Fachhochschule Bielefeld
Campus Minden
Master Informatik
Spezielle Gebiete zum Software Engineering

Evaluation der GitOps-Toollandschaft

Leon Brandt
(Matr.-Nr.: 1151305)

30.06.2022

Abstract

Das Betriebsmodell GitOps kombiniert Infrastructure as Code (IaC) mit Continuous Deployment (CD). Es ist durch eine Reihe an Tools zu implementieren. Diese Arbeit untersucht eine Auswahl relevanter Technologien innerhalb der GitOps-Toollandschaft hinsichtlich ihrer Genügung von GitOps als Prinzip sowie hinsichtlich ihres Beitrags zu dem Nutzen von GitOps. Hierzu wird eine fachlich reduzierte Fallstudie unter Nutzung verschiedener Tools implementiert. Anhand dieser Implementierung werden Beobachtungen und Untersuchungen hinsichtlich Produkt- und Gebrauchsqualität von Tools und Ansätzen angeführt. Es zeigt sich eine umfassende Genügung der untersuchten Tools hinsichtlich GitOps als Betriebsmodell. Es lässt sich ein ausgereifter Technologiestack formulieren.

Inhaltsverzeichnis

1 Einleitung

Cloud Native ist ein Ansatz, der durch Technologien ermöglicht "skalierbare Anwendungen in modernen, dynamischen Umgebungen zu implementieren und zu betreiben" [1]. Die Cloud Native Computing Foundation (CNCF) benennt hier konkret unter Anderem Container, immutable Infrastruktur und deklarative APIs als Best Practices. Das Einsetzen von Cloud Native Technologien ermöglicht "die Umsetzung von entkoppelten Systemen, die belastbar, handhabbar und beobachtbar sind" [1]. Unter Implementierung einer "robusten Automatisierung können Softwareentwickler mit geringem Aufwand flexibel und schnell auf Änderungen reagieren" [1].

Continuous Deployment (CD) ist die Praxis des automatischen und kontinuierlichen Bereitstellens von Software. Sie erweitert die Praxis der Continuous Integration (CI), also des kontinuierlichen Integrierens von Software und die Praxis der Continuous Delivery (CDE), also des kontinuierlichen Auslieferns von Software. CDE hat zum Ziel, Software dauerhaft in einem produktionsfertigen Zustand zu halten, während CD eine automatische Bereitstellung beinhaltet. [2]

Beetz et. al. beschreiben GitOps als "Cloud-Native Continuous Deployment" [3]. GitOps kann also als die Umsetzung von Continuous Deployment unter Berücksichtigung von Cloud Native verstanden werden. Der Begriff GitOps wurde 2017 durch Alexis Richardson geprägt [4] und später von Kelsey Hightower mit den Worten "GitOps: versioned CI/CD on top of declarative infrastructure" beschrieben. [3]

GitOps kann als Betriebsmodell für Cloud Native Technologien verstanden werden. Es beschreibt auch einen Weg zur Herstellung von Developer Experience bei der Verwaltung von Software. Dies geschieht durch das Übertragen der Nutzung von Pipelines zum Continuous Deployment und Git-Workflows von der Anwendungsentwicklung in den Betrieb dieser. [4]

GitOps bietet operationellen und damit geschäftlichen Nutzen. [3, 4, 5, 6] Ziel dieser Arbeit ist, eine Auswahl relevanter Werkzeuge innerhalb der GitOps-Toollandschaft hinsichtlich ihrer Genügung von GitOps als Prinzip sowie hinsichtlich ihres Beitrags zu dem Nutzen von GitOps zu untersuchen.

1.1 Prinzipien von GitOps

Anforderung für das Einsetzen von GitOps als Betriebsmodell ist die Tatsache, dass die gesamte Infrastruktur deklarativ beschrieben ist. Konfiguration wird also durchgeführt durch die Definition von Fakten und nicht durch Spezifikation einer Menge von Instruktionen. [4] Dieser Ansatz wird auch als Infrastructure as Code (IaC) bezeichnet. [7] Teil von GitOps ist weiterhin der Einsatz von Version Control Systems (VCS) wie Git. Hierbei dienen Repositories als "single source of truth", also als zentrale Stelle für die Verwaltung von deklarativen Definitionen. [4, 7]

Änderungen werden hierbei an Definitionen vorgenommen und automatisch auf den Systemzustand angewendet. [4, 7] Dies geschieht durch Softwareagenten, die den Soll-Zustand mit dem Ist-Zustand abgleichen und notwendige Schritte zur Herstellung des Soll-Zustand

einleiten. Ein solches Vorgehen ist nicht auf das Anwenden von Änderungen beschränkt. So können Infrastrukturen permanent überwacht werden und somit die Eigenschaft des "self-healing" erlangen. [4]

Im Oktober 2021 formulierte die GitOps Working Group der CNCF durch das Projekt OpenGitOps vier Prinzipien von GitOps. Auch hier wird eine deklarative Definition des Soll-Zustands eines Systems beschrieben. Weiterhin wird eine Versionierung dieser Definitionen gefordert. Auch wird, wie bereits durch vorhergehende Literatur formuliert, eine kontinuierlicher Abgleich des tatsächlichen Systemzustands mit dem geforderten beschrieben. Bei Abweichungen ("Drift") wird der Versuch unternommen, den geforderten Systemzustand automatisch herzustellen ("Reconciliation"). Zusätzlich zu diesen Prinzipien wird zur Reconciliation gefordert, dass Agents den Soll-Zustand von Systemen selbstständig abrufen ("pull"). [8, 9]

Weaveworks, Vorreiter mit prägenden Einfluss auf GitOps als Methodologie, formulieren ein Reifegradmodell. Dieses beschreibt den Grad der Anpassung an und Umsetzung von GitOps in drei Stufen. Als Voraussetzung für GitOps wird Stufe 0 "Prequisites for GitOps" beschrieben. Diese beschreibt das Nutzen von IaC-Werkzeugen zur Bereitstellung von Infrastruktur und den Einsatz von CI und CD zur Auslieferung von Software. Die Stufe 1 "Core GitOps" entspricht dem Umsetzen der bereits genannten Prinzipien von GitOps. Nach Weaveworks finden auf dieser Stufe Teile der Prinzipien häufig nur für die Bereitstellung von Software nicht jedoch für Infrastruktur Anwendung. Stufe 2 "Enterprise GitOps" geht aus der Anforderung hervor, zahlreiche Deployments in unterschiedlichen Umgebungen wie Test- und Produktivumgebungen vorzunehmen. Hierbei kommen also die Prinzipien für die Konfiguration von Infrastruktur zur Erzeugung mehrerer Umgebungen zum Einsatz. Stufe 3 "GitOps at Scale" geht aus Anforderungen von vornehmlich Telekommunikationsprovidern hervor. Diese müssen bis zu tausende Edge-Umgebungen konfigurieren und verwalten. Diese Stufe kann für alle Unternehmen, die Deployments in verschiedenen Regionen oder bei unterschiedlichen Cloudprovidern durchführen von Relevanz sein. [5]

	Umsetzungsgrad
Level 0	- Deklarative Systembeschreibung - Versionskontrolle - Provisioning der Auslieferung
Level 1	- Deklarative Infrastruktur ohne Reconciliation - Deklaratives Deployment mit Reconciliation - Pull-Based Ansatz - Paketmanagement von Deployments
Level 2	- GitOps für komplette Umgebung (mit Reconciliation) - ...

Tabelle 1: Übersicht über die Stufen im Reifegradmodell für GitOps (Abwandlung nach [5])

Tabelle 1 listet zusammenfassend die Hauptaspekte der Level 0 bis 2 des Reifegradmodells auf.

1.2 Nutzen von GitOps

Für Unternehmen wie Amazon, Google, Facebook und Netflix sind IT-Systeme nicht kritisch und entscheidend für das Geschäft sondern sie sind das Geschäft selbst. Es ist nur plausibel, dass diese Unternehmen Vorreiter für neue Praktiken beim Betrieb von skalierbaren und hochverfügbaren Systemen sind. In solchen Umgebungen mit keiner Toleranz für Ausfälle hat sich Infrastructure as Code bewährt. [7]

In der Literatur werden zahlreich operationelle und geschäftliche Nutzen von GitOps diskutiert. Die GitOps Working Group der CNCF [10] und Weaveworks [4] formulieren die selben Nutzen aus der Umsetzung von GitOps. Beetz et al. bestätigen diesen Nutzen zum Teil. [3] Das bereits erwähnte Reifegradmodell von Weaveworks listet zahlreiche weitere Vorteile auf. [5] Morris verfolgt einen technischen und operationellen Blickwinkel auf den Nutzen von GitOps, argumentiert jedoch ähnlich. [7] Yuen et. al. fügen sich in das allgemeine Bild ein. [11]

Das DevOps Research and Assessment (DORA) Team der Google Cloud benennt vier Metriken zur Bewertung von Auslieferung und Betrieb von Software. [12] Ein Umsetzen von GitOps kann helfen, diese Metriken zu verbessern. [13] In den folgenden Abschnitt wird zusätzlich Bezug auf diese Metriken genommen. Dieser Abschnitt soll einen strukturierten Überblick über einen möglichen Konsens hinsichtlich des Nutzen von GitOps bieten.

1.2.1 Produktivität und Effizienz

GitOps erhöht die Produktivität und Effizienz von Teams. [10, 4, 3, 5, 7, 11] Dieser Effekt zeigt sich in einem schnelleren und häufigeren Deployment. [4, 3, 5] Diese Verbesserung geht auch aus einer besseren Repeatability [5, 11] oder auch Reproducability [7] hervor. So können häufig zu wiederholende Aufgaben einfach skaliert werden. [7, 11] Messbar wird dieser Effekt durch Metriken wie die Deployment frequency, die Lead time for Changes [12] sowie die Mean time to deployment [13]. Bei der Deployment frequency sowie der Lead time for Changes sieht Google ein Gefälle von einem 6-Monats-Rhythmus oder schlechter bei schlecht performenden Teams hinzu einem nahezu stündlichen Rhythmus bei optimal performenden Teams. [12]

1.2.2 Stabilität und Zuverlässigkeit

GitOps erhöht die Stabilität und Zuverlässigkeit von Deployments. [10, 4, 3, 5, 7, 11] Dies geht aus einer Verbesserung von Error Prevention [11], Error Detection [13, 11] sowie Error Recovery [4, 3, 5, 11] hervor.

Eine proaktive Error Prevention kann durch ein Durchführen von Code Reviews der deklarativen Infrastrukturbeschreibungen erreicht werden. So kann sichergestellt werden, dass nur geprüfte Änderungen in den Produktivbetrieb übernommen werden. Weiterhin verringert eine Automatisierung von Prozessen die Chance für menschliche Fehler. [11] Messbar wird dieser Effekt durch die Metrik Change failure rate. [12]

Deklarativ beschriebene Systeme implizieren die Eigenschaft, dass der Soll-Zustand einsehbar ist. Diese Eigenschaft beschreiben Yuen et. al. als Observability. [11] Eine hohe Observability ermöglicht eine effektive Error Detection. Messbar wird dieser Effekt durch Metriken wie Mean time to detect (MTTD) und Mean time to locate (MTTL). [5, 13]

Eine Versionierung durch zum Beispiel Git impliziert die Existenz eines vollständigen Audit Logs. Dies erhöht die Traceability, da Informationen darüber existieren, wer weshalb, wann welche Änderung vorgenommen hat. [5, 7] Weiterhin erlaubt eine Versionierung das Vornehmen von Rollbacks. Dies ermöglicht eine einfache, effiziente und schnelle Error Recovery. [4, 3, 5, 7, 11] Messbar wird dieser Effekt durch Metriken wie Mean time to recovery (MTTR) [4, 5] sowie Time to restore service [12].

Zuletzt wird Infrastruktur unter GitOps so ausgelegt, dass sie ständiger Änderung unterliegt. Dies erhöht die Disposability von Komponenten oder ganzen Systemen. Diese können also einfach erzeugt, entfernt, ausgetauscht oder verschoben werden. Dieses Paradigma erhöht die Fehlertoleranz und somit Zuverlässigkeit. [7]

Teil der Definition von GitOps ist eine automatische Reconciliation. [8] Dies bedeutet, dass der tatsächliche Systemzustand immer dem definierten Systemzustand entspricht oder mindestens in Richtung des definierten Systemzustand konvergiert. Bereits das erhöht bereits die Stabilität und Zuverlässigkeit. [5]

1.2.3 Developer Experience

GitOps erhöht die Developer Experience. [10, 4] Es basiert auf existierenden und bekannten Workflows wie zum Beispiel Pull-Requests. [4, 3, 5, 11] Software und Deployments werden also nach ähnlichen Prinzipien entwickelt. Dies erhöht nicht nur die Effizienz [5, 11] sondern macht auch das Erlernen notwendiger Fähigkeiten sowie das Onboarding neuer Mitarbeiter einfacher [4].

1.2.4 Konsistenz und Standardisierung

GitOps erhöht die Konsistenz und den Grad der Standardisierung. [10, 4, 7, 11] Wie bereits unter Developer Experience erwähnt bietet GitOps einen einheitlichen, konsistenten Workflow für die Entwicklung sowie die Bereitstellung von Software. [4] Zudem kann durch Reproducability ähnliche Infrastruktur mit geringen Abweichungen in der Konfiguration konsistent erzeugt werden. Dies gewährt eine gewisse Zusicherung, dass bestehende Prozesse und Automatisierung auf dieser ebenfalls funktionieren. [7] Dies wird weiterhin durch Code Reviews gefördert, da hier Richtlinien durchgesetzt werden können. [11]

1.2.5 Sicherheit

GitOps erhöht Sicherheit. [10, 4, 3, 5] Systemen zur Versionskontrolle wie Git bieten die Möglichkeit Änderungen zu signieren um dessen Herkunft nachzuweisen. [4, 5] Weiterhin werden Deployments vollständig aus einer Umgebung heraus verwaltet. Externer Zugriff muss lediglich auf das Repository bestehen. Einzelne Entwickler benötigen keinen direkten Zugriff auf die Infrastruktur. [3]

1.2.6 Selbst-Dokumentation

GitOps sorgt dafür, dass Systeme selbst-dokumentiert sind. [3, 5, 7, 11] In Zusammenhang mit der bereits beschriebenen Eigenschaft der Observability stellt eine versionierte, deklarative Beschreibung des Soll-Zustands eines Systems sicher, dass dieses selbst dokumentiert ist. [3, 5, 7, 11] Jede Änderung ist für jeden Entwickler sichtbar. Die daraus entstehende Audability [5], Traceability sowie Visibility [7] erhöhen das Bewusstsein bei jedem Entwickler für zuletzt durchgeführte Änderungen und helfen, mögliche durch diese erzeugte Fehler zu erkennen. [7]

1.3 GitOps-Toollandschaft

Morris identifiziert zur Umsetzung von GitOps als Betriebsmodell vier Klassen von Tools. [7] Die CNCF formuliert mit der stetig aktualisierten "Cloud Native Interactive Landscape" eine Übersicht von über 1100 konkreten Cloud Native Tools und Technologien inklusive einer Klassifizierung dieser. [14] Dieser Abschnitt soll einen Überblick über die GitOps-Toollandschaft sowie Arten und Klassen von Tools und Technologien bieten.

1.3.1 Dynamische Infrastrukturplattformen

Morris benennt die Notwendigkeit von dynamischen Infrastrukturplattformen. Als eine solche sind Systeme zu verstehen, die Rechen-, Speicher- und Netzwerkressourcen zur Verfügung stellen. Diese müssen programmierbar sein, damit weitere Tools mit diesen interagieren können. Konkret muss also eine API zur Verfügung stehen. Die Art der Infrastrukturplattform spielt keine Rolle. So können Bare-metal Clouds sowie Public oder Private Infrastructure as a Service (IaaS) Clouds zum Einsatz kommen. [7]

Die CNCF beschreibt zwar auch eine Klasse "Platform". Hier werden jedoch keine IaaS-Technologien betrachtet. Vielmehr Platform as a Service (PaaS) oder Container as a Service (CaaS) Provider wie zertifizierte Kubernetes-Hostingprovider. [14]

1.3.2 Tools zur Definition von Infrastruktur

Weiterhin sind Tools zur Definition von Infrastruktur notwendig. Diese ermöglichen die Spezifikation der durch Infrastrukturplattformen bereitzustellende Infrastruktur sowie deren Konfiguration. Infrastruktur-Definitionstools implementieren die durch IaC formulierte Spezifikation von Infrastruktur. Nicht alle dieser Tools verfolgen nativ IaC-Ansätze. Morris beschreibt für diese Klasse an Werkzeugen drei Anforderungen. Zuerst benötigen Infrastruktur-Definitionstools eine programmierbare Schnittstelle. Hiermit sind vor allem auch Commandline-Interfaces (CLI) gemeint. Morris argumentiert, dass solche den Nutzern eine tiefere und direktere Kontrolle über alle Details der Bedienung ermöglichen. Weiterhin müssen diese Werkzeuge unbeaufsichtigt ausführbar sein. Im Fall von CLI-basierten Tools wird eine Ausführung durch Scripte durch geeignete Input- und Output-Schnittstellen ermöglicht. Diese Schnittstellen sollten eine Interaktion mit weiteren Tools ermöglichen. Die letzte Anforderung an Tools zur Definition von Infrastruktur ist eine Externalisierung von Konfiguration. Dies bedeutet, dass Konfigurationsdaten nicht intern durch ein Tool verwaltet werden sollen, sondern diese extern Verfügbar sind und durch viele mögliche Tools bearbeitet werden können. [7]

Die CNCF gruppiert in der Oberklasse "Provisioning" Klassen von Tools, mit denen die Basis von Cloud Native Anwendungen gebildet werden. Innerhalb dieser existiert die Klasse "Automation & Configuration". Tools innerhalb dieser sind entsprechend zu Morris damit definiert, dass diese ein Erzeugen und Konfigurieren von Infrastruktur-Ressourcen ermöglichen. [15]

1.3.3 Tools zur Konfiguration von Servern

Als dritte Toolklasse zur Umsetzung von GitOps benennt Morris Tools zur Konfiguration von Servern. Auch diese sollen einen IaC-Ansatz verfolgen. Konfiguration wird also durch gegebenenfalls Domain Specific Languages (DSL) spezifiziert und durch Konfigurations-Tools angewendet und implementiert. Weiterhin können bei diesen Tools zwei Ansätze unterschieden werden. Bei einem Pull-Ansatz wird ein Agent auf dem zu konfigurierenden Server ausgeführt, der periodisch Konfigurationsdefinitionen abruft und auf den Server anwendet. Beim Push-Ansatz wird das Anwenden von Konfigurationen durch Befehle via zum Beispiel SSH-Verbindungen extern ausgelöst. Morris ordnet dieser Klasse an Tools außerdem Technologien zur Containerisierung sowie Ausführung, also Orchestrierung von Containern zu. Letztere verwalten Container zur Laufzeit. Sie starten, ersetzen, skalieren und überwachen Container verteilt auf eine Menge an Hosts. [7]

Die CNCF fasst Tools zur Anwendung von Konfigurationsspezifikationen ebenfalls in der Klasse "Automation & Configuration" zusammen, in der auch Infrastruktur-Definitionstools einzuordnen sind. Weiterhin wird der Oberklasse "Runtime" die Klasse "Container Runtime" untergeordnet. Diese fasst Laufzeitumgebungen zur Ausführung von Containern zusammen. In der Oberklasse "Orchestration & Management" werden alle Toolklasse zur Ausführung von Cloud Native Anwendungen zusammengefasst. Innerhalb dieser existiert die Klasse "Scheduling & Orchestration". Analog zur Definition von Morris sind in dieser Klasse Tools zusammengefasst, die ein Management von Containern über ein Cluster von Hosts vornehmen. [15]

1.3.4 Allgemeine Infrastruktur-Tools

Als allgemeine Infrastruktur-Tools bezeichnet Morris alle weiteren notwendigen Tools zur Umsetzung von GitOps. Hierunter fallen zum Beispiel DNS, Monitoring-Tools oder Datenbanken und Message-Queues. Auch enthält diese Klasse Deployment Pipeline Software, also Tools zur Implementierung von CI und CD sowie Packaging Software. [7]

Die CNCF benennt zahlreiche weitere Klassen in höherem Detailgrad als Morris. Die Oberklasse "App Definition & Development" enthält alle Klassen von Tools, die Entwicklern das erzeugen von Cloud Native Anwendungen ermöglichen. In der dort untergeordneten Klasse "Application Definition & Image Build" befinden sich Tools, die in etwa Morris' Definition von Packaging Software entsprechen. Diese werden hier weiter in zwei Arten unterteilt. So benennt die CNCF Entwickler-Fokussierte Tools, die helfen, Anwendungen zu containerisieren. Die zweite Art sind Tools mit Fokus auf operationelle Aspekte. Diese betrachten das Deployment von Anwendungen. In der ebenfalls in der Oberklasse "App Definition & Development" enthaltenen Klasse "Continuous Integration & Delivery" sind Tools zusammengefasst, die eine Implementierung von Pipelines ermöglichen. [15]

2 Methode

Ziel dieser Arbeit ist, eine Auswahl relevanter Werkzeuge innerhalb der GitOps-Toollandschaft hinsichtlich ihrer Genügung von GitOps als Methode sowie hinsichtlich ihres Beitrags zu dem Nutzen von GitOps zu untersuchen. Hierzu soll eine fachlich stark reduzierte Fallstudie implementiert werden und die Prinzipien von GitOps durch unterschiedliche Ansätze und Tools implementiert werden. Zu diesem Zweck ist zunächst eine Methode zur Erhebung der Genügung dieser Tools hinsichtlich der Prinzipien beziehungsweise dem Nutzen von GitOps zu spezifizieren. Bei der Untersuchung von Tools ergeben sich gewissermaßen zwei Ebenen der Evaluierung. Zunächst sind innerhalb von Toolklassen potentielle konkrete Tools auszuwählen, die Gegenstand genauerer Untersuchung sein sollen. Hierzu sind grobe, schnell evaluierbare Kriterien zu formulieren, die ein konkretes Werkzeug erfüllen muss, um für eine präzise Untersuchung innerhalb einer Werkzeugklasse in Frage zu kommen. Zwar wurde in Abschnitt 1.3 "GitOps-Toollandschaft" bereits eine grobe Klassifizierung von Tools zur Umsetzung von GitOps beschrieben. Zur Bestimmung des konkreten Bedarfs an Tools ist jedoch zunächst die Architektur der GitOps-Umgebung zu spezifizieren. Alle beschriebenen notwendigen Festlegungen werden innerhalb dieses Abschnitts getätigt.

2.1 Fallstudie

2.1.1 Spezifikation der fachlichen Architektur

Entwickelt werden soll ein in seiner Komplexität stark vereinfachtes Beispiel-System, welches ein Speichern von Text-Records (z.B. Notizen, Todo-Liste, etc.) ermöglicht. Benutzer interagieren über einer webbasierten GUI im Browser mit einem API-Service, welcher Datenbank-Zugriffe implementiert. Um den Kontext dieses Beispiels um IoT-Anwendungsfälle zu erweitern sollen Text-Records auch via MQTT gespeichert werden können. Hierzu kommt ein MQTT-Broker sowie ein weiterer Service zur Persistierung zum Einsatz.

Dieser Anwendungsfall erscheint geeignet, da in diesem Beispiel zwei Schnittstellen auf dem Backend heraus existieren. Weiterhin ist eine Web-GUI auszuliefern. Es werden selbst implementierte Komponenten mit bereits existierenden kombiniert. Die Komplexität der Implementierung der eigenen Komponenten ist sehr gering.

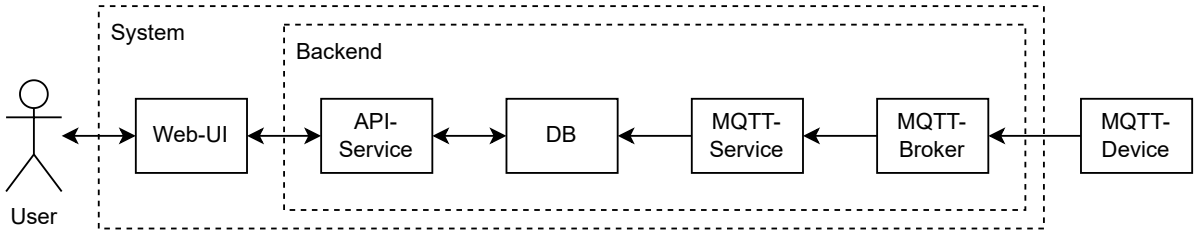

Abbildung 1: Blockdiagramm der fachlichen Architektur der Fallstudie (eigene Darstellung)

Abbildung 1 zeigt die fachliche Architektur der Fallstudie in einem Blockdiagramm. Alle Komponenten innerhalb des Dargestellten Scopes "System" sind zu deployen und auszuliefern. Der dargestellte Scope "Backend" besitzt zwei Schnittstellen (HTTP sowie MQTT). Letztendlich wird einer Interaktion durch menschliche Benutzer sowie durch MQTT-fähige Systeme ermöglicht.

Da beide Service-Komponenten in Typescript auf Node implementiert werden, kann eine gemeinsame Codebasis geteilt werden. Diese befindet sich zentral in einem Paket in der NPM-Registry und wird als Abhängigkeit innerhalb beider Service-Komponenten verwendet. Es existieren CD-Pipelines, die die geteilte Bibliothek in der NPM-Registry veröffentlicht sowie CD-Pipelines, die Docker-Images aus den Servicekomponenten baut und diese in einer Image-Registry (hier Dockerhub) veröffentlicht. Ausgangspunkt beziehungsweise Prämisse aller weiteren Umsetzungen hinsichtlich der GitOps-Methodologie ist, dass dies der Fall ist.

2.1.2 Spezifikation der GitOps-Architektur

Grundsätzlich sind zwei Bereitstellungen durchzuführen. Zum Einen muss eine Infrastrukturplattform genutzt werden, um eine Container-Orchestrierung zu erreichen. Dieser Teil der Gesamtbereitstellung wird folgend als Bereitstellung der Infrastruktur bezeichnet. Weiterhin muss diese Container-Orchestrierung genutzt werden, um den anzubietenden Service in Form von Software bereitzustellen. Dieser Teil der Gesamtbereitstellung wird folgend als Bereitstellung der Anwendung bezeichnet. Es besteht die Maßgabe, beide dieser Bereitstellungen durch IaC, also Berücksichtigung von GitOps-Prinzipien zu erreichen. IaC-Manifeste sollen also weiterhin versioniert sowie durch CI- und CD-Pipelines ausgeliefert werden.

Zudem lässt sich der Grad des Managements der Infrastruktur durch den Provider der Infrastrukturplattform unterscheiden. Bei der Nutzung von Container as a Service (CaaS) wird die Plattform zu Orchestrierung von Containern gänzlich vom Provider verwaltet. Es lässt sich ebenso ein Ansatz via Infrastructure as a Service (IaaS) verfolgen. Hierbei muss eine Orchestrierungsplattform selbst bereitgestellt und betrieben werden.

Abbildung 2 zeigt die GitOps-Architektur der Fallstudie in einem Blockdiagramm. Dargestellt werden die beiden Bereitstellungsschritte ("Deployment"). Diese implementieren die Bereitstellung der konzeptionellen Bestandteile der Architektur. Eine Container-Orchestrierungsplattform bildet gewissermaßen die Trennung zwischen Infrastruktur und Anwendung. Unterschieden wird zwischen den bereits beschriebenen CaaS- sowie IaaS-Ansatz. Gekennzeichnet wird hierzu die Verantwortlichkeit der Providers hinsichtlich der Verwaltung von Bestandteilen der Architektur.

2.2 Qualitätsmodelle

Sommerville unterscheidet zur Qualitätsbeurteilung von Software und Prozessen zwischen Produktcharakteristiken, organisatorischen Charakteristiken sowie externen Charakteristiken. Produktcharakteristiken beschreiben nicht-funktionale Eigenschaften von Softwareprodukten. Diese schließen Useability, Performanzeigenschaften, Verlässlichkeit und

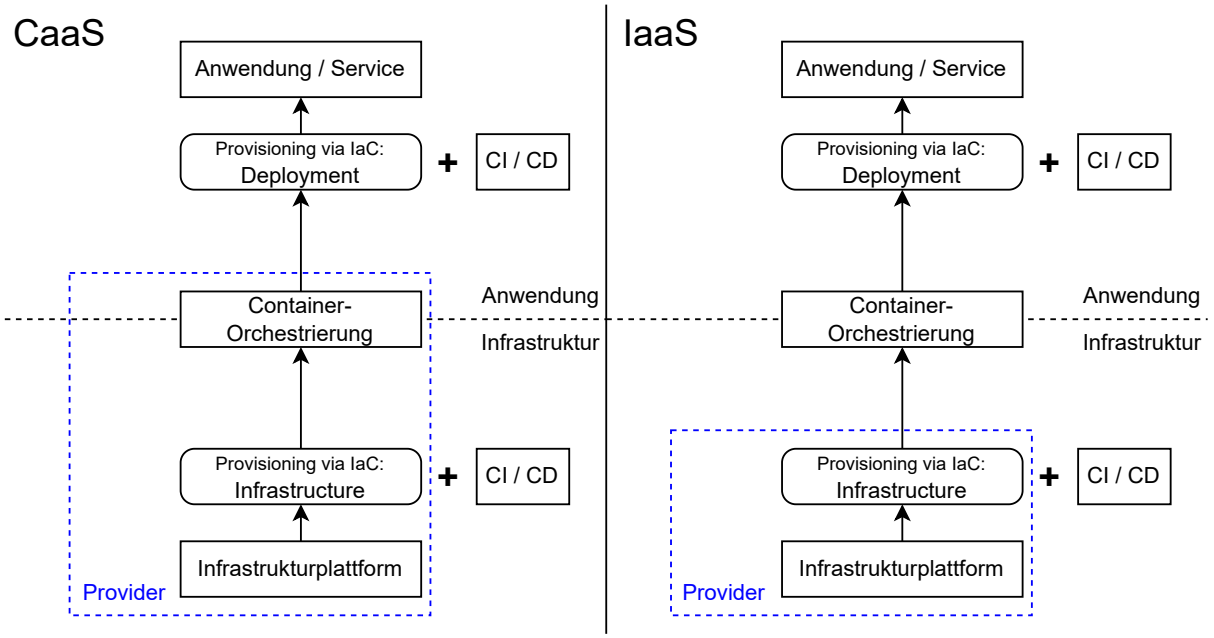

Abbildung 2: Blockdiagramm der GitOps-Architektur der Fallstudie. CaaS- sowie IaaS-Ansatz im Vergleich (eigene Darstellung)

Sicherheit mit ein. Organisatorische Charakteristiken umfassen Anforderungen an Entwicklungsumgebung, betriebliche Abläufe sowie an die Entwicklung. Externe Charakteristiken meinen die Erfüllung von regulatorischen, ethischen sowie rechtlichen Anforderungen. [16]

Auch, wenn in der Literatur die Compliance gegenüber externen Vorgaben als ein Nutzen von GitOps genannt wird [5, 11] sollen externe Charakteristiken im Zuge dieser Arbeit nicht betrachtet werden.

Die ISO 25010 formuliert zwei Qualitätsmodelle. Zum einen ein Modell zur Bewertung von Produktqualität (Product Quality) sowie ein Modell zur Bewertung von Gebrauchsqualität (Quality in Use). Das Modell zur Produktqualität betrachtet acht Kategorien an Charakteristiken von Softwareprodukten. Neben funktionalen Eigenschaften werden hier sieben Kategorien an nicht-funktionalen Charakteristiken betrachtet, die ähnlich der Definition von Sommerville sind. Konkret werden hier Eigenschaften zur Laufzeitperformanz, Kompatibilität, Useability, Zuverlässigkeit, Sicherheit, Wartbarkeit sowie Übertragbarkeit betrachtet. Das Modell zur Gebrauchsqualität betrachtet fünf Kategorien an Charakteristiken hinsichtlich der Interaktion mit Systemen. Konkret werden hier Eigenschaften zur Effektivität, Effizienz, Zufriedenheit, Risikofreiheit sowie Kontextabdeckung betrachtet. [17]

Diese Qualitätsmodelle stehen in Beziehung zueinander. So können Eigenschaften der Gebrauchsqualität von Eigenschaften der Produktqualität beeinflusst werden. Neben funktionalen Eigenschaften haben die nicht-funktionalen Charakteristiken Laufzeitperformanz, Useability, Zuverlässigkeit sowie Sicherheit einen Einfluss auf die Gebrauchsqualität von primären Nutzern. Bei Aufgaben mit Bezug zu Wartung haben die nicht-funktionalen

Produktqualitätseigenschaften Kompatibilität, Wartbarkeit sowie Übertragbarkeit einen Einfluss auf die Gebrauchsqualität. [17]

Die ISO 25022 formuliert eine Reihe an Metriken zur Erhebung von Charakteristiken zur Gebrauchsqualität. [18] Die ISO 25023 verfolgt das selbe Ziel entsprechend der Charakteristiken zur Produktqualität. [19]

2.3 Evaluierungsgegenstand

Es gilt zu definieren, welches Element Gegenstand der Evaluierung sein soll. So können Produktqualitätseigenschaften der zu untersuchenden Tools untersucht werden. Die selben Eigenschaften können jedoch auch für alle GitOps-Relevanten Artefakte zur Herstellung von Deployments innerhalb dieses Betriebsmodells untersucht werden. Prinzipiell bleibt auch die Möglichkeit die These zu formulieren, dass Produktqualitätseigenschaften der bereitzustellenden Anwendungen durch das verwendete Betriebsmodell beeinflusst werden.

Bei Betrachtung von Gebrauchsqualität können entsprechende Eigenschaften der genutzten Tools zur Umsetzung von GitOps als Betriebsmodell festgestellt werden. Prinzipiell kann auch hier wieder eine Untersuchung der bereitzustellenden Anwendungen erfolgen.

		Produktqualität	**Gebrauchsqualität**
Anwendung / Service		-	
Bereitstellung: Anwendung	Tool	Teilweise relevant	Hohe Relevanz
	IaC-Manifeste	Hohe Relevanz	Hohe Relevanz
Container-Orchestrierung		-	
Bereitstellung: Infrastruktur	Tool	Teilweise relevant	Hohe Relevanz
	IaC-Manifeste	Hohe Relevanz	Hohe Relevanz
Infrastrukturplattform		Teilweise relevant	

Tabelle 2: Konzeptionelle Bestandteile der GitOps-Architektur in Beziehung zu Qualitätsmodellen

Tabelle 2 fasst die Beziehung zwischen den konzeptionellen Bestandteilen der GitOps-Architektur zu den benannten Qualitätsmodellen zusammen. So soll im Zuge dieser Arbeit nicht die durch GitOps bereitzustellende Software, also Services selbst untersucht werden. Auch sind Plattformen zur Container-Orchestrierung nicht Bestandteil dieser Untersuchung. Die beiden bereits benannten Bereitstellungsschritte stellen den Kern der Evaluierung dar. Die teilweise gegebene Relevanz der Produktqualität von Tools geht aus der bereits beschriebenen Wechselwirkung und Beeinflussung von Eigenschaften der Gebrauchsqualität durch Eigenschaften der Produktqualität hervor.

2.4 Zu betrachtende Charakteristiken und Metriken

2.4.1 Produktqualität

Die ISO 25023 definiert Metriken zur Messung von Produktqualität. Weiterhin wird zwischen internen und externen Eigenschaften unterschieden. Interne Eigenschaften werden in der Regel durch statische Metriken beschrieben. Diese können ohne Ausführung bereits zur Entwicklung erhoben werden. Externe Eigenschaften werden in der Regel durch Messen des Verhaltens im Betrieb erhoben. Diese können nur zur Phase des Testens oder des Betriebs erhoben werden. Interne Eigenschaften beeinflussen externe und diese wiederum die Gebrauchsqualität. [19]

Im Zuge dieser Arbeit soll Produktqualität für Tools zur Umsetzung von GitOps sowie für IaC-Manifeste zur Nutzung dieser Tools bewertet werden. Für Tools bietet sich eine Betrachtung von externen, also dynamischen Eigenschaften an. Für IaC-Manifeste können nur interne, also statische Eigenschaften beurteilt werden.

Funktionale Eignung Die Produktqualitätseigenschaft funktionale Eignung beurteilt den Grad zu dem ein Produkt Funktionalität bereitstellt, die aus explizite und implizite Anforderungen innerhalb eines spezifizierten Nutzungskontextes hervorgeht. Unterschieden wird hierzu zwischen funktionaler Vollständigkeit oder Abdeckung sowie funktionaler Korrektheit. Die funktionale Abdeckung betrachtet den Grad zu dem spezifizierte Funktionalität implementiert ist. Die funktionale Korrektheit betrachtet den Grad zu dem diese Funktionalität korrekte Ergebnisse liefert. Weiterhin wird durch die ISO 25023 die Eigenschaft der funktionalen Angemessenheit formuliert. Diese Eigenschaft misst den Grad zu dem Funktionalität zur Erledigung spezifizierter Aufgaben genügt. [19]

Funktionale Angemessenheit steht in enger Beziehung zur in Abschnitt 2.4.2 "Gebrauchsqualität" beschriebenen Gebrauchsqualitätseigenschaft Effektivität. Im Zuge dieser Arbeit ist also zunächst erwartete Funktionalität zu spezifizieren.

Kompatibilität Die Produktqualitätseigenschaft Kompatibilität teilt sich in zwei konkrete Charakteristiken auf. Koexistenz beschreibt den Grad zu dem ein Produkt seine spezifizierte Funktionalität auf geteilten Ressourcen bereitstellen kann ohne ein Anderes negativ zu beeinflussen. Interoperabilität beschreibt den Grad zu dem zwei oder mehr Produkte Informationen austauschen und diese erfolgreich verarbeiten können. [19]

Kompatibilität steht gewissermaßen in Beziehung zu der später in diesem Abschnitt beschriebenen Produktqualitätseigenschaft Portabilität. Eine Beurteilung von Tools hinsichtlich Kompatibilität im Zuge dieser Arbeit kann in Beziehung zur Portabilität von Relevanz sein.

Useability Die Produktqualitätseigenschaft Useability beschreibt den Grad zu dem ein bei Nutzung eines Produkts durch Benutzer Ziele erreicht werden können. Benutzer sowie Ziele sind zu spezifizieren. Unter dieser Eigenschaft sind sechs konkrete Charakteristiken definiert. Die Eigenschaft der Erkennbarkeit der Angemessenheit (Appropriateness recognizability) beschreibt den Grad zu dem Benutzer beurteilen können, dass ein Produkt ihrem Anwendungsfall genügt. Erlernbarkeit beschreibt den Grad zu dem Benutzer die Benutzung einer Produkts erlernen können. Die Bedienbarkeit eines Produkts beschreibt

den Grad zu dem es Eigenschaften besitzt, die seine Bedienung einfach machen. Eine weitere Charakteristik beschreibt den Grad des Schutzes vor Benutzerfehlern. Hierbei ist die Verhinderung von Fehlern von Bedeutung. Die letzten Produktqualitätseigenschaften beurteilen die Ästhetik der Benutzeroberfläche sowie dessen Zugänglichkeit. [19]

Eine Untersuchung der Useability von Tools zur Umsetzung von GitOps erscheint als Relevant. Auf die Beurteilung von Ästhetik der Benutzeroberfläche sowie Zugänglichkeit soll im Zuge dieser Arbeit verzichtet werden.

Wartbarkeit Die Produktqualitätseigenschaft Wartbarkeit beurteilt den Grad der Effektivität und Effizienz zu dem ein Produkt modifiziert werden kann. Es sind fünf konkrete Charakteristiken innerhalb dieser Kategorie definiert. Modularität bewertet den Grad zu dem ein Produkt aus Komponenten zusammengesetzt ist, die unabhängig und ohne beziehungsweise mit minimaler gegenseitige Beeinflussung verändert werden können. Wiederverwendbarkeit beschreibt den Grad zu dem ein Asset in mehr als einem System oder zur Erzeugung weiterer Assets verwendet werden kann. Die Charakteristik Analysierbarkeit bewertet den Grad der Effektivität und Effizienz mit der die Auswirkung einer Änderung abgeschätzt werden kann. Weiterhin umfasst diese Eigenschaft die Diagnostizierbarkeit im Fehlerfall. Modifizierbarkeit bewertet den Grad zu dem ein Produkt modifiziert werden kann ohne die bestehende Produktqualität negativ zu beeinflussen. Die Eigenschaft Testbarkeit befasst sich mit dem Grad der Effektivität und Effizienz mit dem Testkriterien formuliert und Tests zur Messung dieser durchgeführt werden können. [19]

Wartbarkeit ist innerhalb dieser Arbeit zur Beurteilung von IaC-Manifesten von höchster Bedeutung.

Portabilität Die Produktqualitätseigenschaft Portabilität bewertet den Grad der Effektivität und Effizienz mit der ein Produkt von einer Umgebung in eine Andere übertragen werden kann. Hierbei können sowohl Hardware- als auch Softwareumgebungen sowie alle weiteren betrieblichen Umgebungen von Relevanz sein. Unter Portabilität existieren drei konkrete Charakteristiken. Anpassungsfähigkeit beschreibt den Grad zu dem ein Produkt effektiv und effizient an unterschiedliche oder sich verändernde Umgebungen angepasst werden kann. Die Eigenschaft der Installierbarkeit bewertet den Grad der Effektivität und Effizienz zu dem Produkt erfolgreich in eine spezifizierte Umgebung installiert oder von dieser deinstalliert werden kann. Austauschbarkeit beschreibt den Grad zu dem ein Produkt ein Anderes für den selben Zweck in der selben Umgebung ersetzen kann. [19]

Im Zuge dieser Arbeit ist zur Beurteilung von Tools die Eigenschaft der Portabilität von hoher Bedeutung.

Weitere Produktqualitätseigenschaften Die ISO 25010 in Kombination mit der ISO 25023 formulieren noch vier weitere Produktqualitätseigenschaften. [17, 19] Laufzeitperformanz sowie Zuverlässigkeit sollen kein Bestandteil der Betrachtung innerhalb dieser Arbeit sein. Eine Bewertung dieser Charakteristiken genügen nicht dem Ziel dieser Arbeit und haben maximal einen marginalen Wert zur Beantwortung der Fragestellung.

Die Eigenschaft Sicherheit wird in Abschnitt 1.2.5 "Sicherheit" indirekt als Nutzen von GitOps genannt. Die innerhalb der ISO 25023 für diese Charakteristik spezifizierten Metriken sind für diese Arbeit als nicht nützlich einzuschätzen.

Kompatibilität von Tools und Portabilität von IaC-Manifesten Die Kompatibilität von Tools steht in direkter Beziehung zur Portabilität von IaC-Manifesten. Wenn IaC-Manifeste als geteilte Ressourcen betrachtet werden, erzeugen Tools mit einer hohen Kompatibilität hinsichtlich unterschiedlicher Formate von IaC-Manifesten eine hohe Portabilität dieser.

2.4.2 Gebrauchsqualität

Die ISO 25022 definiert eine Aufgabe (Task) als Aktivitäten, die zur Erreichung eines Ziels erforderlich sind. Ein Ziel (Goal) ist definiert als ein angestrebtes Ergebnis. Die ebenfalls in der ISO 25022 spezifizierten Metriken zur Messung von Gebrauchsqualität sind teilweise im Kontext eines Ziels zu betrachten. Es wird zu jeder Metrik die notwendige Methode zur Erhebung dieser beschrieben. So kann das Messen der Performanz oder des Verhaltens von Benutzern erforderlich sein. Andere Metriken erfordern ein Durchführen von Umfragen sowie Geschäfts- oder Risikoanalysen. Wieder weitere Methoden basieren auf der Betrachtung von Nutzungsstatistiken. [18]

Im Zuge dieser Arbeit soll die ISO 25022 nur als Richtlinie zur Messung von Eigenschaften fungieren. Eine umfassende konforme Analyse nach Vorgaben der Norm wird nicht möglich sein. Es wird eine Abwägung vorgenommen, hinsichtlich der zu erwartenden Qualität der Durchführung der Erhebungsmethode. So können Methoden, dessen Ergebnis von der Größe einer Stichprobe abhängen nicht zu einem ausreichenden Grad durchgeführt werden. Solche Metriken sollen trotzdem zur Argumentation der Beurteilung herangezogen werden.

Folgend werden nun für die in Tabelle 2 beschriebenen konzeptionellen Bestandteile der GitOps-Architektur Metriken zur Erhebung von Eigenschaften der Gebrauchsqualität unter Berücksichtigung von Aufgaben formuliert werden.

Effektivität Die Gebrauchsqualitätseigenschaft Effektivität betrachtet den Grad der Erreichung spezifizierter Ziele. Hierbei ist die Art der Erreichung beziehungsweise der Lösungsweg nicht von Interesse. Die Metrik "Task completion" entspricht dem Anteil von Aufgaben, die korrekt abgeschlossen werden können. Die Metrik "Task effectiveness" entspricht dem Anteil der Ziele einer Aufgabe, die korrekt erreicht werden konnten. Die Metrik "Error frequency" entspricht der Frequenz von Fehlern im Bezug zu einem Referenzwert wie der Anzahl an Aufgaben. [18]

Diese drei Metriken stehen in enger Abhängigkeit zur Produktqualität der untersuchten Tools. Gewissermaßen stellt diese Eigenschaften eine andere Betrachtung von funktionalen Produktqualitätseigenschaften dar. Besonders die in Abschnitt 2.4.1 "Produktqualität" beschriebene Produktqualitätseigenschaft der funktionalen Eignung steht in unmittelbarer Wechselwirkung mit Effektivität.

Die Gebrauchsqualitätseigenschaft Effektivität hängt prinzipiell mit allen in Abschnitt 1.2 "Nutzen von GitOps" beschriebenen Nutzen von GitOps zusammen. So können funktionale Eigenschaften von Tools, die zur Erreichung spezifizierter Ziele erforderlich sind den Nutzen von GitOps beeinflussen, da dieser durch entsprechende Tools implementiert wird.

Effizienz Die Gebrauchsqualitätseigenschaft Effizienz betrachtet die Effektivität unter Berücksichtigung aufgewendeter Ressourcen. Betrachtet wird im Kern die Ressource Zeit. So werden in diversen Metriken gemessene aufgewendete Zeit für Aufgaben in Bezug zu Zielwerten gestellt. Auch werden Vergleiche mit dem Zeitaufwand von Experten vorgeschlagen. Zur Erhebung der tatsächlichen Effizienz wird die Effektivität in direktem Bezug zur gemessenen Zeit gestellt. In einer weiteren Metrik wird der Gesamtzeitaufwand zum Abschluss einer Aufgabe zerlegt um die tatsächliche produktive Zeit zu erheben. [18]

Diese Metriken sind im Zuge dieser Arbeit nicht praktikabel zu erheben. Es kann lediglich eine grobe Einschätzung vorgenommen werden, dessen Qualität nahe der Aussagelosigkeit liegen sollte. Aus diesem Grund wird auf eine empirische Betrachtung der Effizienz verzichtet.

Die ISO 25010 beschreibt für Produktqualitätseigenschaften in der Eigenschaftskategorie Useability die Eigenschaft Erlernbarkeit. [17] Mögliche Erwartungswerte an Effizienz können durch eine Abschätzung der Erlernbarkeit formuliert werden. Hierzu kann unter Anderem beurteilt werden, wie viel Wissen zum Verständnis sowie der Nutzung von Bestandteilen der GitOps-Architektur wiederverwendet werden kann.

Diese Gebrauchsqualitätseigenschaft steht in direkter Verbindung mit der in Abschnitt 1.2.1 "Produktivität und Effizienz" als Nutzen von GitOps beschriebenen Erhöhung von Produktivität und Effizienz. Aus diesem Grund soll keinesfalls auf eine Untersuchung von Effizienz verzichtet werden. Eine solche Beurteilung muss jedoch deduktiv auf Basis von Effektivität vorgenommen werden.

Zufriedenheit Die Gebrauchsqualitätseigenschaft Zufriedenheit beurteilt den Grad zu dem Benutzer-Bedürfnisse oder -Anforderungen bei der Benutzung eines Produkts erfüllt werden. Diese Betrachtung erfolgt immer im Hinblick auf einen spezifizierten Kontext. Die Zufriedenheit eines Benutzers mit einem Produkt ist beeinflusst von allen anderen Gebrauchsqualitätseigenschaften. In der Eigenschaftskategorie Zufriedenheit werden vier konkrete Charakteristiken benannt. [18]

Die Eigenschaft Nützlichkeit bewertet den Grad zu dem ein Benutzer zufrieden mit seiner selbst wahrgenommenen Performanz ist. Die Eigenschaft Vertrauen bewertet den Grad zu dem ein Benutzer oder Stakeholder ein Produkt vertraut, dass es sich wie beabsichtigt verhält. Die Eigenschaft Freude oder Vergnügen (Pleasure) bewertet den Grad zu dem ein Benutzer Freude an der Erreichung von Zielen verspürt. Zuletzt bewertet die Eigenschaft Komfort den Grad zu dem ein Benutzer Komfort bei der Benutzung eines Produkts verspürt. Metriken zur Messung dieser Eigenschaften setzen zur Erhebung auf Umfragen unter der Verwendung von psychometrischen Skalen. [18]

Im Zuge dieser Arbeit können diese Eigenschaften nur anhand einer Person erhoben werden. Diese Stichprobengröße macht das Messen eines aussagekräftigen Ergebnisses unmöglich. Eine Erhebung würde immer nur eine Einzelmeinung abbilden. Aus diesem Grund soll im Zuge dieser Arbeit auf eine Betrachtung von Zufriedenheit verzichtet werden.

Die Gebrauchsqualitätseigenschaft Zufriedenheit steht in direkter Verbindung mit dem in Abschnitt 1.2.3 "Developer Experience" beschrieben Vorteil der Verbesserung der Developer Experience.

Risikofreiheit Die Gebrauchsqualitätseigenschaft Risikofreiheit beurteilt den Grad zu dem ein Produkt potentielle Risiken abmindert. Diese Risiken können wirtschaftliche, gesundheitliche bis hin zu Menschenleben sowie umweltbezogene Risiken einschließen. Die ISO 25022 definiert die Eigenschaft der Risikominderung (Risk mitigation) mit dem Grad zu dem Produktqualitätseigenschaften Risiken abmindern. Hierzu wird das Risiko unter hoher Qualität dieser Eigenschaften mit dem Risiko unter niedriger Qualität dieser Eigenschaften verglichen. Die wirtschaftlichen Aspekte dieser Kategorie betrachten klassische Metriken wie Return of investment (ROI) sowie die Zeit zur Erreichung dieser. Es werden aber auch Messgrößen wie Auslieferungszeit oder individueller Kundenwert hinzugezogen. Aspekte zur gesundheitlichen Auswirkungen sowie Sicherheit werden unter Betrachtung der Anzahl betroffener Personen bewertet. [18]

Diese Arbeit ist nicht in einen kommerziellen Kontext eingebettet. Aus diesem Grund ist eine realistische Bewertung von wirtschaftlichen Aspekten nicht möglich. Jedoch können für Teile der Metriken Erwartungswerte abgeschätzt werden. Hierfür bietet sich prinzipiell die Metrik Return of Investment (ROI) an. Diese stellt den wirtschaftlichen Nutzen mit den notwendigen Investitionen gegenüber. Der wirtschaftliche Nutzen kann hierbei eine Reduktion jeglicher Ressourcen bedeuten. [18] Auch kann die Lieferzeit beurteilt werden. Diese stellt tatsächliche Zeiträume zur Auslieferung einem Erwartungswert gegenüber. [18]

Bei Betrachtung von betrieblichen Risiken steht die Gebrauchsqualitätseigenschaft Risikofreiheit gewissermaßen in Verbindung mit dem in Abschnitt 1.2.2 "Stabilität und Zuverlässigkeit" beschriebenen Nutzen der Verbesserung von Stabilität und Zuverlässigkeit von bereitzustellender Software durch GitOps.

Kontextabdeckung Die Gebrauchsqualitätseigenschaft Kontextabdeckung beurteilt den Grad der Erfüllung aller anderen Gebrauchsqualitätseigenschaften außerhalb eines initial spezifizierten Kontexts. Hierzu wird zunächst die Kontextvollständigkeit betrachtet. Diese beschreibt den Grad der Erfüllung aller anderen Gebrauchsqualitätseigenschaften innerhalb eines initial spezifizierten Kontexts. Die Eigenschaft Flexibilität misst diese Eigenschaft eines Produkts außerhalb eines initial spezifizierten Kontexts. [18]

Innerhalb dieser Arbeit ist durch die Spezifikation einer Fallstudie auch ein Kontext definiert. Eine konkrete Untersuchung außerhalb dieses Kontexts ist nicht vorgesehen. Es kann jedoch im Zuge der Untersuchungen zu Beobachtungen kommen, die eine Einschätzung der Kontextabdeckung erlauben. Hierbei ist jedoch anzumerken, dass ein

Fehlen einer solchen Einschätzung nicht bedeutet, dass eine Kontextabdeckung zu einem schlechten Grad vorliegt.

Unterscheidung zwischen Tools und IaC-Manifesten bei Betrachtung der Gebrauchsqualität Die Gebrauchsqualität von Tools und IaC-Manifesten ist gemeinsam zu betrachten. IaC-Manifeste stellen gewissermaßen das Interface dieser Tools dar und können ohne diese gar nicht erst eingesetzt und genutzt werden. Trotzdem ist auch eine getrennte Untersuchung denkbar. Hierbei können unter Anderem Eigenschaften wie die Unterstützung durch Editoren beim Erzeugen von IaC-Manifesten als Gebrauchsqualität dieser verstanden werden. Bei Betrachtung der Effektivität können Ziele weiterhin so formuliert werden, dass sie nur Tools beziehungsweise nur IaC-Manifeste betrachten. Zur Vereinfachung wird im Folgenden bei der Untersuchung von Gebrauchsqualität immer die Kombination aus Tools und zugehörigen IaC-Manifesten impliziert, auch wenn vereinzelt Eigenschaften existieren können, die nur auf eines dieser Objekte anwendbar sind.

2.4.3 Zusammenfassung

Infrastrukturplattformen Wie bereits in Abschnitt 2.1.2 "Spezifikation der GitOps-Architektur" dargestellt ist eine Infrastrukturplattform unbedingt notwendig. An diese bestehen funktionale Mindestanforderungen zur Umsetzung von GitOps. Aus diesem Grund ist die funktionale Eignung von Infrastrukturplattform im Zuge dieser Arbeit von Relevanz.

Tabelle 3 fasst den Zusammenhang von Produkt- und Gebrauchsqualitätseigenschaften mit den Untersuchungsgegenständen, also den Elementen der GitOps-Architektur zusammen.

2.4.4 Zusammenhang mit Nutzen von GitOps

Ziel dieser Arbeit ist die Evaluation des in Abschnitt 1.2 "Nutzen von GitOps" beschriebenen Nutzens von GitOps als Betriebsmodell. Es ist nun zu definieren durch welche Qualitätseigenschaften welcher konkrete Nutzen untersucht werden kann. Tabelle 4 fasst diesen Zusammenhang zusammen.

Zur Umsetzung von GitOps als Betriebsmodell sind entsprechende Technologien in Form von Tools notwendig. Diese implementieren also notwendige Funktionalität, die unbedingt notwendig ist, damit überhaupt ein Nutzen aus GitOps gezogen werden kann. Aus diesem Grund nimmt die funktionale Eignung von Tools sowie die Effektivität hinsichtlich Gebrauchsqualität unmittelbaren Einfluss auf jeden Nutzen von GitOps.

Produktivität und Effizienz Der in Abschnitt 1.2.1 "Produktivität und Effizienz" beschriebene Nutzen der Verbesserung von Produktivität und Effizienz lässt sich vor allem durch Gebrauchsqualitätseigenschaften untersuchen. Die Wechselwirkung zwischen Risikofreiheit und Produktivität und Effizienz ist gewissermaßen in beide Richtungen vorhanden. So hängen wirtschaftliches Risiko und Produktivität zusammen. Mit der Kontextabdeckung lässt sich so weiterhin feststellen, ob Produktivität und Effizienz in abweichenden Kontexten gegeben sein kann.

	Tools	IaC-Manifeste	Infra.-Plattformen
Produktqualität			
Funktionale Eignung	X		X
Funktionale Vollständigkeit	X		X
Funktionale Angemessenheit	X		X
Kompatibilität	X		
Koexistenz	X		
Interoperabilität	X		
Useability	X		
Erkennbarkeit der Angemessenheit	X		
Erlernbarkeit	X		
Bedienbarkeit	X		
Schutz vor Benutzerfehlern	X		
Wartbarkeit		X	
Modularität		X	
Wiederverwendbarkeit		X	
Analysierbarkeit		X	
Modifizierbarkeit		X	
Testbarkeit		X	
Portabilität	X	X	
Adaptierbarkeit	X	X	
Installierbarkeit	X	X	
Austauschbarkeit	X	X	
Gebrauchsqualität			
Effektivität		X	X
Effizienz		(X)	(X)
Risikofreiheit		(X)	(X)
Kontextabdeckung		(X)	(X)

Tabelle 3: Zusammenfassung des Zusammenhangs von Produkt- sowie Gebrauchsqualitätseigenschaften mit Untersuchungsgegenständen
X = Untersuchbar
(X) = Bedingt Untersuchbar

Stabilität und Zuverlässigkeit Der in Abschnitt 1.2.2 "Stabilität und Zuverlässigkeit" beschriebene Nutzen der Verbesserung von Stabilität und Zuverlässigkeit lässt sich vor allem durch die Wartbarkeit von IaC-Manifesten bewerten. Hier besteht eine enge Beziehung. So wird als Teil von Stabilität und Zuverlässigkeit die Eigenschaft beschrieben, Fehler schnell zu erkennen. Dies geht unmittelbar aus der Charakteristik Analysierbarkeit hervor, welche per Definition auch Diagnostizierbarkeit umfasst. Zuverlässigkeit kann weiterhin durch ein effizientes Beheben von Fehlern erreicht werden. Die Charakteristik Modifizierbarkeit drückt diese Eigenschaft aus.

Developer Experience Der in Abschnitt 1.2.3 "Developer Experience" beschriebene Nutzen der Verbesserung der Developer Experience hängt mit der Useability von Tools

zusammen. So wird beispielsweise sowohl unter hoher Developer Experience als auch Useability eine gute Erlernbarkeit benannt.

Konsistenz und Standardisierung Der in Abschnitt 1.2.4 "Konsistenz und Standardisierung" beschriebene Nutzen der Verbesserung von Konsistenz und den Grad der Standardisierung hängt mit diversen Qualitätseigenschaften zusammen. So ist ein hohes Maß an Kompatibilität und Portabilität Nachweis für Standardisierung. Selbes gilt für die Charakteristik der Wiederverwendbarkeit unter der Eigenschaftskategorie Wartbarkeit.

	Produktivität und Effizienz	Stabilität und Zuverlässigkeit	Developer Experience	Konsistenz und Standardisierung	Sicherheit	Selbst-Dokumentation
Produktqualität						
Tools						
Funktionale Eignung	X	X	X	X	X	X
Kompatibilität				X		
Useability			X			
Portabilität				X		
IaC-Manifeste						
Wartbarkeit		X		X		
Portabilität				X		
Gebrauchsqualität						
Effektivität	X	X	X	X	X	X
Effizienz	X					
Risikofreiheit	X					
Kontextabdeckung	X					

Tabelle 4: Zusammenfassung der Beziehung zwischen GitOps-Nutzen und Qualitätseigenschaften
X = Zusammenhang besteht

3 Realisation

In diesem Abschnitt wird die Implementierung der in Abschnitt 2.1.2 "Spezifikation der GitOps-Architektur" spezifizierten GitOps-Architektur dargelegt. Es werden konkrete Tools für konzeptionelle Bestandteile der Architektur gewählt sowie das Vorgehen dargelegt.

Für die Implementierung der Architektur ist zunächst ein Versionskontrollsystem zu wählen. Innerhalb dieser Arbeit kommt Git unter der Plattform GitLab zum Einsatz.

3.1 Infrastrukturplattform

3.1.1 Anforderungen

Dieser Abschnitt formuliert Anforderungen an Infrastrukturplattformen zur Umsetzung von GitOps. Wie bereits in Abschnitt 1.3.2 "Tools zur Definition von Infrastruktur" definiert, stellen Infrastrukturplattformen Rechen-, Speicher- sowie Netzwerkressourcen zur Verfügung. Die müssen dazu programmierbar sein. Hieraus geht bereits unmittelbar die erste konkrete Anforderung hervor. Eine Infrastrukturplattform muss also eine API in beliebiger Form zur Herstellung von Programmierbarkeit bieten. Weiterhin müssen Rechenressourcen in beliebiger Form zu Verfügung gestellt werden. Hierbei ist unerheblich ob diese in Form von Bare-metal-Systemen, virtuellen Maschinen oder Containern angeboten werden. An diese müssen Speicherressourcen gebunden sein und diese müssen über ein Netzwerk kommunizieren können. Dies stellt die Mindestanforderung an Infrastrukturplattformen dar.

Diverse Infrastrukturplattformen bieten zudem zusätzliche Services an. So sind entkoppelte Speicherlösungen wie Block-Storage-Volumes sowie Netzwerklösungen wie virtuelle Netzwerke oder Load-Balancer üblich. Zwar sind solche für die Umsetzung von GitOps nicht zwingend notwendig sollen jedoch trotzdem Beachtung finden.

IaaS vs. CaaS Diese Anforderungen beschreiben jene an eine IaaS-Plattform. Je nach Strategie zur Orchestrierung von Containern fehlt zur Bereitstellung von CaaS lediglich das Management der Orchestrierungsplattform. In Kombination mit IaaS-Ressourcen kann dann CaaS hergestellt werden.

3.1.2 Toolauswahl

Die drei größten Provider für Cloudressourcen nach Umsatz sind Microsoft mit Azure, Amazon mit AWS und Google mit der Google Cloud. [20] Diese sollen zunächst auf ihr Potential zur Genügung von GitOps untersucht werden. Weiterhin werden DigitalOcean, Vultr und Linode betrachtet. Zuletzt sollen auch OVH und Hetzner untersucht werden.

Bis auf Hetzner gelten alle genannten Infrastrukturplattformen nach der CNCF als zertifizierte Kubernetes-Plattform. Insgesamt werden 50 solcher benannt. [14]

	Microsoft Azure	Amazon AWS	Google Cloud	DigitalOcean	Vultr	Linode	OVHcloud	Hetzner cloud
API	X	X	X	X	X	X	X	X
Rechenressourcen								
Computing-Instanzen	X	X	X	X	X	X	X	X
Speicherressourcen								
Block-Storage-Volumes	X	X	X	X	X	X	X	X
Netzwerkressourcen								
Virtuelle Netzwerke	X	X	X	X	X	X	X	X
Loadbalancer	X	X	X	X	X	X	X	X
CaaS								
Managed Kubernetes	X	X	X	X	X	X	X	-
Kein Aufpreis	X	-	-	-	X	X	X	N/A
Quelle	[21]	[22]	[23]	[24]	[25]	[26]	[27]	[28]

Tabelle 5: Zusammenfassung von Features ausgewählter Infrastrukturplattformen
X = Feature vorhanden
- = Feature nicht vorhanden

Tabelle 5 fasst genannte Funktionalität, also funktionale Eignung der genannten Infrastrukturplattformen zusammen. Alle Provider erfüllen alle Anforderungen mit Ausnahme bei CaaS. Hier ist Hetzner der einzige Anbieter, der dieses Feature nicht zur Verfügung stellt. Eine weitere Betrachtung lässt sich hinsichtlich der Kostenstruktur bei CaaS durchführen. Hier ist bei Amazon AWS, Google Cloud sowie DigitalOcean mit Mehrkosten zu rechnen.

Abschließend ist festzuhalten, dass alle genannten Anbieter für die Umsetzung von GitOps geeignet sind.

3.1.3 Umsetzung

Die Fallstudie dieser Arbeit wird unter Nutzung von Linode implementiert. Linode stellt alle genannten Funktionalitäten zur Verfügung. Weiterhin wird diese Wahl aus pragmatischen Gründen getroffen. So bietet Linode neuen Kunden einen Evaluationsguthaben von 100 US-Dollar. [29]

Zur Implementierung der Fallstudie wurde das CaaS-Angebot von Linode "Linode Kubernetes Engine (LKE)" genutzt. Dies ist Produkt zum Deployment von verwalteten Kubernetes-Clustern. [26] Mehr zu Kubernetes in Abschnitt 3.4 "Container-Orchestrierung".

3.2 Bereitstellung der Infrastruktur

3.2.1 Anforderungen

In Abschnitt 1.3.2 "Tools zur Definition von Infrastruktur" wurden Tools zur Definition von Infrastruktur bereits beschrieben. Diese können, müssen jedoch nicht, IaC-Ansätze verfolgen. Wie in Abschnitt 1.1 "Prinzipien von GitOps" beschrieben ist ein Verwenden von IaC ein essentieller Bestandteil von GitOps als Methodologie. Außerdem wurden bereits Anforderungen an diese formuliert. So müssen diese eine programmierbare Schnittstelle besitzen und unbeaufsichtigt ausführbar sein. Dies bedeutet, dass diese eine geeignete Schnittstelle wie zum Beispiel ein Commandline-Interface (CLI) besitzen. Außerdem muss Konfiguration externalisiert sein.

3.2.2 Toolauswahl

Alle in Abschnitt 3.1 "Infrastrukturplattform" vorgestellten Infrastrukturplattformen besitzen eine webbasierte grafische Benutzeroberfläche zur Definition und Management von Ressourcen. Während diese prinzipiell legitime Tools zur Bereitstellung von Infrastruktur darstellen sind diese nicht programmierbar. Zwar ist anzunehmen, dass diese die genannte APIs der Provider nutzen, jedoch geht in der Regel durch die Schicht einer grafischen für Benutzer konzipierten Oberfläche die Eigenschaft der Programmierbarkeit verloren.

Die CNCF zählt in der Kategorie "Provisioning" innerhalb der Klasse "Automation & Provisioning" 37 Technologien und Tools auf. [14] Über diese können nun Metriken betrachtet werden, die eine Einschätzung über die Community hinter diesen liefern können. Dies wiederum erlaubt in Open Source-Projekten eine Einschätzung der Qualität von Wartung und Pflege dieser Software. Weiterhin kann über die Größe der Community die Relevanz einzelner Werkzeuge abgeschätzt werden. Wenn das Erlernen der Arbeit mit einer Technologien als Investition betrachtet wird erlaubt dies nicht zuletzt auch eine Abschätzung des Wertes dieser Investition. Gegebenenfalls sind entsprechende Tools längerfristig von Relevanz und in der Praxis im Einsatz.

Aus der Auflistung der CNCF fallen hier besonders die Werkzeuge Ansible und Terraform auf. Beide sind Open Source und werden auf der Plattform GitHub entwickelt. Ansible besitzt hier im Juni 2022 fast 54.000 "Starrings" sowie über 5000 Contributors. [30] Terraform über 33.000 Starrings beziehungsweise über 1.600 Contributors. [31] Auf der Plattform Stackshare können Unternehmen und Individuen Technologieentscheidungen veröffentlichen. [32] Dort machen (Stand Juni 2022) über 1.800 Unternehmen die Angabe, Ansible zu nutzen. [33] Für Terraform sind dies über 1.700 Unternehmen. [34] Die beiden Tools sind damit in ihren Kategorien jeweils auf Platz eins der meistgenutzten Tools. [35, 36]

Klassifikation Auch Morris nennt Ansible und Terraform als relevante Tools. Terraform klassifiziert er als "Infrastructure Definition Tool". [7] Diese sind in Abschnitt 1.3.2 "Tools zur Definition von Infrastruktur" beschrieben. Ansible wird hier als "Server Configuration Tool" kategorisiert. [7] Diese sind in Abschnitt 1.3.3 "Tools zur Konfiguration von Servern" beschrieben. Eine ähnliche Klassifikation nimmt Stackshare vor. Hier wird Terraform als "Infrastructure Build Tool" bezeichnet. [34] Ansible gilt als "Server

Configuration and Automation Tool". [33] Die CNCF nimmt keine solche Unterscheidung vor. [14]

Diese Arbeit wird zeigen, dass beide Tools alle innerhalb dieser Fallstudie erforderlichen Anforderungen erfüllt. Dies geht in der Theorie daraus hervor, dass in containerisierten Umgebungen implizit das Modell der immutable Infrastructure umgesetzt wird. Hierbei werden Änderungen an Ressourcen nicht durch Änderung dieser selbst sondern durch Erzeugung einer geänderten Version und Löschung der alten Version vorgenommen. [7] Diese Verringerung in der Komplexität von Konfiguration sorgt dafür, dass auch Terraform für diesem Zweck eingesetzt werden kann.

Master & Agents Bereitstellungs- beziehungsweise Konfigurationstools können so konzipiert sein, dass alle Aktionen über einen Master-Server durchgeführt werden. Dieser kann weiterhin zentral den Zustand der gesamten Infrastruktur halten. Der Gegenentwurf hierzu wird als masterless bezeichnet. [37] Morris beschreibt dieses Vorgehen als ein eigenes Pattern des "Masterless Configuration Management". Als Vorteil führt er unter Anderem an, dass ein zentraler Server einen Single Point of Failure darstellt. [7] Das Konzept der Masters verbessert unmittelbar die in Abschnitt 1.2.6 "Selbst-Dokumentation" beschriebene Eigenschaft der Visibility bei selbst-dokumentierten Systemen.

Die Konfiguration von Servern kann durch Agent-Software durchgeführt werden, welche vorher installiert werden muss. Der Gegenentwurf hierzu wird als agentless bezeichnet. Hierbei besteht die Herausforderung des Bootstrapping. Es stellt sich die Frage wie ein notwendiger Agent installiert werden kann. [37] Dies kann prinzipiell nur agentless geschehen. Bei der Bereitstellung von Infrastruktur selbst ist zudem die Tatsache zu beachten, dass noch keine Ressourcen zur Ausführung von Agents zur Verfügung steht.

Ansible ist standardgemäß masterless. [38] Es lässt sich Ausführung jedoch delegieren. [39] So können Änderungen prinzipiell von einem zentralen Server durchgeführt werden. Terraform ist standardgemäß ebenfalls vollständig masterless. [40] Die Terraform cloud ermöglicht jedoch ein Master-basierten Betrieb. [41] Terraform kennt weiterhin das Konzept der "Backends". Durch diese kann der Zustand von Infrastruktur zentral auf einem Server verwaltet werden. [42] Ansible [38] sowie Terraform [40] sind agentless.

Evaluation von Anforderungen In Ansible wird Infrastruktur definiert in dem Tasks in sogenannten Playbooks definiert werden. Dies geschieht in YAML-Syntax. [43] Ein Task kann hierbei die Bereitstellung einer Ressource implementieren. Die meisten Module erlauben eine Prüfung, ob der Soll-Zustand bereits erreicht ist. In diesem Fall wird keine Modifikation der Infrastruktur vorgenommen. Ansible beschreibt diese Eigenschaft als "indempotency". [44] Somit bietet Ansible einen deklarativen IaC-Ansatz.

In Terraform wird Infrastruktur durch Spezifikation von Ressourcen definiert. [45] Dies geschieht entweder in Terraforms eigener Konfigurationssprache HCI (Hashicorp Configuration Language) oder alternativ in JSON. [46] Da JSON eine Untermenge von YAML ist [47], ist ebenso wie in Ansible eine Definition durch YAML möglich. Terraform implementiert damit präzise einen IaC-Ansatz.

Ansible [43] sowie Terraform [48] bieten ein Commandline-Interface (CLI) zur Ausführung.

Sowohl Ansible [49] sowie Terraform [50] kennen das Konzept von Variablen. Als Quelle für Werte können sowohl unter Ansible [51] als auch Terraform [52] Umgebungsvariablen genutzt werden. Damit ist eine Externalisierung von Konfiguration in Einklang mit Programmierbarkeit optimal gegeben.

Zusammenfassung Tabelle 6 fasst die Haupteigenschaften von Ansible und Terraform zusammen. So zeigt sich, dass beide Tools innerhalb dieser Betrachtung exakt den selben Funktionsumfang bieten.

	Ansible	Terraform
IaC	X	X
CLI	X	X
Externe Konfiguration	X	X
Master	- / X	- / X
Agent	-	-

Tabelle 6: Zusammenfassung der Haupteigenschaften von Ansible und Terraform
X = Eigenschaft gegeben
- = Eigenschaft nicht gegeben

Integration mit Infrastrukturplattformen Tools zur Bereitstellung von Infrastruktur nutzen APIs von Infrastrukturplattformen. Aus der Differenz von Soll-Zustand und Ist-Zustand der Infrastruktur werden notwendige API-Aufrufe abgeleitet. Diese werden unter Ansible durch Module implementiert [53], welche in Form von Collections in einer Registry wie zum Beispiel Ansible Galaxy [54] verteilt werden können. [55] Unter Terraform werden diese durch Provider implementiert [56], die in einer Registry wie zum Beispiel der Terraform Registry [57] verteilt werden können. Es kann nun Untersucht werden, für welche Infrastrukturplattformen Module beziehungsweise Provider zur Verfügung stehen um so die Kompatibilität zu beurteilen.

Tabelle 7 fasst die Unterstützung von Infrastrukturplattformen durch Ansible beziehungsweise Terraform zusammen. Für die drei großen Provider Microsoft Azure, Amazon AWS und Google Cloud existiert bei beiden Tools eine offiziell betreute und gewartete Unterstützung. Unter Ansible existieren verschiedene Formen von Collections. So werden solche von manchen Providern selbst betreut. In diesen Fällen ist mit einer bestmöglichen Qualität zu rechnen. Für DigitalOcean existiert eine durch die Community betreute Collection. Für OVH existiert eine solche zwar auch, jedoch ist die Qualität fragwürdig. Unter Terraform wird durch die HashiCorp ein Verifzierungsprozess durchgeführt. Die Qualität von verifizierten Providern von offizieller Seite geprüft. [74] Hierbei ist ebenfalls von einer bestmöglichen Qualität zu rechnen. Aus diesem Grund ist die Gesamtsituation der Unterstützung der betrachteten Provider bei Terraform als besser zu bewerten.

	Ansible Collection	Terraform Provider
Microsoft Azure	X+ [58]	X+ [59]
Amazon AWS	X+ [60]	X+ [61]
Google Cloud	X+ [62]	X+ [63]
DigitalOcean	X [64]	X!* [65]
Vultr	X! [66]	X!* [67]
Linode	X! [68]	X!* [69]
OVHcloud	(X) [70]	X!* [71]
Hetzner cloud	X+ [72]	X!* [73]

Tabelle 7: Zusammenfassung der Unterstützung von Infrastrukturplattformen durch Ansible und Terraform
X+ = Offizielle Unterstützung
X* = Verifizierte Unterstützung
X! = Unterstützung durch Provider selbst
X = Unterstützung durch Community
(X) = Unterstützung durch Community (Qualität unklar bzw. fragwürdig)

Da Linode unter beiden Tools eine gute Unterstützung aufweist, bestätigt sich damit erneut die Wahl von Linode als Infrastrukturplattform für die Implementierung der Fallstudie innerhalb dieser Arbeit.

3.2.3 Umsetzung

Im Zuge dieser Arbeit wurde die Bereitstellung von Infrastruktur auf Basis der Linode Kubernetes Engine (LKE) durch Ansible und Terraform implementiert. Dieser Abschnitt beschreibt das Vorgehen bei dieser Implementierung. Am Ende wird das konzeptionelle Vorgehen bei einer Umsetzung auf IaaS-Basis beschrieben.

Ansible Ansible kennt das Konzept von Roles als eine Einheit von wiederverwendbaren Inhalten. [43] Zu dem Zweck der Bereitstellung von Infrastruktur auf der LKE wird eine solche Role implementiert. Das CLI-Tool "ansible-galaxy" hilft bei der Erstellung der notwendigen Struktur eines Role. [75] Innerhalb dieser Role werden unter Verwendung der Linode-Collection [68] zwei Tasks implementiert. Einer erzeugt über das Modul `linode.cloud.lke_cluster` eine Ressource, ein Weiterer zerstört diese. Diese Ressource repräsentiert ein LKE-Cluster inklusive darin enthaltener Node-Pools. Der Output dieser Ressource, die unter Anderem Daten zur Authentifizierung mit dem Cluster enthält wird in ein Register gespeichert. [76] Mehr zur Authentifizierung mit Clustern in Abschnitt 3.4 "Container-Orchestrierung". Listing 1 zeigt den Task zur Erstellung der Ressource. Die Authentifizierungsdaten werden in einem weiteren Task in eine Datei geschrieben. Das Modul `linode.cloud.lke_cluster` benötigt zur Authentifizierung mit Linode einen API-Key. Die Ansible-Collection von Linode implementiert, dass dieser über eine Umgebungsvariable konfiguriert werden kann. [77]

Letztendlich ist also ein Playbook implementiert worden, dass ein LKE-Cluster unter Angabe eines API-Keys bereitstellt und Authentifizierungsdaten in einer Datei speichert.

Terraform Terraform kennt das Konzept von Modulen als eine Einheit von wiederverwendbaren Inhalten. [78] Zu dem Zweck der Bereitstellung von Infrastruktur auf der LKE wird ein solches Modul implementiert. Anders als bei Ansible existiert kein Tool zur Erzeugung der Struktur von Modulen. Diese umfasst jedoch nur wenige Dateien. [78] Innerhalb dieses Moduls werden unter Verwendung des Linode-Providers [69] notwendige Ressourcen spezifiziert. Die Ressource `linode_lke_cluster` repräsentiert ein LKE-Cluster inklusive darin enthaltener Node-Pools. [79] Sie liefert als Output unter Anderem Daten zur Authentifizierung mit dem Cluster. Listing 2 zeigt die Ressource. Die Authentifizierungsdaten werden in eine Datei geschrieben, welche durch eine weitere Ressource repräsentiert wird. Der Provider benötigt zur Authentifizierung mit Linode einen API-Key. Innerhalb der Terraform-Moduls wird eine Variable für diesen definiert und der Provider mit dieser konfiguriert. Terraform ermöglicht ein Setzen von Variablen über Umgebungsvariablen. [52]

Letztendlich ist also ein Modul implementiert worden, dass ein LKE-Cluster unter Angabe eines API-Keys bereitstellt und Authentifizierungsdaten in einer Datei speichert.

```yaml
- name: Create cluster
  linode.cloud.lke_cluster:
    label: "cluster"
    k8s_version: 1.23
    region: eu-central
    node_pools:
      - type: g6-standard-2
        count: 3
    state: present
  register: cluster_res
```

Listing 1: Task zur Bereitstellung eines LKE-Clusters in Ansible via YAML

```
resource "linode_lke_cluster" "cluster" {
  label       = "cluster"
  k8s_version = "1.23"
  region      = "eu-central"

  pool {
    type  = "g6-standard-2"
    count = 3
  }
}
```

Listing 2: Ressource zur Bereitstellung eines LKE-Clusters in Terraform via HCL

Listing 1 und 2 zeigen die Erstellung eines LKE-Clusters in Ansible beziehungsweise Terraform. Es wird deutlich, dass sich die Spezifikationen prinzipiell sehr ähnlich sind.

```
resource:
- linode_lke_cluster:
  - cluster:
    - label: "cluster"
      k8s_version: 1.23
      region: eu-central
      pool:
        - type: g6-standard-2
          count: 3
```

Listing 3: Ressource zur Bereitstellung eines LKE-Clusters in Terraform (YAML-Äquivalent)

Wie zuvor erwähnt, lassen sich Ressourcen in Terraform auch in YAML-Äquivalenten spezifizieren. Listing 3 zeigt das YAML-Äquivalent von Listing 2. Wenn auch eine starke Ähnlichkeit mit Listing 1 besteht, ist eine unmittelbare Wiederverwendung für Ansible nicht möglich. Prinzipiell ließe sich diese Konvertierung in beide Richtungen automatisieren.

Umsetzung auf IaaS-Basis Sowohl die Ansible-Collection als auch der Terraform-Provider von Linode bieten ein Bereitstellen von herkömmlichen Computing-Ressourcen. Unter Ansible existiert hier das Modul `instance` [80], unter Terraform die Ressource `linode_instance`. [81] Auf dieser ist dann eine Orchestrierungs-Software wie Kubernetes zu installieren. Die CNCF benennt 68 Distributionen für Kubernetes. [14] Es ist kein guter Grund denkbar, der einen solchen Aufwand rechtfertigt. Ein solches Vorgehen kann in Situationen erforderlich sein, in denen eine Infrastrukturplattform ohne CaaS-Unterstützung verwendet wird. Im Zuge dieser Arbeit wird auf eine Implementierung dieser Strategie verzichtet.

3.3 Continuous Deployment für Infrastruktur

3.3.1 Anforderungen

Morris beschreibt ein grundlegendes Designschema von Pipelines. Dieses umfasst ein Unterteilen in Stages. Es sind automatisch auszulösende und manuell auszulösende Stages zu unterscheiden. Build- und Test-Stages sollen automatisch laufen. Es sollte möglich sein, ein Übertragen in den Produktivbetrieb manuell auszulösen. Weiterhin sollten nach erfolgreicher Ausführung Konfigurationsartefakte veröffentlicht werden können. [7] Dieses Schema hilft, funktionale Anforderungen an CD-Software herzuleiten. Weiterhin muss eine Solche dem Anwendungsfall genügen. Es müssen also Ansible beziehungsweise Terraform ausführbar sein.

3.3.2 Toolauswahl

Push vs. Pull In Abschnitt 1.1 "Prinzipien von GitOps" wird bereits der Pull-basierte Ansatz für Continuous Deployment beschrieben. Hierbei rufen Agents den Soll-Zustand von Ressourcen selbstständig ab. Der Pull-Ansatz benötigt somit unbedingt Agents.

Der Gegenentwurf hierzu ist der Push-Ansatz. Hierbei werden Änderungen von Außen auf Ressourcen angewendet. Wie in Abschnitt 3.2 "Bereitstellung der Infrastruktur" erläutert, basiert die Fallstudie in dieser Arbeit auf ein agentless Provisioning von Infrastruktur. Deshalb muss für Continuous Deployment von Infrastruktur der Push-Ansatz implementiert werden.

Die CNCF benennt in der Klasse "Continuous Integration & Delivery" 50 Tools und Technologien. [14] Weiterhin führt die CNCF ein Reifegradmodell für Technologien, welches diese anhand von Gütekriterien in die Klassen "Incubating Project" oder "Graduated Project" einteilt. [82] Innerhalb der CI/CD-Tools existieren drei auf der Stufe eines "Incubating Project". Namentlich sind dies Argo, Flux und Keptn. [14] Alle diese Tools implementieren den Pull-Ansatz innerhalb von Kubernetes-Clustern. [83, 84, 85] Sie können deshalb in dem Kontext von CD für Infrastruktur im Zuge dieser Fallstudie nicht zum Einsatz kommen. Zwar können diese prinzipiell trotzdem genügen. Jedoch besteht hier wieder ein Bootstrapping-Problem, da bereits Infrastruktur benötigt wird um weitere zur Verfügung zu stellen.

Weiterhin werden von der CNCF CI/CD-Integrationen innerhalb von Git-Plattformen genannt. Hier ist GitLab CI und GitHub Actions zu nennen. [14] Bei beiden dieser Integrationen werden Pipelines durch Runner ausgeführt [86, 87] was ein Implementieren des Push-Ansatzes ermöglicht. GitHub Actions und GitLab CI befinden sich bei Stackshare in der Kategorie "Continuous Integration" auf Platz 5 beziehungsweise 6 der meistgenutzten Technologien. Die ersten drei Plätze werden von Jenkins, Travis CI und CircleCI belegt. [88] Diese drei Tools werden auch von der CNCF genannt. [14]

Jenkins ist eine Open Source CI-Plattform. Auf GitHub besitzt das Projekt über 19.000 "Starrings" und über 700 Contributors. [89] Travis CI und CircleCI sind proprietäre CI-Plattformen. [90, 91] Alle diese Tools führen Pipelines in Runnern aus [92, 93, 94] und eignen sich deshalb ebenfalls für das Implementieren des Push-Ansatzes.

GitLab CI bietet cloud-basierte SaaS-Angebote, die Möglichkeit eigene Runner zu betrieben oder die gesamte Plattform in einer kostenlosen Community- oder zahlungspflichtigen Enterprise-Edition selbst zu hosten. Da es eine unmittelbare Integration mit GitLab als Git-Plattform darstellt können Pipelines nur auf Basis von in GitLab befindlichen Repositories ausgeführt werden. [95] Die Community-Edition von GitLab ist Open Source. [96] GitHub Actions ist eine unmittelbare Integration mit GitHub als Git-Plattform und nur als cloud-basiertes SaaS-Angebot verfügbar. Es lassen sich jedoch Runner selbst betrieben. [97] Auch hier können Pipelines nur aus Basis von in GitHub befindlichen Repositories ausgeführt werden. [98] Jenkins bietet eine kostenlos selbst hostbare CI-Plattform. [99] Travis CI und CircleCI bieten cloud-basierte SaaS-Angebote. [90, 91] Beide Produkte bieten auch die Möglichkeit unter Lizenzierung auf eigener Infrastruktur betrieben zu werden. [100, 91] Sowohl Jenkins [101], Travis CI [102] als auch CircleCI [103] bieten eine Integration mit GitHub. Jenkins [104] und Travis CI [102] bieten zudem eine Integration mit GitLab.

Es gilt zu bewerten ob innerhalb von Runnern Ansible beziehungsweise Terraform genutzt werden kann. Eine Ausführung in einer containerisierten Umgebung würde genügen, denn

so können Images mit beliebigen Abhängigkeiten genutzt werden. Alternativ kann eine direkte Integration von Runnern mit Ansible beziehungsweise Terraform ebenfalls genügen. GitLab CI [105], GitHub Actions [106], Jenkins [107] und CircleCI [108] bieten eine direkte Unterstützung der Ausführung von Jobs in Docker. Travis CI erlaubt ein Ausführen von Docker-Containern in CI-Jobs. [109] Damit erlauben alle Tools ein Ausführen von Ansible und Terraform.

Alle genannten Tools bieten die Möglichkeit Build-Artefakte zu exportieren. [110, 111, 112, 113, 114] Damit erfüllt sie die Anforderung nach Ausführung Konfigurationsartefakte veröffentlichen zu können.

Alle genannten Tools erlauben weiterhin eine Konfiguration von Jobs in Pipelines durch Variablen. [115, 116, 117, 118, 119] Dies sorgt für eine Externalisierung von Konfiguration und einer Zentralisierung von möglicherweise sensiblen Daten.

GitLab CI [120], GitHub Actions [121], Travis CI [122] und CircleCI [123] erlauben eine Spezifikation von Pipelines durch YAML-Manifeste. Jenkins nutzt hierzu ein eigenes Format. [124] Damit erlauben alle Tools ein deklaratives Definieren von Pipelines und verfolgen einen "Everything as Code"-Ansatz.

	GitLab CI	GitHub Actions	Jenkins	Travis CI	CircleCI
Features					
Runnerbasierte Ausführung	X	X	X	X	X
Ausführung in Docker	X	X	X	(X)	X
Build-Artefakte	X	X	X	X	X
Variablen	X	X	X	X	X
"as Code"-Spezifikation	X	X	X	X	X
Betrieb					
SaaS	X	X	-	X	X
Self-hosted Plattform	X	-	X	X	X
Self-hosted Runner	X	X	X	X	X
Integration					
GitLab	X	-	X	X	X
GitHub	-	X	X	X	-
Open Source	(X)	-	X	-	-

Tabelle 8: Zusammenfassung von Eigenschaften von CI/CD-Tools
X = Gegeben
(X) = Eingeschränkt gegeben

Tabelle 8 fasst die Eigenschaften der untersuchten CI/CD-Tools zusammen. Es zeigt sich beim Vergleich der Features ein einheitliches Bild bei der Leistungsfähigkeit der betrachteten Tools. Somit sind alle im Kontext dieser Fallstudie zur Umsetzung von GitOps als Betriebsmodell geeignet. Zur konkreten Implementierung soll GitLab CI zum Einsatz kommen, da zur Versionierung bereits auf GitLab gesetzt wird und so ein Aufwand für Integration gänzlich entfällt.

3.3.3 Umsetzung

In GitLab werden Jobs innerhalb von Stages ausgeführt. Mehrere Stages setzen eine Pipeline zusammen. Jobs innerhalb einer Stage werden parallel ausgeführt. [125]

Ansible Es ist ein Docker-Image zu definieren, welches als Basis zur Ausführung von Jobs dienen soll. Es existiert ein offizielles Docker-Image mit Ansible. [126] Jedoch enthält dieses keine Shell zur Ausführung von Job-Scripten innerhalb einer Pipeline. Deshalb wurde zur Implementierung dieser Fallstudie ein eigenes Docker-Image definiert. Dieses basiert auf Alpine und enthält Ansible sowie bereits die ohnehin notwendige Linode-Collection. Dadurch, dass dieses Docker-Image alle notwendigen Abhängigkeiten vollständig enthält muss innerhalb von Pipeline-Jobs keine weitere Betrachtung dieser vorgenommen werden.

Es werden nun zwei Stages definiert "validate" und "deploy" in denen jeweils ein Job "check" beziehungsweise "play-create" auszuführen ist. Innerhalb des Check-Jobs wird das Ansible-Playbook mit Hilfe des Check-Mode des ansible-playbook CLI-Tools geprüft. [127] Im zweiten Job wird nun bei erfolgreicher Ausführung des ersten Jobs das Playbook tatsächlich ausgeführt und Infrastrukturressourcen somit erzeugt. In Abschnitt 3.2 "Bereitstellung der Infrastruktur" wird beschrieben, dass das Playbook einen API-Key von Linode als Umgebungsvariable benötigt und Authentifizierungsdaten in Form einer Datei erzeugt. Umgebungsvariablen können unter GitLab zentral in den CI/CD-Einstellungen eines Repositories definiert und gesetzt werden. Die Datei wird als Build-Artefakt exportiert und steht damit in der Pipeline-Übersicht zum Download zur Verfügung. Die Ausführung der Jobs wird weiterhin an die Bedingung gebunden, dass ein Tag vorliegt, der die Zeichenkette "deploy" enthält. Der Create-Job muss zudem manuell ausgelöst werden. So wird bei einem normalen Commit keine Aktion ausgeführt. Bei Erstellung eines entsprechenden Tags wird eine Pipeline erzeugt und dessen erster Job (check) ausgeführt. Ist dieser erfolgreich kann ein zweiter Job (play-create) manuell ausgeführt werden.

```
stages:
  - validate
  - deploy

check:
  stage: validate
  image: leonbrandt/ansible-linode:latest
  script:
    - ansible-playbook --check create.yml
  rules:
```

```
    - if: $CI_COMMIT_TAG =~ /^.*deploy.*$/

play-create:
  stage: deploy
  image: leonbrandt/ansible-linode:latest
  script:
    - ansible-playbook create.yml
  artifacts:
    paths:
      - .kubeconfig
  rules:
    - if: $CI_COMMIT_TAG =~ /^.*deploy.*$/
      when: manual
```

Listing 4: Pipeline-Spezifikation unter Ansible

Listing 4 zeigt die beschriebene Spezifikation der Infrastruktur-Deployment-Pipeline unter Ansible.

Terraform GitLab CI erlaubt bei der Spezifikation von Pipelines ein Importieren von bestehenden Spezifikationen. [128] So können Pipelines durch Erweiterung von Templates konstruiert werden. GitLab stellt zur Nutzung von Terraform ein solches Template zur Verfügung. Dieses basiert auf einem eigens von GitLab gepflegten Docker-Image und spezifiziert unter Anderem die abstrakten Jobs fmt (format), validate, build und deploy. [129] Die Jobs fmt und validate ein Formatieren und Validieren von Terraform-Manifesten. [130, 131] Weiterhin erlaubt das CLI-Tool von Terraform ein Vor-Erzeugen von Ausführungsplänen und ein späteres Anwenden dieser. [132, 133] Dieser Mechanismus wird hier durch die zwei CI-Jobs build und deploy genutzt. Von diesen können nun tatsächliche Jobs abgeleitet werden. Konkret werden in dieser Implementierung wieder wie unter Ansible Artefakte definiert und eine Einschränkung definiert. So sollen die Jobs fmt und validate immer, die Jobs build und deploy nur bei einem Tag und der Job deploy zusätzlich nur manuell ausgeführt werden.

GitLab implementiert ein Terraform-Backend. Die Integration innerhalb von CI-Pipelines ist bei Nutzung des Docker-Images von GitLab trivial. Hierzu muss innerhalb des Terraform-Moduls lediglich die Existenz einer HTTP-Backends spezifiziert werden. Innerhalb der Pipeline-Spezifikation muss dann über eine Variable der Name der Umgebung spezifiziert werden. [134] Dies ermöglicht ein Deployment in zum Beispiel eine getrennte Test- und Produktiv-Umgebung.

```
include:
  - template: Terraform/Base.gitlab-ci.yml

variables:
  TF_ROOT: "cluster"
  TF_STATE_NAME: "default"
```

```
stages:
  - validate
  - build
  - deploy

fmt:
  extends: .terraform:fmt
  needs: []

validate:
  extends: .terraform:validate
  needs: []

build:
  extends: .terraform:build
  rules:
    - if: '$CI_COMMIT_TAG =~ /^.*deploy.*$/'

deploy:
  extends: .terraform:deploy
  dependencies:
    - build
  environment:
    name: $TF_STATE_NAME
  artifacts:
    paths:
      - ${TF_ROOT}/.kubeconfig
  rules:
    - if: '$CI_COMMIT_TAG =~ /^.*deploy.*$/'
      when: manual
```

Listing 5: Pipeline-Spezifikation unter Terraform

Listing 5 zeigt die beschriebene Spezifikation der Infrastruktur-Deployment-Pipeline unter Terraform.

3.4 Container-Orchestrierung

Wie in Abschnitt 1.3.3 "Tools zur Konfiguration von Servern" beschrieben übernimmt Software zur Orchestrierung von Containern das Management von Containern und deren Ausführung über eine Menge von Hosts. Diese Menge von Hosts (auch Nodes) wurde mit der Bereitstellung von Infrastruktur (Abschnitt 3.2 "Bereitstellung der Infrastruktur") bereits erzeugt und zur Verfügung gestellt.

Die CNCF benennt in der Toolklasse "Scheduling & Orchestration" 19 Technologien. Zwei davon besitzen die Klassifikation als "Incubating Project". Namentlich sind dies Crossplane und Volcano. Auch befindet sich in dieser Klasse ein als "Graduating Project"

klassifiziertes Produkt: Kubernetes. [14] Kubernetes hat sich zu einer Art De-facto-Standard zur Container-Orchestrierung entwickelt. [135] Aus diesem Grund soll im Zuge dieser Arbeit auf eine genauere Betrachtung von Alternativen verzichtet werden und die Nutzung von Kubernetes als Prämisse gelten. Bereits in vorherigen Abschnitten (unter Anderem Abschnitt 3.2 "Bereitstellung der Infrastruktur") wurde deshalb impliziert, dass Kubernetes in Form eines CaaS-Angebots zum Einsatz kommt.

Eine Container-Orchestrierungs-Software implementiert das in Abschnitt 1.1 "Prinzipien von GitOps" beschriebene Konzept der Reconciliation. In Kubernetes geschieht dies über die "Control Plane". [136] Diese kann im Cluster selbst ausgeführt werden. Beim CaaS-Ansatz wird diese vom Provider der Infrastrukturplattform verwaltet. Dies ist bereits in Abbildung 2 dargestellt.

Der Umgang mit Kubernetes wird im nächsten Abschnitt (3.5 "Bereitstellung der Software") genauer beschrieben.

3.5 Bereitstellung der Software

3.5.1 Anforderungen

Wie auch in Abschnitt 3.2 "Bereitstellung der Infrastruktur" beschrieben besteht die Anforderung an die Umsetzung eines IaC-Ansatzes. Weiterhin muss eine programmierbare Schnittstelle vorliegen. Zur unbeaufsichtigten Ausführung bietet sich auch hier ein Commandline-Interface (CLI) an. Zuletzt muss Konfiguration externalisiert sein.

3.5.2 Toolauswahl

Aus der in Abschnitt 3.4 "Container-Orchestrierung" beschriebenen Nutzung von Kubernetes ist eine Toolauswahl gewissermaßen implizit vorbestimmt. In Kubernetes stellt die Control Plane eine API zur Verwaltung von Ressourcen zur Verfügung. [137] Mit dieser kann über das CLI-Tool kubectl interagiert werden. [138] Damit sind die Anforderungen an eine API sowie ein CL-Interface vollständig erfüllt. Kubernetes sieht das Konzept von ConfigMaps zur Verwaltung von Konfiguration vor. [139] Zwar ist es möglich ConfigMap-Objekte in dedizierten Manifest-Dateien zu verwalten, jedoch ist eine Externalisierung von Konfiguration damit nur bedingt gegeben, da diese unmittelbar von Kubernetes selbst verwaltet wird.

Tabelle 9 fasst die Haupteigenschaften der betrachteten Ansätze beziehungsweise Tools zur Softwarebereitstellung zusammen. Es zeigt sich, dass Helm alle Eigenschaften am Besten erfüllt.

In Abschnitt 2.4 "Zu betrachtende Charakteristiken und Metriken" wird beschrieben, wie die Eigenschaften Wartbarkeit und Portabilität im Bezug auf IaC-Manifeste von hoher Bedeutung sind. Zur Verbesserung dieser Eigenschaften sowie zur Herstellung der Externalisierung von Konfiguration existieren die Ansätze des Patching und des Templating. Diese gehen vor Allem aus den Tools Kustomize [140] (Patching) sowie Helm [141] (Templating) hervor. Der nächste Abschnitt beschreibt das Vorgehen bei Verwendung dieser Tools zur Umsetzung dieser Anforderungen.

	Kubernetes-Manifeste	Kustomize	Helm
IaC	X	X	X
CLI	X	X	X
Externe Konfiguration	-	(X)	X

Tabelle 9: Zusammenfassung der Haupteigenschaften von Ansätzen beziehungsweise Tools zur Softwarebereitstellung
X = Eigenschaft gegeben
(X) = Eigenschaft bedingt gegeben
- = Eigenschaft nicht gegeben

Weiterhin soll die Kompatibilität mit Ansible sowie Terraform untersucht werden. Hier ist ein starker Einfluss auf Eigenschaften wie Konsistenz zu erwarten. Die Möglichkeit der Nutzung dieser Tools in allen Bereichen der GitOps-Architektur ermöglicht ein hohes Maß an Standardisierung.

3.5.3 Umsetzung

IaC-Manifeste Ein IaC-Ansatz kann in Kubernetes durch ein Ablegen von Request-Payloads entsprechend der Kubernetes-API in YAML-Dateien geschehen. Diese können dann über das CLI-Tool kubectl angewendet werden. Bei diesem Vorgehen müssen vollständige Manifeste für jedes Objekt definiert werden. [142]

Patching (via Kustomize) Unter Annahme, dass eine Kubernetes-Instanz Ressourcen verwaltet können diese durch Patches in-place modifiziert werden. [143] Dies entspricht einem imperativen Patching-Mechanismus.

Kustomize (seit Kubernetes v1.14 in kubectl) integriert ermöglicht ein Patching von Kubernetes-Manifesten. Hierzu werden bestehende Manifeste (Base) mit sogenannten Overlays (Patches) gepatcht. Overlays enthalten nur anzuwendende Differenzen von der Base. [144]

Templating via Helm Helm verfolgt den Ansatz des Templating. Hierbei werden alle konfigurierbaren Parameter von Kubernetes-Manifesten durch Variabel-Platzhalter ersetzt. Diese Variablen werden in einer Werte-Datei gesetzt. [145] Templates und (Default-)Werte werden in sogenannten Charts zusammengefasst. Charts beschreiben das Paket-Format unter Helm. Diese können in Repositories veröffentlicht werden. Charts werden zu Releases in Kubernetes-Cluster installiert und bei Bedarf vorher durch Änderung von

Werte-Dateien angepasst. Releases können komfortabel aktualisiert oder deinstalliert werden. [146]

Im Zuge der Implementierung der Fallstudie innerhalb dieser Arbeit sind für alle Komponenten Helm-Charts definiert beziehungsweise vorhandene verwendet worden. Das CLI-Tool "helm" erleichtert ein Erzeugen von allen notwendigen Objekt-Templates zur Bereitstellung üblicher Services erheblich. [147] Die Package-Registry von GitLab ist kompatibel mit Helm-Charts. [148] In den CI/CD-Pipelines der für die fachlichen Architektur der Fallstudie notwendigen Komponenten wurden in entsprechenden Repositories befindliche Helm-Charts in der GitLab-Registry der Projekt veröffentlicht. Hierzu wurde das Helm-Plugin `helm-push` zur Hilfe genommen. [149] Zunächst musste hierfür wieder ein Docker-Image als Basis zur Ausführung von CI-Jobs erzeugt werden.

```
variables:
  CHART_NAME: "api-service"
  HELM_PATH: "deploy/helm"

helm:
  stage: build
  image: leonbrandt/helm:latest
  variables:
    CHANNEL: "stable"
  before_script:
    - helm repo add \
      --username gitlab-ci-token \
      --password ${CI_JOB_TOKEN} \
      gitlab https://gitlab.com/api/v4/projects/
        ${CI_PROJECT_ID}/packages/helm/${CHANNEL}
  script:
    - cd ${HELM_PATH}
    - 'helm cm-push $CHART_NAME/ \
      --version="$CI_COMMIT_TAG" gitlab'
  only:
    - tags
```

Listing 6: CI-Job für Publishing von Helm-Charts in der GitLab-Registry

Listing 6 zeigt entsprechenden CI-Job. Hierbei wird sich weiterhin der Versionierung-Mechanismus von Helm-Charts zu Nutze gemacht. Die Version wird auf den notwendigen Git-Tag gesetzt.

Kompatibilität mit Ansible und Terraform Unter Ansible existiert die Collection `kubernetes.core`. Diese ermöglicht ein Anwenden von Manifesten, die Nutzung von Kustomize sowie Helm. [150] Unter Terraform existiert der offizielle Provider `hashicorp/kubernetes` zur Anwendung von Kubernetes-Manifesten. [151] Zur Nutzung von Helm existiert der offizielle Provider `hashicorp/helm`. [152] Eine Unterstützung von Kustomize besteht nicht.

	Ansible	Terraform
Kubernetes-Manifeste	X	X
Kustomize	X	-
Helm	X	X

Tabelle 10: Zusammenfassung der Kompatibilität von Tools zur Infrastrukturbereitstellung mit Ansätze und Tools zur Softwarebereitstellung
X = Kompatibilität gegeben
- = Kompatibilität nicht gegeben

Tabelle 10 fasst die Kompatibilität von Tools zur Bereitstellung von Infrastruktur mit Ansätzen und Tools zur Bereitstellung von Software zusammen.

Im Zuge der Implementierung innerhalb dieser Arbeit wurde eine Ansible-Role sowie ein Terraform-Modul unter Verwendung von Helm implementiert. Diese erhalten als Variablen notwendige Credentials zur Authentifizierung mit der GitLab-Registry, in der die Helm-Charts abgelegt sind. Weiterhin erhalten diese die bei der Bereitstellung der Infrastruktur erzeugte Datei zur Authentifizierung mit dem Kubernetes-Cluster. Anschließend werden Tasks beziehungsweise Ressourcen zur Erzeugung von Helm-Releases spezifiziert. Alle zur Implementierung der fachlichen Architektur notwendigen Komponenten können so durch Helm-Releases bereitstellt werden. Analog zur Bereitstellung von Infrastruktur kann diese Bereitstellung durch CI/CD-Pipelines in GitLab vorgenommen werden.

```
data "http" "api-service-values" {
  url = local.api_service_values_url

  request_headers = {
    Accept = "application/yaml"
  }
}

resource "helm_release" "api-service" {
  depends_on = [data.http.api-service-values]
  name       = "api-service"

  repository          = local.api_service_repository
  repository_username = var.gitlab_username
  repository_password = var.gitlab_access_token
  chart               = "api-service"

  values = [
    data.http.api-service-values.response_body
  ]
}
```

Listing 7: Helm-Release in Terraform

Listing 7 zeigt das Erzeugen eines Helm-Releases in Terraform. Komponenten-Repositories enthalten neben Helm-Charts zugehörigen Default-Werte-Dateien. Diese werden für das Release durch eine HTTP-Request abgerufen.

3.6 Continuous Deployment für Software

3.6.1 Anforderungen

Für eine Bereitstellung von Software via Kubernetes ist eine Containerisierung dieser notwendig. Zunächst muss also ein kontinuierlicher Prozess hierfür hergestellt werden. Ansonsten gelten die selben Anforderungen wie in Abschnitt 3.3 "Continuous Deployment für Infrastruktur" beschrieben.

3.6.2 Toolauswahl

Anders als bei Continuous Deployment für Infrastruktur kann hier prinzipiell ein Pull-Basierter Ansatz via Agenten implementiert werden. Dies ist der Fall, weil bereits Infrastruktur zur Ausführung dieser zur Verfügung gestellt ist. Zur Erhaltung von Konsistenz erscheint hier eine Nutzung eines Push-Basierten Ansatzes via GitLab CI sinnvoll.

3.6.3 Umsetzung

Es sind drei CI-Jobs zu implementieren. In einem Ersten wird auf Basis eines Dockerfiles ein Docker-Image gebaut und dieses in einer Docker-Registry veröffentlicht. Hier kommt Dockerhub zum Einsatz.

In einem zweiten Job wird ein Helm-Chart in einer Registry veröffentlicht. Dieser Prozess ist bereits in Listing 6 dargestellt.

In einem dritten Job kann unmittelbar ein Helm-Release ausgelöst werden. So kann unter Nutzung der Cluster-Authentifizierungsdaten und via `helm upgrade` ein Helm-Release aktualisiert werden. [153]

```
deploy:
  image: leonbrandt/helm:latest
  stage: deploy
  script:
    - helm dependency build deploy/helm/api-service
    - helm upgrade
      --values deploy/helm/api-service/values.yaml
      --set image.tag=\${CI_COMMIT_TAG}
      api-service ./deploy/helm/api-service
  only:
    - tags
```

Listing 8: CI-Job für Helm-Release

Listing 8 zeigt entsprechenden CI-Job zur Erzeugung eines Helm-Releases. Dieses Vorgehen verdeutlicht, wie Helm den Prozess der Implementierung eines Push-basierten CD-Prozesses vereinfachen kann.

4 Ergebnis

4.1 Infrastrukturplattform

4.1.1 Funktionale Eignung

Die funktionale Eignung von Infrastrukturplattformen wurde in Abschnitt 3.1 "Infrastrukturplattform" im Detail untersucht. Hierzu wurden 6 Funktionalitäten und eine weitere Charakteristik unter 8 Providern untersucht. Tabelle 5 in Abschnitt 3.1 stellt das Ergebnis dar.

Nun kann die nach ISO 25023 definierte Metrik "Funktionale Vollständigkeit" berechnet werden. Diese repräsentiert den Anteil der implementierten an den spezifizierten Funktionalitäten. Sie berechnet sich mit $X = 1 - A/B$ mit X: Funktionale Vollständigkeit, A: Anzahl fehlender Funktionalitäten, B: Anzahl spezifizierter Funktionalitäten. [19] Zu beachten ist, dass diese Metrik alle Funktionalitäten gleich gewichtet.

Weiterhin kann die Metrik "Funktionale Angemessenheit des Nutzungsziels" von Interesse sein. Diese repräsentiert den Anteil der Funktionalitäten, die dem Erreichen eines Nutzungsziels dienen an den spezifizierten Funktionalitäten. Sie berechnet sich mit $X = 1 - A/B$ mit X: Funktionale Angemessenheit des Nutzungsziels, A: Anzahl fehlender Funktionalitäten zur Erreichung des Nutzungsziels, B: Anzahl notwendiger Funktionalitäten zur Erreichung des Nutzungsziels. [19] Wie bereits in Abschnitt 2.1.2 "Spezifikation der GitOps-Architektur" dargelegt ist CaaS zur Implementierung der Fallstudie nicht notwendig. Diese kann auch auf Basis von IaaS erfolgen. Damit besitzen Infrastrukturplattformen, die CaaS anbieten prinzipiell einen negativen Wert für A, da dies eine zwar nützliche aber nicht notwendige Funktionalität zur Erreichung des Nutzungsziels darstellt. Dies hat einen Wert von > 1 für die funktionale Angemessenheit des Nutzungsziels zur Folge.

	Microsoft Azure	Amazon AWS	Google Cloud	DigitalOcean	Vultr	Linode	OVHcloud	Hetzner cloud
Funktionale Vollständigkeit	1	1	1	1	1	1	1	≈ 0.857
Funktionale Angemessenheit	> 1	> 1	> 1	> 1	> 1	> 1	> 1	1

Tabelle 11: Zusammenfassung der funktionalen Eignung über die untersuchten Infrastrukturplattformen

Tabelle 11 fasst die funktionale Eignung über die untersuchten Infrastrukturplattformen zusammen. Es zeigt sich, dass Hetzner Cloud aufgrund des fehlenden CaaS-Angebots eine geringere funktionale Vollständigkeit als die anderen Infrastrukturplattformen besitzen. Eine funktionale Angemessenheit ist, da ein CaaS-Angebot nicht unbedingt notwendig ist, jedoch bei allen Plattformen vollständig gegeben. Diese ließe sich bei Plattformen

mit > 1 lediglich als "besser" klassifizieren, weshalb entsprechende Darstellung mit > 1 gewählt wurde.

4.1.2 Gebrauchsqualität

Eine Einschätzung der Effektivität geht unmittelbar aus der funktionalen Eignung hervor. Zwar wurde im Zuge dieser Fallstudie nur eine Implementierung auf einer Infrastrukturplattform vorgenommen, jedoch ist bei solchen mit der gleichen funktionalen Eignung keine Abweichung in Effektivität anzunehmen.

Wie in Abschnitt 3.2 "Bereitstellung der Infrastruktur" beschrieben, hätte eine Implementierung auf IaaS-Basis anstelle auf CaaS-Basis weitere notwendige Arbeitsschritte zur Folge. Dies hätte negative Auswirkungen auf die Effizienz des Gesamtziels. Aus diesem Grund kann unter Berücksichtigung, dass eine funktionale Angemessenheit aller untersuchten Infrastrukturplattformen vollständig gegeben ist, argumentiert werden, dass die Effizienz bei der Gebrauchsqualität unter der Hetzner Cloud schlechter ist.

Die ISO 25022 benennt eine Metrik "Kontextabdeckung", die diesen Sachverhalt zusammenfassen kann. Diese repräsentiert den Anteil der Nutzungskontexte in der ein Produkt mit akzeptierbarer Gebrauchstauglichkeit genutzt werden kann. Sie berechnet sich mit $X = A/B$ mit X: Kontextabdeckung, A: Anzahl Nutzungskontexte mit nicht akzeptierbarer Gebrauchstauglichkeit, B: Anzahl Nutzungskontexte. [18] Wenn sowohl eine Implementierung auf IaaS- sowie auf CaaS-Basis als Nutzungskontext angesehen wird und die IaaS-Variante als nicht gebrauchstauglich klassifiziert wird, ergeben sich deutliche Unterschiede in der Kontextabdeckung. Für eine solche Klassifikation ließe sich auf Basis des erheblichen Mehraufwandes argumentieren.

	Microsoft Azure	Amazon AWS	Google Cloud	DigitalOcean	Vultr	Linode	OVHcloud	Hetzner cloud
Effizienz	Basis	Basis	Basis	Basis	Basis	Basis	Basis	Schlechter
Kontextabdeckung	1	1	1	1	1	1	1	0.5

Tabelle 12: Zusammenfassung der Gebrauchsqualität über die untersuchten Infrastrukturplattformen

Tabelle 12 fasst die Gebrauchsqualität über die untersuchten Infrastrukturplattformen zusammen. Der Wert "Basis" unter Effizienz drückt einen konzeptionellen Ausgangswert für Effizienz aus, gegen den relativ verglichen werden soll. Auch hier zeigt sich, dass Hetzner Cloud abweichend vom Gesamtbild schlechter zu bewerten ist.

4.2 Bereitstellungstools für Infrastruktur

4.2.1 Funktionale Eignung

Die funktionale Eignung von Bereitstellungstools wurde in Abschnitt 3.2 "Bereitstellung der Infrastruktur" im Detail untersucht. Hierzu wurden fünf Eigenschaften von zwei Tools untersucht. Drei dieser Eigenschaften sind als Anforderungen zu betrachten. Tabelle 6 in Abschnitt 3.2 stellt das Ergebnis dar.

Da beide Tools alle Anforderungen erfüllen ergibt sich eine funktionale Vollständigkeit gleich eins für beide Tools. Weiterhin sind alle diese Anforderungen unbedingt erforderlich und dienen gleichermaßen der Erreichung des Nutzungsziels. Aus diesem Grund ist die funktionale Eignung hier als zu einem gleichen Grad gegeben zusammenzufassen.

4.2.2 Kompatibilität

In Abschnitt 3.2 "Bereitstellung der Infrastruktur" wurde die Integration der Bereitstellungstools mit den untersuchten Infrastrukturplattformen betrachtet. Tabelle 7 innerhalb dieses Abschnitts stellt das Ergebnis dar. Diese Untersuchung stellt eine unmittelbare Erhebung der Interoperabilität dar. Bei der Kompatibilität zwischen Bereitstellungstools und Infrastrukturplattformen lassen sich verschiedene Grade unterscheiden. So kann eine offizielle, verifizierte, providergestützte oder communitybasierte Integration vorliegen. Bei den ersten drei Varianten ist eine maximale Qualität zu erwarten. Grundsätzlich gibt eine gut gepflegte communitybasierte Integration bei vielen Mitwirkenden Personen keinen Anlass zur Vermutung, dass eine geringe Qualität vorliegt. Es ließe für das Gegenteil argumentieren, da eine große Personenzahl ein hohen Grad an Reaktivität und gegenseitiger Kontrolle ermöglicht. Aus diesem Grund soll die Art der Integration bei der Erhebung von Kompatibilität nicht weiter betrachtet werden. Es ist in einem Fall eine Ausnahme zu machen. So existiert eine communitybasierte Integration, die den Eindruck erweckt, schlecht gewartet und wenig genutzt zu sein. In diesem Fall wird hier angenommen, dass keine Kompatibilität herrscht.

In Abschnitt 3.5 "Bereitstellung der Software" wurde die Kompatibilität von Bereitstellungstools für Infrastruktur mit Bereitstellungstools für Software betrachtet. Tabelle 10 innerhalb dieses Abschnitts stellt das Ergebnis dar.

Es soll nun der Anteil der kompatiblen Infrastrukturplattformen sowie Softwarebereitstellungsansätze und -Tools hinsichtlich der beiden untersuchten Bereitstellungstools für Infrastruktur als Metrik für Kompatibilität herangezogen werden.

	Ansible	Terraform
Kompatibilität mit Infrastrukturplattformen	$7/8 = 0.875$	$8/8 = 1$
Kompatibilität mit Softwarebereitstellung	$3/3 = 1$	$2/3 \approx 0.667$

Tabelle 13: Zusammenfassung der Kompatibilität von Bereitstellungstools für Infrastruktur

Tabelle 13 fasst die Kompatibilität von Bereitstellungstools für Infrastruktur zusammen. Abschließend lässt sich nicht eindeutig sagen, für welches Tool eine bessere Kompatibilität besteht. Hier muss in der Praxis zur Entscheidungsfindung eine individuelle vom Anwendungsfall abhängige Entscheidung getroffen werden.

4.2.3 Useability

Die Produktqualitätseigenschaft Useability umfasst auch die Charakteristik der Erkennbarkeit der Angemessenheit. Wie in Abschnitt 3.2 "Bereitstellung der Infrastruktur" unter "Klassifikation" dargelegt wird der Anwendungszweck von Ansible und Terraform teilweise unterschiedlich kommuniziert und dargestellt. Hierbei wird Ansible als Tool zur Konfiguration und Terraform als Tool zur Bereitstellung verstanden. Diese Arbeit konnte erfolgreich zeigen, dass beide Tools beide Bereiche abdecken.

Die ISO 25023 definiert für diesen Sachverhalt die Metrik "Vollständigkeit der Beschreibung". Diese repräsentiert den Anteil der Nutzungsszenarien, die in der Produktbeschreibung erläutert werden. [19] Seien die Bereitstellung sowie die Konfiguration von Infrastruktur nun die beiden hier relevanten Nutzungsszenarien. Auf der offiziellen Webseite von Terraform wird beschrieben: "Terraform is an infrastructure as code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently. This includes both low-level components like compute instances, storage, and networking, as well as high-level components like DNS entries and SaaS features." [154] Hieraus geht unmittelbar hervor, dass Terraform sowohl zur Bereitstellung als auch Konfiguration von Infrastruktur genügt. Auf der offizielle Webseite von Ansible werden Use-Cases aufgelistet. Der Use-Case Cloud Automation wird beschrieben mit: "Provision instances, networks, and infrastructure with support modules that ensure deployments work across public and private clouds." [155] Hieraus geht unmittelbar hervor, dass Ansible zur Bereitstellung von Infrastruktur genügt. Der Use-Case Configuration Management wird beschrieben mit: "Centralize configuration file management and deployment [...]" [155] Hieraus geht unmittelbar hervor, dass Ansible zur Konfiguration von Infrastruktur genügt. Es zeigt sich also, dass beide Tools beide Nutzungsszenarien abdecken. Es ergibt sich also eine Vollständigkeit der Beschreibung gleich eins.

Die anscheinenden Mängel in der Erkennbarkeit der Angemessenheit lässt sich somit nicht durch Produktqualitätseigenschaften erklären. Zusammenfassend lässt sich festhalten, dass die Erkennbarkeit der Angemessenheit nicht vollumfänglich abschließend beurteilt werden kann.

Eine weitere Charakteristik von Useability ist der Schutz vor Benutzerfehlern. In Abschnitt 3.3 "Continuous Deployment für Infrastruktur" wird dargestellt, dass sowohl Ansible als auch Terraform Mechanismen zur Prüfung von IaC-Manifesten implementieren. Dies ermöglicht es Benutzern Fehler in solchen zu erkennen, bevor diese im Produktivbetrieb mögliche negative Auswirkungen nehmen.

4.2.4 Portabilität

Eine Charakteristik unter der Produktqualitätseigenschaft Portabilität ist Austauschbarkeit. In Abschnitt 3.2 "Bereitstellung der Infrastruktur" wurde beschrieben,

inwiefern sich IaC-Manifeste bei der Benutzung von Ansible und Terraform unterscheiden. So lassen sich unter beiden Tools äquivalente Manifeste spezifizieren, die nahezu gleich sind. Zwar besteht hier keine vollständige Austauschbarkeit von Tools bei Benutzung der selben Manifeste. Die ISO 25023 beschreibt die Metrik "Nutzungsähnlichkeit" jedoch mit dem Anteil der Funktionalität, die ein Benutzer ohne zusätzliches Lernen oder Workarounds nutzen kann. [19] Zur Herstellung von vollständiger Portabilität ist ein Workaround notwendig. Jedoch ist nahezu kein zusätzliches Lernen erforderlich. Aus diesem Grund kann dafür argumentiert werden, dass Portabilität zwischen Ansible und Terraform zu einem geringen Grad vorliegt.

4.2.5 Gebrauchsqualität

Hinsichtlich der Effizienz hat sich im Zuge dieser Arbeit kein Anlass der Annahme ergeben, dass es zwischen den untersuchten Tools hier signifikante Unterschiede gibt.

Wie bereits in Abschnitt 4.2.3 "Useability" erläutert, decken beide Tools die Nutzungsszenarien der Bereitstellung sowie Konfiguration von Infrastruktur ab. Sie sind also hinsichtlich ihrer Kontextabdeckung als gleich zu bewerten.

4.3 Bereitstellungstools für Software

4.3.1 Funktionale Eignung

Die funktionale Eignung von Bereitstellungstools für Software wurde in Abschnitt 3.5 "Bereitstellung der Software" im Detail untersucht. Hierzu wurden drei Eigenschaften von drei Ansätzen beziehungsweise Tools untersucht. Tabelle 9 in Abschnitt 3.5 stellt das Ergebnis dar.

Die Eigenschaft der Externalisierung von Konfiguration liegt zu unterschiedlichen Graden vor. Zur Berechnung der funktionalen Vollständigkeit soll hier angenommen werden, dass bei einer bedingt gegebenen Eigenschaft die Funktionalität "zur Hälfte" (Wert 0.5) implementiert ist.

Da bei der Implementierung der Fallstudie zur Konfiguration der fachlichen Komponenten eine externe Konfiguration vielfach genutzt wurde, ist zur Beurteilung der funktionalen Angemessenheit anzunehmen, dass diese Funktionalität unbedingt notwendig ist.

Tabelle 14 fasst die funktionale Eignung der untersuchten Ansätze und Tools zur Softwarebereitstellung zusammen. Es zeigt sich ein Gefälle. So stellt im Hinblick auf die funktionale Eignung Kustomize als eine bessere Lösung als Kubernetes-Manifeste und Helm wiederum als die beste Lösung heraus.

4.3.2 Kompatibilität

Alle hier betrachteten Tools sind zu einem hohen Grad mit Kubernetes integriert. Es ist also die Verwendung von Kubernetes als Containerorchestrierungsplattform unbedingt vorausgesetzt. Aus diesem Grund ist die Kompatibilität als schlecht zu bewerten. Dies sollte in der Gesamtbetrachtung jedoch keinen hohen Stellenwert einnehmen. Wie in

	Kubernetes-Manifeste	Kustomize	Helm
Funktionale Vollständigkeit	≈ 0.667	≈ 0.833	1
Funktionale Angemessenheit	≈ 0.667	≈ 0.833	1

Tabelle 14: Zusammenfassung der funktionalen Eignung über die untersuchten Ansätze und Tools zur Softwarebereitstellung

Abschnitt 3.4 "Container-Orchestrierung" beschrieben, stellt Kubernetes den De-facto-Standard zur Container-Orchestrierung dar. Deshalb ist innerhalb die Verwendung von Kubernetes als Prämisse dieser Arbeit ohnehin implizit.

4.3.3 Useability

Da Kustomize sowie Helm auf Kubernetes-Manifesten basieren ist eine Kenntnis dieser zur Verwendung von Kustomize und Helm unerlässlich. Bei Betrachtung der Erlernbarkeit ergibt sich daraus bei Kustomize und Helm ein Nachteil. Hier muss zusätzlich zur Verwendung von Kubernetes-Manifesten weiteres Wissen erlangt werden.

Helm bietet eine toolgestütze Template-Generierung. [156] Kubectl hat ein vergleichbares Feature zur Generierung von Kubernetes-Objekten. [157] Dies erhöht die Bedienbarkeit stark, da die Tools so zu einem gewissen Grad selbstbeschreibend wird. Jedoch fällt eine Abschließende Einschätzung der Bedienbarkeit schwer, da Kustomize unmittelbar auf Kubernetes-Manifesten basiert.

Die ISO 23023 beschreibt unter der Charakteristik Bedienbarkeit die Metrik "Rückgängigmachbarkeit". [19] Helm erlaubt ein komfortables Deinstallieren oder Zurückrollen von Releases. Bei Verwendung von Kubernetes-Manifesten oder Kustomize müssten hierzu Kubernetes-Objekte manuell gelöscht werden. Bei dieser Eigenschaft besitzt Helm also einen eindeutigen Vorsprung.

Tabelle 15 fasst die Useability der untersuchten Ansätze und Tools zur Softwarebereitstellung zusammen. Hier kommt ein Wert "Basis" zum Einsatz, mit dem Eigenschaften relativ zueinander verglichen werden können. Es zeigt sich je nach Gewichtung der untersuchten Charakteristiken ein uneindeutiges Bild. Kustomize liegt unter den betrachteten Charakteristiken zurück.

4.3.4 Portabilität

Da sowohl Kustomize als auch Helm auf der Verwendung von Kubernetes-Manifesten basieren, stellen diese gewissermaßen eine Art Adapter für diese dar. Es wird jedoch keine

	Kubernetes-Manifeste	Kustomize	Helm
Erlernbarkeit	Basis	Schlechter	Schlechter
Bedienbarkeit	Unklar		
Rückgängigmachbarkeit	Basis	Basis	Besser

Tabelle 15: Zusammenfassung der Useability über die untersuchten Ansätze und Tools zur Softwarebereitstellung

Kompatibilität zu anderen Umgebungen hergestellt. Aus diesem Grund lässt sich durch diese Betrachtung die Eigenschaft Adaptierbarkeit nicht bewerten. Diese Tatsache sorgt weiterhin dafür, dass keine Austauschbarkeit besteht. Kustomize und Helm stellen keine Alternativen zu Kubernetes-Manifesten sondern Ergänzungen dar. Aus diesen Gründen lässt sich die Portabilität hier nicht bewerten.

4.3.5 Gebrauchsqualität

Bei der Betrachtung der Gebrauchsqualität von Tools zur Bereitstellung von Software ist die Charakteristik Effizienz von besonderem Interesse. Gerade mit Blick auf eine mögliche verbesserte Wiederverwendbarkeit die sich durch Patching oder Templating ergibt, ist anzunehmen, dass sich in solchen Szenarien wiederum ein Vorteil hinsichtlich Effizienz ergibt. Dies geht daraus hervor, dass bei Konfiguration zweier gleicher Infrastrukturen mit geringfügiger Abweichung voneinander bei der Nutzung von Kubernetes-Manifesten zwei vollständige Sets dieser gepflegt werden müssen. Bei einem Patching-Ansatz durch Kustomize können aus einer Basis durch Patches effizient mehrere Umgebungen abgeleitet und individuell konfiguriert werden. Bei einem Templating-Ansatz durch Helm kann dies durch simples Konfigurieren entsprechender Templates geschehen. Es ist unter Kustomize und Helm also hinsichtlich Effizienz ein starker Vorteil anzunehmen. Dies gilt im Fall von Helm besonders bei der Bereitstellung von nicht selbst entwickelten Services. Hier kann im besten Fall auf bereits existierende Charts zurückgegriffen werden. Ein solches Vorgehen hat auch bei der Implementierung im Zuge dieser Arbeit einen deutlichen Gewinn hinsichtlich Zeiteffizienz dargestellt.

Tabelle 16 fasst entsprechenden Zusammenhang zusammen. Abschließend lässt sich festhalten, dass der Ansatz des Verwendens von Kubernetes-Manifesten hinsichtlich Effizienz zurück liegt.

	Kubernetes-Manifeste	Kustomize	Helm
Effizienz	Basis	Besser	Besser

Tabelle 16: Zusammenfassung der Gebrauchsqualität über die untersuchten Ansätze und Tools zur Softwarebereitstellung

4.4 CD-Tools

4.4.1 Funktionale Eignung

Die funktionale Eignung von Tools für Continuous Deployment wurde in Abschnitt 3.3 "Continuous Deployment für Infrastruktur" im Detail untersucht. Hierzu wurden elf Eigenschaften von fünf Tools untersucht. Fünf dieser Eigenschaften sind als notwendige Anforderungen zu verstehen. Tabelle 8 in Abschnitt 3.3 stellt das Ergebnis dar.

In Fällen, in denen Eigenschaften als eingeschränkt gegeben festgestellt wurden, soll dies zur Berechnung der funktionalen Vollständigkeit als "zur Hälfte" (Wert 0.5) implementierte Funktionalität gelten. Alle betrachteten Anforderungen sind im Sinne des Anwendungsfalls innerhalb der implementierten Fallstudie als unbedingt notwendig zu betrachten.

	GitLab CI	GitHub Actions	Jenkins	Travis CI	CircleCI
Funktionale Vollständigkeit	1	1	1	0.8	1
Funktionale Angemessenheit	1	1	1	1	1
Funktionale Vollständigkeit (Betrieb)	1	≈ 0.667	≈ 0.667	1	1

Tabelle 17: Zusammenfassung der funktionalen Eignung über die betrachteten CD-Tools

Tabelle 17 fasst die funktionale Eignung der betrachteten CD-Tools zusammen. Unter Travis CI gilt die Funktionalität der Ausführung in Docker als eingeschränkt gegeben. Die Art der Implementierung in Travis CI genügt jedoch dem innerhalb der implementierten Fallstudie relevanten Anwendungsfall. Aus diesem Grund besitzt Travis CI eine vollständige funktionale Angemessenheit jedoch keine funktionale Vollständigkeit, was auf den ersten Blick unintuitiv wirken mag. Funktionale Vollständigkeit hinsichtlich Betrieb errechnet sich aus jenen Eigenschaften, die mögliche Betriebsmodelle unterscheiden.

Eine Bewertung dieser kann je nach konkreten Anforderungen abweichen. Im Zuge dieser Arbeit hätte sich hieraus eine vollständige funktionale Angemessenheit über alle Tools ergeben.

4.4.2 Kompatibilität

Zur Bewertung der Kompatibilität sind zwei Blickwinkel zu betrachten. So ist hier sowohl die Kompatibilität hinsichtlich der Ausführung von Pipeline-Jobs sowie die Kompatibilität hinsichtlich Code-Quellen zu untersuchen. Die Eigenschaft der Kompatibilität hinsichtlich Ausführung von Pipeline-Jobs wurde in Abschnitt 4.4.1 "Funktionale Eignung" durch die Funktionalität der Ausführung in Docker bereits implizit betrachtet. Auch hier gilt wieder, dass die Art der Implementierung dieser Funktionalität in Travis CI im Zuge dieser Betrachtung als Ausreichend zu bewerten ist.

Die Integration mit Code-Quellen, konkret Repositories oder genauer Git-Server wurde in Abschnitt 3.3 "Continuous Deployment für Infrastruktur" ebenfalls untersucht. Hier wurde die Kompatibilität der betrachteten CD-Tools mit GitLab sowie GitHub untersucht.

	GitLab CI	GitHub Actions	Jenkins	Travis CI	CircleCI
Ausführung von Pipeline-Jobs	1	1	1	1	1
Code-Repositories	0.5	0.5	1	1	0.5

Tabelle 18: Zusammenfassung der Kompatibilität über die betrachteten CD-Tools

Tabelle 18 fasst die Kompatibilität der betrachteten CD-Tools zusammen. Es zeigen sich Unterschiede bei der Kompatibilität mit Code-Repositories. Bei GitLab CI sowie GitHub Actions ist das damit begründet, dass diese Produkte unmittelbare Integrationen von GitLab beziehungsweise GitHub darstellen.

Im Zuge dieser Arbeit wurde als Code-Repository GitLab sowie als CD-Tool GitLab CI eingesetzt. Aus diesem Grund hatte die Kompatibilität keine weitere Auswirkung.

4.4.3 Useability

Die Eigenschaft des Schutzes vor Benutzerfehlern wird in GitLab durch ein Linting und Validieren [158] der Spezifikationsdateien von CI/CD-Pipelines erreicht. Weiterhin besteht die Möglichkeit zur Visualisierung von Pipeline-Spezifikationen. [158] Dies erlaubt ein umfassendes Überprüfen dieser und somit ein proaktives Verhindern von Benutzerfehlern.

4.5 IaC-Manifeste

4.5.1 Wartbarkeit

Ansible Modularität und Wiederverwendbarkeit wird in Ansible durch Roles hergestellt. Diese bündeln zum Beispiel Tasks und sind in Playbooks importierbar. Sie sind parametrisierbar durch Variablen. [75] Outputs können über globale Variablen gehandhabt werden. [49]

Analysierbarkeit und Testbarkeit wird durch Ansibles diff- beziehungsweise check-mode umgesetzt. Diese zeigen Änderungen am Soll-Zustand der Infrastruktur beziehungsweise führen eine Simulation dieser durch. [127] Diese Funktionen können auch automatisiert in Pipelines zum Einsatz kommen. Dies wurde im Zuge dieser Arbeit auch implementiert.

Terraform Modularität und Wiederverwendbarkeit wird in Terraform durch Module hergestellt. Diese bündeln Ressourcen und sind aus anderen Modulen aufrufbar. Sie sind vollständig durch Variablen parametrisierbar und können Ausgabewerte erzeugen. [78]

Analysierbarkeit und Testbarkeit wird unter Terraform durch die validate, plan beziehungsweise graph-Funktion umgesetzt. Die validate-Funktion führt eine statische Analyse aller Manifeste eines Moduls durch. [131] Die plan-Funktion erzeugt einen "dryrun" durchzuführender Änderungen (vergleichbar zu Ansibles diff-mode). Es ist möglich einen Ausführungsplan zu erzeugen und diesen zu inspizieren. [132] Über die graph-Funktion können solche Ausführungspläne visualisiert werden. [159] Dies zusammen erzeugt ein hohes Maß an Kontrolle über Änderungen an Infrastrukturspezifikationen. Diese Funktionen können auch automatisiert in Pipelines zum Einsatz kommen. Dies wurde im Zuge dieser Arbeit auch implementiert.

Kubernetes (IaC-Manifeste) Modularität ist bei Kubernetes unter Verwendung von IaC-Manifesten implizit gegeben. Es ist immer möglich ein Kubernetes-Objekt durch eine Manifestdatei zu beschreiben. Diese können logisch zusammenhängen. Dies ist dann jedoch fachlichen Anforderungen geschuldet.

Eine Wiederverwendung ist unter Verwendung von IaC-Manifesten nur durch ein Vervielfältigen jener möglich.

Eine Analysierbarkeit muss durch externe Werkzeuge wie kubeval hergestellt werden. Dieses kann Kubernetes-Manifeste validieren. [160]

Eine dynamische Testbarkeit ist durch das Tool `kubetest2` [161] gegeben. [162] Innerhalb dieser Arbeit soll dieses Vorgehen nicht weiter untersucht werden.

Kustomize Der Ansatz der Verwendung von Kustomize verhält sich in seinen Wartbarkeits-Eigenschaften ähnlich wie einfache Kubernetes-Manifeste. Auch ist Modularität implizit gegeben und Analysierbarkeit durch kubeval implementiert.

Wiederverwendbarkeit wird durch den durch Kustomize implementierten Patch-Mechanismus hergestellt. So kann eine Menge an Manifesten (Basis) wiederverwendet werden und durch Overlays (Patches) angepasst beziehungsweise konfiguriert werden.

Helm Unter der Verwendung von Helm ist die Eigenschaft der Modularität neu zu interpretieren. Helm erlaubt ein Packaging von Mengen an Manifesten und liefert ein Interface für diese Pakete (Charts). Unter Helm sind dann keine einzelnen Manifeste mehr zu handhaben, sondern lediglich Charts.

Wiederverwendbarkeit wird innerhalb jener Charts durch einen Templating-Mechanismus hergestellt. Hier können in Manifesten definierte Variablen zentral gesetzt werden. Dies ermöglicht eine komfortable Konfiguration der selben Manifeste für zum Beispiel viele Umgebungen.

Analysierbarkeit wird unter Helm durch eine Lint-Funktion implementiert. Diese führt eine statische Analyse auf Charts durch. [163]

Helm implementiert einen eigenen dynamischen Testmechanismus. Hierbei können für Charts Test spezifiziert werden. Zu diesem Zweck werden innerhalb von Charts Pods definiert, die einen Container ausführen, der zum Beispiel einen Healthcheck durchführt. Bei erfolgreichem Test müssen solche Container mit Code 0 terminieren. Jene Pods müssen entsprechend annotiert sein. Sie erhalten dann CLI-Unterstützung und können mit `helm test <chart>` ausgeführt werden. [164]

	Ansible	Terraform
Modularität	Gegeben	Gegeben
Wiederverwendbarkeit	Gegeben	Gegeben
Analysierbarkeit	Gegeben	Gegeben
Testbarkeit	Gegeben	Gegeben

Tabelle 19: Zusammenfassung der Wartbarkeit über IaC-Manifeste in Tools zur Bereitstellung von Infrastruktur

	Kubernetes-Manifeste	Kustomize	Helm
Modularität	Basis	Besser	Am Besten
Wiederverwendbarkeit	Nicht gegeben	Gegeben	Gegeben
Analysierbarkeit	Gegeben	Gegeben	Gegeben
Testbarkeit	Gegeben	Gegeben	Gegeben

Tabelle 20: Zusammenfassung der Wartbarkeit über IaC-Manifeste in Tools zur Bereitstellung von Software

Zusammenfassung Tabellen 19 beziehungsweise 20 fassen die Wartbarkeit über IaC-Manifeste in den betrachteten Tools zur Bereitstellung von Infrastruktur beziehungsweise Software zusammen. Es zeigt sich über alle Tools ein einheitliches Bild. Eine Verwendung von einfachen Kubernetes-Manifesten hat Nachteile hinsichtlich Modularität und Wiederverwendbarkeit. Helm bietet die beste Modularität durch den Packaging-Ansatz.

4.5.2 Portabilität

Ansible & Terraform Bei der Installierbarkeit ergibt sich zwischen Ansible-Collections beziehungsweise Playbooks und Terraform-Modulen kein beobachtbarer Unterschied. Wie in Abschnitt 4.2.4 "Portabilität" beschrieben, besteht keine unmittelbare Austauschbarkeit zwischen Ansible und Terraform-Manifesten. Aus diesen Gründen sind Ansible und Terraform hinsichtlich Portabilität nicht signifikant zu unterscheiden.

Kubernetes (IaC-Manifeste) & Kustomize & Helm Die in Abschnitt 4.3.4 "Portabilität" beschriebene Tatsache, dass Kustomize und Helm auf Kubernetes-Manifesten basieren sorgt dafür, dass keine Austauschbarkeit besteht.

Die ISO 25023 beschreibt unter Portabilität die Charakteristik Installierbarkeit und wiederum die Eigenschaft der "Bequemlichkeit der Installation". [19] Hier kann also die Installation von IaC-Manifesten unter den drei betrachteten Vorgehensweisen beziehungsweise Tools untersucht werden. Prinzipiell können unter allen Ansätzen mit gleichem Aufwand eine beliebige Menge an Manifesten installiert werden. Nach ISO 25023 gehört zu dieser Eigenschaft auch der Grad zu dem Installationen angepasst werden können. [19] Diese Eigenschaft geht unmittelbar aus der in Abschnitt 4.5.1 "Wartbarkeit" beschriebenen Wiederverwendbarkeit hervor. Somit besitzt Kustomize eine bessere Installierbarkeit als Kubernetes-Manifeste. Helm-Charts besitzen die Beste.

	Kubernetes-Manifeste	Kustomize	Helm
Installierbarkeit	Basis	Besser	Am Besten

Tabelle 21: Zusammenfassung der Portabilität über IaC-Manifeste in Tools zur Bereitstellung von Software

Tabelle 21 fasst die Portabilität der IaC-Manifeste in den untersuchten Tools zur Bereitstellung von Software zusammen. Es zeigt sich, dass Helm beziehungsweise Helm-Charts am Besten abschneiden.

4.6 Zusammenfassung

Infrastrukturplattform

Tabelle 22 fasst die Untersuchung von Infrastrukturplattformen zusammen. Es zeigt sich, dass Hetzner cloud hinter allen Anderen liegt, zwischen denen sich keine Unterschiede zeigen.

Bereitstellungstools für Infrastruktur

Tabelle 23 fasst die Untersuchung von Bereitstellungstools für Infrastruktur zusammen. Es zeigt sich kein klarer Unterschied.

	Microsoft Azure	Amazon AWS	Google Cloud	DigitalOcean	Vultr	Linode	OVHcloud	Hetzner cloud
Funktionale Eignung								
Funktionale Vollständigkeit	1	1	1	1	1	1	1	≈ 0.857
Funktionale Angemessenheit	> 1	> 1	> 1	> 1	> 1	> 1	> 1	1
Gebrauchsqualität								
Effizienz	Basis	Basis	Basis	Basis	Basis	Basis	Basis	Schlechter
Kontext-abdeckung	1	1	1	1	1	1	1	0.5

Tabelle 22: Zusammenfassung der Untersuchung von Infrastrukturplattformen

	Ansible	Terraform
Funktionale Eignung		
Funktionale Vollständigkeit	1	1
Funktionale Angemessenheit	1	1
Kompatibilität		
Kompatibilität mit Infrastrukturplattformen	0.875	1
Kompatibilität mit Softwarebereitstellung	1	≈ 0.667
Useability		
Erkennbarkeit der Angemessenheit	Unklar	
Schutz vor Benutzerfehlern	Gegeben	Gegeben
Portabilität	Gering	
Gebrauchsqualität		
Effektivität	Gleich	

Tabelle 23: Zusammenfassung der Untersuchung von Bereitstellungstools für Infrastruktur

Bereitstellungstools für Software

Tabelle 24 fasst die Untersuchung von Bereitstellungstools für Infrastruktur zusammen. Hier ergibt sich eine klare Rangfolge. So liegt Kustomize vor Kubernetes-Manifesten. Helm wiederum liegt vorne.

CD-Tools

Tabelle 25 fasst die Untersuchung von CD-Tools zusammen. Hier zeigt sich kein klares Bild. Prinzipiell genügen alle Tools dem hier untersuchten Anwendungsfall. Sie unter-

	Kubernetes-Manifeste	Kustomize	Helm
Funktionale Eignung			
Funktionale Vollständigkeit	≈ 0.667	≈ 0.833	1
Funktionale Angemessenheit	≈ 0.667	≈ 0.833	1
Kompatibilität	Nur Kubernetes		
Useability			
Erlernbarkeit	Basis	Schlechter	Schlechter
Bedienbarkeit	Unklar		
Rückgängigmachbarkeit	Basis	Basis	Besser
Gebrauchsqualität			
Effizienz	Basis	Besser	Besser

Tabelle 24: Zusammenfassung der Untersuchung von Bereitstellungstools für Software

	GitLab CI	GitHub Actions	Jenkins	Travis CI	CircleCI
Funktionale Eignung					
Funktionale Vollständigkeit	1	1	1	0.8	1
Funktionale Angemessenheit	1	1	1	1	1
Funktionale Vollständigkeit (Betrieb)	1	≈ 0.667	≈ 0.667	1	1
Kompatibilität					
Ausführung von Pipeline-Jobs	1	1	1	1	1
Code-Repositories	0.5	0.5	1	1	0.5
Useability					
Schutz vor Benutzerfehlern	Gegeben	Nicht untersucht			

Tabelle 25: Zusammenfassung der Untersuchung von CD-Tools

scheiden sich in Details, die individuell abzuwägen sind.

IaC-Manifeste

Tabelle 26 fasst die Untersuchung von IaC-Manifesten von Bereitstellungstools für Infrastruktur zusammen. Es zeigt sich kein Unterschied. Dennoch lässt sich eine gute Eignung

	Ansible	Terraform
Wartbarkeit		
Modularität	Gegeben	Gegeben
Wiederverwendbarkeit	Gegeben	Gegeben
Analysierbarkeit	Gegeben	Gegeben
Testbarkeit	Gegeben	Gegeben
Portabilität		
Installierbarkeit	Gleich	
Austauschbarkeit	Nicht Gegeben	

Tabelle 26: Zusammenfassung der Untersuchung von IaC-Manifesten von Bereitstellungstools für Infrastruktur

feststellen.

	Kubernetes-Manifeste	Kustomize	Helm
Wartbarkeit			
Modularität	Basis	Besser	Am Besten
Wiederverwendbarkeit	Nicht gegeben	Gegeben	Gegeben
Analysierbarkeit	Gegeben	Gegeben	Gegeben
Testbarkeit	Gegeben	Gegeben	Gegeben
Portabilität			
Installierbarkeit	Basis	Besser	Am Besten

Tabelle 27: Zusammenfassung der Untersuchung von IaC-Manifesten von Bereitstellungstools für Software

Tabelle 27 fasst die Untersuchung von IaC-Manifesten von Bereitstellungstools für Software zusammen. Ähnlich wie bei den Bereitstellungstools selbst zeigt sich hier eine Rangfolge. Kustomize liegt vor Kubernetes-Manifesten. Helm vor Kustomize.

5 Fazit

Innerhalb dieser Arbeit werden unterschiedliche Ansätze sowie entsprechende Tools und Technologien in verschiedenen Bereichen zur Umsetzung von GitOps als Betriebsmodell untersucht. Es zeigt sich eine breite prinzipielle Unterstützung des GitOps-Ansatzes bei Infrastrukturplattformen. Auch kann gezeigt werden, dass die aktuell relevanten CI/CD-Tools eine Umsetzung von GitOps erlauben. Es werden Tools zur Bereitstellung und Konfiguration von Infrastruktur untersucht. Mit Ansible und Terraform existieren hier zwei Technologien, die GitOps gut und in gleichem Umfang genügen. Bei der Nutzung von Kubernetes zur Container-Orchestrierung kann gezeigt werden, dass zur Handhabung von IaC-Manifesten ein Patching- oder Templating-Ansatz durch Tools wie Kustomize beziehungsweise Helm signifikante Vorteile bringen können.

Aus den untersuchten Technologien lässt sich nun ein geeigneter Technologiestack konstruieren. So erweist sich eine Kombination aus Ansible oder Terraform mit Kubernetes und Helm sowie GitLab auf einem nahezu beliebigen Cloud-Provider als bestens geeignet um GitOps als Methodologie zu implementieren.

Weiterhin wird deutlich, dass die Wahl des Betriebsmodells von Infrastruktur (IaaS vs. CaaS) hinsichtlich GitOps keine Rolle spielt. Ein Nutzen von CaaS im Bezug auf Effizienz kann nachgewiesen werden. Gerade durch das breite bereits bestehende Angebot an CaaS bei Infrastrukturprovidern erscheint eine Nutzung dieser Services deshalb als sinnvoll.

Wie in Abschnitt 1 "Einleitung" dargestellt, ist der Begriff GitOps zum Zeitpunkt der Verfassung dieser Arbeit etwa fünf Jahre alt. Trotzdem erscheint die entsprechende Toollandschaft bereits heute bestens ausgereift.

5.1 Ausblick

Die Betrachtung innerhalb dieser Arbeit setzt auf Breite hinsichtlich der konzeptionellen Toolklassen innerhalb der GitOps-Toollandschaft sowie dessen Eigenschaften. Dadurch können nicht alle Alternativen innerhalb einer Toolklasse im Detail betrachtet werden. Weiterhin können nicht alle Eigenschaften im vollen Detail untersucht werden. So wird innerhalb dieser Arbeit stellenweise deduktiv Argumentiert, wobei Prämissen stellenweise auf Plausibilität beruhen.

Für eine umfassendere Evaluation des GitOps-Toolstacks sind empirische Untersuchungen durchzuführen. Dies ermöglicht eine detailliertere Betrachtung aller Technologie-Eigenschaften. Hierbei sind weitere Ansätze, wie zum Beispiel Pull-basiertes Continuous Deployment zu betrachten. Auch können alternative Technologien, zum Beispiel zur Container-Orchestrierung betrachtet werden.

Literatur

[1] *CNCF Cloud Native Definition v1.0*. URL: https://github.com/cncf/toc/blob/main/DEFINITION.md (visited on 04/28/2022).

[2] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*. 2017. DOI: 10.1109/access.2017.2685629. URL: http://dx.doi.org/10.1109/ACCESS.2017.2685629.

[3] Florian Beetz, Anja Kammer, and Simon Harrer. *GitOps Cloud-native Continuous*. 1st ed. innoQ Deutschland GmbH, 2021. ISBN: 978-3-9821126-8-8.

[4] *Guide To GitOps*. URL: https://www.weave.works/technologies/gitops (visited on 04/28/2022).

[5] Paul Fremantle and Anita Buehrle. *The GitOps Maturity Model*. weaveworks, 2021.

[6] *GitOps for Cost Efficiency, Compliance, Velocity, Security, and Resilience*. URL: https://www.weave.works/blog/gitops-for-cost-efficiency-compliance-velocity-security-and-resilience (visited on 04/28/2022).

[7] Kief Morris. *Infrastructure as Code*. 1st ed. O'Reilly Media, Inc., 2016. ISBN: 978-1-491-92435-8.

[8] *OpenGitOps: GitOps Principles v1.0.0*. URL: https://github.com/open-gitops/documents/blob/v1.0.0/PRINCIPLES.md (visited on 04/28/2022).

[9] *OpenGitOps: GitOps Glossary v1.0.0*. URL: https://github.com/open-gitops/documents/blob/v1.0.0/GLOSSARY.md (visited on 04/28/2022).

[10] *About OpenGitOps*. URL: https://opengitops.dev/about (visited on 04/28/2022).

[11] Billy Yuen et al. *GitOps and Kubernetes*. 1st ed. Manning Publications Co., 2021. ISBN: 978-1617297274.

[12] Google Cloud: DevOps Research and Assessment (DORA). *State of DevOps 2021*. 2021. URL: https://services.google.com/fh/files/misc/report_2021_accelerate_state_of_devops.pdf (visited on 04/28/2022).

[13] *GitOps for Cost Efficiency, Compliance, Velocity, Security, and Resilience*. 2021. URL: https://www.weave.works/blog/gitops-for-cost-efficiency-compliance-velocity-security-and-resilience (visited on 04/28/2022).

[14] *CNCF Cloud Native Interactive Landscape*. URL: https://landscape.cncf.io (visited on 04/30/2022).

[15] *CNCF Landscape Guide*. URL: https://landscape.cncf.io/guide (visited on 04/30/2022).

[16] Ian Sommerville. *Software Engineering*. 10th ed. Pearson Education Limited, 2016. ISBN: 978-1-292-09613-1.

[17] *ISO/IEC 25010-2011 - International Standard - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Standard. Geneva, CH, Mar. 2011.

[18] *ISO/IEC 25010-2012 - International Standard - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of quality in use*. Standard. Geneva, CH, July 2021.

[19] *ISO/IEC 25023-2016 - International Standard - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality*. Standard. Geneva, CH, June 2016.

[20] *Cloud Wars Top 10: The World's Top Cloud Vendors*. URL: https://accelerationeconomy.com/cloud-wars-top-10 (visited on 06/07/2022).

[21] *Microsoft Azure*. URL: https://azure.microsoft.com (visited on 06/07/2022).

[22] *Amazon AWS*. URL: https://aws.amazon.com (visited on 06/07/2022).

[23] *Google Cloud*. URL: https://cloud.google.com (visited on 06/07/2022).

[24] *DigitalOcean*. URL: https://digitalocean.com (visited on 06/07/2022).

[25] *Vultr*. URL: https://vultr.com (visited on 06/07/2022).

[26] *Linode*. URL: https://linode.com (visited on 06/07/2022).

[27] *OVHcloud*. URL: https://ovhcloud.com (visited on 06/07/2022).

[28] *Hetzner: cloud*. URL: https://hetzner.com/cloud (visited on 06/07/2022).

[29] *Linode Docs: Getting Started on the Linode Platform*. URL: https://www.linode.com/docs/guides/getting-started (visited on 06/07/2022).

[30] *GitHub: ansible/ansible*. URL: https://github.com/ansible/ansible (visited on 06/07/2022).

[31] *GitHub: hashicorp/terraform*. URL: https://github.com/hashicorp/terraform (visited on 06/07/2022).

[32] *Stackshare*. URL: https://stackshare.io (visited on 06/07/2022).

[33] *Stackshare: Ansible*. URL: https://stackshare.io/ansible (visited on 06/07/2022).

[34] *Stackshare: Terraform*. URL: https://stackshare.io/terraform (visited on 06/07/2022).

[35] *Stackshare: Server Configuration and Automation Tools*. URL: https://stackshare.io/server-configuration-and-automation (visited on 06/07/2022).

[36] *Stackshare: Infrastructure Build Tools*. URL: https://stackshare.io/infrastructure-build-tools (visited on 06/07/2022).

[37] *Why use Terraform?* URL: https://www.oreilly.com/content/why-use-terraform (visited on 06/08/2022).

[38] *Ansible: How Ansible works*. URL: https://www.ansible.com/overview/how-ansible-works (visited on 06/08/2022).

[39] *Ansible: Delegation*. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_delegation.html (visited on 06/08/2022).

[40] *Terraform: What is Terraform?* URL: https://www.terraform.io/intro (visited on 06/08/2022).

[41] *Terraform: Using Terraform Cloud with Terraform CLI.* URL: https://www.terraform.io/cli/cloud (visited on 06/08/2022).

[42] *Terraform: Backends.* URL: https://www.terraform.io/language/settings/backends (visited on 06/08/2022).

[43] *Ansible: Concepts.* URL: https://docs.ansible.com/ansible/latest/user_guide/basic_concepts.html (visited on 06/08/2022).

[44] *Ansible: Intro to playbooks.* URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html (visited on 06/08/2022).

[45] *Terraform: Language.* URL: https://www.terraform.io/language (visited on 06/08/2022).

[46] *Terraform: Syntax.* URL: https://www.terraform.io/language/syntax (visited on 06/08/2022).

[47] *YAML Specification version 1.2.2.* URL: https://yaml.org/spec/1.2.2 (visited on 06/08/2022).

[48] *Terraform: CLI.* URL: https://www.terraform.io/cli (visited on 06/08/2022).

[49] *Ansible: Using variables.* URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html (visited on 06/08/2022).

[50] *Terraform: Variables and Outputs.* URL: https://www.terraform.io/language/values (visited on 06/08/2022).

[51] *Ansible: env lookup.* URL: https://docs.ansible.com/ansible/latest/collections/ansible/builtin/env_lookup.html (visited on 06/08/2022).

[52] *Terraform: Environment Variables.* URL: https://www.terraform.io/cli/config/environment-variables (visited on 06/08/2022).

[53] *Ansible: Developing Modules.* URL: https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html (visited on 06/08/2022).

[54] *Ansible Galaxy.* URL: https://galaxy.ansible.com (visited on 06/08/2022).

[55] *Ansible: Developing Collections.* URL: https://docs.ansible.com/ansible/latest/dev_guide/developing_collections.html (visited on 06/08/2022).

[56] *Terraform: Providers.* URL: https://www.terraform.io/language/providers (visited on 06/08/2022).

[57] *Terraform Registry.* URL: https://registry.terraform.io (visited on 06/08/2022).

[58] *Ansible Galaxy: azure/azcollection.* URL: https://galaxy.ansible.com/azure/azcollection (visited on 06/08/2022).

[59] *Terraform registry: hashicorp/azurerm.* URL: https://registry.terraform.io/providers/hashicorp/azurerm/latest (visited on 06/08/2022).

[60] *Ansible Galaxy: amazon/aws*. URL: https://galaxy.ansible.com/amazon/aws (visited on 06/08/2022).

[61] *Terraform registry: hashicorp/aws*. URL: https://registry.terraform.io/providers/hashicorp/aws/latest (visited on 06/08/2022).

[62] *Ansible Galaxy: google/cloud*. URL: https://galaxy.ansible.com/google/cloud (visited on 06/08/2022).

[63] *Terraform registry: hashicorp/google*. URL: https://registry.terraform.io/providers/hashicorp/google/latest (visited on 06/08/2022).

[64] *Ansible Galaxy: community/digitalocean*. URL: https://galaxy.ansible.com/community/digitalocean (visited on 06/08/2022).

[65] *Terraform registry: digitalocean/digitalocean*. URL: https://registry.terraform.io/providers/digitalocean/digitalocean/latest (visited on 06/08/2022).

[66] *Ansible Galaxy: vultr/cloud*. URL: https://galaxy.ansible.com/vultr/cloud (visited on 06/08/2022).

[67] *Terraform registry: vultr/vultr*. URL: https://registry.terraform.io/providers/vultr/vultr/latest (visited on 06/08/2022).

[68] *Ansible Galaxy: linode/cloud*. URL: https://galaxy.ansible.com/linode/cloud (visited on 06/08/2022).

[69] *Terraform registry: linode/linode*. URL: https://registry.terraform.io/providers/linode/linode/latest (visited on 06/08/2022).

[70] *Ansible Galaxy: Search "ovh"*. URL: https://galaxy.ansible.com/search?keywords=ovh (visited on 06/08/2022).

[71] *Terraform registry: ovh/ovh*. URL: https://registry.terraform.io/providers/ovh/ovh/latest (visited on 06/08/2022).

[72] *Ansible Galaxy: hetzner/hcloud*. URL: https://galaxy.ansible.com/hetzner/hcloud (visited on 06/08/2022).

[73] *Terraform registry: hetznercloud/hcloud*. URL: https://registry.terraform.io/providers/hetznercloud/hcloud/latest (visited on 06/08/2022).

[74] *Terraform: Providers*. URL: https://www.terraform.io/registry/providers (visited on 06/08/2022).

[75] *Ansible: Creating Roles*. URL: https://galaxy.ansible.com/docs/contributing/creating_role.html (visited on 06/09/2022).

[76] *Linode Ansible Collection Docs: lke_cluster*. URL: https://github.com/linode/ansible_linode/blob/main/docs/modules/lke_cluster.md (visited on 06/09/2022).

[77] *Linode Docs: Using the Linode Ansible Module to Deploy Linodes*. URL: https://www.linode.com/docs/guides/deploy-linodes-using-ansible (visited on 06/09/2022).

[78] *Terraform: Creating Modules*. URL: https://www.terraform.io/language/modules/develop (visited on 06/09/2022).

[79] *Linode Terraform Provider Docs: linode_lke_cluster*. URL: https://registry.terraform.io/providers/linode/linode/latest/docs/resources/lke_cluster (visited on 06/09/2022).

[80] *Linode Ansible Collection Docs: instance*. URL: https://github.com/linode/ansible_linode/blob/main/docs/modules/instance.md (visited on 06/09/2022).

[81] *Linode Terraform Provider Docs: instance*. URL: https://registry.terraform.io/providers/linode/linode/latest/docs/resources/instance (visited on 06/09/2022).

[82] *CNCF: Gradudated and Incubating Projects*. URL: https://www.cncf.io/projects (visited on 06/10/2022).

[83] *Argo Docs: Quickstart*. URL: https://argoproj.github.io/argo-workflows/quick-start (visited on 06/10/2022).

[84] *Flux Docs*. URL: https://fluxcd.io/docs (visited on 06/10/2022).

[85] *Keptn Docs: Architecture*. URL: https://keptn.sh/docs/concepts/architecture (visited on 06/10/2022).

[86] *GitLab Docs: Runner*. URL: https://docs.gitlab.com/runner (visited on 06/10/2022).

[87] *GitHub Actions Docs: Understanding GitHub Actions*. URL: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions (visited on 06/10/2022).

[88] *Stackshare: Continuous Integration*. URL: https://stackshare.io/continuous-integration (visited on 06/10/2022).

[89] *GitHub: jenkinsci/jenkins*. URL: https://github.com/jenkinsci/jenkins (visited on 06/10/2022).

[90] *Travis CI: Product*. URL: https://www.travis-ci.com/product (visited on 06/10/2022).

[91] *CircleCI: Product*. URL: https://circleci.com/product (visited on 06/10/2022).

[92] *Jenkins Docs: Pipeline*. URL: https://www.jenkins.io/doc/book/pipeline (visited on 06/10/2022).

[93] *Travis CI Docs: Core Concepts*. URL: https://docs.travis-ci.com/user/for-beginners (visited on 06/10/2022).

[94] *CircleCI Docs: Concepts*. URL: https://circleci.com/docs/2.0/concepts (visited on 06/10/2022).

[95] *GitLab: About*. URL: https://about.gitlab.com (visited on 06/10/2022).

[96] *GitLab: gitlab-org/gitlab*. URL: https://gitlab.com/gitlab-org/gitlab (visited on 06/10/2022).

[97] *GitHub Actions Docs: About self-hosted runners*. URL: https://docs.github.com/en/actions/hosting-your-own-runners/about-self-hosted-runners (visited on 06/10/2022).

[98] *GitHub Actions Docs*. URL: https://docs.github.com/en/actions (visited on 06/10/2022).

[99] *Jenkins Docs: Installing*. URL: https://www.jenkins.io/doc/book/installing (visited on 06/10/2022).

[100] *Travis CI: Pricing*. URL: https://www.travis-ci.com/pricing (visited on 06/10/2022).

[101] *Jenkins Docs: Jenkins with GitHub*. URL: https://www.jenkins.io/solutions/github (visited on 06/10/2022).

[102] *Travis CI Docs: Tutorial*. URL: https://docs.travis-ci.com/user/tutorial (visited on 06/10/2022).

[103] *CircleCI: Seamless integration with GitHub*. URL: https://circleci.com/integrations/github (visited on 06/10/2022).

[104] *GitLab Docs: Jenkins integration*. URL: https://docs.gitlab.com/ee/integration/jenkins.html (visited on 06/10/2022).

[105] *GitLab Docs: Run your CI/CD jobs in Docker containers*. URL: https://docs.gitlab.com/ee/ci/docker/using_docker_images.html (visited on 06/10/2022).

[106] *GitHub Actions Docs: Creating a Docker container action*. URL: https://docs.github.com/en/actions/creating-actions/creating-a-docker-container-action (visited on 06/10/2022).

[107] *Jenkins Docs: Using Docker with Pipeline*. URL: https://www.jenkins.io/doc/book/pipeline/docker (visited on 06/10/2022).

[108] *CircleCI Docs: Using the Docker execution environment*. URL: https://circleci.com/docs/2.0/using-docker (visited on 06/10/2022).

[109] *Travis CI Docs: Using Docker in Builds*. URL: https://docs.travis-ci.com/user/docker/#using-a-docker-image-from-a-repository-in-a-build (visited on 06/10/2022).

[110] *GitLab Docs: Caching in GitLab CI/CD*. URL: https://docs.gitlab.com/ee/ci/caching (visited on 06/10/2022).

[111] *GitHub Actions Docs: Storing workflow data as artifacts*. URL: https://docs.github.com/en/actions/using-workflows/storing-workflow-data-as-artifacts (visited on 06/10/2022).

[112] *Jenkins Docs: Recording tests and artifacts*. URL: https://www.jenkins.io/doc/pipeline/tour/tests-and-artifacts (visited on 06/10/2022).

[113] *Travis CI Docs: Uploading Artifacts on Travis CI*. URL: https://docs.travis-ci.com/user/uploading-artifacts (visited on 06/10/2022).

[114] *CircleCI Docs: Storing Build Artifacts*. URL: https://circleci.com/docs/2.0/artifacts (visited on 06/10/2022).

[115] *GitLab Docs: GitLab CI/CD variables*. URL: https://docs.gitlab.com/ee/ci/variables (visited on 06/10/2022).

[116] *GitHub Actions Docs: Environment variables*. URL: https://docs.github.com/en/actions/learn-github-actions/environment-variables (visited on 06/10/2022).

[117] *Jenkins Docs: Using environment variables*. URL: https://www.jenkins.io/doc/pipeline/tour/environment (visited on 06/10/2022).

[118] *Travis CI Docs: Environment Variables*. URL: https://docs.travis-ci.com/user/environment-variables (visited on 06/10/2022).

[119] *CircleCI Docs: Using Environment Variables*. URL: https://circleci.com/docs/2.0/env-vars (visited on 06/10/2022).

[120] *GitLab Docs: The .gitlab-ci.yml file*. URL: https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html (visited on 06/10/2022).

[121] *GitLab Actions Docs: Quickstart for GitHub Actions*. URL: https://docs.github.com/en/actions/quickstart (visited on 06/10/2022).

[122] *Travis CI Docs: Travis CI Tutorial*. URL: https://docs.travis-ci.com/user/tutorial (visited on 06/10/2022).

[123] *CircleCI Docs: Introduction to YAML Configurations*. URL: https://circleci.com/docs/2.0/introduction-to-yaml-configurations (visited on 06/10/2022).

[124] *Jenkins Docs: Using a Jenkinsfile*. URL: https://www.jenkins.io/doc/book/pipeline/jenkinsfile (visited on 06/10/2022).

[125] *GitLab Docs: CI/CD Pipelines*. URL: https://docs.gitlab.com/ee/ci/pipelines (visited on 06/10/2022).

[126] *Dockerhub: ansible/ansible*. URL: https://hub.docker.com/r/ansible/ansible (visited on 06/10/2022).

[127] *Ansible Docs: Validating tasks: check mode and diff mode*. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_checkmode.html (visited on 06/10/2022).

[128] *GitLab Docs: GitLab CI/CD include examples*. URL: https://docs.gitlab.com/ee/ci/yaml/includes.html (visited on 06/10/2022).

[129] *GitLab CI-Template: Terraform-Base*. URL: https://gitlab.com/gitlab-org/gitlab/blob/master/lib/gitlab/ci/templates/Terraform/Base.gitlab-ci.yml (visited on 06/10/2022).

[130] *Terraform Docs: Command: fmt*. URL: https://www.terraform.io/cli/commands/fmt (visited on 06/10/2022).

[131] *Terraform Docs: Command: validate*. URL: https://www.terraform.io/cli/commands/validate (visited on 06/10/2022).

[132] *Terraform Docs: Command: plan*. URL: https://www.terraform.io/cli/commands/plan (visited on 06/10/2022).

[133] *Terraform Docs: Command: apply*. URL: https://www.terraform.io/cli/commands/apply (visited on 06/10/2022).

[134] *GitLab Docs: GitLab-managed Terraform state*. URL: https://docs.gitlab.com/ee/user/infrastructure/iac/terraform_state.html (visited on 06/10/2022).

[135] Carmen Carrión. *Kubernetes Scheduling: Taxonomy, ongoing issues and challenges*. en. June 2022. DOI: 10.1145/3539606. URL: http://dx.doi.org/10.1145/3539606.

[136] *Kubernetes Docs: Kubernetes Components*. URL: https://kubernetes.io/docs/concepts/overview/components (visited on 06/11/2022).

[137] *Kubernetes Docs: The Kubernetes API*. URL: https://kubernetes.io/docs/concepts/overview/kubernetes-api (visited on 06/11/2022).

[138] *Kubernetes Docs: Command line tool (kubectl)*. URL: https://kubernetes.io/docs/reference/kubectl (visited on 06/11/2022).

[139] *Kubernetes Docs: ConfigMaps*. URL: https://kubernetes.io/docs/concepts/configuration/configmap (visited on 06/11/2022).

[140] *Kustomize*. URL: https://kustomize.io (visited on 06/11/2022).

[141] *Helm*. URL: https://helm.sh (visited on 06/11/2022).

[142] *Kubernetes Docs: Declarative Management of Kubernetes Objects Using Configuration Files*. URL: https://kubernetes.io/docs/tasks/manage-kubernetes-objects/declarative-config (visited on 06/11/2022).

[143] *Kubernetes Docs: Update API Objects in Place Using kubectl patch*. URL: https://kubernetes.io/docs/tasks/manage-kubernetes-objects/update-api-object-kubectl-patch (visited on 06/11/2022).

[144] *Kubernetes Docs: Declarative Management of Kubernetes Objects Using Kustomize*. URL: https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization (visited on 06/11/2022).

[145] *Helm Docs: Chart Template Guide - Getting Started*. URL: https://helm.sh/docs/chart_template_guide/getting_started (visited on 06/11/2022).

[146] *Helm Docs: Using Helm*. URL: https://helm.sh/docs/intro/using_helm (visited on 06/11/2022).

[147] *Helm Docs: Helm*. URL: https://helm.sh/docs/helm/helm (visited on 06/11/2022).

[148] *GitLab Docs: Helm charts in the Package Registry*. URL: https://docs.gitlab.com/ee/user/packages/helm_repository (visited on 06/11/2022).

[149] *GitHub: chartmuseum/helm-push*. URL: https://github.com/chartmuseum/helm-push (visited on 06/11/2022).

[150] *Ansible Galaxy: kubernetes/core*. URL: https://galaxy.ansible.com/kubernetes/core (visited on 06/11/2022).

[151] *Terraform Registry: hashicorp/kubernetes*. URL: https://registry.terraform.io/providers/hashicorp/kubernetes/latest (visited on 06/11/2022).

[152] *Terraform Registry: hashicorp/helm*. URL: https://registry.terraform.io/providers/hashicorp/helm/latest (visited on 06/11/2022).

[153] *Helm Docs: Helm Upgrade*. URL: https://helm.sh/docs/helm/helm_upgrade (visited on 06/11/2022).

[154] *Terraform Documentation*. URL: https://www.terraform.io/docs (visited on 06/17/2022).

[155] *Ansible: Use Cases*. URL: https://www.ansible.com/use-cases (visited on 06/17/2022).

[156] *Helm Docs: Helm Create*. URL: https://helm.sh/docs/helm/helm_create (visited on 06/18/2022).

[157] *Kubernetes Docs: kubectl commands*. URL: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands (visited on 06/18/2022).

[158] *GitLab Docs: Pipeline Editor*. URL: https://docs.gitlab.com/ee/ci/pipeline_editor (visited on 06/20/2022).

[159] *Terraform Docs: Command: graph*. URL: https://www.terraform.io/cli/commands/graph (visited on 06/20/2022).

[160] *Kubeval*. URL: https://www.kubeval.com (visited on 06/20/2022).

[161] *GitHub: kubernetes-sigs/kubetest2*. URL: https://github.com/kubernetes-sigs/kubetest2 (visited on 06/20/2022).

[162] *Kubernetes Docs: End-to-End Testing in Kubernetes*. URL: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-testing/e2e-tests.md (visited on 06/20/2022).

[163] *Helm Docs: Helm Lint*. URL: https://helm.sh/docs/helm/helm_lint (visited on 06/20/2022).

[164] *Helm Docs: Chart Test*. URL: https://helm.sh/docs/topics/chart_tests (visited on 06/20/2022).