HTWK DIT Fakultät Digitale Transformation

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Engineering

im Studiengang Telekommunikationsinformatik
der Fakultät Digitale Transformation
der Hochschule für Technik, Wirtschaft und Kultur Leipzig

Analyse und Vergleich des Quellcode-basierten Ressourcenmanagements und des automatischen Deployments von Webapplikationen auf Cloud-Plattformen

– am Beispiel von Microsoft Azure und der Open Telekom Cloud –

vorgelegt von:	Peter Prumbach
Geburtsort und -datum:	Düren, den 21. Juni 2001
Abgabe:	Hamburg, den 24. Februar 2023

Erstgutachter: Prof. Dr. rer. nat. Andreas Thor, HTWK Leipzig
Zweitgutachter: Dipl.-Inf. Andreas Kortüm, T-Systems MMS

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die an der Hochschule für Technik, Wirtschaft und Kultur Leipzig, konkret an der Fakultät Digitale Transformation, eingereichte Arbeit zum Thema Analyse und Vergleich des Quellcode-basierten Ressourcenmanagements und des automatischen Deployments von Webapplikationen auf Cloud-Plattformen selbstständig, ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe.

Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kenntlich gemacht. Die Abbildungen in dieser Arbeit wurden von mir selbst erstellt oder mit einem entsprechenden Hinweis auf die Quelle versehen.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Hamburg, den 24. Februar 2023 Peter Prumbach Digital unterschrieben von Peter Prumbach Datum: 2023.02.24 13:19:58 +01'00'

Ort, Datum Unterschrift Studierender

Vorwort

Vor langer, langer Zeit, in einem weit, weit entfernten Rechenzentrum, wurde die Infrastruktur durch eine Gruppe alter mächtiger Wesen, bekannt als die „Systemadministratoren“ manuell eingerichtet. Jeder Server, jede Datenbank, jeder Load-Balancer und jedes Stückchen Netzwerkkonfiguration wurde von Hand erstellt und verwaltet. Es handelte sich um eine dunkle Zeit und es galt eine ständige und furchterregende Angst vor Ausfallzeiten, vor versehentlicher Fehlkonfiguration und der Angst davor, was passieren würde, wenn die Systemadministratoren auf die dunkle Seite wechselten (d.h. Urlaub machten). Die gute Nachricht ist jedoch, dass es dank der DevOps-Bewegung nun einen besseren Weg gibt, diese Dinge zu erledigen [1].

Um genau diese „Dinge“ soll es im Verlauf dieser Bachelorarbeit gehen.

Kurzfassung

In dieser Thesis werden unterschiedliche Wege erläutert, Webanwendungen mit einem Cloud-agnostischen Ansatz bereitzustellen. Ein Cloud-agnostischer Ansatz zielt auf die Unabhängigkeit von einem bestimmten Cloud Service Provider (CSP) und dessen Technologien ab. Um dies zu ermöglichen, werden verschiedene Tools unter anderem hinsichtlich ihrer unterstützten Sprachen und Technologien, ihrer Modularität, ihres State und Secret Managements, ihres Bekanntheitsgrades und des Community Supports verglichen. Die Einführung erfolgt entlang der theoretischen Grundlagen, der Erläuterungen und Vorteile des Konzepts der Infrastructure-as-Code (IaC), anhand der Grundlagen zur imperativen und deklarativen Programmierung und mittels der Unterscheidung zwischen Domain-Specific Languages und General-Purpose Languages. In den folgenden Kapiteln folgt, bezogen auf die in dieser Thesis behandelten Beispiele Microsoft Azure (Azure) und Open Telekom Cloud (OTC), ein Vergleich der unterschiedlichen Möglichkeiten, Webanwendungen auf diesen Plattformen bereitzustellen. Dieser Ansatz soll anschließend durch eine Automatisierung mittels eines ausgewählten Frameworks als Prototyp anhand einer bestehenden Webanwendung implementiert werden. Zur Implementierung werden vorher die bekanntesten Frameworks auf Grundlage dieser Problemstellung verglichen und das passendste ausgewählt. Als Abschluss der Thesis folgt eine Zusammenfassung, in welcher die gelernten Kenntnisse und Erfahrungen im Umgang mit der Bereitstellung von Infrastruktur für Webanwendungen mittels IaC in einem Cloud-agnostischen Einsatz dargelegt werden.

Keywords: IaC, Cloud-agnostisch, Cloud-Infrastruktur, Deklaratives Ressourcenmanagement

Inhaltsverzeichnis

Abkürzungsverzeichnis

AWS Amazon Web Services

Azure Microsoft Azure

CNCF Cloud Native Computing Foundation

CSP Cloud Service Provider

DORA Devops Research & Asessment

DSL Domain-Specific Language

GPL General-Purpose Language

HCL HashiCorp Configuration Language

IaC Infrastructure-as-Code

KM Konfigurationsmanagement

OAM Open Application Model

OTC Open Telekom Cloud

SaaS Software-as-a-Service

VCS Version Control System

VM Virtuelle Maschine

VPC Virtual Private Cloud

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

1.1 Problemstellung

In dem folgenden Kapitel soll das Problem erläutert werden, eine vorhandene Webanwendung sowie die dazugehörige Infrastruktur innerhalb einer Cloud-Umgebung bereitzustellen. Es soll zunächst dargestellt werden, welche Komplexität die Bereitstellung einer Infrastruktur in umfangreichen Umgebungen mit sich bringt. Eine wichtige Rolle spielt hierbei das Multi-Cloud-Deployment, sprich das Replizieren von Konzepten und Technologien einer Cloud-Umgebung auf die eines anderen Anbieters.

Die derzeitigen Cloud-Technologien leiden unter einem Mangel an Standardisierung, da verschiedene Anbieter ähnliche Ressourcen auf unterschiedliche Weise anbieten, welche ebenfalls abweichende Funktionalitäten und entsprechende Vor- und Nachteile vorweisen. Daher ist die Migration einer Anwendung bzw. der Infrastruktur, welche zum Betrieb dieser notwendig ist, von einer Cloud in eine andere ein kostspieliger und fehleranfälliger Prozess. Infolgedessen neigen Cloud-Nutzer dazu, sich an die von ihnen genutzte Cloud-Plattform zu binden, da es für sie nicht trivial umsetzbar ist, ihre Anwendung über verschiedene Cloud-Plattformen hinweg zu migrieren oder aufzuteilen [2].

In der gesamten Betrachtung und Analyse der Bachelorarbeit bezieht sich der Autor auf die Cloud-Lösungen von Microsoft und der Deutschen Telekom AG. Da eine Allgemeingültigkeit bei der Variation an bestehenden Cloud-Lösungen nicht abgebildet werden kann, bezieht sich die Analyse, die Konzeption und die Implementierung eines Prototyps auf die Beispiele Azure und OTC.

1.1.1 Aufkommende Probleme bei der Infrastrukturbereitstellung

Die Modellierung einer Infrastruktur zählt zum Lebenszyklus eines Softwareprodukts und beinhaltet die Definition und Konfiguration der Infrastruktur-

Komponenten, die für diese Software bereitgestellt werden müssen. Innerhalb dieser Modellierung wird sowohl der Anbieter der Cloud-Plattform, als auch beispielhaft die Anzahl an benötigten VMs bedacht. Die benötigte Infrastruktur muss jedoch nicht nur bereitgestellt, sondern auch orchestriert und konfiguriert werden. Hierzu zählen unter anderem die Installation von benötigten Paketen oder die Vernetzung verschiedener Systeme untereinander [3]. Die Infrastruktur ist somit zwingend notwendig für den Betrieb einer Anwendung.

Werden diese Prozesse zunehmend manuell durchgeführt, so kann dies schnell Nachteile mit sich ziehen (siehe Tabelle 1.1). Die Bereitstellung wird durch den manuellen Aufwand kostenintensiv und fehleranfällig. Soll im Fall eines Ausfalls oder eines Datenverlusts innerhalb der Cloud-Infrastruktur ein gleicher Stand der Systeme zurückgerollt werden, so ist dies anhand vorher ausgeführter manueller Kommandozeilenbefehle oder Eingaben in einer Weboberfläche nicht vollständig reproduzierbar.

Heterogenität von Cloud-Ressourcen	Cloud-Ressourcen haben als solche keine definierten Standards. Die Definition einer virtuellen Maschine in Cloud A unterscheidet sich von einem Vorgehen in Cloud B.
Fehlende Kompatibilität	Cloud-Plattformen basieren auf unterschiedlichen Standards und dementsprechend anderen Schnittstellen. Während viele Anbieter den Standard OpenStack implementieren, so setzten andere auf eigene Managementmethoden und Entwicklungsschnittstellen.
Fehlende Integration	Durch fehlende Schnittstellen fällt der Austausch zwischen verschiedenen Umgebungen untereinander komplex aus.
Transfer	Ein Umzug von einer in die andere Cloud-Umgebung birgt eine erhöhte Komplexität. Möglicherweise sind nicht alle Ressourcentypen gleichermaßen verfügbar oder ansteuerbar.
Manueller Aufwand	Die Durchführung regelmäßiger Deployments und Wartungen einer Infrastruktur erfordern eine erhöhte Automatisierung oder hohe personelle Ressourcen.

Tabelle 1.1: Probleme, die bei der Bereitstellung von Infrastrukturen in der Cloud auftreten können [4]

1.1.2 Nutzung von Multi-Cloud Umgebungen

Bereits 2002 brachte Amazon dessen Produkt Amazon Web Services (AWS) mit dem Gedanken, neben dem Hauptgeschäft weiteren Umsatz generieren zu können, auf den Markt. Heute ist dieses Geschäftsfeld mit ca. 33 % Marktanteil hochprofitabel und eins der umsatzstärksten innerhalb des gesamten Konzerns [5].

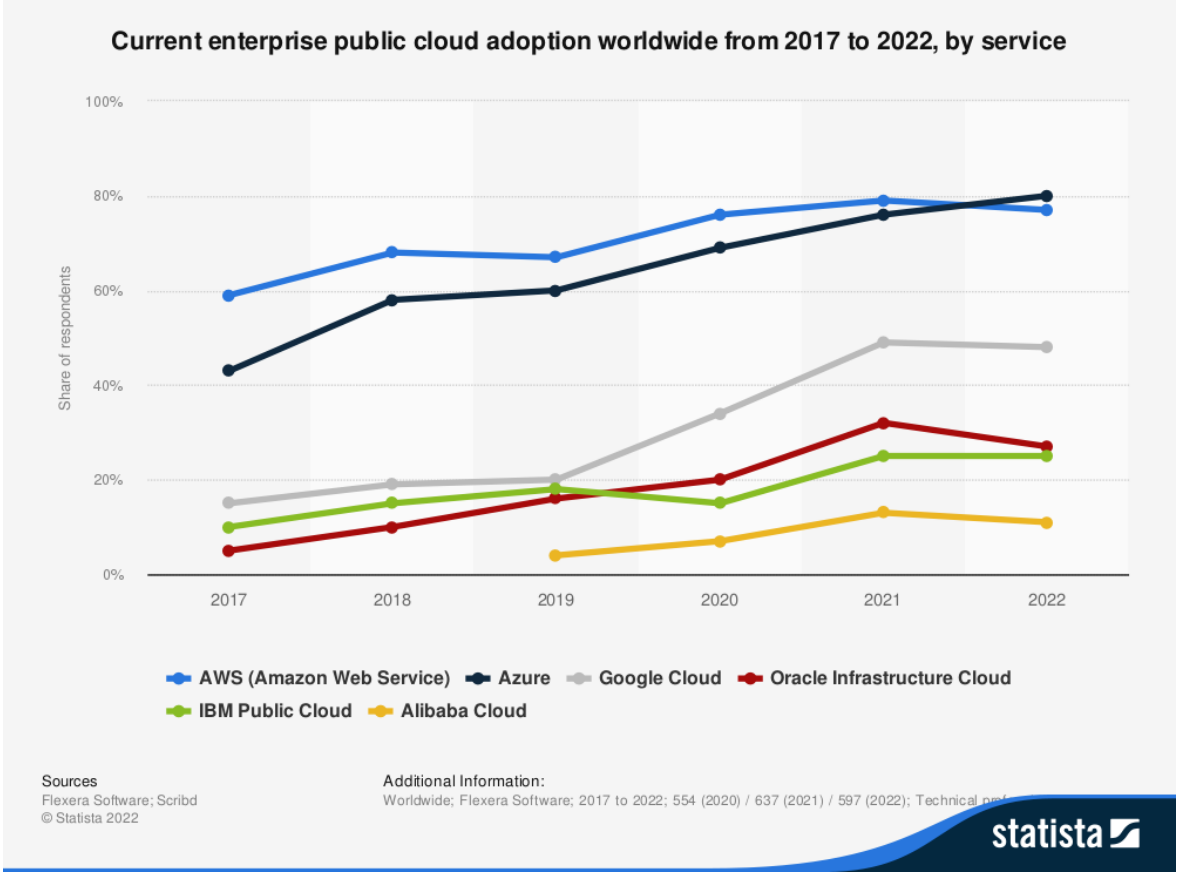

Abbildung 1.1: Aktuelle Verteilung der Nutzung von Public-Clouds in Unternehmen von 2017 bis 2022, aufgeteilt nach Anbieter [6]

Seitdem skalierte der Markt um die Produktportfolios für Cloud-Lösungen namentlicher Software- und Hardwareanbieter im letzten Jahrzehnt um ein Vielfaches. Immer mehr Anbieter stellen eigene Public-Cloud-Lösungen und die dazugehörige Infrastruktur über teils eigene Rechenzentren bereit (siehe Abbildung 1.1).

Daher ist der Ansatz, mehrere Cloud-Lösungen verschiedener Anbieter parallel einzusetzen, ein sehr beliebter, um so Vorteile wie die Erweiterung der Skalierbarkeit, die erhöhte Verfügbarkeit oder eine mögliche Kostenersparnis zu erlangen.

In einer von Microsoft durchgeführten und veröffentlichten Studie aus Januar 2022 gaben 50 % aller Befragten an, dass Ihr Unternehmen sich für die Verwendung von

Hybrid- oder Multi-Cloud-Umgebungen entschieden hat, um die Gesamtheit an Vorzügen und technische Möglichkeiten einzelner Lösungen als Ganzes ausschöpfen zu können. 49 % gaben an, mit diesem Ansatz das Risiko minimieren zu können, für den Fall eines Ausfalls oder Wegfalls einer Lösung eine schnelle Alternative bieten zu können. Der Ansatz wurde ebenfalls von 43 % verteidigt, die sich aufgrund von verschiedenen Datenschutzregularien und Anforderungen in unterschiedlichen Regionen zu dieser Nutzung überzeugen ließen [7].

1.1.3 Probleme in Multi-Cloud Umgebungen

Ein Deployment einer Anwendung auf mehrere Cloud-Lösungen birgt aber auch Probleme und Risiken. Bevor dieses Deployment stattfinden kann, muss zunächst die Provisionierung der Infrastruktur abgeschlossen werden. An dieser Stelle erhöht die Komplexität eines Parallelbetriebs den Aufwand in der Bereitstellung. Dieser Mehraufwand kann zwar mit technischen Hilfsmitteln und Methoden der IaC gemindert, jedoch nicht vollständig vernachlässigt werden [8].

Ein weiteres Problem besteht auch in dem Prozess des Deployments selbst, da hierfür kein Standard definiert ist und dieser je nach Lösung abweichen kann. Da sich die verfügbaren Ressourcen der verschiedenen Cloud-Lösungen sowie deren Prozesse für das Deployment teilweise in maßgebenden Punkten unterscheiden, wird in diesem Fall neben einer automatischen Prozesskette ebenfalls eine Abstraktion eingeführt.

1.2 Zielstellung

Ziel der vorliegenden Arbeit soll es sein, eine Lösung zu evaluieren, um eine bereits bestehende Webanwendung auf den Cloud-Lösungen Azure und OTC bereitzustellen. Innerhalb der Analyse soll erarbeitet werden, inwiefern eine Automatisierung der Schritte der Provisionierung und des Deployments für unterschiedliche CSPs als allgemeiner Prozess darstellbar ist. Dazu gilt es, die technischen Möglichkeiten der beiden Plattformen zu erläutern und einen Prozess für eine Automatisierung innerhalb der Schritte der Provisionierung und des Deployments mithilfe technischer Mittel zu etablieren. Die Erkenntnisse der Arbeit sollen mithilfe der Bereitstellung bzw. Entwicklung eines Prototyps abgerundet und demonstriert werden.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich zunächst in einen theoretischen Teil, indem die Grundlagen der IaC sowie angrenzenden Technologien erläutert und vorgestellt werden sollen.

Innerhalb der Analyse sollen die Möglichkeiten der Cloud-Lösungen Azure und OTC geprüft werden, eine Webanwendung bereitstellen zu können. Es sollen technische Tools vorgestellt werden, um eine Prozesskette der Schritte Provisionierung und Deployment abbilden zu können.

Nach Abschluss der Konzeption sollen die vorgestellten Hilfsmittel genutzt werden, um einen Prototyp implementieren zu können.

2 Theoretische Grundlagen

2.1 Prozess der Infrastrukturbereitstellung

Die Infrastrukturbereitstellung beschreibt den Prozess des Managements und der Konfiguration von Infrastruktur-Ressourcen, wie beispielsweise einer virtuellen Maschine, Speicher oder eines virtuellen Netzwerkes, überwiegend in Cloud-Umgebungen jedoch ebenfalls in Verbindung mit physischer Hardware. Diese Ressourcen werden im Unternehmenskontext meist zum Betrieb eigens entwickelter Anwendungen bzw. dem Vernetzen einzelner Komponenten untereinander benötigt, um diese für Kunden verfügbar zu machen.

Der Gesamtprozess der IaC besteht aus dem Schreiben von Quellcode für das Definieren, Ausrollen, Ändern und Löschen von Infrastruktur [1]. Aufgrund der verschiedenen Möglichkeiten existieren fünf größere Kategorien, in denen sich die einzelnen Prozessschritte sowie die jeweiligen Tools gliedern lassen. Diese lauten Provisionierung, Orchestrierung, Konfigurationsmanagement (KM), Server-Templating bzw. Deployment und die Ad-hoc-Skripte [1]. Ad-hoc-Skripte sollen im Nachgang jedoch nicht weiter betrachtet werden, da sich diese eher für kleine, einmalige Aufgaben und nicht für wiederkehrende und komplexe Aufgaben wie die Verwaltung der Infrastruktur eignen [1].

2.1.1 Provisionierung

Der Begriff der Provisionierung wird allgemein dazu verwendet, den Prozess zur Bereitstellung und Erstellung von Ressourcen zu beschreiben [1]. Zu der Provisionierung gehören in den meisten Fällen zusätzlich die Installation von, für den Betrieb notwendigen Software-Bibliotheken oder vorausgesetzten Diensten [9]. Der Prozess hinter einer Provisionierung wird in dem Lebenszyklus einer Anwendung meist nur einmal zu Beginn durchgeführt und stellt die Voraussetzungen für ein Deployment bereit. Bereits provisionierte Systeme können mithilfe des KMs rekonfiguriert und

verwaltet werden.

Die Schritte innerhalb der Provisionierung werden üblicherweise über ein Orchestrierungs-Tool der Gruppe IaC, wie beispielsweise Terraform[1], durchgeführt.

2.1.2 Orchestrierung

Der Schritt der Orchestrierung dient dazu, eine Vielzahl von zur Verfügung gestellten IT-Systemen bzw. Ressourcen zu koordinieren und das Zusammenwirken dieser zu arrangieren [1], sodass dadurch Prozesse und Workflows in der IT optimiert werden können [10]. Meistens geschieht die Umsetzung als ein erweiterter Schritt innerhalb der Provisionierung, da hierbei die gleichen Technologien zum Einsatz kommen.

2.1.3 Konfigurationsmanagement

Das Konzept des KMs erweitert die Prinzipien der Provisionierung. Während die Provisionierung oft bedeutet, eine Initialkonfiguration bereitzustellen, impliziert das KM die kontinuierlichen Erweiterungen der Konfiguration. Es kommen hierbei meist abweichende Tools als bei der Provisionierung, wie beispielsweise Chef[2], Puppet[3] oder Ansible[4] zur Anwendung [1]. Eine stetige Konfigurationsänderung ist ein besonders wichtiger Schritt bei dem Betrieb einer Plattform, um so stets aktuelle Updates, die Verbesserung der Performanz und die Stabilität der Systeme gewährleisten zu können. Konfigurationsänderungen werden ebenfalls durch Updates vorgenommen, um die weitere Sicherheit der Anwendung garantieren zu können. Schritte des KMs können auch notwendig werden, wenn die Anforderungen an eine Plattform angepasst werden müssten.

2.1.4 Deployment

Während die grundlegende Infrastruktur inklusive der Netzwerkkonfiguration in den vorhergegangenen Schritten bereitgestellt wurde, erfolgt abschließend das Deployment einer Anwendung auf die Cloud-Infrastruktur. Das Deployment stellt

[1] https://terraform.io
[2] https://chef.io
[3] https://puppet.com
[4] https://ansible.com

entweder eine Bereitstellung eines Updates einer bereits vorhandenen Anwendung oder die Initialbereitstellung einer Anwendung auf bereits provisionierte und konfigurierte Systeme dar. Hierbei zu beachten ist, dass während dieses Schrittes der Prozesskette keine Konfiguration oder die Installation von Abhängigkeiten durchgeführt wird.

2.2 Einführung Infrastructure-as-Code

2.2.1 Warum Infrastructure-as-Code?

Den Problemen der manuellen Bereitstellung von Infrastruktur in der Cloud versucht der Ansatz der *IaC* zu entgegnen, indem dieser auf der Basis von Automatisierung und Praktiken der Softwareentwicklung aufbaut [11]. Der Ansatz der IaC siedelt sich im Bereich des Development & Operations (DevOps) an. Ziel ist es, die benötigte Infrastruktur und deren Konfiguration möglichst als High-Level-Abstraktion im Quelltext definieren zu können. Mithilfe dieses Quelltextes sollen die Konsistenz und wiederholbare Routinen für die Provisionierung hervorgehoben und ein Ändern der IT-Systeme und deren Konfigurationen ermöglicht werden [11].

Eine Änderung der Infrastruktur beinhaltet schließlich nur noch die Änderung des Quelltextes, wodurch die Automatisierung und Tests angestoßen werden und die Änderungen auf die Ressourcen einer Cloud-Plattform angewendet werden [11].

Der Quelltext, welcher die Infrastruktur definiert, wird überwiegend innerhalb eines Version Control System (VCS), wie beispielsweise Git, verwaltet.

Wie in den meisten Konfigurations-Tools ist auch bei den überwiegenden Frameworks sowohl ein *push*-Ansatz (Managementserver (Master) transferiert Konfiguration an Knoten (Clients)) als auch ein *pull*-Ansatz (Clients laden Konfiguration von einem Master) realisierbar [1].

2.2.2 Vorteile der Infrastructure-as-Code

Durch die Definition der Infrastruktur als Quellcode können Unternehmen in vielerlei Hinsicht Vorteile erzielen, mit denen eine starke Verbesserung im Auslieferungsprozess des entwickelten Softwareprodukts erreicht werden kann (siehe Tabelle 1.1) [1].

Durch die Verwendung von Quellcode zur Verwaltung der Infrastruktur kann zudem erreicht werden, dass das Wissen über die Konfiguration nicht nur bei einem kleinen Personenkreis beherbergt ist, sondern dieses von allen Entwicklern und Teammitgliedern eingesehen und geprüft werden kann. Dies vereinfacht auch den Schritt des Ausrollens[5] der Infrastruktur, wenn dieser entsprechend über Pipelines automatisch gestaltet wird, indem dieses durch einen beliebigen Entwickler jederzeit durchgeführt werden kann [1].

Zudem verbessert eine Automatisierung mit Quellcode auch die Performanz und die Dauer, die es benötigt, um ein Ausrollen durchzuführen. Neben der Dauer verringert sich bei einem automatischen Ausrollen auch die Wahrscheinlichkeit an Fehlern, die während des Prozesses bei manuellem Aufwand auftreten können, sodass der Prozess an Konsistenz gewinnt [1].

Zusätzlich agiert die Umsetzung der Konfiguration der Infrastruktur im Quellcode als Dokumentation und ist für jeden dazu berechtigten Mitarbeiter einsehbar und verständlich festgehalten [1].

Damit jedoch bei der Konfiguration keine Anomalien auftreten können, bietet es sich an, den Quellcode der Infrastruktur in einem Tool für die Versionsverwaltung abzulegen. Das Debugging von Problemen kann so direkt auf hinzugefügte Änderungen eingegrenzt und Probleme so schneller behoben werden. Im schlimmsten Fall kann die Konfiguration ebenfalls auf einen vorherigen Stand zurückgesetzt werden.

Mithilfe der Verwendung von Automatisierungsplattformen, auch CI/CD[6] genannt, kann der geschriebene Quellcode und dessen Auswirkungen in Cloud-Umgebungen bei jeder Änderung direkt mit zusätzlichen Tools validiert und getestet werden [1].

Die Vorzüge der IaC nutzt man besonders dann aus, wenn bereits definierte Infrastruktur an anderer Stelle wiederverwendet werden soll. Alle populären und bekannten Orchestrierungs-Tools setzen hierfür eine Methode zur Verbesserung der Modularität und der Wiederverwendbarkeit ein.

Durch die mit der IaC einhergehenden Governance-, Compliance- und Sicherheitskontrollen kann jederzeit nachvollzogen werden, welcher Entwickler Änderungen an der Konfiguration durchgeführt hat und wie diese sich ausgewirkt haben [11].

[5] Das Ausrollen beschreibt das Ausführen der aktuellen Konfigurationsdateien, sodass die Einstellungen in die Cloud-Umgebung übernommen werden.
[6] Continous Integration/Continous Delivery

Aus betriebswirtschaftlicher Sicht wirkt sich der Einsatz von IaC ebenfalls sehr positiv aus. Benötigte Ressourcen für den Betrieb einer Anwendung können so besonders schnell, effizient und mit einem wesentlich geringerem manuellen Aufwand genau dann bereitgestellt werden, wenn diese benötigt werden [11].

Die Verwendung des Quellcode-basierten Ressourcenmanagement zeigt zudem positive Auswirkungen auf die Anwenderzufriedenheit. Da es sich bei dem Ausrollen und der Konfiguration der Infrastruktur meistens um eine Aufgabe ohne Einsatz von Kreativität, ohne Herausforderungen und ohne Anerkennung handelt, ist es für den Anwender schöner, diese Tätigkeit über eine Automatisierung zu lösen. Somit können sich die Entwickler auf das Wesentliche, also das Schreiben von Quellcode für die eigentliche Anwendung, konzentrieren [1].

2.3 Deklarativer und imperativer Ansatz

2.3.1 Deklarativer Ansatz

Viele Frameworks der IaC nutzen einen deklarativen Lösungsansatz. Hierbei wird die gewünschte Konfiguration der Infrastruktur innerhalb des Quelltextes definiert. Das jeweilige Tool kümmert sich im Hintergrund darum, dass dieser Konfigurationsstand schließlich erreicht wird. Der Quelltext selbst enthält in diesem Ansatz keinerlei Logik, wie dieser Konfigurationsstand erreicht wird. Das Tool vergleicht auch die einzelnen Attribute miteinander und bringt die Infrastruktur bei einer Abweichung auf den neusten Stand, welcher laut Quelltext vorherrschen soll [11].

Das deklarative Vorgehen **trennt** die Anliegen der Frage *was* umgesetzt werden soll von der Frage, *wie* dies geschieht. Das Ergebnis liefert einen sauberen Quelltext, welcher direkt und verständlicher ist. Zudem wird ein deklaratives Vorgehen eher als Konfiguration anstatt einer wirklichen Programmierung gesehen [11].

```
virtual_machine:
    name: my_application_server
    source_image: 'base_linux'
    cpu: 2
    ram: 2GB
    network: private_network_segment
    provision:
        provisioner: servermaker
        role: tomcat_server
```

Quellcode 2.1: Beispiel eines deklarativen Quellcodes [11]

2.3.2 Imperativer Ansatz

Im Vergleich zum prozeduralen bzw. imperativen Ansatz erreicht der deklarative Ansatz bei gleicher Eingabe immer das gleiche Ergebnis. Möchten man seine Infrastruktur jedoch dynamisch anhand von Abhängigkeiten oder Umständen definieren, so bietet sich ein imperativer oder auch programmierbarer Ansatz an. Mit diesem Ansatz können anders als bei dem Deklarativen Logiken, wie beispielsweise Bedingungen oder Schleifen, implementiert werden [11].

Das imperative Vorgehen **verknüpft** die Anliegen der Frage *was* umgesetzt werden soll mit der Frage, *wie* dies geschieht [11].

```
import vm

def createVM(name, cpu, ram):
    vm = new VM(name=this.name, cpu=this.cpu, ram=this.ram)
    return vm

vm1 = createVM("vm1", 2, 2)
vm2 = createVM("vm2", 2, 8)
vms = [vm1, vm2]

for vm in vm:
    deploy(vm)
```

Quellcode 2.2: Beispiel eines imperativen Quellcodes

2.3.3 Vergleich deklarativ und imperativ

Ein deklarativer Ansatz ist besonders sinnvoll, wenn ein gewünschter Konfigurationsstand eines Systems definiert und erreicht werden soll. Dies ist gerade effektiv, wenn keine Sonderfälle abgedeckt werden müssen. Der Ansatz bietet eine hohe Konsistenz und Wiederverwendbarkeit. Mithilfe von Konfigurationsparametern können auch limitierte Variationen erreicht werden.

Der imperative Ansatz hingegen bietet dem Nutzer einen wiederverwendbaren, teilbaren Quelltext, welcher einen anderen Ausgang abhängig von der Situation liefern kann.

Auch ein deklarativer Ansatz kann diese komplexeren Variationen unterstützen, indem in den Frameworks eine eigene Logik implementiert wird. Ab einer gewissen Komplexität wird jedoch auch diese Logik zu umfangreich, um diese in YAML, JSON, XML oder anderen deklarativen Sprachen abzubilden [11].

Der Vorteil des imperativen Ansatzes liegt hier eindeutig in der Konstruktion von Bibliotheken oder Abstraktionsebenen [11].

Im späteren Verlauf der Arbeit sollen die Eigenschaften aktueller Frameworks verglichen werden, die sowohl imperativ als auch deklarativ arbeiten.

2.4 Domain-Specific Languages

Viele deklarative Tools verwenden eine eigene Domain-Specific Language (DSL). Eine DSL definiert sich als schmale, limitierte Konfigurationssprache, welche Ihren Fokus auf einen bestimmten Aspekt eines Softwaresystems lenkt [12]. Dies kann den Entwicklern die Arbeit vereinfachen, indem in einer solchen DSL Domain-spezifische Muster und Ideen bereits aufgegriffen werden.

Beispielsweise verwenden Ansible, Chef und Puppet jeweils eine eigene DSL, um Server zu konfigurieren. Diese jeweiligen DSLs bringen die Grundkonzepte für Pakete, Dateien, Services und Nutzer bereits mit [11].

Einige Orchestrierungs-Tools verwenden ebenfalls eigene DSLs. Das populärste Framework Terraform verwendet für die Konfiguration die HashiCorp Configuration Language (HCL), bei welcher die Ressourcen mithilfe einer deklarativen Definition provisioniert werden können [13]. Es wird so erreicht, dass näher an der realen Infrastruktur gearbeitet werden kann.

Die meisten DSLs, die bei Tools für die Infrastruktur verwendet werden, wurden als Erweiterung bereits bestehender Markup Languages wie beispielsweise *YAML* (Ansible, Kubernetes) entwickelt.

```
resource ”azurerm_network_interface” ”example” {
    name                = ”example-nic”
    location            = azurerm_resource_group.example.location
    resource_group_name = azurerm_resource_group.example.name

    ip_configuration {
        name                          = ”internal”
        subnet_id                     = azurerm_subnet.example.id
        private_ip_address_allocation = ”Dynamic”
    }
}
```

Quellcode 2.3: Beispiel der HashiCorp Configuration Language (HCL)

2.4.1 Vorteile der General-Purpose Languages über Domain-Specific Languages

Von der Gegenseite betrachtet bieten General-Purpose Language (GPL) ebenfalls einige Vorteile gegenüber DSLs.

Dadurch, dass GPLs in vielen Entwicklungsprojekten unabhängig vom Domänen-Hintergrund eingesetzt werden, kann es möglich sein, dass man sich zur Verwaltung der Infrastruktur mittels Quellcode keine neue Programmiersprache aneignen muss, sondern bereits bekannte Kenntnisse aus anderen Softwareprojekten einsetzen kann [1].

Ein weiterer Vorteil der breit eingesetzten GPLs ist die breite Unterstützung und die große Community sowie die für die Programmiersprache bereitgestellten Hilfsmittel. GPLs finden meist ein Vielfaches an Plugins, Bibliotheken oder Test-Software als eine DSL.

Durch die Verwendung einer GPL, wie beispielsweise Python, können beliebig komplexe Programmieraufgaben gelöst werden als mit einer limitierten DSL. DSLs bieten meist keine bzw. limitierte Möglichkeiten der Verwendung einer Kontrollstruktur, eines automatisierten Testens, einer Abstraktion durch Klassen oder einer Integration in weitere Tools [1].

2.4.2 Vorteile der Domain-Specific Languages über General-Purpose Languages

Durch den Grundgedanken einer DSL beschäftigt sich diese nur mit einem bestimmten Anwendungszweck und versucht nicht ein breites Feld der Entwicklung abzubilden. DSLs sind dadurch meistens kompakter und einfacher zu erlernen als GPLs [1].

Da DSLs meistens auf einen bestimmten Anwendungszweck limitiert sind, vereinfacht dies den Überblick der in der Sprache möglichen Ausdrücke, Keywords und Syntax und somit die allgemeine Verständlichkeit des Quellcodes [1].

Durch den limitierten Funktionsumfang und das Einsatzgebiet bestimmter DSLs ist in den meisten Fällen ebenfalls eine vereinfachte und einheitliche Struktur des Quellcodes vorgesehen. Ein Quellcode, geschrieben in einer GPL, ist schwieriger zu interpretieren, da jeder Entwickler ein Problem auf eine unterschiedliche Weise lösen kann.

2.5 Abstraktionsebene

Das Hauptziel der Implementierung einer Abstraktionsebene ist die Bereitstellung einer vereinfachten Sicht („High-Level“) auf Ressourcen einer tieferen Ebene („Low-Level“). So eine Ebene könnte aus wiederverwendbaren Komponenten bestehen, aus welchen Ressourcen der Infrastruktur gebildet werden.

2.5.1 Low-Level Quelltext

Die meisten Tools zur Infrastrukturbereitstellung verwenden eine Low-Level-Sprache. Dies bedeutet, dass genau die Ressourcen beschrieben werden, welche die Infrastruktur-Plattform bereitstellt. Diese einzelnen Ressourcen können auch untereinander Attribute austauschen und miteinander verknüpft werden. Das verwendete Framework führt also ein direktes Mapping der beschriebenen Ressourcen auf die dazugehörige API eines Cloud-Anbieters durch [11].

```
virtual_machine:
  name: "vm"
  cpu: 2
  ram: 2

network_interface:
  type: "private"
  ip_type: "static"
  ip: 10.0.0.5

firewall_rule:
  name: "Allow SSH"
  type: "allow"
  protocol: "tcp"
  destination: "*"
  source: "*"
  port: 22
```

Quellcode 2.4: Beispiel eines Low-Level Quelltextes

2.5.2 High-Level Quelltext

Im Gegensatz zum Low-Level werden im High-Level Entitäten beschrieben, welche so direkt nicht bei dem Cloud-Anbieter verfügbar sind [11]. Ziel ist es, die Definition allgemeiner zu halten. Das darunterliegende Framework ist dann selbständig für die Evaluierung der Werte und die Umsetzung verantwortlich. High-Level-Definitionen referenzieren sogenannte Module oder Vorlagen, in denen das Low-

Level definiert ist. Durch diese Verwendung kann eine Wiederverwendbarkeit gebildet werden. High-Level-Sprachen können verwendet werden, um eigene Abstraktionsebenen zu bilden oder Bibliotheken zu verwenden [11].

```
internal_server:
  cpu: 2
  ram: 2
  ip: "10.0.0.5"
```

Quellcode 2.5: Beispiel eines Low-Level Quelltextes

2.5.3 Cloud-agnostisch

Für die Zusicherung an Redundanz einer Anwendung kann ein Cloud-agnostischer oder auch Cloud-neutraler Ansatz gewählt werden. Hierbei entkoppelt man sich vollkommen von einem einzigen Cloud-Anbieter und verteilt die Anwendung über mehrere Lösungen [11].

Es bietet sich hier an, neben dem Verteilen der Anwendung in Container auf unterschiedliche Cloud-Lösungen, auch die Einarbeitung einer Abstraktionsebene, welche die Kommunikation und das Verhalten der CSPs zusammenfasst und steuert. Hierdurch erlangt man unter anderem Vorteile hinsichtlich der Flexibilität, der Zuverlässigkeit und die Vermeidung eines Vendor Lock-in, also die Abhängigkeit von Ressourcen eines Anbieters [11].

Durch eine solche Abstraktionsebene steigen die Komplexität und die Kosten zur Erreichung dieser ungemein. Man kann mit diesem Vorgehen auch einem Vendor Lock-out zum Opfer fallen, wenn ein Cloud-Anbieter eine praktische und innovative neue Möglichkeit bereitstellt, sich jedoch gegen eine Nutzung dieser entscheidet, weil diese Funktionalität auf anderen Plattformen noch nicht verfügbar ist.

2.6 Stand der Technik

Der Bereich rund um die Technologie IaC ist seit dem Jahr 2000 stetig durch neue Möglichkeiten erweitert worden. Es existieren bereits diverse Softwarelösungen für die Umsetzung einzelner Prozessschritte in der Infrastrukturbereitstellung, die nachfolgend vorgestellt werden sollen:

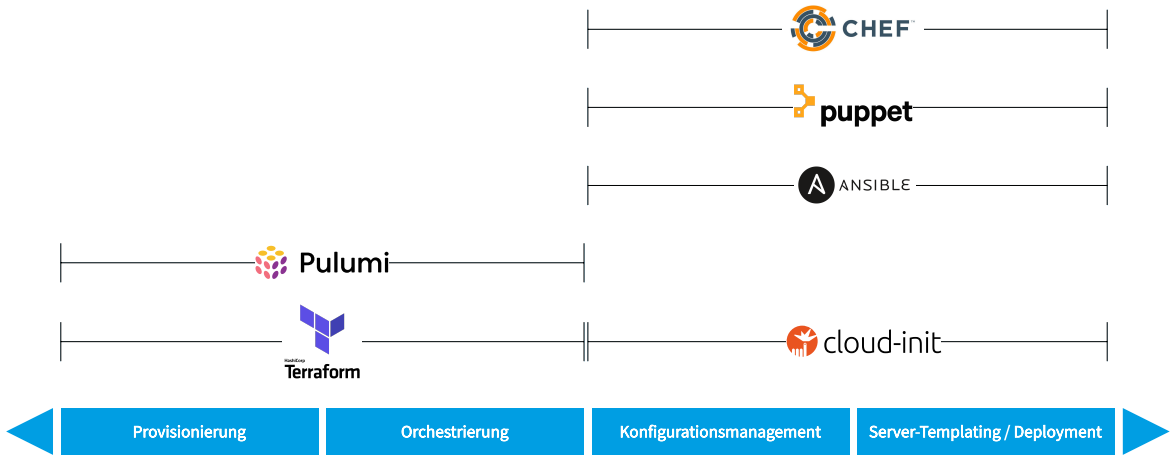

Abbildung 2.1: Das Spektrum der heute verfügbaren führenden IaC-Tools

2.6.1 Terraform

Terraform ist ein von HashiCorp entwickeltes Open-Source-Tool, mit dem die IaC mithilfe einer einfachen, deklarativen Programmiersprache, der HCL, definiert und ausgerollt werden kann [1]. Terraform unterstützt eine Vielzahl an Public-Cloud-Providern wie beispielsweise AWS, Google Cloud oder Azure, Private-Clouds und andere Virtualisierungsplattformen. Es gilt als Orchestrierungs-Tool, da mit Terraform nahezu alle möglichen Ressourcen eines Anbieters wie beispielsweise Compute-Instanzen, Speicher oder Netzwerke erstellt und verwaltet werden können [14].

2.6.2 Pulumi

Pulumi wurde 2017 veröffentlicht und ist ein universell einsetzbares Tool zur Verwaltung von IaC, welches es ermöglicht, bekannte GPLs, wie bspw. JavaScript, C#, Python oder Go für die Definition und die Verwaltung der Infrastruktur zu verwenden [15]. Pulumi ist ebenfalls kostenlos, quelloffen und kann ebenfalls mit vielen CSPs zusammenarbeiten. Es wird aktuell seltener eingesetzt als Terraform, gewinnt jedoch stetig an Popularität (siehe Tabelle 3.2)

2.6.3 Ansible

Ansible ist ein von RedHat entwickeltes Tool und zählt zur Gruppe des KMs, mit welchem die Konfiguration von Infrastruktur automatisiert werden kann. Das Tool arbeitet mit einer DSL basierend auf YAML-Syntax und prozeduraler Beschreibung

zur Verwaltung der Konfiguration, indem eine Schritt-für-Schritt-Anleitung die abzuarbeitenden Schritte bestimmt. Diese verschiedenen Schritte werden meist in einem „Playbook“ zusammengefasst. Ein solches „Playbook“ kann zur Konfiguration mehrerer Systeme genutzt werden und bildet den gewünschten Konfigurationsstand ab, der nach Ausführung erreicht werden soll [16].

Neben der Konfiguration von bspw. Linux-basierten Systemen kann das Tool ebenfalls für das Deployment, also bspw. das Generieren von Binaries oder statischen Dateien, das Installieren dieser auf einen Server und das Starten einer Anwendung, genutzt werden [16].

2.6.4 Puppet

Puppet ist weiteres Tool für den Schritt des KMs, welches es ermöglicht, den gewünschten Zustand der Infrastruktur deklarativ zu definieren. Wie auch einige Tools dieser Reihe verwendet Puppet ebenfalls eine auf Ruby basierende DSL [17].

Puppet verwendet ein Client-Server-Modell, bei dem die Serversoftware auf dem Server installiert ist, während jeder verwaltete Rechner die Installation der Client-Software enthält [17]. Im Vergleich zu Ansible nimmt die Installation von Puppet einige Zeit in Anspruch und erfordert komplexe Konfigurationseinstellungen. Es bietet hohe Skalierbarkeit und Verfügbarkeit durch Replikation der Daten des Masters auf einen anderen Server, welches jedoch wiederum komplexe Einstellungen erfordert. Es verwendet ein Pull-Deployment-Modell, bei dem die Agenten den Pull-Ansatz initiieren und regelmäßig nach Updates durch den Master suchen [17].

2.6.5 Chef

Chef zählt seit langem zur Reihe der IaC-Tools und siedelt sich ebenfalls am ehesten im KM an. Die, wie bei Puppet, auf Ruby basierende DSL ermöglicht die Erstellung von „Rezepten“ und „Kochbüchern“ zur modularen und wiederverwendbaren Konfiguration von Infrastruktur [18].

Der Hauptunterschied zwischen Puppet und Chef besteht darin, dass Chef eine prozedurale Sprache verwendet, welche Schritte definiert, wie der gewünschte Zustand erreicht werden soll und Puppet mit einem deklarativen Ansatz beschreibt, wie die Konfiguration aussehen soll und das Tool die daraus resultierenden Schritte automatisch ableitet und durchführt [18].

2.6.6 cloud-init

Cloud-init ist ein von Canonical veröffentlichtes Open-Source-Projekt und der Industriestandard für eine anbieterunabhängige Initialkonfiguration von Cloud-Instanzen [19]. Die Technologie ist in Python geschrieben und findet bei vielen Cloud-Lösungen, wie beispielsweise Azure und AWS, Anwendung [19].

Cloud-init arbeitet mit einer YAML-Syntax und stellt Cloud-Anbietern die Möglichkeit zur automatischen Konfiguration der Instanz mit unter anderem Netzwerkkonfigurationen, Speicher oder das Hinzufügen von SSH-Schlüsseln oder Softwarepaketen zur Verfügung [19]. Für Anwender bietet cloud-init eine Initialkonfiguration ohne die Installation von notwendigen Paketen, die beim Bereitstellen einer Cloud-Instanz ausgeführt wird.

Obwohl cloud-init ursprünglich nur auf Ubuntu-Installationen unterstützt wurde, ist dieses jetzt für die meisten Linux- und FreeBSD-Distributionen wie unter anderem ArchLinux, CentOS, Fedora oder openSUSE nutzbar [19].

2.6.7 Abstraktion durch das Open Application Model

Alibaba Cloud und Microsoft waren gemeinsam an der Entwicklung des Open Application Model (OAM) beteiligt. Dieses Modell folgt den offenen Standards für den Anwendungsbetrieb mit Kubernetes und weiteren Plattformen und stellt dabei einen Standard einer High-Level-Abstraktion dar, welcher sich aktuell noch in starker Entwicklung befindet. Bei diesem soll es möglich sein, Cloud-Native Applikationen vereinfacht in Hybrid- bzw. Multi-Cloud Umgebungen implementieren zu können. Der Fokus liegt hierbei auf den bereitzustellenden Anwendungen und nicht darauf, wie die Infrastruktur orchestriert werden muss [20].

Das OAM ist eine Spezifikation[7] für die Beschreibung von Anwendungen, mit Fokus auf der Trennung der Anwendungsbeschreibung von den Details der Bereitstellung und der Verwaltung der Anwendungsinfrastruktur. Die Trennung der Anwendungsdefinition von den betrieblichen Details eines Clusters ermöglicht es den Anwendungsentwicklern, sich auf die Kernpunkte der Anwendung selbst zu konzentrieren anstatt auf die Bereitstellung dieser. Zudem bietet es den Plattformarchitekten die Entwicklung wiederverwendbarer Komponenten, um schnell und einfach zuverlässige Anwendungen zu erstellen [20].

[7] https://github.com/oam-dev/spec

Im OAM besteht eine Anwendung aus mehreren Konzepten. Zunächst werden die notwendigen Komponenten beschrieben, aus denen eine Anwendung besteht. Mögliche Komponenten können hierbei eine Datenbank oder ein Frontend-Server mit den untereinander bestehenden Verbindungen darstellen. Diese Komponenten können anschließend als Quellcode definiert und an beliebiger Stelle wiederverwendet werden. Die Komponenten werden anschließend durch eine Konfiguration beschrieben, um eine spezifische Instanz einer Anwendung zu bilden, die bereitgestellt werden soll. Als letzte Komponente wird eine Sammlung von Merkmalen, den sogenannten Traits verwendet, welche die Eigenschaften, die für den Betrieb wichtig sind, wie bspw. Skalierung oder Routing, beschreiben [20].

Dieser Verbund an Komponenten erlaubt es, dass das OAM plattformunabhängig einsetzbar und nicht zwingendermaßen an Kubernetes gebunden ist. Ebenso ermöglicht das OAM den Plattformanbietern, die einzigartigen Merkmale ihrer Plattform über das Trait-System zu konfigurieren, so dass Anwendungsentwickler plattformübergreifende Anwendungen erstellen können, wenn die erforderlichen Traits unterstützt werden. Das OAM gibt den Entwicklern die Freiheit, auf standardisierte Weise eine Portabilität auf verschiedenen Plattformen herzustellen [20].

3 Analyse

3.1 Methodik und Umsetzung

Die Entwicklung des Konzeptes und die Implementierung wurden anhand einer Fallstudie durchgeführt. Weitergehend wurden diese Schritte durch eine Literaturrecherche und Praxistests gestützt.

Für den Vergleich von technischen Tools, die im Schritt der Orchestrierung der Infrastrukturbereitstellung zum Tragen kommen, wurden entsprechende Kriterien erarbeitet, die später in der Analyse zur Auswahl der am besten geeigneten Technologie führen sollen. Diese Auswahl soll anhand einer gewichteten Entscheidungsmatrix aufgezeigt und getroffen werden.

Für die Realisierung des Prototyps wurden verschiedene Konzepte erarbeitet, mit denen es möglich ist, eine vorgegebene Webanwendung auf den beispielhaft betrachteten CSPs zur Verfügung zu stellen. Es werden zunächst drei verschiedene Konzeptionen vorgestellt und anschließend eine Entscheidung auf Grundlage einer argumentativen Bewertung getroffen. Die Implementierung des Prototyps stützt sich hierbei auf die Grundlagen des Software-Engineerings und wertet die Ergebnisse anschließend durch eine Nutzwertanalyse aus.

3.2 Kriterien für einen Vergleich verschiedener IaC-Werkzeuge

Da zur heutigen Zeit bereits mehrere moderne IaC-Tools zur Bereitstellung von Cloud-Infrastruktur zur Verfügung stehen, erarbeiteten Özel et. al mit der Hilfe von durchgeführten Experteninterviews Auswahlkriterien zur Identifizierung des am besten geeigneten Tools [21]. Die Experteninterviews wurden mit Personen aus jenem Berufskreis geführt, welche „ein tiefes Verständnis der vielen Aufgaben, die mit dem Einsatz von Cloud-Infrastrukturen verbunden sind“, vorweisen [21].

Konfigurationsmanagement vs. Orchestrierung
Die in den Grundlagen definierten vier Teilschritte innerhalb der Infrastrukturbereitstellung können als Orchestrierung und KM generalisiert werden [21]. Tools zur Orchestrierung sind dafür vorgesehen, um die grundlegende Infrastruktur auf der Ebene der CSPs bereitzustellen. Der Schwerpunkt liegt hierbei auf dem Arrangieren und dem Koordinieren der Konfiguration in komplexen Umgebungen [22]. Die Konfiguration einzelner Ressourcen werden anschließend von Tools des KMs durchgeführt. Hierzu gehören bspw. die „Installation von Paketen, das Starten von Diensten oder die Installation von Skripten auf den Instanzen“ [21].

Die Trennung der Belange hinsichtlich der einzelnen Rollen kann jedoch nur bedingt gewährleistet werden. Teilweise sind Tools, die ursprünglich für das KM konzipiert wurden, auch dazu geeignet, Aufgaben der Orchestrierung zu erfüllen und vice versa [22]. In den meisten Fällen wird jedoch empfohlen, jeweils ein eigenständiges Tool für das KM und für die Orchestrierung zu nutzen [1].

Wandelbare Infrastruktur vs. Unwandelbare Infrastruktur
Das Paradigma der sich wandelnden Infrastruktur beschreibt das ursprüngliche Starten eines Servers mittels eines Tools zur Orchestrierung ohne jegliche Konfiguration, auf welchem dann mittels KM die gewünschte Konfiguration hergestellt wird [21]. Die sich zur Laufzeit ergebenden Änderungen an der Konfiguration werden anschließend über das KM vorgenommen. Durch immer wieder stattfindende Aktualisierungen kann das Phänomen des Konfigurationsdrifts auftreten. Ein solcher Konfigurationsdrift tritt dann auf, wenn die Konfiguration auf mindestens einem vorhandenen Server abweicht, sodass Konfigurationsfehler auftreten, welche zur Instabilität und zu Fehlern der Anwendung führen können [1].

Im Gegensatz dazu steht die unwandelbare Infrastruktur. Hierbei werden vorgefertigte Images, beispielsweise Docker-Images, eingesetzt, welche mittels eines Tools der Orchestrierung bereitgestellt werden [1]. Diese Images beinhalten bereits alle Konfigurationsdateien- und Einstellungen, sodass ein Bereitstellungsprozess über diesen Weg ohne die Notwendigkeit eines Tools des KMs auskommt. Konfigurationsänderungen werden anschließend mit dem Austausch des Images realisiert und alte Instanzen einfach entfernt [1]. Durch diesen Ansatz wird die Wahrscheinlichkeit von Konfigurationsdrifts minimiert und die Erkennbarkeit der aktuell vorherrschenden Konfigurationsversion, bspw. anhand eines Image-Tags, vereinfacht [1].

Prozedurale Sprache vs. Deklarative Sprache (siehe Abschnitt 2.3.3)

Cloud-Nativ vs. Cloud-Agnostisch
Die meisten populären Tools arbeiten cloud-agnostisch und lassen sich daher in Verbindung mit vielen CSPs wie AWS oder Azure integrieren [21]. Es existieren im Gegensatz dazu jedoch auch einzelne herstellerbezogene Lösungen wie CloudFormation, welches ein cloud-natives Tool für die AWS darstellt und sich auch nur in dieser vollständig integrieren lässt [21].

Gerade bei einem Multi-Cloud-Ansatz ist die Verwendung eines cloud-agnostischen Tools empfehlenswert, da mit diesem mehrere Plattformen gleichzeitig bedient werden können. Eine direkte Weiterverwendung des gleichen Quelltextes ist jedoch in den meisten Fällen nicht direkt möglich, da jeder CSP eigene Namen für Ressourcen und Technologien verwendet, was eine Anpassung im Code unumgänglich macht [21].

Open Source vs. Enterprise
Unterschieden werden können IaC-Tools ebenfalls hinsichtlich ihrer Lizenz, unter welcher diese veröffentlicht wurden. Allgemein kann so zwischen Open-Source und Closed-Source unterschieden werden. Bei Open-Source-Software ist der Quellcode öffentlich zugängig. Über die vergebene Lizenz ist zudem geregelt, inwiefern weitere Änderungen und Veröffentlichungen stattfinden dürfen.

Bei Closed-Source-Software ist der Quellcode nicht öffentlich zugängig. Der Anwender erhält, meistens von einem Unternehmen, eine ausführbare Version der Software, ohne Zugriff auf den Quellcode zu besitzen.

Closed-Source sowie Open-Source-Tools haben die Eigenschaft, dass diese meist über den Kauf einer Enterprise-Version Vorteile wie erweiterten Support beinhalten. In der Regel bietet Open-Source-Software ohne kostenpflichtige Erweiterungen jedoch den Vorteil, kosteneffektiver zu sein, welches die OpEx minimieren könnte [21].

Master vs. Masterlos
Tools zur Orchestrierung arbeiten standardmäßig masterlos, wohingegen Tools des KMs einen Master-Server benötigen, von dem aus der Zustand der Cloud-Infrastruktur verwaltet wird und Aktualisierungen ausgelöst werden [1].

Die Verwendung eines Master-Servers bietet sich in der Hinsicht an, dass damit ein Konfigurationsdrift möglichst vermieden werden kann. Der Master-Server verwaltet zentral den Status der Konfiguration und stellt sicher, dass manuelle Änderun-

gen an der Infrastruktur zum Teil auch erkannt und bei Bedarf auch wiederhergestellt werden können [1].

Der Betrieb eines solchen Master-Servers ist jedoch mit hohem Aufwand verbunden, da dieser über einen zusätzlichen Server unter Gewährleistung von hoher Verfügbarkeit und Skalierbarkeit bereitgestellt werden muss. Zusätzlich wird eine ständige, sichere Kommunikationsverbindung zwischen den einzelnen Knoten und dem Master-Server benötigt, über welchen die Konfigurationsanweisungen ausgetauscht werden können [1].

Agent vs. Agentlos
Das Gegenstück zu einem Master-Server stellt der Client-Agent dar, welcher zusätzlich auf jedem Server installiert werden muss, um die übermittelte Konfiguration zu empfangen. Ein solcher Agent arbeitet auf den Knoten als Hintergrundprozess und sorgt dafür, dass die neuesten Updates und Konfigurationsänderungen übernommen werden [1]. Auch dieser muss instandgehalten werden und mit einer ständigen, sicheren Kommunikationsverbindung mit dem Master-Server kommunizieren können [21].

Große Community vs. Kleine Community
Besonders wichtig bei der Auswahl eines IaC-Tools ist die Größe der dahinter aktiven Community, welche sich um die Weiterentwicklung kümmert und Nutzerfragen beantwortet [21]. Die Popularität eines Tools, welche sich anhand der Nutzer, der verfügbaren Provider oder der zu dem Thema beantworteten Fragen messen lässt bestimmt zumeist die Attraktivität eines solchen [1].

	Terraform	**Pulumi**	**Chef**	**Puppet**	**Ansible**
Typ	Orchestrierung	Orchestrierung	Konfig.-Management	Konfig.-Management	Konfig.-Management
Infrastruktur	unwandelbar	unwandelbar	wandelbar	wandelbar	wandelbar
Sprache	deklarativ	deklarativ	prozedural	deklarativ	prozedural
Cloud	Cloud-Agnostisch				
Source	Open-Source				
Master-Server	masterlos	masterlos	Master	Master	masterlos
Client-Agent	agentlos	agentlos	Agent	Agent	agentlos

Tabelle 3.1: Übersicht und Vergleich populärer IaC-Tools [21, Tab. 2] (erweitert)

Alle abgebildeten Tools in Tabelle 3.1 sind für einen Cloud-agnostischen Ansatz vorgesehen, unter einer Open-Source-Lizenz veröffentlicht und für Unternehmen nutzbar. Für den weiteren Verlauf sollen ausschließlich Tools der Kategorie Orchestrie-

rung betrachtet werden. Dies geschieht auf der Grundlage der Kriterien der Infrastruktur und der Notwendigkeit von Master-Server und Agent-Client in 3.1.

Eine unwandelbare Infrastruktur soll angestrebt werden, um Konfigurationsdrifts auszuschließen und zudem einen Überblick über die installierte Version zu erlangen. Die Images, welche hierfür genutzt werden, können zudem ebenfalls im Entwicklungsprozess genutzt werden. Da die Bereitstellung eines Master-Servers und die eines Client-Agents auf jedem Server mit einem hohen Aufwand verbunden sind, soll auch hier möglichst ein Tool gewählt werden, welches von der Notwendigkeit dieser absieht [21]. Terraform und Pulumi sollen aufgrund dessen im weiteren Verlauf detaillierter betrachtet und miteinander verglichen werden, da mit diesen die Bereitstellung einer unwandelbaren Infrastruktur möglich gemacht wird (siehe Tabelle 3.1).

3.3 Kriterien für einen funktionalen Vergleich von Tools zur Orchestrierung

Populäre Orchestrierungs-Tools stellen eine breite Palette an Funktionalitäten zur Verfügung, um verschiedenste Problemstellungen auf unterschiedlichsten Cloud-Lösungen zu realisieren. Um diese Tools miteinander vergleichen zu können, bieten sich die folgenden Kriterien hinsichtlich der beispielhaften CSPs an (vgl. [15]):

Sprache
Verfügbare Unterstützung unterschiedlicher Programmiersprachen zur Abbildung der Konfiguration. Es wird allgemein zwischen DSL und GPL unterschieden.

State/Service
Der State wird genutzt, um existente Ressourcen und die Konfiguration dieser zu verknüpfen, um Metadaten zu vergleichen und auch große Infrastrukturen verwalten zu können. Es wird allgemein zwischen einem Self-Hosted Ansatz und einem PaaS-Angebot des Herstellers unterschieden.

Provider/Plugins
Gibt an, welche Cloud-Lösung verschiedener Anbieter mithilfe des Orchestrierungs-Tools konfiguriert werden können.

Modularität
Beschreibt die Wiederverwendbarkeit vorhandener Ressourcentypen mit geringe-

rem Aufwand.

Secrets
Beschreibt die Möglichkeiten, wie Secrets, welche innerhalb der Konfiguration genutzt werden, sicher gespeichert und abgerufen werden können.

Test/Validierung
Zeigt die Möglichkeiten auf, wie eine vorhandene Konfiguration getestet und validiert werden kann. So können Fehler in der Konfiguration bereits vor einem Ausrollen kenntlich gemacht und behoben werden.

Community
Vergleicht die Größe der hinter dem jeweiligen Tool aktiven Community.

3.4 Detaillierterer Vergleich von Terraform und Pulumi

		Terraform		Pulumi	
Kriterium	**Gewichtung**[1]	**Punkte**[2]	**gewichtete Punkte**	**Punkte**	**gewichtete Punkte**
Sprache	0.10	4	0.4	6	0.6
State/Service	0.15	6	0.9	4	0.6
Provider/Plugins	0.20	6	1.5	4	0.8
Modularität	0.15	5	0.75	5	0.75
Secrets	0.10	2	0.2	4	0.4
Test/Validierung	0.10	5	0.5	6	0.6
Community	0.20	6	1.2	3	0.6
Summe gewichteter Punkte			5.45		4.35

Tabelle 3.2: Gewichtete Entscheidungsmatrix Terraform und Pulumi

Sprachen
Terraform verwendet eine DSL, welche unter dem Namen HCL geführt wird [1]. Die HCL beschreibt deklarativ den gewünschten Zustand der Infrastruktur. Die DSL zeichnet sich durch ihr einfach gehaltenes, menschenlesbares Format aus und erreicht schnell einen hohen Lernerfolg. Die Verwendung des Cloud Development Kit for Terraform (CDKTF)[3] macht zusätzlich die Verwendung GPLs möglich. Die gesamte Verwendung von Terraform ist dementsprechend auch ohne das Lernen der HCL möglich [23].

[1] Die Gewichtung der evaluierten Kriterien für den Vergleich ist in die Kategorien „niedrig“ (0.1), „mittel“ (0.15) und „hoch“ (0.20) eingeteilt.
[2] 0 Punkte: Kriterium überhaupt nicht erfüllt, 6 Punkte: Kriterium vollständig erfüllt
[3] https://developer.hashicorp.com/terraform/cdktf

Pulumi arbeitet im Gegensatz dazu mit GPLs. Somit können die Konfigurationen in populären Programmiersprachen wie beispielsweise TypeScript, Go, Python oder Java deklariert werden [15]. Obwohl Pulumi ebenfalls als deklaratives Werkzeug agiert, verwendet es im Unterbau imperative Programmiersprachen, was bedeutet, dass die Anwender die Logik zur Bereitstellung der Konfiguration selbst durch Kontrollstrukturen verändern könnten [15].

State/Service

Standardmäßig wird der aktuelle Stand der Cloud-Infrastruktur bei der Verwendung von Terraform in der lokalen Datei „terraform.tfstate“ gespeichert [14]. In Umgebungen, in welchen die Infrastruktur von Teams verwaltet wird, bietet sich ein solcher Ansatz nicht an. Terraform stellt hierfür das Konzept der „Remote States“ bereit. Bei diesem Ansatz wird die State-Datei zentral gespeichert und kann so von mehreren Anwendern abgefragt und modifiziert werden. Möglichkeiten zur Speicherung sind beispielsweise die Terraform Cloud[4], Amazon S3, Azure Blob Storage oder Google Cloud Storage [14]. Die Verwendung der von HashiCorp gegen Lizenzgebühr bereitgestellten Software-as-a-Service (SaaS)[5] ist auch im Business-Kontext nicht zwingend notwendig.

Pulumi setzt standardmäßig auf einen PaaS-Ansatz und verwaltet den aktuellen Stand der Konfiguration in dem für Unternehmen kostenpflichtigen Pulumi Service[6] [15]. Dieses Vorgehen macht es zunächst einfacher, Pulumi als Team nutzen zu können, verursacht jedoch auch erhöhte OpEx. Für tiefergehende Anwendungszwecke und die Härtung der Sicherheit kann auch die Verwendung des Managed Pulumi Services umgangen und ein Self-Hosted-Ansatz[7] gewählt werden. Diese Umsetzung ist mit zeitlichem Aufwand verbunden, unterstützt jedoch nach Konfiguration die gleichen Speichermöglichkeiten wie Terraform [15].

Provider/Plugins

Sowohl Terraform als auch Pulumi unterstützen eine Vielzahl der führenden CSP und moderner SaaS. Laut Terraform Registry bietet Terraform aktuell für 2.872 Plattformen, einschließlich AWS, Azure, Google Cloud oder Kubernetes, einen Provider an. Pulumi bietet zusätzlich native Provider für AWS, Azure, Google und Kubernetes an, welche eine Unterstützung für Same-Day Updates bei neu veröffentlichten Versionen der API anbieten. Pulumi ist zudem Mitglied der Cloud Native Computing

[4]Terraform Cloud ist die PaaS für die Verwendung von Terraform in der Cloud
[5]https://app.terraform.io/
[6]https://app.pulumi.com/
[7]https://www.pulumi.com/docs/guides/self-hosted/

Foundation (CNCF)[8].

Pulumi bietet bisher keinen Provider für die Verwendung der OTC an. Pulumi kann aus der Historie gewachsen über die Pulumi Terraform Bridge[9] zwar Terraform Provider in einem gewissen Rahmen nutzen, jedoch können hierbei nicht alle Funktionalitäten der OTC abgebildet werden.

Modularität

Im Gegensatz zu den limitierten Möglichkeiten in der prozeduralen Entwicklung bieten sowohl Terraform als auch Pulumi mit Ihrer deklarativen Definitionssprachen Vorteile hinsichtlich der Wiederverwendbarkeit von bereits vorhandener Infrastruktur [1].

Terraform stellt zur Gewährleistung der Wiederverwendbarkeit das Konzept der Terraform Modules bereit. Der Quellcode, in dem eine bestimmte Komponente, wie beispielsweise ein SQL-Server, beschrieben wird, kann so einmalig in ein Modul gegossen werden (siehe Abbildung 3.1). Terraform verzichtet somit auf das Kopieren von Quellcode und bietet die Verwendung des definierten Moduls an beliebigen Stellen im Quellcode. Durch die Verwendung von Eingangs- und Ausgangsvariablen während der Verwendung eines Moduls bleibt dieses trotzdem individualisierbar [1]. Die Module, die in Terraform definiert werden, können in der Terraform Registry publiziert werden. Dort finden sich unter anderem offizielle, von HashiCorp entwickelte Module, sowie weitere, die durch jeden Nutzer beliebig bereitgestellt werden können [14].

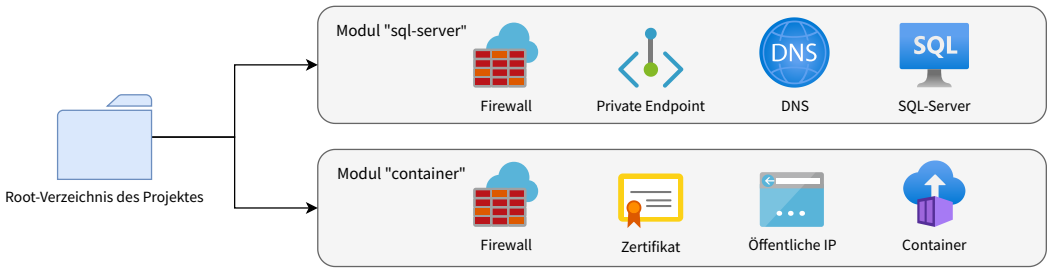

Abbildung 3.1: Beispiel der Module „sql-server“ und „container“ mit möglichen Bestandteilen, die als Modul zusammengefasst werden

Pulumis Vorteil liegt bei diesem Punkt in der Unterstützung populärer GPLs. In den meisten dieser GPLs können Funktionen, Klassen und Pakete definiert und wiederverwendet werden, welches für die Abstraktion bereits definierter Infrastruktur in

[8]Die CNCF ist Teil der Linux Foundation und hostet Projekte wie Kubernetes und Prometheus
[9]https://github.com/pulumi/pulumi-terraform-bridge

Pulumi verwendet werden kann. Wie auch bei Terraform bietet Pulumi die Möglichkeit, dass Anwender ihre erstellten Pakete über den Dienst Pulumi Packages für Dritte zur Verwendung bereitstellen können [15].

Secrets

Pulumi speichert sensible Werte als Secrets für zusätzlichen Schutz standardmäßig verschlüsselt. Diese werden in keiner Situation für Anwender sichtbar, wenn dies nicht explizit gewünscht ist, sodass das versehentliche Preisgeben eines Secrets ausgeschlossen werden kann. Zusätzlich bietet Pulumi eine breite Möglichkeit der Integration externer Dienste für die Verwaltung von Secrets [15].

Bei der Verwendung von Terraform werden als geheim markierte Secrets zwar in Ausgaben und Logs geschwärzt, jedoch unverschlüsselt in der State-Datei gespeichert. Eine mögliche Variante, um dieses Problem zu lösen ist unter anderem die Verwendung von einem sicheren und verschlüsselten Backend, in welchem die State-Datei gespeichert wird. Zudem könnten Secrets auch vollständig außerhalb des Quellcodes in Umgebungsvariablen gespeichert werden und über eine Variable zur Laufzeit eingelesen werden (siehe Quellcode 3.1) [14].

```
variable "password" {
    description = "The password for the admin user"
    type        = string
    sensitive   = true
}
```

Quellcode 3.1: Beispiel einer sensitiven Variable in Terraform

Mithilfe des Prefixes `TF_VAR` vor einem in Terraform definierten Variablennamen, kann dieser Wert injiziert werden. Der Inhalt der Umgebungsvariable `TF_VAR_password` würde in Terraform als Wert der Variable `password` aus Quellcode 3.1 interpretiert werden.

Test/Validierung

Um erstellte Infrastruktur-Ressourcen zu testen, kann nicht die lokale Hard- und Software verwendet werden. Dieses Problem trifft auf die meisten IaC Tools zu. Die einzige Möglichkeit, um die Infrastruktur sinnvoll und effektiv testen zu können ist das Ausrollen auf die reale Cloud-Lösung, wenngleich natürlich bevorzugt zuerst in einen Testabschnitt bzw. einer Sandbox innerhalb dieser. Wichtig ist auch, dass die zum Test erzeugten Ressourcen auch nach erfolgreichem Abschluss von dieser Umgebung wieder deaktiviert und entfernt werden, um Kosten und Ressourcen zu schonen [1].

Die Idee hinter dem Testen und der Validierung des Quellcodes ist jedoch ein automatisierter Test, um zu prüfen, ob die Konfiguration genau das bewirkt, was bezweckt werden soll. Es wird allgemein zwischen *Unit tests*, *Integration tests* und *End-to-end tests* unterschieden [1].

Im Kontext der IaC erweitert man diese Arten jedoch um die *Static analysis*, das *Plan testing* und das *Server testing* [1]. Terraform bietet in allen drei Bereichen eine Vielzahl an Erweiterungen, die zum Testen und Validieren der Konfiguration genutzt werden können. Für die *Static analysis* kann u.a. die eingebaute Funktion „terraform validate“ oder tfsec[10] genutzt werden. Das *Plan testing* kann mithilfe von Technologien wie Terratest[11] oder Checkov[12] realisiert werden. Für das *Server testing* können Tools wie InSpec[13] oder Serverspec[14] verwendet werden.

Mit Pulumi können hingegen native Test-Frameworks ohne die Verwendung externer Tools genutzt werden. Diese Möglichkeiten beinhalten unter anderem Unit tests, die externe API-Aufrufe nachbilden, Property-Tests, welche Tests bezogen auf einzelne Ressourcen ausführen, während diese ausgerollt werden und Integration tests, welche die gewünschte Konfiguration ephemer ausrollen und externe Tests durchführen [15].

Community

	Contributors	Sterne	Commits (2022)	Issues (2022)	Libraries	Stack Overflow[15]
Terraform	1.691	35.918	1.357	1.349	12.213[16]	16.182
Pulumi	206	14.902	2.158	1.224	125[17]	407

Quelle: Stand: 28. Januar 2023

Tabelle 3.3: Vergleich von IaC-Tool Communities

Vergleicht man die Community-Größen beider Tools miteinander kommt man schnell zu dem Schluss, das jene von Terraform der von Pulumi quantitativ überlegen ist (siehe Tabelle 3.3). Durch die langjährige Verfügbarkeit kann Terraform eine höhere Anzahl an „Contributors“, also am Projekt mitarbeitenden Personen, verzeichnen. Obwohl Pulumi erst seit wenigen Jahren verfügbar ist, konnte das

[10] https://github.com/aquasecurity/tfsec
[11] https://github.com/gruntwork-io/terratest
[12] https://github.com/bridgecrewio/checkov
[13] https://github.com/inspec/inspec
[14] https://github.com/mizzy/serverspec
[15] https://stackoverflow.com/
[16] Anzahl der Module in der Terraform Registry
[17] Anzahl der Packages in der Pulumi Registry

Tool jedoch bereits an Popularität gewinnen und kommt so auf circa die Hälfte der Sterne von Terraform. Ebenfalls erkennt man anhand der Anzahl der Commits des vergangenen Jahres, dass sich Pulumi noch in stärkerer Entwicklung befindet als Terraform. Besonders auffallend ist der Unterschied in den für das jeweilige Tool verfügbaren Libraries. Libraries beinhalten unter anderem Module, Plugins oder Provider für das jeweilige Tool, welche von der Community oder offiziellen Entwicklern zentral bereitgestellt werden. Die Popularität lässt sich ebenfalls an den gestellten und beantworteten Fragen auf Stack Overflow definieren. Hier erkennt man, dass Terraform ausgiebig von Anwendern genutzt und diese von anderen Anwendern dabei unterstützt werden.

3.4.1 Auswahl des zu verwendenden Tools für den Prototyp

Sowohl Terraform als auch Pulumi unterstützen ein breites Spektrum von Cloud-Lösung verschiedenster Anbieter. Beide sind zudem unter einer Open-Source-Lizenz veröffentlicht, sodass beide Technologien, auch im Business-Kontext, kostenfrei genutzt werden und durch die Community stets weiterentwickelt werden können. In Bezug auf die Grundfunktionalitäten gleichen sich die beiden Tools stark.

Pulumi ist ein neueres und entwicklungsfreundliches Tool, welches seit Veröffentlichung schnell an Popularität gewinnen konnte. Ein besonderer Vorteil von Pulumi ist die Verwendung einer Vielzahl an unterstützten Programmiersprachen zur Konfiguration der Infrastruktur, bei welcher auf Grundkonzepte der Softwareentwicklung, wie beispielsweise Schleifen oder Bedingungen zurückgegriffen werden kann.

Da Pulumi jedoch noch nicht lange verfügbar ist, verfügt es noch nicht über das gleiche Maß an Plattformunterstützung und umfangreicher Dokumentation wie die von Terraform. Zu den wichtigsten Vorteilen von Terraform zählen die breite Plattformunterstützung, die einfache Bedienung und die Community-Module, welche von Nutzern entwickelt und über die Terraform Registry[18] bereitgestellt werden können. Eine breite Plattformunterstützung bedeutet, dass Terraform für die Verwaltung eines Cloud-agnostischen Ansatzes besser geeignet ist, als die untersuchte Konkurrentin. Die Benutzerfreundlichkeit und die der großen Community zu verdankende Stabilität von Terraform zeichnen die Technologie für die Verwendung in Business-Kontexten aus.

[18] https://registry.terraform.io/

3.5 Docker zur Bereitstellung von Webanwendungen

Es existieren verschiedene Wege, um eine Anwendung für Kunden bereitzustellen. Beispiele sind unter anderem Container, SaaS, VMs, FaaS oder Bare-Metal-Server [24].

Das Devops Research & Asessment (DORA)[19] State of DevOps research program beschäftigt sich bereits seit 2014 mit der Identifizierung der effizientesten und effektivsten Wege, um Software zu entwickeln und bereitzustellen. Die Organisation veröffentlicht jährlich ihren „State of DevOps Report“ [24]. Der Bericht liefert wichtige Kriterien, mit denen die Bereitstellung und die operative Performanz von Infrastruktur in einem Unternehmen gemessen werden können. Erstmalig 2021 fragte DORA die Teilnehmer nach dem jeweiligen Deployment-Ziel, welches diese für ihre Anwendung nutzen. Die Teilnehmer gaben an, dass mit 54% die meisten Deployments über die Bereistellung von Containern wie Docker oder Kubernetes durchgeführt wurden [24].

Docker ist ein Open-Source Tool, über welches Anwendungen schnell als Image verpackt und gestartet werden können. Die Container werden häufig für die Entwicklung, das Testen und das Deployment von Anwendungen eingesetzt. Eigene Inhalte, wie Anwendungen oder Tools, werden als Docker-Image verpackt und können so plattformunabhängig ohne weitere Abhängigkeiten auf verschiedenen Systemen ausgeführt werden [25]. Durch die auf Container basierende Plattform werden einzelne Container voneinander isoliert betrieben. Hierfür werden diverse Linux-Konzepte wie beispielsweise namespaces, cgroups oder UFS eingesetzt [25]. Docker-Container können auf VMs, physischer Hardware oder Cloud-Infrastruktur von bspw. Google, Microsoft oder Amazon betrieben werden [25]. Jeder Container besitzt hierbei ein eigenes Dateisystem, eigene Prozesse, eigenen Arbeitsspeicher und eine eigene Netzwerkkonfiguration, die auf physische Hardwareschnittstellen gemappt werden kann [25].

Docker-Container erfüllen das Kriterium der Unwandelbarkeit, indem Docker-Images anhand eines Tags bzw. einer Versionsnummer definiert und gebaut werden und bereits alle abhängigen Bibliotheken sowie die Anwendung selbst beinhalten. Bei der Veröffentlichung einer neuen Version der Konfiguration oder der Anwendung selbst muss lediglich ein neues Image erzeugt und dieses im Container anhand des Image-Namens oder des Tags ausgetauscht werden.

[19] DORA ist ein Teil von Google [24]

4 Konzeption

In diesem Kapitel sollen verschiedene entwicklerische Ansätze der Infrastrukturbereitstellung für eine Webanwendung auf den Cloud-Lösungen Azure und OTC vorgestellt werden und auf Grundlage eines Vergleiches der Ansätze untereinander eine Auswahl für die Implementierung des Prototyps getroffen werden.

Die Besonderheit findet sich in der Abstraktion des vorgesehenen Ansatzes wieder. Es soll versucht werden, die Webanwendung in einer möglichst modularen Form über beide Umgebungen bereitzustellen. Die auf den Clouds verfügbaren Ressourcen sollen möglichst einheitlich genutzt werden, sodass beiderseitig die gleichen technischen Rahmenbedingungen gegeben sind.

Neben der Bereitstellung einer Webanwendung sollte ein nachhaltiger Ansatz entwickelt werden, mit dem die Webanwendung unabhängig von den Anbietern und unter möglichst geringen Anforderungen über verschiedene Anbieter abgebildet werden kann.

4.1 Modellierung der Abstraktion

Die Konzeption beinhaltet ein Multi-Cloud-Vorhaben, bei welchem eine VM mithilfe einer Abstraktion auf die Cloud-Lösungen der Anbieter Azure und OTC bereitgestellt werden soll. Der Ressourcentyp der virtuellen Maschine bietet sich für dieses Vorhaben besonders an, da dieser auf beiden CSPs mit einer Vielzahl an Ähnlichkeiten abgebildet wird. Die Abstraktion dieser virtuellen Maschine lässt sich in Terraform über das Konzept der Module (siehe Abschnitt 3.4) darstellen. Es soll schließlich ein Modul pro CSP entwickelt werden, welche die gleichen Eingangs- und Ausgangsvariablen nutzen, um möglichst abstrakt, unabhängig von dem CSP, agieren zu können. Der Aspekt der unwandelbaren Infrastruktur soll mithilfe von cloud-init realisiert werden. Mittels cloud-init wird anschließend das jeweilige Docker-Image nach Erstellen der VMs als Container ausgeführt.

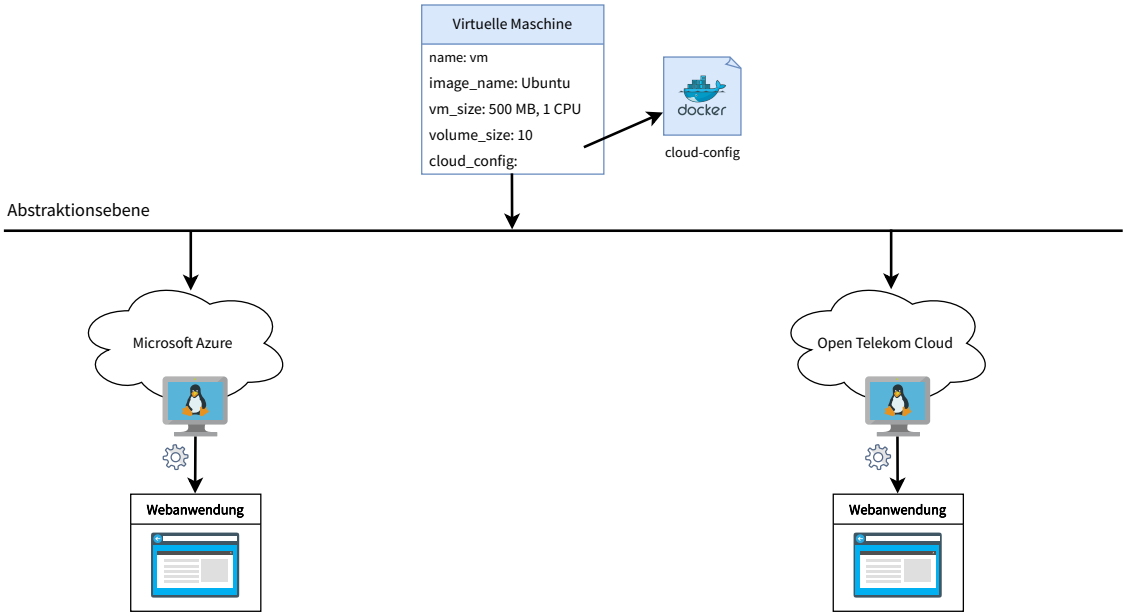

Abbildung 4.1: Modellierung der Abstraktionsebene

Zu Testzwecken wird ein Beispiel[1] verwendet. Dieses Beispiel zeigt bei erfolgreichem Deployment hilfreiche Metadaten zum Debugging als statische Seite an.

4.2 Betrachtung der Konzepte

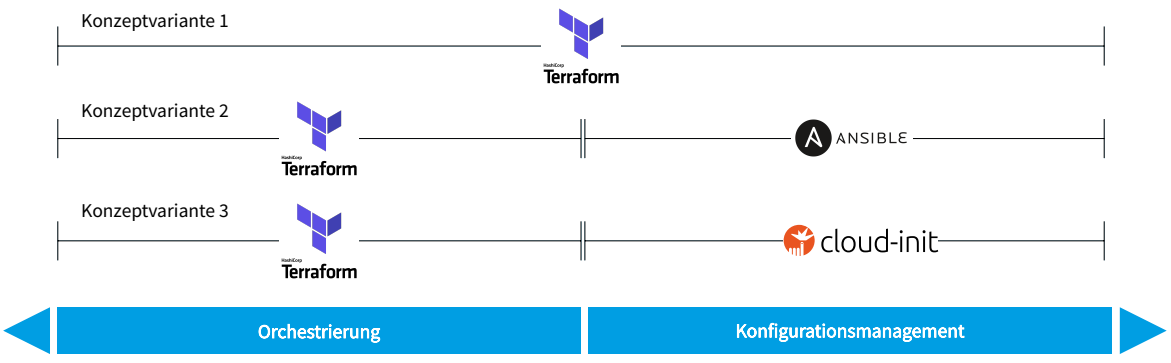

Abbildung 4.2: Vergleich der verschiedenen Konzeptvarianten

[1] https://hub.docker.com/r/nginxdemos/hello/

4.2.1 Konzeptvariante 1 - Umsetzung aller Schritte innerhalb der Provisionierung

Aufgrund der bei Terraform unterstützen Ergänzungs- und Erweiterungsmöglichkeiten können verschiedene Module aus der Terraform Registry genutzt werden, um in einer Konfigurationsdatei nicht nur die Infrastruktur bereitzustellen, sondern auch direkt Einstellungen, wie die Installation und Konfiguration eines Webservers, durchzuführen.

Hierzu existieren neben der Erstellung von Ressourcen auf verschiedenen Cloud-Lösungen beispielsweise auch Provider zum Generieren und Hinzufügen von SSH-Schlüsseln zum Server, zur Verwaltung von Docker-Containern oder Kubernetes-Cluster oder gar Provider zur selbständigen Installation von Softwarelösungen wie Grafana.

Für von der Community noch nicht angebotene Anwendungsfälle kann zudem der von Terraform bereitgestellte Provisioner `remote-exec` genutzt werden. Über eine aufgebaute SSH-Verbindung ist es dadurch möglich, beliebige Konsolenbefehle über Terraform auf einem entfernten Server auszuführen.

4.2.2 Konzeptvariante 2 - Provisionierung mit Konfigurationsmanagement

Die in den Grundlagen zur IaC erläuterten Prozessschritte können mit guter Integration miteinander kombiniert und die Tätigkeiten auf zwei Tools aufgeteilt werden. So bietet es sich an, nach der Orchestrierung direkt das KM durchzuführen.

Hierfür können beispielsweise Terraform und Ansible im Verbund verwendet werden. Mithilfe von Terraform wird als Erstes die grundlegende Infrastruktur, inklusive der Netzwerktopologie, der Datenbanken und der Server erzeugt. Ansible wird anschließend dafür genutzt, um eine Konfiguration der einzelnen Bestandteile durchzuführen und die Anwendung auf die Server zu deployen [1].

Dieses Vorgehen bietet sich bevorzugt an, da Terraform als auch Ansible Client-seitig arbeiten und somit kein weiterer Verwaltungsaufwand, wie die Installation von Master- oder Client-Knoten, für diesen Prozess nötig wird. Das Zusammenspiel dieser Technologien kann man bis ins kleinste Detail verfeinern. Beispielhaft vergibt Terraform unterschiedliche Tags an verschiedene Server, um so Ansible kenntlich zu machen, welche Aufgaben bei dieser Art Server notwendig sind [1].

4.2.3 Konzeptvariante 3 - cloud-init

Mit dem Tool cloud-init ist es möglich, bei dem Bereitstellen einer virtuellen Maschine einmalige Skripte und Befehle zur Konfiguration während des ersten Startvorgangs auszuführen. Hierzu zählen Funktionalitäten wie bspw. die Installation bestimmter Software, die Erstellung von Verzeichnissen, Benutzern, Gruppen oder das Hinzufügen von SSH-Schlüsseln.

Cloud-init verfügt bereits über eine breite Auswahl von vorgefertigten Modulen zur Durchführung von standardmäßigen Aufgaben der Konfiguration[2]. Weiterhin bietet cloud-init die Möglichkeit, individuelle Shell-Befehle anhand des Moduls `runcmd` auszuführen.

4.3 Bewertung der Konzepte

4.3.1 Vergleich und Kritikpunkte der einzelnen Konzeptionen

Konzeption 1

Mit der Konzeptvariante 1 bietet sich eine einfache Möglichkeit, eine erweiterte Konfiguration einer bereitgestellten Infrastruktur durchzuführen. Die Konfiguration wird hierbei über die Vielzahl an verfügbaren Provider direkt in Terraform durchgeführt. Es entfällt also die Notwendigkeit an weiterführender Software. Mithilfe der angebotenen Provider können einfache Schritte zur Konfiguration, wie der Upload von SSH-Schlüsseln ausgeführt werden.

Obwohl die Ressourcen zunächst anhand deklarativer Definition bereitgestellt werden, erfolgt die Erweiterung mittels eines imperativen Ablaufs. Es gilt nicht als best-practice, eine Mischung aus deklarativem und imperativen Quellcode zu verwenden. Falls dies trotzdem Anwendung findet, sollte eine strikte Trennung bezüglich der Belange gezogen und möglichst kein Mix aus deklarativem und imperativem Quellcode verwendet werden [11].

Konzeption 2

Die Verwendung von Terraform und Ansible ist eine häufig verwendete Kombination von Tools in der Praxis. Durch strikte Trennung der Belange können die verschiedenen Quellcodes getrennt verwaltet und ausgeführt werden.

[2] https://cloudinit.readthedocs.io/en/latest/reference/modules.html

Es besteht in dieser Konzeption jedoch der zusätzliche Aufwand zur Pflege zwei voneinander verschiedener Quellcodes in unterschiedlichen Konfigurationssprachen. Terraform arbeitet mit der HCL, Ansible mit Konfigurationsdateien im YAML-Format. Zudem ist die Installation und die Wartung eines extra Tools notwendig. Wenn zuvor ebenfalls keine Erfahrung in der Verwendung von Ansible vorhanden sind, muss dieses zunächst eingeführt und dessen Konzepte erlernt werden, welches einen erhöhten zeitlichen Aufwand darstellt.

Konzeption 3

Die Verwendung von Terraform zur Orchestrierung in Verbindung mit cloud-init zur Initialkonfiguration bietet sich besonders an. Cloud-init findet in nahezu jeder Public- oder Private-Cloud Umgebung Unterstützung, sodass bereitgestellte Server mittels einer cloud-init Konfiguration erweiterbar sind [19]. Von Haus aus bietet cloud-init die Möglichkeit, den endgültigen Konfigurationsstand als deklarative Definition anzugeben. Die auszuführende Schritte evaluiert cloud-init selbständig. Die Konfiguration erfolgt ebenfalls wie bei Ansible in einem YAML-Dateiformat und bietet somit eine steile Lernkurve. Durch die einmalige Initialkonfiguration, welche durchgeführt wird, erreicht man eine unwandelbare Infrastruktur.

Die unwandelbare Infrastruktur wird dadurch gewährleistet, dass keine manuellen Konfigurationsänderungen durchgeführt, sondern diese über Terraform direkt abgewickelt werden sollen. Wird eine neue Version der Webanwendung veröffentlicht, so reicht der Tausch des Image-Tags in Terraform und ein erneutes Ausrollen aus, um ein Deployment durchzuführen. Auch Anpassung an den Parametern der VM selbst, wie das vertikale Skalieren des Arbeitsspeichers oder der Anzahl der CPU-Kerne kann ebenfalls über Terraform konfiguriert und beim nächsten Ausrollen umgesetzt werden.

Die formulierte cloud-init Konfiguration wird während des Schritts der Orchestrierung der bereitzustellenden virtuellen Maschine als Parameter bzw. als YAML-Datei hinzugefügt. Nach Abschluss der Bereitstellung startet automatisiert durch Mechanismen der Cloud-Provider die Ausführung der gewünschten Schritte zur Konfiguration im Hintergrund.

4.3.2 Auswahl der Konzeption

Aufgrund der geringen Komplexität und der Gewährleistung der Unwandelbarkeit der Infrastruktur wird die prototypische Umsetzung anhand der Konzeption 3 mittels Terraform und cloud-init durchgeführt.

Kriterium	**Konzept 1**	**Konzept 2**	**Konzept 3**
Anzahl Sprachen	1	2	2
Anzahl Tools	1	2	2
Komplexität	hoch	hoch	gering
Best-practice	nein	ja	ja
Unwandelbar	nein	nein	ja
Master/Client-Notwendigkeit	nein	ja	nein

Tabelle 4.1: Vergleich der Konzepte

5 Prototyp

5.1 Vorstellung der Voraussetzungen

Ziel des zu implementierenden Prototypen soll es sein, eine vorgegebene Webanwendung und die dazugehörige Konfiguration auf den Cloud-Plattformen Azure und OTC bereitstellen zu können. Hierbei soll versucht werden, die Unterschiede der Cloud-Lösungen untereinander mithilfe einer geeigneten Abstraktion zu umgehen bzw. zu vereinfachen.

- Zu Beginn wird die notwendige Infrastruktur automatisch bereitgestellt.
- Die Webanwendung soll als Docker-Image realisiert werden.
- Besteht die Webanwendung aus mehreren Bestandteilen (bspw.: Backend und Frontend getrennt), so soll dies über eine Docker-Compose Konfiguration realisiert werden.
- Die Firewall muss so konfiguriert werden, dass nur berechtigten Personen Zugriff auf die Einsicht und die Verwaltung der Anwendung gewährt wird.
- Konfigurationsänderungen der Webanwendung müssen einfach durchzuführen sein.
- Die Webanwendung soll über eine öffentliche IP-Adresse aufrufbar sein.

Die Cloud-Lösungen sind zunächst hinsichtlich Ihrer technischen Möglichkeiten für das Hosting einer Webanwendung zu überprüfen und möglichst ähnliche Konzepte zueinander zu finden. Es wird in beiden Umgebungen eine Test-Netzwerkumgebung erzeugt, in welchen die Ressourcen später angebunden werden sollen. Weiterhin ist es gewünscht, dass die zu implementierenden Terraform Module für beide Cloud-Lösungen gleiche Eingangs- und Ausgangsvariablen nutzen.

5.2 Implementierung des Prototyps

Für die Umsetzung wird auf die Terraform Provider *azurerm*[1] für die Konfiguration innerhalb von Azure und den Provider *opentelekomcloud*[2] für die OTC zurückgegriffen. Der Provider *azurerm* gilt als offiziell und wird von HashiCorp selbst veröffentlicht wohingegen der Provider für die OTC von einem eigenen Team innerhalb der T-Systems International GmbH entwickelt und über die Terraform Registry veröffentlicht wird.

5.2.1 Virtuelles Netzwerk

Zur Trennung des Netzwerkverkehrs wird üblicherweise ein virtuelles Netzwerk eingesetzt, welches mit geringfügigen Unterschieden auf beiden Plattformen technisch realisierbar ist.

Azure

Ein virtuelles Netzwerk bei Azure (siehe Quellcode 5.1) ist eine logische Abschottung innerhalb der Azure-Cloud, in der sich virtuelle Maschinen, Container und andere Ressourcen befinden können. Es bietet die Möglichkeit, die Kommunikation zwischen diesen Ressourcen zu steuern und sicherzustellen, dass nur autorisierte Verbindungen hergestellt werden können. Innerhalb eines virtuellen Netzwerkes ist es zudem möglich, Subnetze zu erstellen und somit Ressourcen besser organisieren und verwalten zu können. Dieser Service wird von Microsoft kostenfrei angeboten [26].

```
resource "azurerm_virtual_network" "vnet" {
    name                = "vnet"
    address_space       = ["10.0.0.0/16"]
    location            = azurerm_resource_group.rg.location
    resource_group_name = azurerm_resource_group.rg.name
}

resource "azurerm_subnet" "subnet" {
    name                 = "subnet"
    resource_group_name  = azurerm_resource_group.rg.name
    virtual_network_name = azurerm_virtual_network.vnet.name
    address_prefixes     = ["10.0.1.0/24"]
}
```

Quellcode 5.1: Beispiel eines virtuellen Netzwerks in Azure

[1] https://registry.terraform.io/providers/hashicorp/azurerm/latest
[2] https://registry.terraform.io/providers/opentelekomcloud/opentelekomcloud/latest

OTC

Die gleiche Art der Netzwerkkonfiguration wird innerhalb der OTC als Virtual Private Cloud (VPC) angeboten (siehe Quellcode 5.2). Eine VPC bietet neben den Vorteilen eines virtuellen Netzwerkes bei Azure zusätzlich eine virtuelle Umgebung, die eine strikte Trennung der Netzwerksegmente innerhalb eines Tenants[3] zusichert. Mit dieser Funktionalität können unter anderem Kunden oder Entwicklungs- und Produktivumgebung voneinander getrennt werden.

```
resource "opentelekomcloud_vpc_v1" "vpc" {
    name = "prototyp"
    cidr = "10.0.0.0/16"
}

resource "opentelekomcloud_vpc_subnet_v1" "subnet" {
    name       = "subnet"
    cidr       = "10.0.1.0/24"
    vpc_id     = opentelekomcloud_vpc_v1.vpc.id
    gateway_ip = "10.0.1.1"
}
```

Quellcode 5.2: Beispiel eines virtuellen Netzwerks in der OTC

Für beide wird ein Subnetz mithilfe von Terraform definiert, welches den privaten IPv4-Bereich `10.0.0.0/16` abbildet.

5.2.2 Netzwerksicherheit

Um die virtuellen Netzwerke abzusichern und zudem die Netzwerksicherheit im öffentlichen Netzwerk zu gewährleisten, werden hierfür eigene Netzwerkregeln definiert, die dies steuern. Diese Regeln werden dazu genutzt, um zu verhindern, dass unerwünschter Traffic zu Azure-Ressourcen gelangt, indem sie festlegen, welcher Traffic zugelassen oder blockiert wird. Gesteuert werden diese Regeln allgemein über die Informationen wie Quelle, Quellport, Zielport und Protokoll eines Zugriffes [27].

Bei diesem Prototyp sollen folgende eingehende Regeln exemplarisch zugelassen werden:

- Port 22 TCP für SSH-Zugriff
- Port 80 TCP für HTTP-Requests

[3] Eine Cloud-Lösung ist mandantenfähig und bietet die Möglichkeit der Trennung der Mieter (Tenants)

- Port 443 TCP für HTTPS-Requests
- Zugriffe über das ICMP-Protokoll

Der ausgehende Netzwerkverkehr wird zunächst nicht beschränkt.

Azure

In Azure wird diese Funktionalität über die Network Security Groups (NSG) abgebildet (siehe Quellcode 5.3). Die NSGs beinhalten mehrere Sicherheitsregeln und können sowohl auf Subnetze als auch auf Ressourcen wie Netzwerkinterfaces, Application Gateways oder weiteren Azure-Ressourcen direkt angewendet werden.

Eine Sicherheitsregel beinhaltet neben einem eindeutigen Namen ebenfalls eine Priorität, die Quelle und das Ziel als IP-Adresse bzw. CIDR-Block, das Protokoll, die Richtung des Verkehrs, einen Port-Bereich und eine auszuführende Aktion [27].

```
resource "azurerm_network_security_group" "secgroup" {
  name                = format("%s-secgroup", var.name)
  location            = var.location
  resource_group_name = var.resource_group_name
}

resource "azurerm_network_security_rule" "secgroup_rule_ssh" {
  name                        = "SSH"
  priority                    = 100
  direction                   = "Inbound"
  access                      = "Allow"
  protocol                    = "Tcp"
  source_port_range           = "*"
  source_address_prefix       = "*"
  destination_port_range      = "22"
  destination_address_prefix = "*"
  resource_group_name         = var.resource_group_name
  network_security_group_name = azurerm_network_security_group.secgroup.name
}
```

Quellcode 5.3: Beispiel einer Network Security Group in Azure

OTC

Innerhalb der OTC nennt sich diese Funktionalität Security Group und beinhaltet ebenfalls wie die NSGs bei Azure einzelne Access Control Rules (siehe Quellcode 5.4). Die Security Groups können wiederverwendet werden und so bei mehreren Ressourcen genutzt werden. Einer Ressource kann zudem eine oder mehrere Security Groups zugewiesen werden [28].

Eine Security Group Rule benötigt für die Erstellung ebenfalls wie bei Azure Parameter wie das Protokoll, Port, Quelle, Ziel und eine Beschreibung [28].

```
resource "opentelekomcloud_networking_secgroup_v2" "secgroup" {
  name = format("%s-secgroup", var.name)
}

resource "opentelekomcloud_networking_secgroup_rule_v2" "secgroup_rule_ssh" {
  direction         = "ingress"
  ethertype         = "IPv4"
  protocol          = "tcp"
  port_range_min    = 22
  port_range_max    = 22
  security_group_id = opentelekomcloud_networking_secgroup_v2.secgroup.id
}
```

Quellcode 5.4: Beispiel einer Security Group in der OTC

5.2.3 Bereitstellung von VMs

Die Webanwendung soll als Docker-Container auf VMs bereitgestellt werden. Virtuelle Maschinen sind als Ressourcentyp auf beiden Cloud-Plattformen verfügbar und bieten diverse Konfigurationsmöglichkeiten. Dieser Ressourcentyp wird verwendet, da er besonders gut abstrahiert werden kann und die Bereitstellung des Prototyps als solchen vereinfacht.

Da für den Prototyp kein großer Ressourcenbedarf anfällt, werden die VMs als kleinstmögliche Varianten bzw. mit ressourcensparenden Tarifen bereitgestellt. Als Betriebssystem wird Canonical Ubuntu 18.04 LTS verwendet.

Die Authentifizierung für eine SSH-Verbindung auf die VMs erfolgt mit einem festgelegten Benutzernamen und einem SSH-Schlüssel. Dieser SSH-Schlüssel wird vor Erzeugung der VM erstellt und in eine lokale Datei ausgegeben, um sich später über SSH auf die Maschine verbinden zu können.

Azure

Die Ressource `azurerm_linux_virtual_machine` kann in Azure ausgiebig konfiguriert werden (siehe Quellcode 5.5). Neben grundlegenden Parametern wie den Namen, der Speicherkapazität oder die Größe der Ressource können ebenfalls Netzwerkkonfigurationen vorgenommen werden.

Die Netzwerkkonfiguration wird in dieser Ressource lediglich zugewiesen und referenziert ein Objekt der Ressource `azurerm_network_interface`, welches zusätzlich erzeugt wird. Über dieses Netzwerkinterface findet die VM Anbindung an das öffentliche und das virtuelle Netzwerk und bekommt jeweils eine IP-Adresse zugewiesen. Neben dem Netzwerk erfolgt ebenfalls die Konfiguration eines Administrationsbenutzers.

```
resource "azurerm_linux_virtual_machine" "vm" {
  name                = var.name
  resource_group_name = var.resource_group_name
  location            = var.location
  size                = var.vm_size
  admin_username      = var.admin_username
  network_interface_ids = [azurerm_network_interface.nic.id]

  admin_ssh_key {
    username   = var.admin_username
    public_key = tls_private_key.keypair.public_key_openssh
  }

  os_disk {
    disk_size_gb         = var.volume_size
  }

  source_image_reference {}
}
```

Quellcode 5.5: Beispiel einer virtuellen Maschine in Azure

OTC

Die Konfiguration der virtuellen Maschine in der OTC erfolgt ähnlich zu der in Azure (siehe Quellcode 5.6). Die Netzwerkkonfiguration erfolgt in diesem Fall auch über eine weitere Ressource `opentelekomcloud_networking_port_v2`. Das SSH-Schlüsselpaar für den Administrationsnutzer kann im Gegensatz zu Azure direkt über eine Ressource in der OTC generiert werden. Hierfür wird der Ressourcentyp `opentelekomcloud_compute_keypair_v2` angeboten.

```
resource "opentelekomcloud_compute_instance_v2" "vm" {
  name       = var.name
  image_name = var.image_name
  flavor_id  = var.vm_size
  key_pair   = opentelekomcloud_compute_keypair_v2.keypair.name

  network {
    port = opentelekomcloud_networking_port_v2.port.id
  }
}
```

Quellcode 5.6: Beispiel einer virtuellen Maschine in der OTC

5.2.4 Initialkonfiguration und Installationen notwendiger Pakete

Wie bereits in der Konzeption erläutert, soll die Initialkonfiguration und die Installation der notwendigen Pakete über das IaC-Tool cloud-init geschehen. Die Beschreibung hierfür erfolgt in einer YAML-Datei.

```
1  package_update: true
2  package_upgrade: true
3
4  groups:
5    - docker
6
7  packages:
8    - curl
9
10 runcmd:
11   - apt-get update
12   - apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-
       plugin
```

Quellcode 5.7: Auszug der cloud-config.yaml

Über die cloud-init-Konfiguration können gewünschte Zustände sowie durchzuführende Kommandozeilenbefehle angegeben werden. In Zeile 4 des Quellcodes 5.7 wird so beispielsweise über das Modul `groups` definiert, sodass nach Durchführung eine Benutzergruppe mit dem Namen Docker existiert und über das Modul `packages`, dass das Paket curl installiert ist (siehe Zeile 8). Mithilfe des Moduls `runcmd` können außerdem individuelle Befehle, wie die Installation von Docker, auf der virtuellen Maschine durchgeführt werden (siehe Zeile 13 ff.).

5.2.5 Konfiguration und Deployment der Webanwendung

Für die Verwaltung und Wartung der Webanwendung durch einen möglicherweise unwissenden (Service-)Mitarbeiter soll das Anpassen, Stoppen und Starten dieser besonders einfach gestaltet werden. Hierfür bietet sich die Verwendung von systemd an. Systemd überwacht automatisch den Zustand von Diensten und stellt sicher, dass diese bei Bedarf automatisch neu gestartet werden. Zudem sind die Services einfach zu konfigurieren. Systemd-Service-Dateien sind einfach zu schreiben und zu verwalten, was die Administration des Systems erleichtert.

Für die Erzeugung der systemd-Konfiguration und der für die Erstellung des Docker-Container notwendigen Dateien soll ein Modul aus der Terraform Registry verwendet und für die Voraussetzungen (siehe Abschnitt 5.1) angepasst werden. Dieses Mo-

dul stammt ursprünglich von Christopher Tippett, einem britischen Cloud DevOps und Data Engineer, und wird über die MIT License auf GitHub veröffentlicht [29]. Die vergebene Lizenz erlaubt die Modifikation des Moduls passend für den Anwendungszweck, sodass eigene Entwicklungen, wie beispielsweise die vorherige Installation von Docker auf der VM, hinzugefügt werden konnten. Innerhalb dieses Moduls werden ein oder mehrere Docker-Container mithilfe einer Docker-Compose-Konfiguration definiert und als systemd-Service angelegt. Die dabei entstehenden Konfigurationsdateien inklusive der dafür notwendigen Kommandozeilenfehle werden über die cloud-init-Konfiguration abgebildet. Als Ausgabe des Moduls erhält man eine fertige cloud-init.yaml zur weiteren Verwendung für das Provisionieren einer virtuellen Maschine.

Gibt es Änderungen an der Docker-Konfiguration, so können beispielsweise das Docker-Image oder die freigegebenen Ports des Containers im Terraform-Quellcode geändert werden. Durch diese Änderungen wird automatisch beim nächsten Ausrollen von Terraform eine aktualisierte cloud-init.yaml generiert, welche dann über eine neu erzeugte VM initialisiert wird. Eine VM ist nur während der Laufzeit des Containers verfügbar und wird auch mit diesem automatisch entfernt oder erstellt. Kleine Konfigurationsänderungen, wie das Ändern von Umgebungsvariablen zur Anwendungskonfiguration, können auch ohne das neu Erstellen der VM innerhalb einer Env-Datei geschehen. Die Änderungen an der Env-Datei erkennt der systemd-Service und startet die Anwendung daraufhin automatisch neu.

5.3 Analyse des Prototyps

Prozessschritte	**Manuell**	**IaC**
Bereitstellung der Netzwerkumgebung	4 min	1 min 40 s
Garantieren der Netzwerksicherheit	5 min	
Bereitstellung von VMs	6 min	
Initialkonfiguration und Installation notwendiger Pakete	3 min	
Konfiguration und Deployment der Webanwendung	6 min	3 min
Durschschnittliche Gesamtdauer	$\bar{x}_1 = 24$ min	$\bar{x}_2 = 4$ min 40 s

Tabelle 5.1: Zeitlicher Vergleich der Prozessschritte zwischen manuellem Aufwand und IaC

Durch die Verwendung von Terraform und cloud-init zur Bereitstellung von Docker-Containern können sowohl viele manuelle Schritte für das Deployment der Weban-

wendung eingespart, als auch Änderungen an dieser als Ganzes automatisch abgebildet werden.

Die in Tabelle 5.1 dargestellte Zeitdauer bezieht sich auf die parallele Bereitstellung zweier Webanwendungen auf Azure und OTC. Der manuelle Aufwand wurde durch wiederholtes Durchführen benötigter Prozessschritte über die Weboberflächen der CSPs und mithilfe von anschließenden Konsolenbefehlen zur Konfiguration getestet und zeitlich erfasst. Die Zeitdauer bei Verwendung des Tools der IaC wurde mithilfe der Bibliothek „time“ während der Ausführung des Tools gemessen. Das Tool cloud-init ist zudem so konfiguriert, den benötigten Zeitaufwand nach Durchführung in eine Log-Datei auszugeben. Die Dauer beider automatisierten Schritte wurde mehrfach durchgeführt und zeitlich erfasst. Die Zeit, welche das Tool benötigt hängt u.a. von der verfügbaren Rechenleistung des ausführenden Clients, sowie der Erreichbarkeit und Performanz der APIs der CSPs ab. Für beide Ergebnisse wurden die arithmetischen Mittel $\bar{x}_1$ und $\bar{x}_2$ gebildet.

Man erkennt anhand der benötigten Zeiten für die zwei verschiedenen Wege der Bereitstellung der Anwendung eindeutig, dass ein automatisierter Ansatz nicht nur effizienter, sondern auch aufgrund der Vorteile der IaC weniger fehleranfälliger ist als der manuelle Prozess. Besonders eindeutig zeigt sich die Differenz bei der Bereitstellung von mehreren Containern auf VMs, da Terraform auch dies parallel abwickeln kann. Die Umsetzung mit IaC bietet in dem dargestellten Beispiel eine Zeitersparnis von

$$\frac{\bar{x}_2 - \bar{x}_1}{\bar{x}_1} \cdot 100 = \frac{280s - 1.440s}{1.440s} \cdot 100 = -80,\overline{5} \approx -81\% \tag{5.1}$$

Weiterhin ist mit der Amortisation zu berechnen, ab wann sich die Entwicklung eines Prototyps rentiert hat. Die Entwicklungdauer des Prototyps betrug 4,5 Stunden ($= 16.200s$).

$$\left\lceil \frac{16.200s}{\bar{x}_1 - \bar{x}_2} \right\rceil = 14 \tag{5.2}$$

Bereits bei der Bereitstellung einer 14. virtuellen Maschine oder dem wiederholten Deployment auf einer vorhandenen virtuellen Maschine amortisiert sich die Verwendung des entwickelten Prototyps.

Durch die einfache Beschreibung und die Verwendung von Docker-Images fallen alternative Optionen, wie beispielsweise das Bereitstellen von Anwendungsdatei-

en über FTP, im Deployment weg. Docker stellt aufgrund der Versionierung der verwendeten Images eine unwandelbare Infrastruktur bereit, sodass keine Konfigurationsänderungen außerhalb von Terraform geschehen können, da die Anwendung und dessen Konfiguration bereits in dem jeweiligen Image verpackt sind. Auch wenn das hier dargestellte Beispiel einer Webanwendung einfach aufgebaut ist, können mittels Docker und der vorgestellten Konfiguration ebenfalls beliebig komplexe Projekte realisiert werden, sodass ein Einsatz in Produktivumgebungen auch möglich wäre.

Zusätzlich erfüllt Docker die Kriterien der Sicherheit, indem die Container sowie die VMs, auf welchen diese laufen, jeweils von anderen Systemen isoliert sind. Datenverkehr ist lediglich mit der passenden Konfiguration der jeweiligen Firewalls über Terraform möglich. Eine VM betreibt immer nur einen Container und ist nur mit diesem koexistent, sodass kein unbenötigter Ressourcenverbrauch und somit keine folgenden Kosten verursacht werden.

Die laufenden Container und die Webanwendung werden automatisch über den systemd-Service überwacht und sind somit einfach verwaltbar.

Der vollständige Quellcode des Prototyps steht unter dem Link https://gitlab.dit.htwk-leipzig.de/peter.prumbach/bachelor-prototyp mit der neuesten Commit-ID 20bfa6ee633ebd326ef1dfa4b406e650c3e74898 zur Verfügung.

6 Fazit

6.1 Ausblick

Es zeigt sich durch die Anzahl an bisher verfügbaren Möglichkeiten für die Verwendung von IaC-Tools und die stetigen Weiterentwicklungen, dass die Automatisierung der Infrastrukturbereitstellung ein immer populärerer Ansatz werden wird. Bereits heute bieten vorhandene Tools diverse Möglichkeiten für die Bereitstellung aller verfügbaren Ressourcen einer Vielzahl an CSPs. Das einzige Problem liegt heutzutage an der Diversität der CSPs und deren Kompatibilität der Ressourcentypen untereinander.

Projekte wie das vorgestellte OAM arbeiten aufgrund dessen daran, die Infrastruktur auf Cloud-Lösungen mittels Abstraktion über Kubernetes und weitere Cloud-Native-Technologien zu generalisieren und somit die Umsetzung eines Cloud-agnostischen Ansatzes in Zukunft zu vereinfachen. Falls die Kompatibilität des OAM dann die Bereitstellung gleicher Webanwendungen auf unterschiedliche Cloud-Lösungen realisieren kann, kann der vorgestellte Prototyp ebenfalls nochmal in seiner Komplexität gesenkt werden.

Auch die Integration künstlicher Intelligenz und maschinellen Lernens innerhalb des Prozesses der Infrastrukturbereitstellung könnte zukünftig in der Lage sein, auftretende Probleme bei der Bereitstellung zu erkennen und teilweise Vorhersagen zu tätigen sowie Problemlösungsstrategien vorzuschlagen.

6.2 Zusammenfassung

Das Ziel der Arbeit war es, eine Lösung für das Problem zu erarbeiten, eine bestehende Webanwendung mittels eines entwicklerischen Ansatz und Werkzeuge der Kategorie IaC auf mehreren CSP bereitstellen zu können. Die dafür notwendigen Schritte sollten allesamt in einer Automatisierung abgebildet werden.

Untersucht wurden hierbei die einzelnen Prozessschritte der Infrastrukturbereitstellung und der Vergleich dieser untereinander. Hierfür wurden die notwendigen Grundlagen rund um die Begrifflichkeiten der IaC und dessen Vorteile gegenüber eines manuellen Prozesses erläutert. Zudem wurden verschiedenste Tools der Kategorie IaC vorgestellt und anschließend miteinander verglichen, um so eine Auswahl für die Umsetzung des abschließenden Prototyps zu treffen.

Bei der Auswertung wurde festgestellt, dass sich zwei Tools besonders gut für die Umsetzung der Problemstellung eignen. Es wurde sich auf Grundlage der vorgestellten Tools für Terraform und Pulumi zu einem detaillierten Vergleich dieser Beiden entschieden. Außerdem waren für die abschließende Entscheidung zur Umsetzung des Prototyps auch allgemeine Kriterien von hoher Bedeutung, die unabhängig der Auswahl des Tools getroffen wurde. Durch die Implementierung des Prototyps in Verwendung von Terraform und cloud-init wird eine effizientere und fehlerfreie Ausführung der Prozesse der Infrastrukturbereitstellung ermöglicht.

In der Arbeit wurde keine Allgemeingültigkeit der vorgestellten Lösung untersucht. Die Auswahl der Tools kann je nach individuellen Präferenzen sowie betrieblicher Anforderungen und deren Anwendungszweck geringfügig abweichen. Zusätzlich wurde der Prototyp lediglich innerhalb der CSP Azure und OTC entwickelt und dafür ausgelegt. Aufgrund der Verwendung von Terraform als Cloud-agnostisches Tool und der breiten Unterstützung für cloud-init kann der entwickelte Ansatz auch mit geringfügigem Aufwand auf andere CSPs repliziert werden.

Abschließend bleibt festzuhalten, dass durch die Verwendung von IaC für die Automatisierung ein enormes Einsparpotenzial durch den Wegfall eines manuellen Prozesses geschaffen wurde. Dies senkt nicht nur die OpEx, sondern steigert im Gegenzug auch die Produktivität des ausführenden Personals.

Literaturverzeichnis

[1] Y. Brikman, *Terraform: Up and Running*. O'Reilly Media, 2022, ISBN: 9781098116705. Adresse: https://books.google.de/books?id=dFaKEAAAQBAJ.

[2] D. Petcu, G. Macariu, S. Panica und C. Crăciun, „Portable Cloud applications—From theory to practice," *Future Generation Computer Systems*, Jg. 29, Nr. 6, S. 1417–1430, 2013, Including Special sections: High Performance Computing in the Cloud & Resource Discovery Mechanisms for P2P Systems, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2012.01.009. Adresse: https://www.sciencedirect.com/science/article/pii/S0167739X12000210.

[3] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero und D. A. Tamburri, „DevOps: Introducing Infrastructure-as-Code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, S. 497–498. DOI: 10.1109/ICSE-C.2017.162.

[4] V. Shvetcova, O. Borisenko und M. Polischuk, „Domain-Specific Language for Infrastructure as Code," in *2019 Ivannikov Memorial Workshop (IVMEM)*, 2019, S. 39–45. DOI: 10.1109/IVMEM.2019.00012.

[5] Canalys, *Cloud infrastructure services vendor market share worldwide from 4th quarter 2017 to 1st quarter 2022*, [Online; Stand 24. Februar 2023], 2022. Adresse: https://www.statista.com/statistics/967365/worldwide-cloud-infrastructure-services-market-share-vendor/.

[6] Scribd, *Current enterprise public cloud adoption worldwide from 2017 to 2022, by service*, [Online; Stand 24. Februar 2023], 2022. Adresse: https://www.statista.com/statistics/511508/worldwide-survey-public-coud-services-running-applications-enterprises/.

[7] Microsoft, *Top reasons for hybrid or mutlicloud adoption worldwide in 2021*, [Online; Stand 24. Februar 2023], 2022. Adresse: https://www.statista.com/statistics/1332011/reasons-for-hybrid-cloud-adoption-worldwide/.

[8] C. Tozzi, *5 Multi-Cloud Deployment Challenges (and How to Solve Them)*, [Online; Stand 24. Februar 2023], 2020. Adresse: https://www.f5.com/company/blog/5-multi-cloud-deployment-challenges-and-how-to-solve-them.

[9] P. Sangode, *Understanding terms - Infrastructure As Code, Orchestration, Provisioning & Configuration Management (Ansible & Terraform, as example)*, [Online; Stand 24. Februar 2023], 2021. Adresse: https://www.linkedin.com/pulse/understanding-terms-infrastructure-code-management-ansible-sangode/.

[10] *Was ist Orchestrierung?* [Online; Stand 24. Februar 2023], 2019. Adresse: https://www.redhat.com/de/topics/automation/what-is-orchestration.

[11] K. Morris, *Infrastructure as Code*. O'Reilly Media, 2020, ISBN: 9781098114640. Adresse: https://books.google.de/books?id=UW4NEAAAQBAJ.

[12] M. Fowler und R. Parsons, *Domain-specific Languages*, Ser. Addison-Wesley signature series. Addison-Wesley, 2011, ISBN: 9780321712943. Adresse: https://books.google.de/books?id=7lLHmAEACAAJ.

[13] M. Howard, *Terraform – Automating Infrastructure as a Service*, [Online; Stand 24. Februar 2023], 2022. DOI: 10.48550/ARXIV.2205.10676. Adresse: https://arxiv.org/abs/2205.10676.

[14] Terraform, *Terrafom - Documentation*, [Online; Stand 24. Februar 2023], 2022. Adresse: https://developer.hashicorp.com/terraform.

[15] Pulumi, *Pulumi - Documentation*, [Online; Stand 24. Februar 2023], 2022. Adresse: https://www.pulumi.com/docs/.

[16] L. Hochstein und R. Moser, *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media, 2017, ISBN: 9781491979778. Adresse: https://books.google.de/books?id=h5YtDwAAQBAJ.

[17] J. Loope, *Managing Infrastructure with Puppet: Configuration Management at Scale*. O'Reilly Media, 2011, ISBN: 9781449313227. Adresse: https://books.google.de/books?id=hYb2U-ZZByMC.

[18] M. Marschall, *Chef Infrastructure Automation Cookbook - Second Edition*, Ser. Quick answers to common problems. Packt Publishing, 2015, ISBN: 9781785289019. Adresse: https://books.google.de/books?id=gp-9CQAAQBAJ.

[19] R. Blum und C. Bresnahan, *CompTIA Linux+ Study Guide: Exam XK0-005*. Wiley, 2022, ISBN: 9781119878964. Adresse: https://books.google.de/books?id=waZ5EAAAQBAJ.

[20] Microsoft, *Announcing the Open Application Model (OAM), an open standard for developing and operating applications on Kubernetes and other platforms*, [Online; Stand 24. Februar 2023], 2019. Adresse: https://cloudblogs.microsoft.com/opensource/2019/10/16/announcing-open-application-model/.

[21] A. Özel, T. Pautz und N. Schmidt, „Infrastructure as Code als Maßnahme zur Cloud Automatisierung – Hilfestellung zur Auswahl des richtigen Werkzeugs," *HMD Praxis der Wirtschaftsinformatik*, Jg. 57, Nr. 5, S. 936–948, 2020. DOI: 10.1365/s40702-020-00657-0.

[22] S. Strutt, *Infrastructure as Code: Chef, Ansible, Puppet, or Terraform?* [Online; Stand 24. Februar 2023], 2018. Adresse: https://www.ibm.com/cloud/blog/chef-ansible-puppet-terraform.

[23] D. Sokolowski, „Infrastructure as Code for Dynamic Deployments," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Singapore, Singapore: Association for Computing Machinery, 2022, S. 1775–1779, ISBN: 9781450394130. DOI: 10.1145/3540250.3558912. Adresse: https://doi.org/10.1145/3540250.3558912.

[24] „2022 Accelerate State of DevOps Report," DevOps Research & Asessment, Google, Techn. Ber., 2022.

[25] A. Lingayat, R. R. Badre und A. Kumar Gupta, „Performance Evaluation for Deploying Docker Containers On Baremetal and Virtual Machine," in *2018 3rd International Conference on Communication and Electronics Systems (ICCES)*, 2018, S. 1019–1023. DOI: 10.1109/CESYS.2018.8723998.

[26] *Azure Virtual Network*, [Online; Stand 24. Februar 2023]. Adresse: https://learn.microsoft.com/en-us/azure/virtual-network/virtual-networks-overview.

[27] *Azure Network security groups*, [Online; Stand 24. Februar 2023]. Adresse: https://learn.microsoft.com/en-us/azure/virtual-network/network-security-groups-overview.

[28] *Virtual Private Cloud - User Guide*, English, Version 2022-06-25, Open Telekom Cloud, 25. Juni 2022, 288 S.

[29] C. Tippett, *Deploy an application to any Cloud VM with Terraform, Docker & cloud-init*, [Online; Stand 24. Februar 2023], 2022. Adresse: https://github.com/christippett/terraform-cloudinit-container-server.

[30] *OPEX - Gabler Wirtschaftslexikon*, [Online; Stand 24. Februar 2023], 2021. Adresse: https://wirtschaftslexikon.gabler.de/definition/opex-52701/version-383552.

Glossar

Infrastruktur IT-Infrastruktur beschreibt eine Kombination von Hardware, Software und Netzwerkkomponenten um Anwendungen zu entwickeln, bereitzustellen und warten zu können.

OpEx Abk. für engl. operational expenditures. Beschreibt die laufenden Ausgaben für einen funktionierenden operativen Geschäftsbetrieb. Hierzu gehören unter anderem die Kosten für Rohstoffe, Betriebskosten, Personal, Leasing oder Energie [30].

Provider Der Begriff Provider, welcher in Tools zur Verwaltung der Infrastruktur genutzt wird beschreibt ein Plugin, welches dazu verwendet wird, um mit verschiedenen Cloud-Providern, SaaS-Anbietern oder APIs zu kommunizieren. Ein Provider bildet so eine Schnittstelle und die Übersetzung von Quellcode in API-Aufrufe der Anbieter.

Ressourcen Assets innerhalb von Cloud-Umgebungen, wie beispielsweise Server, Rechenleistung, Speicherplatz oder Software. Die Abrechnung dieser Ressourcen erfolgt meistens nach Bedarf und Verbrauch.

State Bei dem State handelt es sich um eine strukturierte Datei, welche alle verwaltetenden Ressourcen beinhaltet. Neben einer Auflistung der Ressourcen enthält diese Datei zusätzlich Informationen darüber, wie die Ressourcen konfiguriert sind.