim Beziehen eines Cloud-Services.

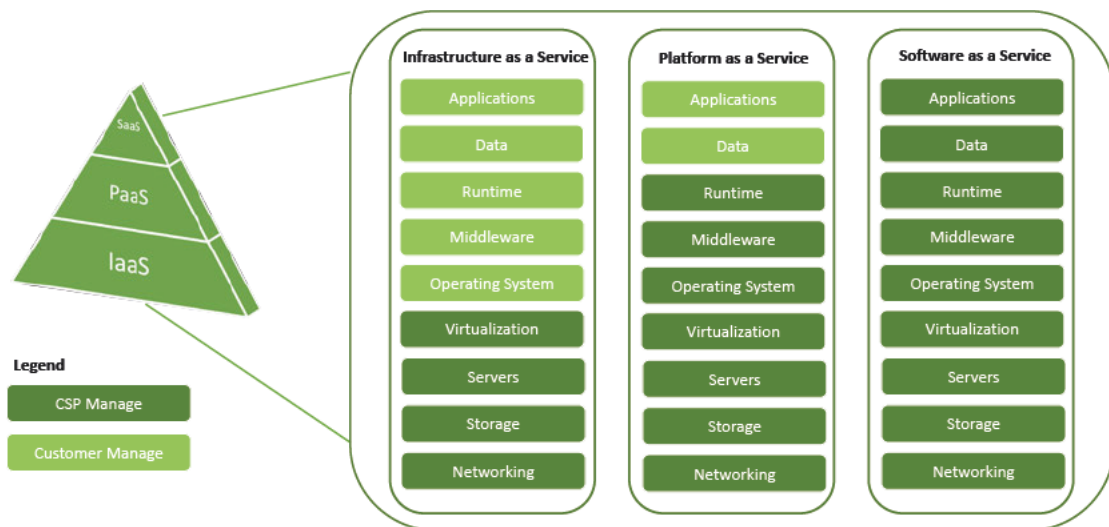


Abbildung 2.1: Einordnung und Vergleich zwischen IaaS, PaaS und SaaS, Bildquelle: eigene Darstellung in Anlehnung an ((Liebel, 2019) S.56)

2.2 Virtuelle Maschinen vs Container

Der klassische Einsatzschwerpunkt ((Liebel, 2019) S.56) der Cloud-Umgebungen, wie beispielsweise das Hosten von normalen virtuellen Maschinen, hat sich im Zuge der zunehmenden Verbreitung von Container- und Orchestrierungslösungen bereits deutlich verlagert. Die IaaS-Landschaften fungieren zunehmend als Plattformen für *Container as a Service* (kurz *CaaS*) und *PaaS*. Bei *CaaS* (siehe Abbildung 2.2) handelt es sich um eine Lösung zwischen *IaaS* und *PaaS*. Der Unterschied zu *IaaS* liegt darin, dass Container über kein eigenes Betriebssystem verfügen und das Betriebssystem des Host-Systems nutzen, um mit dem *Kernel* zu kommunizieren. Bei diesen Verfahren spricht man von Containerisierung einer oder mehrerer Anwendungen. Der genaue Unterschied zwischen Virtualisierung und Containerisierung wird im Folgenden erläutert.

¹<https://azure.microsoft.com/de-de/services/mysql/#overview>, letzter Zugriff 21.06.2022

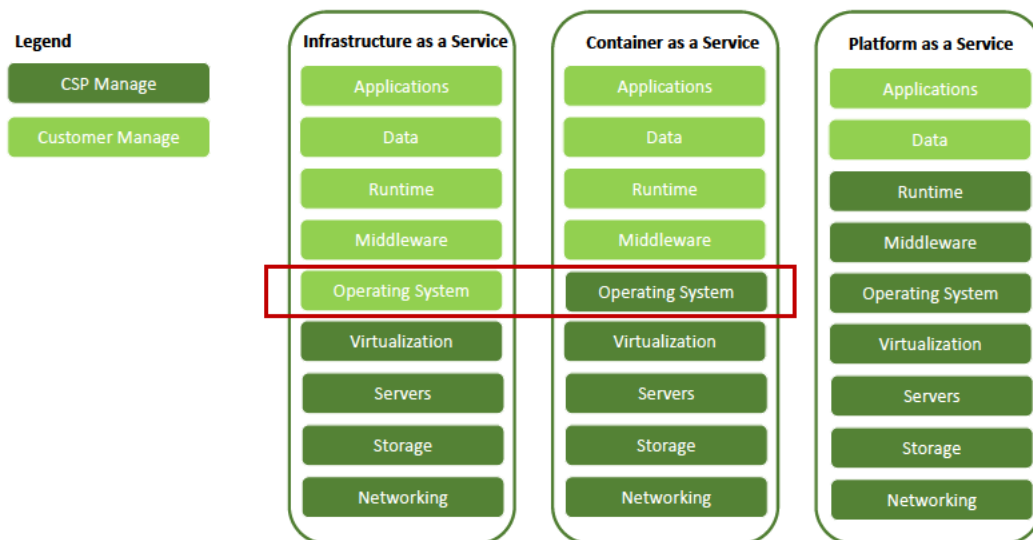


Abbildung 2.2: Einordnung von CaaS zwischen IaaS und PaaS, Bildquelle: eigene Darstellung in Anlehnung an ((Liebel, 2019) S.56)

2.2.1 Virtuelle Maschinen

Bei der **Virtualisierung** (Red Hat, Inc., 2020) trennt eine Software, die als „*Hypervisor*“ (siehe Abbildung 2.4 (a)) bezeichnet wird, die Ressourcen von der physischen Hardware. Durch die Trennung werden Ressourcen partitioniert und für VMs reserviert. Die Virtualisierung stellt Anforderungen an die Hardware und das darunter liegende Betriebssystem. Aus physischer Central-Processing-Unit (CPU), mit physischen Kernen (**engl.** *Cores*) entsteht eine virtuelle CPU (vCPU) mit virtuellen Kernen. Die Kommunikation und Provisionierung einer VM erfolgt über VM-Anweisungen eines Nutzers für die zusätzlichen physischen Ressourcen wie CPU, die der *Hypervisor* entgegennimmt. Die Anforderungen des Nutzers werden vom *Hypervisor* an das physische System durchgereicht. Die Änderungen bzw. die Anforderungen werden als Metadaten zwischengespeichert. Für den Nutzer wirken und verhalten sich VMs wie physische Server. Im Gegensatz zu Containern stellen VMs (Red Hat, Inc., 2020) ein eigenständiges Betriebssystem mit einem eigenen *Kernel* dar und sind deswegen nicht fest an den *Kernel* des unterliegenden Host-Systems gebunden. Die Abhängigkeit zu dem *Kernel* bedeutet, dass alle Container-Kinder zumindest von derselben Distribution kommen müssen. Dabei spielt es bei Linux Containern keine Rolle, ob es sich um CentOS¹ oder Ubuntu² handelt, solange alle Distributionen denselben *Kernel* unterstützen. Dadurch können mehrere Linux-Distributionen (siehe Abbildung 2.4 (a)) als Gast-Betriebssystem (**engl.** *Guest-OS*) parallel auf einem Server virtualisiert und betrieben werden.

2.2.2 Container

Bei **Containersierung** handelt es sich, um eine Technologie (Pahl et al., 2017) zur Virtualisierung (Red Hat, Inc., 2020) einer kompletten Anwendung. Die Anwendung läuft als *Microservice* (siehe Abbildung 2.4, (b)) oder eine *App* in einem Container. Darüber hinaus enthält ein Container auch alles,

¹<https://www.centos.org>, letzter Zugriff 23.05.2022

²<https://ubuntu.com>, letzter Zugriff 23.05.2022

was zu Ausführung einer *App* oder von *Microservices* benötigt wird. Der Inhalt eines Containers wird in Form eines *Images* abgebildet. Ein *Image* ist eine codebasierte Datei, die alle Abhängigkeiten und Libraries enthält. Dadurch, dass das *Image* alle RPM-Pakete und Konfigurationsdateien enthält, stellt es eine Installationsdatei einer leichtgewichtigen Linux-Distribution dar. Aufgrund der kleinen Größe von Containern werden normalerweise Hunderte von losen Containern, die miteinander gekoppelt werden können, in einer Umgebung gestartet. Es handelt sich bei Containern um eine leichtgewichtige virtuelle Maschinen, die kein eigenes Betriebssystem enthält. Container gibt es nicht erst seit Docker¹ ((Liebel, 2019), S. 80-81). Durch die Software Docker wurden Container zwar bekannt, aber der Gedanke entstand bereits in den 60ern des letzten Jahrtausends. Die ersten verwendeten *Hypervisor* auf Mainframes von IBM bei der Mainframe-Virtualisierung wurden als containerartig bezeichnet. Fast alle heutigen Container basieren auf der Funktionalität, die der *Linux-Kernel* seit Version 2.6.32 mitbringt, welcher am 3. Dezember 2009 (KernelNewbies, 2017) veröffentlicht wurde. Mit der Auslieferung des Kernels sind die sogenannten *Kernel Namensräume* (**engl.** *Namespaces*) erschienen. Die *Namespaces* ((Liebel, 2019), S. 82) ermöglichen es, Ressourcen des Kernsystems voneinander zu isolieren und diese Teile anderen Prozessen bzw. Anwendungen zur Verfügung zu stellen.

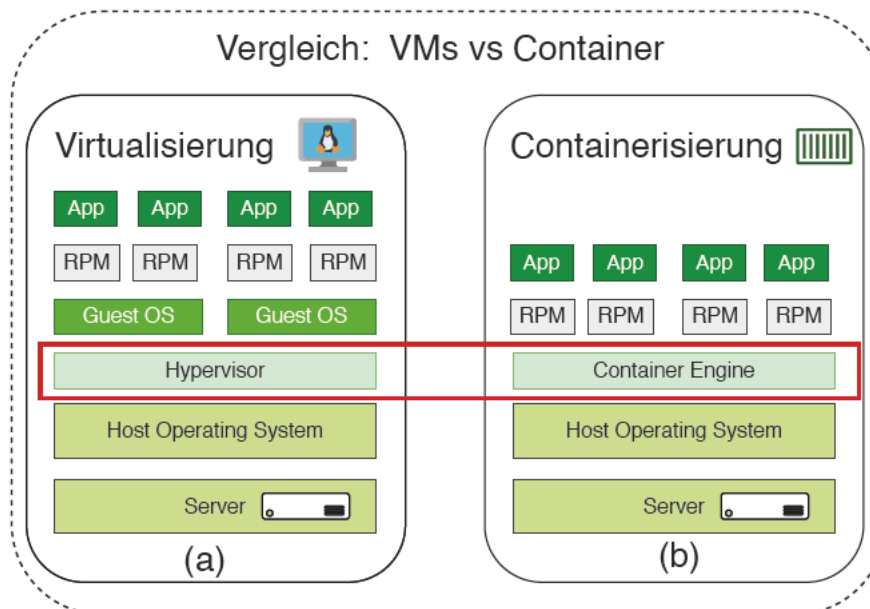


Abbildung 2.3: Vergleich: VMs vs Container, Bildquelle: eigene Darstellung in Anlehnung an (Red Hat, Inc., 2020)

Ein Container ist somit eine leichtgewichtige Virtuelle-Maschine ((Liebel, 2019), S. 88-90) in Form eines Prozesses, der über keinen eigenen *Kernel* verfügt und den *Kernel* des Host-Systems nutzt. Da es sich bei Containern um kein eigenständiges Betriebssystem handelt, wird weder Basic-Input-Output-System (BIOS) bzw. Unified-Extensible-Firmware-Interface (UEFI) noch eine virtuelle Hardware benötigt. Daher erfolgt der Bootvorgang bei Container-Prozessen in Sekundenbruchteilen. Es wird aufgrund der Schlankheit eines Containers eine höhere Packungsdichte als bei VMs erzielt. Zwar sind Container wesentlich schlanker als VMs, aber betrachtet man zum Beispiel die reine Konfiguration eines Apache *Webservers*² ((Liebel, 2019), S. 92), dann läuft der Apache *Webserver* in einem Container nicht mit weniger RAM als in einer VM.

¹<https://www.docker.com/>, letzter Zugriff 22.05.2022

²<https://httpd.apache.org>, letzter Zugriff 23.05.2022

2.2.3 Fazit

Container machen somit VMs nicht obsolet ((Liebel, 2019), S. 88) und langfristig betrachtet geht es nicht um eine Entweder-oder-Entscheidung. Die Ansätze, Anforderungen und Konzepte können sich stark zwischen IT-Landschaften unterscheiden. Deswegen gibt es vielfältige Lösungen, die aufeinander aufbauen. Container können direkt auf realen Maschinen (*Bare-Metal*) oder in VMs auf realen Maschinen (*Hypervisor*) gestartet werden. Sowohl VMs als auch Container sind paketierte Computing-Umgebungen, die verschiedene IT-Komponenten wie CPU, RAM und Storage vereinen und vom Rest des Systems isolieren. Die hauptsächlichsten Unterschiede zwischen Containern und virtuellen Maschinen beziehen sich auf ihre Skalierbarkeit und Portierbarkeit. Da Container als Prozessbäume betrachtet werden, können diese einfacher als eine VM skaliert oder portiert werden. Das wird dadurch ermöglicht, dass Container im Gegensatz zu Virtuellen-Maschinen nicht durch einen *Hypervisor* ein eigenes *Guest-OS* (siehe Abbildung auf dem Host-OS) erhalten, sondern sich den *Kernel* mit dem Host-System teilen. Dies wird in der Abbildung 2.4 durch das rote Rechteck hervorgehoben.

2.3 Architektonische Entwürfe von Software-Entwicklung

Bei Softwareanwendungen ((Billy et al., 2021), S. 7) handelt es sich um komplexe Gebilde. Sie bestehen aus einer Vielzahl von Bestandteilen. Bei webbasierten Anwendungen können neben den softwarebasierten Elementen auch eine Menge Systeme beteiligt sein. Wenn Unternehmen webbasierte Anwendungen entwickeln, sind sie für weit mehr als nur die Softwareentwicklung verantwortlich, da sie sich auch um die Bereitstellungsplattformen kümmern müssen. Dies ist ein bedeutender Unterschied zur Entwicklung nativer Anwendungen (d.h. Anwendungen, die direkt auf dem Desktop laufen), bei denen der Endnutzer für die Plattformen verantwortlich ist, auf denen die Software läuft.

2.3.1 Traditionelles Anwendungsdesign

Bei webbasierten Anwendungen ((Dang & Kohgadari, 2021), S. 7) wird traditionell ein sogenanntes N-Tier-Design verwendet, das ein gängiger Ansatz für die Softwareentwicklung im Allgemeinen ist. Ein N-Tier-Anwendungsdesign ((Dang & Kohgadari, 2021), S. 7-8) besteht aus mehreren Komponenten, die auf genau definierte Weise miteinander kommunizieren. Ein gängiger Ansatz für das Design von N-Tier-Anwendungen ist das Model-View-Controller (MVC)-Designmuster. Das *Modell* ist der Ort, an dem die Daten gespeichert und innerhalb der Anwendung dargestellt werden. Die *View* ist die Art und Weise, wie dieselben Daten dem Benutzer präsentiert werden. Dabei kann es sich um eine ganz andere Darstellung handeln, als die des *Models*, denn das *Model* muss aus programmtechnischer Sicht sinnvoll sein, während die tatsächliche *View* visuell sein muss. Das heißt, ein Benutzer muss in der Lage sein, sie zu betrachten und zu wissen, was in der Anwendung geschieht. Der *Controller* ist schließlich die Zwischenschicht, die die Daten aus dem *Model* entnimmt und sie umwandelt (entweder aus dem *Model* oder in das *Model*). Auf dieser Ebene kann sich die Geschäftslogik befinden und sie verhindert, dass die *View* direkt mit dem *Model* kommuniziert. Dies ermöglicht die Erstellung von schnittstellenunabhängigen Programmen. Somit kann dasselbe *Model* für verschiedene Arten von Anwendungen verwendet werden. Die *View* kann sich ändern, aber das *Model* und der *Controller* bleiben bestehen, solange die Kommunikation zwischen den Schnittstellen gleich bleibt. Eine herkömmliche Webanwendung verwendet etwas Ähnliches wie das beschriebene Entwurfsmuster. Sie wird jedoch nicht als *MVC*, sondern als dreistufiges oder n-stufiges Design bezeichnet. Der Grund dafür ist, dass es zwar mehrere Ebenen geben kann, diese aber nicht unbedingt mit der Beschreibung von *MVC* übereinstimmen.

Die Abbildung 2.4 visualisiert eine mögliche Three-Tier-Architecture.

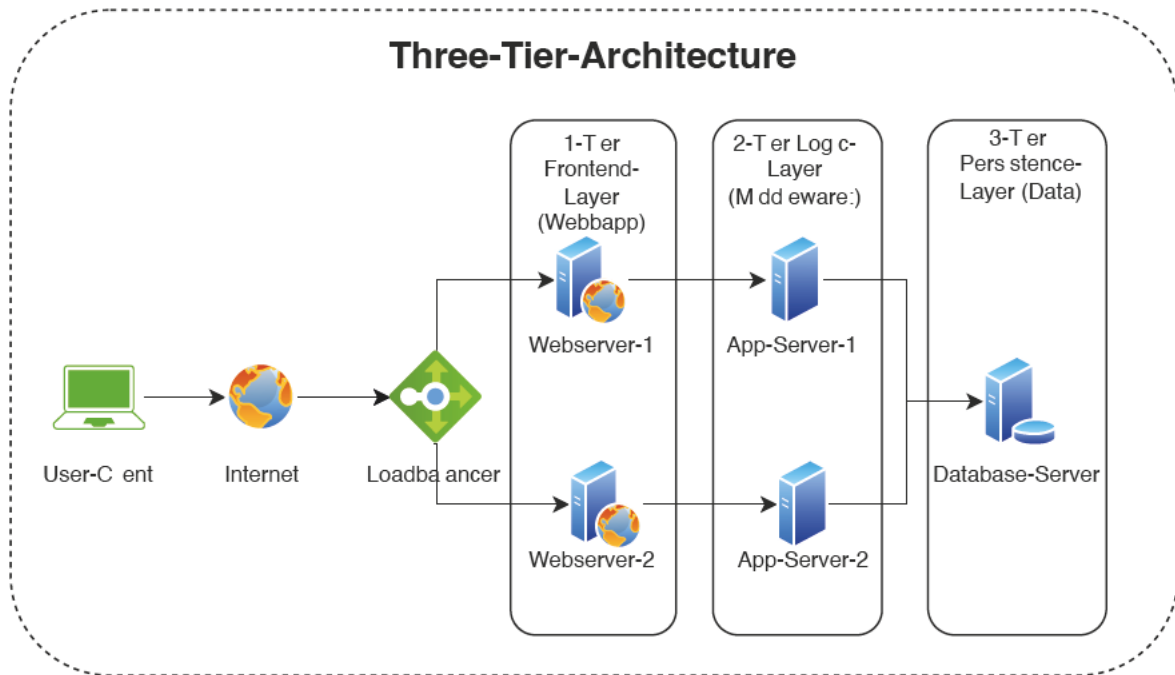


Abbildung 2.4: Visualisierung einer Three-Tier-Architecture, Bildquelle: eigene Darstellung in Anlehnung an ((Dang & Kohgadai, 2021), S. 9)

2.3.2 Serviceorientierte-Architektur

Bei dem soeben beschriebenen traditionellen Design, dem N-Tier-Design (siehe Abbildung 2.2) bestehen die Anwendungen aus mehreren Diensten ((Dang & Kohgadai, 2021), S. 9-10), die zusammenarbeiten. Es gibt einen *Webserver*, einen Anwendungsserver und einen Datenbankserver. Alle Server, egal ob physisch oder virtuell, sind nur Dienste, die dem Rest des Netzes eine Schnittstelle anbieten. Wenn jeder der Server in die Funktionen zerlegt wird, die sie darunter bereitstellen, entsteht eine Vielzahl von Diensten innerhalb der Anwendung. Eine herkömmliche Anwendung hat einen zentralen Verarbeitungsknoten, wie den Anwendungsserver. In einer serviceorientierten Architektur ((Dang & Kohgadai, 2021), S. 11) existiert ein zentrales Kommunikationsmittel, der sogenannten Dienstbus (**engl.** *Service-Bus*). Die Nachrichten werden in den *Bus* eingegeben und der *Bus* sorgt dafür, dass jede Nachricht an den richtigen Dienst weitergeleitet wird, um dort verarbeitet zu werden. Wenn eine Anwendung oder ein Benutzer sich zum Beispiel einloggt, dann wird eine Authentifizierungsfunktion aus dem Profildienst heraus aufgerufen. Die Nachricht (siehe Abbildung 2.5) wird im Wesentlichen über den Methodenaufruf einschließlich des Namens und aller Parameter, die die Methode möglicherweise benötigt, auf den *Bus* gelegt und der *Bus* sorgt für den Aufruf der Funktion. Die Antwort auf den Methodenaufruf wird ebenfalls auf den *Bus* gelegt und an den richtigen Aufrufer zurückgegeben. Der Entwurf basiert auf den Aufrufen und Antworten von Funktionen und Methoden, genau wie die Programmierung von Anwendungen. Der Unterschied bei diesem Design ist, dass der *Bus* Dienste verwaltet, die über mehrere Systeme verteilt sind. Es besteht somit keine Notwendigkeit, ein monolithisches Anwendungsdesign mit allen Komponenten auf einem System zu haben. In der Implementierung ((Dang & Kohgadai, 2021), S. 12) wird die serviceorientierte Architektur zu einer Application-Programming-Interface (API). Die gesamte Anwendung kann von außerhalb verwaltet werden, indem die verschiedenen verfügbaren Methoden aufgerufen werden. Dies öffnet die Tür zu sogenannten *Microservices*. Ohne die Notwendigkeit einer monolithischen Softwareanwendung kann die Anwendung in einzelne Teile zerlegt werden, mit denen andere Komponenten oder Dienste auf

unterschiedliche Weise interagieren. Dies kann dazu führen, dass eine Anwendung von mehreren Anwendungen genutzt wird, da die Funktionen auf unterschiedliche Weise zusammengefügt werden. Der Ablauf der Anwendung wird nicht mehr von einer einzigen Instanz kontrolliert.

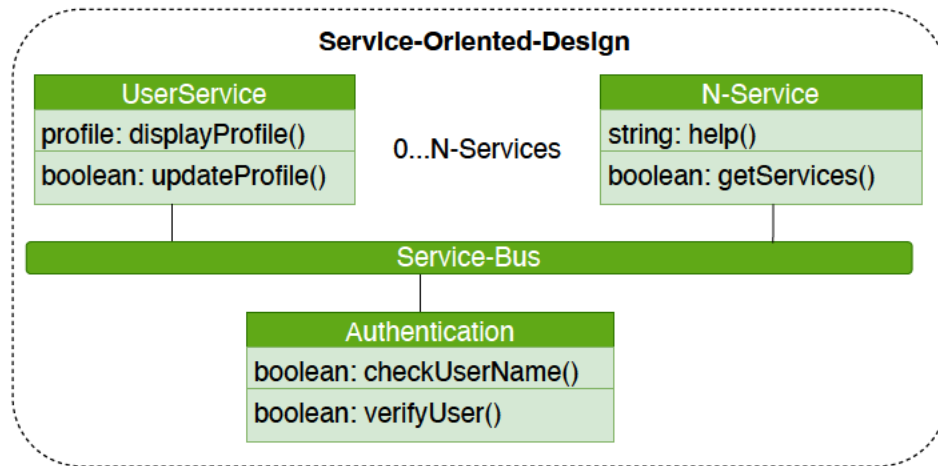


Abbildung 2.5: Service-Oriented-Architecture, Bildquelle: eigene Darstellung in Anlehnung an ((Dang & Kohgadai, 2021), S. 11)

2.3.3 Microservices

Betrachtet man die Geschichte von *SOA*, dann sind *Microservices* (Red Hat Inc., 2018) kein neues Konzept. Die Realisierbarkeit ist dank der Fortschritte bei der Entwicklung der Containerisierungstechnologie zwischenzeitlich enorm angestiegen und ermöglicht dadurch eine bessere Umsetzung. Durch Linux-Container werden mehrere Teile einer *App* auf derselben Hardware unabhängig voneinander ausgeführt. Es handelt sich um ein Architekturkonzept, bei dem als Architektur-Frameworks einzelne Dienste verteilt und lose gekoppelt sind. Dies hat den Vorteil, dass eine manuelle Änderung an der Anwendung durch das Team im Gegensatz zu einer monolithischen Anwendung im schlimmsten Fall nur einen Service zum Absturz bringen und nicht die gesamte Anwendung. Dazu kommt, dass die Teams schnell neue Anwendungskomponenten flexibel an die sich ändernden geschäftlichen Anforderungen bauen und adaptieren. Der Unterschied zwischen traditionellen, monolithischen Ansätzen und einer Microservice-Architektur besteht in der Art und Weise, wie *Apps* in ihre Kernfunktionen entwickelt werden. Jede Funktion bzw. jeder Dienst kann unabhängig entwickelt und bereitgestellt werden. Ein *Microservice* stellt zwar eine Kernfunktion einer Anwendung dar, wird aber unabhängig von anderen Diensten ausgeführt. Die lose Kopplung von Kernfunktionen stellt das Fundament der Microservice-Architektur dar. Dabei findet eine Umstrukturierung der Entwicklungsteams im Sinne des DevOps-Ansatzes statt. Auf die Art und Weise werden die Entwicklungsteams darauf vorbereitet, zukünftiger Features als Services skalierbar und integrierbar zu implementieren. Die Haupteigenschaften von *Microservices* sind, dass diese üblicherweise zustandslos sind und trotzdem miteinander kommunizieren. Dadurch werden die *Apps*, basierend auf einer Microservice-Architektur, fehlertoleranter. Darüber hinaus sind *Microservices* durch die Verwendung von *APIs* unabhängig von der Programmiersprache und die Entwicklerteams haben die Möglichkeit ihre eigenen *Tools* zu wählen. Somit bilden containerisierte *Microservices* zusammen mit *APIs* und dem DevOps-Ansatz die Basis für Cloud-native Anwendungen.

2.3.4 Cloud-Native-Design

Das Cloud-Native-Design ((Dang & Kohgadai, 2021), S. 12) baut auf der Abkehr von den traditionellen monolithischen Anwendungen mit jeweils eigenem Server auf, wie sie die serviceorientierte Architektur verfolgt. Das Cloud-Native-Design nutzt die Vorteile dieser Entkopplung von traditionellen Diensten und Servern. Es verwendet virtualisierte Bereitstellungen, aber anstatt ein ganzes System zu virtualisieren, virtualisiert es eine Anwendung. Es kombiniert somit unter anderem den Ansatz einer Microservice-Architektur ((Dang & Kohgadai, 2021), S. 13) und den Fortschritt bei der Containerisierung von Anwendungen. Der Einsatz von Containern unterstützt diese Vorgehensweise, da damit eine ideale Anwendungsbereitstellung und eine eigenständige Ausführungsumgebung ermöglicht werden. Mit DevOps und Containern können Entwickler *Apps* als Ansammlung loser gekoppelter Komponenten wie *Microservices* veröffentlichen und müssen nicht auf das eine große Release warten. Die Cloud-native Entwicklung richtet den Fokus auf eine modulare Architektur, eine lose Kopplung und Unabhängigkeit der Services. Jeder *Microservice* implementiert eine Geschäftsfunktion, führt seine eigenen Prozesse aus und kommuniziert via *API* oder *Messaging*. Diese Kommunikation kann über eine Service-Mesh-Schicht verwaltet werden. Ein weiterer großer Vorteil virtualisierter Anwendungen ((Dang & Kohgadai, 2021), S. 13-14) ist, dass sie sehr einfach und schnell gestartet werden können. Das Starten einer containerisierten Anwendung kann so schnell sein, dass sie erst als Reaktion auf eine Benutzeranfrage gestartet wird, um Ressourcen im Leerlauf zu schonen. Die Anwendung (Jangda et al. (2019)) existiert nur so lange, wie die Anfrage bearbeitet wird und verschwindet danach. Die Anwendung ist, solange sie nicht benötigt wird, in einem *idle* Zustand und verbraucht keine Ressourcen. Bei Art von Funktionen spricht man von Function-as-a-Service (FaaS), die serverlos (**engl.** *serverless*) laufen. Es handelt sich dabei um ein Ausführungsmodell, bei dem der *Cloud-Service-Provider* wie Microsoft-Azure für die Ausführung von *Code* verantwortlich ist. Die Ressourcen werden dynamisch zugewiesen und der Kunde bezahlt nur Kosten für die Zeit, die der *Code* auch tatsächlich zum Ausführen braucht. Da sich der Kunde keine Gedanken um die Infrastruktur-Komponenten machen muss, wird dieses Modell als serverlos bezeichnet. An diesem Punkt gibt es theoretisch nichts mehr, was unter der Funktion liegt und in irgendeiner Weise gefährdet wird. Der Kunde erhält einen kleinen Teil des Speichers, der zur Funktion gehört, genau wie eine Funktion einen Stack-Frame in einer nativen Anwendung erhalten würde. Wenn die Anfrage ((Dang & Kohgadai, 2021), S. 13-14) von einem Angreifer stammt, der es geschafft hat, die virtualisierte Anwendung zu starten, wird der Zugriff auf den Container nach einer bestimmten Zeit verschwinden.

2.3.5 Cloud-Native-Design mit Kubernetes

Die Virtualisierung bildet eine wichtige Grundlage ((Dang & Kohgadai, 2021), S. 16) für das Cloud-Native-Design. Beim Cloud-Native-Design geht es um weit mehr als nur um Virtualisierung oder die Verwendung eines Microservice-Modells für das Komponenten-Design. Beim Cloud-Native-Design geht es vor allem darum, auf die Anforderungen der Benutzer schnellmöglichst zu reagieren. Deswegen gibt es neben Container-Lösung auch die Möglichkeiten eine *FaaS* zu nutzen und eine Funktion serverlos zu betreiben. Bei dem Cloud-Native-Design mit Kubernetes handelt es sich um eine besondere Form, die den Schwerpunkt auf Containerisierung legt. Genau für diese Art von Problemen wurde Kubernetes¹ entwickelt. Die Reaktionsfähigkeit der Anwendung wird durch das Cloud-Native-Design stark erhöht. Die Anwendung oder ihre Instanzen können sehr schnell gestartet werden, da nicht erst ein ganzes Betriebssystem hochgefahren werden muss. Bei der Implementierung einer Cloud-nativen Anwendung entstehen unterschiedliche Möglichkeiten und Herausforderungen. Eine der Herausforderungen bei jeder Anwendung unabhängig vom Architektur-Design stellen die Upgrades dar. So sorgt Kubernetes dafür, dass die neueste Software für jede Instanz der Anwendung oder ihrer Komponenten vorhanden

ist und löst die Upgrade-Herausforderung. Somit löst Kubernetes die Herausforderung durch den automatischen Austausch der alten Version durch eine Version bei erfolgreicher Rotierung der Container und macht ebenfalls automatisch ein Rollback, falls die neue Version Probleme enthält. Bei einem Microservices-Modell ((Dang & Kohgadai, 2021), S. 16-17) mit mehreren Komponenten, die jeweils ihren eigenen virtualisierten Bereich benötigen, gibt es viele bewegliche Teile und dadurch auch viele Abhängigkeiten. Kubernetes kann sich um den Austausch von Instanzen der einzelnen *Microservices* kümmern, ohne dass die Benutzer, die die Dienste verwenden, etwas von dem Austausch mitbekommen. Jede Instanz, die in Gebrauch ist, kann ausgetauscht werden, sobald sie nicht mehr gebraucht wird oder in einem *unhealthy* Zustand ist. Das *Life-Cycle-Management* von zusammengesteckten *Microservices* wird durch eine *PaaS* wie Kubernetes für den Betrieb stabilisiert und vereinfacht. Da Kubernetes in der Lage ist, Instanzen einer Anwendungskomponente oder eines *Microservices* schnell zu aktivieren, ist es eine optimale Plattform für eine schnelle Skalierung. Außerdem kann Kubernetes dafür sorgen, dass weitere Instanzen von Anwendungskomponenten je nach Bedarf und Anzahl der Anfragen der Benutzer bereitgestellt werden. Dies passiert durch die horizontale Skalierung weiterer Replikationen des Servers nach oben, wenn der Bedarf steigt und nach unten, wenn der Bedarf sinkt. Somit werden die darunter liegenden Ressourcen wie CPU, RAM oder Storage bestmöglich genutzt. Im Falle von *Cloud Computing* mit einem *Cloud-Service-Provider*, bei dem die Dienste nach Bedarf des Kunden aktiviert und abgerechnet werden, spart der Kunde Geld. Außerdem ist die Reaktionsfähigkeit beim Beziehen eines Cloud-Services auf ein schwer vorhersagbares Ereignis aufgrund der größeren Ressourcen schneller. Die zusätzlichen Ressourcen wie virtuelle Maschinen, die als Knoten dienen, auf denen die Arbeitslast der *Microservices* läuft, können beim *CSP* durch horizontale Skalierung der Knoten bereitgestellt werden. Dies setzt voraus, dass die Anwendungen mit den richtigen Komponenten und der richtigen Konfiguration erstellt worden sind. Dazu eignet sich zum Beispiel Kubernetes für das Orchestrationsmanagement der Anwendungen, ohne, dass zusätzlicher Bedarf in der Anpassung der Infrastruktur vorgenommen werden muss.

2.4 Orchestrierung

Im letzten Jahrzehnt sind neue Software-Architekturmuster (al Jawarneh et al., 2019) wie *SOA* oder *Microservices* entwickelt worden, um die Anwendungsmodularität zu verbessern. Die Entwicklung sollte ein besseres Testen, Skalieren und Ersetzen von Komponenten ermöglichen. Zur Unterstützung dieser neuen Trends werden neue Praktiken wie DevOps-Methoden- und -Tools eingesetzt. Dadurch soll eine bessere Zusammenarbeit zwischen Softwareentwicklungs- und -Betriebsteams erzielt werden. Dabei spielten Container-basierte Technologien eine entscheidende Rolle, indem sie die schnelle Bereitstellung von *Microservices* und deren Skalierbarkeit bei geringem *Overhead* ermöglichten. Moderne containerbasierte Anwendungen können leicht aus Hunderten von *Microservices* bestehen. Deswegen erfordern Services mit komplexen Abhängigkeiten fortschrittliche Orchestrierungsfunktionen. Die Orchestrierung (Red Hat, 2019) von Containern ist zuständig für das Automatisieren von Deployments, dem Management, der Skalierung und der Vernetzung von Containern. Wenn Unternehmen eine Umgebung mit mehreren Hunderten oder Tausenden von Linux-Containern betreiben, dann profitieren diese von einer Container-Orchestrierung-Lösung. Die Container-Orchestrierung ((al Jawarneh et al., 2019)) ermöglicht es, automatisierte Bereitstellungs- und Änderungsmanagement-Workflows zu definieren, um eine bessere Verfügbarkeit zu gewährleisten. Die Umsetzung erfolgt durch eine

¹<https://kubernetes.io>, letzter Zugriff 24.05.2022

Orchestrierungs-Engine, die auf mehreren Schichten basiert und der Zusammenarbeit von Maschinen durch ihren *Kernel* und ihre Container-Laufzeit-Umgebung wie Docker.

Die Struktur der Orchestrierungs-Engine ((al Jawarneh et al., 2019)) besteht aus drei Schichten (siehe Abbildung 2.6): Ressourcenverwaltung (**engl.** *Resource Management*), Ablaufplanung (**engl.** *Scheduling*) und Dienstverwaltung (**engl.** *Service-Management*). Die **Ressourcenverwaltungsschicht** verwaltet *Low-Level* Ressourcen, die funktionalen Elemente wie zum Beispiel RAM, CPU/GPU, Festplattenspeicher und Volumes. Volumes werden benötigt, um die Möglichkeit der Interaktion mit dem Dateisystem des lokalen Host-Rechners zu ermöglichen. Dies ermöglicht auch, die Daten oder Änderung durch persistente Volumes, auch mit einem entfernten Cloud-Dateisystem zu interagieren, wie zum Beispiel Azure-SMB-Fileshare¹. Darunter fallen auch Netzwerkressourcen wie Port und IP für die Konfiguration von UDP- oder TCP-Ports und IPs innerhalb des virtuellen Containernetzes. Das Ziel dabei ist Container nur dann anzusprechen, wenn sie bereit (d.h. der Prozess vollständig gestartet ist) sind. Darüber hinaus wird über das Netzwerk auch der Gesundheitszustand (**engl.** *Healthchecks*) von Containern überwacht sollte ein Container sich in einer vordefinierten Zeit nicht antworten, dann wird dieser durch einen Neustart wiederbelebt. Dadurch wird gewährleistet, dass langlebige Prozesse, deren Aufgabe es ist, immer betriebsbereit sein müssen, auch betriebsbereit bleiben. Die Umplanung zum automatischen Neustart und die Planung abgestürzter Container, die auf einem ausgefallenen Knoten laufen, übernimmt die **Ablaufplanungsschicht**. Darunter fallen auch *Rolling Deployment* zum automatischen *Up-* und *Downgrading* der Anwendungsversion. Das *Scheduling* ist auch für Durchsetzung von Einsatzbeschränkungen verantwortlich, um zum Beispiel Container so zu platzieren, dass die lokale Kommunikation zwischen Prozessen genutzt werden kann. Schließlich bietet die **Dienstverwaltungsschicht** funktionale Funktionen für die Erstellung und den Einsatz von Unternehmensanwendungen. Sie verwaltet High-Level-Aspekte wie die Kennzeichnungen, um Cocontainerobjekten Metadaten zuzuordnen. Unter Metadaten fallen Gruppen oder Namensräume, die zur Isolierung von Containern und zur Unterstützung der Mehrmandantenfähigkeit eingesetzt werden. Die Metadaten stellen Bereitschaftsprüfung, um die Anwendung nur dann als online zu kennzeichnen, wenn sie bereit ist, eingehenden Datenverkehr zu akzeptieren. Die Schichten bauen aufeinander auf. Das *Service-Management* gibt an wann ein Container als *unhealthy* gilt und der *Scheduler* kümmert sich dann, um einen Neustart oder Umsiedlung des Containers auf einen *Node* mit mehreren verfügbaren Ressourcen. Dazu fragt der *Scheduler* über das *Resource-Management* die benötigten Ressourcen an. Die Schichten stellen zwar das Fundament für die Orchestrierung von Container dar, aber setzen selbst auf weiteren Schichten einer Container-Infrastruktur auf (siehe Abbildung 2.6). Eine *Container-Engine* (Red Hat, 2019) wie Docker stellt noch lange kein eigenständiges Container-Container-Orchestrierungs-Tool dar. Denn Container-Orchestrierungs-Tools bieten ein großes *Framework* für die Verwaltung von Microservices-Architekturen und Containern. Es gibt viele Container-Orchestrierungs-Tools oder Distributionen, die sich für das Lifecycle-Management von Containern eignen. Das beliebteste und bekannteste Container-Orchestrierungs-Tool ist Kubernetes. Bei Kubernetes handelt es sich, um ein Open-Source-Tool für die Orchestrierung von Containern. Es wurde ursprünglich von Google entwickelt. Einen tieferen Einblick zu Kubernetes gibt das Kapitel 3.

¹<https://docs.microsoft.com/de-de/azure/storage/files/storage-files-introduction>, letzter Zugriff 25.05.2022

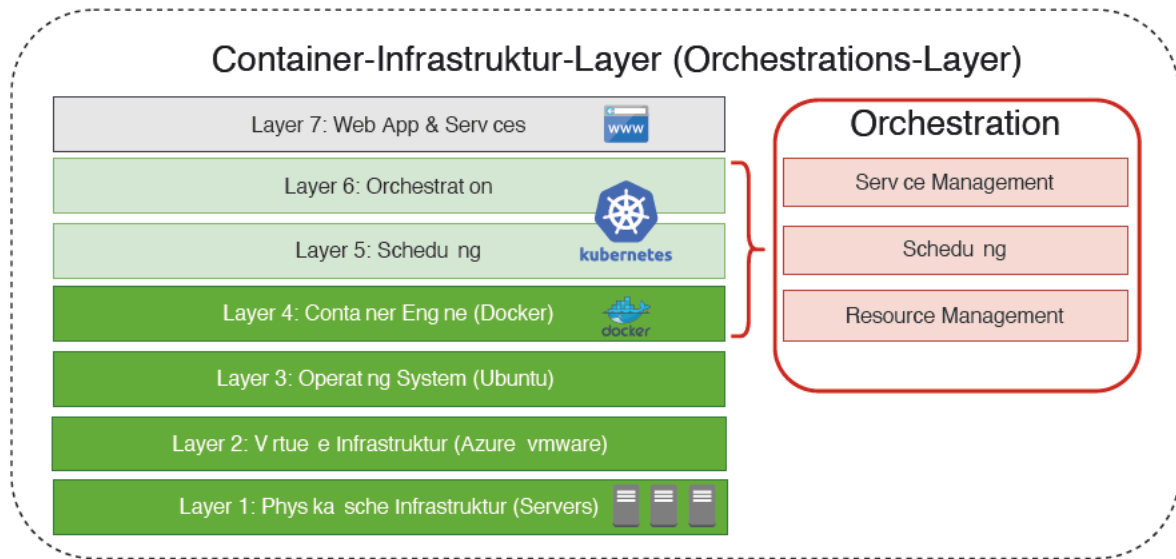


Abbildung 2.6: Container-Orchestrierungs-Schichten, Bildquelle: eigene Darstellung in Anlehnung an ((Liebel, 2019), S. 79), (al Jawarneh et al., 2019)

Welche Vorteile bietet eine Container-Orchestrierung?

Wenn die Abläufe und das Lifecycle-Management von Containern automatisiert verwaltet werden sollen, dann lässt es sich am einfachsten durch den Einsatz eines Container-Orchestrations-Tools lösen. Diese bieten unter anderem folgende Vorteile (Red Hat, 2019) bei der Automatisierung und Verwaltung von Aufgaben:

- *Deployment* und Provisionierung von Containern
- Ressourcenzuweisung wie CPU, RAM und Storage
- Konfiguration und Planung der Container, sowie von Anwendungen basierend auf dem Container, in dem sie ausgeführt werden
- Container-Verfügbarkeit und *Life-Cycle-Management* inklusive *Healthchecks* (Überwachung des Containerzustandes)
- Lastverteilung von Anfragen und das Routing des Traffics zu den entsprechenden Containern, sowie Sicherstellung der Interaktionen zwischen Containern
- Skalierung, *Redeployment* oder Entfernen von Containern, um die Arbeitslast gleichmäßig über die Infrastruktur zu verteilen

Die DevOps-Teams werden beim Management der *Container-Lifecycles* durch die Orchestrierung unterstützt und können die Tools in ihre Workflows integrieren. Somit bilden containerisierte *Microservices* zusammen mit *APIs* und DevOps-Teams die Basis für Cloud-native Anwendungen.

2.5 Agile Methoden

Der Begriff „Agilität (engl. *Agile*)“ bedeutet übersetzt schnell und leicht bewegen (Rök & Guttenberger, 2018) können und lässt sich zurückführen auf Kanter und Peters. Im Projektmanagement ist *Agile* eine Methodik, die in der IT-Branche vor allem in der Softwareentwicklung sehr beliebt ist. Eine „Methodik“ ist die Art und Weise, wie etwas erledigt wird. Es sind die Methoden, Techniken und Ansätze, um etwas zu erreichen. Somit gibt es verschiedene Projektmanagement-Methoden und *Agile* ist eine der am häufigsten verwendeten Methoden. Das traditionelle Projektmanagement zeichnet sich durch einen linearen, sequentiellen Ansatz aus, der in der Regel einige oder alle dieser allgemeinen Schritte umfasst. Im Gegensatz dazu ist *Agile* ((Moreira, 2017), S.22-23) nicht mehr und nicht weniger als eine Reihe von Werten und Prinzipien. In Bezug auf den Erfolg ist *Agile* der Wegbereiter, der die Kraft der Mitarbeiter und das Feedback der Kunden für eine erfolgreiche und häufige Lieferung nutzbar macht. Es ist wichtig, das Manifest für die agile Softwareentwicklung zu lesen und zu verinnerlichen, um eine agile Geisteshaltung im Team aufzubauen. Das *Manifesto for Agile Software Development* besteht aus nur 73 Wörtern und wurde 2001 von 17 Autoren unterzeichnet. Das *Agile* Manifest besagt, dass bessere Wege gefunden werden, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen, es zu tun. Durch diese Zusammenarbeit sind folgende Werte entstanden:

- **Individuen und Interaktionen** stehen über Prozesse und Werkzeuge
- **Funktionierende Software** steht über umfassende Dokumentation. Funktionierende Software ist der wichtigste Maßstab für den Fortschritt eines Produkts.
- **Zusammenarbeit mit dem Kunden**, statt Vertragsverhandlungen
- **Auf Veränderungen reagieren**, statt einen Plan zu befolgen

Das heißt nicht, dass die Werte ((Moreira, 2017), S. 23) der Objekte auf der rechten Seite nicht wichtig sind. Die Objekte auf der rechten Seite haben auch einen Wert für die Softwareentwicklung, aber die Werte der Objekte auf der linken Seiten werden mehr geschätzt. Es ist wichtig, das richtige Gleichgewicht zu finden. Wenn die Entwicklung in Richtung Agile gehen soll, dann wird schnell festgestellt, dass die Tendenz zu den Punkten auf der linken Seite stärker ist.

Die Grundsätze des agilen Manifests

Die höchste Priorität ((Moreira, 2017), S. 23-24) ist es, den Kunden durch frühzeitige und kontinuierliche Lieferung von hochwertiger Software zufrieden zustellen. Die Agilen-Prozesse nutzen den Wandel in der IT zum Wettbewerbsvorteil des Kunden. Es muss häufig funktionierende Software durch inkrementelle Auslieferung bereitgestellt werden. Dabei spricht man von einigen Wochen bis zu einigen Monaten, wobei der kürzere Zeitrahmen bevorzugt wird. Alle *Stakeholder* wie Kunden, Fachbereiche und Entwickler müssen während des gesamten Projekts eng zusammenarbeiten. Die Projekte sollten um motivierte *Stakeholder* aufgebaut werden, denn die Motivation treibt Innovation, sowie die Weiterentwicklung voran. Deswegen muss ein Umfeld geschaffen werden, welches die nötige Unterstützung für die Teams liefert. Die besten Architekturen, Anforderungen und Entwürfe entstehen in selbstorganisierenden und motivierten Teams. Die Informationen müssen möglichst effizient und effektiv innerhalb eines Entwicklungsteams übermittelt werden. In regelmäßigen Abständen überlegt das Team, wie es effektiver werden kann und passt sein Verhalten entsprechend an. Das waren die wesentlichsten Werte und Prinzipien von agilen Methoden.

Scrum

In der heutigen Softwareentwicklungsumgebung (Carvalho, Henrique & Mello, 2011) sind die Anforderungen während des Produktentwicklungszyklus ständigen Änderungen unterworfen, damit auf eine schnelle Reaktion auf Veränderungen möglich ist. Mitte der 90er Jahre wurden agile Entwicklungstechniken für Softwareprodukte verfügbar. Diese Disziplin wurde stark von den bewährten Praktiken der japanischen Industrie beeinflusst, vor allem von den Lean-Manufacturing-Prinzipien, die von Honda und Toyota eingeführt wurden. In diesem Zusammenhang wird Scrum, ein Ansatz zur schlanken Produktentwicklung, hervorgehoben. Dieser Prozess wurde 1993 von Jeff Sutherland¹ entwickelt. Er basiert auf einem Artikel von 1986 (Takeuchi & Nonaka, 1986) in dem die Vorteile kleiner Teams bei der Produktentwicklung erörtert werden. Agile Softwareentwicklungsmethoden haben in letzter Zeit an Popularität gewonnen. Die agilen Entwicklungsmethoden entstanden aus der Überzeugung, dass ein Ansatz, der sich stärker an der menschlichen Realität (Carvalho et al., 2011) orientiert und an der Realität der Produktentwicklung, die vom Lernen, von Innovation und von Veränderung geprägt ist, bessere Ergebnisse liefern würde. Die bei weitem beliebteste agile Methode ist Scrum. Bei Scrum handelt es sich um einen iterativen und inkrementellen Rahmen für Produkt- oder Anwendungsentwicklung. Es strukturiert die Entwicklung in Arbeitszyklen, die *Sprints* (siehe Abbildung 2.7) genannt werden. Diese Iterationen (Sutherland, 2021) dauern zwei bis vier Wochen und finden ohne Unterbrechung nacheinander statt. Die *Sprints* sind zeitlich begrenzt und enden zu einem bestimmten Datum, unabhängig davon, ob die Arbeit abgeschlossen ist oder nicht. Zu Beginn jedes *Sprints* wählt ein funktionsübergreifendes Team Elemente aus einer nach Prioritäten geordneten Liste (*Sprint Backlog*) aus. Das Team verpflichtet sich, die Aufgaben bis zum Ende des *Sprints* zu erledigen. Während des *Sprints* ändern sich die ausgewählten Punkte nicht. Jeden Tag (*Daily Scrum*) trifft sich das Team kurz, um seinen Fortschritt zu überprüfen und die nächsten Schritte festzulegen, die zur Fertigstellung der verbleibenden Arbeit erforderlich sind. Am Ende des *Sprints* bespricht das Team den Sprint mit den Beteiligten (*Sprint-Review*) und zeigt, was es gebaut hat. Die Beteiligten erhalten Rückmeldungen, die in den nächsten Sprint (*Sprint-Planning*) einfließen können. Scrum legt Wert darauf, dass am Ende des *Sprints* ein funktionierendes Produkt vorliegt, das wirklich „fertig“ ist. Im Falle von Software bedeutet dies, dass der *Code* integriert, vollständig getestet und potenziell auslieferbar ist. Die wichtigsten Rollen, Artefakte und Ereignisse sind nochmals Abbildung 2.7 zusammengefasst.

Im Folgenden erfolgt eine Beschreibung der Rollen (Sutherland, 2021) im Scrum.

Scrum-Roles:

- **Product Owner:** ist für die Maximierung der Kapitalrendite (**engl.** Return-On-Investment (ROI)) verantwortlich, indem er Produktfunktionen (Product Backlog) identifiziert, diese in eine Prioritätenliste überträgt und entscheidet, welche davon im nächsten Sprint ganz oben auf der Liste (Sprint Backlog) stehen sollen. Die Liste wird ständig neu priorisiert und verfeinert. Der *Product Owner* trägt die Verantwortung für Gewinn und Verlust des Produkts. In einigen Fällen sind der *Product Owner* und der Kunde ein und dieselbe Person. Dies ist bei internen Anwendungen üblich. In anderen Fällen kann es sich bei dem Kunden um Millionen von Menschen mit einer Vielzahl von Bedürfnissen handeln. Der *Product Owner* unterscheidet sich von einem traditionellen Product Manager, da er aktiv und häufig mit dem Team interagiert.
- **Team:** besteht aus sieben Personen plus oder minus zwei und umfasst die Fähigkeiten in den Bereichen Analyse, Entwicklung, Testen, Schnittstellendesign, Datenbankdesign, Architektur oder

¹<http://jeffsutherland.org/scrum/scrumpapers.pdf>, letzter Zugriff 25.05.2022

Dokumentation. Das Team entwickelt das Produkt und liefert dem *Product Owner* Ideen, wie man das Produkt verbessert.

- **Scrum Master:** hilft der Produktgruppe, Scrum zu erlernen und anzuwenden, um einen geschäftlichen Nutzen zu erzielen. Er tut alles, was in seiner Macht steht, um dem Team und dem *Product Owner* zum Erfolg zu verhelfen. Der *Scrum Master* ist nicht der Manager des Teams oder ein Projektmanager, stattdessen dient der *Scrum Master* dem Team, schützt es vor Einmischung von außen. Im Gegensatz zu einem Projektmanager sagt der *Scrum Master* dem Team nicht, was sie zu tun haben. Der *Scrum Master* und der *Product Owner* können nicht dieselbe Person sein.
- **The Scrum Team:** *Product Owner* + *Team* + *Scrum Master*

Es gibt in Scrum keine Rolle (Sutherland, 2021) des Projektmanagers. Das liegt daran, dass keine benötigt wird. Die traditionellen Aufgaben eines Projektmanagers wurden aufgeteilt und den drei Scrum-Rollen neu zugewiesen.

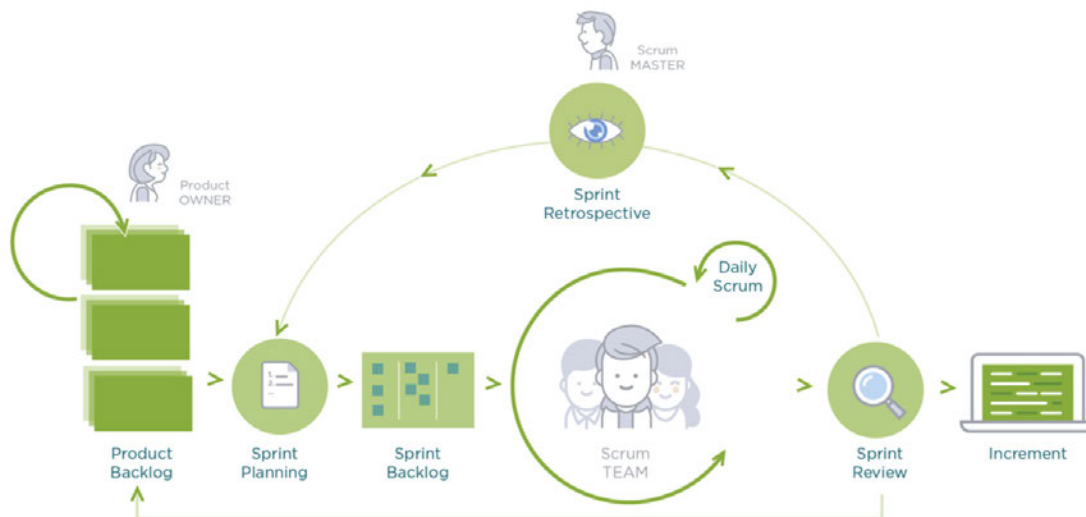


Abbildung 2.7: SCRUM, Bildquelle: <https://www.micromata.de/blog/projektmanagement/erfolgreich-agil-arbeiten-teil-3/>, letzter Zugriff 25.05.2022

2.6 DevOps

Viele Unternehmen konzentrierten (Jha & Khan, 2018) sich auf ein zentrales Problem und beschäftigten sich mit der Frage *Wie können die Hindernisse zwischen Verbesserungen und Aktivitäten für die Weiterentwicklung des Unternehmens überwunden werden?* Eine der ersten Diskussionen zu dem DevOps-Ansatz fand zwischen den IT-Experten Patrick Dubois und Andrew Clay Shaffer auf dem Speed Gathering 2008 statt. Als Ergebnis der Diskussion entstand, eine der wichtigsten Gruppen, die sich mit diesem Thema befassten, die *Lithe Frameworks Organization Gathering*. Diese Gruppe und die daraus resultierenden Diskussionen veranlassten Dubois, das wichtigste *Speed Meeting* unter dem Namen *DevOps Days* zu veranstalten. Die Idee verbreitete sich im Jahr 2009 mit der ersten DevOpsDays-Veranstaltung in Belgien. Damit war der Begriff DevOps geboren. DevOps wurde zunächst als ein Verfahren angesehen, das nur in großen Organisationen wie zum Beispiel bei Google zum

Einsatz kam. In der DevOps-Fachwelt werden diese Art von Unternehmen häufig als „*Cloud-Locals*“, „in der Cloud konzipiert“ oder „Einhörner (**engl.** *Unicorns*)“ bezeichnet.

Der Begriff „DevOps“ spielt regelmäßig (Jha & Khan, 2018) auf die zunehmende fachliche Entwicklung an, die eine gemeinschaftsorientierte Arbeitsverbindung zwischen Entwicklungs- und IT-Aktivitäten unterstützt. Dadurch wird ein schneller Fluss der geplanten Arbeit ermöglicht und gleichzeitig wird die Zuverlässigkeit, Stabilität, Flexibilität und Sicherheit der Umgebung erhöht. Der Begriff setzt sich aus Entwicklung (**engl.** *Development*) und Betrieb (**engl.** *Operation*) zusammen und verbindet Entwicklungs- und Betriebsaufgaben miteinander. Durch DevOps soll der Einsatz von korrespondierenden Werkzeugen eine effizientere und reibungslose Zusammenarbeit von Entwicklung und Operation realisieren. Das Ziel ist es, eine schnelle Umsetzung von hochwertiger und stabiler Software in einem automatisierten Prozess zu schaffen. Es ist eine Verschmelzung von Softwareentwicklung und Systemadministration, die durch korrespondierende *Tools* einen Automatisierungsprozess ermöglicht. Es handelt sich dabei nicht um einen Prozess oder um ein Modell, sondern um eine verbesserte Version von der Agilen-Entwicklung, die sich auf operative Aspekte konzentriert.

Das Kernkonzept Culture-Automation-Measurement-Sharing (CAMS) (Perera et al., 2017) von DevOps (Jha & Khan, 2018) wird wie folgt definiert:

- **Kultur** (**engl.** *Culture*): Gegenseitiges Vertrauen innerhalb einer Gruppe von Entwicklern und Administratoren und Verbesserung der bereichsübergreifenden Korrespondenz. Aufbau einer Kultur der koordinierten Anstrengungen, Trennung von konventionellen Lagern und grundlegenden Leistungsmessungen. Aktualisierte Quellkorrespondenz, die das stetige Lernen und die Beseitigung von Hindernissen ermöglicht.
- **Automatisierung** (**engl.** *Automation*): Spart Zeit, verhindert Fehler, schafft Konsistenz und stärkt die Selbstverwaltung von bestimmten Arbeitsprozessen. Die Automatisierung setzt die richtigen Mitarbeiter, Prozesse und Werkzeuge voraus, um Automatisierungs-Frameworks zu erstellen. Entwickler müssen im vollen Umfang über die Bedeutung der Aufrechterhaltung der Anwendungsleistung in großen Umgebungen unter hoher Last unterrichtet werden.
- **Messung** (**engl.** *Measurement*): Einheitliche Bewertungskriterien. Verkürzung der Zeit bis zur Veröffentlichung von Anzeigen mithilfe von Kundeninformationen. Untermauert die Schätzungen und Entscheidungen in Hinblick auf offensichtlich zu durchschauende Informationen. Systemmetriken und Anwendungsverhalten müssen offen, transparent, zugänglich, ressourcenschonend und sinnvoll sein, um von allen Komponenten des Modells visualisiert zu werden.
- **Teilen** (**engl.** *Sharing*): Die Bereitschaft, das Wissen zu teilen, steigert die Qualität der Software. Der Austausch von Wissen beinhaltet Tools, Entdeckungen und Lektionen. Der Austausch von Problemen stellt einen Erfolgsfaktor für jedes Unternehmen dar.

Zusammenfassend lässt sich sagen, dass DevOps eine Reihe von Methoden (Jha & Khan, 2018) ist, bei denen Entwicklung und Betrieb kommunizieren und zusammenarbeiten, um Software und Dienstleistungen schnell, zuverlässig und mit höherer Qualität bereitzustellen. Um den Gedanken hinter DevOps zu realisieren, benötigt es eine IT-Landschaft, welche einer dynamischen Elastizität unterliegt, die sich bezüglich Erweiterbarkeit, Skalierung und Portierbarkeit anpassen lässt. Dies wird durch Container und Virtualisierungen, welche die Orchestrierung von Containern ermöglichen, realisierbar. Bei DevOps handelt es sich somit primär um das Mindset und einen kulturellen Wandel in einem Team. Der Fokus liegt somit auf der Kultur im Unternehmen. Der Einsatz von *Tools*, *Frameworks* und Methoden wird durch den Wandel in der IT regelmäßig ausgetauscht. Das Mindset, wenn es sich im Unternehmen etabliert hat, bleibt länger erhalten.

2.7 Site Reliability Engineering

In der Vergangenheit haben Unternehmen Systemadministratoren ((Betsy Beyer, 2016), S.3) beschäftigt, um komplexe Kommunikationssysteme zu betreiben. Bei diesem Systemadministrator-Ansatz werden vorhandene Softwarekomponenten zusammengestellt und so eingesetzt, dass sie zusammen einen Dienst bilden. Systemadministratoren haben dann die Aufgabe, den Dienst zu betreiben und auf auftretende Ereignisse und Aktualisierungen zu reagieren. Mit zunehmender Komplexität des Systems und steigendem Verkehrsaufkommen, das eine entsprechende Zunahme von Ereignissen und Aktualisierungen mit sich bringt, wächst das Team der Systemadministratoren, um die zusätzliche Arbeit zu bewältigen. Da die Rolle des Systemadministrators ((Betsy Beyer, 2016), S.4-5) deutlich andere Fähigkeiten erfordert als die der Produktentwickler, werden Entwickler und Systemadministratoren in getrennte Teams aufgeteilt. Ein Team besteht aus Entwicklung und ein anderes Team besteht aus Betrieb. Durch die strikte Trennung der Teams fühlt sich jedes Team nur für einen gewissen Teil eines Dienstes verantwortlich. Die Entwickler liefern funktionierende Software aus, ohne sich dabei Gedanken, um die darunter liegende Infrastruktur zu machen. Das Operation-Team rollt dann die Anwendung auf der betriebenen Infrastruktur und macht sich keine Gedanken, um die Implementierung der Anwendung. Kommt es zu einem Problem, weil die Anwendung zu langsam ist oder regelmäßig abstürzt und der angebotene Dienst für die Kunden sehr unzufriedenstellend ist, dann entstehen Konflikte zwischen den Teams. Deswegen hat sich Google entschieden, seine Systeme mit einem anderen Ansatz zu betreiben. Die *Site-Reliability-Engineering-Teams* konzentrieren sich darauf, Software-Ingenieure einzustellen, um Produkte zu betreiben und Systeme zu schaffen, die die Arbeit erledigen, die sonst oft manuell von Systemadministratoren ausgeführt wird. *SRE* ist das, was passiert, wenn man einen Software-Ingenieur bittet, ein Betriebsteam aufzubauen. Das Konzept von *Site Reliability Engineering* stammt vom Google Engineering-Team ((Betsy Beyer, 2016), S.5) und wird Ben Treynor Sloss zugeschrieben. Somit handelt es sich bei *SRE*, um eine einzigartige Rolle, die entweder einen Hintergrund als Softwareentwickler mit zusätzlicher Operations-Erfahrung oder als Systemadministrator mit Softwareentwicklungsfähigkeiten erfordert. Deswegen ist laut Benjamin Treynor Sloss der wichtigste Baustein des Google-Ansatzes für das *Service-Management*, die Zusammensetzung der einzelnen *SRE-Teams*. Google unterteilt die Teams in zwei Hauptkategorien. Bei 50-60 % handelt es sich, um Google-Software-Ingenieure, oder genauer gesagt, um Personen, die über das Standardverfahren für Google-Software-Ingenieure eingestellt wurden. Die anderen 40-50 % sind Kandidaten, die den Qualifikationen eines Google Software Engineers sehr nahe kamen (d. h. 85-99 % der geforderten Fähigkeiten) und darüber hinaus über eine Reihe von technischen Fähigkeiten verfügten, die für *SRE* nützlich sind, aber bei den meisten Softwareingenieuren selten vorkommen. Bei weitem sind UNIX-Systeminterne und Netzwerkkennnisse die beiden häufigsten Arten von alternativen technischen Fähigkeiten, die Google sucht.

Bei *SRE* wird ein Service-Orientierter-Ansatz (Red Hat, 2020) verfolgt, bei dem nicht nur fertige Software ausgeliefert und betrieben wird, sondern auch, dass diese Zuverlässig für den Kunden läuft. Die Verantwortung für *SRE-Teams* liegt somit bei der Bereitstellung, Konfiguration und Überwachung von *Code*. Darüber hinaus auch für die Verfügbarkeit und Latenz des Produktes. Sowie auch für Issue- und Kapazitätsmanagement von Services in der Produktion. Um dies zu gewährleisten, werden Service-Level-Indicators (SLI), Service-Level-Agreements (SLA) und Service Level Objectives (SLO) mithilfe von *SRE* bestimmt. Über *SLAs* wird bestimmt, welche neuen Features und vor allem wann eingeführt werden können, um die erforderliche Zuverlässigkeit des Systems mithilfe von *SLIs* und *SLOs* zu definieren. Bei *SLIs* handelt es sich um definierte Maßnahmen von spezifischen Aspekten von einem bereitgestellten Service. Darunter gehören zu den wichtigsten Aspekten von *SLIs*, die Latenz, die

Verfügbarkeit, die Fehlerrate und der Durchsatz eines Systems. Bei *SLO* handelt es sich um objektiv akzeptable Kriterien, basierend auf dem Zielwert eines spezifischen Indikators von *SLI*. Für die Zuverlässigkeit eines Systems wird zum Beispiel als Akzeptanz eine bestimmte Ausfallzeit bestimmt. Die akzeptierte Ausfallzeit wird als „Fehlerbudget“ bezeichnet. Das ist der maximal zulässige Schwellenwert für Fehler und Ausfälle eines Systems. Es wird mit *SRE* somit keine 100% Zuverlässigkeit eines Systems erwartet. Die damit verbundenen Ausfälle sind eingeplant und akzeptiert, solange diese den vordefinierten Schwellenwert nicht überschreiten.

2.8 DevOps vs SRE

Die Kernprinzipien ((Betsy Beyer, 2016), S. 6-7) von DevOps, wie Einbeziehung der IT-Funktion in jeder Phase des Systemdesigns und der Systementwicklung stimmen mit vielen Prinzipien und Praktiken von *SRE* überein. Man könnte DevOps als eine Verallgemeinerung einiger zentraler *SRE*-Prinzipien auf ein breiteres Spektrum von Organisationen, Managementstrukturen und Personal betrachten. Gleichmaßen könnte man *SRE* als eine spezifische Implementierung von DevOps mit einigen idiosynkratischen Erweiterungen betrachten. Das Konzept von DevOps (Red Hat, 2020) umfasst die Aspekte Unternehmenskultur, Automatisierung und Plattformdesign. Das Ziel ist dabei, den geschäftlichen Mehrwert und die Reaktionsfähigkeit durch die schnelle Bereitstellung hochwertiger Services zu steigern. Somit legt DevOps den primären Fokus auf die **Liefergeschwindigkeit** von Produkten, Software oder Inkrementen. Der Unterschied zwischen *SRE* und DevOps liegt darin, dass sich bei *Site Reliability Engineers* innerhalb des eigenen Teams auf das Wissen von Personen referenziert wird, die über einen Operations-Hintergrund verfügen. Die Rolle des *SRE* erfordert eine Kombination der Fähigkeiten aus Operations- und DevOps-Teams. Dadurch kann *SRE* mit spezielleren OPs-Fähigkeiten den DevOps-Teams helfen, wenn die Entwickler mit Operations-Aufgaben überfordert sind. Der primäre Fokus von *SRE* liegt darauf, ein Gleichgewicht zwischen Funktionssicherheit und der Erstellung neuer Features zu schaffen, um eine **Verlässlichkeit** eines Produktes zu gewährleisten.

Das Unternehmen IBM (LeBris, 2021) beschreibt die Hauptmerkmale, die DevOps und *Site Reliability Engineering* voneinander unterscheiden, folgendermaßen. Bei DevOps geht es um die Kernentwicklung. Im Gegensatz dazu geht es bei *SRE* um die Implementierung des Kerns. Dabei unterscheidet IBM nach drei folgenden Hauptmerkmalen:

- **Entwicklung und Implementierung** (engl. *Development and Implementation*): die **DevOps-Teams** konzentrieren sich auf die Kernentwicklung. Sie arbeiten an einem Produkt oder einer Anwendung, die die Lösung für das Problem eines Kunden darstellt. Sie verfolgen einen agilen Ansatz für die Softwareentwicklung, der ihnen hilft, Anwendungen schnell, qualitativ hochwertig und kontrolliert zu erstellen, zu testen, bereitzustellen und zu überwachen. Bei **SRE** wird an der Implementierung des Kerns gearbeitet. Das *SRE-Team* gibt der Kernentwicklungsgruppe ständig Rückmeldung zu dem entworfenen Produkt und evaluiert die Funktionsweise des Produkts. Somit nutzt *SRE* Betriebsdaten und Software-Engineering, um IT-Betriebsaufgaben zu automatisieren und die Softwarebereitstellung zu beschleunigen, während gleichzeitig das IT-Risiko minimiert wird.
- **Fähigkeiten** (engl. *Skills*): bei **DevOps** sind die Kernentwickler von DevOps, Leute, die gerne Software schreiben. Sie schreiben *Code*, testen ihn und bringen ihn in die Produktion ein, um eine Anwendungslinie zu erhalten, die bei der Lösung eines Problems hilft. **SRE** führt Analysen durch, um herauszufinden, warum etwas schiefgelaufen ist. Es soll sichergestellt werden, dass die gleichen Probleme nicht immer wieder auftreten. Es ist ein proaktives und nicht reaktives

vorgehen. Die Aufgaben, die sich wiederholen, sollen bei *SRE* automatisiert werden, damit sie innovativ sind.

- **Automatisierung** (engl. *Automation*): immer wiederkehrende Aufgabe, sollten automatisiert werden, um die Mühen zu verringern. **DevOps** wird die Bereitstellung automatisieren. Es werden Aufgaben und Funktionen automatisiert. **SRE** automatisiert Redundanz und manuelle Aufgaben, die in programmatische Aufgaben umgewandelt werden können.

DevOps wie auch *SRE* ((Betsy Beyer, 2016), S. 7-8) sollen die Lücke zwischen den Entwicklungs- und Operations-Teams schließen, um eine beschleunigte Servicebereitstellung zu gewährleisten. Dabei setzen beide Methoden verschiedene Schwerpunkte. Devops fokussiert sich auf Liefargeschwindigkeit von Produkten, Software oder Inkrementen und *SRE* fokussiert sich auf die Verlässlichkeit eines Produktes.

2.9 Das Spotify Modell

Das Spotify Modell wurde erstmals 2012 (Ruth, 2022) von Henrik Kniberg und Anders Ivarsson in dem Whitepaper *Scaling Agile @ Spotify*¹ vorgestellt. Bei Spotify² handelt es sich, mit circa 286 Millionen Benutzern um den größten und beliebtesten Audiostreaming-Abonnementdienst der Welt. Der einzigartige Ansatz des Spotify Modells wird als Schlüsselfaktor für den Erfolg von Spotify genannt. Die Agilität des Teams wird durch mitarbeiterorientierten und autonomen Ansatz zur Agile-Skalierung mit einem Schwerpunkt auf Kultur und Netzwerk erreicht. Die Produktivität und Innovationskraft in Unternehmen wurde durch Autonomie, Kommunikation, Verantwortung und Qualität in den Teams enorm gesteigert. Deswegen haben die Entwicklerteams von Spotify ihre Erfahrungen dokumentiert und weltweit geteilt. Bei dem Spotify-Modell handelt es sich um kein *Framework*. Der Spotify-Coach Henrik Kniberg beschreibt es eher als eine spezielle Sichtweise von Spotify auf die technischen und kulturellen Aspekte der Skalierung. Es ist ausgelegt für die Organisation mit mehreren Teams in einer Produktentwicklungsorganisation. Dabei wird die Notwendigkeit von Kultur und Netzwerken in einem Unternehmen unterstrichen.

Der Umgang mit mehreren Teams (Kniberg & Ivarsson, 2022) in einer Produktentwicklungsorganisation ist immer eine Herausforderung, vor allem wenn die Teams global verteilt sind. Spotify hat bewiesen, dass trotz seiner Skalierung auf über 30 Teams in drei Städten eine agile Denkweise beibehalten wird. Spotify als Unternehmen ist faszinierend, weil es erst seit 6 Jahren besteht und bereits über 15 Millionen aktive Nutzer und davon über 4 Millionen zahlende Nutzer hat. Die Schlüsselemente des Spotify-Modells sind folgende:

- **Kader** (engl. *Squads*): stellt die Grundeinheit der Entwicklung bei dem Spotify-Modell dar. Ein Squad ähnelt einem Scrum-Team und ist so konzipiert, dass es sich wie ein Mini-Startup anfühlt. Sie sitzen zusammen und verfügen über alle Fähigkeiten und Werkzeuge, die zum Entwerfen, Entwickeln, Testen und Überführen in die Produktion erforderlich sind. Sie sind ein selbstorganisiertes Team und entscheiden selbst über ihre Arbeitsweise wie *Scrum-Sprints* oder *Kanban-Boards*. Jedes Team hat eine langfristige Aufgabe, wie z. B. die Entwicklung und Verbesserung des *Android-Clients*. Die *Squads* werden ermutigt, Lean-Startup-Prinzipien wie Minimum-Viable-Product (MVP) und validiertes Lernen anzuwenden. MVP bedeutet, dass das

¹<https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf>, letzter Zugriff 27.05.2022

²<https://www.spotify.com/de/>, letzter Zugriff 27.05.2022

Produkt früh und oft veröffentlicht wird, und validiertes Lernen bedeutet, dass Metriken und A/B-Tests verwendet werden, um herauszufinden, was wirklich funktioniert und was nicht. Ein Squad hat keinen formell ernannten *Squad-Leader*, aber er hat einen Produktverantwortlichen. Der *Product Owner* ist für die Priorisierung der vom Team zu erledigende Arbeit verantwortlich, hat aber keinen Einfluss darauf, wie das Team seine Arbeit erledigt. Die *Product Owner* der verschiedenen *Squads* arbeiten zusammen, um ein übergeordnetes Roadmap-Dokument zu pflegen, um zu zeigen, wohin ein Produkt als Ganzes gesteuert wird. Ein Team hat auch Zugang zu einem agilen Coach, der ihm hilft, seine Arbeitsweise weiterzuentwickeln und zu verbessern.

- **Stämme (engl. Tribes):** sind eine Ansammlung von *Squads*, die in verwandten Bereichen arbeiten, wie zum Beispiel der Entwicklung vom *Backend*. Der Stamm kann als „Inkubator“ für die Ministartups der *Squads* betrachtet werden. Jeder Stamm hat einen Stammesleiter, der dafür verantwortlich ist, den *Squads* innerhalb des Stammes den bestmöglichen Lebensraum zu bieten. Die *Squads* eines Stammes befinden sich alle im selben Büro, normalerweise direkt nebeneinander, und die nahe gelegenen Aufenthaltsräume fördern die Zusammenarbeit zwischen den *Squads*. Stämme sind auf eine Größe von weniger als 100 Personen ausgelegt. Die *Squads* veranstalten regelmäßig Versammlungen, ein informelles Treffen, bei dem sie dem Rest des Stammes zeigen, woran sie arbeiten. Dazu gehören zum Beispiel Live-Demos von funktionierender Software. Bei mehreren *Squads* in einem *Tribe* wird es immer Abhängigkeiten geben. Die unterschiedlichen *Squads* müssen manchmal zusammenarbeiten, um etwas wirklich Großartiges zu schaffen.
- **Verbände (engl. Chapters):** alles hat seine Kehrseite, und die potenzielle Kehrseite der völligen Autonomie ist der Verlust von Größenvorteilen. Wenn jedes *Squad* völlig autonom wäre und keine Kommunikation mit anderen *Squads* hätte, dann wäre die Struktur eines Unternehmens nicht besonders sinnvoll. Damit das nicht passiert und die *Tribes* und *Squads* sich nicht in eigene Mini-Unternehmen verwandelt, gibt es *Chapters* und *Guilds*. Diese halten das gesamte Unternehmen zusammen, ohne, dass zu viel Autonomie eingebüßt werden muss. Jedes *Chapter* trifft sich regelmäßig, um sein Fachgebiet und seine spezifischen Herausforderungen zu besprechen, zum Beispiel das *Web-Developer-Chapter* oder das *Backend-Chapter*. Der Chapter-Leiter ist der Vorgesetzte seiner Chapter-Mitglieder, mit allen traditionellen Aufgaben wie Personalentwicklung und Gehaltsfestsetzung. Er ist jedoch auch Teil eines Teams und an der täglichen Arbeit beteiligt, was ihm hilft, den Kontakt zur Realität zu halten.
- **Gilde (engl. Guilds):** ist eine weitreichendere „Interessengemeinschaft“, eine Gruppe von Menschen, die Wissen, Werkzeuge, *Code* und Praktiken teilen wollen. *Chapter* sind immer lokal auf einen *Tribe* beschränkt, während eine Gilde in der Regel die gesamte Organisation umfasst. Eine tribesübergreifend Gilde ist zum Beispiel die Webtechnologie-Gilde. Eine Gilde umfasst oft alle *Chapter*, die in diesem Bereich arbeiten und ihre Mitglieder. Die Webtechnologie-Gilde umfasst somit alle Webentwickler in allen Webtechnologien-Chaptern. Es kann aber jeder, der Interesse hat, jeder Gilde beitreten.

Die folgende Abbildung 2.8 veranschaulicht das Spotify-Modell und deren Unterteilung in *Squads*, *Tribes*, *Chapter* und *Guilds*. Die Abbildung zeigt nur eine mögliche Darstellung von *Chaptern*. Die Realität ist jedoch immer komplizierter als eine Abbildung. Zum Beispiel sind die Chaptermitglieder nicht gleichmäßig über die *Squads* verteilt. Einige *Squads* haben viele Webentwickler, andere haben keine. Die Abbildung soll eine allgemeine Vorstellung vermitteln.

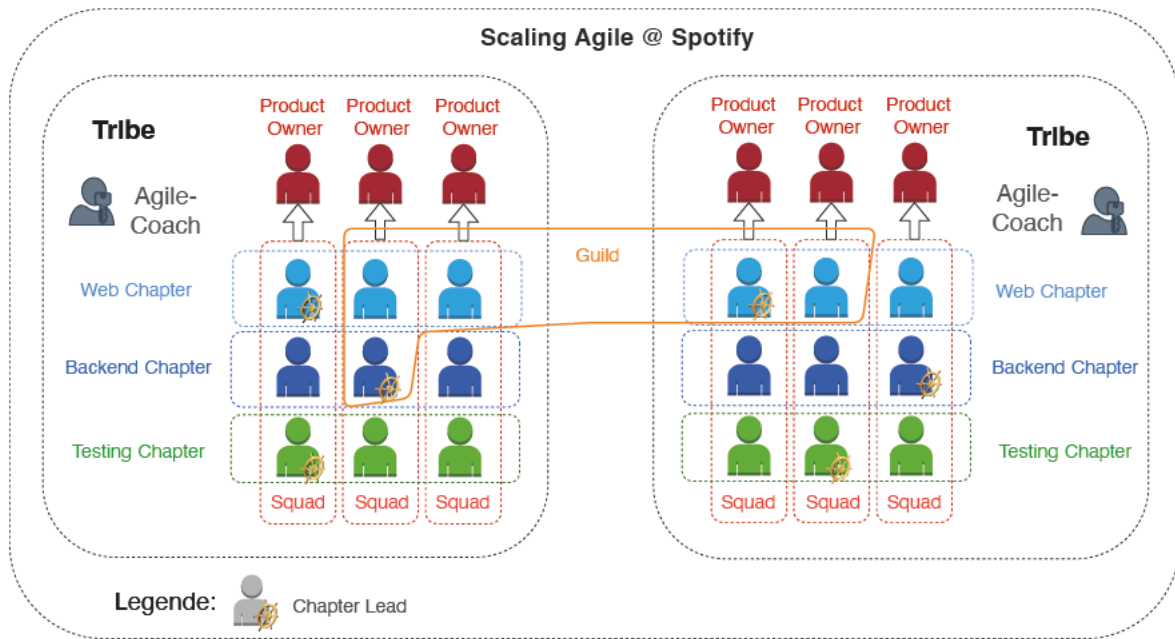


Abbildung 2.8: Das Spotify-Modell, Bildquelle: eigene Darstellung in Anlehnung an (Kniberg & Ivarsson, 2022)

Auch bei dem Spotify-Modell (Kniberg & Ivarsson, 2022) wird deutlich, dass die Unternehmenskultur im Vordergrund steht und verschiedene Teams mit verschiedenen Schwerpunkten zusammengetan werden, um eine bestmögliche Lösung und damit auch ein solides Produkt auf dem Markt zu bringen. Der Gestaltungsraum der Teams darf nicht durch Vorgaben eingeschränkt werden, damit diese sich entfalten können. Da es kein *Framework* ist, gibt es so gesehen kein „Spotify Modell“, welches ein Unternehmen anwenden kann. Spotify hat seine Erfahrung geteilt und Unternehmen können sich von diesem Modell inspirieren lassen und es für die eigenen Teams anpassen und etablieren.

2.10 Netzwerk

In diesem Abschnitt werden die wichtigsten Grundlagen der Netzwerktechnik erläutert, welche notwendig sind, um zu verstehen, wie Netzwerke unter Linux konfiguriert werden und wie diese von Container genutzt werden, um die Interprozesskommunikation zwischen Container zu ermöglichen. Des Weiteren beschreibt der Abschnitt die Funktionsweise eines *Cert-Issuers* und erläutert dies an einem der bekanntesten *Open-Source Cert-Issuer* Letsencrypt¹. Die Netzwerkgrundlagen beinhalten unter anderem das Zusammenspiel eines DNS-Servers, um die generierten Domains aufzulösen, TLS und SSL für die verschlüsselte Kommunikation zwischen Virtuellen-Maschinen oder Containern, sowie die Rolle einer Public-Key-Infrastructure (PKI). Dabei werden nur die notwendigsten Aspekte der einzelnen Themen entsprechend unter dem oben genannten Fokus erläutert.

2.10.1 DNS

Der Domain-Name-System (DNS) stellt einen der wichtigsten Dienste (Elektronik-Kompodium.de, 2022) in vielen IP-basierten Netzwerken dar. Ein DNS-Server löst Computernamen, auch Domain

¹<https://letsencrypt.org>, letzter Zugriff am 05.06.2022

genannt, in IP-Adressen auf. Deswegen wird ein DNS-Server auch als das Telefonbuch des Internets bezeichnet. Um eine Verbindung zu einem Server aufzubauen, wird eine IP-Adresse benötigt. Das DNS löst nicht nur Namen in IP-Adressen auf, sondern agiert auch umgekehrt, um zu einer IP-Adresse einen Namen (*Reverse-IP-Lookup*) zu erhalten. Ist der DNS-Server nicht in der Lage, einen Namen aufzulösen, dann kann keine aktive Verbindung zum Host hergestellt werden. Bei DNS gibt es keine einzelne zentrale Datenbank, sondern viele tausend DNS-Server, auf denen die Informationen verteilt sind. Wenn der Benutzer eine Seite aufruft, dann fragt der Browser einen DNS-Server, der in der Konfiguration des Hosts hinterlegt ist. In der Regel ist der Router des Internet-Zugangs. Die Router wie Fritz!Box¹ nutzen per Default die DNS-Server von Google wie zum Beispiel 8.8.8.8. Die Kommunikation im Netzwerk wäre somit ohne DNS praktisch unmöglich. Der Benutzer müsste die IP-Adresse aller Server kennen und diese müssten im besten Fall alle statisch sein. Der Benutzer kann auch alle IP-Adressen und die dazugehörige Domain in die lokale Host-Datei eintragen, aber dies ist aufgrund der ständig ändernden IP-Adresse nicht zielführend. Das Verfahren mit manuellen Einträgen in der Host-Datei wurde zur Namensauflösung im ARPANET (Vorgänger des Internets) händisch gepflegt. Die fehlende Eindeutigkeit in der Host-Datei, welche für die Validierung der Richtigkeit der DNS-Einträge konnte, nicht gewährleistet werden. Deswegen sieht das DNS einen autoritative Nameserver vor, welcher mit DNSSEC² die Möglichkeit bietet, zu prüfen, ob eine *DNS-Response* von einem vertrauenswürdigen DNS-Server stammt. Es gibt verschiedene Arten von DNS-Server, welche verschiedene Funktionen wie zum Beispiel: *DNS-Root-Server*, *autoritativer Nameserver* (zuständig für eine DNS-Zone) und *Nicht-autoritativer Nameserver*. Die Funktionen und Aufgaben werden bei nicht-autoritativer Nameserver nochmals weiter unterteilt in: *Cache*, *Forwarder* und *Resolver*.

Eine weitere (Elektronik-Kompendium.de, 2022) Aufgabe des DNS ist die Verwaltung von DNS-Zonen. Ganz oben befindet sich die DNS-Wurzel (**engl.** *DNS-Root*), die Informationen zu den Top-Level-Domain (TLD) speichert. Eine Ebene tiefer Second-Level-Domain (SLD) befinden sich weitere Namensserver, die für Domains und Subdomains zuständig sind. Bei der Verwendung einer DNS-Zone werden die Einträge in einer Zonendatei erstellt. Diese werden als *Resource Records* bezeichnet. Bei jedem *Resource Record* handelt es sich um einen *Record-Type*, welcher eine bestimmte Information zur Verfügung stellt. Zum Beispiel beschreibt der Eintrag *MX* einen zuständigen Mailserver für die Zone (Mail Exchange) oder der Eintrag (*Record-Type: A/AAAA*) ordnet eine IP-Adresse der Zone zu.

2.10.2 Domain

Der Domain-Namen (Elektronik-Kompendium.de, 2022) ordnet für den Menschen schwierig zu merkende IP-Adressen Namen zu. Ein Domain-Name besteht typischerweise aus drei oder mehr Teilen. Die einzelnen Teile werden voneinander, durch einen Punkt getrennt. Ein Domain-Name beginnt ganz rechts mit einem abschließenden Punkt und wird von rechts nach links gelesen. Bei der Domain `www.masterarbeit.edu.` stellt der Punkt vor `edu.` die DNS-Wurzel dar. Das „`edu`“ ist die Top-Level-Domain. Die SLD ist *masterarbeit*. Der Computernamen (Dienst oder Host) ist zum Beispiel `www.` oder `ftp`. Zusätzlich gibt es Sub-Level-Domain (kurz Subdomain), die die folgende Struktur aufweisen können, `www.dortmund.masterarbeit.de`. Die Subdomain wird zum Beispiel verwendet, um eine logische Gliederung bei großen Unternehmen von Abteilungen oder Produktionsangeboten zu schaffen. Bei dem Beispiel `www.dortmund.masterarbeit.de` steht die Subdomain für den Standort *Dortmund* und leitet zum Beispiel auf die Prüfungsverordnung von der FH-Dortmund. Es handelt sich um eine ausgedachte Domain, die zur Erklärung der Struktur dient. Die SLD kann zwar einen beliebigen

¹<https://www.heise.de/tipps-tricks/Fritzbox-DNS-Server-aendern-5054919.html>, letzter Zugriff am 06.06.2022

²<https://www.elektronik-kompendium.de/sites/net/1706071.htm>, letzter Zugriff am 06.06.2022

Namen annehmen, aber dieser muss einzigartig unter der *TLD* sein. Wenn die Anforderung besteht, dass mehrere Subdomains auf denselben Server zeigen, dann bietet *Canonical-Name* (kurz *CNAME*) eine mögliche Abhilfe, verschiedene Einträge zu erstellen, die auf eine bestimmte Domain zeigen.

2.10.3 Root-CA, Intermediate und Server-Cert

Eine **Root-CA** (Nohe, 2019) ist eine Zertifizierungsstelle (**engl.** *Certificate Authority*), die über eine oder mehrere vertrauenswürdige *Roots* verfügt. Das bedeutet, dass sie über Wurzeln in den Vertrauensspeichern der wichtigsten Browser verfügt. Das Root-Zertifikat, oft auch als vertrauenswürdige Wurzel bezeichnet, steht im Mittelpunkt des Vertrauensmodells, das der *Public-Key-Infrastructure* und damit auch SSL/TLS zugrunde liegt. Jedes Gerät enthält einen sogenannten Root-Speicher. Ein *Root-Store* ist eine Sammlung von vorab heruntergeladenen Root-Zertifikaten und deren öffentlichen Schlüsseln, die sich auf dem Gerät selbst befinden. In der Regel verwendet das Gerät den *Root-Store*, der in seinem Betriebssystem enthalten ist, andernfalls kann es einen *Root-Store* eines Drittanbieters über eine Anwendung wie einen Webbrowser verwenden. Die Zertifizierungsstellen stellen keine Server-/Leaf-Zertifikate für Endbenutzer direkt von ihren *Roots* aus. Die Root-CA sind zu wertvoll und das Risiko ist zu groß, dass das Zertifikat der *Root-CA* missbraucht werden kann. Um sich abzusichern, stellen die Zertifizierungsstellen daher in der Regel ein sogenanntes Zwischenstammzertifikat (**engl.** *intermediate-Root-CA*) aus. Die Zertifizierungsstelle (*Root-CA*) signiert das Zwischenzertifikat (Intermediate) mit ihrem privaten Schlüssel, wodurch es vertrauenswürdig wird. Dann verwendet die CA den privaten Schlüssel des Zwischenzertifikats, um SSL-Zertifikate für Endbenutzer zu signieren und auszustellen. Dieser Prozess kann sich mehrmals wiederholen. Um das Zertifikat zu authentifizieren, folgt der Browser der Zertifikatskette. Um ein SSL-Zertifikat ausstellen zu lassen, muss zunächst eine Zertifikatsanforderung (**engl.** *Certificate-Signing-Request* (CSR)) und ein privater Schlüssel erstellt werden. Im einfachsten Fall wird dann, geht der *CSR* an die Zertifizierungsstelle, die dann das SSL-Zertifikat mit dem privaten Schlüssel des Stammzertifikats signiert und es zurückschickt. Die Verbindungen von der *Root-CA* über *Intermediate-CA* zum Server-Zertifikat bilden die Zertifikatskette (**engl.** *full-chain*).

Das folgende Beispiel (Nohe, 2019) beschreibt mithilfe der Abbildung 2.9 die gesamte Vertrauenskette vom Server bis zu *Root-CA* und was passiert, wenn ein Benutzer eine Seite aufruft. Die *Root-CA* verfügt über ein Schlüsselpaar (*Private- und Public-Key*). Die *Intermediate-CA* stellt einen *CSR* gegen die *Root-CA*. Die *Root-CA* signiert (verschlüsselt) den *CSR* mit dem privaten Schlüssel und liefert die Signatur an die *Intermediate-CA* zurück. Die *Intermediate-CA* hat als *Subject* die *Root-CA* in der Signatur eingetragen. Die Verifizierung der Signatur der *Root-CA* erfolgt über den *Public-Key*. Analog dazu verhält es sich ähnlich zwischen *Intermediate-CA* und den einzelnen Servern. Sollte der private Schlüssel der *Intermediate-CA* im Netz publiziert werden, dann annulliert die *Root-CA* den *CSR* von der *Intermediate-CA* und darunter auch alle von der *Intermediate-CA* signierte Server. Dadurch können einzelne Zweige, die sich durch die Struktur ergeben, abgeschaltet werden. Die *Root-CA* kann auf die *Intermediate-CA* verzichten und die Server-Zertifikate direkt signieren. Das Problem ist dann, dass alle Server an der *Root-CA* hängen und falls der *Private-Key* geleakt wird, muss der ganze Baum von der Wurzel abgeschaltet werden. Das war die Erläuterung zu der *Root-CA* Vertrauenskette. Doch wie interagiert ein Benutzer und welche Rolle spielen dabei die ausgestellten Zertifikate. Dies wird nochmals an der Abbildung 2.9. Der Benutzer (1) ruft die Seite <https://masterarbeit.edu> auf. Der Browser (1) fragt eine Verbindung zum Server an. Der Server (2) sendet ein Zertifikat mit dem *Public-Key* zurück. Der Browser (3) überprüft und validiert das Zertifikat, verschlüsselt die Daten (4) und schickt diese an den Server zurück. Der Server entschlüsselt (5) die Daten mit dem *Private-Key*.

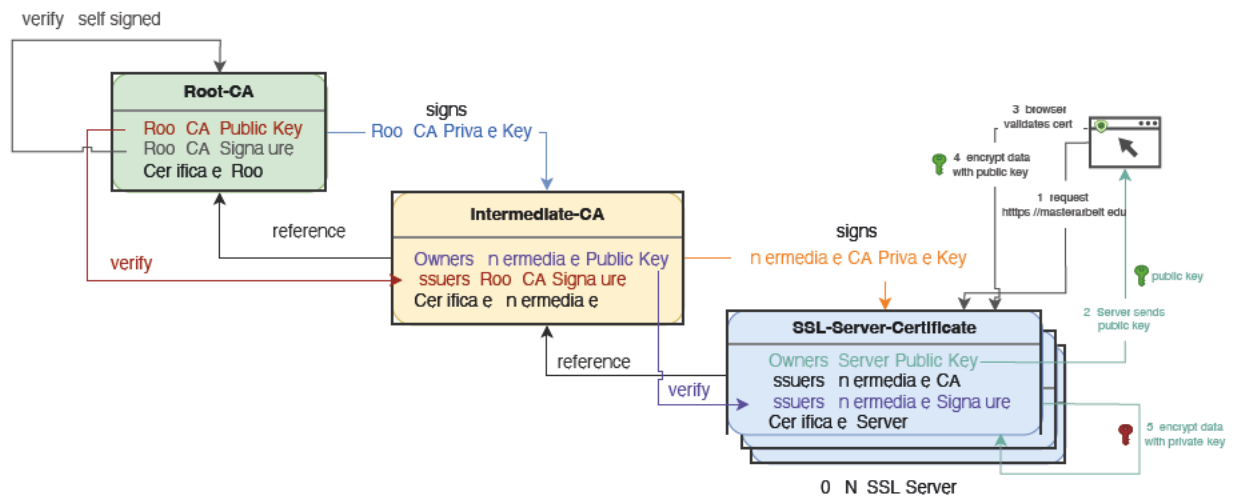


Abbildung 2.9: Root-CA-Chain-Trust, Bildquelle: eigene Darstellung in Anlehnung an https://en.wikipedia.org/wiki/Root_certificate, letzter Zugriff am 05.06.2022 und <https://www.thesslstore.com/blog/root-certificates-intermediate/>, letzter Zugriff am 05.06.2022

2.10.4 TLS/SSL

Im letzten Jahrzehnt des 20. Jahrhunderts ((Ristic, 2014), S. 1) wurde das Internet immer beliebter und hat unser Leben für immer verändert. Wir leben in einer zunehmend vernetzten Welt. Heute verlassen wir uns auf unsere Telefone und Computer, um zu kommunizieren, einzukaufen, Rechnungen zu bezahlen oder zu reisen. Viele von uns, die ständig ein Gerät in der Tasche haben, sind nicht mit dem Internet verbunden, sondern stellen zum Teil das Internet dar. Es gibt bereits mehr Telefone als Menschen. Die Zahl der Smartphones geht in die Milliarden und steigt rasant an. Inzwischen gibt es Pläne, alle möglichen Geräte mit demselben Netz zu verbinden. Alle Geräte, die mit dem Internet verbunden sind, haben eines gemeinsam. Sie verlassen sich auf die Protokolle Secure-Socket-Layer (SSL) und Transport-Layer-Security (TLS), um die Datenübertragung zu schützen. Als das Internet ((Ristic, 2014), S. 1-2) ursprünglich entwickelt wurde, hat man sich wenig Gedanken über die Sicherheit gemacht. Infolgedessen sind die wichtigsten Kommunikationsprotokolle von Natur aus unsicher und verlassen sich auf das ehrliche Verhalten aller beteiligten Parteien. Das mag damals funktioniert haben, als das Internet noch aus einer kleinen Anzahl von Knoten bestand und überwiegend von Universitäten zur Forschungszwecke genutzt werden. Bei SSL und TLS handelt es sich um kryptografische Protokolle, die eine sichere Kommunikation über eine unsichere Infrastruktur ermöglichen sollen. Das bedeutet, wenn diese Protokolle ordnungsgemäß eingesetzt werden, ein Kommunikationskanal zu einem beliebigen Dienst im Internet mit hinreichender Sicherheit geöffnet wird und es wird sichergestellt, dass mit dem richtigen Server kommuniziert wird. Die Informationen werden austauscht, ohne dass Daten in die Hände eines anderen gelangen. Diese Protokolle schützen die Kommunikationsverbindung oder die Transportschicht, daher auch der Name TLS.

Sicherheit ist nicht das einzige Ziel von TLS. Es hat eigentlich vier Hauptziele, die hier in der Reihenfolge ((Ristic, 2014), S. 2) ihrer Priorität aufgeführt sind:

- **Kryptografische Sicherheit** (engl. *Cryptographic Security*): eine sichere Kommunikation zwischen zwei Parteien zu ermöglichen, die Informationen austauschen wollen.
- **Interoperabilität** (engl. *Interoperability*): Unabhängige Programmierer sollten in der Lage

sein, Programme und Bibliotheken zu entwickeln, die in der Lage sind, unter Verwendung gemeinsamer kryptografischer Parameter miteinander zu kommunizieren.

- **Erweiterbarkeit** (engl. *Extensibility*): TLS ist ein Rahmenwerk für die Entwicklung und den Einsatz von kryptografischen Protokollen. Sein wichtiges Ziel ist es, unabhängig von den tatsächlich verwendeten kryptografischen Primitiven, wie zum Beispiel Hash-Funktionen zu sein.
- **Effizienz** (engl. *Efficiency*): das endgültige Ziel besteht darin, alle vorangegangenen Ziele mit akzeptablen Leistungskosten zu erreichen. Dabei sollen kostspielige kryptografische Operationen auf ein Minimum reduziert werden und ein Zwischenspeicherschema für Sitzungen bereitgestellt werden.

2.10.5 Public-Key-Infrastruktur (PKI)

Da die Sicherheit (Albarqi et al., 2015) bei der Kommunikation über elektronische Netze von wesentlicher Bedeutung ist, sollten Strukturen entwickelt werden, die ein hohes Maß an Sicherheit bieten. Die Public-Key-Infrastruktur ist eine Möglichkeit, Sicherheitsmaßnahmen durch die Implementierung von Schlüsselpaaren zwischen Nutzern bereitzustellen. In diesem Beitrag wird ein Überblick über die Public-Key-Infrastruktur gegeben und der damit verbundenen verschiedenen Komponenten und Funktionsweisen.

Auf der untersten Ebene ((Ristic, 2014), S. 2-3) stützt sich die Kryptografie auf verschiedene kryptografische Primitive. Jedes Primitiv wurde mit Blick auf eine bestimmte nützliche Funktion entwickelt. So kann beispielsweise ein Primitiv für die Verschlüsselung und ein anderes für die Integritätsprüfung verwendet werden. Die Primitive allein sind nicht sehr nützlich, aber wenn diese in Schemata und Protokollen kombiniert werden, dann entsteht eine robuste Sicherheit. Es gibt symmetrische und asymmetrische Verschlüsselungen. Die **symmetrische Verschlüsselung** oder Private-Key-Kryptografie ist eine Methode zur Verschleierung, die eine sichere Übertragung von Daten über unsichere Kommunikationskanäle ermöglicht. Um sicher zu kommunizieren, einigen sich die Parteien zunächst auf den Verschlüsselungsalgorithmus und einen geheimen Schlüssel. Wenn Client-A später Daten an Client-B senden möchte, verwendet der Client den geheimen Schlüssel, um die Daten zu verschlüsseln. Client-B verwendet denselben Schlüssel, um die Daten zu entschlüsseln. Client-A und Client-B können so lange sicher miteinander kommunizieren, solange der geheimen Schlüssel auch geheim halten wird. Der Schlüssel (Albarqi et al., 2015) ist schwer zu verwalten, da es nur einen Schlüssel pro Benutzer gibt. Um dieses Problem zu überwinden, wird bei *PKI* eine andere Art der Kryptografie vorgeschlagen, die als öffentlicher Schlüssel oder **asymmetrische Kryptografie** bekannt ist und bei der jeder Benutzer über ein Schlüsselpaar verfügt. Jeder Benutzer verfügt über einen öffentlichen und einen privaten Schlüssel. Da der öffentliche Schlüssel nicht geheim ist, wurde das Problem der Schlüsselverwaltung gelöst. Dafür entsteht ein neues Problem, nämlich das Problem der Authentifizierung oder Namensverwaltung. Die Public-Key-Infrastruktur wurde entwickelt, um dieses Problem zu lösen und die Public-Key-Kryptografie zu unterstützen. Die Authentifizierung ist der Prozess der Nutzung aller *PKIs*. Es ist außerdem (BSI, 2022) mit demselben privaten Schlüssel möglich, eine Datei digital zu signieren. Die Signatur, ob die Datei unverändert ist, kann dann mit dem zugehörigen öffentlichen Schlüssel geprüft werden. Der Fokus der Arbeit liegt in der verschlüsselten Kommunikation zwischen virtuellen Maschinen, Containern oder virtualisierten Anwendungen. Bei der Kommunikation zwischen virtuellen Maschinen, von- oder zu einer Anwendung werden digitale Zertifikate verwendet. Ein solches digitales Zertifikat beinhaltet immer den öffentlichen Schlüssel eines Schlüsselpaares. Zwei Kommunikationspartner tauschen ihre Zertifikate für die Kommunikation und Übermittlung der Nachrichten aus. Damit erhalten sie die Möglichkeit, Nachrichten so zu verschlüsseln, dass sie nur jeweils von einem

der entschlüsseln Kommunikationspartner werden können. Die Ausstellung eines digitalen Zertifikats für einen Server basiert auf einer *Root-CA*, einem optionalen Intermediate und den tatsächlich ausgestellten Zertifikaten für die Kommunikation der Maschinen untereinander (*Server-Certs*), welcher von der *Root-CA* signiert sind.

2.10.6 Zertifikatsaussteller (Cert-Issuer)

Ein Zertifikat ((Ristic, 2014), S. 66-67) ist ein digitales Dokument, das einen öffentlichen Schlüssel, einige Informationen über die damit verbundene Entität und eine digitale Signatur des Zertifikatsausstellers (**engl. Issuers**) enthält. Mit anderen Worten, es ist eine Hülle, die es uns ermöglicht, öffentliche Schlüssel auszutauschen, zu speichern und zu verwenden. Damit sind Zertifikate der Grundbaustein der *PKI*. Ein Zertifikat besteht aus Feldern und aus einer Reihe von Erweiterungen. Oberflächlich betrachtet, ist die Struktur flach und linear, obwohl einige Felder andere Strukturen enthalten. Darunter befindet sich ein Feld für den Issuer. Das Feld **Issuer** enthält den Distinguished-Name (DN)¹ des Zertifikatsausstellers. Es handelt sich um ein komplexes Feld, das je nach dem vertretenen Unternehmen viele Komponenten enthalten kann. Der Issuer enthält vier Komponenten, jeweils eine für Land (C), Organisation (O), Organisationseinheit (OU) und Common-Name (CN).

Dies ist zum Beispiel der Distinguished-Name, der für eines der Stammzertifikate von FH-Dortmund verwendet wird:

- /C=DE
- /O=Verein zur Foerderung eines Deutschen Forschungsnetzes e. V.
- /OU=DFN-PKI
- /CN=DFN-Verein Global Issuing CA Authority

2.10.7 Letsencrypt

Bei Letsencrypt¹ oder Let´s Encrypt, (letsencrypt.org, 2022) was übersetzt „Lasst uns verschlüsseln“ heißt, handelt es sich um eine freie, automatisierte und offene Zertifizierungsstelle. Die Zertifizierungsstelle stellt einen Dienst von Internet-Security-Research-Group (ISRG)² zur Verfügung, welcher den Menschen digitale und signierte Zertifikate ausstellt. Diese werden dann zur Aktivierung von SSL/TLS auf den Servern- und Webseiten eingebunden. Dadurch soll die Sicherheit durch verschlüsselte und verifizierte Kommunikation bei Wahrung der Privatsphäre ermöglicht werden.

Wie funktioniert Letsencrypt?

Das Ziel von Let´s Encrypt ist, mithilfe des ACME-Protokolls³ die Einrichtung eines *HTTPS-Servers* (letsencrypt.org, 2019) automatisiert zu ermöglichen. Dabei soll automatisch ein vertrauenswürdige Browserzertifikat erstellt und signiert werden, ohne dass ein Mensch manuell aktiv werden muss. Dies wird durch einen sogenannten Zertifikatsverwaltungsagenten auf dem *Webserver* erreicht. Der *Webserver* identifiziert sich bei Let´s Encrypt mit seinem öffentlichen Schlüssel. Beim allerersten Kontakt, generiert der *Webserver* ein neues Schlüsselpaar und weist gegenüber der Let´s Encrypt Zertifizierungsstelle nach, dass der Server eine oder mehrere Domains kontrolliert. Der Beweis erfolgt in einer

¹<https://letsencrypt.org>, letzter Zugriff am 06.06.2022

²<https://www.abetterinternet.org>, letzter Zugriff am 06.06.2022

³<https://datatracker.ietf.org/doc/html/rfc8555>, letzter Zugriff am 06.06.2022

Herausforderung, der *ACME-Challenge*. Der Server startet einen *ACME-Request* und die Zertifizierungsstelle von Let's Encrypt gibt eine oder mehrere Herausforderungen heraus, um sicherstellen zu können, dass der Server der Eigentümer der Domain ist. Dabei wird in der Regel das Skript *acme.sh* auf dem Server ausgeführt, dieses generiert einen Hash-Wert, denn der Eigentümer initial das erste Mal unter der Domain einträgt. Anschließend validiert die CA von Let's Encrypt den Eintrag und bestätigt den Eigentümer. Zum Schluss erhält der Benutzer ein von Let's Encrypt signiertes Zertifikat, einen Private-Key, mit welchem der Server die verschlüsselten Anfragen von Clients entgegennimmt, das Intermediate Zertifikat, welches die *Root-CA* von Let's Encrypt enthält und das Full-Chain-Zertifikat. Einige Anwendungen benötigen das Full-Chain-Zertifikat, damit sie Anfragen akzeptieren.

Das folgende Sequenzdiagramm in der Abbildung 2.10 stellt eine visuelle, vereinfachte Darstellung der beschriebenen Schritte dar.

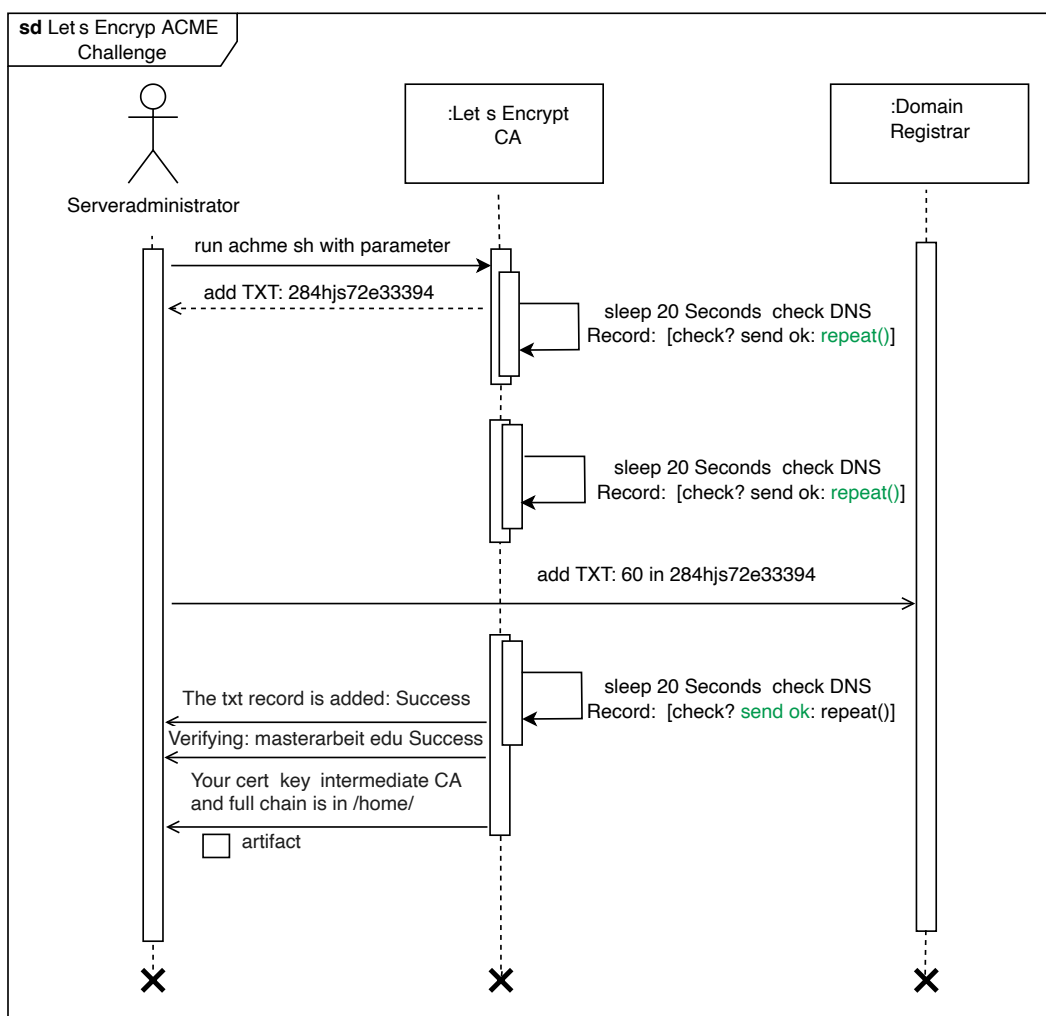


Abbildung 2.10: Sequenzdiagramm: Let's Encrypt ACME-Challenge, Bildquelle: eigene Darstellung

2.10.8 Container-Netzwerke

Bei Container Networking (VMware, 2022) handelt es sich, um einen aufstrebenden Mechanismus für das *Sandboxing*¹ von Anwendungen, der in Home-Desktops und Netzwerklösungen für Unternehmen im Web verwendet wird und vom Konzept her einer virtuellen Maschine ähnelt. Alle Anwendungen innerhalb des Containers dürfen nur auf Dateien oder Ressourcen innerhalb des Containers zugreifen oder diese ändern. Es ist möglich, mehrere Container gleichzeitig auszuführen, jeder mit seinen eigenen Installationen und Abhängigkeiten. Um Isolation zu schaffen, stützt sich ein Container auf zwei Funktionen des Linux-Kernels: Namespace und *cgroups*. Um dem Container eine eigene Sicht auf das System zu geben und ihn von anderen Ressourcen zu isolieren, wird für jede Ressource ein Namespace erstellt, der vom restlichen System nicht gemeinsam genutzt wird. Kontrollgruppen (*cgroups*) werden dann zur Überwachung und Begrenzung von Systemressourcen wie CPU, Speicher oder Netzwerk verwendet.

Es gibt fünf Arten von Containernetzwerken, die heute verwendet werden. Dabei konzentrieren sich die Merkmale auf IP-pro-Container- bzw. IP-pro-Pod-Modelle und die Notwendigkeit der Netzwerk-adressübersetzung (NAT) im Gegensatz zu einer nicht erforderlichen Übersetzung. Es handelt sich um folgende fünf Arten von Containernetzwerken:

- **Keine** (**engl.** *None*): der Container erhält einen Netzwerkstapel, hat jedoch keine externe Verbindung. Dies ist nützlich für das Testen von Containern, die keine externe Kommunikation benötigen.
- **Brücke** (**engl.** *Bridge*): die Container, die über ein internes Host-Netzwerk verbunden, können über die Netzwerke des Hosts kommunizieren. Auf die Container kann von außerhalb des Hosts nicht zugegriffen werden. Das Bridge-Netzwerk ist der Standard für Docker-Container.
- **Host**: diese Konfiguration ermöglicht es einem erstellten Container, den Netzwerk-Namensraum des Hosts gemeinsam zu nutzen, wodurch der Container Zugriff auf alle Netzwerkschnittstellen des Hosts erhält. Diese Konfiguration ist die am wenigsten komplexe der externen Netzwerkkonfigurationen, ist aber aufgrund der gemeinsamen Nutzung der Netzwerkschnittstellen anfällig für Port-Konflikte oder Angriffe von außen.
- **Unterliegende-Schicht** (**engl.** *Underlay*): öffnet die Host-Schnittstellen direkt für Container, die auf dem Host laufen und führen Port-Mapping aus. Dies macht sie effizienter als Brücken.
- **Oberliegende-Schicht** (**engl.** *Overlay*): verwenden Netzwerk-tunnel für die Kommunikation zwischen den Hosts, sodass sich die Container so verhalten, als befänden sie sich auf demselben Rechner, obwohl sie auf verschiedenen Hosts gehostet werden.

¹<https://www.security-insider.de/was-ist-eine-sandbox-a-740133/>, letzter Zugriff am 06.06.2022

Das Kapitel befasst sich mit der Einführung in Kubernetes und verschafft einen Überblick über die Entwicklung, Architektur und die wichtigsten Komponenten in einem Kubernetes-Cluster. Anschließend gibt das Kapitel einen Einblick in *Kubernetes-Managed-Service* von Microsoft. Das Kapitel beschreibt nicht die gesamte Funktionsweise von Kubernetes und die verschiedenen Distributionen. Es dient als Verständnisgrundlage für die weiteren Kapitel.

3.1 Kubernetes im Überblick

In den letzten zehn Jahren (Vayghan et al., 2019) hat die Computergemeinschaft eine Umstellung auf die Cloud erlebt. Infolgedessen hat die Microservices-Architektur in der Branche große Aufmerksamkeit erregt. Im Gegensatz zum monolithischen Architekturstil geht der Microservices-Architekturstil die Herausforderungen bei der Erstellung von Cloud-nativen Anwendungen an, indem er die Vorteile der Cloud nutzt. Der Architekturstil soll die IT-Branche revolutionieren, hat er bisher nur begrenzte Aufmerksamkeit in der akademischen Welt und der Forschung gefunden. *Microservices* sind eine Umsetzung des dienstorientierten Architekturstils zur Entwicklung von Software, die aus kleinen Containern besteht. Diese verbessern die Verfügbarkeit der Dienste, über die Bereitstellung auf Kubernetes. Bei Kubernetes ((Liebel, 2019), S. 555) handelt es sich um das griechische Wort Steuerermann (**engl.** *Helmsman*), welches als ein offizielles Projekt von Google im Jahr 2014 gestartet worden ist. Google übergab das Projekt an Cloud-Native-Computing-Foundation (CNCF) und das Projekt wurde unter der Apache-2.0-Lizenz¹ gehostet. Es ist die populärsten Container-Cluster und Orchestrierungslösung für Self-Hosted- und Cloud-Umgebungen. Bereits Mitte 2018 arbeiteten rund 1.500 Kontributoren an dem Projekt. Darunter auch namhafte Unternehmen wie Red Hat², welcher als zweitgrößter Upstream-Kontributor nach Google die Kubernetes-Engine seit OpenShift-Version 3 im Unterbau einsetzt. Bei Google handelt es sich nicht nur um einen Kontributor, sondern auch um den größten Geldgeber, welcher für den internen Service Google-Kubernetes-Engine (GKE)³ der eigenen Google-Cloud-Plattform (GCP)⁴ Kubernetes selbst einsetzt. Die Hauptaufgabe von Kubernetes ((Liebel, 2019), S. 556) als Engine ist die Verwaltung und Orchestrierung von Containern innerhalb eines bestehenden Clusters, welcher in Regel in einer produktiven Umgebungen aus mindestens drei *Control-Plane-Nodes* (ehemalig *Master-Nodes*) und *N-Worker-Nodes* besteht. Die Aufgabe von Kubernetes ist unter anderem Software als *Microservices* in *Pods*, welche einen logischen Behälter für eine Ansammlung von Containern darstellen, zu verwalten. Dazu gehört zum Beispiel die Überwachung der Erreichbarkeit einer Anwendung und bei Bedarf einer Umverteilung der Anwendung auf einen entsprechenden *Node*, der genug Ressourcen zur Verfügung hat. Es handelt sich um ein selbstheilendes *Life-Cycle-Management* der gesamten Kubernetes-Plattform und der darauf laufenden *Services*, Anwendungen und *Workloads*.

¹<https://www.apache.org/licenses/LICENSE-2.0>, letzter Zugriff 31.05.2022

²<https://www.redhat.com/>, letzter Zugriff 31.05.2022

³<https://cloud.google.com/kubernetes-engine>, letzter Zugriff 31.05.2022

⁴<https://cloud.google.com/gcp/>, letzter Zugriff 31.05.2022

Control-Plane-Nodes wie drei, fünf oder sieben empfohlen. Über das Command-Line-Interface (CLI) *kubectl* kommuniziert der Benutzer mit dem Cluster, um Ressourcen einzupflegen, zu löschen oder zu manipulieren. Die *Nodes* sollten in einer produktiven Umgebung als dedizierte Management-Plattformen laufen und selbst keine Benutzer-Pods hosten, sondern nur *Pods*, die für das Aufrechterhalten des Cluserbetriebes notwendig sind. Damit soll gewährleistet werden, dass die Ressourcen wie CPU, RAM und Storage durch die Arbeitslast der Anwendungen nicht vollständig ausgeschöpft werden und die *Control-Planes* ihrer Hauptaufgabe der Orchestrierung nachkommen. Auf den *Control-Plane-Nodes* laufen in der Regel Dienste wie *API-Server*, *Scheduler*, *Controller-Manager*, *Key-Value-Store* wie der *etcd-Store* und der *kubelet* Service. Die Funktionsweise der Komponenten wird in den folgenden Abschnitten genauer erläutert.

Worker-Node

Bei *Worker-Nodes* ((Liebel, 2019), S. 564) handelt es sich um Arbeitsknoten, auf denen die Anwendungen des Benutzers in *Pods* laufen. Somit hosten die *Worker-Nodes* die eigentliche Arbeitslast des gesamten Kubernetes-Clusters. Jede *Node* die als *Worker* fungiert hat mindestens zwei Dienste laufen. Einmal die *Container-Engine* wie Docker, um den Container-Betrieb zu ermöglichen. Und einmal eine Kubelet-Instanz, die als Vorarbeiter Befehle von den *Control-Planes* erhält und diese an die *Container-Engine* weitergibt. Die Kubelet-Instanz und die *Container-Engine* laufen in der Regel als native Dienste, die vom *systemd* gesteuert werden. Die *Container-Engine* wird für die Steuerung des *Workloads* über das Container-Runtimes-Interface (CRI) benötigt. Der Rest läuft komplett containerisiert. In den meisten Setups eines Multi-Node-Cluster-Betriebs laufen auf den *Worker-Nodes* auch noch Proxy-Instanzen, sowie ein Dienst, der den Knoten über das gespannte Overlay-Netzwerk ins Cluster für die Netzwerkkommunikation einbindet.

3.2.2 Control-Plane-Komponente: API-Server

Der *API-Server* ((Liebel, 2019), S. 564) stellt die zentrale administrative Service-Komponente des Kubernetes-Clusters auf der *Control-Plane* dar. Über diesen werden nicht nur alle Ressourcen des Kubernetes-Clusters verwaltet, sondern auch die gesamte Kommunikation zu den einzelnen Cluster-Ressourcen gewährleistet. An der Stelle verwendet Kubernetes (The Kubernetes Authors, 2021b) ein Hub-and-Spoke-API-Muster. Der *API-Server* stellt den **Hub** und alle Cluster-Komponenten wie *Control-Planes*, *Worker-Nodes* und die vom Benutzer ausgeführten *Pods*, die **Spokes** dar. Somit endet die gesamte API-Nutzung von Knoten bis hin zum *Pod* am *API-Server*. Keine der anderen Komponenten der Steuerebene sind für die Bereitstellung von Remote-Diensten vorgesehen. Der *API-server* ist so konfiguriert, dass er auf einem sicheren *HTTPS-Port* (typischerweise 443) auf Remote-Verbindungen wartet, wobei eine oder mehrere Formen der Client-Authentifizierung aktiviert sind. Die Knoten sollten mit dem öffentlichen Root-Zertifikat für den Cluster ausgestattet sein, sodass sie sich zusammen mit gültigen Client-Anmeldedaten sicher mit dem *API-server* verbinden können. Ein guter Ansatz ist, dass die Client-Anmeldeinformationen, die dem *kubelet* zur Verfügung gestellt werden, in Form eines Client-Zertifikats vorliegen. Der *API-Server* ((Liebel, 2019), S. 565) dient nicht nur der sicheren Kommunikation zwischen *Nodes*, *Pods* und Clients, sondern stellt vielmehr im Prinzip ein allwissendes Archiv dar. Er verwaltet die Ressourcen und *Workloads*, welche unter der Hauptkategorie *apiVersion* geführt werden. Die Kategorie wird nach Unterkategorien *Kind* unterteilt. So bietet zum Beispiel die *apiVersion: v1* eine *Kind: Service*, welcher für die interne Kommunikation zwischen den *Pods* im Kubernetes-Cluster genutzt wird. Wenn der Benutzer eine Ressource im Cluster ausrollen möchte, dann validiert der *API-Server* den Request wie einen Bauplan, in dem er unter der *apiVersion* den *Kind* durchgeht und überprüft, ob der Benutzer alle notwendige Konfiguration gesetzt hat. Ist der Request

zulässig, dann setzt der *API-Server* diesen um. Durch Custom-Resource-Definition (CRD) können Drittanbieter eigene Erweiterung als API-Ressourcen anbieten, welche als Bauplan vom *API-Server* im Archiv aufgenommen werden. Der *API-Server* verarbeitet unterschiedliche Requests wie Anlegen, Lesen, Modifizieren oder Löschen von API-Ressourcen wie *Pods*. Zudem weist der *API-Server* *Pods* über den *Controller-Manager* und *Scheduler* bestimmten *Worker-Nodes* zu. Außerdem synchronisiert er die Pod-Informationen mit den Service-Konfigurationen. Das Archiv bzw. der Zustand des Clusters wird vom *API-Server* in dem *Key-Value-Store* etcd ablegt. Der *API-Server* legt den Soll-Zustand des Clusters in dem *etcd-KV-Store* ab und setzt diesen mithilfe des *Controller-Managers* um. Wenn ein Zustand nicht umgesetzt werden kann, dann schreiben die Kubelet-Prozesse die Informationen im *KV-Store* und setzen somit den *API-Server* darüber wiederum in Kenntnis.

3.2.3 Control-Plane-Komponente: Controller Manager

Bei dem *Controller-Manager* ((Lieber, 2019), S. 565-566) oder besser passend zu der tatsächlichen verwendeten *Binary kube-controller-manager* handelt es sich um eine Instanz, die weitere *Controller* verwaltet. Die *Controller* wie *ReplicaSet*- oder *Deployment-Controller* sind somit integraler Bestandteil des *Controller-Managers*. Der *Controller-Manager* erhält seine Befehle direkt von dem *API-Server*, um den Cluster-Zustand in den Soll-Zustand mithilfe von Controllern zu überführen. Um dies zu erreichen, kümmert er sich unter anderem für die Deployment- oder Replication-Controller-Objekte im gesamten Kubernetes-Clusters. Die genaue Aufgabe des *Controller-Managers* besteht dann darin, dafür zu sorgen, dass für eine K8s-Ressource die definierten *Replicas* mit der im K8s-Cluster ausgerollten Anzahl übereinstimmen. Die Vorgaben zu dem Soll-Zustand des Clusters werden im *etcd-KV-Store* abgelegt. Bei einer singulären Instanz ist immer nur eine Instanz des *Controller-Managers* aktiv. Bei einem Hochverfügbarkeits-Setup mit drei *Control-Planes* sind auch drei Instanzen des *Controller-Managers* aktiv. Es darf aber auch beim *HA-Setup* nur eine aktive Instanz des *Controller-Managers* vorhanden sein. Um dies zu gewährleisten, arbeiten die Prozesse im Multi-Control-Plane-System mit *Single-Active Flags* und die nicht aktiven Instanzen erhalten einen expliziten *Lock*. Dadurch wird sichergestellt, dass die Ressourcen in einem Kubernetes-Cluster immer nur von einer zentralen Stelle aus erfolgen und nicht mehrere Instanzen versuchen, eine Ressource mehrmals auszurollen. Ein Parallel-Betrieb, wo sich die Prozesse synchronisieren und drei Instanzen zur gleichen Zeit aktiv sind, hat bis jetzt nur zu Problemen geführt. Die Hochverfügbarkeit der Prozesse mit dem expliziten *Locks* wird durch einen Hot-Failover-Mechanismus gelöst. Bei einem Failover des aktiven Prozesses mit *Single-Active-Flag*, übernimmt eine der beiden gelockten Instanzen. Es gibt unterschiedliche *Controller*, mit verschiedenen Schwerpunkten. Im Folgenden werden *Controller* erläutert, die für den Betrieb eines Kubernetes-Clusters notwendig sind. Es werden dabei einige *Controller* nicht betrachtet.

Controllers:

In der Robotik und Automatisierung (The Kubernetes Authors, 2021a) ist ein Regelkreis eine nicht abschließende Schleife, die den Zustand eines Systems regelt. *Controller* fungieren somit wie Maschinen, die von einem Cluster-Zustand bei einem Event-Trigger in einen anderen Zustand Cluster-Zustand überführen. In Kubernetes sind *Controller* Regelkreise, die den Zustand ihres Clusters überwachen und dann bei Bedarf Änderungen vornehmen oder anfordern. Jeder *Controller* (The Kubernetes Authors, 2021c) versucht, den aktuellen Zustand des Clusters näher an den gewünschten Zustand heranzuführen.

- **Node Controller:** ist eine Komponente des *Control-Planes* und verwaltet verschiedene Aspekte von *Nodes*. Eine Aufgabe ist, die Zuordnung eines Classless Inter-Domain-Routing (CIDR)-Blocks zu dem *Node*, wenn dieser erfolgreich im Kubernetes-Cluster registriert ist. Eine weitere Aufgabe

ist die Überwachung und Pflege einer Node-Liste. Der *Node-Controller* ist dafür verantwortlich, einen Node-Status zum Beispiel von *NodeReady* auf *ConditionUnknown* zu aktualisieren, falls die *Node* nicht erreichbar ist.

- **Service Controller:** ist dafür verantwortlich auf Events zu lauschen, um *Services* zu erstellen, zu aktualisieren oder zu löschen, falls eine Änderung an einer API-Ressource vorliegt.
- **Replication Controller¹:** stellt sicher, dass zu jedem Zeitpunkt eine bestimmte Anzahl von *Pods* in Betrieb ist. Wenn ein Manifest festlegt, dass drei *Pods* in Betrieb sein sollen, dann sorgt der *Controller* dafür, dass dieses durch den *kubelet* auf den Nodes umgesetzt wird.
- **Deployment Controller²:** wird verwendet, um einen *Pod* mit einer gewünschten Anzahl von Replikationen zu betreiben. Diese *Pods* haben keine eindeutigen Identitäten. Das *Deployment* kann die Konfiguration über einen eigenständigen *Pod* und ein *ReplicaSet* festlegen und die Deployment-Strategie vorgeben.
- **DaemonSet Controller³:** stellt sicher, dass alle oder einige Knoten innerhalb des Clusters eine Kopie eines *Pods* ausführen. Es ist der richtige *Controller* für die Aufgabe, einen *Pod* pro Knoten zu implementieren. Sobald Sie die DaemonSet-Spezifikation an den *API-Server* übermitteln, wird auf jedem Knoten nur ein *Pod* geplant.

3.2.4 Control-Plane-Komponente: Scheduler

Der Prozess *kube-scheduler* kennt den Workload ((Liebel, 2019), S. 566) der *Worker-Nodes* und kann die *Deployments* oder *Pods* deswegen bestmöglich auf den *Worker-Nodes* platzieren. Somit kümmert sich der Scheduler-Prozess, um die Verteilung von *Pods* auf den richtigen *Nodes*. Dazu bekommt der *Scheduler* vom *API-Server* die Befehle. Für jeden neu erstellten *Pod* oder andere nicht eingeplante *Pods* wählt der *kube-scheduler* einen optimalen Knoten aus, auf dem sie ausgeführt werden können. Die Container (The Kubernetes Authors, 2021d) in den *Pods* haben unterschiedliche Anforderungen an die Ressourcen. Daher müssen die vorhandenen Knoten nach den spezifischen Anforderungen für die Planung gefiltert werden. In einem Cluster werden die Knoten, die die Planungsanforderungen für einen *Pod* erfüllen, als machbare Knoten bezeichnet. Das *Deployment* realisiert der Scheduler-Prozess mithilfe des Default-Placement-Algorithmus. Dabei sucht der *Scheduler* nach neu erstellten *Pods*, die keinem Knoten zugewiesen sind und führt dann eine Reihe von Funktionen aus, um die geeigneten Knoten zu bewerten, und wählt den Knoten mit der höchsten Bewertung unter den geeigneten Knoten aus, um den *Pod* zu platzieren. Der *Scheduler* benachrichtigt dann den *API-Server* über diese Entscheidung in einem Prozess, der *Binding* genannt wird. Die Auswahl der Knoten des *kube-schedulers* erfolgt durch die Ausführung von zwei Funktionen:

- **Filterung (engl. Filtering):** Der Filterschritt findet die Menge der Knoten, bei denen es möglich ist, den *Pod* einzuplanen. Der Filter *PodFitsResources* prüft zum Beispiel, ob ein Kandidatenknoten über genügend verfügbare Ressourcen verfügt, um die spezifischen Ressourcenanforderungen eines *Pods* zu erfüllen. Als Rückgabe der Filterfunktion wird eine Knotenliste mit allen geeigneten Knoten zurückgegeben. Es gibt in der Regel mehr als einen Knoten. Es gibt auch Fälle wo die Liste leer ist, dann ist der *Pod* nicht planbar und bleibt erstmal in dem Zustand *not scheduled*. Die Filterung erfolgt anhand von Planungsrichtlinien (**engl. Scheduling Policies**), welche

¹<https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>, letzter Zugriff 02.07.2022

²<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>, letzter Zugriff 02.07.2022

³<https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>, letzter Zugriff 02.07.2022

die Voraussagen definieren, ab wann ein *Pod* einem oder mehreren Nodes zugewiesen werden kann. Es können auch Planungsprofile-Plugins (engl. *Scheduling Profiles*) konfiguriert werden, die verschiedene Planungsstufen implementieren wie zum Beispiel *QueueSort*, *Filter* oder *Score*. Dadurch kann der *kube-scheduler* so konfiguriert werden, dass er verschiedene Profile ausführt.

- **Bewertung (engl. Scoring):** im Scoring-Schritt ordnet der *Scheduler* die verbleibenden Knoten, um die am besten geeignete Pod-Platzierung zu wählen. Der *Scheduler* weist jedem Knoten, der die Filterung überstanden hat, eine Punktzahl zu und stützt sich dabei auf die aktiven Bewertungsregeln. Die Regeln werden durch die *Scheduling Policies* gesetzt und mit Prioritäten für die Punktwertung versehen.

Schließlich weist *kube-scheduler* den *Pod* dem Knoten mit der höchsten Punktzahl zu. Wenn es mehr als einen Knoten mit gleicher Punktzahl gibt, wählt *kube-scheduler* einen davon nach dem Zufallsprinzip aus.

3.2.5 Control-Plane-Komponente: Key-Value-Store etcd

Bei *etcd* ((Liebel, 2019), S. 566) handelt es sich um einen *Key-Value-Store*, der den zentralen Speicher der *Control-Planes* darstellt. Der *KV-Store* speichert den Zustand des *API-Servers* persistent und damit auch alle verbundenen Änderung der Ressourcen. Die weiteren Komponenten wie zum Beispiel der Kubelet-Prozess überwachen den *etcd* auf Änderungen, um diese dann clusterweit in den Soll-Zustand umzusetzen. Im *etcd-KV-Store* werden zudem auch Änderungen zu der Netzwerk-Konfiguration der Overlay-Netze abgelegt. Die Abbildung 3.2 zeigt, vereinfacht und stark abstrahiert, das Zusammenspiel der Kubernetes-Clusters-Komponenten mit dem *etcd-KV-Store*.

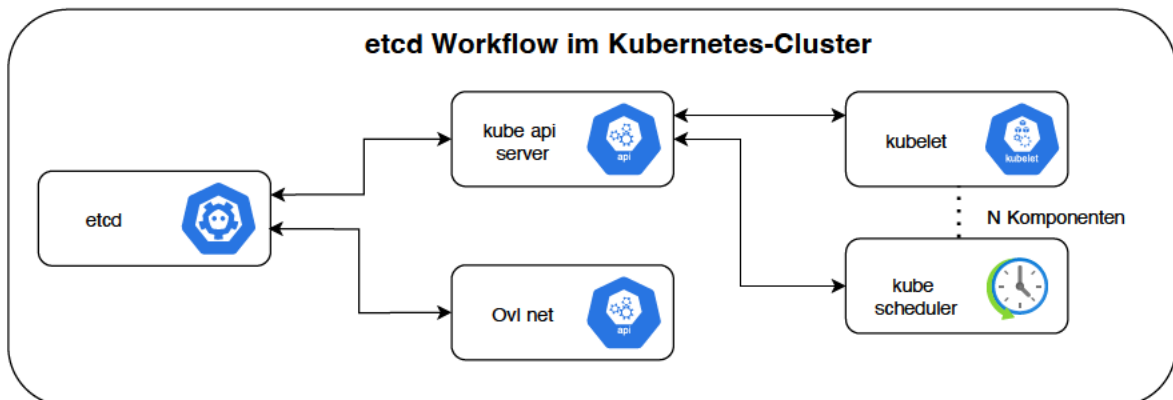


Abbildung 3.2: Kubernetes-Kernkomponenten, Bildquelle: eigene Darstellung in Anlehnung an ((Liebel, 2019), S. 566)

Bei dem *etcd-KV-Store* ((Liebel, 2019), p. 532) selbst handelt es sich um einen in Go¹ geschriebenen hierarchisch verteilte Datenbank, welches als Nebenprodukt von CoreOS entstanden ist. Der *KV-Store* soll die Verwaltung von Daten über mehrere Knotenpunkte ermöglichen. Der *etcd-KV-Store* ist aufgrund seiner einfachen Bedienbarkeit durch *REST*, *JSON* und general-purpose-Remote-Procedure-Calls (gRPC)² API sehr beliebt und deswegen auch zu der Standard-Lösung von Containern-Clustern

¹<https://golang.org>, letzter Zugriff 15.06.2022

²<https://grpc.io>, letzter Zugriff 15.06.2022

geworden. Die *API* lässt sich einfach per *curl*³ über den entsprechenden Pfad bedienen. Seit der *etcd*-Version 3 wird jede *API* Anfrage in einen *gRPC* umgewandelt, welches wiederum auf dem Standard von *HTTP/2* und *Protocol Buffers* aufsetzt. Hierdurch wird die Logik um Funktionen erweitert, welche mehr Möglichkeiten bieten als *REST API Calls*. Der *etcd-KV-Store* ((Liebel, 2019), p. 531-533) basiert auf dem Raft-Konsensus-Algorithmus¹. Der Raft-Konsensus-Algorithmus ist die Basis für die Erstellung von *etcd*-Clustern. Der Konsens im Raft-Konsensus-Algorithmus wird über einen *Elected Leader* gebildet. Ein Server in einem Raft-Cluster kann eine der drei Rollen annehmen:

- **Führer (engl. *Leader*):** kümmert sich um die Log-Replikationen auf den *Followern*. Sendet regelmäßig einen *Heartbeat* an die *Follower* und *Candidates*. Hierdurch werden die Teilnehmer in Kenntnis gesetzt, dass der *Leader* noch aktiv ist.
- **Anhänger (engl. *Follower*):** Die *Follower* sind passiv und reagieren lediglich auf die Anfragen des Anführers oder der Kandidaten. Erhält der *Follower* eine Änderung vom Client, dann wird die Anfrage an den *Leader* weitergeleitet.
- **Kandidat (engl. *Candidate*):** Wird ein *Follower* zum *Candidate*, dann bittet dieser die übrigen Mitglieder des Clusters um Stimmen, damit er zum *Leader* gewählt wird. Ein *Follower* wird zum *Candidate*, wenn er keinen *Heartbeat* mehr vom *Leader* erhält und der *Election-Timer* in einen Timeout läuft.

Der *etcd KV-Store* stellt somit die zentrale und somit auch eine sehr kritische Komponente, ohne die ein Cluster-Betrieb nicht möglich ist. Deswegen sollte der *etcd-KV-Store* bei einem produktiven K8s-Cluster in einem Cloud-native aufgesetzt werden.

3.2.6 Control-Plane- und Workernode-Komponente: kubelet

Der Kubelet-Dienst ((Liebel, 2019), S. 566-567) stellt die Verbindung zwischen der Docker-Engine und dem *API-Server* dar. Es handelt sich primär, um einen Knotenagenten, der auf jedem Knoten läuft. Der Agent kann den Knoten beim *API-server* registrieren, indem es einen den Hostnamen als Parameter verwendet. Der *kubelet* erhält vom *API-Server* die Information, welche Ressourcen auf den *Worker-Node* ausgerollt, modifiziert oder gelöscht werden sollen. Nach einer Änderung der Ressourcen überwacht der Kubelet-Dienst den Zustand der Ressourcen. In einem Pod-basierten Kubernetes-Clusters kümmert sich der Dienst, um komplett alles, was den Cluster anbetrifft und sorgt dafür, dass alle benötigten Komponenten als *Pods* ausgerollt werden. Durch den Dienst wird ein vollautomatisiertes *Deployment* von *Pods*, *ReplicaSets*, *Deployments*, etc. per *kubectrl run* oder Manifesten ermöglicht. Ein Manifest stellt zum Beispiel ein *PodSpec* dar, welches als *YAML*² - oder *JSON-File*³ vorliegt. Das Manifest kann an den *API-Server* übergeben werden. Der Kubelet-Dienst nimmt eine Reihe von *PodSpecs* in erster Linie vom *API-Server* entgegen und stellt sicher, dass die in diesen *PodSpecs* beschriebenen Container laufen und gesund sind. Der Kubelet-Dienst verwaltet keine Container, die nicht von Kubernetes erstellt wurden.

³<https://curl.se>, letzter Zugriff 15.06.2022

¹<https://raft.github.io>, letzter Zugriff 15.06.2022

²<https://www.redhat.com/de/topics/automation/what-is-yaml>, letzter Zugriff 05.08.2022

³<https://www.oracle.com/de/database/what-is-json/>, letzter Zugriff 05.08.2022

3.2.7 Control-Plane- und Workernode-Komponente: Container-Engine

Die *Container-Engine* ((Liebel, 2019), S. 567) wie Docker kümmert sich nur noch um Kernaufgaben wie das Hosten des *Images* im jeweiligen *Storage-Backend* auf den *Nodes*. Außerdem kümmert sich die *Container-Engine* um das Abfeuern *docker pull and run* der Container-Instanzen. Seit Kubernetes 1.19 hat sich die Standard Container-Engine von *docker* zu *containerd*¹ geändert. Mit der Docker-Engine als Container-Laufzeitumgebung ist alles gestartet.

3.2.8 Control-Plane- und Workernode-Komponente: kube-proxy

Auf jedem *Node* läuft eine Netzwerk-Kube-Proxy-Instanz ((Liebel, 2019), S. 568-569) und kümmert sich primär um zwei Aufgaben. Die erste Aufgabe ist die interne und clusterweite Bereitstellung von *Services*, worüber die *Pods* untereinander oder namespaceübergreifend kommunizieren können. Die zweite Aufgabe ist die Annahme und Weiterleitung von eingehenden Request, die von außerhalb des Clusters kommen. Die Kube-Proxy-Instanz kann als ein primitiver *Layer-4-TCP-Loadbalancer* fungieren. Die Instanz leitet eine von außen gestellte Anfrage zu der zuständigen K8s-Ressource durch Mapping des der externen IP-Adresse auf einen internen Service, der wiederum auf einen *Pod* als Endpunkt zeigt. Der *kube-proxy* überwacht außerdem die *API* des *K8s-Control-Planes*, um Änderungen in den *Services* in seinen lokalen iptables-Rulesets entsprechend anzupassen. Dadurch, dass auf jedem *Node* eine Kube-Proxy-Instanz läuft, kann die gesamte Auflösung von clusterweiten und gültigen erreichbaren *Services* umgesetzt werden. Da der *kube-proxy* als ein sehr einfacher L4-Loadbalancer fungiert, fehlen diesem wichtige Implementierungsfunktionen wie zum Beispiel *Retries* bei Fehlern, um einen Betrieb von *HTTP-Sessions* effektiv zu verwalten. Deswegen wird für HTTP-Frontendbasierte Anwendungen auf *Ingresses* und *Ingress-Controller* zurückgegriffen, welche die dahinter liegenden Dienste verschlüsselt und namensbasiert mit einem *HTTP-Sessions* verwalten. Für das letzte Stück der Strecke von dem *Ingress* zu dem Service wird dennoch der *kube-proxy* benötigt.

3.2.9 Networking in Kubernetes und CNI

In einem Kubernetes-Cluster ((Liebel, 2019), S. 569) wird im Vergleich zu Docker eine flache Netzwerkstruktur verwendet. Bei Docker kommt in der Regel Network-Address-Translation (NAT) und *Port-Forwarding* oder *Port-Mapping* vom Container-Port zum Host-Port zum Einsatz. Bei Kubernetes wird kein NAT für Container, die sich in einem *Pod* befinden, benötigt, weil diese das Routing des *Nodes* nutzen. Da IP-Adressen dynamisch und somit schwer ansprechbar sind, werden *Services* dazwischen geschaltet. Diese ermöglichen die Kommunikation zwischen *Pods* oder Containern über Service-Namen, welche wie interne DNS-Namen verwendet werden. Um dies zu ermöglichen, verfügt jede Kubelet-Instanz in einem Kubernetes-Cluster über ein Default-Netzwerk-Plugin und somit auch ein Default-Netz. Darüber wird die initiale Kommunikation innerhalb eines Kubernetes-Cluster eingeleitet und abgewickelt. Dafür prüft die Kubelet-Instanz beim Starten auf einem *Node*, ob weitere Container-Network-Interface (CNI) Plugins vorhanden sind. Das *CNI-Plugin* wie Weave² übernimmt die Aufgabe ein virtuelles Interface zu erzeugen und es mit dem darunter liegenden Netzwerk des *Nodes* und der Netzwerkkarte zu verbinden. Außerdem setzt das CNI-Plugin IP-Adresse und Routing-Regeln für *Pods* fest. Anschließend wird dann die virtuelle Netzwerkkarte des *Nodes* mit dem *NET-Namespace* des *Pods* verbunden. Das *CNI-Plugin* reicht nicht komplett für das Routing aus, deswegen kommt *kube-proxy* zum Einsatz und übernimmt das Routing zum *Pod* auf der letzten Ebene durch Festlegen

¹<https://containerd.io>, letzter Zugriff 04.08.2022

²<https://www.weave.works>, letzter Zugriff 17.06.2022

von Regeln über *iptables*. Die Kommunikation ((Liebel, 2019), S. 572) in einem Kubernetes-Cluster kann in vier verschiedene Bereiche eingeteilt werden.

- **Container-zu-Container-Kommunikation:** Container, die sich in einem gemeinsamen *Pod* befinden, teilen sich den Netzwerk-Namespaces. Deswegen können sie direkt innerhalb des *Namespaces* direkt miteinander kommunizieren.
- **Pod-zu-Pod-Kommunikation:** *Pods* erhalten eine IP-Adresse aus einem Subnetz durch das *CNI-Plugin* innerhalb des gleichen *Namespaces*. Dadurch können *Pods* mit anderen *Pods* über Routing-Tabellen kommunizieren als wären es virtuelle Maschinen.
- **Pod-zu-Service-Kommunikation:** ein Service ist vergleichbar mit einer Floating-IP-Adresse. Ein *Pod* erhält über einen Service einen Endpunkt zugeordnet. Der Endpunkt erweitert die internal-IP-Adresse des *Pods*, um eine external IP-Adresse. Dadurch ist die Kommunikation zwischen *Pods* namespaceübergreifend möglich.
- **Externe-zu-interner-Kommunikation:** der Zugriff auf einen Cluster, internen Service von außen wird über externe Loadbalancer oder einen *Ingress* abgebildet. Der Service erhält eine Floating-IP-Adresse und sobald der Request diese IP-Adresse anspricht, geht der Request an einen *Worker-Node*, welcher dank des Kube-Proxy-Dienstes an den richtigen zuständigen Service weitergeleitet wird.

3.2.10 Control-Plane- und Workernode-Komponente: kube-dns

Um die *Services* oder DNS-Namen (The Kubernetes Authors, 2022b) in IP-Adresse aufzulösen, wird ein K8s-interner DNS-Server eingesetzt. Diese Aufgabe übernimmt der Dienst *kube-dns*. Das Kubernetes-Cluster erstellt DNS-Einträge für Dienste und *Pods*. Anschließend können die Dienste mit einheitlichen DNS-Namen anstelle von IP-Adressen kontaktiert werden. Dafür werden *DNS-Pods* auf dem Cluster so konfiguriert, dass die mit der Kubelet-Instanz auf dem *Node* kommunizieren. Die Kubelet-Instanz teilt einzelnen Containern die IP-Adresse zu dem internen DNS-Namen. Dafür wird jedem im Cluster definierten Dienst, einschließlich des *DNS-Servers* selbst, ein DNS-Name zugewiesen. Standardmäßig enthält die DNS-Suchliste eines *Client-Pods* den eigenen Namensraum des *Pod* und die Standarddomäne des Clusters. Eine DNS-Abfrage kann je nach dem Namensraum des *Pod*, der sie stellt, unterschiedliche Ergebnisse liefern. DNS-Abfragen, die keinen Namespace angeben, sind auf den Namespace des *Pod* beschränkt. Die *Pod* können auf Dienste in anderen *Namespaces* zugreifen, indem sie DNS-Abfragen verwenden.

Anhand des folgenden Beispiels wird das Zusammenspiel zwischen Kubernetes-Networking, Kube-Proxy und Kube-DNS erläutert. Die Abbildung 3.3 zeigt die interne DNS-Auflösung der *Pod* und den dafür zuständigen DNS-Server. Bei 1. wird für beide *Namespaces* Test und Prod der DNS-Server abgefragt. Beide *Pod* zeigen auf die IP-Adresse 10.96.0.10, welche zu einem Service gehört (2.). Die Endpunkte (3.) des *Services* zeigen dabei auf mehrere IP-Adressen, da mehrere Instanzen vom DNS-Server vorhanden sind. Die IP-Adressen (4.) gehören zu den *Core-DNS-Pods*¹, die für die internen Clusterauflösungen der DNS-Namen zuständig sind. Der Kube-Proxy-Dienst übernimmt die Verantwortung zwischen Punkt 2. und 3. Er stellt die Verbindung zwischen dem Service und den Endpunkten, den IP-Adressen der *Pods* her.

¹<https://coredns.io>, letzter Zugriff 17.06.2022

```

OSX ~ /kubernetes-service-catalog [?] main 0.0.1 k run nginx-test --image nginx -n kube-ns-prod
pod/nginx-test created
OSX ~ /kubernetes-service-catalog [?] main 0.0.1 k exec -it -n kube-ns-prod nginx-test -- cat /etc/resolv.conf
search kube-ns-prod.svc.cluster.local svc.cluster.local cluster.local dns.podman Fritz.Box
nameserver 10.96.0.10 1
options ndots:5
OSX ~ /kubernetes-service-catalog [?] main 0.0.1 k exec -it -n kube-ns-test nginx-test -- cat /etc/resolv.conf
search kube-ns-test.svc.cluster.local svc.cluster.local cluster.local dns.podman Fritz.Box
nameserver 10.96.0.10 1
options ndots:5
OSX ~ /kubernetes-service-catalog [?] main 0.0.1 kg svc -n kube-system
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kube-dns ClusterIP 10.96.0.10 <none> 53/UDP,53/TCP,9153/TCP 24m50s
OSX ~ /kubernetes-service-catalog [?] main 0.0.1 kg ep -n kube-system
NAME ENDPOINTS AGE
kube-dns 10.244.0.3:53,10.244.0.4:53,10.244.0.3:53 4m43s
OSX ~ /kubernetes-service-catalog [?] main 0.0.1 kg pods -n kube-system -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
coredns-64897985d-2kmtj 1/1 Running 0 5m9s 10.244.0.4 kind-control-plane <none> <none>
coredns-64897985d-9f2jj 1/1 Running 0 5m9s 10.244.0.3 kind-control-plane <none> <none>
etcd-kind-control-plane 1/1 Running 0 5m25s 10.89.0.2 kind-control-plane <none> <none>
kindnet-lzltg 1/1 Running 0 5m9s 10.89.0.2 kind-control-plane <none> <none>
kube-apiserver-kind-control-plane 1/1 Running 0 5m26s 10.89.0.2 kind-control-plane <none> <none>
kube-controller-manager-kind-control-plane 1/1 Running 0 5m26s 10.89.0.2 kind-control-plane <none> <none>
kube-proxy-gr9mf 1/1 Running 0 5m9s 10.89.0.2 kind-control-plane <none> <none>
kube-scheduler-kind-control-plane 1/1 Running 0 5m27s 10.89.0.2 kind-control-plane <none> <none>

```

Abbildung 3.3: Zusammenspiel Kubernetes-Networking, Kube-DNS und Kube-Proxy, Bildquelle: eigene Darstellung

Das nächste Beispiel (siehe Abbildung 3.4) zeigt, wie die Kommunikation zwischen zwei *Pod*, die in unterschiedlichen *Namespaces* liegen, anhand der internen Cluster-DNS-Auflösung funktioniert. Es wurde ein Service (1.) zu dem *Pod* `nginx-test` in dem *Namespace* `kube-ns` erstellt. Die dazugehörige IP-Adresse (2.) wird über den Endpunkt abgefragt. Zur Sicherheit wird nochmal die IP-Adresse (3.) des *Pod* direkt mit dem Endpunkt abgeglichen. Anschließend erfolgt ein Aufruf des *Services* (4.) des *Pod* `nginx-test` in dem *Namespace* `kube-ns-test` über den *Pod* `nginx-test` aus dem *Namespace* `kube-ns-prod` über den DNS-Namen `nginx-service-test.kube-ns-test`. Der DNS-Name setzt sich aus dem `svc-name.namespace` zusammen. Der DNS-Name liefert die IP-Adresse des *Pod* `nginx-test` aus dem *Namespace* `kube-ns-test` wieder. Diese Umsetzung ermöglicht, dass die Kommunikation in einem Kubernetes-Cluster zwischen *Pod*, die in unterschiedlichen Namensräumen liegen, stattfinden kann.

```

OSX ~ /kubernetes-service-catalog [?] main 0.0.1 ? kg svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
nginx-service-test 1 ClusterIP 10.96.10.193 <none> 80/TCP 4m8s
OSX ~ /kubernetes-service-catalog [?] main 0.0.1 ? kg ep
NAME ENDPOINTS AGE
nginx-service-test 10.244.0.7:80 2m10s
OSX ~ /kubernetes-service-catalog [?] main 0.0.1 ? kg pod -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
nginx-test 1/1 Running 0 4m16s 10.244.0.7 3 kind-control-plane <none> <none>
OSX ~ /kubernetes-service-catalog [?] main 0.0.1 ? k exec -it -n kube-ns-prod nginx-test -- curl -v -vvv nginx-service-test.kube-ns-test
* Trying 10.96.10.193:80... 4
* Connected to nginx-service-test.kube-ns-test (10.96.10.193) port 80 (#0)
> GET / HTTP/1.1

```

Abbildung 3.4: Kommunikation zwischen *Pod* in unterschiedlichen Namespaces, Bildquelle: eigene Darstellung

3.3 Managed Kubernetes - Azure-Kubernetes-Service

Als offene Plattform ermöglicht Kubernetes es (Microsoft, 2022a) Anwendungen mit bevorzugten Programmiersprachen, Betriebssystemen, Bibliotheken oder Messagingbussen zu erstellen. Die Planung und Bereitstellung von Releases kann in Kubernetes über *CI/CD-Tools* integriert werden. Bei der herkömmlichen Lösung von Kubernetes verwaltet der Administrator alle Komponenten wie *Control-Planes*- und *Worker-Nodes* sowie die darauf laufende Dienste. Der Administrator ist stark damit beschäftigt, die Infrastruktur und deren Bestandteile wie Netzwerke, Storage und Rechenleistung zu verwalten. Die Verwaltung erfordert nicht nur viel Zeit, sondern auch ein tiefes Wissen über das Zusammenspiel der einzelnen Infrastrukturkomponenten und der darunter liegenden Hardware. Eine

fehlerhafte Konfiguration kann zum Ausfall einzelner Komponenten führen und somit auch Bestandteile eines Kubernetes-Clusters betreffen. An dieser Stelle gibt es Kubernetes als einen verwalteten Service, welcher von unterschiedlichem Cloud-Service Provider wie Microsoft zur Verfügung gestellt wird. Azure-Kubernetes-Service stellt einen solchen Service dar. Die Komplexität wie die Bereitstellung und somit auch die wichtigen Verwaltungsaufgaben liegen an der Stelle bei Microsoft. Der Benutzer konsumiert den Service und ist für die Konfiguration wie zum Beispiel die Netzwerk-Segmentierung oder die Anzahl der Knoten zuständig. Die Verwaltungsaufgaben reduzieren sich dadurch stark und ermöglichen es den System-Ingenieuren, den Fokus anders zu setzen. Die Azure-Plattform verwaltet die Azure-Kubernetes-Service (AKS) Steuerungsebene (*Control-Plane-Nodes*) und der Kunde zahlt nur für den Workload der Anwendung, der auf den *AKS-Worker-Nodes* läuft.

3.3.1 Cloud-Controller-Manager

Mit Cloud-Infrastrukturtechnologien (The Kubernetes Authors, 2022a) kann Kubernetes in öffentlichen, privaten und hybriden Clouds ausgeführt werden. Kubernetes setzt auf eine automatisierte, API-gesteuerte Infrastruktur ohne enge Kopplung zwischen den Komponenten. Der *Cloud-Controller-Manager* ist eine Komponente der Kubernetes-Kontrollebene, die Cloud-spezifische Steuerungslogik einbettet. Mit dem *Cloud-Controller-Manager* können Cluster mit der *API* des jeweiligen Cloud-Service-Anbieters wie Azure von Microsoft verbunden werden. Die Komponenten, die mit dieser Cloud-Plattform interagieren, können von den Komponenten getrennt werden, die nur mit Ihrem Cluster interagieren. Durch die Entkopplung der Interoperabilitätslogik zwischen Kubernetes und der zugrundeliegenden Cloud-Infrastruktur ermöglicht die *Cloud-Controller-Manager*-Komponente Cloud-Anbietern, Funktionen in einem anderen Tempo als das Kubernetes-Hauptprojekt zu veröffentlichen. Die Interoperabilitätslogik ermöglicht dynamisches Ensemble von vernetzten Knoten. Der *Cloud-Controller-Manager* ist über einen Plugin-Mechanismus aufgebaut, der es verschiedenen Cloud-Anbietern ermöglicht, die Plattformen des Anwenders in Kubernetes zu integrieren. Der *Cloud-Controller-Manager* ist also eine Komponente, die erst interessant wird, wenn Cloud-Infrastrukturtechnologien wie bei dem Dienst Azure-Kubernetes-Service von Microsoft zum Einsatz kommen.

3.3.2 Architektur

Ein Azure Kubernetes-Cluster (Microsoft, 2022a) ist im Wesentlichen in drei Infrastruktur-Komponenten unterteilt:

- **Steuerungsebene (engl. *Control-Planes*):** stellt die grundlegenden Dienste der Orchestrierungsplattform Kubernetes dar und stellt die wichtigsten Orchestrierungsfunktionen bereit, um den Workload für Anwendungen auf den *Nodes* zu verteilen. Die Steuerungsebene befindet sich in der Region, in der auch das Cluster erstellt wird. Die Steuerungsebene bekommt einen dedizierten *API-Server*, über den mit der Steuerungsebene kommuniziert wird. Die Kommunikation und Interaktion zwischen Steuerungsebene und den Knoten erfolgt über die *Kubernetes-API*. Es gibt keinen direkten Zugriff auf die Steuerungsebene. Die Interaktion mit der Steuerungsebene erfolgt über Azure-CLI oder das Azure-Portal.
- **Knoten (engl. *Worker-Nodes*):** auf diesen läuft der hauptsächliche Workload der Anwendungen. Die Größe eines Knotens wird über die VM definiert. Die Größe wird durch CPUs, Arbeitsspeicher, Typ des verfügbaren Speichers (Hard-Disk-Drive (HDD) oder Solid-State-Drive (SSD)) bestimmt. Als Betriebssystem wird ein *Image* auf Basis von Ubuntu oder Windows Server 2019 bereitgestellt.

- **Knotenpools:** (engl. *Set of Worker-Nodes*): gleich konfigurierte Knoten werden in Knotenpools gruppiert. Ein AKS-Cluster enthält mindestens einen Knotenpool mit mindestens einem Knoten. Die Knotenpools ermöglichen den Betrieb unterschiedlicher Anwendungen und mit unterschiedlichen Anforderungen an die Ressourcen oder des Betriebssystems einer VM.

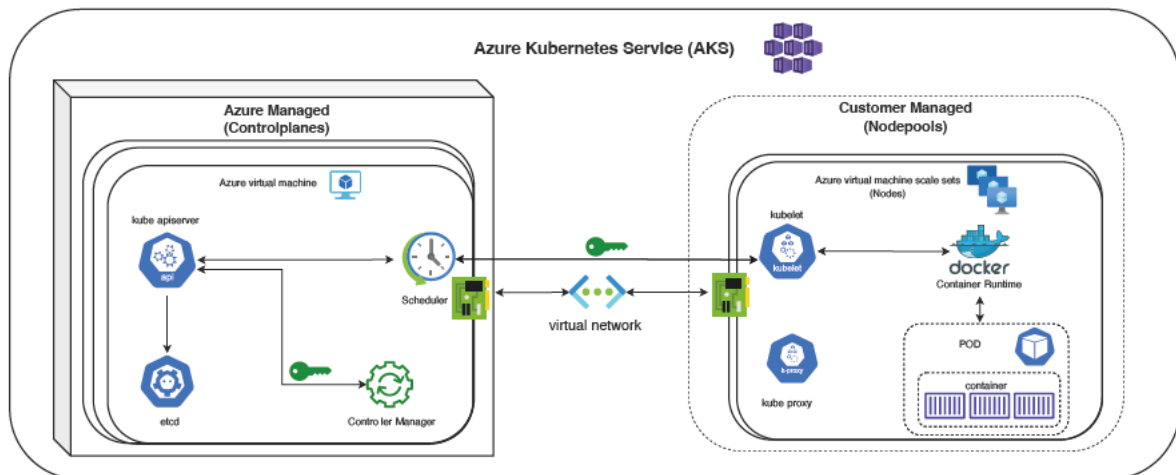


Abbildung 3.5: Azure Kubernetes Service Kernkonzept, Bildquelle: eigene Darstellung in Anlehnung an (Microsoft, 2022a)

Die Abbildung 3.5 veranschaulicht die Verantwortlichkeit des Benutzers beim Nutzen des verwalteten *Cloud-Services* AKS. Der Fokus und die Verantwortung des Benutzers liegt hauptsächlich beim Customer-Managed-Part.

Bei der Verwendung und der richtigen Konfiguration der Knotenpools des Azure-Kubernetes-Service muss Microsoft dem Kunden einen stabilen Betrieb gewährleisten. Um dies zu ermöglichen, werden Knotenressourcen von den gesamten Ressourcen der VM abgezogen. Dies führt zu Abweichung von den provisionierten Ressourcen einer VM-Klasse zu den tatsächlich verfügbaren Ressourcen.

3.4 Stateless vs Statefull

Ein wichtiges Kriterium, das vor dem produktiven Einsatz einer neuen Anwendung zu berücksichtigen ist, ist die der Anwendung zugrunde liegende Architektur. In diesem Zusammenhang wird häufig der Begriff *Stateless* für zustandslose oder *Statefull* für zustandsabhängige Anwendungen verwendet. Beide Arten haben ihre eigenen Vor- und Nachteile, welche im Folgenden genauer erläutert werden.

Stateless

In einem Kubernetes-Cluster (The Kubernetes Authors, 2022c) ist einer zustandslosen Anwendung kein persistenter Speicher oder Datenträger zugeordnet. Verschiedene *Pods* im gesamten Cluster können unabhängig voneinander arbeiten, wobei mehrere Anfragen gleichzeitig an sie gerichtet werden. Wenn etwas schiefgeht, können Sie die Anwendung einfach neu starten, und sie wird ohne lange Ausfallzeiten in den Ausgangszustand zurückversetzt. Der Nachteil ist, jegliche Änderungen, die nicht mit der Anwendung ausgeliefert werden und über den *Pod* gemacht worden sind, sind nachdem Neustart weg. Deswegen eignen sich Frontend-Anwendungen sehr gut als Stateless-Anwendungen, da die Daten auf einer Webseite meistens nachgelagert aus einer Datenbank geladen werden und nicht persistent im Frontend eingebettet sind.

Statefull

Bei *StatefulSets* ((Liebel, 2019), S. 1046-1048) sind für die Verwendung mit zustandsabhängigen Anwendungen und verteilten Systemen vorgesehen. Die Verwaltung zustandsbehafteter Anwendungen und verteilter Systeme auf Kubernetes ist jedoch ein umfangreiches und komplexes Thema. Die zustandsbehafteten Dienste müssen sich um eine Vielzahl von Randfällen und Problemen kümmern. Ein Pod wird von mindestens einem *Volume* begleitet und wenn die Daten in diesem *Volume* beschädigt sind, bleibt dies auch dann bestehen, wenn der gesamte Cluster neu gestartet wird. Wenn beispielsweise eine Datenbank in einem Kubernetes-Cluster betrieben wird, dann müssen alle Pod über ein lokales *Volume* zum Speichern der Datenbank verfügen. Alle Daten müssen synchronisiert sein. Wenn also jemand einen Eintrag in der Datenbank ändert, was auf Pod A geschah und eine Leseanforderung auf Pod B kommt, um die geänderten Daten zu sehen, dann muss Pod B die neuesten Daten anzeigen oder eine Fehlermeldung ausgeben. Hier kommt dann das CAP-Theorem¹ ins Spiel und legt die Regeln fest. In der Regel sind *Nodes* und gesamte Cluster so gestaltet, dass die gelöscht oder rotiert werden können. Wenn der Zustand eines Pod auf einem *Host-Node* gespeichert wird, geht dieser verloren. Für solche Fälle werden Offsite-Speicherlösungen wie Network-File-Share (NFS) verwendet.

3.5 Operatoren

Für das *Lifecycle-Management* (2022 Red Hat, Inc., 2020) von Anwendungen werden bei Kubernetes-Anwendungen Operatoren verwendet. Bei Operatoren handelt es sich nicht, wie aus der Mathematik bekannt, um keine mathematische Vorschrift, sondern um Methoden zur Paketierung, Bereitstellung und Verwaltung einer oder mehrere Kubernetes-Anwendungen. Ein *Operator* stellt einen anwendungsspezifischer *Controller* dar. Dieser erweitert die Funktionalität der *Kubernetes-API*, um Instanzen komplexer Anwendungen für einen Kubernetes-Nutzer zu erstellen und zu verwalten. Dabei werden allgemeine Kubernetes-Konzepte für die Verwaltung von Ressourcen und Controllern verwendet. Die Überwachung der *Controller* wird in Kubernetes in Kontrollschleifen implementiert, den Ist-Zustand mit dem Soll-Zustand im Cluster wiederholt miteinander vergleichen. Sollte der Zustand nicht übereinstimmen, dann ergreift der *Controller* Maßnahmen, um das Problem zu beheben. Ein *Operator* stellt somit einen benutzerdefinierten *Kubernetes-Controller* dar. Dieser ist für die Verwaltung von Custom-Resources (CR) von Anwendungen und der dazugehörigen Komponenten zuständig. Der Benutzer legt der in der CR allgemeine Anweisungen und Konfigurationen wie zum Beispiel die Mindestanzahl an vorhanden *Replicas* einer Anwendung. Anschließend übersetzt der Kubernetes *Operator* die allgemeinen Anweisungen basierend auf den in seiner Logik eingebetteten Implementierung. Somit stellen *CRs* einen API-Erweiterungsmechanismus in Kubernetes dar. Ein Benutzer schreibt nicht direkt eine CR, sondern definiert eine CRD. Diese definiert wiederum eine CR und listet alle für den Nutzer des Operators verfügbaren Konfigurationsmöglichkeiten. Somit werden mithilfe von Kubernetes Operatoren neue Objekttypen in Form von *CRDs* bereitgestellt. Dies ermöglicht dem Benutzer die Verwaltung von den neuen Objekttypen über eine gewohnte Interaktion via *kubectl* und die *Kubernetes-API*.

¹<https://wikis.gm.fh-koeln.de/Datenbanken/CAP>, letzter Zugriff 15.06.2022

Das Kapitel setzt sich mit der Vertiefung von CI/CD auseinander. Dabei liegt der Schwerpunkt auf den verschiedenen Kulturen DevOps und GitOps, die einen starken Anteil zu Entwicklung von CI/CD beigetragen haben und weiterhin beitragen werden. Des Weiteren werden die eingesetzten *Tools* vorgestellt, die für die Realisierung einer möglichen CI/CD Lösung oder einer GitOps-Lösung vonnöten sind.

4.1 Continuous Integration, Continuous Delivery und Continuous Deployment

Angesichts des zunehmenden Wettbewerbs (Shahin et al., 2017) auf dem Softwaremarkt widmen Unternehmen der Entwicklung und Bereitstellung qualitativ hochwertiger Software in einem wesentlich schnelleren Tempo erhebliche Aufmerksamkeit und stellen entsprechende Ressourcen bereit. Continuous-Integratio (CI), Continuous-Delivery (CDE) und Continuous-Deploymen (CD), werden als kontinuierliche Praktiken bezeichnet. Es sind einige der Praktiken, die Unternehmen dabei helfen sollen, ihre Entwicklung und Bereitstellung von Softwarefunktionen zu beschleunigen, ohne dabei Kompromisse bei der Qualität einzugehen. Während CI die mehrfache Integration von laufenden Arbeiten pro Tag vorsieht, geht es bei CDE und CD um die Fähigkeit, Werte schnell und zuverlässig an die Kunden zu liefern, indem so viel Automatisierungsunterstützung wie möglich geboten wird. Es wird erwartet, dass kontinuierliche Praktiken mehrere Vorteile bieten. Einige Vorteile sind häufiges und schnelleres Feedback aus dem Softwareentwicklungsprozess von dem Kunden, häufige und zuverlässige Releases, die zu einer verbesserten Kundenzufriedenheit und Produktqualität führen. Durch CD wird die Verbindung zwischen Entwicklungs- und Betriebsteams gestärkt und manuelle Aufgaben können eliminiert werden. Eine wachsende Zahl von Fallbeispielen aus der Industrie zeigt, dass die kontinuierlichen Praktiken in der Softwareentwicklung in verschiedenen Bereichen und Größenordnungen von Unternehmen Einzug halten. Gleichzeitig ist die Einführung kontinuierlicher Praktiken keine triviale Aufgabe, da die organisatorischen Prozesse, Praktiken und Werkzeuge möglicherweise nicht bereit sind, die hochkomplexe und anspruchsvolle Natur dieser Praktiken zu unterstützen. Um die Prozesskette von dem Implementieren der Software, bis hin zu ihrer Integration und der automatischen Auslieferung an den Endkunden oder ein Endsystem zu verstehen, werden im Folgenden die Prozesse einzeln genauer betrachtet. Des Weiteren werden bei den Prozessen beispielhafte *Tools*, die für notwendige Erfüllung der Aufgabe in dem jeweiligen Schritt exemplarisch aufgeführt.

Continuous Integration:

Die kontinuierliche Integration (Shahin et al., 2017) ist eine in der Softwareentwicklungsbranche weit verbreitete Entwicklungspraxis, bei der die Mitglieder eines Teams die Entwicklungsarbeit, zum Beispiel den Quellcodestand häufig (zum Beispiel mehrmals am selben Tag) integrieren und zusammenführen. Dadurch ermöglicht es Softwareunternehmen kürzere und häufigere Veröffentlichungszyklen zu liefern. Dies führt zu einer Erhöhung der Softwarequalität und der Produktivität der Teams. Diese Praxis ((Liebel, 2019), S. 62) umfasst die automatisierte Erstellung und Prüfung von Software. Trotzdem wird CI im einfachsten Fall als semiautomatisiertes Verfahren betrachte, weil es sich erstmal, um die Integration des neuen Codes handelt. Die volle Automatisierung wird durch Tests einer Software während und nach ihrem Build-Prozess in der *CI-Stage* erreicht. Ein oder mehrere Entwickler

pflügen neuen *Code* im lokalen *Git-Repository* ein, der dann anschließend neu gepushte *Code* wird in einem *Git-Repository* automatisiert auf Korrektheit überprüft. Tritt ein Fehler auf, dann erhält der Entwickler direkt ein Feedback. Dieser behebt dann Fehler und pusht eine neue Version hoch, die wieder erneut getestet wird. Werden nun alle Tests erfolgreich durchlaufen, dann wird beispielsweise ein Build-Prozess gestartet. Dies hängt von der implementierten Logik der Pipelines und Branch-Logik in der verwendeten Git-Versionsverwaltung Software ab. Wenn dabei ebenfalls alles zufriedenstellend funktioniert, dann wird das Softwareprodukt *CD-Pipeline* weitergegeben und stößt dort weitere Tests an. Somit umfasst die *CI-Pipeline* (siehe Abbildung 4.1) das pushen und evaluieren des Codes mithilfe eines Versionsverwaltung-Tools wie git¹. Des Weiteren umfasst die *CI-Pipeline* den Build-Prozess, um zum Beispiel mithilfe von maven² ein Java³-Projekt zu bauen, dann durch die Einbindung von JUnit⁴ zu Testen und anschließend mithilfe von *Tools* wie Jenkins⁵ an CD zu übergeben.

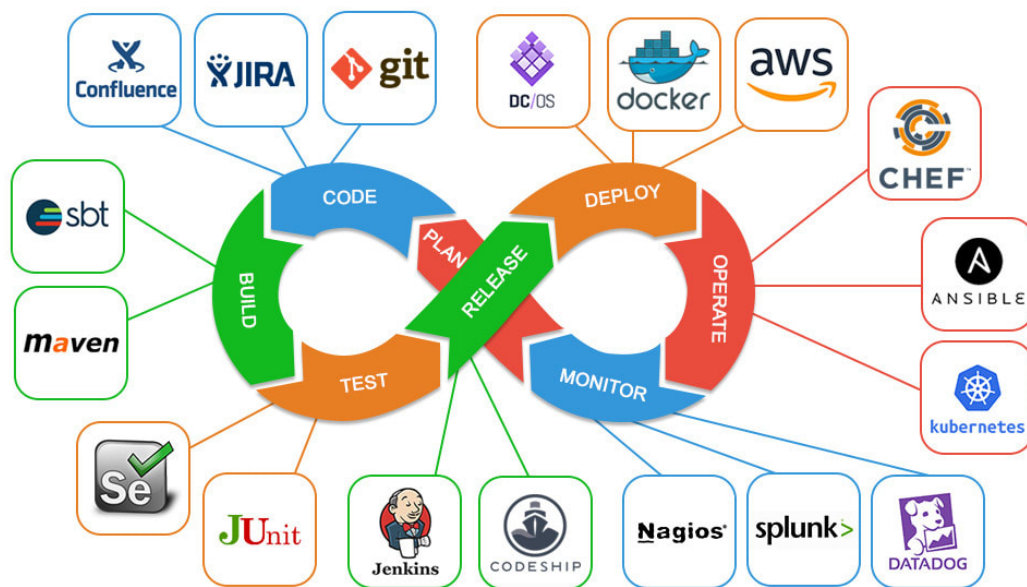


Abbildung 4.1: CI/CD, Bildquelle: <https://www.fiverr.com/momassarwi/do-devops-support-aws-azure-kubernetes>, letzter Zugriff 06.08.2022

Continuous Delivery:

Mit Continuous Delivery (Shahin et al., 2017) soll sichergestellt werden, dass eine Anwendung immer produktionsbereit ist, nachdem sie automatisierte Tests und Qualitätsprüfungen erfolgreich durchlaufen hat. Dabei nutzt CDE eine Reihe von Praktiken, wie zum Beispiel CI und Deployment-Automatisierung, um Software automatisch in einer produktionsähnlichen (*Integration-Stage*) Umgebung bereitzustellen. Diese Praxis bietet mehrere Vorteile, wie zum Beispiel ein geringeres Bereitstellungsrisiko, niedrigere Kosten und schnelleres Benutzerfeedback. Die Abbildung 4.1 zeigt, dass eine kontinuierliche Bereitstellung eine kontinuierliche Integration voraussetzt. Somit setzt CDE ((Liebel, 2019), S. 62-63) die CI fort und ergänzt sie um weitere Elemente im Bereich der Commit und Build-

¹<https://git-scm.com>, letzter Zugriff am 15.07.2022
²<https://maven.apache.org>, letzter Zugriff am 15.07.2022
³<https://www.java.com>, letzter Zugriff am 15.07.2022
⁴<https://junit.org/junit5/>, letzter Zugriff am 15.07.2022
⁵<https://www.jenkins.io>, letzter Zugriff am 15.07.2022

Prozesse, fügt Post-Deployment-Tests hinzu und kümmert sich zusätzlich um das Rollout der validierten Systeme. Dies passiert alles möglichst automatisiert und wird in kleine *Release-Chunks* unterteilt. Die *CDE-Pipeline* erweitert somit die *CI-Pipeline*, um das containerisieren einer Anwendung mithilfe von Docker und die Bereitstellung des Containers auf einer Orchestrierungsplattform wie Kubernetes in einer Produktions ähnlichen Umgebung. Die *CDE-Pipeline* rollt die Änderung nicht automatisiert in eine produktive Umgebung aus. Um die Bereitstellung kümmert sich die *CD-Pipeline*.

Continuous Deployment:

Die Praxis der kontinuierlichen Bereitstellung (Shahin et al., 2017) geht noch einen Schritt weiter und stellt die Anwendung automatisch und kontinuierlich in der Produktions- oder Kundenumgebung bereit. In akademischen und industriellen Kreisen wird heftig über die Definition und Unterscheidung zwischen *Continuous Deployment* und *Continuous Delivery* diskutiert. Was *Continuous Deployment* von *Continuous Delivery* unterscheidet, ist die Auslieferung einer Software in eine Produktionsumgebung des Kunden. Das Ziel der *Continuous-Deployment-Praxis* ist es, jede Änderung automatisch und kontinuierlich in die Produktionsumgebung zu übertragen. Es ist wichtig zu beachten, dass die *CD-Praxis* die *CDE-Praxis* impliziert, aber nicht umgekehrt. Während das endgültige *Deployment* bei *CDE* ein manueller Schritt ist, sollte es bei *CD* keine manuellen Schritte geben. Sobald die Entwickler eine Änderung festschreiben, wird die Änderung über eine *Deployment-Pipeline* in der Produktion bereitgestellt. Die *CDE-Praxis* ist ein Pull-Basierter-Ansatz (*Pull-Request*), bei dem das Unternehmen entscheidet, was und wann bereitgestellt werden soll. Die *CD-Praxis* ist ein Push-basierter Ansatz. Mit anderen Worten, der Anwendungsbereich von *CDE* umfasst keine häufigen und automatisierten Freigaben. Bei *CD* handelt es sich, um eine Erweiterung von *CDE*. Während die *CDE-Praxis* für alle Arten von Systemen und Organisationen angewendet werden kann, eignet sich die *CD-Praxis* möglicherweise nur für bestimmte Arten von Systemen.

4.2 GitOps

Obwohl Kubernetes ((Billy et al., 2021), S. 3–4) eine weit verbreitete Open-Source-Plattform ist, die Operationen orchestriert und automatisiert, hat Kubernetes häufig Probleme, die Komplexität der Freigabe von Anwendungen zu verwalten. Da Git heute das am häufigsten verwendete Versionskontrollsystem in der Softwarebranche ist, setzt GitOps auf Git auf, um die Leistungsfähigkeit von Versionskontrollsystemen maximal auszunutzen. Bei GitOps handelt es sich, um eine Reihe von Verfahren und *Tools*, um sowohl die Versions- als auch die Änderungskontrolle innerhalb der Kubernetes-Plattform zu gewährleisten. Deswegen wird GitOps auch als Teil der DevOps-Kultur betrachtet. Die Wahl der richtigen GitOps-Strategie hat einen großen Einfluss darauf, wie schnell und einfach Teams die Umgebungserstellung, die Promotion und den Betrieb ihrer Dienste verwalten. Die Verwendung von GitOps mit Kubernetes ist eine natürliche Ergänzung, da die Bereitstellung von deklarativen Kubernetes-Manifestdateien durch allgemeine Git-Operationen gesteuert wird. Die zentralen Vorteile von *Infrastructure as Code* und unveränderlicher Infrastruktur liegen in der Bereitstellung, Überwachung und dem Lebenszyklusmanagement von Kubernetes-Anwendungen. Der Zugang für den Benutzer erfolgt auf eine intuitive und eine zugängliche Weise.

Es gibt zwei alltägliche ((Billy et al., 2021), S. 4) Aufgaben bei der Verwaltung und dem Betrieb von Computersystemen. Es ist die Konfiguration der Infrastruktur und die Bereitstellung von Software. Bei der Konfiguration von Infrastrukturen werden die Computerressourcen vorbereitet, damit die

Softwareanwendung ordnungsgemäß funktionieren kann. Die Softwarebereitstellung ist der Prozess, bei dem eine bestimmte Version einer Softwareanwendung für den Betrieb auf der Computerinfrastruktur vorbereitet wird. Die Verwaltung dieser beiden Prozesse ist der Kern von GitOps. Der Begriff GitOps wurde im August 2017 in einer Reihe von Blogs¹ von Alexis Richardson, Mitbegründer und CEO von Weaveworks. Seitdem hat sich der Begriff in der Cloud-Native-Community im Allgemeinen und in der Kubernetes-Community im Besonderen durchgesetzt. Deswegen ist GitOps ein DevOps-Prozess, der durch Verfahren und Methoden gekennzeichnet ist. Die wichtigsten ((Billy et al., 2021), S. 7) Methoden und Verfahren sind:

- Bewährte Verfahren für die Bereitstellung, Verwaltung und Überwachung von containerisierten Anwendungen.
- Eine entwicklerzentrierte Erfahrung für die Verwaltung von Anwendungen mit vollständig automatisierten *Pipeline* und Workflows unter Verwendung von Git für Entwicklung und Betrieb.
- Verwendung des Git-Revisionskontrollsystems zur Verfolgung und Genehmigung von Änderungen an der Infrastruktur und Laufzeitumgebung von Anwendungen.

Der Hauptunterschied: Alles liegt im Repo!

Durch netzbasierter Dienste ((Billy et al., 2021), S. 8) zur Versionsverwaltung für Software Entwicklungsprojekte, wie GitHub, entstehen ein zentraler Bestandteil des modernen Lebenszyklus der Softwareentwicklung. Deshalb erscheint es nur natürlich, dass eine Versionsverwaltung auch für den Betrieb und die Verwaltung von Systemen verwendet wird. In einem GitOps-Modell wird die gewünschte Konfiguration des Systems in einem Revisionskontrollsystem wie Git gespeichert. Anstatt Änderungen direkt am System über eine Benutzeroberfläche oder eine Befehlszeile vorzunehmen, nimmt ein *Engineer* Änderungen an den Konfigurationsdateien vor, die den gewünschten Zustand repräsentieren. Ein Unterschied zwischen dem in Git gespeicherten Soll-Zustand und dem Ist-Zustand des Systems zeigt an, dass noch nicht alle Änderungen implementiert wurden. Diese Änderungen können durch standardmäßige Revisionskontrollprozesse wie *Pull-Requests*, *Code-Reviews* und *Merges* zu dem *Master-Branch* überprüft und genehmigt werden. Wenn die Änderungen genehmigt und in den Hauptzweig zusammengeführt wurden, ist ein Operator-Softwareprozess dafür verantwortlich, den aktuellen Zustand des Systems auf der Grundlage der in Git gespeicherten Konfiguration in den gewünschten Zustand zu ändern. In einer idealen Implementierung von GitOps sind **manuelle Änderungen am System nicht zulässig** und alle Änderungen an der Konfiguration müssen an den in Git gespeicherten Dateien vorgenommen werden. Im Extremfall wird die Erlaubnis, das System zu ändern, nur dem Softwaresystem des Betreibers erteilt. Die Rolle der Infrastruktur- und Betriebsingenieure in einem GitOps-Modell verlagert sich von der Durchführung von Infrastrukturänderungen und Anwendungsbereitstellungen auf die Entwicklung und Wartung der GitOps-Automatisierung und die Unterstützung der Teams bei der Überprüfung und Genehmigung von Änderungen mithilfe von Git. Git verfügt über zahlreiche Funktionen und technische Möglichkeiten, die es zu einer idealen Wahl für die Verwendung mit GitOps machen. Durch die Veränderung der Rolle des typischen Entwicklers, der nicht mehr nur zuständig für die Entwicklung und den Abhängigkeiten im Entwicklungskontext ist, gewinnt DevOps für GitOps eine wichtige Bedeutung und wird deshalb als Prozess im DevOps-Kontext bezeichnet.

Durch GitOps ((Billy et al., 2021), S. 9) ergeben sich für Entwickler viele Vorteile, denn es ermöglicht ihnen, die Konfiguration der Infrastruktur und die Bereitstellung des Codes ähnlich zu handhaben

¹<https://www.weave.works/blog/devops-the-next-evolution-gitops>, letzter Zugriff 06.08.2022

wie ihren Softwareentwicklungsprozess, und zwar mit einem vertrauten Tool *Git*. Darüber hinaus wird Nachvollziehbarkeit von Änderungen erhöht und dadurch ist ein einfaches Rollback, Wiederherstellbarkeit und Selbstheilung durch *Git* ermöglicht. Soweit so gut, jetzt liegt alles im *Repository* und lässt sich durch *Git* verwalten. Es stellen sich dennoch zwei wichtige Fragen an der Stelle.

1. *Wie werden sentimentale Daten wie Passwörter geschützt?* Dies wird durch Operatoren wie *Sealed-Secret* ermöglicht, welche verschiedene Verschlüsselungsalgorithmen anwenden, um die Daten im Rahmen von Security- und Compliance-Richtlinien zu schützen. Der *Sealed-Secret*-Ansatz wird in dem Unterabschnitt 4.3.5 genauer erläutert.
2. *Wie werden die Änderung im Zielsystem ausgerollt?* An der Stelle kommen Operatoren wie *Flux* zum Einsatz, welche dem Continuous-Deployment von dem CI/CD-Ansatz umdrehen, um den Mechanismus von *Push* auf *Pull* umstellen. Der *Flux*-Ansatz wird in dem Unterabschnitt 4.3.2 genauer erläutert.

4.3 DevOps-Tools

Der Abschnitt beschäftigt sich nicht mit der Kultur, da diese bereits bei den Grundlagen erläutert worden ist. Es geht bei dem Abschnitt, um die unterschiedlichen DevOps-Tools, die für den Bau von *CI/CD-Pipelines* verwendet werden, um den DevOps-Ansatz zu ermöglichen.

4.3.1 Git

Ein elementarer Grundbaustein ((Chacon & Straub, 2014), S.12-16) von DevOps ist die Möglichkeiten einer dezentralen Versionsverwaltung des Codes. Somit stellt *Git* das Fundament für die Zusammenarbeit zwischen den verschiedenen Teams dar. Bei *Git* handelt es sich, um ist das am weitesten verbreitete Versionskontrollsystem. Dies liegt daran, dass *Git* Änderungen verfolgt, die an Dateien vorgenommen werden, sodass eine Aufzeichnung darüber existiert, was getan wurde. Das Zurückkehren zu bestimmten Versionen wird durch eindeutige *Commit-IDs* ermöglicht. Die Zusammenarbeit wird durch *Git* erleichtert, da Änderungen von mehreren Personen in einer Quelle zusammengeführt werden. Es ist eine Software, die lokal ausgeführt wird. Die Dateien und deren Verlauf werden auf Client Computer gespeichert. Es besteht auch die Möglichkeiten, diese online und remote wie zum Beispiel bei *GitHub* einzuchecken, um eine Kopie der Dateien und deren Änderungshistorie zu speichern. Das zentrale Abspeichern an einem Ort, an dem Änderungen gepusht oder gepullt werden, vereinfacht die Zusammenarbeit mit Entwicklern. Dazu werden *Git-Repository* verwenden, welche alle Projektdateien und den gesamten Änderungsverlauf enthalten. Das Branching- und Merging-Konzept von *Git* ermöglicht das Abzweigen von der ursprünglichen Codebasis, um einen Bug zu heben oder ein Feature zu implementieren. Dadurch haben die Entwickler mehr Flexibilität in ihrem Arbeitsablauf. Dieses Konzept ermöglicht außerdem das Implementieren einer Logik, um auf verschiedene *Branches* zu triggern und diese zu nutzen, um auf verschiedene Zielumgebungen zu deployen. Das Branching-Konzept wird auch unter anderem von *GitOps* ausgenutzt, um einen Workflow zu ermöglichen, der mit dem Pull-Ansatz funktioniert.

4.3.2 Flux

Bei *Flux* handelt ((Billy et al., 2021), S. 284-285) es sich um ein Open-Source-Projekt, das eine *GitOps* gesteuerte kontinuierliche Bereitstellung für *Kubernetes* implementiert. Das Projekt wurde 2016

bei Weaveworks¹ gestartet und trat drei Jahre später der CNCF Sandbox bei. Das Unternehmen Weaveworks ist dasselbe Unternehmen, das außerdem den Begriff GitOps geprägt hat. Zusammen mit anderen großartigen Open-Source-Projekten für Kubernetes hat das Unternehmen *Best Practices* für GitOps formuliert und viel zur Verbreitung von GitOps beigetragen. Die Entwicklung von Flux veranschaulicht, wie sich die Idee von GitOps im Laufe der Zeit auf der Grundlage praktischer Erfahrungen zu ihrer heutigen Form entwickelt hat. Das Flux-Projekt wurde ins Leben gerufen, um die Bereitstellung von *Container-Images* für Kubernetes zu automatisieren und die Lücke zwischen den kontinuierlichen Integrations- und Deployment-Prozessen zu schließen. Nach mehreren Iterationen erkannte das Flux-Team alle Vorteile eines Git-zentrierten Ansatzes. Vor der Veröffentlichung der Version 1.0 wurde die Projektarchitektur überarbeitet, um Git als Quelle der Wahrheit zu nutzen und die Hauptphasen des *GitOps-Workflows* zu formulieren.

Die genaue Aufgabe von Flux

Der Fokus von Flux ((Billy et al., 2021), S. 286-287) liegt auf der automatisierte Bereitstellung von Manifesten für den Kubernetes-Cluster. Dabei führt Flux keine zusätzlichen Schichten über Kubernetes ein. Eine einzelne Flux-Instanz verwaltet einen Kubernetes-Cluster und erfordert, dass der Benutzer ein *Git-Repository* verwaltet, das den Zustand des Clusters wiedergibt. Anstatt eine Benutzerverwaltung, SSO-Integration und eine eigene Zugriffskontrolle einzuführen, läuft Flux normalerweise innerhalb des verwalteten Clusters und verlässt sich auf Kubernetes Role-Based-Access-Control (RBAC). Dieser Ansatz vereinfacht die Flux-Konfiguration erheblich und trägt dazu bei, die Lernkurve abzuflachen. Eine Kubernetes-Plattform unterstützt die rollenbasierte Zugriffskontrolle, die es ermöglicht, Container an Rollen zu binden, die ihnen das Recht geben, auf verschiedene Ressourcen innerhalb des Clusters zuzugreifen. Die Einfachheit von Flux macht es auch praktisch wartungsfrei und einfach in das *Cluster-Bootstrapping* zu integrieren, da keine neue Komponente oder Admin-Rechte erforderlich sind. Mit der Flux-Befehlszeilenschnittstelle kann die Flux-Bereitstellung problemlos in die Skripte zur Clusterbereitstellung integriert werden, um eine automatisierte Clustererstellung zu ermöglichen. Somit ist Flux nicht auf das *Bootstrapping* von Clustern beschränkt und wird erfolgreich als kontinuierliches Deployment-Tool für Anwendungen eingesetzt. In einer mandantenfähigen Umgebung kann jedes Team eine Instanz von Flux mit eingeschränktem Zugriff installieren und zur Verwaltung eines einzelnen Namespace verwenden. Das gibt dem Team die Möglichkeit, die Ressourcen im Namespace der Anwendung zu verwalten, und ist dennoch sicher, da der Flux-Zugriff über Kubernetes *RBAC* verwaltet wird.

Zusätzlich zu den Kernfunktionen ((Billy et al., 2021), S. 286-287) von GitOps bietet das Projekt eine weitere bemerkenswerte Funktion an. Dabei ist Flux in der Lage, die Docker-Registry zu scannen und Images im *Deployment-Repository* automatisch zu aktualisieren, wenn neue *Tags* in die Registry gepusht werden. Obwohl diese Funktion keine Kernfunktion von GitOps ist, vereinfacht sie das Leben der Entwickler und erhöht die Produktivität. In der Regel besteht die Lösung eine neue Image-Version zu beziehen darin, Manifestaktualisierungen mithilfe der *CI-Pipeline* zu automatisieren. Der CI-Ansatz löst das Problem, erfordert aber *Scripting* und kann anfällig für Fehler sein. Anstatt das CI-System und Skripte zu verwenden, ermöglicht die Konfiguration von Flux, dass das *Deployment-Repository* jedes Mal automatisch aktualisiert wird, wenn ein neues *Image* in die Docker-Registry gepusht wird.

¹<https://fluxcd.io>, letzter Zugriff am 18.07.2022

Flux Architektur und Workflow

Die Architektur von Flux ((Billy et al., 2021), S. 288) ist relativ flach und besteht aus nur zwei Komponenten. Zum einen dem *Flux-Daemon* und dem Key-Value-Speicher Memcached¹. Bei Memcached handelt es sich um ein quelloffenes, hochleistungsfähiges, verteiltes Speicherobjekt-Caching-System. Der Hauptzweck von Memcached besteht darin, das Scannen der Docker-Registry zu unterstützen und stellt einen Speichermechanismus, um die verfügbaren Image-Versionen durch Flux abzuspeichern. Die Image-Versionen werden verwendet, um eine Liste der verfügbaren Image-Versionen der einzelnen *Docker-Images* zu speichern. Das *Memcached-Deployment* ist eine optionale Komponente und nicht erforderlich. Es darf immer nur eine Replika des *Flux-Daemons* ausgeführt werden. Dies ist jedoch kein Problem, denn selbst wenn der *Daemon* mitten in einer Bereitstellung abstürzt, startet er schnell neu und setzt den Bereitstellungsprozess idempotent fort. Der *Flux-Workflow* wird an der Abbildung 4.2 erläutert. Der Entwickler pusht sein Update wie zum Beispiel Namensänderung des Deployments in ein *Git-Repository* (1.) auf einen bestimmten *Branch*. Der *Flux-Daemon* (2.) lauscht in einem bestimmten Zeitintervall und holt sich die aktuelle Änderung anhand des neuen Commits, der durch den Push des Entwicklers entstanden ist. Dafür hat der *Daemon* einen *Source-Controller*, der für die Referenzen auf die *Repositories* zuständig ist. Es gibt auch einen optionalen *Image-Controller* (3.) der auch in einem vordefinierten Intervall oder über ein Webhook nach einer neuen Version in der Registry nachschaut. Existiert eine Änderung am *Image* und die Annotation ist auf *fluxcd.io/automated: 'true'* gesetzt, dann aktualisiert der *Flux-Daemon* das *Deployment* mit der neuen Version. Dabei erstellt Flux in der Memcached-Datenbank (4.) eine Liste aller vorhandenen Versionen des Images in Form einer Abbildung. Im letzten Schritt (5.) synchronisiert der *Flux-Daemon* die Änderungen in dem Kubernetes-Cluster, in dem er läuft. Dadurch wird gewährleistet, dass die ausgerollte Version im Cluster der eingetragenen Version im *Git-Repository* entspricht.

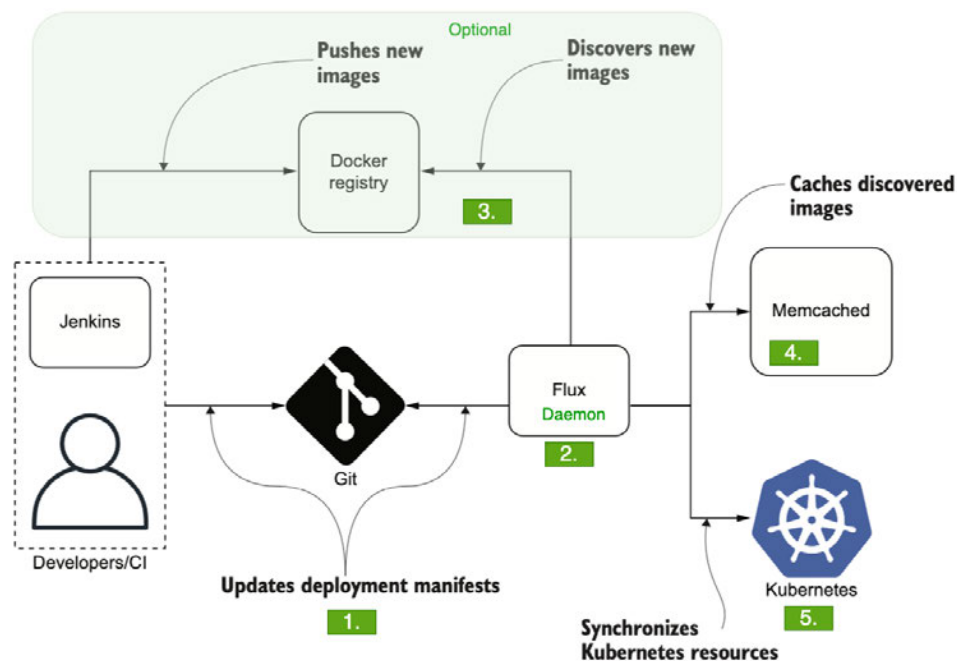


Abbildung 4.2: Flux Architektur, Bildquelle: eigene Darstellung in Anlehnung an ((Billy et al., 2021), S. 288)

¹<https://memcached.org/>, letzter Zugriff am 18.07.2022

Zusammenfassend lässt sich sagen, dass Flux einfach zu installieren und zu warten ist, da Flux keine neuen Komponenten benötigt und Kubernetes *RBAC* für die Zugriffskontrolle verwendet. Außerdem kann Flux neue *Images* und Updates aus dem *Git-Repository* automatisch beziehen. Es gibt eine *Flux-CLI*, die die Interaktion mit Flux vereinfacht. Des Weiteren kann Flux durch die zentrale Bereitstellung von Namespaces mit Zugriffskontrolle und Namespace-spezifischen Flux-Instanzen für die Mandantenfähigkeit konfiguriert werden. Verbraucht, aber dafür zusätzliche Ressourcen im Cluster, da pro Namespace bzw. pro Team ein *Flux-Stack* benötigt wird.

4.3.3 Helm

Die Verwaltung ((Billy et al., 2021), S. 292) von einfachen YAML-Dateien im Deployment Repository ist keine besonders schwierige Aufgabe, aber in der Praxis auch nicht sehr sinnvoll. Deshalb hat der *Flux-Stack* einen *Helm-Controller* und einen *Kustomization-Controller*, um Manifeste zu verwalten. Es ist gängige Praxis, den Basissatz von Manifesten für die Anwendung zu pflegen und umgebungsspezifische Manifeste mit *Tools* wie Kustomize oder Helm zu generieren. Die Integration mit Konfigurationsmanagement-Tools löst dieses Problem und Flux ermöglicht diese Funktion mithilfe von Generatoren zu nutzen. Helm ist als erstes Konfigurationstool ((Billy et al., 2021), S. 72–73) ein fester, integraler Bestandteil des Kubernetes-Ökosystems und deswegen sehr stark im Kubernetes-Kontext verwurzelt. Das Wichtigste an Helm ist, dass es ein selbstgeschriebener Paketverwalter für Kubernetes ist und nicht behauptet, ein Konfigurationsmanagement-Tool zu sein. Da jedoch viele Personen *Helm-Templating* für genau diesen Zweck verwenden, gehört es in diesen Abschnitt. Die Benutzer pflegen schließlich mehrere *values.yaml*, eine für jede Umgebung, wie zum Beispiel *values-base.yaml*, *values-prod.yaml* und *values-dev.yaml* und parametrisieren dann ihr Diagramm (siehe Abbildung 4.3 so, dass umgebungsspezifische Werte im Diagramm verwendet werden können. Diese Methode funktioniert mehr oder weniger gut, aber sie macht die Vorlagen unhandlich, da die Go-Vorlagen flach sind und jeden denkbaren Parameter für jeden einzelnen Parameter unterstützen müssen.

Im Folgenden werden die Stärken und Schwächen von Helm und die damit verbundenen Herausforderungen erläutert:

- **Stärke von Helm:** ist zweifelsohne sein hervorragendes Diagramm-Repository. Es kann zum Beispiel ein HA-Diagramm einfach im Namespaces erstellt werden. So kann eines einfachen Values in der jeweiligen Stage-Konfiguration das persistente Speichern einer Datenbank *persistentVolume.enabled: false* für eine Redis-Datenbank deaktiviert werden. Dadurch dass jemand anderes die harte Arbeit erledigt, herauszufinden, wie man Redis mit einer Vorkonfiguration zuverlässig in einem Kubernetes-Cluster betreibt.
- **Schwäche von Helm:** ist das *Go templating* von Helm. Es ist bekannt, dass Helm-Schablonen unter einem Problem der Lesbarkeit leiden. Das liegt daran, dass *Go templating* auf geschweifte Klammern und Funktionsaufrufe setzt. Die zu konfigurierten Diagramm-Parameter werden durch das Helm-Chart des Entwicklers gesetzt. Wenn die vorhandenen Diagramm-Parameter die gewünschten Änderungen nicht unterstützen, muss ein neues Helm-Diagramm erstellt werden. Es muss die semantische Version erhöht werden, in einem Diagramm-Repository veröffentlicht werden und mit einem Helm-Upgrade erneut bereitgestellt werden. Das muss jedes Mal bei einer spezifischen Änderung erfolgen.

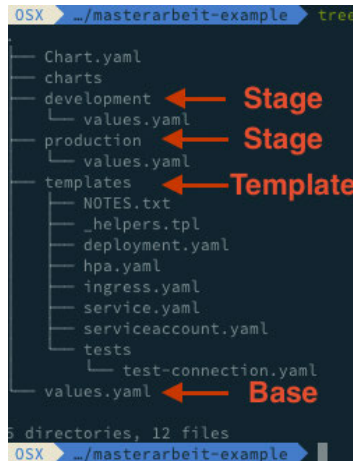


Abbildung 4.3: Helm-Diagramm Beispiel Baumansicht, Bildquelle: eigene Darstellung

4.3.4 Kustomization

Eine Alternative ((Billy et al., 2021), S. 76) zu Helm stellt Kustomize dar und wurde auf der Grundlage der Designprinzipien entwickelt, die in Brian Grants hervorragender Dissertation über deklaratives Anwendungsmanagement beschrieben sind. Der Anstieg von Kustomize war ziemlich stark und Kustomize wurde in den acht Monaten seit seinem Start bereits in *kubectl* integriert. Unabhängig davon, ob man mit der Art und Weise der Zusammenführung einverstanden ist oder nicht, ist es selbstverständlich, dass Kustomize-Anwendungen nun ein fester Bestandteil des Kubernetes-Ökosystems ist.

Im Folgenden werden die Stärken und Schwächen von Kustomize und die damit verbundenen Herausforderungen erläutert:

- **Stärke von Kustomize:** es gibt keine Parameter und Vorlagen. Dadurch sind *Kustomize-Apps* extrem einfach zu verstehen. Es kommt Kubernetes YAML-Manifesten so nahe wie möglich, da die Overlays, die zur Durchführung von Anpassungen erstellt werden, einfach Teilmengen von Kubernetes YAML-Manifesten sind.
- **Schwäche von Kustomize:** es gibt keine Parameter und Vorlagen. Dieselbe Eigenschaft, die Kustomize-Anwendungen so lesbar macht, schränkt diese gleichzeitig auch ein. Das Setzen eines *Build-Tags* für eine benutzerdefinierte Ressource anstelle eines Deployments ist über die CLI nicht möglich. Kustomize hat ein Konzept von *vars*, die wie Parameter aussehen, es aber nicht sind, und die nur in der von Kustomize genehmigten Whitelist von Feldpfaden verwendet werden können. Das ist einer der Fälle, in denen die Lösung, obwohl sie die schwierigen Dinge einfach macht, am Ende die einfachen Dinge schwer macht.

Die Funktionsweise von einer *Base* und *Overlays* funktioniert bei Kustomize einfach. Das Basisverzeichnis (siehe Abbildung 4.4 enthält die gemeinsame Konfiguration, die von den verschiedenen Umgebungen gemeinsam genutzt werden soll. Das Verzeichnis *development* und *production* in *overlays* enthält die Konfiguration für die jeweilige Umgebung. Es wird in der jeweiligen *Stage* in der *kustomization.yaml* die Base angegeben und dazu ein *Patch*, welcher dann ein Kubernetes-Manifest in dem Base-Verzeichnis überschreibt.



Abbildung 4.4: Kustomize Base und Overlay Beispiel Baumansicht, Bildquelle: eigene Darstellung

Kustomization vs Flux-Kustomization

Die *Kustomization-API* definiert eine Möglichkeit zum Abrufen, Entschlüsseln, Erstellen, Validieren und Anwenden von Kubernetes-Manifesten. Es gibt einen Unterschied zwischen den beiden Manifesten *kustomization.yaml*¹ und *flux-kustomization.yaml*. In jeder Flux Kustomization² gibt es ein implizit generiertes *kustomization.kustomize.config.k8s.io kustomize CLI Overlay*. Dieses wird von Flux Kustomization Custom Resource aus *kustomize.toolkit.fluxcd.io/v1beta1* generiert. Es kann auch ein eigenes Overlay hinzugefügt werden, wenn das automatisch generierte Overlay nicht den Anforderungen entspricht. Somit handelt es sich bei *kustomization.yaml* um ein Manifest, welches von *kustomize* verwaltet und verarbeitet wird. Dieses Manifest wird von Flux Kustomization Custom Resource generiert oder manuell erstellt.

In der Abbildung 4.5 werden die Unterschiede zwischen *kustomization.yaml* (Abbildung 4.5 rechts) und *flux-kustomization.yaml* (Abbildung 4.5 links) veranschaulicht. Bei der *flux-kustomization.yaml* wird eine andere *API* (1.) verwendet als bei *kustomization.yaml* (1.). Des Weiteren wird bei der *flux-kustomization.yaml* der Pfad (2.) zu dem *Gitrepo* (3.), welches als Quelle dient, referenziert. In diesem Pfad sucht Flux nach der *kustomization.yaml*. Falls keine vorhanden ist, dann generiert Flux Kustomization Custom Resource eine *kustomization.yaml* und fügt alle Ressourcen rekursive in dem angegebenen Pfad der *kustomization.yaml* hinzu. Falls eine *kustomization.yaml* vorhanden ist, dann werden die Ressourcen ausgerollt, die in der *kustomization.yaml* angegeben sind (4.). Zur besseren Unterscheidung wird für *flux-kustomization.yaml* der Dateiname *release-[resource-name].yaml* und für *Kustomize-Kustomization kustomization.yaml* verwendet.

```

! flux-kustomization.yaml U x
clusters > aks-development > cert-manager > ! flux-kustomization.yaml > ...
io.fluxcd.toolkit.kustomize.v1beta1.Kustomization (v1beta1@kustomization.json)
1
2 apiVersion: kustomize.toolkit.fluxcd.io/v1beta1 1.
3 kind: Kustomization
4 metadata:
5   name: cert-manager
6   namespace: cert-manager
7 spec:
8   interval: 30m0s
9   retryInterval: 15s
10  path: ./clusters/aks-development/cert-manager 2.
11  prune: true
12  sourceRef:
13    kind: GitRepository 3.
14    name: flux-system
15    namespace: flux-system
16  validation: client

! kustomization.yaml M x
clusters > aks-development > cert-manager > ! kustomization.yaml > ...
kustomization.yaml (kustomization.json)
1 apiVersion: kustomize.config.k8s.io/v1beta1 1.
2 kind: Kustomization
3 namespace: cert-manager
4
5 resources:
6 - namespace.yaml
7 - helmrelease.yaml 4.
8 - repository.yaml
9
10
11
12
13
14
15
16

```

Abbildung 4.5: Vergleich kustomization.yaml und flux-kustomization.yaml, Bildquelle: eigene Darstellung

¹<https://kubect1.docs.kubernetes.io/references/kustomize/>, letzter Zugriff 21.07.2022
²<https://fluxcd.io/docs/components/kustomize/kustomization/>, letzter Zugriff 21.07.2022

4.3.5 Sealed-Secrets-Operator

Der GitOps-Ansatz (Bitnami Labs, 2022) strebt das Ziel an, dass jegliche Konfiguration und der Quellcode an einem Ort abgelegt werden. Dies stellt bei dem GitOps-Ansatz ein *Git-Repository* als Source-Control-Management dar. Dabei ergibt sich eine Herausforderung mit den *Secrets* und sensiblen Daten. Der *Sealed-Secret-Operator* ermöglicht das Verschlüsseln von *Secrets* in ein *SealedSecret*. Das *SealedSecret* kann sogar sicher in einem öffentlichen *Repository* gespeichert werden kann. Es kann nämlich nur von dem *Controller* entschlüsselt werden, der im Zielcluster läuft, und niemand sonst ist in der Lage, das ursprüngliche *Secret* aus dem *SealedSecret* zu erhalten. Aus dem *SealedSecret* wird ein normales Kubernetes-Geheimnis nach ein paar Sekunden im Cluster erzeugt. Dieses kann wie jedes andere Geheimnis verwendet werden. Ein *Secret* aus dem *SealedSecret* ist ein abhängiges Objekt des *SealedSecret*-Objekts und wird als solches aktualisiert und gelöscht, sobald das *SealedSecret*-Objekt aktualisiert oder gelöscht wird. Der *Sealed-Secret-Operator* basiert auf einer PKI-Infrastruktur mit asynchroner Verschlüsselung. Das Schlüsselzertifikat (*public-key*) wird zum Versiegeln von Geheimnissen verwendet und muss überall dort verfügbar sein, wo *kubeseal* verwendet wird. Das Zertifikat ist keine geheime Information, obwohl sichergestellt werden muss, dass das richtige Zertifikat beim Versiegeln verwendet wird. Der *Operator* holt das Zertifikat zur Laufzeit vom *Controller* und erfordert deshalb einen sicheren Zugriff auf den *Kubernetes-API-Server*, was für die interaktive Nutzung praktisch ist. Es kann zu Problemen führen, wenn ein vom Benutzer erstelltes Cluster mit speziellen Konfigurationen und Einschränkungen versehen ist. Durch *Tools* wie den *Sealed-Secrets-Operator* kann der GitOps Ansatz im vollen Umfang umgesetzt werden.

4.3.6 externalDNS

Bei externalDNS (Kubernetes Sigs, 2022) handelt es sich um einen von Kubernetes DNS, dem cluster-internen DNS-Server von Kubernetes inspirierten Dienst, der Kubernetes-Ressourcen über öffentliche DNS-Server auffindbar macht. Die Hauptaufgabe vom externalDNS besteht in der Synchronisation von exponierte Kubernetes-Dienste und *Ingresses* mit DNS-Anbietern. Dabei ruft externalDNS wie KubeDNS es eine Liste von Ressourcen wie Dienste oder *Ingresses* von der *Kubernetes-API* ab, um eine gewünschte Liste von DNS-Einträgen zu bestimmen. Im Gegensatz zu *KubeDNS* ist es jedoch kein eigener DNS-Server, sondern konfiguriert lediglich andere DNS-Anbieter entsprechend wie zum Beispiel AWS Route 53¹ oder Azure DNS². Im weiteren Sinne ermöglicht externalDNS die dynamische Steuerung von DNS-Einträgen über Kubernetes-Ressourcen auf eine DNS-Anbieter-unabhängige Weise.

4.3.7 Azure DevOps Pipelines

Für die Umsetzung (Microsoft, 2022c) und für den möglichen Einsatz von DevOps-Tools im CI/CD-Kontext werden *Pipelines* benötigt, die in Abhängigkeit von einem Ereignis angestoßen werden und einen Job verrichten. Deswegen bietet Microsoft mit Azure DevOps *Pipelines* eine Lösung, die CI und CD kombiniert, um den *Code* zu erstellen, zu testen und in eine Zielumgebung zu liefern. Dabei erweitert Azure DevOps Pipelines CI/CD um Continuous-Testing (CT), um lokal oder in Cloud die Verwendung von automatisierten *Testworkflows* mit einer Auswahl von *Frameworks* zu ermöglichen. Die Basis und somit auch den Ausgangspunkt für die Entwicklung des Quellcodes stellt Azure DevOps mit einem Versionskontrollsystem zur Verfügung. Dabei kann zwischen zwei unterschiedlichen Versi-

¹<https://aws.amazon.com/de/route53/>, letzter Zugriff am 22.07.2022

²<https://azure.microsoft.com/de-de/services/dns/>, letzter Zugriff am 22.07.2022

onskontrollsystemen gewählt werden. Es wird Github und *Azure-Repos* unterstützt. Es werden die gängigsten Programmiersprachen durch native *Azure DevOps Tasks* unterstützt. Darüber hinaus werden unterschiedliche Bereitstellungsziele durch Service-Connections von Azure DevOps unterstützt. Zu den Zielen gehören darunter virtuelle Computer, Container, lokale Umgebungen oder Orchestrierungsplattformen wie Kubernetes. Für die Verwendung von Azure DevOps *Pipelines* wird eine Organisation benötigt, in welcher der Quellcode in einem Versionskontrollsystem gespeichert wird. Die kostenlose Version ist beim Ausführen der *Pipeline* auf 30 Stunden pro Monat beschränkt.

4.3.8 Terraform

Bei Terraform (HashiCorp, 2022) handelt es sich um ein von Hashicorp entwickeltes *Infrastruktur-as-Code-Tool*, mit dem sowohl Cloud- als auch On-Premise-Ressourcen in von Menschen lesbaren Konfigurationsdateien definiert werden. Dabei hat Hashicorp eine eigene Konfigurationssprache, die *Hashicorp-Language* entwickelt, welche als Wrapper um JSON-Syntax fungiert. Es ist sehr stark an der Syntax von Programmiersprachen angelehnt und ermöglicht dadurch versionieren, wiederverwenden und teilen von Codeabschnitten. Es kann ein konsistenter Arbeitsablauf verwendet werden, um die gesamte Infrastruktur während ihres Lebenszyklus bereitzustellen und zu verwalten. Dabei kann Terraform sowohl Low-Level-Komponenten wie Rechen-, Speicher- und Netzwerkressourcen als auch High-Level-Komponenten wie DNS-Einträge und SaaS-Funktionen verwalten. Die Verwaltung von Infrastrukturen als Ressourcen in der Cloud- oder *On-Premise* erfolgt über Anwendungsprogrammierschnittstellen. Dabei ermöglichen Anbieter es Terraform, mit praktisch jeder Plattform oder jedem Dienst mit einer zugänglichen *API* zu arbeiten. Die HashiCorp und die Terraform-Community haben bereits mehr als 1700 Provider geschrieben, um Tausende von verschiedenen Arten von Ressourcen und Diensten zu verwalten und diese Zahl wächst weiter. Es können alle öffentlich verfügbaren Cloud-Service-Provider in der Terraform Registry gefunden werden. Zu den größten *CSP* gehören Amazon Web Services, Azure, Google Cloud Platform, Kubernetes, Helm, und viele mehr.

Terraform Workflow:

Der zentrale Arbeitsablauf von Terraform besteht aus drei Phasen:

- **Write:** definiert Ressourcen, die sich über mehrere Cloud-Anbieter und -Dienste erstrecken können. Es kann zum Beispiel eine Konfiguration erstellt werden, um eine Anwendung auf virtuellen Maschinen in einem Virtual-Private-Cloud-Netzwerk mit Sicherheitsgruppen und einem Load Balancer bereitzustellen.
- **Plan:** erstellt einen Ausführungsplan, der die Infrastruktur beschreibt, die erstellt, aktualisiert oder zerstört wird, basierend auf der bestehenden Infrastruktur und ihrer Konfiguration. Der erstellte Plan stellt den *Terraform-State* dar. Anhand des *States* ermittelt Terraform das Delta zwischen dem Ist- und Soll-Zustand.
- **Apply:** nach der Genehmigung führt Terraform die vorgeschlagenen Operationen in der richtigen Reihenfolge aus, wobei fast alle Ressourcenabhängigkeiten berücksichtigt werden. Wenn zum Beispiel die Eigenschaften eines Virtual-Network (*VNET*) aktualisiert wird und sich die Anzahl der virtuellen Maschinen in diesem *VNET* ändern, wird Terraform das *VNET* neu erstellen, bevor die virtuellen Maschinen skaliert werden.

Somit bietet Terraform ein gutes Tool für die Verwaltung von *Cross-Shared-Infrastruktur* durch das *Tracking* des Ist und Soll-Zustandes der Infrastruktur. Des Weiteren sind Terraform Konfigurationsdateien deklarativ. Sie beschreiben den Endzustand der ausgerollten Infrastruktur. Für die meisten

Ressourcen baut Terraform ein Ressourcendiagramm auf, um die impliziten Abhängigkeiten zwischen den Ressourcen zu bestimmen und erstellt oder modifiziert nicht abhängige Ressourcen parallel. Da Ihre Konfiguration in eine Datei geschrieben wird, kann diese im *Git-Repo* übergeben werden. Der *Terraform-State* kann dabei an einem beliebigen Ort gespeichert werden. Die Integration von Terraform in CI/CD ermöglicht es große, komplexe und wiederverwendbare Infrastrukturen per Knopf-Druck auszurollen.

5 | Design und Implementierung

Das Kapitel beschreibt die Design-Entscheidungen, welche verwendet werden, um die beiden Ansätze CI/CD und GitOps zu vergleichen. Dafür wurden zum Beginn im Kapitel Anforderung erhoben, welche als Grundlage für die Architekturentscheidungen und Implementierung dienen. Des Weiteren werden in dem Kapitel unterschiedliche Anwendungsfälle simuliert, um die Schwächen, Stärken und Abhängigkeiten zwischen den Entscheidungen zu demonstrieren.

5.1 Anforderungen: CI/CD und GitOps

In diesem Abschnitt geht es um die notwendigen Anforderungen, die im Rahmen der Arbeit für die technische Umsetzung beim Bau eines Prototyps benötigt werden. Dabei wird zwischen Anforderungen und Rahmenbedingungen unterschieden. Die Anforderungen stellen eine Kombination aus funktionalen und nicht funktionalen Anforderungen dar. Bei den Rahmenbedingungen handelt es sich organisatorische oder technologische Einschränkungen, die für die Umsetzung notwendig sind. Eine Rahmenbedingung ist somit zum Beispiel ein unterzeichnetes *Microsoft-Customer-Agreement* für die Bereitstellung der *Subscription*, um Cloud-Services zu nutzen.

In der folgenden Tabelle 5.1 werden die Mindestanforderungen und die damit verbundenen Rahmenbedingungen an Azure dargestellt.

Azure			
Azure Service	Name	Anforderungen	Rahmenbedingung
DNS zone	azure-cloud.la-cc.de	Namensauflösung durch Subzonen Delegation	Registrierte Domain beim Registrar (AWS)
Kubernetes service	aks-development	Bereitstellung Kubernetes-API	Subscription vorhanden
Route table	kubernetes	Routing-Tabellen ins Kubernetes-Subnet	Subscription vorhanden
Storage account	saterraformazurestate	Verschlüsseltes Speichern Terraform-State	Subscription vorhanden
Virtual network	vnet-azure-terraform-aks-development	Bereitstellung Adressen: Pods, Services	Subscription vorhanden
Managed Identity	aks-development-agentpool	Role: Network-Contributor	Subscription vorhanden
Virtual machine scale set	aks-node-21387130-vmss	Linux OS	Subscription vorhanden
Virtual machine	aks-node-21387130-vmss [0-1]	2x Standard_B4ms (vCPU: 4, RAM: 16 GB)	Subscription vorhanden
Azure DevOps			
Azure Service	Name	Anforderungen	Rahmenbedingung
Projekte	terraform-azure-aks, terraform-azure-modules	Git Basic Unterstützung	Azure Tenant u. Organisation vorhanden
Pipelines	terraform-cicd-pipeline, apply flux, deploy webapp	Syntax und Agents zur Ausführung von Skripten	Pipelines in Konfiguration erlaubt
Git-Repositories	terraform-azure-aks, terraform-azure-kubernetes, terraform-azure-storageaccount, terraform-azure-devops, terraform-azure-network, terraform-azure-resource-group	HTTPS- oder SSH-Zugriff	Azure Tenant u. Projekt vorhanden

Abbildung 5.2: Tabelle: Azure DevOps Anforderungen

Es existieren außerdem weitere Anforderungen wie die Orchestrierungsplattform Kubernetes oder den *Flux-Stack*, der den Pull-Ansatz ermöglicht. Diese werden in diesem Abschnitt nicht explizit aufgeführt, da die Aufgaben und die damit verbundenen Anforderungen im Laufe des Kapitels genauer beschrieben werden.

5.2 Bootstrapping

Der Abschnitt beschreibt die unterschiedlichen Bootstrapping-Segmente für die Bereitstellung der gesamten notwendigen Umgebung in Form von Flussdiagrammen. Bei einer noch nicht vorhandenen Infrastruktur ergibt sich das Henne-Ei-Problem. Es kann keine Infrastruktur als *IaC* definiert werden, da der gesamte Unterbau wie ein Azure-Projekt, ein *Git-Repository* und ein *Backend* für Terraform nicht vorhanden ist. Die Implementierung wird vom *Scratch* aufgesetzt und an keiner bereits vorhandenen Infrastruktur angedockt. Dabei gilt es wie in anderen Bereichen der Automatisierung die Henne-Ei-Herausforderungen zu lösen. Hierbei werden unter anderem folgende Fragen geklärt wie *Wo fängt die Automatisierung der Infrastruktur an?* oder *Wie erschaffe ich eine Semi-Automatisierung der noch nicht vorhandenen IaC?* Das Bootstrapping unterscheidet sich in bei der Umsetzung in drei Phasen. Die erste Phase „*Azure-Backend*“ stellt einen Semi-Automatisierten Schritt dar. Es wird Shell-Skript verwendet, um die notwendige grundlegende Infrastruktur bereitzustellen, auf denen die restlichen Phasen aufbauen. Der Schritt wurde gezielt so gewählt, damit der Unterbau unabhängig von der später verwalteten Infrastruktur mittels *IaC* nicht durch einen Fehler gelöscht wird. Der erste Schritt stellt somit eine Isolationsschicht, welche keinen manuellen Eingriff nach der Bereitstellung benötigt. Bei der zweiten Phase „*Azure-DevOps*“ handelt es sich, um die initiale Bereitstellung eines Azure DevOps-Projektes, eines *Git-Repositorys* und der damit notwendigen Konfiguration der *CI/CD-Pipeline*. Somit handelt es sich auch eine Semi-Automatisierung, die zwar ein fertig gebautes Terraform-Modul als *IaC* verwendet, aber welches einmalig manuell ausgeführt wird. Die letzte Phase „*Kubernetes und Flux-Stack*“ stellt einen vollständig automatisierten Schritt mithilfe von *IaC* und Terraform als Tool dar, um durch eine *CI/CD-Pipeline* die notwendige Infrastruktur auszurollen. Hierbei werden die ersten beiden Phasen als Fundament verwendet, um den *Kubernetes-Service* von Azure, den *Flux-Stack* und alle relevanten Systemkontext bezogenen Ressourcen auszurollen. Es bestand auch die Möglichkeiten, die Schritte in einem Schritt zu kombinieren. Die Implementierung erfolgt gezielt, aber nach dem *Keep It Simple and Stupid (KISS)* Prinzip, um von Gebrauch einer Microservice-Architektur zu machen.

In den folgenden Unterabschnitten werden die Bootstrapping-Phasen genauer erläutert und mithilfe von Flussdiagramm wird der Ablauf der jeweiligen Phase veranschaulicht.

5.2.1 Flussdiagramm: Bootstrapping „*Azure-Backend*“

Bei der ersten Bootstrapping-Phase handelt es sich um eine initiale Bereitstellung der grundlegenden Infrastruktur, die später die Basis für die weiteren Phasen und vor allem für den Betrieb darstellt. Dazu wurde ein Bash-Skript angefertigt, welches mehrere Funktionen beinhaltet zur Bereitstellung und Validierung von unterschiedlichen Services ermöglicht. Zur Beginn authentifiziert sich das Skript gegen die *Microsoft-Graph-API* (siehe Abbildung 5.3 mit dem eingeloggten Benutzer. Der Benutzer erhält ein *OAuth-Token*, welches die Berechtigung des Benutzers im Azure-Active-Directory und in dem *Tenant* enthält. Der Benutzer benötigt für die weitere Ausführung die *Role Application-Developer*, um eine *App-Registration* zu erstellen, die dann als technischer Benutzer in der späteren *CI/CD-Pipeline*

eingepflegt wird. Handelt es sich bei dem *Token*, um ein gültiges *Token* mit der notwendigen Berechtigung, dann wird eine *App-Registration* erstellt. Außerdem erfolgen zwei weitere Schritte parallel, die einen Terraform-Ordner mit der Grundstruktur erstellen, die Backend-Datei für den *Terraform-State* konfigurieren und einen *PAT-Token* generieren, welcher in der zweiten Phase gebraucht wird. Neben der *App-Registration* wird auch eine Ressourcen-Gruppe erstellt, in welcher ein *Storageaccount* provisioniert wird und dieser beinhaltet dann einen *Blob-Storage*, in welchem die *Terraform-State* pro *State* gespeichert werden. Zum Schluss wird das *Terraform-Backend* initialisiert. Dabei wird pro *Stage* durch das Mapping auf den Terraform-Workspace ein *Container* im *Blob* des *Storageaccounts* angelegt.

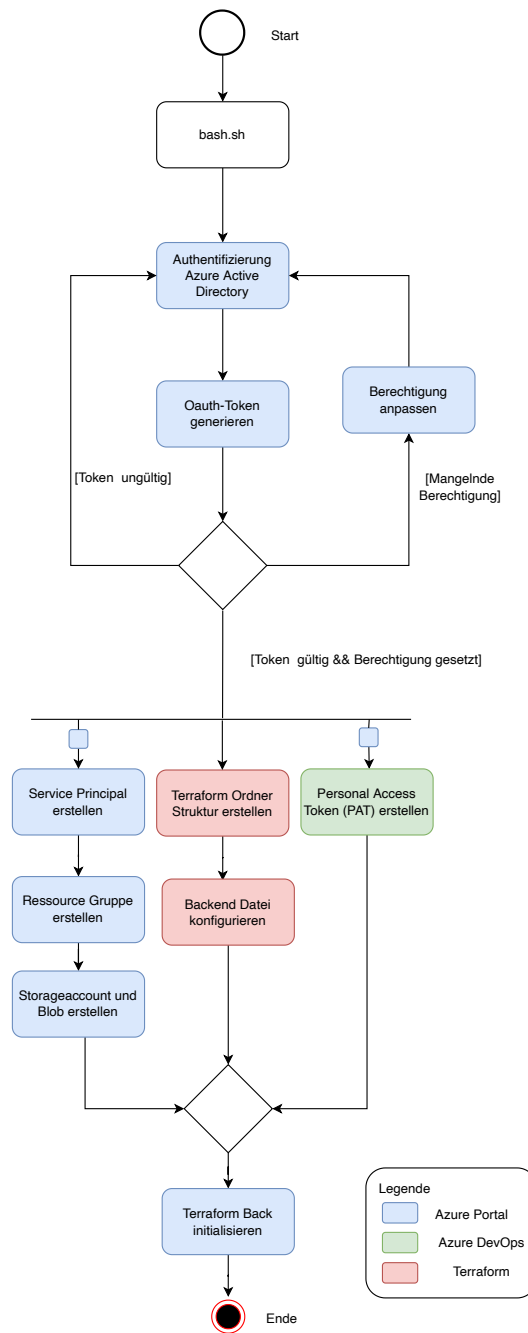


Abbildung 5.3: Flussdiagramm: Initiales Bootstrapping „Azure-Backend“, Bildquelle: eigene Darstellung

5.2.2 Flussdiagramm: Bootstrapping „Azure-DevOps“

Die zweite Bootstrapping Phase (siehe Abbildung 5.4 baut auf der ersten Phase auf und verwendet das generierte *PAT-Token*, um sich gegenüber der *Azure-DevOps-API* zu authentifizieren. Anschließend wird das Modul *terraform-azure-devops* geladen und durch das Setzen von Variablen konfiguriert. Über den Azure-DevOps-Provider wird ein Projekt erstellt, eine Variablengruppe angelegt und mit Variablen für die *CI/CD-Pipeline* befüllt. Zum Schluss wird eine Azure-DevOps-Pipeline erstellt, die auf die Variablengruppe und die darin liegenden Variablen referenziert, um zum Beispiel automatisiert gegen die *Microsoft-API* über ein *App-Registration* Ressourcen zu provisionieren. Die *CI/CD-Pipeline* wird in der nächsten und letzten Phase verwendet, um die Plattform und das damit verbundene Ökosystem bereitzustellen.

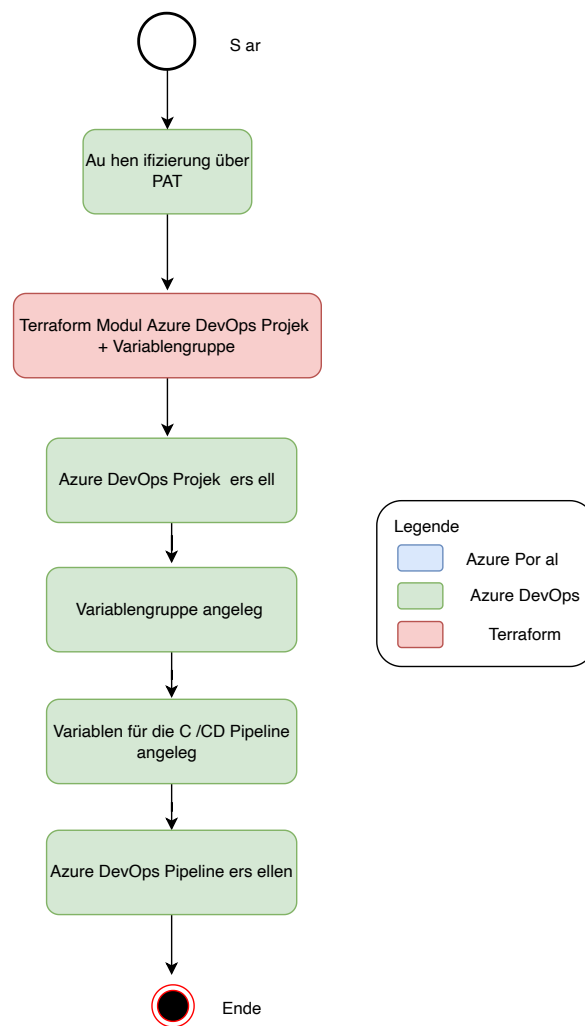


Abbildung 5.4: Flussdiagramm: Bootstrapping „Azure-DevOps“, Bildquelle: eigene Darstellung

5.2.3 Flussdiagramm: Bootstrapping „Kubernetes und Flux-Stack“

In der letzten Phase (siehe Abbildung 5.5) wird das Azure-Kubernetes-Cluster, der *Flux-Stack* und der notwendige Systemkontext ausgerollt. Dafür authentifiziert sich die *Pipeline* mit den *Credentials* der *App-Registration* als technischer Benutzer gegenüber der *Microsoft-API* und generiert ein *Token*. Wenn die *App-Registration* nicht die notwendigen Rechte hat, dann bricht die *Pipeline* an der Stelle ab und wird als *failed* mit einer entsprechenden Error-Massage gekennzeichnet. Ist das *To-*

ken valide, dann initialisiert die *Pipeline* in dem Schritt *validate* die benötigten Provider und lädt die Terraform-Module. Es existieren dabei Abhängigkeiten zwischen den Modulen. Zuerst wird eine Ressource-Gruppe erstellt, welche anschließend vom Netzwerk-Modul referenziert wird, um ein virtuelles Netzwerk für das Kubernetes-Cluster zu erstellen. Zum Schluss wird das Modul *terraform-azure-kubernetes* geladen, welches als Referenz sowohl die Ressource-Gruppe als auch das virtuelle Netzwerk enthält, um ein Subnetz für die Services und *Pods* zu provisionieren. Es handelt sich bei dem Teil, um den CI der CI/CD-Pipeline, weil an der Stelle die Validierung der Syntax erfolgt, die Initialisierung der benötigten Provider und anschließend ein Plan in der *Stage plan* erzeugt. Der Plan aus dem *Terraform-Code* gebaut, der im *Git-Repository* auf den *Main-Branch* eingchecked ist. Hierbei wird der Plan in die bereits (falls vorhanden) Infrastruktur integriert und erzeugt ein Delta zwischen dem Ist-Zustand und Soll-Zustand der im *Code* auszurollenden Ressourcen.

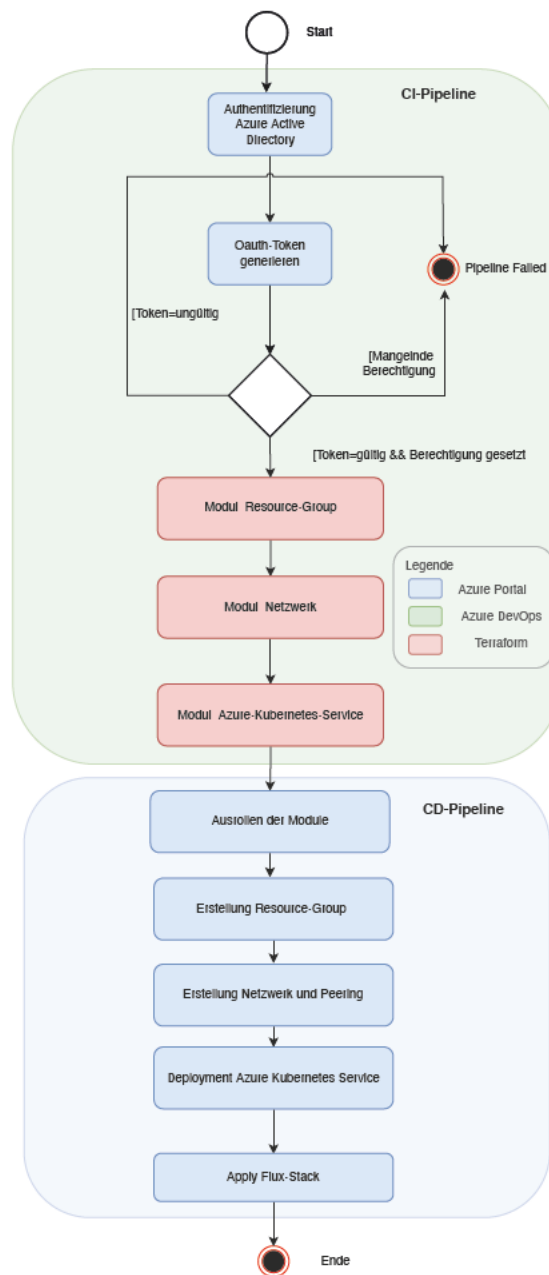


Abbildung 5.5: Flussdiagramm: Bootstrapping „Kubernetes und Flux-Stack“, Bildquelle: eigene Darstellung

Ist der CI-Teil erfolgreich, dann startet entweder der *Continuous-Delivery* oder *Continuous-Deployment* der Teil der *Pipeline* in Abhängigkeit von der *Stage*. Für die Implementierung wurde zwar die Logik eingebaut, bei einer produktiven *Stage* ein *Approved* einer bestimmten Gruppe einzuholen, wird dennoch aufgrund von *Overhead* und zusätzlichen Kosten nicht verwendet, da der Mehrwert für die Ausarbeitung nicht vorhanden ist. Somit handelt es sich bei der Implementierung um eine *Continuous-Deployment-Pipeline*. Der erzeugte Plan in dem CI-Teil wird als Artefakt an die *Stage apply* weitergereicht, um sicherzustellen, dass der validierte Plan auch gebaut wird. Zur Beginn der *CD-Pipeline* werden die Module ausgerollt. Hierbei wird, anders als bei dem CI-Teil, die Module nicht nur validiert und in den Plan integriert, sondern auch ausgerollt. Aufgrund der definierten Abhängigkeiten wird zuerst die Ressourcengruppe erstellt, in welcher das virtuelle Netzwerk angelegt wird und schließlich wird das AKS-Cluster ausgerollt. Am Ende der *Pipeline* wird eine weitere *Pipeline apply flux*, die ebenfalls auf den *Main-Branch* zeigt, getriggert. Diese rollt dann den *Flux-Stack* und die notwendigen Komponenten aus dem Systemkontext aus.

5.3 Architekturen

Der Abschnitt beschreibt die implementierte Architektur und deren einzelne Komponenten, die für das Zusammenspiel wichtig sind. Es handelt sich hierbei, um eine abstrahierte Ansicht der Architektur (siehe 5.6). Die Architektur besteht aus folgenden drei Komponenten.

- **Azure DevOps:** stellt die primäre Funktion zu Quellcodeverwaltung zur Verfügung und beinhaltet somit das *Git-Repository*. Des Weiteren liegen die Terraform-Module in einem eigenen Projekt, welches pro Modul ein *Git-Repository* bereitstellt. Der Zugriff auf die *Repositories* wird über SSH-Keys geregelt, da Azure DevOps über keine *Deploy-Tokens* verfügt und die Personal-Access-Token (PAT) maximal für ein Jahr ausgestellt werden können. Darüber hinaus stellt Azure DevOps Pipelines zur Verfügung, die durch den Benutzer als Manifeste beschrieben werden.
- **Azure:** stellt die *Azure-API* für Terraform als *Azure-Resource-Manager-Provider* zur Verfügung, um die Azure Services zu verwalten. Darüber erfolgt die Provisionierung vom Azure-Kubernetes-Service, *Public-DNS*, Netzwerken und *Storageaccount*. Des Weiteren bietet Azure über Azure-Active-Directory (AAD) und dem Identity-and-Access-Management (IAM) eine differenzierte Zugriffskontrolle auf Ressourcen von außen und integriert diese auch in das *RBAC* von Kubernetes.
- **AWS:** dient als Registrar und verwaltet die für die Masterarbeit registrierte Domain *azure-cloud.la-cc.de*. Es erfolgt eine Subzonen-Delegation von Azure zu AWS, in dem die DNS-Server von Azure unter der registrierten Domain eingetragen werden.

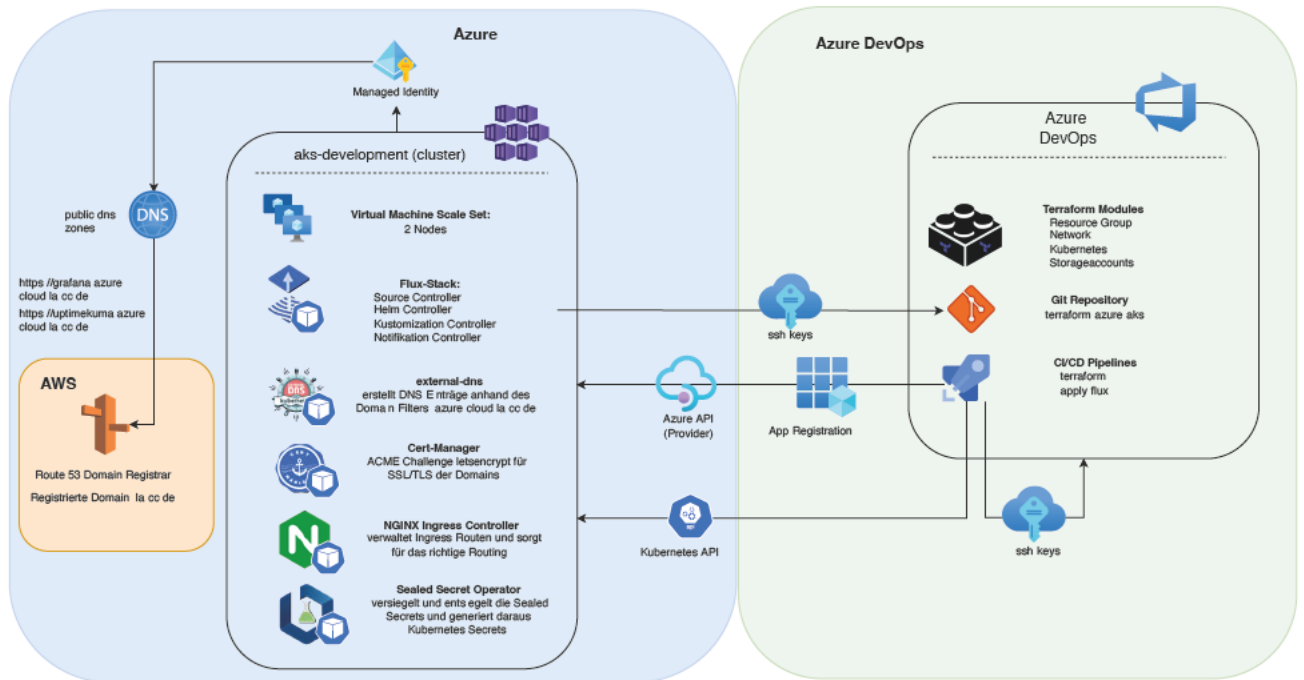


Abbildung 5.6: Abstrahierte Gesamtarchitektur, Bildquelle: eigene Darstellung

Das Zusammenspiel der einzelnen Komponenten und deren Abhängigkeiten sieht dann wie folgt aus. Der Terraform-Code, der für die Bereitstellung der AKS-Plattform, des *Public-DNS-Services*, des Netzwerkes und des *Storageaccounts* zuständig ist, wird im *Git-Repository* von Azure DevOps gepflegt. Ein Manifest *terraform-cicd-pipeline* definiert die *CI/CD-Pipeline*, die den *Terraform-Code* in den drei Stages *validate*, *plan*, *apply* ausführt. Dabei erkennt die *Pipeline* durch eingebaute Logik die Umgebung *development* oder *production* und führt diese entsprechend der Definition aus. Bei der *Stage production* handelt es sich um eine *Continuous-Integration-Continuous-Delivery-Pipeline*, die nach dem Vier-Augen-Prinzip die Änderung ins Zielsystem ausgerollt. Bei der *Stage development* handelt es sich um eine *Continuous-Integration-Continuous-Deployment-Pipeline*, die die Änderung direkt ins Zielsystem ausrollen. Bei der Implementierung gibt es nur die *Stage development*. Der *Terraform-Code* greift über *ssh-keys* auf Module zu, die in *Git-Repositories* in Azure-DevOps liegen, um die Ressourcen auszurollen. Die Kommunikation zu der *Azure-API* erfolgt über eine *App-registration*, die als technischer Nutzer dient, in der *Pipeline terraform-cicd-pipeline* hinterlegt ist und die Rolle *Contributor* erhält. Sobald die Ressourcen ausgerollt sind und die *Pipeline terraform-cicd-pipeline* erfolgreich durchgelaufen ist, wird die zweite *Pipeline apply-flux* getriggert. Diese verbindet sich über die *App-registration* gegen den bereitgestellten Azure-Kubernetes-Service, erstellt die notwendigen Ressourcen wie die API-Ressource *Gitrepo*, das *Secret* für die *ssh-keys* und rollt den *Flux-Stack* aus. Der *Flux-Stack* verwendet *flux-kustomizations*, die auf das *Gitrepo-Manifest* zeigen und auf einen bestimmten Ordner in dem *Gitrepo*, welcher als Startordner dient. Das *Gitrepo* stellt die Verbindung über die *ssh-keys* zu dem Repository *terraform-azure-aks* her und Flux bedient sich an dieser Verbindung. Die implementierte Flux Referenzen Architektur wird in der Abbildung 5.8 und der *Flux-Flow* in der Abbildung 5.7 genauer beschrieben. Die durch *Flux-Stack* ausgerollten Ressourcen, die im Azure-Kubernetes laufen, übernehmen folgende Aufgaben.

- **Source-Controller:** ist ein *Kubernetes-Operator* und dient zu der Erfassung von Artefakten aus Git. Der *Source-Controller* implementiert die *API source.toolkit.fluxcd.io* und ist eine Kernkomponente des *Flux-Stacks*. Er ist in der Implementierung für die Authentifizierung per SSH

gegenüber Azure-DevOps *Git-Repository* verantwortlich. Des Weiteren erkennt er Quelländerungen auf der Grundlage von Aktualisierungsrichtlinien und holt die Ressourcen (Kubernetes-Manifeste) auf Abruf und nach Zeitplan ab.

- **Helm-Controller:** ist für die Überwachung von Helm-Release-Objekten zuständig und erzeugt Helm-Chart-Objekte aus den geholten Manifesten des Source-Controllers. Da *Controller* Helm-Chart-Artefakte unterstützt, die aus *Helm-Repository*- und *Git-Repository*-Quellen erzeugt werden, wird er in der Architektur dazu verwendet, *Helmreleases* wie zum Beispiel externalDNS und die dazugehörigen Artefakte aus dem *Helmrepo* zu installieren.
- **Kustomization-Controller:** ist für das Ausgleichen des Zustands des Clusters und des eingeeckten Standes im *Git-Repository*, denn er in der implementieren Architektur vom *Source-Controller* als Quelle bereitgestellt bekommt. Daraus erzeugt er Manifeste mit *Kustomize* aus einfachen Kubernetes-YAMLS oder *Kustomize-Overlays*, validiert diese gegen die *Kubernetes-API*, führt eine Zustandsbewertung des bereitgestellten *Workloads* und gleicht diese an. Außerdem entfernt er nicht mehr vorhandene Objekte und führt den Job des *Garbage-Collectors* aus.
- **Notification-Controller:** verarbeitet Ereignisse, die von Azure-DevOps kommen und benachrichtigt den Source-Controller über Quelländerungen.
- **External-DNS:** übernimmt die Verantwortung für die dynamische Eintragung eines DNS-Eintrags, sobald ein *Ingress* erzeugt wird. Dafür verwendet der External-DNS eine *Managed Identity*, welche auf die *Public-DNS-Zone cloud-azure.la-cc.de* berechtigt ist. Damit der External-DNS auf die richtigen *Ingresses* triggern kann, hat er einen Domain-Filter auf die Domain *cloud-azure.la-cc.de* eingestellt. Sobald ein *Ingress* erzeugt wird, erhält der externalDNS vom *Ingress-Controller* die Aufforderungen einen A-Eintrag zu erstellen. Anschließend überprüft der External-DNS, ob alle Einträge auf dem aktuellen Stand sind.
- **NGINX-Ingress-Controller:** hat die Aufgabe Routen, die als *Ingress* erstellt werden, an den richtigen Service zu routen, damit die richtige Anwendung darauf antworten kann. Außerdem verwendet der *NGINX-Ingress-Controller* die *Managed-Identity*, um einen Firewall-Eintrag in der Network-Security-Group zu erstellen, welcher den Azure-Loadbalancer konfiguriert. Das ist notwendig, damit das Routing von der *Public-IP*, die durch den DNS-Namen aufgelöst wird, an die richtig private IP erfolgen kann.
- **Cert-Manager:** ist dafür verantwortlich, dass über eine *ACME-Challenge* zu dem DNS-Eintrag ein gültiges Letsencrypt-Zertifikat für die verschlüsselte Kommunikation über SSL/TLS erstellt wird. Sobald eine *Ingress* erstellt wird, geht ein *Certificaterequests* an den Cert-Manager. Der Cert-Manager erstellt eine temporäre Ingress-Ressource, um den Datenverkehr an die *acmesolver-Pods* weiterzuleiten, die für die Beantwortung von ACME-Prüfungsanfragen zuständig sind. Für das Lösen der Prüfung wird *ACME HTTP-01*¹ Challenge verwendet.
- **Sealed-Secret-Operator:** ermöglicht das sichere Ablegen von sensiblen Daten in einem *Git-Repository*. Die *Secrets* werden mit einem Public-Key verschlüsselt und nur der *Sealed-Secret-Operator* hat den Private-Key, um die *Sealed-Secrets* zu entschlüsseln und in *Kubernetes-Secrets* umzuwandeln. Auch wenn der Private-Key in die falschen Hände geraten sollte, ist es nicht

¹<https://cert-manager.io/docs/configuration/acme/http01/>, letzter Zugriff 02.08.2022

möglich, mit diesem das *Sealed-Secret* zu entschlüsseln. Dafür muss der Besitzer ein quasi identisches Kubernetes-Cluster mit denselben Metainformationen aufbauen, damit das versiegelte *Secret* gleich ist. Der *Sealed-Secret-Operator* wird in der implementierten Architektur verwendet, um zum Beispiel Benutzer und Passwort im *Repository* abzulegen, welcher dann von der Anwendung referenziert wird.

Somit soll das notwendige Zusammenspiel zwischen den einzelnen Komponenten und deren Abhängigkeit in der implementierten Architektur geklärt sein. Die einzelnen Komponenten sind noch stärker ineinander verschachtelt und der System-Kontext ist breiter als in der Abbildung 5.6 dargestellt. Deswegen wurde die Architektur auf das notwendigste abstrahiert reduziert dargestellt.

Flux Referenzen Architektur

Es gibt unterschiedliche Möglichkeiten, eine Flux Referenzen Architektur zwischen *flux-kustomization* und *kustomization* zu erstellen. Diese ist notwendig, damit der *Flux-Stack* die Ressourcen kontrolliert ausrollen und verwalten kann. Die Entscheidung bei der Referenz-Architektur (siehe Abbildung 5.7) fiel bei der Implementierung auf das Ablegen der *flux-kustomization (release-anwendung.yaml)* in den jeweiligen Anwendungsordner. Somit liegt das zugehörige Release in dem Anwendungsordner. Die Referenzen und auch Abhängigkeiten werden wie in der Abbildung 5.7 dargestellt aufgebaut. Nachdem der *Flux-Stack* ausgerollt worden ist, benötigt Flux einen Startpunkt. Dabei wird eine Verbindung zum *Gitrepo aks-development* erstellt. Anschließend wird die initiale Referenz auf das *Gitrepo* und den Pfad *.clusters/aks-development* aufgebaut. In diesem Verzeichnis befindet sich eine benutzerdefinierte *kustomization.yaml*. Diese enthält Referenzen auf die weiteren *Release-Files*, welche den jeweiligen Pfad zu der Anwendung definieren. In den Anwendungsordnern gibt es pro Anwendung eine benutzerdefinierte *kustomization.yaml*, die beschreibt, welche Ressourcen aus dem Anwendungsordner ausgerollt werden sollen. Zum Schluss wird in Abhängigkeit davon, ob es ein *Helm-Release* oder ein *Kustomization-Deployment* der *Helm-Controller* oder der *Kustomization-Controller* aktiv und rollt die Ressourcen aus. Die Abbildung 5.7 veranschaulicht die abhängigen Referenzen von der Root-Ebene *release-flux-system.yaml* bis hin zu den einzelnen Blättern, die als *Release-Files* in den Anwendungsordnern hinterlegt sind. Um die Abhängigkeiten zu verdeutlichen, wurden zwei Sonderfälle der Assoziation aus Unified-Modeling-Language (UML) verwendet und mit Multiplizität versehen. Somit existiert die *kustomization.yaml* unter der Root-Ebene nur, wenn das *release-flux-system.yaml* vorhanden ist. Die *kustomization.yaml* kann auch nur einer *release-flux-system.yaml* zugeordnet werden. Deswegen gibt es eine Komposition zwischen der *release-flux-system.yaml* und der *kustomization.yaml* auf der Ebene-0 und Ebene-1. Es existiert auch eine Komposition zwischen der *kustomization.yaml* auf der Ebene-0 und zwischen den *Release-Files* auf der Ebene-1. In der *kustomization.yaml* auf der Ebene-0 werden die Referenzen durch das Einbinden der *Release-Files* erstellt und entfernt. Wenn die *kustomization.yaml* auf der Ebene-0 gelöscht wird, dann verschwinden auch alle Referenzen zu den *Release-Files* auf Ebene-1.

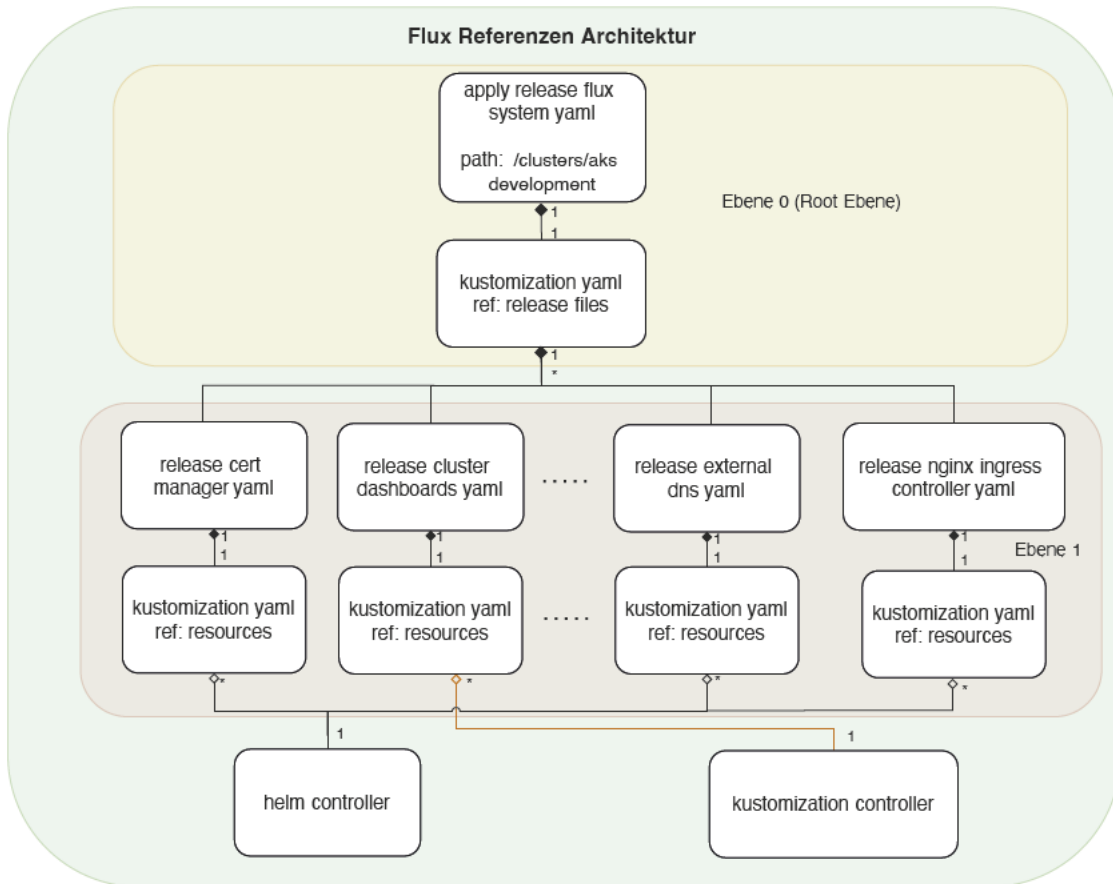


Abbildung 5.7: Flux Referenzen Architektur Bildquelle: eigene Darstellung

Die Abhängigkeit der Referenzen bestimmt nicht nur, in welcher Reihenfolge die Referenzen aufgebaut werden, sondern auch in welcher Reihenfolge die Referenzen abgebaut werden, sobald eine *flux-kustomization.yaml* gelöscht wird. Beim Löschen einer *flux-kustomization.yaml* werden auch die dazugehörigen Ressourcen gelöscht. Dies kann negative Folgen haben, wenn versehentlich die Root-Ebene *release-flux-system.yaml* gelöscht wird. Dies hat dann zur Folge, dass alle dazugehörigen Referenzen und Ressourcen ebenfalls gelöscht werden.

Zustandsdiagramm: Flux-Flow

Bei dem folgenden Diagramm (siehe Abbildung 5.8) handelt es sich zwar um keine Architektur, sondern um ein Zustandsdiagramm, der die implementierte Architektur (siehe Abbildung 5.6) unterstützt. Der *Flux-Flow* besteht in der implementierten Konstellation aus einem *Pull-System*, welcher eine Ähnlichkeit mit einem Regelkreis aus der Automatisierungstechnik hat. Wenn eine Änderung gepusht wird, dann erfolgt ein *Pull*, um den Ist-Zustand an den Soll-Zustand im *Repository* durch Einspielen der Änderung zu bringen. Dafür wird zuerst der *Flux-Stack* ausgerollt und der *Source-Controller* in einen lauschenden Zustand versetzt. Sobald die Zeit abläuft oder ein manueller Reconcile-Prozess angestoßen wird, holt der *Source-Controller* die Quelländerung ab. Anschließend erfolgt die Zusammenführung der Änderung. Schlägt die Zusammenführung fehl, dann wechselt der Zustand von „zusammenführend“ in „wiederholend“ und toggelt zwischen den beiden Zuständen bis die maximale konfigurierte Anzahl an Versuchen verbraucht ist. Danach wechselt der Zustand von „wiederholend“ in den Zustand „hängend“. Dieser Zustand ist wie eine Dauerschleife und der Wechsel ist nur möglich, wenn ein Ereignis wie eine Quelländerung erfolgt und eine neue Version durch die Zusammenführung der Ressourcen

ausgerollt wird. Dann erfolgt der Wechsel wieder in den Zustand „zusammenführend“. Der Wechsel aus dem Zustand „zusammenführend“ erfolgt in Abhängigkeit davon, ob es ein *Helm-Release* oder ein *Kustomize-Deployment* ist in den Zustand „kustomization-installierend“ oder „helm-installierend“. Ist die Installation erfolgreich, dann erfolgt wieder der Wechsel in den Zustand „lauschend“, andernfalls erfolgt der Wechsel nach einer bestimmten Anzahl von Fehlversuchen in den Zustand „hängend“.

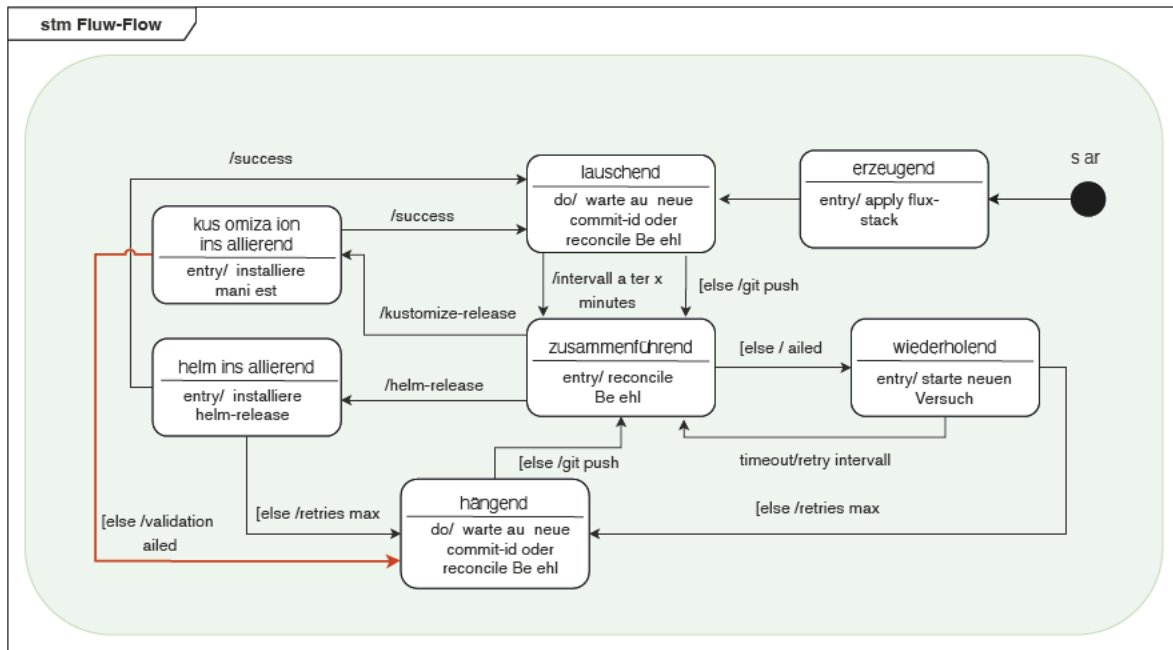


Abbildung 5.8: Zustandsdiagramm: Flux-Flow, Bildquelle: eigene Darstellung

Da es hierbei um mehrere Ressourcen handelt, die parallel abgearbeitet werden, ist das Erreichen eines Zustands pro Ressource möglich. Somit werden mehrere Zustände gleichzeitig erreicht. Der Reset aller Ressourcen und der Wechsel aller Ressourcen in den Zustand „zusammenführend“ erfolgt entweder durch einen Push in das observierte *Git-Repository* oder durch den Anstoß des Reconcile-Prozesses.

5.4 Anwendungsfall: Deployment CD vs Flux

Bei dem Anwendungsfall wird ein *Deployment* über den Push-Ansatz, welcher durch die *CI/CD-Pipeline* ermöglicht wird im direkten Vergleich zu dem Pull-Ansatz, welcher von Flux ermöglicht wird gegenübergestellt. Bei dem CD-Teil der *CI/CD-Pipeline* (siehe Abbildung 5.9 links) ist der Trigger auf den *Main-Branch* gesetzt und wird ausgelöst, sobald eine Änderung gepusht wird. Danach läuft die Pipeline los und rollt die Änderung aus. Bei dem *Flux-Deployment* (siehe Abbildung 5.9 rechts) ist der Trigger ebenfalls auf den *Main-Branch* gesetzt, mit einem Intervall von 30 Minuten. Nach 30 Minuten führt Flux einen Reconcile-Prozess aus und vergleicht den ausgerollten Zustand mit dem *Git-Repository-Zustand*. Wird von Flux ein Delta festgestellt, dann pullt Flux automatisch die Änderung und spielt diese auf dem AKS-Cluster ein. Es handelt sich bei dem *Deployment*, um eine einfache Web-Anwendung, die auf Python basiert und mithilfe der Bibliothek *Flask*¹ eine statische Website anzeigt. Die statische Webseite wählt per Zufall eine Farbe aus einem definierten Farb-Pool,

¹<https://flask.palletsprojects.com>, letzter Zugriff 07.08.2022

setzte diese als Umgebungsvariable in einem Alpine-Linux-Container¹ und färbt die statische Webseite damit ein. Für eine Unterscheidung zwischen Flux und *CD-Deployment* werden zusätzlich die Pod-Namen extrahiert und auf Seite als Text angezeigt. Die Anwendung dient nur zu Demonstrationszwecken für die Anwendungsfälle und ist deshalb minimalistisch gehalten. Der Abschnitt versucht die Anwendungsfälle weitestgehend objektiv und nicht wertend zu beschreiben. Die Auswertung zu den Anwendungsfällen erfolgt in dem Kapitel 6, in welchem die Ergebnisse und Rückschlüsse aus den Anwendungsfällen gezogen werden.

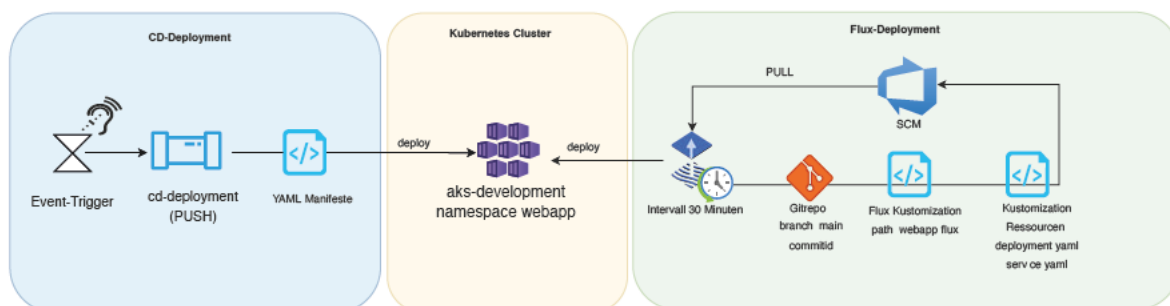


Abbildung 5.9: Deployment CD vs Flux, Bildquelle: eigene Darstellung

5.5 Anwendungsfall: manuelle Änderung – Skalierung, Deployment löschen und Umgebungsvariable manipulieren

Der Anwendungsfall „manuelle Änderung“ unterteilt sich in mehrere kleinere Anwendungsfälle, die unterschiedliche Aspekte von CI/CD und Flux vergleichen. Es wird pro unterteilten Anwendungsfall erst das *CD-Deployment* betrachtet und dann das *Flux-Deployment*.

5.5.1 Manuelle Änderung: Skalierung

Hierbei handelt es sich um eine Manipulation des Deployments. Dabei wird das *Replicaset* mithilfe der *Kubernetes-API* manuell angepasst. Dabei wird der folgende Befehl ausgeführt, um ein *Replicaset* entweder nach oben oder nach unten zu skalieren.

```
kubectl scale deployment webapp-color --replicas=5
oder
kubectl scale deployment webapp-color-flux --replicas=5
```

Manuelle Änderung: Skalierung (CD-Deployment)

Bei dem CD-Deployment ist der ausgerollte Zustand mit drei *Replicas* konfiguriert und es laufen drei *Pods*.

```
kubectl get deployment webapp-color
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
------	-------	------------	-----------	-----	------------

¹https://hub.docker.com/_/alpine, letzter Zugriff 01.08.2022

```
webapp-color 3/3 3 3 14h webapp-color
```

```
kubectl get pods --namespace webapp
```

NAME	READY	STATUS	RESTARTS	AGE
webapp-color-65f5f9d4d-8d7x5	1/1	Running	0	12h
webapp-color-65f5f9d4d-bxc1g	1/1	Running	0	12h
webapp-color-65f5f9d4d-vzd5j	1/1	Running	0	12h

Nun erfolgt die Skalierung auf ein *Replica-Set* von fünf:

```
kubectl scale deployment webapp-color --replicas=5
```

Der nun ausgerollte Zustand nach circa zwei Minuten sieht wie folgt aus:

```
kubectl get deployment webapp-color
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
webapp-color	5/5	5	5	14h	webapp-color

```
kubectl get pods --namespace webapp
```

NAME	READY	STATUS	RESTARTS	AGE
webapp-color-65f5f9d4d-851kh	1/1	Running	0	89s
webapp-color-65f5f9d4d-8d7x5	1/1	Running	0	13h
webapp-color-65f5f9d4d-bxc1g	1/1	Running	0	13h
webapp-color-65f5f9d4d-hhcwm	1/1	Running	0	89s
webapp-color-65f5f9d4d-vzd5j	1/1	Running	0	13m

Der nun ausgerollte Zustand nach circa 10h sieht wie folgt aus:

```
kubectl get deployment webapp-color
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
webapp-color	5/5	5	5	1d	webapp-color

```
kubectl get pods --namespace webapp
```

NAME	READY	STATUS	RESTARTS	AGE
webapp-color-65f5f9d4d-851kh	1/1	Running	0	10h
webapp-color-65f5f9d4d-8d7x5	1/1	Running	0	23h
webapp-color-65f5f9d4d-bxc1g	1/1	Running	0	23h
webapp-color-65f5f9d4d-hhcwm	1/1	Running	0	10h
webapp-color-65f5f9d4d-vzd5j	1/1	Running	0	23h

Der Zustand beim *CD-Deployment* ändert sich so lange nicht, bis die *CI/CD-Pipeline* durch ein Event wie eine Quellcode-Änderung auf dem *Main-Branch* getriggert wird.

Manuelle Änderung: Skalierung (Flux-Deployment)

Bei dem *Flux-Deployment* ist der ausgerollte Zustand mit drei *Replicas* konfiguriert und es laufen drei *Pods*.

```
kubectl get deployment webapp-color-flux
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
webapp-color	3/3	3	3	14h	webapp-color

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
webapp-color-flux	3/3	3	3	89m	webapp-color-flux

```
kubectl get pods --namespace webapp-flux
```

NAME	READY	STATUS	RESTARTS	AGE
webapp-color-flux-76c7fb4f9d-fsvlc	1/1	Running	0	90m
webapp-color-flux-76c7fb4f9d-g2sw7	1/1	Running	0	90m
webapp-color-flux-76c7fb4f9d-jhjrk	1/1	Running	0	90m

Nun erfolgt die Skalierung auf ein Replica-Set von fünf:

```
kubectl scale deployment webapp-color-flux --replicas=5
```

Der nun ausgerollte Zustand nach circa zwei Minuten sieht wie folgt aus:

```
kubectl get deployment webapp-color-flux
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
webapp-color	5/5	5	5	93m	webapp-color-flux

```
kubectl get pods --namespace webapp
```

NAME	READY	STATUS	RESTARTS	AGE
webapp-color-flux-76c7fb4f9d-8czk6	1/1	Running	0	2m
webapp-color-flux-76c7fb4f9d-fsvlc	1/1	Running	0	103m
webapp-color-flux-76c7fb4f9d-g2sw7	1/1	Running	0	103m
webapp-color-flux-76c7fb4f9d-jhjrk	1/1	Running	0	103m
webapp-color-flux-76c7fb4f9d-knbvw	1/1	Running	0	2m

Der nun ausgerollte Zustand nach circa 16 Minuten sieht wie folgt aus:

```
kubectl get deployment webapp-color-flux
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
webapp-color	3/3	3	3	1d	webapp-color

```
kubectl get pods --namespace webapp
```

NAME	READY	STATUS	RESTARTS	AGE
webapp-color-flux-76c7fb4f9d-g2sw7	1/1	Running	0	110m
webapp-color-flux-76c7fb4f9d-jhjrk	1/1	Running	0	110m
webapp-color-flux-76c7fb4f9d-knbvw	1/1	Running	0	18m

Der Zustand beim *Flux-Deployment* und der manuelle Eingriff, der ein Delta verursacht hat, wird von Flux im *Worst-Case* nach 30 Minuten erkannt. Das hängt davon ab, wann der letzte Reconcile-Intervall-Prozess gelaufen ist und wie das Intervall eingestellt ist.

5.5.2 Manuelle Änderung: Deployment löschen

Bei dem Anwendungsfall handelt es sich, um ein Szenario, bei dem ein *Deployment* gelöscht wird und somit eine *Downtime* verursacht wird, weil die Anwendung nicht mehr erreichbar ist. Der *Ingress-Controller* kann den Service nicht mehr erreichen und somit auch die dafür zuständigen *Pod*, die die Anwendung bereitstellen.

Manuelle Änderung: Deployment löschen (CD-Deployment)

Bei dem CD-Deployment ist der ausgerollte Zustand mit drei *Replicas* konfiguriert und es laufen drei *Pods*.

```
kubectl get deployment webapp-color
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
webapp-color	3/3	3	3	15h	webapp-color

Nun erfolgt das manuelle Löschen des Deployments:

```
kubectl delete deployment webapp-color
```

```
deployment.apps "webapp-color" deleted
```

```
kubectl get deployment webapp-color
```

No resources found in webapp namespace.

Ein Abrufen der Seite liefert nur einen Fehler-Code 503, welcher durch das Default-Backend vom *Ingress* abgefangen wird, sobald ein Service nicht mehr erreichbar ist.

```
curl -vvv https://webapp.azure-cloud.la-cc.de/
```

```
.....  
<html>  
<head><title>503 Service Temporarily Unavailable</title></head>  
<body>  
<center><h1>503 Service Temporarily Unavailable</h1></center>
```

```
<hr><center>nginx</center>
</body>
</html>
.....
```

Der Zustand beim *CD-Deployment* ändert sich so lange nicht, bis die *CI/CD-Pipeline* durch ein Event wie eine Quellcode-Änderung auf dem *Main-Branch* getriggert wird.

Manuelle Änderung: Deployment löschen (Flux-Deployment)

Bei dem *Flux-Deployment* ist der ausgerollte Zustand mit drei *Replicas* konfiguriert und es laufen drei *Pods*.

```
kubectl get deployment webapp-color-flux
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE	CONTAINERS
webapp-color-flux	3/3	3	3	144m	webapp-color

Nun erfolgt das manuelle Löschen des Deployments:

```
kubectl delete deployment webapp-color-flux
```

```
deployment.apps "webapp-color-flux" deleted
```

```
kubectl get deployment webapp-color-flux
```

```
No resources found in webapp-flux namespace.
```

Ein Abrufen der Seite liefert nur einen Fehler-Code 503, welcher durch das Default-Backend vom *Ingress* zur Verfügung gestellt, sobald ein Service nicht mehr erreichbar ist.

```
curl -vvv https://webapp.azure-cloud.la-cc.de/
```

```
.....
<html>
<head><title>503 Service Temporarily Unavailable</title></head>
<body>
<center><h1>503 Service Temporarily Unavailable</h1></center>
<hr><center>nginx</center>
</body>
</html>
.....
```

Der Zustand beim *Flux-Deployment* und der manuelle Eingriff, der ein Delta verursacht hat, wird von Flux im *Worst-Case* nach 30 Minuten erkannt. Das hängt davon ab, wann der letzte Reconcile-Intervall-Prozess gelaufen ist und wie das Intervall eingestellt ist.

5.5.3 Manuelle Änderung: Umgebungsvariable im Container manipulieren

In dem Anwendungsfall geht es darum, eine Umgebungsvariable innerhalb eines Containers zu manipulieren und einen zweiten Prozess zu starten, um die Auswirkung zu demonstrieren. In einem Container läuft die Python-Flask Anwendung, welche von der *Kubernetes-API* als *Pod* erkannt und konfiguriert wird. Der laufende Inhalt in dem Container in einem *Pod* wird dennoch von der *Kubernetes-API* nicht überwacht. Bei dem Anwendungsszenario wird die Annahme gemacht, dass der Kubernetes-Scheduler aufgrund von Ressourcenoptimierung die *Pod* nicht umplatziert.

Manuelle Änderung: Umgebungsvariable im Container manipulieren (CD-Deployment)

Es wird bei dem *CD-Deployment* eine interaktive Session mit dem Container gestartet. Danach wird eine Umgebungsvariable gesetzt, die die Hintergrundfarbe der Anwendung überschreibt. Bevor die Session gestartet wird, wird ein bestimmter Container *webapp-color-5f5cbdc594-ncxg* ausgesucht und die aktuelle Farbe abgefragt. Um den Unterschied und die Manipulation zu veranschaulichen, wird die Anwendung im Container zum 2. Mal gestartet und interpretiert dann die Umgebungsvariable. Dafür wird ein *Port-Forwarding* auf den Localhost vom ausführenden Client-Gerät durchgeführt.

```
curl -vvv https://webapp.azure-cloud.la-cc.de

....
<!doctype html>
<title>Hello from Flask</title>
<body style="background: #16a085;"></body>
<div style="color: #e4e4e4;
    text-align: center;
    height: 90px;
    vertical-align: middle;">

    <h1>Hello from webapp-color-5f5cbdc594-ncxgb!</h1>
</div>%
....
```

Die aktuelle Hintergrundfarbe ist *#16a085* und dies entspricht *green*.

```
##Starte interactive Session mit dem Container

kubectl exec -it webapp-color-5f5cbdc594-ncxgb -- /bin/sh

## Setze Umgebungsvariable
export APP_COLOR=red

## Starte die Anwendung zum 2. Mal
python3 app.py

## Aktiviere Port-Forwarding zum localhost:8081
kubectl port-forward webapp-color-5f5cbdc594-ncxgb 8081:8081
```

Die geänderte Hintergrundfarbe ist `#e74c3c` und dies entspricht *red*.

```
curl -vvv localhost:8081

....
<!doctype html>
<title>Hello from Flask</title>
<body style="background: #e74c3c;"></body>
<div style="color: #e4e4e4;
    text-align: center;
    height: 90px;
    vertical-align: middle;">

    <h1>Hello from webapp-color-5f5cbdc594-ncxgb!</h1>

* Closing connection 0
</div>%
....
```

Manuelle Änderung: Umgebungsvariable im Container manipulieren (Flux-Deployment)

Es wird bei dem *Flux-Deployment* eine interaktive Session mit dem Container gestartet. Danach wird eine Umgebungsvariable gesetzt, die die Hintergrundfarbe der Anwendung überschreibt. Bevor die Session gestartet wird, wird ein bestimmter Container `webapp-color-flux-76c7fb4f9d-g2sw` ausgesucht und die aktuelle Farbe abgefragt. Um den Unterschied und die Manipulation zu veranschaulichen, wird die Anwendung im Container zum 2. Mal gestartet und interpretiert dann die Umgebungsvariable. Dafür wird ein *Port-Forwarding* auf den Localhost vom ausführenden Client-Gerät durchgeführt.

```
curl -vvv https://webapp-flux.azure-cloud.la-cc.de

....
<!doctype html>
<title>Hello from Flask</title>
<body style="background: #e74c3c;"></body>
<div style="color: #e4e4e4;
    text-align: center;
    height: 90px;
    vertical-align: middle;">

    <h1>Hello from webapp-color-flux-76c7fb4f9d-g2sw7!</h1>
* Connection #0 to host webapp-flux.azure-cloud.la-cc.de left intact
</div>%
....
```

Die aktuelle Hintergrundfarbe ist `#e74c3c` und dies entspricht *red*.

```

##Starte interactive Session mit dem Container

kubectl exec -it webapp-color-flux-76c7fb4f9d-g2sw -- /bin/sh

## Setze Umgebungsvariable
export APP_COLOR=green

## Starte die Anwendung zum 2. Mal
python3 app.py

## Aktiviere Port-Forwarding zum localhost:8081
kubectl port-forward webapp-color-flux-76c7fb4f9d-g2sw 8081:8081

curl -vvv localhost:8081

....
<!doctype html>
<title>Hello from Flask</title>
<body style="background: #16a085;"></body>
<div style="color: #e4e4e4;
    text-align: center;
    height: 90px;
    vertical-align: middle;">

    <h1>Hello from webapp-color-flux-76c7fb4f9d-g2sw!</h1>

* Closing connection 0
</div>%
....

```

Die geänderte Hintergrundfarbe ist #16a085 und dies entspricht *green*.

6 | Ergebnisse

Die Ergebnisse stellen die Auswertung und Rückschlüssen aus den Anwendungsfällen der Arbeit dar. Diese beziehen sich auf verschiedene Gesichtspunkte. Dabei wird die Auswertung in mehrere Teile entsprechend der Anwendungsfälle unterteilt. Bei der Implementierung und der genauen Untersuchung zwischen den Unterschieden zwischen Flux und CI/CD und deren Vor- und Nachteilen ist stark aufgefallen, dass die genaue Unterscheidung zwischen dem Flux- und dem CI/CD-Ansatz auf dem *Deployment* Schritt liegt.

6.1 Anwendungsfall: Deployment CD vs Flux

Overhead und Life-Cycle-Management

Bei dem direkten Vergleich der beiden notwendigen Manifeste für ein *Deployment* der Web-Anwendung benötigt das *CD-Deployment* weniger Manifeste (siehe Abbildung 6.1 links) als das *Flux-Deployment* (siehe Abbildung 6.1 rechts). Das *CD-Deployment* benötigt nur drei Manifeste, um die Anwendung inklusive eines Clusterservices und einem *Ingress* bereitzustellen. Der *Flux-Stack* benötigt ebenfalls die drei Manifeste und zusätzlich wird noch ein *flux-kustomization* Manifest benötigt, der die Referenz auf den Ordner enthält und ein *kustomization* Manifest, welcher die einzusammelnden Ressourcen referenziert. Die Logik liegt im Gegensatz zum *Flux-Stack* bei dem *CD-Deployment* in der *Pipeline* und nicht den zuständigen Controllern. Dadurch gibt es bei der *CD-Deployment-Pipeline* kein richtiges *Life-Cycle-Management* der Anwendung. Die *Pipeline* führt nur eine Reihenfolge, mit Logik versehenen Befehle durch.



Abbildung 6.1: Deployment CD vs Flux Dateibaum, Bildquelle: eigene Darstellung

Qualität und Clean-Code¹, letzter Zugriff 01.08.2022

Durch das eingestellte Intervall, welches für die jeweilige Umgebung und *Stage* konfiguriert werden kann, sorgt Flux dafür, dass das Delta zwischen dem ausgerollten Zustand und dem im *Repository* eingeeckten Zustand synchronisiert wird. Damit die Einstellung pro Anwendung und pro *Stage* passieren kann, erfordert Flux eine Menge an redundanten Dateien. Es wird somit pro Anwendung und *Stage* mindestens eine *flux-kustomization* und eine *kustomization* benötigt. Ist das Intervall sehr klein

¹<https://clean-code-developer.de>

eingestellt, zum Beispiel auf 30- oder 60-Sekunden, dann ist ein manueller Eingriff so gut wie sinnlos. Dies zwingt anders als bei einem CD-Deployment, welches meistens durch ein Vier-Augen-Prinzip auf eine produktive Umgebung ausgerollt wird, die Entwicklerteams im Vorfeld sorgfältiger zu arbeiten. Ein *Push* auf den gemappten *Branch* zu der Produktiven-Umgebung sorgt für ein automatisches Deployment bei dem Flux-Ansatz. Dies sorgt aber auch gleichzeitig dafür, dass die Teams gezwungen werden alle Versionen im eingechekten *Repository* aktuell zu halten. Dadurch wird die Qualität des Deployments gesteigert und die Fehlerwahrscheinlichkeit reduziert. Die Fehler werden bei dem Flux-Ansatz durch den ständige *pullen* schneller erkannt als beim CD-Deployment. Beim CD-Deployment werden die Fehler erst beim nächsten *Triggern* der *Pipeline* erkannt und dies kann in Abhängigkeit vom Produkt und dem geführten Ansatz in den Entwicklerteams Wochen bis Monate ausmachen.

GitOps

Der GitOps-Ansatz ist bei Flux stärker ausgeprägt, da es nicht vorgesehen ist, dass Flux an die Variablen und *Secrets* aus den Variablengruppen ausliest. Deswegen wird alle notwendigen Konfigurationen und *Secrets* mit im *Repository* zum Beispiel als *Sealed-Secrets* mit eingechekkt. Bei dem *CD-Deployment* wird viel mit Variablen aus Variablengruppen gearbeitet, welche zur Laufzeit geladen und verarbeitet werden.

Historie und Logging

Die Art und Weise der Historie der Logs verändert sich zudem bei dem Flux-Ansatz. Es ist nicht mehr so einfach nachvollziehbar, wie eine Änderung ohne ein externes Monitoring ausgerollt worden ist und welche Ressourcen konfiguriert worden sind. Anders als beim *CD-Deployment* wird für jeden Schritt ein Log erstellt und es kann besser nachvollzogen werden, wann welche Änderung gemacht worden ist. Außerdem vereinfacht es beim *CD-Deployment* das *Debugging* für unerfahrene oder neu eingestiegenen Entwickler im Kubernetes-Bereich.

Build-Prozess

Der Build-Part bleibt sowohl beim *CD-Deployment* als auch beim Flux-Ansatz gleich, da sich nur die Art und Weise des Ausrollens einer Anwendung ändert. Der einzige Unterschied ist, dass Flux durch ein Rollback schneller reagiert, da sich nur die Commit-ID als Referenz im Vergleich zum *CD-Deployment* ändert, bei welchem eine *Pipeline* die benötigte Zeit durchlaufen muss.

Abhängigkeit

Das CD-Deployment wird als *Pipeline* auf einem Agenten entweder bei einem Service-Provider wie Azure-DevOps als Microsoft gehosteter CI/CD-Auftrag oder als selbst gehosteter CI/CD-Auftrag ausgeführt. Ein Agent wird in der Regel nicht nur dediziert von ein Projekt genutzt, sondern von vielen Projekten als geteilter Service, um die Ressourcen bestmöglich zu nutzen. Dies ist mit einer Queue und Wartezeit verbunden, bevor die *Pipeline* losläuft. Bei Flux besteht diese Abhängigkeit zwar nicht, dafür ist Flux auf die Stack-Komponenten wie den *Source-Controller* angewiesen. Dies kann dazu führen, wenn der *Node* auf dem der *Pod* läuft ein Netzwerk- oder Ressourcen-Problem hat, dass der *Pod* nicht ordnungsgemäß ausgeführt wird und dadurch ist Flux nicht in der Lage auf eine Änderung im *Repository* zu reagieren.

6.2 Manuelle Änderung: Skalierung

Bei dem Anwendungsfall erfolgte ein manueller Eingriff, durch das Hochskalieren der Anzahl der *Pods*. Der CD-Deployment-Ansatz bemerkt die Änderung bis zur Ausführung der nächsten *Pipeline* nicht.

Durch das Hochskalieren der *Pod* bei der betroffenen Anwendung führt es zu einer Verbesserung der Performance, da nun nach dem Round-Robin-Scheduling-Verfahren¹ die Anfragen anstatt auf drei Anwendungen auf fünf verteilt werden. Dies kann besonders hilfreich sein, wenn die Last auf die Anwendung unbekannt ist und eine Horizontale-Pod-Autoskalierung aufgrund von speziellen Metriken wie zum Beispiel Latenzen nicht durch bereitgestellten Metriken CPU- und RAM möglich ist. Der Flux-Ansatz bemerkte die Änderung in dem vordefinierten Intervall von 30 Minuten nach circa 16 Minuten und rollte diese Änderung von fünf *Replicas* wieder auf drei *Replicas* zurück. Dies kann zu einem Problem führen, falls die Performance der Anwendung durch zu viele Anfragen negativ beeinflusst wird, denn eine manuelle schnelle Anhebung zum Verbrenen der Last durch das Rollback von Flux nur für das Zeitintervall im *Worst-Case* möglich ist.

6.3 Manuelle Änderung: Deployment löschen

Der Anwendungsfall demonstriert ein versehentliches Löschen eines Deployments und somit das Verursachen einer *Downtime*. Es wird in der Annahme vernachlässigt, dass das Löschen eines Deployments in einer idealen Umgebung, welche von erfahrenen Kubernetes-Administratoren aufgesetzt wurde, durch *Policies* unterbindet. Das Löschen des Deployments verhält sich beim manuellen Eingriff ähnliche wie dem Anwendungsfall der Skalierung, nur dass die Folgen durch die *Downtime* eine negative Auswirkung auf die Anwendung haben. Der CD-Deployment-Ansatz bemerkt die Änderung bis zur Ausführung der nächsten *Pipeline* nicht. Der Flux-Ansatz bemerkt die Änderung in dem vordefinierten Intervall von 30 Minuten nach circa neun Minuten und rollt das *Deployment* erneut aus. Für eine bessere Ansicht wird im folgenden durch Visualisierung der Verfügbarkeit und dem Einsetzen von Monitoring der Anwendungsfall veranschaulicht. Die Abtastrate wurde wie folgt konfiguriert:

CD-Deployment:

```
Monitor-Typ: HTTP(s)
Anzeigename: webapp-color-cicd-avail
URL: https://webapp.azure-cloud.la-cc.de/
Pruefintervall: 30 Sekunden
Wiederholungen: 0
Erlaubte HTTP-Statuscodes: 200-299
```

Flux-Deployment

```
Monitor-Typ: HTTP(s)
Anzeigename: webapp-color-flux-avail
URL: https://webapp-flux.azure-cloud.la-cc.de/
Pruefintervall: 30 Sekunden
Wiederholungen: 0
Erlaubte HTTP-Statuscodes: 200-299
```

Der dazu verwendete Service und die Auswertung der Verfügbarkeit und deren Interpretation erfolgt im folgenden.

¹https://www.fernuni-hagen.de/mathinf/studium/pdf/Leseprobe-komplett_01802.pdf, letzter Zugriff 05.06.2022

Manuelle Änderung: Deployment löschen Verfügbarkeitsüberwachung

Es wurde ein Service *uptime-kuma*¹ eingerichtet, der die Verfügbarkeit von verschiedenen Protokollen ermöglicht. Durch den Service soll die Übersicht zwischen der Verfügbarkeit der beiden Deployments dargestellt werden. Die Deployments wurden fast zeitgleich mit einer Abweichung von +/- 1 Sekunde gelöscht. Die folgende Abbildung 6.2 stellt das Ergebnis einer Überwachung von circa 30 Minuten bis Flux in dem Intervall von 30 Minuten den Reconcile-Prozess durchgeführt hat.



Abbildung 6.2: Deployment CD vs Flux Dateibaum, Bildquelle: eigene Darstellung

Die Verfügbarkeit in der Abbildung 6.2 verdeutlicht die Vorteile von Flux. Das System stabilisiert sich schnell, trotz eines manuellen Fehlers selbst in einem Intervall von 30 Minuten. In dem Fall sogar bereits nach schon nach circa 9 Minuten (siehe Abbildung 6.2 *webapp-color-flux-avail*). Dieses Verhalten lässt sich ohne großen Konfigurationsaufwand durch einen höheren Verbrauch von Ressourcen und Senkung vom Intervall auf Sekunden stark optimieren. Da die Annahme bei der Implementierung gemacht wurde, dass es sich, um eine *Development-Stage* handelt, ist das Intervall von 30 Minuten vollkommen ausreichend. Beim *CD-Deployment* (siehe Abbildung 6.2 *webapp-color-cicd-avail*) bleibt das System down, bis ein *Monitoring* und *Alerting* einen Alarm rausschickt oder die Nutzer der Website den Fehler melden. Die meisten *CI/CD-Pipeline*s sind in der Regel so gebaut, dass die *CD-Pipeline* erst losläuft, wenn die *CI-Pipeline* erfolgreich durchgelaufen ist. Damit wird sichergestellt, dass keine fehlerhafte Anwendung, die beim *CI-Schritt* gescheitert ist, dann im *CD-Deployment-Schritt* ausgerollt wird. Je nach Größe der und Komplexität der *CI-Pipeline* kann der gesamte Vorgang 20–30 Minuten benötigen. Die *Downtime*, die dadurch entsteht, von der Meldung des Ausfalls bis hin zum erneuten *Deployment* ist somit relativ hoch. Wenn jetzt noch *SLAs* dran hängen, dann hat es negative Auswirkung auf die Verfügbarkeit des Dienstes. In einer idealen Umgebung, welche von erfahrenen Kubernetes-Administratoren aufgesetzt wurde, wird die Möglichkeit ein *Deployment* auf einer produktiven Umgebung durch *Policies* unterbunden.

¹<https://github.com/louislam/uptime-kuma>, letzter Zugriff 02.08.2022

6.4 Manuelle Änderung: Umgebungsvariable im Container manipulieren

In dem Anwendungsfall wurde eine Umgebungsvariable innerhalb eines Containers manipuliert und ein zweiter Prozess gestartet, um die Auswirkung zu demonstrieren. In einem Container läuft die Python-Flask Web-Anwendung, welche von der *Kubernetes-API* als *Pod* erkannt und konfiguriert wird. Der laufende Inhalt in dem Container in einem *Pod* wird dennoch von der *Kubernetes-API* nicht überwacht. Zwar ist ein *Pod* ein logischer Behälter für die darin laufenden Container, welche von der *Kubernetes-API* durch Interaktion mit dem *etcd-Key-Value-Store* konfiguriert werden, aber die darin laufende Anwendung wird von der *Kubernetes-API* nicht verwaltet. Dies hat zur Folge, dass alle Änderungen, die die Software in einem Container anbetreffen, von außen durch zum Beispiel das Setzen von Umgebungsvariablen manipuliert werden, aber nicht, wenn die Änderung direkt im Container selbst erfolgt. Aufgrund dieser Tatsachen operieren sowohl das *Flux-Deployment* als auch das *CD-Deployment* auf der Ebene der *Kubernetes-API* und nehmen keine direkte Veränderung im Container wahr. Deswegen läuft die Anwendung in dem *Use-Case*, solange mit den manipulierten Farbwerten durch das Setzen der Umgebungsvariable bis der *Pod* neu gestartet wird. Das Neustarten des *Pod* sorgt dafür, dass dieser mit den Werten wie im *Deployment-Manifest* beschrieben gestartet wird. Dass ein *Pod* neu gestartet wird, kann dennoch Minuten, Stunden oder Tage andauern und hängt von diversen Faktoren ab. Ein Faktor ist zum Beispiel, wenn der *Node* nicht genügend Ressourcen hat, dann wird der *Scheduler* aktiv und platziert die *Pod* zielgerichtet auf einen anderen *Node* und verursacht dadurch einen Neustart. Ein anderer Faktor ist, ein Release einer neuen Version, die die *Pod* rotiert und dadurch einen Neustart der Anwendungen erzwingt. Es ist keine direkte Schwäche von Flux oder einer *CI/CD-Pipeline*, sondern hängt viel mehr mit der implementierten Logik der Anwendung und der Konfiguration des *Pod* durch die *Kubernetes-API* zusammen. Eine mögliche Lösung, die das Problem beheben wird, ist die Implementierung einer Logik, die über einen Endpunkt die Ursprungsfarbe abfragt und falls diese abweicht, dann den *Pod* terminiert. An der Stelle wird schnell bewusst, dass die verwendete Anwendung nur Mittel zum Zweck ist und keine wirkliche sinnvolle Funktion abbildet.

Ein *Deployment* über eine *CI/CD-Pipeline* oder Flux steht zum Beginn der Automatisierung vor der Herausforderung, das Henne-Ei-Problem zu lösen. Dabei spielt es auch keine große Rolle, welchen Ansatz ein Team verfolgt. Der initiale Prozess, um die Systeme zu verknüpfen und das Setzen von Berechtigungen erfolgt teilweise manuell oder durch Skripte (Semi-Automatisiert). Ist die Grundstruktur vor dem Aufbau von *CI/CD-Pipelines* oder die Bereitstellung eines *Flux-Stacks* erfolgt, dann legt CI/CD den Schwerpunkt darauf, dass alle *Credentials* und alle notwendige Konfiguration in Variablen oder Dateien außerhalb der *Pipeline* liegen. Zwar können statische Werte gesetzt werden, aber das macht die *Pipelines* starr und erfordert einen enormen Konfigurationsaufwand, welcher mit einer Fehlerquote verbunden ist. Die Konfiguration liegt somit nicht im *Repository* selbst, sondern in dem System und der Umgebung des *Repositories* wie Azure DevOps. Dadurch wird eine Abhängigkeit zur Plattform geschaffen. Dies erfordert Vertrauen in den Plattformen-Betreiber oder bei *self hosted Services* in die interne IT. Im Gegensatz zum CI/CD verfolgt Flux den GitOps-Ansatz und versucht alles im *Repository* zu halten, darunter fallen auch sensible Daten wie *Credentials* zu weiteren Systemen. Dies erfordert kryptografische Verfahren, die meistens mit asynchroner Verschlüsselung abgebildet werden. Der Schlüssel zur Entschlüsselung muss ebenfalls wie bei CI/CD einem fremden System anvertraut werden. Es gibt die Möglichkeiten, dass Systeme selbst Schlüsselpaare generieren und den privaten Schlüssel von allen fernhalten. Die Anwender bekommen nur den öffentlichen Schlüssel, um sensible Daten zu verschlüsseln. Das *Deployment* und die Migration von Quellcode wird durch den GitOps-Ansatz zwar vereinfacht, aber dadurch kommt eine Komponente als *Black-Box* zum Einsatz dazu. Wenn es dann zu einem Problem kommt, dann erfordert es Zeit und tiefes Verständnis, um die Ursache für die das Problem zu finden. Bei CI/CD ist die Komplexität durch den Service und die bereits vorhandenen Logs wesentlich geringer. Außerdem wird die Logik von den Teams gebaut und lässt sich deshalb besser von diesen analysieren. Der GitOps-Ansatz erfordert, dass die Teams eine wesentlich bessere Arbeit nach den *Clean-Code* Prinzipien abliefern, da jeder Fehler, der im *Repository* eingecheckt wird, fatale Folgen für den Betrieb der Anwendungen haben kann. Dies kann am Anfang zum Beginn zum Frust bei den Teams führen. Dafür wird Abhängigkeit zum Fremdsystem wie Azure DevOps bei GitOps größtenteils abgelöst. Desto mehr Logik und Konfiguration in einem *Repository* vorhanden ist, desto weniger Abhängigkeit besteht zu der Plattform und dies ermöglicht eine einfache Migration im Vergleich zu reinem CI/CD-Konzept. Bei CI/CD muss berücksichtigt werden, ob die neue Plattform alle Features abbildet, die bei der aktuellen Plattform genutzt werden. Da der initiale *Flux-Stack* durch CI/CD ausgerollt wird und dann das Synchronisieren in einem fest definierten Intervall pro *Flux-Kustomization* übernimmt, besteht trotzdem eine Abhängigkeit zu der Pipelines-Plattform. Hinzu kommt, dass der initiale Aufbau des Unterbaus, sowohl für CI/CD als für GitOps mit Flux, mit sehr viel Konfigurationsaufwand und einer guten Knowledge-Base verbunden ist. Dies liegt daran, dass der hauptsächliche Unterschied zwischen Flux und CI/CD sich bei dem Deployment-Schritt der Anwendung ins Zielsystem unterscheidet. Der starke Unterschied bei CI/CD liegt in dem letzten Teil von CD, wobei es sich um *CD-Deployment* und nicht *Delivery* handelt. Das ist der Hauptunterschied zwischen dem GitOps-Ansatz unter Verwendung von Flux, welcher auf einem Pull-Mechanismus basiert. Deswegen kann der CI/CD-Teil bis hin zu *Delivery* vernachlässigt werden und für beide Ansätze als Basis verwendet werden. Dies ist bei der Ausarbeitung und vor allem bei der Implementierung im Rahmen der Masterarbeit bewusst geworden. Es muss außerdem unterschieden werden, dass Flux kein Synonym für GitOps ist. Dies wird zwar häufig im Kontext so verwendet, stellt dennoch nur eine Möglichkeit dar, den GitOps-Ansatz zu implementieren. Somit kann auch ein reiner

CI/CD-Ansatz auf GitOps basieren und ist nur abhängig von der Implementierung.

Ein großer Nachteil, der sich durch *Flux-Stack* ergibt, ist die Möglichkeit einer Multimandantenfähigkeit in einem geteilten Kubernetes-Cluster. Ein *Flux-Stack* steuert und kontrolliert in der Regel ein gesamtes Cluster und somit auch alle Namespaces und damit auch den Zugriff bei unterschiedlichen Teams und kann dadurch die Ressourcen in den anderen Namespaces über Flux manipulieren. Die Multimandantenfähigkeit wird vermutlich in Zukunft gelöst, da auch namhafte Unternehmen wie Microsoft den GitOps-Ansatz (Microsoft, 2022b) durch zusätzliche Konfigurationsmöglichkeiten in ihren *Managed-Kubernetes-Service* integrieren. Aktuell handelt es sich eher um einen Workaround, der ein Flux-Cluster pro Namespace-Block ermöglicht. Dies verbraucht dennoch pro *Flux-Stack* zusätzliche Ressourcen wie CPU, RAM und Storage. Diesen Nachteil kann ein reiner CI/CD-Ansatz durch die RBAC-Einschränkung der Rechte des verwendeten Tokens in der *Pipeline* gelöst werden. Somit eignet sich GitOps aktuell am besten, wenn ein Kubernetes-Cluster pro Team zur Verfügung gestellt wird und nicht viele Teams auf einem geteilten Cluster arbeiten. Außerdem ändert sich die Art- und Weise beim Staging zwischen den beiden Ansätzen. Denn CI/CD setzt den Schwerpunkt auf *Branch* pro *Stage*, während Flux einen Trunk-Based-Ansatz¹ verfolgt und die Stages auf Ordnerstrukturen verteilt. Des Weiteren ist der GitOps-Ansatz auf Flux überwiegend auf den Betrieb in einem Kubernetes-Cluster ausgerichtet und entfaltet an der Stelle das größte Potenzial. Diesen starken Bezug hat CI/CD nicht und kann deswegen vielseitiger verwendet werden, um auch zum Beispiel ein *Deployment* und die Konfiguration auf einer virtuellen Maschine durchzuführen.

Abschließend lässt sich sagen, dass die Kombination aus beiden Ansätzen, einen großen Vorteil für die Teams und Entwicklung bringt und den DevOps-Ansatz verstärkt. Es handelt es sich auch nicht, um die Frage ob CI/CD mit Pipelines oder GitOps mit Flux, sondern wie ergänzen sich die Teile am besten und wann ist es sinnvoll auf einen reinen CI/CD-Ansatz und wann auf ein Flux zu setzen. Des Weiteren kommt es auch stark auf die Erfahrung im Team an. Während CI/CD im DevOps eher für Anfänger geeignet ist, ist der GitOps-Ansatz mit Flux unter DevOps für kleinere Teams bis zur zehn Personen mit einer größeren Erfahrung in der Softwareentwicklung und Versionsverwaltung gedacht. Die Entwicklung der beiden Ansätze und damit auch die Auswirkung auf den kulturellen Wandel in den Unternehmen bleibt spannend und wird in den nächsten Jahren einen starken Einfluss auf den *IT-Stack* haben. Der DevOps-Kontext stellt somit die Kultur und das Mindset für ein Team bereit. Der CI/CD oder GitOps-Ansatz ergänzt den DevOps-Kontext um das notwendige Tooling.

¹<https://cloud.google.com/architecture/devops/devops-tech-trunk-based-development?>, letzter Zugriff 01.08.2022

Literaturverzeichnis

- 2022 Red Hat, Inc. (2020, Mai). *Was ist ein Kubernetes Operator?* Website. Zugriff am 17.08.2022 auf <https://www.redhat.com/de/topics/containers/what-is-a-kubernetes-operator>
- BSI. (2022, Jan). *Public key infrastrukturen (pkien)*. Website. Bundesamt für Sicherheit in der Informationstechnik. Zugriff am 05.07.2022 auf <https://www.bsi.bund.de/DE/Themen/Oeffentliche-Verwaltung/Elektronische-Identitaeten/Public-Key-Infrastrukturen/public-key-infrastrukturen.html>
- HashiCorp. (2022, April). *What is Terraform?* Website. Zugriff am 16.09.2022 auf <https://www.terraform.io/intro>
- Microsoft. (2022a). *Konzepte – Grundlagen zu Kubernetes für azure kubernetes service (AKS) - azure kubernetes service*. Website. Zugriff am 25.07.2022 auf <https://docs.microsoft.com/de-de/azure/aks/concepts-clusters-workloads>
- Microsoft. (2022b, April). *Tutorial: Verwenden von GitOps mit Flux v2 in Kubernetes-Clustern mit Azure Arc-Unterstützung oder AKS-Clustern (Vorschau)*. Website. Zugriff am 01.09.2022 auf <https://docs.microsoft.com/de-de/azure/azure-arc/kubernetes/tutorial-use-gitops-flux2>
- Microsoft. (2022c, April). *Was ist Azure Pipelines?* Website. Zugriff am 16.09.2022 auf <https://docs.microsoft.com/de-de/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops>
- Microsoft. (2022d). *Was ist cloud computing? Leitfaden für Einsteiger: Microsoft Azure*. Website. Zugriff am 21.06.2022 auf <https://azure.microsoft.com/de-de/overview/what-is-cloud-computing/#cloud-computing-models>
- Red Hat Inc. (2018, Apr). *Was sind microservices?* Website. Zugriff am 23.05.2022 auf <https://www.redhat.com/de/topics/microservices/what-are-microservices>
- Red Hat, Inc. (2020, Januar). *Vergleich zwischen Containern und VMs*. Website. Zugriff am 23.05.2022 auf <https://www.redhat.com/de/topics/containers/containers-vs-vms>
- The Kubernetes Authors. (2021a, June). *Controllers*. Website. Zugriff am 02.06.2022 auf <https://kubernetes.io/docs/concepts/architecture/controller/>
- The Kubernetes Authors. (2021b, March). *Control Plane-Node Communication*. Website. Zugriff am 01.07.2022 auf <https://kubernetes.io/docs/concepts/architecture/control-plane-node-communication/>
- The Kubernetes Authors. (2021c, April). *Kubernetes Architektur*. Website. Zugriff am 02.06.2022 auf https://kubernetes.io/de/docs/concepts/architecture/_print/
- The Kubernetes Authors. (2021d, December). *Kubernetes Scheduler*. Website. Zugriff am 02.06.2022 auf <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>
- The Kubernetes Authors. (2021e, Apr). *Nodes*. Website. Zugriff am 01.07.2022 auf <https://kubernetes.io/de/docs/concepts/architecture/nodes/>
- The Kubernetes Authors. (2022a, Januar). *Cloud Controller Manager*. Website. Zugriff am 02.06.2022 auf <https://kubernetes.io/docs/concepts/architecture/cloud-controller/>
- The Kubernetes Authors. (2022b, Januar). *DNS for Services and Pods*. Website. Zugriff am 03.07.2022 auf <https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/>
- The Kubernetes Authors. (2022c, March). *Example: Deploying PHP Guestbook application with Redis*. Website. Zugriff am 02.06.2022 auf <https://kubernetes.io/docs/tutorials/stateless-application/guestbook/>

- Albarqi, A., Alzaid, E., Alghamdi, F., Asiri, S. & Kar, J. (2015, 01). Public Key Infrastructure: A Survey. *Journal of Information Security*, 06, 31-37. doi: 10.4236/jis.2015.61004
- al Jawarneh, I., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G. & Palopoli, A. (2019, 05). Container Orchestration Engines: A Thorough Functional and Performance Comparison. In (S. 1-6). doi: 10.1109/ICC.2019.8762053
- Bendel, O. (2021, 07). *Definition: Digitalisierung*. Website. Springer Fachmedien Wiesbaden GmbH. Zugriff am 13.06.2022 auf <https://wirtschaftslexikon.gabler.de/definition/digitalisierung-54195>
- Betsy Beyer, J. P. N. R. M., Chris Jones. (2016). *Site Reliability Engineering: How Google Runs Production Systems* (1. Aufl.). O'Reilly Media.
- Billy, Y., Alexander, M., Todd, E. & Jesse, S. (2021). *GitOps and Kubernetes: CONTINUOUS DEPLOYMENT WITH ARGO CD, JENKINS X, AND FLUX*. Manning Publications Co.
- Bitnami Labs. (2022). *Sealed Secrets for Kubernetes*. Website, Github. Zugriff am 15.08.2022 auf <https://github.com/bitnami-labs/sealed-secrets>
- Böhlke, L. (2021, Oct). *Continuous integration*. Website. mindsquare AG. Zugriff am 13.06.2022 auf <https://mindsquare.de/knowhow/continuous-integration/>
- Carvalho, B., Henrique, C. & Mello, C. (2011, 01). Scrum agile product development method - literature review, analysis and classification. *Product: Management Development*, 9, 39-49. doi: 10.4322/pmd.2011.005
- Chacon, S. & Straub, B. (2014). *Pro git: Everything you need to know about Git* (Second, Version 2.1.338-2-g8a81047, 2022-04-05 Aufl.). Apress. Zugriff auf <https://git-scm.com/book/en/v2>
- Dang, W. L. & Kohgadai, A. (2021). *DevSecOps in Kubernetes*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media.
- Elektronik-Kompendium.de. (2022). *DNS - Domain Name System*. Website. Autor. Zugriff am 06.07.2022 auf <https://www.elektronik-kompendium.de/sites/net/0901141.htm>
- Gai, K. & Li, S. (2012, Nov). Towards Cloud Computing: A Literature Review on Cloud Computing and Its Development Trends. In *2012 fourth international conference on multimedia information networking and security* (S. 142-146). doi: 10.1109/MINES.2012.240
- Heath, F. (2021). *The professional scrum master (PSM I) guide*. Packt Publishing Ltd.
- Jangda, A., Pinckney, D., Brun, Y. & Guha, A. (2019, oct). Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3 (OOPSLA). Zugriff auf <https://doi.org/10.1145/3360575> doi: 10.1145/3360575
- Jha, P. & Khan, R. (2018, 06). A Review Paper on DevOps: Beginning and More To Know. *International Journal of Computer Applications*, 180, 16-20. doi: 10.5120/ijca2018917253
- KernelNewbies. (2017, Dezember). *Linux_2_6_32 - Linux Kernel Newbies*. Website. Zugriff am 22.05.2022 auf https://kernelnewbies.org/Linux_2_6_32
- KitelyTech. (2021, Oct). *The six basic steps of software development by KitelyTech*. Website. Zugriff am 13.06.2022 auf <https://kitelytech.com/six-basic-steps-software-development/>
- Kniberg, H. & Ivarsson, A. (2022). *Scaling Agile @ Spotify*. Whitepaper. Zugriff am 10.07.2022 auf <https://blog.crisp.se/wp-content/uploads/2012/11/SpotifyScaling.pdf>
- Kubernetes Sigs. (2022). *ExternalDNS*. Website, Github. Zugriff am 16.09.2022 auf <https://github.com/kubernetes-sigs/external-dns>
- LeBris, E. (2021, Jul). *Three differences between DevOps and SRE*. Website. Zugriff am 26.07.2022 auf <https://www.ibm.com/cloud/blog/three-differences-between-devops-and-sre>
- letsencrypt.org. (2019, October). *Wie es funktioniert*. Website. Zugriff am 06.07.2022 auf <https://letsencrypt.org/de/how-it-works/>

- letsencrypt.org. (2022). *Über let's Encrypt*. Website. Zugriff am 06.07.2022 auf <https://letsencrypt.org/de/about/>
- Liebel, O. (2019). *Skalierbare Container-Infrastrukturen - Das Handbuch für Administratoren* (2. Aufl.). Bonn: Rheinwerk Verlag.
- Manvi, S. & Shyam, G. K. (2021). *Cloud computing concepts and technologies*. CRC Press.
- Moreira, M. E. (2017). *The Agile Enterprise: Building and Running Agile Organizations*. doi: 10.1007/978-1-4842-2391-8
- Nohe, P. (2019, Jun). *The Difference Between Root Certificates and Intermediate Certificates*. Website. The SSL Store™. Zugriff am 05.07.2022 auf <https://www.thesslstore.com/blog/root-certificates-intermediate/>
- Nollau, S. (2018, Jan). *Hohe fluktuation unter it-fachkräften*. Website. IT-BUSINESS. Zugriff am 13.06.2022 auf <https://www.it-business.de/hohe-fluktuation-unter-it-fachkraeften-a-681654/>
- Pahl, C., Brogi, A., Soldani, J. & Jamshidi, P. (2017, 05). Cloud Container Technologies: a State-of-the-Art Review. *IEEE Transactions on Cloud Computing*, PP, 1-1. doi: 10.1109/TCC.2017.2702586
- Perera, P., Silva, R. & Perera, I. (2017, Sep.). Improve software quality through practicing DevOps. In *2017 seventeenth international conference on advances in ict for emerging regions (icter)* (S. 1-6). doi: 10.1109/ICTER.2017.8257807
- Red Hat, I. (2019, Dec). *Was ist container-orchestrierung?* Website. Zugriff am 25.05.2022 auf <https://www.redhat.com/de/topics/containers/what-is-container-orchestration>
- Red Hat, I. (2020, Mai). *Was ist container-orchestrierung?* Website. Zugriff am 26.05.2022 auf <https://www.redhat.com/de/topics/devops/what-is-sre>
- Ristic, I. (2014). *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. Feisty Duck.
- Ruth, M. (2022). *Das Spotify-Modell: Atlassian*. Website. Zugriff am 27.06.2022 auf <https://www.atlassian.com/de/agile/agile-at-scale/spotify>
- Röß, S. & Guttenberger, M. (2018, 09). Agile Softwareentwicklung mit Scrum und Kanban.
- Shahin, M., Ali Babar, M. & Zhu, L. (2017, 03). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, PP. doi: 10.1109/ACCESS.2017.2685629
- Sutherland, J. (2021, Mar). *THE SCRUM PAPERS: NUT, BOLTS, AND ORIGINS OF AN AGILE FRAMEWORK*. Website, PDF. Zugriff am 01.06.2022 auf <http://jeffsutherland.org/scrum/scrumpapers.pdf>
- Takeuchi, H. & Nonaka, I. (1986). The New New Product Development Game. *Harvard Business Review*.
- Varghese, B., Leitner, P., Ray, S., Chard, K., Barker, A., Elkhatib, Y., ... Zhani, M. (2019, Sep.). Cloud Futurology. *Computer*, 52 (9), 68-77. doi: 10.1109/MC.2019.2895307
- Vayghan, L. A., Saied, M. A., Toeroe, M. & Khendek, F. (2019). Kubernetes as an Availability Manager for Microservice Applications. *CoRR*, abs/1901.04946. Zugriff auf <http://arxiv.org/abs/1901.04946>
- VMware, I. (2022). *What is container networking?* Website. Zugriff am 06.07.2022 auf <https://www.vmware.com/topics/glossary/content/container-networking.html>

A | Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Dortmund, 29. September 2022

B | Disclaimer

Logos, Marken-, Produkt- und Warenzeichen Alle in dieser Seminararbeit veröffentlichten Logos sowie Marken-, Produkt- und Warenzeichen sind Eigentum der jeweiligen Unternehmen und Organisationen.

Alle Informationen in dieser Seminararbeit sind nach bestem Wissen und Gewissen zusammengestellt. Die Überlassung der Seminararbeit erfolgt nur für den internen Gebrauch des Empfängers.

Die verwendeten Bilder/Icons sind lizenzfreie Bilder, welche unter der CC/CC0/Pixabay-Lizenz veröffentlicht worden sind. Alle weiteren Bilder sind direkt unter dem Bild mit einem Quellenverweis vermerkt worden.

Dortmund, 29. September 2022