

Bachelorarbeit  
**Cloud Service Provider Evaluierung  
auf Basis von  
Infrastructure as Code Unterstützung**

im Studiengang Softwaretechnik und Medieninformatik  
der Fakultät Informationstechnik  
Wintersemester 2021/2022

Julian Schallenmüller

**Zeitraum:** 15.10.2021 - 15.01.2022

**Prüfer:** Prof. Dr.-Ing. Kai Warendorf

**Zweitprüfer:** Prof. Dr. rer. nat. Mirko Sonntag

---

**Firma:** Noavtec Consulting GmbH

**Betreuer:** Dipl.-Ing. (BA) Matthias Häussler



# Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 21. Januar 2022 \_\_\_\_\_  
Unterschrift

# Sperrvermerk

Die nachfolgende Bachelorarbeit enthält vertrauliche Daten der Noavtec Consulting GmbH. Veröffentlichungen oder Vervielfältigungen dieser Arbeit – auch nur auszugsweise – sind ohne ausdrückliche Genehmigung der Noavtec Consulting GmbH nicht gestattet. Diese Arbeit ist nur den Prüfern sowie den Mitgliedern des Prüfungsausschusses zugänglich zu machen.

# Zitat

*„Showing a strong success and visible benefits is key to getting others to agree to try your way of doing things.”“*

Frederic Rivain

# Vorwort

Dank an die Firma und die Firmenmitarbeiter, max. 1/2 Seite

# Kurz-Zusammenfassung

„Aushängeschild“ der Arbeit, max 1 Seite

# Inhaltsverzeichnis

1	Einleitung	1
1.1	Einleitung . . . . .	1
2	Grundlagen	3
2.1	Funktionsprinzip, Vorteile und Herausforderungen des modernen Cloud Computings . . . . .	3
2.1.1	Definition und Funktionsweise . . . . .	3
2.1.2	Zu erwägende Vor- und Nachteile des Einsatzes von Cloud Computing	6
2.1.3	<b>Grundlegende Erklärung von DevOps</b> . . . . .	8
2.1.4	Überblick über die wichtigsten Public Cloud Service Provider . . . . .	9
2.2	Infrastructure as Code . . . . .	12
2.2.1	Technologischer Wandel und das Cloud Age Mindset . . . . .	12
2.2.2	Vorteile von IaC und durch IaC gelöste Probleme des manuellen Provisionings . . . . .	13
2.2.3	Herausforderungen und Argumente gegen den Einsatz von IaC . . . . .	14
2.2.4	Die drei Kernverfahren von IaC . . . . .	17
2.2.5	Die Rolle von Terraform im IaC Anwendungsprozess . . . . .	17
2.2.6	Terraform Funktionsprinzip . . . . .	21
3	Aufbau und Untersuchung	26
3.1	Evaluierungsanforderungen . . . . .	26
3.1.1	Ziel der Evaluierung . . . . .	26
3.1.2	Zu untersuchende Komponenten der Terraform Provider . . . . .	26
3.1.3	Grenzen und Limitierungen der Evaluierung . . . . .	27
3.2	Spezifizierung des Vergleichs . . . . .	27
3.2.1	Auswahl der zu untersuchenden Aspekte und Eigenschaften . . . . .	27
3.2.2	Entscheidungskriterien für die Evaluierung . . . . .	33
3.3	Umsetzung . . . . .	34
3.3.1	Eingesetzte Software und Tools . . . . .	34
3.3.2	High-level Aufbau der Infrastruktur des Versuchsobjekts . . . . .	34
3.3.3	Konkreter Aufbau der Versuchssysteme . . . . .	35
3.3.4	Durchzuführende Messungen und Analysen . . . . .	38
4	Ergebnisse und Bewertung	41
4.1	Evaluierung der Functional completeness . . . . .	41
4.2	Time behaviour Tests . . . . .	43

---

4.3	Recoverability Tests . . . . .	45
4.4	Modifiability Tests . . . . .	46
5	Schluss . . . . .	47
5.1	Zusammenfassung der Arbeit . . . . .	47
5.2	Fazit . . . . .	48
5.3	Weitere untersuchenswerte Aspekte und aktuelle Entwicklungen . . . . .	48
A	Kapitel im Anhang . . . . .	50
	Literaturverzeichnis . . . . .	51



# Abbildungsverzeichnis

2.1	Die grundlegenden Cloud Service Modelle im Überblick . . . . .	5
2.2	CSP Market Share Q2 2020 nach Umsatz . . . . .	10
2.3	CSP Gartner Magic Quadrant August 2020 . . . . .	11
2.4	Iron Age vs Cloud Age . . . . .	12
2.5	Iron Age vs Cloud Age IaC Busch S.6 . . . . .	16
2.6	Überblick IaC Tools <a href="https://medium.com/cloudnativeinfra/when-to-use-which-infrastructure-as-code-tool-665af289fbde">https://medium.com/cloudnativeinfra/when-to-use-which-infrastructure-as-code-tool-665af289fbde</a> . . . . .	19
2.7	Terraform Funktionsprinzip . . . . .	22
3.1	Überblick ISO 25010 . . . . .	28
3.2	Infrastructure Showcase High-level Struktur . . . . .	35
3.3	Beispiel Terraform Graph . . . . .	36
3.4	Testsystem in Google Cloud Platform . . . . .	37

# Tabellenverzeichnis

4.1	Functional Completeness von GCP und Azure . . . . .	41
4.2	Time behaviour von GCP und Azure in verschiedenen Szenarien . . . . .	43
4.3	Recoverability von GCP und Azure in Bezug auf individuelle Ressourcen .	45
4.4	Modifiability von GCP und Azure in Bezug auf individuelle Ressourcen . .	46

# 1 Einleitung

## 1.1 Einleitung

Eines der wichtigsten Schlagworte im Zeitalter der fortschreitenden Digitalisierung ist der Begriff des Cloud Computings. Auch in Bereichen der Industrie wie der Finanz- und Versicherungsbranche die sich zu großen Teilen aufgrund von Sicherheits- und anderen Bedenken lange Zeit gegen die Nutzung Cloud-basierter Systeme entschieden hatte gewinnt das Thema mehr und mehr Relevanz. Die Nutzung von Cloud-Technologien verspricht die Möglichkeit schneller auf Anforderungen von Kunden reagieren zu können, kostengünstige und flexible Skalierung der eigenen Rechenkapazitäten, Einsparungen durch den Wegfall eigener IT-Infrastruktur Fachleute und mehr.

Gemeinsam mit der Eröffnung neuer Möglichkeiten bringt die Einführung neuer Technologie auch immer eine Reihe eigener Herausforderungen mit sich. Für eine erfolgreiche und gewinnbringende Einführung dieser ist es essentiell diese zu verstehen und die passende Denkweisen und Werkzeuge mit denen die aufkommenden Probleme gelöst werden können entsprechend einzusetzen.

Das Thema mit dem sich diese Arbeit befassen wird ist das automatisierte Managen und Bereitstellen von Cloud Computing Ressourcen, zusammengefasst unter dem Begriff Infrastructure as Code (IaC). Die Grundlagenkapitel werden zu diesem Zweck auf den technischen Kontext und die Relevanz von Cloud Computing, IaC und dem Software Tool Terraform eingehen. Es wird erläutert werden an welcher Stelle die ausgewählten Plattformen und Software zum Einsatz kommen, welche Probleme dadurch gelöst werden, wo deren Vorteile und Grenzen liegen sowie welche Alternativen existieren und wo ergänzende Werkzeuge zum Einsatz kommen können.

Den Kern der Arbeit bildet ein Vergleich der ausgewählten Cloud Service Provider auf Basis der Unterstützung Von IaC mit Terraform. Hierfür wird die Infrastruktur einer Schulung der Firma Novatec Consulting GmbH in der die wichtigsten grundlegenden Cloud Computing Ressourcen Verwendung finden auf verschiedenen Plattformen mit Terraform

bereitgestellt und einem Vergleich unterzogen. Als Vergleichsmetriken werden Kriterien der Norm ISO/IEC 25000 herangezogen um eine fachgerechte und neutrale Bewertung zu gewährleisten. Dabei soll auch in Betracht gezogen werden wie hoch der Aufwand für die Migration eines bestehenden Systems zu IaC ausfällt. Nach der Auswertung der Vergleichsergebnisse werden die gewonnenen Erkenntnisse zusammengefasst, weitere untersuchenswerte Aspekte aufgeführt und ein Ausblick auf aktuelle und zukünftige Entwicklungen gegeben. Durch die sehr aktuelle Relevanz des Themas werden sich mit hoher Wahrscheinlichkeit auch in Zukunft neue Software und Technologien durchsetzen, alte verdrängt und neue Lösungsansätze für bestehende Herausforderungen und Probleme etabliert werden. Diese Arbeit wird jedoch nur die aktuell relevantesten Plattformen und Tools betrachten um eine Entscheidungsbasis für deren Einsatz zum aktuellen Zeitpunkt bieten zu können.

## 2 Grundlagen

### 2.1 Funktionsprinzip, Vorteile und Herausforderungen des modernen Cloud Computings

#### 2.1.1 Definition und Funktionsweise

Das National Institute of Standards and Technology (NIST) der Vereinigten Staaten von Amerika definiert Cloud Computing im Abstract der NIST SP-800-145 folgendermaßen:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

Cloud Computing beschreibt ein Modell das es ermöglicht ortsunabhängig, zweckdienlich und zeitunabhängig auf einen konfigurierbaren Pool an Computerressourcen (Netzwerke, Server, Datenspeicher, Anwendungen und Services) zuzugreifen die schnell und mit minimalem Aufwand und minimaler notwendiger Interaktion bereitgestellt und wieder abgebaut werden können. Dieses Cloud Modell beschreibt fünf essentielle Charakteristiken, drei Servicemodelle und vier Bereitstellungsmodelle.

Weiter definiert das Dokument die fünf Charakteristiken in den folgenden Punkten:

**On-demand-self-service:** Der Nutzer kann eigenmächtig die benötigten Ressourcen automatisch bereitstellen, es wird keine menschliche Interaktion benötigt.

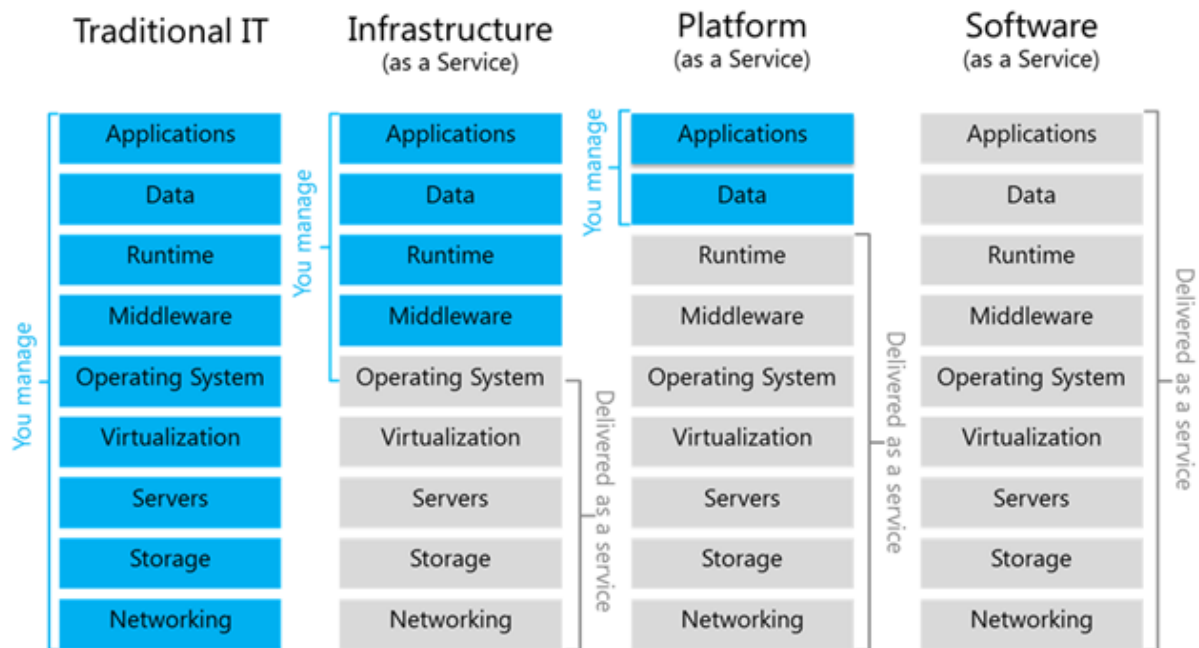
**Broad network access:** Auf Leistungen wird über das normale Internet mit standard Mechanismen die die Nutzung von Thin Clients und Fat Clients (Smartphones, Tablets, Laptops oder Workstations) fördern zugegriffen.

**Resource Pooling::** Die Computerressourcen des Anbieters werden in einem gemeinsamen Pool für mehrere Kunden in einem mandantentauglichen Modell bereitgestellt, physische und virtuelle Ressourcen werden nach dynamisch zugewiesen und entsprechend der Nachfrage neu verteilt. Es wird eine empfundene Ortsunabhängigkeit hergestellt indem der Nutzer kein genaues Wissen darüber besitzt wo sich dessen Resources befinden, auf höherem Level wie beispielsweise dem Staat, der Region oder auch Rechenzentrum kann der Ort vom Nutzer spezifiziert werden. Die bereitgestellten Ressourcen beinhalten zum Beispiel Datenspeicher, Rechenleistung, Rechenspeicher und Netzwerkbandbreite.

**Rapid Elasticity:** Rechenkapazitäten werden dehnbar bereitgestellt und abgebaut, teilweise automatisch, um entsprechend der Nachfrage schnell hoch- und wieder zurück skalieren zu können, Rechenkapazitäten erscheinen dadurch unbegrenzt und zu jeder Zeit in jedem Umfang bereitgestellt werden.

**Measured Service:** Cloud systeme kontrollieren und optimieren Ressourcennutzung automatisch mithilfe eines Messungs-systems dass auf einer abstrakten Ebene den entsprechenden Service (Datenspeicher, Rechenleistung, Benutzerkonten) überwacht, kontrolliert und Bericht erstattet um sowohl für Anbieter als auch Kunden Transparenz herzustellen.

Es wird zwischen drei grundlegende Cloud Service Modellen unterschieden: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) und Software as a Service (SaaS).



**Abb. 2.1:** Die grundlegenden Cloud Service Modelle im Überblick

**Infrastructure as a Service:** Der Nutzer hat die Fähigkeit Rechenleistung, Datenspeicher Netzwerke und weitere fundamentale Computerressourcen bereitzustellen und beliebige Software darauf zu betreiben, dazu können Betriebssysteme und Anwendungen gehören. Die darunterliegende Infrastruktur wird vom Anbieter betrieben, der Nutzer kann aber eingeschränkte Kontrolle über bestimmte Komponenten haben, dazu gehören beispielsweise Firewalls.

**Platform as a Service:** Der Nutzer verfügt über die Fähigkeit seine eingekaufte oder selbst erstellten Anwendungen auf der Cloud Infrastruktur zu betreiben, die notwendige Umgebung die über Sprachen, Bibliotheken, Tools und Services verfügt wird vom Anbieter bereitgestellt. Die darunter liegende Infrastruktur mit Netzwerken, Servern, Betriebssystemen und Speicher wird vom Anbieter betrieben, der Nutzer hat die Kontrolle über die Anwendung und Konfiguration der Umgebung in der die Anwendung betrieben wird.

**Software as a Service:** Dem Nutzer wird der Zugriff auf die vom Anbieter in der Cloud Infrastruktur betriebenen Softwareanwendung gewährt. Auf diese wird mithilfe eines Thin oder Fat Clients zugegriffen, dabei kümmert sich der Nutzer nicht um den Betrieb und die Konfiguration der darunterliegenden Cloud Infrastruktur wie Netzwerke, Server, Betriebssystem, Speicher und die Anwendung selbst mit Ausnahme eingeschränkter Nutzereinstellungen.

Die von NIST unterschiedenen grundlegenden Service Modellen können noch weiter differenziert werden. Ein Beispiel hierfür ist das Modell Function as a Service (FaaS) als Subset von PaaS. FaaS erlaubt es Programmcode auszuführen ohne sich um die Bereitstellung weiterer Infrastruktur kümmern zu müssen wie es der Betrieb eines gewöhnlichen Microservices verlangt.

In Art der Bereitstellung eines Cloud Services werden vier grundlegende Modelle unterschieden; Public, Private, Hybrid und Community Cloud Modelle.

**Private Cloud:** Die Cloud Infrastruktur wird ausschließlich für die Nutzung durch eine einzige Organisation mit mehreren Nutzern bereitgestellt. Besitz und Betrieb liegen dabei entweder bei der selben Organisation, einer Drittpartei oder einer Kombination beider, die Infrastruktur kann dabei On- oder Off Premise betrieben werden.

**Public Cloud:** Die Public Cloud steht für die Nutzung durch die allgemeine Öffentlichkeit bereit. Die Cloud Infrastruktur befindet sich im Besitz eines Unternehmens, Bildungseinrichtung, Regierungsorganisation oder einer Kombination aus diesen und wird auch von der selben Organisation On Premise betrieben.

**Community Cloud:** Eine Community Cloud wird von einer Gemeinschaft von Nutzern mit gemeinsamen Anliegen eingesetzt. Der Besitz und Betrieb liegen dabei bei einem oder mehreren Mitgliedern dieser Gemeinschaft, einer Drittpartei und kann Off oder On Premise betrieben werden.

**Hybrid Cloud:** Die Hybrid Cloud besteht aus einer Kombination der beschriebenen Modelle (Public, Private und Community). Diese bilden dabei eigene Instanzen die aber durch standardisierte oder proprietäre Schnittstellen den Transfer von Daten und Anwendungen zwischen den Instanzen erlauben.

### 2.1.2 Zu erwägende Vor- und Nachteile des Einsatzes von Cloud Computing

#### **Vorteile und Treiber der Adoption von Cloud Computing**

**Wirtschaftliche Vorteile:** Ein Vorteil in der Nutzung von Cloud Computing kann darin liegen dass ein Großteil der für den Betrieb notwendigen Infrastruktur nicht mehr vom Unternehmen selbst eingekauft, eingerichtet und betrieben werden muss (vgl. linke Seite Abb 2.1). Potentiell hohe Kosten die bereits vor der Inbetriebnahme eines Systems mit



einem höheren Risiko aufgewendet werden mussten stellen in Form von individuell geringeren laufenden Beträgen ein deutlich reduziertes Risiko da.

Sofern der Einsatz von Cloud Computing in einer sinnvollen und korrekten Weise erfolgt können je nach Fall die Gesamtkosten um einen hohen Anteil reduziert werden.

Die Gesamtkostenersparnisse stehen auch im Zusammenhang mit Skaleneffekten die für große Cloudanbieter gelten. Der Betrieb eines einzelnen Servers ist im Verhältnis mit bedeutend höheren Kosten verbunden als das hinzufügen eines äquivalenten System zu einem Rechenzentrum im Betrieb von AWS oder einem vergleichbaren Anbieter.

**Skalierbarkeit:** Besonders für schnell wachsende Unternehmen ist die Option der schnellen Skalierbarkeit einer der prominentesten Vorteile der Cloud. Es kann nicht nur auf vorhersagbare Anstiege (zum Beispiel ausgelöst durch eine Verkaufsaktion) sondern auch auf unvorhersehbare Ereignisse reagiert werden. Zusätzlich ist es möglich diese Skalierung nicht nur bis zu einem bestimmten Limit, sondern nahezu unendlich zu betreiben. Wichtig ist auch dass sowohl auf steigende als auch sinkende Nachfrage reagiert werden kann.

**Resilienz:** In einem worst case Szenario kann ein ganzes Rechenzentrum durch unvorhergesehen Ereignisse wie beispielsweise Brände, Naturkatastrophen oder anderes vollständig zerstört werden. Selbst wenn Backup-Rechenzentren verfügbar sind ist eine Übertragung der Operationen kein trivialer Ablauf und birgt oft nicht außer Acht zu lassende Risiken. Die Flexibilität der Cloud erlaubt es die gesamte Infrastruktur mit sehr geringem Aufwand in nicht betroffene Regionen zu verlagern und die Kontinuität der Geschäftstätigkeiten mit minimaler Unterbrechung aufrecht zu erhalten.

**Security:** Security Aspekte können sowohl einen Vor- als auch Nachteil von Cloud Computings darstellen. Hier sollen zuerst Vorteile dargelegt werden, potentielle Probleme werden im nächsten Unterkapitel beschrieben. Die technischen Möglichkeiten und besonders auch die Wahrnehmung des Themas Sicherheit in der Cloud unterlagen und unterliegen auch immernoch einem deutlichen Wandel. Cloud Anbieter investieren viele Ressourcen in Sicherheit und stellen dem Nutzer zum Beispiel bereits sicher implementierte Verschlüsselungen zur Verfügung oder bieten einen gewissen Schutz vor Denial-of-Service Angriffen durch die natürliche Skalierbarkeit.

## Nachteile und Risiken

**Netzwerkabhängigkeit:** Da der Zugriff auf Cloud Dienste über das Internet erfolgt entsteht dadurch entsprechend auch eine hohe Abhängigkeit. Eine stabile und schnelle Netz-

verknüpfung ist Voraussetzung, dass effektiv gearbeitet werden kann, bei lokal gehosteten Systemen ist diese Abhängigkeit entsprechend geringer.

**Vendor Lock-in:** Bei der Nutzung eines Cloud-Anbieters entsteht die Gefahr, sich zu sehr in Abhängigkeit eines einzelnen Anbieters zu begeben. Im Fall einer Änderung der Nutzungsbedingungen oder einer Änderung im Bezahlmodell, die den eigenen Interessen stark entgegensteht, besteht die Gefahr, bereits so abhängig von diesem Anbieter zu sein, dass die Kosten eines Wechsels derart hoch ausfallen, dass man gezwungen ist, die Bedingungen zu akzeptieren.

**Security und Privacy:** Sicherheitsrisiken sind einer der meistgenannten Gründe, die gegen Cloud Computing sprechen, besonders im Fall der Nutzung einer Public Cloud. Die Gefahr, dass Daten in die Hände Dritter gelangen, kann zum Beispiel nicht vollständig ausgeschlossen werden. Da die Verantwortung über die Sicherheit der Daten dem Cloud-Anbieter unterliegt, kann es auch zu Problemen in Hinsicht der Privatsphäre der Daten kommen, sollte eine Regierungsorganisation Zugriff auf bestimmte Daten eines Nutzers verlangen könnten, diese ohne dessen Einverständnis gewährt werden.

**Kosten:** Auch wenn die Nutzung von Cloud Computing mit dem Vorteil geringerer Kosten beworben ist, dies nicht zwingend garantiert. Werden die vorhandenen Systeme ungünstig verwendet, zum Beispiel bleiben viele gebuchte Ressourcen ungenutzt und IP-Adressen unverwendet, können hohe Kosten entstehen, auch in der Phase des Übergangs zu Cloud Computing können höhere Kosten entstehen als in einem vergleichbaren Zeitraum davor.

### 2.1.3 Grundlegende Erklärung von DevOps

Die exakte Definition und Abgrenzung des Begriffes DevOps ist ein Thema, über das es keine uniform akzeptierte, allein gültige Definition und Abgrenzung gibt. Im allgemeinen bezeichnet DevOps eine stärkere Vereinigung der Entwicklungs- (Dev) und Betriebs- (Ops) Teams eines Softwareprojekts durch die Anwendung einer effektiveren Arbeitskultur und -philosophie mit neuen Methoden, Werkzeugen und Prozessen. Verschiedene Organisationen legen hierbei in ihrer Definition die Schwerpunkte auf unterschiedliche Aspekte. Als Beispiel betont Amazon hierbei besonders die schnelle Auslieferung neuer Produkte ("ability to deliver applications and service at high velocity"), Microsoft die Kollaboration zwischen den beteiligten Teams ("DevOps enables formerly siloed roles - development, IT operations, quality engineering, and security - to coordinate and collaborate to produce better, more reliable products.").

Das DevOps Research and Assessment (DORA) Team hat über sieben Jahre mit 32.000

Beteiligten die Praktiken und Fähigkeiten untersucht die hohe Leistungsfähigkeit bei Entwicklung und Auslieferung von Software fördern. Eine Übersicht über die Erkenntnisse dieser Studie ist als Diagramm im Anhang ... zu finden.

Infrastructure as Code spielt als Werkzeug für die automatisierte Bereitstellung der benötigten Computerressourcen eine wichtige Rolle im DevOps Prozess.

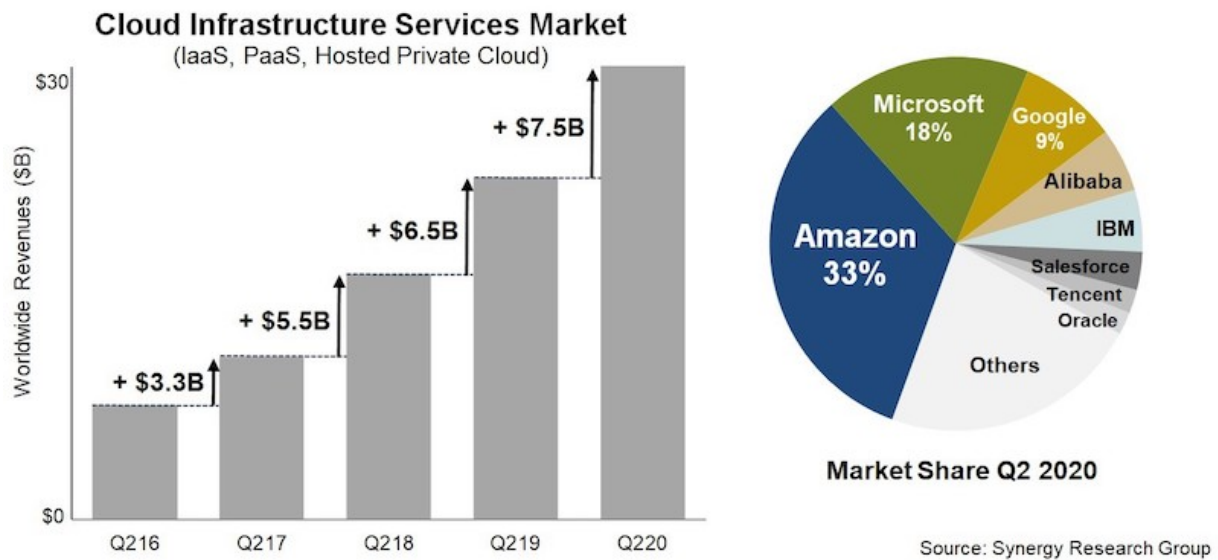
### **GitOps als Werkzeug für Continuous Delivery im Rahmen von DevOps**

GitOps wird von Gitlab als ein betriebliches Rahmenkonzept (operational Framework) definiert das DevOps Best Practices in der automatisierten Bereitstellung von Infrastruktur anwendet. GitOps benötigt dabei drei Hauptkomponenten:

Infrastructure as Code, Merge Requests und CI/CD. Organisationen die mit einer ausgereiften Anwendung der DevOps Kultur arbeiten können über GitOps hunderte Male pro Tag neuen Code auf den Produktionsservern installieren. Im praktischen Teil dieser Arbeit wird GitOps verwendet um da es eine einfache Integration mit Github ermöglicht, Github selbst wird als Versionskontrollsystem verwendet da es weithin etabliert und breit unterstützt ist und von der Novatec Consulting GmbH bevorzugt eingesetzt wird.

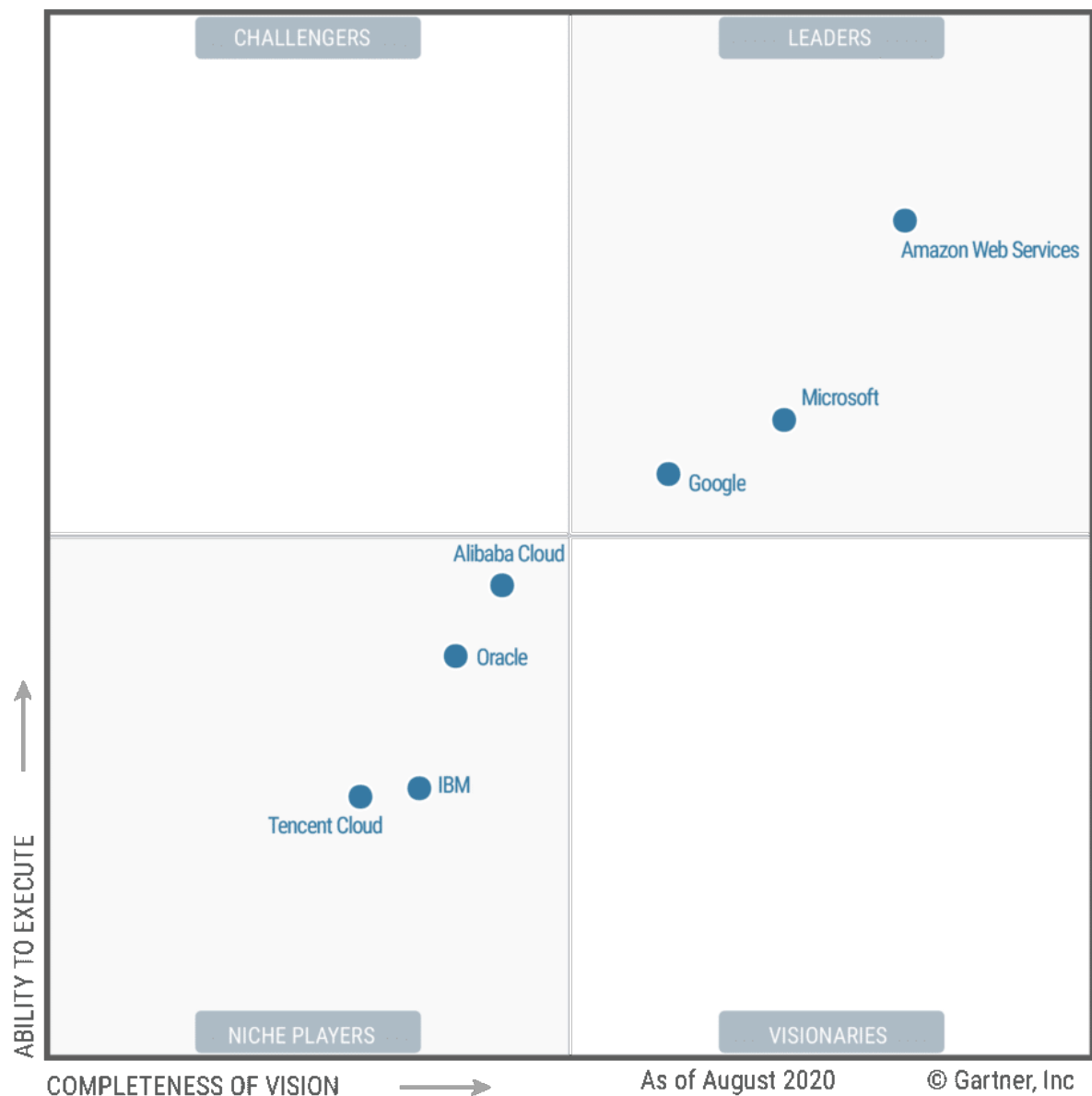
#### **2.1.4 Überblick über die wichtigsten Public Cloud Service Provider**

Da der Kern der Arbeit die Infrastructure as Code Unterstützung der wichtigsten Cloud Service Provider in deren Public Cloud untersucht soll hier ein kurzer Überblick über den aktuellen Markt in diesem Bereich gegeben werden.



**Abb. 2.2:** CSP Market Share Q2 2020 nach Umsatz

- 1. Amazon:** Amazon besitzt mit 33% den größten Anteil am Markt. Die Amazon Elastic Compute Cloud (EC2) ist die älteste der aktuell großen Cloud Computing Plattformen, der erste Release fand im August 2006 statt.
- 2. Microsoft:** Microsoft stellt den zweitgrößten Anteil am Markt mit 18%, der erste Release von Microsoft Cloud Computing Service Microsoft Azure erfolgte im Oktober 2008. Gemeinsam besaßen Amazon und Microsoft 2020 über 50% des gesamten Marktes.
- 3. Google:** Google's Google Cloud Platform steht mit 9% an dritter Stelle. Google Compute Engine, die IaaS Komponente der Cloud Services von Google wurde im Juni 2012 veröffentlicht.
- 4. Alibaba Cloud:** Die viertgrößte Cloud Plattform ist die Alibaba Cloud der Alibaba Group. Alibaba Cloud bietet zusätzlich zu seinem Elastic Compute Service (ECS) einen dedizierte GPU basierten Service an.
- 5. IBM Cloud:** Die aktuell fünftgrößte Cloud Computing Plattform ist die IBM Cloud von IBM. Bis Juni 2013 eine eigene Firma unter dem Namen Softlayer wurde diese von IBM übernommen und 2017 gemeinsam mit den anderen Cloud Services der Firma in ein einheitliches Portfolio unter dem Namen IBM Cloud aufgenommen.



**Abb. 2.3:** CSP Gartner Magic Quadrant August 2020

Der Gartner Magic Quadrant für Cloud Service Provider bietet einen groben Überblick über den Umfang der Angebote (Completeness of Vision) und die Ausgereiftheit einer Plattform (Ability to Execute). Deutlich erkennbar ist hier die Vormachtsstellung von Amazon gegenüber Microsoft und Google sowie der Abstand von Google zum nächst größten Anbieter Alibaba Cloud. TODO: Irgendwas mit IBM Ability to Execute vs Marktanteil verglichen mit Oracle

## 2.2 Infrastructure as Code

Infrastructure as Code beschreibt einen Ansatz zur Automatisierung der Bereitstellung von Infrastruktur basierend auf Methoden aus der Softwareentwicklung (vgl Kief Morris Infrastructure as Code S.4).

Statt eines manuellen Aufbaus und direkter Konfiguration der einzelnen Komponenten werden maschinenlesbare Dateien verfasst welche dann von einem IaC Tool eingelesen und verarbeitet werden. Dabei kommen bevorzugt deklarative Sprachen zum Einsatz deren höhere Abstraktion mehr Flexibilität als ein imperativer Ansatz erlaubt. (vgl. <https://docs.microsoft.com/us/devops/deliver/what-is-infrastructure-as-code>)

### 2.2.1 Technologischer Wandel und das Cloud Age Mindset

Durch die Technologien der Cloud ist es möglich eine gewünschte IT-Infrastruktur sehr viel schneller bereitzustellen als zuvor möglich. Statt des Einkaufs, Anschließens und Einrichtens eines physischen Servers das, je nach Szenario einen Zeitraum von mindestens mehreren Stunden oder Tagen bis zu Wochen gedauert hätte können virtuelle Ressourcen in der Cloud in wenigen Minuten bereitgestellt werden. Der schnellere Ablauf wird durch die Automatisierung von Prozessen wie etwa der Bereitstellung der Infrastruktur mithilfe von IaC Tools noch verstärkt, nicht nur bei der initialen Mit diesen Veränderungen wird das Management und die Erweiterung der bestehenden Systeme jedoch nicht unbedingt einfacher (vgl IaC Kief Morris S.3), die Verwendung der alten IT-Governance Modelle (TODO Fußnotenerklärung) die sich bisher bewährt haben sind jedoch aufgrund der Veränderungen ungeeignet. Kief Morris stellt die fundamentalen Unterschiede des Arbeitens mit Cloud-Technologien mithilfe der folgenden Tabelle dar.

Iron Age	Cloud Age
Cost of change is high	Cost of change is low
Changes represent failure (changes must be "managed," "controlled")	Changes represent learning and improvement
Reduce opportunities to fail	Maximize speed of improvement
Deliver in large batches, test at the end	Deliver small changes, test continuously
Long release cycles	Short release cycles
Monolithic architectures (fewer, larger moving parts)	Microservices architectures (more, smaller parts)
GUI-driven or physical configuration	Configuration as Code

**Abb. 2.4:** Iron Age vs Cloud Age

Veränderungen in der 'Iron Age' sind aufwändig und teuer und stellen ein Risiko dar, es wird versucht diese Risikopunkte zu reduzieren daher werden viele Veränderung gebündelt getestet und eingeführt wodurch lange Release-Zyklen entstehen. Die Architekturen die dadurch befördert werden sind eher monolytisch die Konfiguration erfolgt eher mithilfe von GUI gesteuerten Programmen oder physischen, zum Beispiel wenn ein neuer Server in ein Netzwerk eingebunden wird. Veränderungen in der Cloud stellen fast genau das Gegenteil dar, daher wird erkennbar dass eine auf die 'Iron Age' zugeschnittene Arbeitsweise für die Cloud nicht effektiv ist, es ist ein neues Cloud Age Mindset das die rechte Spalte der Tabelle verinnerlicht erforderlich um die Vorteile der Cloud wirklich effektiv nutzen zu können.

### 2.2.2 Vorteile von IaC und durch IaC gelöste Probleme des manuellen Provisionings

**Kein Configuration Drift durch einheitliche Codebase:** Configuration drift bezeichnet eine über die Zeit wachsende Abweichung zweier ursprünglich identischer Systeme. Wird ein gleiches System, zum Beispiel ein Applikationsserver, in verschiedenen Umgebungen eingesetzt stellen diese oftmals auch leicht verschiedene Anforderungen auf die dann mit Optimierungen, etwa in Form spezifischer Konfigurationsdetails, reagiert werden kann. Wird nun das ursprüngliche System geupdated werden individuelle, oft undokumentierte Anpassungen nicht berücksichtigt und ein Update kann unbequeme Konsequenzen nach sich ziehen. Werden alle Veränderungen in einer einheitlichen Codebase verwaltet und Updates häufig vorgenommen verhindert man starken Configuration Drift der über Zeit stattfindet

**Wiederverwendbarkeit durch einheitlichen Code:** Ein weiterer Vorteil der durch die Verwendung einer einheitlichen Codebase entsteht ist die Wiederverwendbarkeit und Reproduzierbarkeit eines Systems. Wenn ein identisches System an einer anderen Stelle aufgebaut werden soll oder sollte das System aus unvorhergesehen Gründen in seiner Gesamtheit verloren gehen kann es schnell und mit verhältnismäßig geringem Aufwand reproduziert werden. Ein dazu passender Ausdruck in Bezug auf Server ist 'Cattle not Pets'. Statt sich individuell und mit großem Aufwand um einzelne Server zu kümmern wie man es etwa mit dem eigenen Haustier tut sollten Server wie leicht ersetzbares Vieh behandelt werden.

**Schnelleres Provisioning durch Cloud:** Ein bereits häufig angesprochener Vorteil ist die schnelle Bereitstellung, auf tiefere Erklärung kann daher hier verzichtet werden.

**Schnellerer Profit:** Schnellere Bereitstellung der Hardware und schnellere Auslieferung von Software die durch DevOps Methoden begünstigt werden erzeugen entsprechend auch schneller einen Mehrwert.

**Einheitliches Tooling in Dev, Ops und weiteren Beteiligten Teams:** Verwendung von IaC fördert die einheitlicheres Tooling in allen Bereichen die mit einem Softwareprodukt in Zusammenhang stehen. Cloud Technologien fördern und ermöglichen diese Vereinheitlichung, manuelles Provisioning ohne ein Cloud Modell erschwert dies oder macht es sogar unmöglich.

**Stärkere Automatisierung im Arbeitsablauf:** Automatisierung bedeutet immer einen gewissen upfront Overhead, jedoch kann auf längere Sicht deutlich von automatisierten Abläufen profitiert werden. Ein Beispiel hierfür sind automatisierte und in eine CI/CD Pipeline integrierte Tests.

**Höhere Zuverlässigkeit und Sicherheit durch schnelle Updates:** Von eine Struktur die schnelle und häufige Veränderungen fördert kann auch die Sicherheit und Zuverlässigkeit eines Systems profitieren. Auf Sicherheitsrisiken kann in kurzer Zeit und ohne viel Aufwand reagiert werden, eine unsichere Funktion kann etwa schnell durch eine sichere Variante ausgetauscht werden, mögliche damit verbundene Probleme können dann in automatisierten Tests erkannt werden wodurch das System zuverlässig bleibt.

**Schnellere Fehlersuche und -behebung:**

Fehler die dennoch auftreten können dann durch den Einsatz kleinerer Komponenten schneller isoliert, gefunden und behoben werden. Monolithische Systeme erschweren dies durch dadurch dass weniger klar isolierte Komponenten vorhanden sind die häufig mehr Abhängigkeiten aufweisen und daher schwerer als einzelne Einheit getestet werden können.

### 2.2.3 Herausforderungen und Argumente gegen den Einsatz von IaC

Kief Morris benennt drei Argumente die gegen die Einführung von IaC genannt werden, diese sollen hier Im Kontext der genannten Vorteile betrachtet werden.

**Veränderungen werden nicht oft genug durchgeführt um Automatisierung zu rechtfertigen.**



Die Idee dass ein System einmal erstellt und dann "fertig" ist und Automatisierung der Veränderungen daher überflüssig ist entspricht sehr selten der tatsächlichen Realität. IT-Systeme und damit auch IT-Infrastruktur wird während ihrem gesamten Lebenszyklus mehr oder weniger kontinuierlich verändert und erweitert.

Sicherheitslücken in alten Versionen von Softwarepackages oder Betriebssystemen sind keine Seltenheit und müssen gepatcht werden um einen sicheren und Zuverlässig Betrieb zu gewährleisten, neue Features in bestehender Software kann neue Infrastruktur, zum Beispiel in Form eines neuen Datenbankservers, notwendig machen oder eine veränderte Konfiguration ist nötig um die Performance zu steigern. Gerade bei Sicherheitslücken ist es wichtig Anpassungen nicht erst nach längerer Zeit sondern so schnell wie möglich durchzuführen um Sicherheit und Stabilität nicht zu gefährden. Ein weiteres Szenario das die Stabilität eines Systems gefährden kann ist ein schneller Zuwachs an Last die das System erfährt, daher sollte es möglich schnell und unkompliziert das System zu verändern indem mehr Ressourcen zur Verfügung gestellt werden.

Ä fundamental truth of the Cloud Age is: Stability comes from making changes."(IaC Buch S. 5)

Eine fundamentale Wahrheit des Cloud Zeitalters ist: Stabilität entsteht durch Veränderung.

### **Infrastruktur soll zuerst aufgebaut, danach automatisiert werden.**

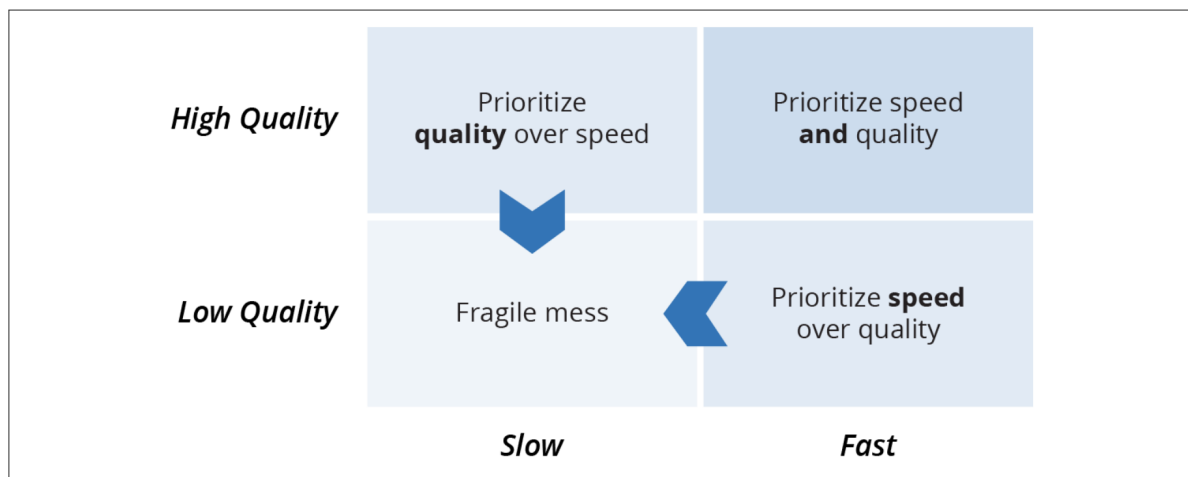
Die Umsetzung von Infrastructure as Code stellt eine durchaus große Herausforderung, das Erlernen eine "steep Curve"(IaC Buch S.6) dar, deren Nutzen nicht unbedingt direkt ersichtlich ist. Dadurch kommt es zu Situationen in denen es attraktiv erscheint Infrastruktur zuerst aufzubauen sich erst später um die Automatisierung zu kümmern. Mit diesem Ansatz werden viele der Vorteile von IaC jedoch verwirkt und die Schwierigkeit der Umsetzung von IaC fällt deutlich größer aus, verglichen mit einem Projekt das von Beginn auf IaC ausgelegt ist da "Automatisierung Teil des Designs und der Implementierung eines Systems ist"(IaC Buch S.6).

Automatisierung mit IaC vereinfacht auch die Implementierung automatisierter Tests eines Systems was es wiederum erlaubt Fehler schneller zu beheben. Ist dies bereits von Beginn an Teil des Arbeitsprozesses verbessert dies die Qualität der Infrastruktur.

TODO: Wenn erfolgreich werden neue Features gefordert, ohne Automatisierung gerät Management der wachsenden Infrastruktur schnell außer Kontrolle Deliver steady stream of value , build system incrementally

**Es muss zwischen schneller Umsetzung und hoher Qualität gewählt werden.**

Die Idee dass der Fokus auf hohe Geschwindigkeit und hohe Qualität sich gegenseitig behindert oder ausschließt mag logisch erscheinen, in der Praxis ist dies jedoch nicht der Fall. Die Accelerate State of DevOps Report Studie kam zu dem Schluss das Organisationen dazu tendieren entweder gut oder schlecht in beiden Kriterien abzuschneiden, wird eines vernachlässigt führt es am Ende in der Regel zu einem "fragile Mess" wie die untenstehende Tabelle erläutert.



**Abb. 2.5:** Iron Age vs Cloud Age IaC Busch S.6

Wird Geschwindigkeit über Qualität gestellt (Quadrant links oben) entstehen mit der Zeit chaotische und instabile Systeme an denen Veränderungen mit der Zeit nur noch erschwert und deshalb langsam durchgeführt werden können.

Wird Geschwindigkeit zu niedrig priorisiert führt es allerdings auch dazu dass letzten Endes durch Druck von Deadlines und schnellen workarounds technische Schulden aufgebaut werden die ebenfalls zu einem qualitativ schlechten System führen da notwendige Veränderungen nicht schnell genug eingearbeitet werden können.

Aufgrund dieser Probleme ist es wichtig Geschwindigkeit und Qualität gleichermaßen zu priorisieren, moderne Entwicklungsansätze DevOps haben das Ziel dies zu erreichen, der Erfolg wird von der Accelerate Studie belegt.

### 2.2.4 Die drei Kernverfahren von IaC

Kief Morris identifiziert drei Kernverfahren ("Core Practices") zur Implementierung von Infrastructure as Code:

- **Define everything as Code:** Alle Teile eines System in Form von Code zu definieren bringt mehrere Vorteile mit sich.  
Code kann mehrfach ausgeführt werden, daher ist eine als Code definiertes System Wiederverwendbar. Es können unkompliziert mehrere identische Instanzen erstellt werden, das Gilt auch für den Fall wenn Fehler auftreten und ein Neuaufbau erforderlich ist. Das Verhalten des Systems ist vorhersehbar, fortlaufendes automatisches Testen ist möglich und damit auch zuverlässiger.  
Definition in Code macht auch den Aufbau eines Systems transparenter, da dieser immer dem tatsächlichen Zustand des Systems entspricht und damit auch dokumentiert.
- **Continuously Test and Deliver All Work in Progress:** Fortlaufendes, automatisiertes Testen und Integrieren aller Komponenten die sich in Entwicklung befinden dient dem Ziel die Qualität eines Systems nicht nur einzutesten sondern von Beginn an und kontinuierlich einzubauen (The idea is to build quality in rather than trying to test quality in. (vgl IaC Buch S.10)). Nach der Accelerate Studie fördert es die Qualität der Arbeit eines Teams wenn jedes Mitglied mindestens einmal am Tag den eigenen Fortschritt integriert.
- **Build Small, Simple Pieces That You Can Change Independently:** Systeme die aus mehreren kleineren voneinander unabhängigen Komponenten bestehen sind generell stabiler als große Monolithen. Eine Änderung die einen Fehler verursacht betrifft nur diese Komponente in der die Änderung stattfindet, diese kann dann leichter isoliert und das Problem behoben werden, kleine Komponenten sind in der Regel auch weniger komplex und dadurch leichter zu verstehen. Ein einzelner Fehler nach einem Update hat auch den Vorteil dass nur diese Komponente und nicht das gesamte restliche System auf eine ältere Version zurückgesetzt werden muss um den Betrieb wieder herzustellen.

### 2.2.5 Die Rolle von Terraform im IaC Anwendungsprozess

Während der Anwendung von Infrastructure as Code kann primär zwischen zwei wichtigen Phasen unterschieden werden, einer initialen Einrichtungsphase und der darauf folgenden

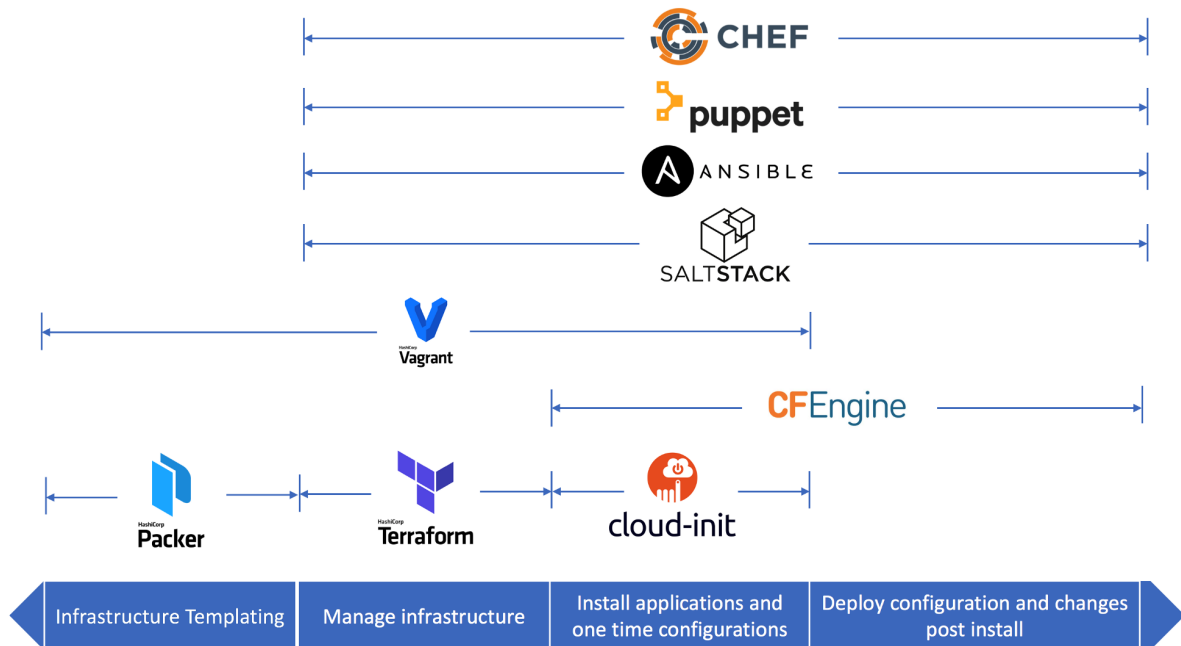
Wartungs- und Betriebsphase.

In der Einrichtung wird die Infrastruktur bereitgestellt und konfiguriert, genauso wird auch Software installiert und eingerichtet.

Nachdem das System dann in Betrieb genommen wurde können Anpassung notwendig werden, Server werden hinzugefügt und abgebaut, Software wird aktualisiert und neu konfiguriert.

### Überblick über die wichtigsten Infrastructure as Code Tools

Infrastructure as Code beinhaltet verschiedene konkrete Anwendungsfälle und entsprechend existieren auch Tools die zum Teil ein breiteres Spektrum von IaC abdecken, zum Teil aber auch eher spezialisiert sind; Terraform ist dabei ein Beispiel für ein spezialisiertes Tool. Die untenstehende Abbildung soll einen Überblick über die aktuell relevantesten Tools verschaffen, die Einordnungen sind dabei aber nicht unbedingt als absolut anzusehen. Es ist zum Beispiel möglich innerhalb eines Terraform-Deployments auch Software zu installieren und zu konfigurieren, allerdings wird die tatsächliche Installation dann eher per von Terraform aufgerufenen Skripten vorgenommen statt von Terraform selbst verwaltet zu werden, daher ist Terraform hier ausschließlich als Infrastruktur-Management Tool eingeordnet.



**Abb. 2.6:** Überblick IaC Tools <https://medium.com/cloudnativeinfra/when-to-use-which-infrastructure-as-code-tool-665af289fbde>

### Vorteile und Limitierungen von Terraform

Da sich diese Arbeit primär mit dem Provisioning von grundlegender Cloud Computing Infrastruktur in Form von VM's, Netzwerken, Datenspeicher und anderen grundlegenden Komponenten beschäftigen soll bietet sich der Einsatz eines darauf spezialisierten Tools an.

Hier gibt es zusätzlich zu Terraform eine weitere prominente Alternative: Pulumi.

Pulumi bietet im Vergleich zu Terraform mehrere Vorteile, es gibt allerdings auch diverse Nachteile die, gegen den Einsatz von Pulumi sprechen. Die folgenden Vorteile bietet Pulumi gegenüber Terraform:

- **Sprache:** Im Gegensatz zu Terraforms Domänen-spezifischer HashiCorp Configuration Language (HCL) setzt Pulumi auf den Einsatz bekannter General-Purpose Programmiersprachen, Python, TypeScript, JavaScript, Go, C# und F#, werden aktuell unterstützt.

Durch den Einsatz dieser Sprachen können Schleifen, Funktionen und weitere bekannte Konstrukte verwendet werden, Terraform bietet diese Möglichkeiten nicht, ähnliche Effekte werden nur mit Workarounds erreicht.

- **Testing:** Um Terraform-code testen zu können werden Third-party Tools benötigt, Terraform selbst bietet kein Test-Framework. Durch den Einsatz bekannter General-Purpose Programmiersprachen in Pulumi ist es auch möglich die dort zum Einsatz kommenden Frameworks zu verwenden, Integrationstest sind allerdings nur in Go unterstützt (<https://phoenixnap.com/blog/pulumi-vs-terraform>).
- **Pulumi mit Terraform:** Pulumi kann sowohl lokale als auch remote Terraform State Files lesen und verarbeiten. Dies erlaubt es Pulumi und Terraform nebeneinander einzusetzen um verschiedene Teile der Infrastruktur mit dem jeweils anderen Tool zu managen, soll zum Beispiel einer Umstellung von Terraform auf Pulumi erfolgen ist dies möglich ohne das gesamte System auf einmal außer Betrieb nehmen zu müssen.

Terraform wiederum bietet die folgenden Vorteile:

- **Modularität:** Terraform Module erlauben es ein System in mehrere klar definierte Komponenten zu strukturieren, dadurch wird die Wiederverwendbarkeit dieser Komponenten ermöglicht und gefördert. Die daraus resultierenden Vorteile wurden bereits in den vorangegangenen Kapiteln dargelegt, daher soll hier nicht weiter darauf eingegangen werden. Pulumi strukturiert Infrastruktur Code entweder in einem großen monolithischen Projekt oder vielen kleinen Mikroprojekten, beide Optionen sind weniger flexibel als die Lösung die Terraform bietet.
- **State Debugging:** Es ist beinahe unvermeidbar dass es während der Arbeit an einem Infrastrukturprojekt über einen längeren Zeitraum irgendwann einmal zu einem Fehler im State kommt. Sowohl Terraform als auch Pulumi bieten hier CLI Befehle die bei der Fehlersuche und -behebung unterstützen, Terraform bietet hier jedoch aktuell noch mehr Optionen mit denen Ressourcen aus dem aktuellen State gelöscht oder hinzugefügt werden können, dadurch wird weniger händische, und potentiell fehleranfällige Arbeit direkt im State File notwendig.
- **Weitere Verbreitung und größere Popularität:** Terraform wurde 2014, Pulumi 2017 veröffentlicht, entsprechend ist Terraform deutlich weiter verbreitet und verfügt über all die Vorteile die eine größere Community mit sich bringt. Dazu gehören mehr Lernressourcen, mehr Codebeispiele, größere Bekanntheit und mehr Jobs für die Arbeit mit Terraform.
- **Dokumentation:** Einen weiteren Vorteil von Terraform stellt dessen umfangreiche und ausgereifte Dokumentation sowie auch die Dokumentation der einzelnen Provider dar. Der genaue Aufbau der einzelnen Ressourcen wie etwa einer VM auf Google

Cloud Platform (google\_compute\_instance) ist mit einem Beispiel und der zugehörigen Argument Reference versehen aus der direkt ersichtlich wird welche Argumente notwendig (required), was der Zweck jedes einzelnen Arguments ist und wo anstelle eines einfachen Wertes ein Block erwartet wird.

- Terraform unterm Strich aktuell interessanter für große Projekte/Unternehmen => daher Wahl für diese Arbeit

- Ein wirklich fundierter Vergleich zwischen Terraform und Pulumi benötigt deutlich mehr Zeit und Anspruch als im Rahmen dieser Arbeit möglich, für die in diesem Fall und zu diesem Zeitpunkt zu treffende Entscheidung soll der grobe Vergleich hier ausreichen. Ein umfangreicherer Vergleich zwischen Terraform und Pulumi könnte in zwei Jahren ein anderes Ergebnis liefern als heute, beide Technologien und insbesondere Pulumi sind noch relativ neu und nicht vollständig ausgereift.

#### Weitere zu beachtende Aspekte

Ein Aspekt über Terraform der häufig falsch interpretiert und verbreitet wird ist dass Terraform als Cloud agnostisch beschrieben ist. Dies ist NICHT der Fall. Die einzelnen Terraform Provider welche jeweils die Abstraktion der Cloud Service Provider tatsächlich vornehmen sind nicht austauschbar das heißt der Terraform Code muss für jeden Cloud Service Provider individuell geschrieben werden, Terraform Code ist daher nicht Cloud agnostisch.

(- Secrets Management? Terraform Enterprise?)

### 2.2.6 Terraform Funktionsprinzip

Terraform ist ein Infrastructure as Code Tool das es ermöglicht Infrastruktur sicher und effizient aufzubauen, zu verändern und zu versionieren.

Terraform verwendet eine High-level DSL die es erlaubt die Infrastruktur in deklarativer und menschenlesbaren Konfigurationsdateien zu beschreiben die versioniert, geteilt und wiederverwendet werden können.

Bevor Veränderungen vorgenommen werden erzeugt Terraform einen Execution Plan der beschreibt welche Veränderungen durchgeführt werden sollen und überprüft werden kann bevor er ausgeführt wird.

Ein Ressourcengraph der Abhängigkeiten erfasst erlaubt es voneinander unabhängige Ressourcen parallel und dadurch effizient zu erzeugen und gibt einen besseren Einblick in den Aufbau des beschriebenen Systems.

Automatisierte Änderungen erlauben es komplexe Veränderungen an der Infrastruktur mit minimaler menschlicher Interaktion durchzuführen. Terraform beachtet bestehende Abhängigkeiten und nimmt Veränderungen durch Execution Plans schrittweise vor bis der definierte Zustand erreicht ist. (Alles obere vgl <https://www.terraform.io/intro/index.html>)

Die folgende Abbildung stellt das Funktionsprinzip von Terraform mit den wichtigsten beteiligten Komponenten dar.

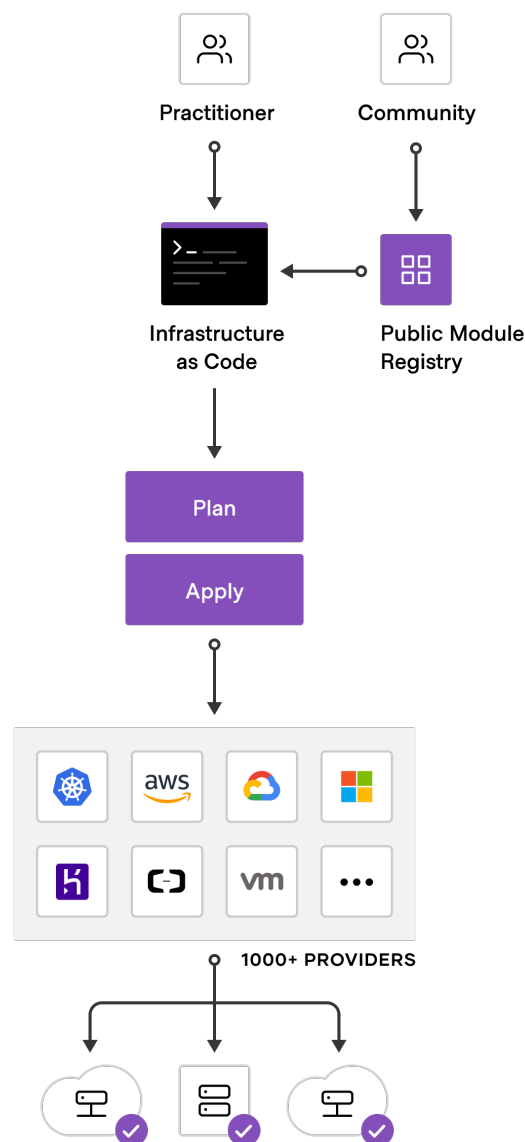


Abb. 2.7: Terraform Funktionsprinzip



Der Terraform Anwender schreibt den Infrastruktur Code in HCL. Dabei kann er auf vorhandene Module aus der Public Module Registry zurückgreifen um typische Strukturen wie Netzwerke aufzubauen. Die Terraform Core-Software nutzt dann Plugins, die Terraform Provider, um die jeweiligen Cloud Plattform API's anzusprechen die dann die Infrastruktur aufbauen. Die aktuell existierenden Ressourcen sind dabei in der terraform.tfstate Datei festgehalten, gemeinsam mit dem aktuellen State und dem Infrastruktur Code ermittelt Terraform welche Änderungen vorgenommen werden müssen um den im Code beschriebenen Zustand zu erreichen.

### Hashicorp Configuration Language

Die von Terraform verwendete Konfigurationssprache wurde mit dem Ziel entwickelt einen Kompromiss zwischen Maschinenfreundlichkeit und Menschenlesbarkeit zu erzielen. Existierende Serialisierungsformate, Konfigurationssprachen und Programmiersprachen konnten die Ziele der Terraform-Entwickler nicht erfüllen daher kommt nun bei Terraform eine DSL in Form der Hashicorp Configuration Language zum Einsatz.

HCL besteht aus drei grundlegenden Elementen: Blöcken (Blocks), Argumenten (Arguments) und Ausdrücken (Expressions).

#### TODO Codebeispiel

**Blöcke** stellen für gewöhnlich ein Objekt, im Fall von Infrastrukturcode meistens eine Computing Resource, dar. Blöcke besitzen einen Typ und Null bis mehrere label. Blöcke beinhalten Argumente und weitere verschachtelte Blöcke.

**Argumente** sind das was in den meisten Programmiersprachen die Variablen darstellen: Ein Wert der einem Namen zugewiesen wird.

**Ausdrücke** sind ähnlich wie in anderen Sprachen ein aus anderen Ausdrücken und Argumenten berechneter Wert, ein Argument ist in diesem Sinne die simpelste Form eines Ausdrucks.

### Input und Output Variablen

Input Variables sind nützlich um Parameter außerhalb des eigentlichen Terraform Codes anzupassen. Die ID des Projektes stellt einen solchen Parameter dar, befindet sich diese außerhalb des Codes muss nur die Datei welche die Inputvariablen enthält angepasst werden, alles andere kann ohne Veränderungen wiederverwendet werden.

### TODO Codebeispiel

Inputvariablen sind einfache Key-Value Paare die einen Standardwert und eine optionale Beschreibung besitzen.

Outputvariablen werden in der Regel verwendet um auf spezifische Parameter einfachen und schnellen Zugriff zu ermöglichen.

### TODO Codebeispiel

Werte wie die IP einer Virtuellen Maschine auf einer Public Cloud Plattform sind vor der Bereitstellung dieser nicht bekannt, werden aber häufig benötigt weshalb es nützlich ist eine Output Variable für diese zu deklarieren.

### Terraform Module

Module sind Container für mehrere Ressourcen die gemeinsam verwendet werden. Jedes Terraform Projekt besitzt ein Root Module das weitere Module verwenden kann. Die Terraform Registry stellt eine Vielzahl an veröffentlichten Modulen zur öffentlichen Verwendung bereit.

### Terraform Provider

Terraform benötigt Plugins, die Provider, um mit Cloud Providern, SaaS Providern und anderen API's interagieren zu können. (vgl <https://www.terraform.io/docs/language/providers/index.html>)  
Provider ermöglichen es Terraform für gewöhnlich einen bestimmten Provider zu konfigurieren, es gibt zum Beispiel einen Provider für Google Cloud Platform, es gibt aber auch Provider die eine lokales Utility Tool für die Nutzung durch Terraform konfigurieren. Öffentliche Provider werden primär auf der Terraform Registry bereitgestellt und dokumentiert.

### Terraform Workflow

Der grundlegende Terraform Workflow besteht aus vier Schritten:

#### TODO Code terraform init

Das init Kommando installiert und konfiguriert die notwendigen Provider und Module und konfiguriert ein Backend (Fußnote Backend) falls angegeben.

TODO Code terraform plan

Mit terraform plan wird ein Execution Plan erstellt. Dazu gehört den die aktuell existierenden Ressourcen zu erfassen, Veränderungen zwischen diesem Zustand und dem im Code konfigurierten Zustand zu erfassen und auf Grundlage dessen einen Plan zu erstellen welche Änderungen vorgenommen werden können um den Soll-Zustand zu erreichen.

Todo Code terraform apply

Durch terraform apply wird ein solcher Execution Plan ausgeführt und die darin enthaltenen Änderungen an der Infrastruktur vorgenommen.

Todo Code terraform destroy

Um den unkomplizierten Abbau eines mit Terraform definierten und aufgebauten Systems bewerkstelligen zu können wird das terraform destroy Kommando verwendet. Dies ist besonders nützlich um nicht mehr benötigte Testsysteme und andere temporäre Infrastrukturen zu entfernen.

## 3 Aufbau und Untersuchung

### 3.1 Evaluierungsanforderungen

#### 3.1.1 Ziel der Evaluierung

Das Ziel der Evaluierung ist es einen Vergleich zwischen den Terraform Providern von Amazon AWS, Google Cloud Platform und Microsoft Azure zu erstellen. Der Hintergrund dazu ist es als erstes zu ermitteln ob alle drei der großen Cloud Plattformen Infrastructure as Code mit Terraform für ein gegebenes Szenario unterstützen. Weiter sollen mehrere Kriterien verglichen werden um zu ermitteln ob alle Plattformen vergleichbare Ergebnisse liefern, oder ob eine Plattform durch besonders gute oder schwache Ergebnisse heraussticht. Funktionieren bestimmte Aspekte (wie zum Beispiel die Containerimage-Registry) unterschiedliche soll dies ebenfalls erfasst und bewertet werden.

Die Untersuchung soll auch dazu dienen die Effektivität von Infrastructure as Code mit Terraform zu demonstrieren. Zusätzlich soll dargestellt werden dass die Technologie einsatzreif für Verwendung in Produktionssystemen ist.

#### 3.1.2 Zu untersuchende Komponenten der Terraform Provider

Die Komponenten der Terraform Provider die untersucht werden sollen hängen von dem System ab das als Untersuchungsobjekt genutzt werden soll. Dieses setzt sich aus einem Netzwerk, einer Container-Registry, einem Kubernetes-Cluster und einem Datenbankserver mit mehreren Datenbanken sowie den dazugehörenden Firewalls zusammen. Zusätzlich sollen einfache virtuelle Maschinen betrachtet werden um ein vollständigeres Bild zu erhalten.

### 3.1.3 Grenzen und Limitierungen der Evaluierung

Die Terraform Provider können weitaus mehr Computing Ressourcen erzeugen als in dieser Arbeit untersucht werden. Aufgrund des Umfangs können nicht alle Ressourcen verglichen werden, die hier verwendeten wurden ausgewählt um die grundlegendsten Szenarien zu untersuchen. Da die durch die verschiedenen Cloud Provider bereitgestellten Computing Ressourcen jedoch nicht immer identisch sind ist ein perfekter Vergleich nicht möglich. Stellen die Cloud Provider nur Virtuelle Maschinen mit unterschiedlicher Leistungsfähigkeit bereit wird jedoch darauf geachtet möglichst ähnliche Konfigurationen zu verwenden um den Vergleich so identisch wie möglich zu gestalten. Das selbe gilt für die Größen von Festplatten und allen anderen vergleichbaren Objekten. Stehen nur sehr teure identische Maschinen bereit werden aus Kostengründen ebenfalls günstigere aber möglichst ähnliche Konfigurationen gewählt. Die genauen Daten der verglichenen Infrastrukturobjekte werden jeweils bei den Vergleichswerten aufgelistet.

## 3.2 Spezifizierung des Vergleichs

### 3.2.1 Auswahl der zu untersuchenden Aspekte und Eigenschaften

Das Softwarequalitätsmodell der ISO 25010 wird als ein Grundstein für die Evaluierung von Software beschrieben (Vergleich <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>) Da die Terraform Provider statt einer ganzheitliche Software nur Plu-Ins und damit eine kleinere Komponente von Terraform darstellen kann nicht jeder der in der ISO 25010 aufgelistet Punkte sinnvoll angewendet werden. Einige Bereiche, zum Beispiel der Bereich Security mit all seinen untergeordneten Themen, benötigen eine deutlich genauere Betrachtung als es im Rahmen dieser Arbeit möglich ist um ein sinnvolles und vollständiges Ergebnis gewährleisten zu können.

Aus diesem Grund sollen nur ein kleinerer Teil der aufgeführten Merkmale betrachtet werden, interessant sind vor allem solche die eine schnelle Einführung von Terraform in den Entwicklungsprozess beeinflussen. Ebenfalls interessant sind die Flexibilität und Wartbarkeit da insbesondere letztere einen der großen Vorteile darstellt die man durch den Einsatz von Infrastructure as Code Tools- und Prinzipien erzielen kann.



Abb. 3.1: Überblick ISO 25010

Im Folgenden sollen die einzelnen Merkmale der ISO 25010 kurz zusammengefasst werden und erläutert werden ob diese in der Evaluierung der Terraform Provider sinnvoll betrachtet werden können und ob eine Betrachtung im Rahmen dieser Arbeit vorgenommen werden kann.

### Functional Suitability

- **Functional completeness:** Die Vollständigkeit mit welcher die Aufgaben und Ziele die vom Nutzer an die Software gestellt werden erfüllt werden können.

Anwendbarkeit: Eine zufriedenstellende Functional Completeness ist die grundlegende Voraussetzung für den Einsatz jeder Software, dies gilt auch für den Einsatz von Terraform und die zu untersuchenden Provider.

- **Functional correctness:** Grad zu welchem die Software korrekte Ergebnisse mit einer ausreichenden Genauigkeit liefert.

Anwendbarkeit: Da im Falle der Bereitstellung von Infrastruktur durch deklarativen Code wenig Spielraum zulässt und im Fall der hier untersuchten Systeme lediglich ein „richtig“ oder „falsch“ zulässt spielt dieser Punkt keine Rolle.

- **Functional appropriateness:** Grad zu welchem die Funktionalitäten der Software die Erfüllung der Aufgaben ermöglichen und unterstützen.

Anwendbarkeit: Da die Terraform Provider im Prinzip nur eine Schnittstelle zu den jeweiligen Plattformen darstellen ist dieser Aspekt hier wenig relevant. Bei einer allgemeinen Betrachtung der Funktionalität von Terraform wäre dieser Punkt jedoch sehr interessant.

### Performance Efficiency

- **Time behaviour:** Grad zu welchem die Anforderungen an Bearbeitungszeit und Durchsatz erfüllt werden.

Anwendbarkeit: Die Dauer in welcher ein System von der Plattform durch den jeweiligen Terraform Provider bereitgestellt werden kann ist ein relevanter Aspekt der sehr gut untersucht und verglichen werden kann.

- **Resource utilization:** Beschreibt Umfang und Typ der Ressourcen die von der Software während des Betriebs benötigt werden.

Anwendbarkeit: Da die Terraform Provider selbst nur die Schnittstelle darstellen ist dieser Aspekt nicht sinnvoll anwendbar. Bei einem Vergleich verschiedener Softwaretools wäre dieser Aspekt von größerer Bedeutung.

- **Capacity:** Grad zu welchem die maximalen Limits der Software die Anforderungen erfüllen.

Anwendbarkeit: Auch dieser Aspekt ist nicht relevant, es trifft die die selbe Argumentation wie beim vorhergehenden Punkt zu.

### Compatibility

- **Co-existence:** Grad zu welchem die Software ihre Funktion erfüllen kann während sie sich eine Umgebung mit anderen Programmen teilt ohne deren Funktionalität zu beeinträchtigen.

Anwendbarkeit: Da in diesem Bereich im Rahmen der Arbeit mit wenigen Providern keine Probleme zu erwarten sind wird dieses Thema nicht explizit betrachtet.

- **Interoperability:** Grad zu welchem die Software mit anderen Produkten Informationen austauschen und verarbeiten kann.

Anwendbarkeit: Im Rahmen der Arbeit wird die Interaktion zwischen den Providern von Google Cloud Platform und Microsoft Azure mit Kubernetes und Helm betrachtet.

## Usability

- **Appropriateness recognizability:** Beschreibt wie einfach ein Nutzer erkennen kann ob die Software eine angemessene Lösung für dessen Anwendungsfall darstellt.

Anwendbarkeit: Dieser Aspekt kann zu einem gewissen Grad betrachtet und bewertet werden. Die Bezeichnungen der verschiedenen Computing Ressourcen können zum Beispiel ihre Funktion gut oder auch weniger genau beschreiben.

- **Learnability:** Grad zu welchem ein bestimmter User bestimmte Lernziele in einem definierten Kontext erreichen kann.

Anwendbarkeit: Eine Untersuchung der Erlernbarkeit eines Terraform Providers könnte durchaus vorgenommen werden. Um ein aussagekräftiges Ergebnis erzielen zu können wären jedoch umfangreichere Untersuchungen mit mehreren Usern notwendig weshalb im Rahmen dieser Arbeit dieses Kriterium nicht betrachtet werden soll.

- **Operability:** Beschreibt Attribute die ein System besitzt das dessen Bedienung vereinfacht.

Anwendbarkeit: Bei einer Betrachtung von Terraform als ganzes könnte dieser Punkt sinnvoll untersucht werden, die einzelnen Provider unterscheiden sich hier jedoch nicht.

- **User error protection:** Grad zu dem die Software den Nutzer gegen eigene Fehler (z.Bsp. während der Eingabe) schützt.

Anwendbarkeit: Hier liegt die selbe Lage wie beim vorhergehenden Punkt vor, wird daher nicht betrachtet.

- **User interface aesthetics:** Beschreibt wie ansprechend und zufriedenstellend das Userinterface bewertet werden kann.

Anwendbarkeit: Dieser Aspekt ist irrelevant da die Terraform Provider nicht direkt verwendet werden und kein User interface besitzen. Terraform selbst wird aus der Kommandozeile heraus bedient.

- **Accessibility:** Grad zu welchem die Software von verschiedenen Nutzern mit verschiedenen Fähigkeiten und Charakteristiken verwendet werden kann.

Anwendbarkeit: Für den Vergleich der Provider nicht anwendbar. Die selbe Argumentation wie bei Operability und User error protection treffen auch hier zu.



## Reliability

- **Maturity:** Beschreibt wie zuverlässig die Software unter normalen Bedingungen arbeitet.

Anwendbarkeit: Durch den begrenzten Umfang der in dieser Thesis durchgeführten Arbeiten kann dieser Aspekt nur begrenzt betrachtet werden. Sollten Auffälligkeiten in diesem Bereich auftreten werden diese jedoch erfasst und dokumentiert.

- **Availability:** Beschreibt ob die Software betriebsbereit und verfügbar ist.

Anwendbarkeit: Da die Terraform Provider nur ein Plug-In darstellen hängt dieser Aspekt sowohl von Terraform und den Cloud Plattformen ab, nicht von den Providern selbst.

- **Fault tolerance:** Grad zu welchem die Software trotz Hard- und Softwarefehlern ihre Funktion erfüllen kann.

Anwendbarkeit: Auch dieses Kriterium ist beim Vergleich zwischen den Providern nicht von Bedeutung da die Funktionalität nicht von diesen abhängt.

- **Recoverability:** Grad zu welchem die Software bei einer Unterbrechung oder einem Fehler betroffene Daten widerherstellen und die Funktionalität widerherstellen kann.

Anwendbarkeit: Hier kann untersucht werden ob Terraform bei den verschiedenen Providern unterschiedlich auf einen Abbruch der Internetverbindung oder bei einer fehlerhaften Eingabe reagiert.

## Security

Das Thema Security beinhaltet die Merkmale Confidentiality, Integrity, Non-repudiation, Accountability und Authenticity. Auf die Sicherheit von Terraform und den jeweiligen Cloud Plattformen soll aufgrund des Umfangs und der Komplexität des Themas im Rahmen dieser Arbeit nicht eingegangen werden. Für einen Vergleich der Terraform Provider selbst ist das Thema ohnehin weniger von Interesse da die Sicherheit stärker von Terraform selbst und den einzelnen Cloud Plattformen abhängt als von den individuellen Providern.

## Maintainability

- **Modularity:** Beschreibt den Grad zu welchem ein System voneinander unabhängigen Komponenten aufgebaut ist.

Anwendbarkeit: Dieser Aspekt wird von Terraform selbst implementiert und spielt bei der Betrachtung verschiedener Provider keine Rolle. Interessant wäre jedoch ein Vergleich zwischen Terraform und anderen IaC Tools welcher im Rahmen dieser Arbeit jedoch nicht vorgenommen wird.

- **Reusability:** Beschreibt wie gut sich Komponenten in anderen Systemen und Komponenten wiederverwenden lassen.

Anwendbarkeit: Auch dieser Aspekt ist für den Vergleich nicht relevant, die Argumentation ist die selbe wie für das Thema Modularity. Für einen Vergleich mit anderen Tools wäre dies ein sehr relevantes Thema da Terraform sehr gute Optionen besitzt um das Schreiben modularen Codes zu fördern.

- **Analysability:** Grad der Effektivität und Effizienz mit der der Einfluss auf das System ermittelt werden kann der durch eine Veränderung verursacht wird. Dazu gehört auch das Ermitteln von Komponenten die verändert werden sollen sowie das diagnostizieren von Fehlern und Schwächen des Produkts.

Anwendbarkeit: Analysability ist ein Aspekt der sowohl von Terraform selbst als auch von der Cloud Plattform abhängt. Dieses Thema soll im Rahmen dieses Vergleichs nicht betrachtet werden da es zu großen Teilen nicht vom Provider abhängt.

- **Modifiability:** Beschreibt wie gut ein System verändert werden kann ohne Fehler aufzuwerfen oder die Qualität zu verschlechtern.

Anwendbarkeit: Hier kann untersucht werden ob es im Fall der Modifizierung von einzelnen Ressourcen Unterschiede gibt. Es kann untersucht werden ob bei den Plattformen Ressourcen bei einer vergleichbaren Modifikation gelöscht und neu erstellt werden oder ob ein stoppen, verändern und neu starten ausreicht bzw. die Eigenschaft im laufenden Betrieb mit minimaler Unterbrechung des Services verändert werden kann.

- **Testability:** Effektivität und Effizienz mit der Testkriterien etabliert werden und entsprechende Tests durchgeführt werden können.

Anwendbarkeit: Im Vergleich der Provider ist das Thema Testen weitgehend irrelevant. Sehr interessant wäre jedoch wieder der Vergleich mit anderen vergleichbaren

Tools, Terraform selbst beinhaltet wenig bis keine Möglichkeiten für Test und setzt mehr auf zusätzliche Tools und Frameworks.

### Portability

Portability fasst die Aspekte Adaptability Installability und Replaceability zusammen. Bei einem Vergleich der Terraform Provider können diese Aspekte nicht sinnvoll miteinander verglichen werden da diese von Terraform selbst und nicht von den Providern abhängen.

### Zusammenfassung der für den Vergleich ausgewählten Aspekte

Für den im Rahmen dieser Arbeit durchgeführten Vergleich werden nur einige der in der ISO 25010 definierten Qualitätsmerkmale betrachtet. Da die Terraform Provider nur einen Teil der gesamten Software ausmachen sind zahlreiche Aspekte kaum oder gar nicht von den Terraform Providern abhängig und werden aus diesem Grund nicht betrachtet, andere Themen sind zu umfangreich und/oder komplex um hier zufriedenstellend bewertet werden zu können.

Kurz zusammengefasst werden die folgenden Merkmale betrachtet:

- **Functional completeness**
- **Time behaviour**
- **Recoverability**
- **Modifiability**

### 3.2.2 Entscheidungskriterien für die Evaluierung

Da im Falle dieser Arbeit ein Vergleich ohne bestimmte quantitative Ziele vorgenommen wird werden hier keine spezifischen Entscheidungskriterien wie in der ISO 25040 beschrieben. Stattdessen soll betrachtet werden wie die drei Provider in Relation zueinander abschneiden. Dadurch kann ermittelt werden ob ein Provider den anderen in bestimmtem Bereichen überlegen ist, ob sich zum Beispiel das zeitliche Verhalten wesentlich unterscheidet oder ob ein Provider grundlegende Bedingungen wie die funktionale Vollständigkeit nicht aufweist.

Die gemessenen Werte und andere auffallende Merkmale sollen jeweils tabellarisch dargestellt werden um die Übersichtlichkeit zu fördern und Unterschiede der Provider leicht einsehbar zu machen.

## 3.3 Umsetzung

### 3.3.1 Eingesetzte Software und Tools


In diesem Abschnitt sollen die Tools die bei der Umsetzung zum Einsatz kommen knapp zusammengefasst werden.

Als Source-code Editor kommt Visual Studio Code zum Einsatz.

Die Versionierung des Source-codes erfolgt durch ein privates Git-Repository auf Github. Github Actions wird eingesetzt um Änderungen am Infrastruktursystem automatisch durchzuführen. Um das automatische Deployment der Infrastruktur zu ermöglichen werden die dafür notwendigen Secrets als Github Actions Secrets direkt im Repository gespeichert. Zeitmessungen werden mit dem Bash Kommando **time <command>** durchgeführt und das Ergebnis der Zeile **real** verwendet.

Terraform und Terraform Provider kommen in den jeweils im Source-code definierten Versionen zum Einsatz.

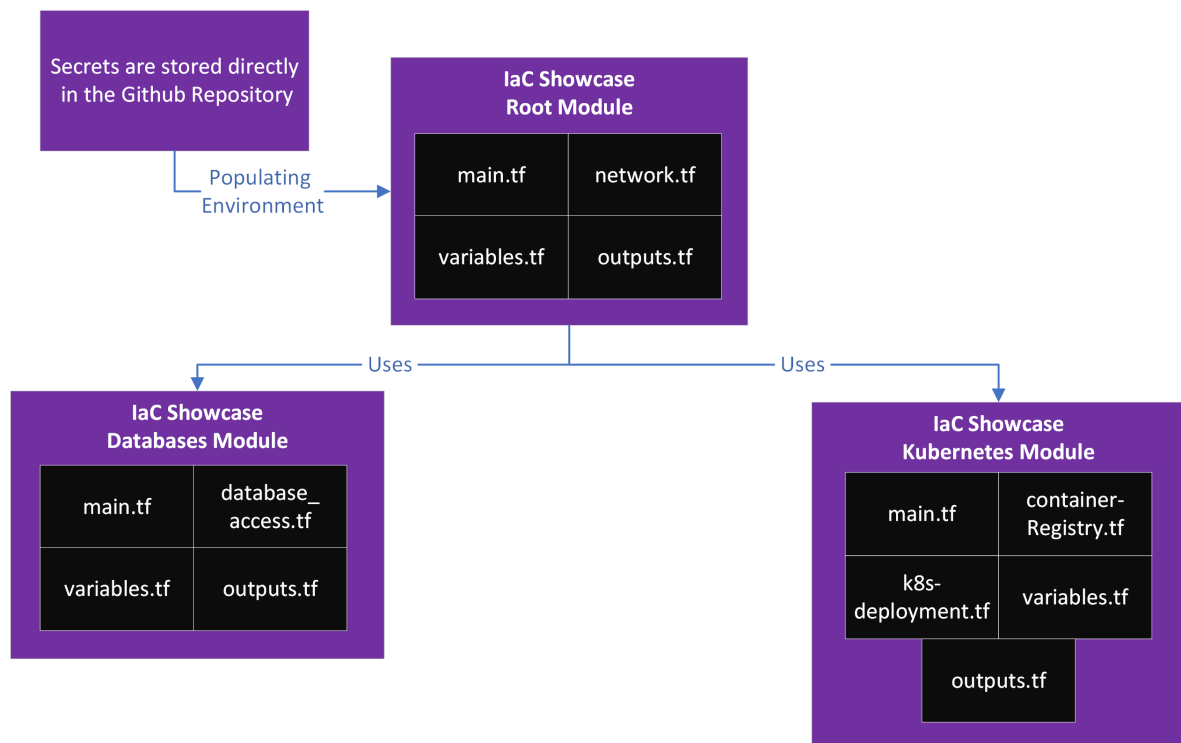
### 3.3.2 High-level Aufbau der Infrastruktur des Versuchsobjekts

Als primäres Testobjekt dient ein Infrastruktursystem das typische Anwendungsfälle repräsentieren soll. Es besteht aus Netzwerk, mehreren Datenbanken und einem Kubernetes Cluster mit mehreren Worker Nodes. Das System ist eine leicht veränderte Variante des öffentlich einsehbaren Infrastruktur-Showcase der Novatec.  [Link?](#)

Weiterhin existiert ein bereits manuell erstellter Object Storage mithilfe dessen auf einer ebenfalls manuell erstellten Container Registry das Dockerimage für den Kubernetes Cluster bereitgestellt wird. Dieses Image bzw. die Registry wird dann im Terraform Projekt als Data Source erfasst.

Ergänzend zu diesem System sollen zusätzlich einige Tests mit einzelnen Ressourcen wie zum Beispiel Virtuellen Maschinen durchgeführt werden.

Der originale Infrastruktur-Showcase besteht aufgrund der verhältnismäßig geringen Größe und Komplexität aus einem einzelnen Terraform Modul "Platform" das die gesamte Infrastruktur enthält. Für diese Arbeit wurden die Kubernetes- und Datenbank-Komponenten zu Demonstrationszwecken in jeweils eigene Module ausgelagert. Diese Module könnten in eine öffentliche oder auch private Terraform Module Registry hochgeladen werden um später in weiteren Projekten verwendet werden zu können.



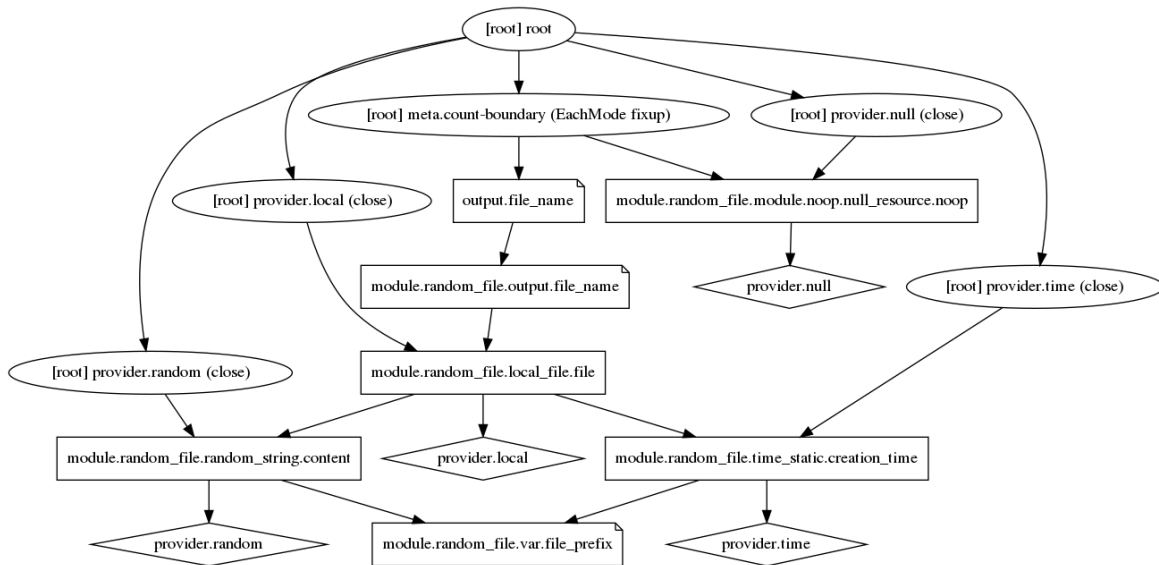
**Abb. 3.2:** Infrastructure Showcase High-level Struktur

In der Abbildung der High-level Struktur wird auf eine detaillierte Darstellung der Ressourcen verzichtet da je nach Plattform verschiedene Funktionalitäten in unterschiedlichen Ressourcen zusammengefasst werden können. Stattdessen wird durch .tf-Dateien dargestellt welche Funktionalität in in welchem Modul vorhanden sein soll und wie der Code des Systems in Hinsicht auf Dateien aufgeteilt werden soll.

### 3.3.3 Konkreter Aufbau der Versuchssysteme

Im folgenden Unterkapitel werden die konkreten Aufbauten der Systeme dargestellt. Die Diagramme der Systeme wurden mithilfe des in terraform integrierten **terraform graph**

erstellt und im Anschluss dem externen Tool **Terraform Graph Beautifier** visuell aufbereitet. In den Diagrammen sind jeweils alle Ressourcen der Systeme sowie alle Variablen, Outputs und deren Abhängigkeiten dargestellt.



**Abb. 3.3:** Beispiel Terraform Graph

Das obenstehende Abbild zeigt einen „rohen“ Graphen der direkt von Terraform mithilfe des entsprechenden Befehls erstellt wurde. Bereits dieses sehr kleine System wirkt schon leicht unübersichtlich, mit wachsender Größe wird es zunehmend schwierig einen Überblick über die einzelnen Komponenten und vor allem die Module eines Systems zu behalten. In den unten stehenden Abbildungen, die von Terraform Graph erstellt wurden, werden die Module farblich hervorgehoben.

Konkreter Aufbau in Microsoft Azure

## Konkreter Aufbau in Google Cloud Platform

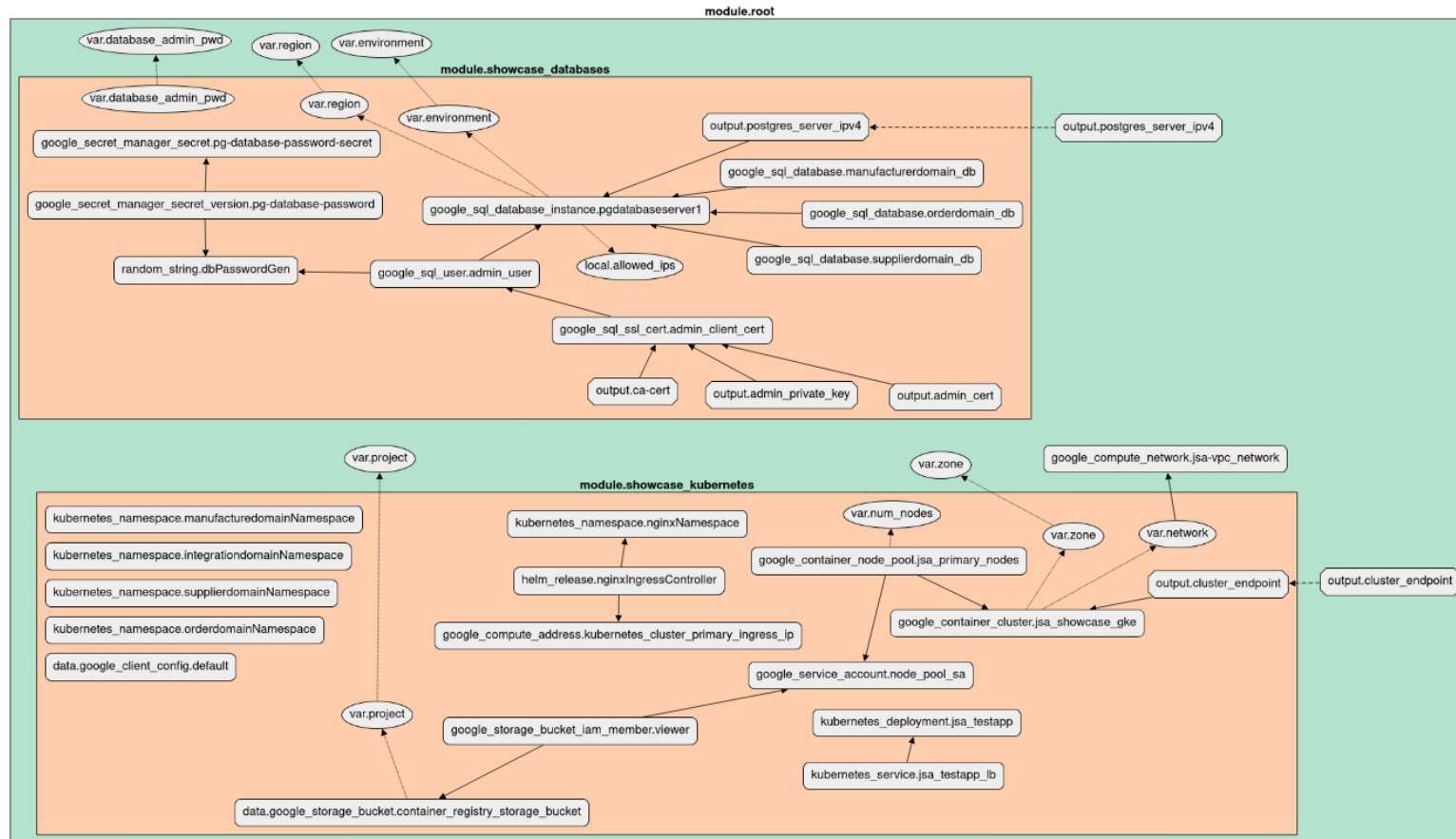


Abb. 3.4: Testsystem in Google Cloud Platform

Eine Auffälligkeit in Google Cloud im Vergleich mit Microsoft Azure ist der Secrets Storage der nur einmal projektweit existiert. Zusätzlich war das Anlegen der Container Registry nicht vollständig ausgereift. Es existieren zwei Optionen: Die aktuelle Artifact Registry deren Benutzung von Google empfohlen wird und die legacy Container Registry. Die Artifact Registry war zum aktuellen Zeitpunkt jedoch noch nicht vollständig im Terraform Provider implementiert, es existierte keine Möglichkeit eine existierende Artifact Registry als Data Source zu erfassen. Die Container Registry war daher die einzige Option einen Speicherort für das Image im Vorfeld anzulegen und in Terraform als Data Source einzubinden.

Weiterhin war es notwendig einen Serviceaccount anzulegen um das Image aus der Registry zu laden und in den Nodes des Kubernetes Clusters zu installieren.

Die weiteren Ressourcen konnten ohne weiteren Aufwand in einer einfachen Weise implementiert und verwendet werden.

### 3.3.4 Durchzuführende Messungen und Analysen

#### Functional completeness

Eine essentielle Voraussetzung für den Einsatz von Terraform ist die Vollständigkeit der Funktionalitäten die für ein gegebenes Infrastruktursystem erforderlich sind. Daher wird bei der Implementierung des Testsystems als erstes analysiert ob alle erforderlichen Ressourcen und Funktionalitäten in einer nutzbaren Form vorhanden sind.

Betrachtet werden sollen die folgenden Ressourcen und Funktionalitäten:

- Anlegen virtueller Maschinen mit dem Betriebssystem Ubuntu 20.04.
- Anlegen eines PostgreSQL-Datenbankservers
- Anlegen mehrerer Datenbanken mit Charset UTF-8 und Collation English\_United States.1252
- Anlegen eines Secret Storage
- Anlegen eines Kubernetes Cluster
- Erfassen einer Container Image Registry als Data Source

Zu beachten gilt dass das Referenzsystem in Microsoft Azure implementiert ist weshalb alle Funktionalitäten dort entsprechend garantiert sind.



### Time behaviour

Eine weiterer kritischer Aspekt ist das zeitliche Verhalten von Terraform in Verbindung mit dem Terraform Provider und der Cloud Plattform. Besonders in der Entwicklungsphase eines Infrastruktursystems wird dieses oft ab- und wiederaufgebaut, daher ist der zeitliche Aufwand der dafür benötigt wird von großer Bedeutung.

Untersucht werden soll das zeitliche Verhalten der aufgelisteten Systeme:

- Das gesamte Versuchssystem
- Virtuelle Maschine mit dem Betriebssystem Ubuntu 20.04.
- PostgreSQL-Datenbankserver
- Kubernetes Cluster mit default Node Pool

Es werden sowohl die Dauer für den Auf- als auch Abbau gemessen. Jeder Test wird dreimal wiederholt und der Mittelwert der gemessenen Werte bewertet. Die Messungen werden nur durchgeführt sofern die Plattform den Status der betroffenen Dienste als Normal/OK führt.

### Recoverability

Für diesen Test wird zunächst der Aufbau des Systems angestoßen durch das gewohnte *terraform apply* angestoßen. Nach Ablauf von ca. der Hälfte der zum vollständigen Aufbau benötigten Zeit wird die Internetverbindung im Betriebssystem manuell unterbrochen. Im Anschluss an diese Unterbrechung wird *terraform apply* erneut ausgeführt und überprüft ob die Ressourcen anschließend zur Verfügung stehen. Sollte dies fehlschlagen wird der aufgetretene Fehler dokumentiert.

Folgende Szenarien werden untersucht:

- Virtuelle Maschine.
- PostgreSQL-Datenbankserver
- Datenbank unter Standardeinstellungen
- Kubernetes Cluster mit default Node Pool

## Modifiability

Hier wird untersucht wie unterschiedliche Ressourcen auf eine Modifikation reagieren. Mithilfe der jeweiligen Browserbasierten GUI der Cloud Plattformen wird der Effekt beobachtet und dokumentiert.

Folgende Szenarien werden untersucht:

- Änderung des Betriebssystems einer Virtuellen Maschine
- Vergrößerung der Festplatte einer Virtuellen Maschine
- Wechsel des Netzwerks einer Virtuellen Maschine
- Vergrößerung der Festplattengröße eines Datenbankservers
- Änderung des Charset einer Datenbank in einem PostgreSQL-Server
- Vergrößerung eines Node Pools von 2 auf 3 Nodes

## 4 Ergebnisse und Bewertung

### 4.1 Evaluierung der Functional completeness

	Google Cloud Platform	Microsoft Azure
VM Ubuntu 20.04	Funktionalität wird durch Ressource google_compute_instance erfüllt.	Funktionalität wird durch Ressource azure_virtual_machine erfüllt.
PostgreSQL Datenbankserver	Funktionalität wird durch Ressource google_sql_database_instance erfüllt.	Funktionalität wird durch Ressource azure_postgresql_server erfüllt.
Anlegen von Datenbanken mit Charset UTF-8 und Collation English_United States.1252	Funktionalität wird durch Ressource google_sql_database teilweise erfüllt. Collation kann bei Erstellung nicht wie erwünscht definiert werden. Default ist en_US.UTF8.	Funktionalität wird durch Ressource azure_postgresql_database erfüllt. Charset und Collation werden bei Erstellung definiert
Secret Storage	Funktionalität wird durch Ressource google_secret_manager_secret erfüllt. Es existiert ein Secret Storage je Google Project.	Funktionalität wird durch Ressource azure_key_vault_secret erfüllt. Mehrere secret storages in Resource Group möglich.
Kubernetes Cluster	Funktionalität wird durch Ressource google_container_cluster erfüllt.	Funktionalität wird durch Ressource azure_kubernetes_cluster erfüllt.
Einbindung von Container Registry als Data Source	Funktionalität wird durch Data Source google_storage_bucket erfüllt werden.	Funktionalität wird durch Data Source azure_container_registry erfüllt.

**Tab. 4.1:** Functional Completeness von GCP und Azure

## Bewertung

Die grundlegenden Computing Ressourcen werden durch beide Terraform Provider erwartungsgemäß bereitgestellt, im Detail liegen jedoch einige Unterschiede vor.

Beim Anlegen der Datenbanken in GCP können die Funktionalitäten von Azure nicht vollständig abgebildet werden, Anpassung an Charset und Collation sind zum Zeitpunkt des Erstellens nicht möglich. Das Charset entspricht dem gewünschten Wert, die Collation jedoch nicht.

Der Secret Storage wird auf GCP einmal für ein GCP Project angelegt, Azure hingegen bietet die Möglichkeit mehrere Key Vaults innerhalb einer Resource Group anzulegen. Die Speicherung des tatsächlichen Secret in der Ressource `google_secret_manager_secret_version` anstelle des zuvor anzulegenden `google_secret_manager_secret` fällt etwas unintuitiv aus

Die Container Registry wird in Google Cloud in Form eines normalen Google Storage Buckets angelegt. Die bevorzugte Methode zur Speicherung eines Container Image in GCP stellt die Artifact Registry dar, diese ist jedoch zum aktuellen Zeitpunkt noch nicht vollständig im Terraform Provider implementiert. Es ist möglich eine Artifact Registry als Terraform Ressource zu erstellen, die Einbindung einer bereits bestehenden Artifact Registry als Data Source ist jedoch noch nicht möglich. Der Aufwand beim Erstellen der Container Registry über den Storage Bucket fällt höher aus als in Azure, hier wird beim Anlegen der Container Registry die Storage-Infrastruktur automatisch erstellt und gemanaged.

## 4.2 Time behaviour Tests

	Google Cloud Platform	Microsoft Azure
Aufbau/Abbau Testsystem	Aufbau 1: 15m48s Abbau 1: 7m54s Aufbau 2: 17:37 Abbau 2: 8m11s Aufbau 3: 15m12s Abbau 3: 6m52	Aufbau 1: 7m7s Abbau 1: 6m5s Aufbau 2: 7m59s Abbau 2: 7m44s Aufbau 3: 6m54s Abbau 3: 6m6s
Aufbau/Abbau VM	Aufbau 1: 0m30s Abbau 1: 1m3s Aufbau 2: 0m23s Abbau 2: 1m4s Aufbau 3: 0m25s Abbau 3: 1m4s	Aufbau 1: 1m12s Abbau 1: 1m36s Aufbau 2: 1m11s Abbau 2: 1m37s Aufbau 3: 1m11s Abbau 3: 1m40s
Aufbau/Abbau PostgreSQL-Datenbankserver	Aufbau 1: 3m49s Abbau 1: 1m3s Aufbau 2: 3m58s Abbau 2: 0m52s Aufbau 3: 3m57s Abbau 3: 0m52s	Aufbau 1: 2m28s Abbau 1: 0m42s Aufbau 2: 2m32s Abbau 2: 0m43s Aufbau 3: 2m28s Abbau 3: 0m40s
Aufbau/Abbau Kubernetes-Cluster mit default Node Pool	Aufbau 1: 3m59s Abbau 1: 2m52s Aufbau 2: 3m39s Abbau 2: 3m3s Aufbau 3: 3m31s Abbau 3: 2m53s	Aufbau 1: 4m36s Abbau 1: 5m33s Aufbau 2: 6m05s Abbau 2: 5m32s Aufbau 3: 5m13s Abbau 3: 5m29s
Terraform plan für Testsystem	Plan 1: 1.2s Plan 2: 1.2s Plan 3: 1.6s	Plan 1: 14.6s Plan 2: 15.2s Plan 3: 14.9s
Terraform plan für Datenbankserver	Plan 1: 0.9s Plan 2: 1.0s Plan 3: 1.0s	Plan 1: 22.2s Plan 2: 23.2s Plan 3: 23.3s
Terraform plan für Kubernetes-Cluster mit default Node Pool	Plan 1: 0.8s Plan 2: 0.9s Plan 3: 0.9s	Plan 1: 13.5s Plan 2: 13.3s Plan 3: 12.3s

**Tab. 4.2:** Time behaviour von GCP und Azure in verschiedenen Szenarien

### Bewertung

Bei der ersten Betrachtung der am Testsystem gemessenen Werte fällt ein starker Unterschied zwischen den für den Aufbau benötigten Zeiten des Testsystems auf, Google Cloud Platform benötigt hier beinahe die doppelte Zeit gegenüber Azure. Bei den zusätzlichen Tests die den Aufbau von Datenbankserver und Kubernetes Cluster analysieren liegt der Vorteil weiterhin bei Azure, jedoch nicht ansatzweise in der selben Ausprägung. Interessant ist ebenfalls der starke Kontrast zwischen den Zeiten die für die Erstellung des Execution Plan benötigt werden, hier liegen die Zeiten des Google Providers extrem unter denen des Azure Providers.

Die Betrachtung der Detailunterschiede im Aufbau des Testsystems und der individuellen Tests offenbart eine wahrscheinliche Ursache für die stark unterschiedlichen Zeiten des Testsystems: Der jeweils zum Einsatz kommende Maschinen-Typ.

Die Wahl für praktisch alle Teile des GCP-Systems fiel auf den Typ „f1-micro“. Dieser zeichnet sich durch besonders niedrige Kosten, aber dementsprechend auch geringe Leistung aus. Für die initiale Phase der Infrastruktur-Entwicklung während der noch praktisch keine Belastung der Komponenten stattfindet kann dies eine attraktive Option darstellen. Die Implementierung des Systems in Azure verwendet im Gegensatz dazu häufig die jeweilig empfohlene Option bzw. den in der Dokumentation als Beispiel verwendeten Typ. Die individuellen Tests werden mit Maschinen-Typen durchgeführt die eine möglichst vergleichbare Leistungsfähigkeit besitzen, konkret wurde dabei die Anzahl der zur Verfügung stehenden CPU Cores, die Größe des Arbeitsspeichers sowie Typ und Größe der Festplatte angepasst.

Die individuellen Tests offenbaren hierbei dass die Wahl des Maschinen- und Festplattentyps nicht nur bei der Leistungsfähigkeit sondern auch bei der für die Bereitstellung benötigten Zeit eine wichtige Rolle spielen. Ein Testsystem das ganz oder auch nur teilweise zu Test- und Entwicklungszwecken häufig zerstört und neu aufgebaut werden muss könnte von der Verwendung besserer Hardware deutlich profitieren.

Eine weitere mögliche Ursache für die Zeit Differenz beim Aufbau der Infrastruktur stellt die verwendete Availability-Region/Zone dar. Alle Tests wurden uniform in der Zone „europe-west3-b“ für Google Cloud Platform und der Location „westeurope“ für Azure durchgeführt. Es ist in jedem Falle denkbar dass es lokale Unterschiede bei der Verfügbarkeit gibt, die verfügbaren Maschinentypen variieren zum Teil ebenfalls.

Die Unterschiede bei der Erstellung des Execution Plan werfen ebenfalls Fragen nach den Ursachen auf, im Rahmen dieser Arbeit kann darauf aber nicht weiter eingegangen werden. Eine mögliche Ursache könnte in der Überprüfung zur Verfügung stehender Ressourcen durch den Terraform Provider liegen, eine fundierte Beantwortung dieser Frage erfordert jedoch weitere Recherche und Tests.

### 4.3 Recoverability Tests

	Google Cloud Platform	Microsoft Azure
Längere Unterbrechung des Internets, State in Remote Backend	State korumpiert, erneutes terraform apply führt zu Fehler, manuelles Entfernen der Ressource notwendig.	State korumpiert, erneutes terraform apply führt zu Fehler, zusätzlich zur VM-Ressource muss die Festplatte ebenfalls separat manuell gelöscht werden.
Temporäre Unterbrechung des Internets, lokaler State	Terraform state wird auch lokal korumpiert, manuelles löschen der VM-Ressource notwendig.	
„Soft Cancel“ von Terraform Apply	State korumpiert	Shutdown sofort, State korumpiert
„Hard Cancel“ von Terraform Apply	State korumpiert	Nicht durchgeführt

**Tab. 4.3:** Recoverability von GCP und Azure in Bezug auf individuelle Ressourcen

#### Bewertung

Die Tests zur Recoverability durchgeführt werden vermitteln ein relativ gleichmäßiges Bild: Das Ergebnis jedes Szenarios ist die Korruption des Terraform State. Dementsprechend konnte die Infrastruktur durch ein einfaches erneutes Ausführen des Execution Plan nicht erfolgreich deployed werden und ein manuelles Löschen der bereits teilweise konfigurierten Cloud Ressource war notwendig.

Aus diesem Ergebnis lässt sich ableiten dass keiner der Provider zusätzliche Mechanismen zur Fehlerprävention bei beim Deployment individueller Ressourcen implementiert. Komplexere Szenarien sollten ein erweitertes und vollständigeres Bild bieten können, mögliche Unterschiede könnten zum Beispiel bei Operationen die Datenspeicherung betreffen zu Tage treten.

Der größte Unterschied der bei die Durchführung betreffend auffiel ist die standardmäßige Timeout-Zeit für das Erstellen der Ressourcen. Google Cloud Platform bricht das Deployment im Falle eines Verbindungsverlusts innerhalb weniger Minuten ab, Azure wartet in diesem Fall bedeutend länger bevor die Operation fehlschlägt.

Die Lösung von Google verspricht schnelleres Feedback im Fall von Problemen, der Ansatz von Azure erlaubt es auch bei einem etwas längeren Ausfall des Internets das aktuelle Deployment ohne weitere manuelle Interaktion vollständig durchzuführen.

## 4.4 Modifiability Tests

	Google Cloud Platform	Microsoft Azure
Änderung des Betriebssystems einer Virtuellen Maschine	Ressource wird zerstört und neu erstellt.	Ressource wird zerstört und neu erstellt.
Vergrößerung der Festplatte einer Virtuellen Maschine	Ressource wird zerstört und neu erstellt.	Führt zu Error.
Veränderung des Machine-Type einer Virtuellen Maschine	Ressource wird modifiziert.	Ressource wird modifiziert.
Vergrößerung des default Node Pools von 1 auf 2 Nodes	Ressource wird zerstört und neu erstellt.	Ressource wird modifiziert.

**Tab. 4.4:** Modifiability von GCP und Azure in Bezug auf individuelle Ressourcen

### Bewertung

Die durchgeführten Tests zum Aspekt der Modifizierbarkeit offenbaren einige Unterschiede zwischen den getesteten Cloud Plattformen. Die Fähigkeit von Azure einen Node Pool erweitern zu können ohne diesen dabei neu anzulegen erlaubt es die gewünschte Änderung sehr viel schneller durchzuführen. Die zur Modifikation des bestehenden Node Pools benötigte Zeit entspricht etwa der Zeit des Erstellens ohne zuvor die alte Ressource zu zerstören. Im Falle von GCP ist dies der Fall, die alte Ressource muss zuerst vollständig zerstört sein bevor der neue und vergrößerte Node Pool angelegt wird.

Ein schwerwiegenderes Problem auf Seiten von Azure offenbart sich beim Erweitern der Festplattenkapazität einer VM. Während dies im Fall von GCP über das Zerstören und neu Erstellen der Ressource möglich ist führt dies im Falle von Azure zu einer Fehlermeldung. Diese Fehlermeldung tritt auf da die Festplatte als eigene Ressource angelegt wird. Die verwendete `azurerm_virtual_machine` Ressource bietet zwar die Option die Festplatte gemeinsam mit der Maschine zu löschen, in dem hier getesteten Szenario ist dies allerdings nicht möglich. Aufgrund dessen wird eine manuelle Interaktion notwendig um zum gewünschten Endzustand zu gelangen.



# 5 Schluss

## 5.1 Zusammenfassung der Arbeit

Im Rahmen dieser Arbeit wurde ein Vergleich der Public Cloud Service Provider Google Cloud Platform und Microsoft Azure in Bezug auf die Unterstützung von Infrastructure as Code mit Terraform durchgeführt. Um den Kontext, die technischen Grundlagen und die Motivation der Arbeit darzustellen werden im ersten Abschnitt Funktionsprinzip, Vorteile und Herausforderungen des modernen Cloud Computings sowie die Grundlagen von Infrastructure as Code und Terraform zusammengefasst. Grundlage für die Erklärung des Funktionsprinzip von Cloud Computing stellt hierbei die weithin akzeptierte Definition des National Institute of Standards and Technology der USA dar. Zusätzlich werden verschiedene Vorteile und Herausforderungen moderner Cloud Technologie kurz zusammengefasst und erläutert sowie ein Überblick über die wichtigsten Cloud Service Provider im aktuellen Markt hergestellt.

Die Grundlagen zu Infrastructure as Code werden durch den technologischen Wandel und eine Erklärung des Cloud Age Mindset eingeleitet. Die Notwendigkeit für den Einsatz von IaC wird durch die bestehenden Probleme welche durch IaC gelöst werden dargestellt, und drei Kernverfahren im Einsatz von IaC erläutert. Zum Abschluss der Grundlagen wird das IaC Tool Terraform vorgestellt. Es wird das Einsatzfeld von Terraform im Rahmen von IaC dargestellt sowie die Funktionsweise von Terraform und die von Terraform verwendete Sprache erläutert.

Der Hauptteil der Arbeit beschäftigt sich mit der Auswahl der Evaluierungskriterien, dem Aufbau des primären Testsystems und der Definition und Durchführung von zusätzlichen ergänzenden Tests.

Die Evaluierungskriterien werden aus dem Qualitätsmodell der ISO 25010 ausgewählt, die individuellen Qualitätsaspekte werden hinsichtlich ihrer Anwendbarkeit beurteilt und geeignete Aspekte zur Untersuchung ausgewählt.

Der nächste Teil besteht aus der Vorstellung des primären Testsystems. Dieses besteht aus drei Terraform-Modulen; dem Root-Modul sowie zwei nach ihrer Funktionalität getrennten Modulen für einen Datenbankserver und einem Kubernetes Cluster.

Dieses System sowie die zusätzlichen Tests werden hinsichtlich der zuvor ausgewählten Qualitätsmerkmale verglichen und bewertet.

Der letzte Abschnitt der Arbeit umfasst eine Zusammenfassung, das abschließende Fazit sowie einen Ausblick auf weitere untersuchenswerte Themen im Zusammen mit Infrastructure as Code und Terraform.

## 5.2 Fazit

Abschließend lässt sich die Aussage treffen dass sowohl Google Cloud Plattform als auch Microsoft Azure gut für den Einsatz von Infrastructure as Code mit Terraform geeignet sind. Beide Terraform Provider befinden sich in einem ausgereiften Zustand, grundlegend fehlende Funktionalitäten wurden im Rahmen dieser Evaluierung nicht festgestellt. Betrachtet man die Details macht Microsoft Azure insgesamt eine leicht bessere Figur. Die Deployment Zeiten sind im Durchschnitt etwas schneller und es existieren einige Funktionalitäten die von Google Cloud Plattform aktuell noch nicht vollständig abgedeckt sind. Die Dokumentation der Terraform Provider beider Cloud Plattformen fällt sehr gut und umfangreich aus, inklusive einiger Beispiele wie die individuellen Ressourcen konfiguriert werden können bzw. müssen.

## 5.3 Weitere untersuchenswerte Aspekte und aktuelle Entwicklungen

Im Verlauf dieser Arbeit wurden eine Vielzahl von Themen angeschnitten oder erwähnt die für die Umsetzung eines kundenorientierten Projekts mitunter große Relevanz besitzen, aufgrund des inhaltlichen und zeitlichen Rahmens jedoch nicht adäquat behandelt werden konnten.

Das vielleicht wichtigste dieser Themen ist das Testen von Infrastruktur Code. Interessant wäre hier nicht nur die Untersuchung der existierenden Möglichkeiten von Terraform sondern auch der Vergleich mit anderen Tools wie Pulumi. Allgemein wäre nicht nur der Vergleich der Testmöglichkeiten interessant, eine grundsätzliche Evaluierung beider Tools

könnte interessante Erkenntnisse über die Eignung dieser in verschiedenen Szenarien liefern und verschiedene Aspekte beider Tools genauer betrachten.

Ein weiteres interessantes Thema ist die Cloud Agnostizität von Terraform. Das Erfassen des Unterschieds zwischen Erwartung und Realität sowie das Erforschen der bestehenden Möglichkeiten zum Erreichen einer „echten“ Cloud Agnostizität in Form einer fundierten Arbeit wäre mit Sicherheit eine wertvolle Referenz für Neueinsteiger in das Thema IaC mit Terraform.

Im Grundlagenkapitel dieser Arbeit wurde dargestellt warum die Automatisierung von Infrastruktur von Beginn an essentiell ist um den vollen Umfang der Vorteile ausschöpfen zu können. Ein Großteil der aktuell existierenden Projekte ist jedoch aufgrund der verhältnismäßigen Neuheit von IaC und Terraform mit hoher Wahrscheinlichkeit noch nicht in dieser Form umgesetzt worden. Eine Erforschung von Best Practices und Tools zur Automatisierung bestehender Systeme hätte zum aktuellen Zeitpunkt und auch in kommenden Jahren einen hohen Wert.

# A Kapitel im Anhang

Alles was den Hauptteil unnötig vergrößert hätte, z. B. HW-/SW-Dokumentationen, Bedienungsanleitungen, Code-Listings, Diagramme

# Literaturverzeichnis

- [1] Thomas Nonnenmacher, LaTeX Grundlagen - Setzen einer wissenschaftlichen Arbeit Skript, 2008, <http://www.stz-softwaretechnik.de>; (*Bei STZ Internetseite unter Publikationen - Skripte*) [V. 2.0 26.02.08]
- [Gun04] Karsten Günther, LaTeX2 — Das umfassende Handbuch, Galileo Computing, 2004, <http://www.galileocomputing.de/katalog/buecher/titel/gp/titelID-768>; 1. Auflage