Hochschule Esslingen
University of Applied Sciences

Bachelorarbeit

Cloud Service Provider Evaluierung auf Basis von Infrastructure as Code Unterstützung

im Studiengang Softwaretechnik und Medieninformatik
der Fakultät Informationstechnik
Wintersemester 2021/22

Julian Schallenmüller

Zeitraum: 15.10.2021-31.01.2022
Prüfer: Prof. Dr.-Ing. Kai Warendorf
Zweitprüfer: Prof. Dr. rer. nat. Mirko Sonntag

Firma: Novatec 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 29. Januar 2022 _____

Unterschrift

Sperrvermerk

Die nachfolgende Bachelorarbeit enthält vertrauliche Daten der Novatec Consulting GmbH. Veröffentlichungen oder Vervielfältigungen dieser Arbeit – auch nur auszugsweise – sind ohne ausdrückliche Genehmigung der Novatec 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

Ich möchte mich an dieser Stelle bei der Firma Novatec bedanken, in der ich seit meiner Zeit als Praktikant, danach als Werkstudent und nun auch während meiner Bachelorarbeit immer willkommen war und bei allen Herausforderungen und Problemen stets unterstützt wurde.

Besonderer Dank geht hierbei an die Mitarbeiter der PA TC und insbesondere an meinen Betreuer Matthias Häussler. Herr Häussler stand bereits vor und vor allem während meiner Zeit als Bachelorant immer für Fragen und Ratschläge, auch über die normalen Arbeitszeiten hinaus, bereit.

Genauso möchte ich mich bei Herrn Prof. Dr.-Ing Warendorf für die Betreuung meiner Bachelorarbeit von Seiten der Hochschule Esslingen bedanken. Durch seine schnelle und unkomplizierte Art der Kommunikation konnten alle organisatorischen Fragen und Aufgaben rund um die Bachelorarbeit immer schnell beantwortet und bewältigt werden.

Kurz-Zusammenfassung

Im Rahmen dieser Arbeit wird ein Vergleich der Cloud Service Provider Microsoft und Google in Bezug auf deren Unterstützung von Infrastructure as Code mit Terraform durchgeführt.
Dieser Vergleich soll die Qualität der Unterstützung von deren Plattformen Micrsoft Azure und Google Cloud Platform in einem gewöhnlichen Szenario evaluieren und die Frage nach der Einsatzreife beider Plattformen beantworten.
Zusätzlich soll diese Arbeit ermitteln, wie einheitlich die konkrete Umsetzung von Infrastructure as Code mit Terraform in einem funktional gleichwertigen Szenario für verschiedene Plattformen ausfällt.

Für die Beantwortung dieser Fragen wurde ein beispielhaftes Infrastruktur-System auf beiden Plattformen mithilfe von Terraform implementiert und deployed. Die in der Evaluierung eingesetzten Bewertungskriterien wurden aus den Kriterien des Softwarequalitätsmodell der ISO/IEC 25010 ausgewählt.
Zusätzlich zu der Untersuchung des Testsystems wurde eine Reihe unterstützender Versuche durchgeführt, wodurch ein Teil der aufgeworfenen Fragen beantwortet und ein vollständigeres Bild bezüglich der Performance, Qualität und Unterschiede der untersuchten Plattformen aufgebaut werden konnte.

Das Ergebnis der Untersuchung lässt darauf schließen, dass beide Plattformen eine gute Unterstützung von Terraform bieten, Azure genießt hierbei einen leichten Vorsprung hinsichtlich Funktionsumfang und Performance.

Die Einheitlichkeit der Umsetzung fällt jedoch eher gering aus. Sie begrenzt sich auf die Anwendung einer einheitlichen Sprache und Bedienung durch Terraform, es ist jedoch weiterhin ein umfangreiches Verständnis der individuellen Cloud Plattform notwendig, um diese erfolgreich einsetzen zu können.

Dennoch ist es sinnvoll Terraform für das Deployment von Infrastruktur zu nutzen. Durch die Umsetzung der Prinzipien von Infrastructure as Code werden die Vorteile moderner Cloud Plattformen in vollem Umfang ausgeschöpft und die Wertschöpfungskette von der Idee zur Auslieferung an Kunden kann deutlich beschleunigt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Eines der wichtigsten Schlagworte im Zeitalter der fortschreitenden Digitalisierung ist der Begriff des Cloud Computings. Cloud Computing spielt heute längst nicht mehr nur in der IT-Industrie eine wichtige Rolle. Selbst in Bereichen wie der Finanzbranche, die besonders hohen Sicherheitsansprüchen gerecht werden muss, findet Cloud-basierte Software eine zunehmende Verbreitung [1].

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 jedoch auch immer eine Reihe eigener Herausforderungen mit sich. Für den erfolgreichen und gewinnbringenden Einsatz dieser Technologie ist es daher essentiell, diese zu verstehen und die neuen Herausforderungen mit angepasster Denkweise und neuen Werkzeugen anzugehen.

Das Thema mit dem sich diese Arbeit befasst wird, ist das automatisierte Management und die Bereitstellung von IT-Infrastruktur-Ressourcen, ein Teil der größeren Fachthematik Infrastructure as Code (IaC). Die Grundlagenkapitel werden zu diesem Zweck auf den technischen Kontext und die Relevanz von Cloud Computing, IaC und das Software Tool Terraform eingehen. Es wird erläutert werden, an welcher Stelle die entsprechenden Plattformen und Software zum Einsatz kommen, welche Probleme durch diese 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 verschiedener Cloud Service Provider unter dem zentralen Kriterium derer Unterstützung von IaC mit Terraform. Zu diesem Zweck wurde ein Infrastruktur-System auf Basis des Infrastruktur-Showcase der Firma Novatec[1] für ver-

1 Link: https://github.com/NovatecConsulting/technologyconsulting-showcase-infrastructure

schiedene Cloud Plattformen in Terraform Code implementiert und auf diesen deployed. Als Grundlage für den Vergleich dienen eine Auswahl von Aspekten aus dem Softwarequalitätsmodell der ISO/IEC 25010. Da jedoch nicht alle enthaltenen Qualitätsaspekte der Norm für diesen konkreten Vergleich geeignet sind, werden diese zunächst auf ihre Anwendbarkeit im vorliegenden Fall hin analysiert und bewertet, anschließend werden einige der relevantesten Kriterien ausgewählt und die Untersuchung anhand derer durchgeführt. Da sich durch Unterschiede zwischen den Cloud Plattformen, zum Beispiel bei Auswahl der Leistungsfähigkeit verwendeter Virtueller Maschinen, einige Auffälligkeiten ergeben, werden zusätzliche Tests definiert und durchgeführt, um ein insgesamt vollständigeres Bild liefern zu können.
Im Anschluss werden die Ergebnisse des Vergleichs bewertet und die daraus resultierenden Erkenntnisse zusammengefasst. Aus diesen Erkenntnissen kann dann ein Fazit gezogen werden, das deren Bedeutung im Kontext des aktuellen Stands der Technik interpretiert und bewertet. Den Abschluss bildet ein Ausblick auf zusätzliche Themen, die als nächstes betrachtet werden sollten, wenn es darum geht Infrastructure as Code für das Infrastruktur-Management in einer realen Produktivumgebung einzusetzen. Diese Themen werden im Verlauf der Arbeit bereits angesprochen, können hier jedoch noch nicht in zufriedenstellendem Umfang betrachtet werden.

2 Grundlagen

2.1 Funktionsprinzip, Vorteile und Herausforderungen des modernen Cloud Computings

Um die Rolle von Infrastructure as Code und Terraform vollständig erläutern zu können, sollte zuerst das grundlegende Funktionsprinzip und die verschiedenen Service- und Bereitstellungs-Modelle moderner Cloud Plattformen erklärt werden. Die am meisten verwendete Definition von Cloud Computing wurde vom National Institute of Standards and Technology der Vereinigten Staaten von Amerika veröffentlicht und wird im folgenden Kapitel zusammengefasst.

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[2] 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 Computing Ressourcen (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 Computing Ressourcen automatisch bereitstellen, es wird keine menschliche Interaktion benötigt.

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

Resource Pooling: Die Computing Ressourcen des Anbieters werden in einem gemeinsamen Pool für mehrere Kunden in einem Multi-Tenancy-fähigen Modell bereitgestellt, physische und virtuelle Ressourcen werden dynamisch zugewiesen und entsprechend der Nachfrage kontinuierlich neu verteilt. Es wird eine empfundene Ortsunabhängigkeit hergestellt, indem der Nutzer kein genaues Wissen darüber besitzt, wo sich dessen Ressourcen 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, Arbeitsspeicher 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 können zu jeder Zeit und in jedem Umfang bereitgestellt werden.

Measured Service: Cloud Systeme kontrollieren und optimieren Ressourcennutzung automatisch mithilfe eines Mess-Systems das auf einer abstrakten Ebene den entsprechenden Service (Datenspeicher, Rechenleistung, Benutzerkonten, etc.) 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).

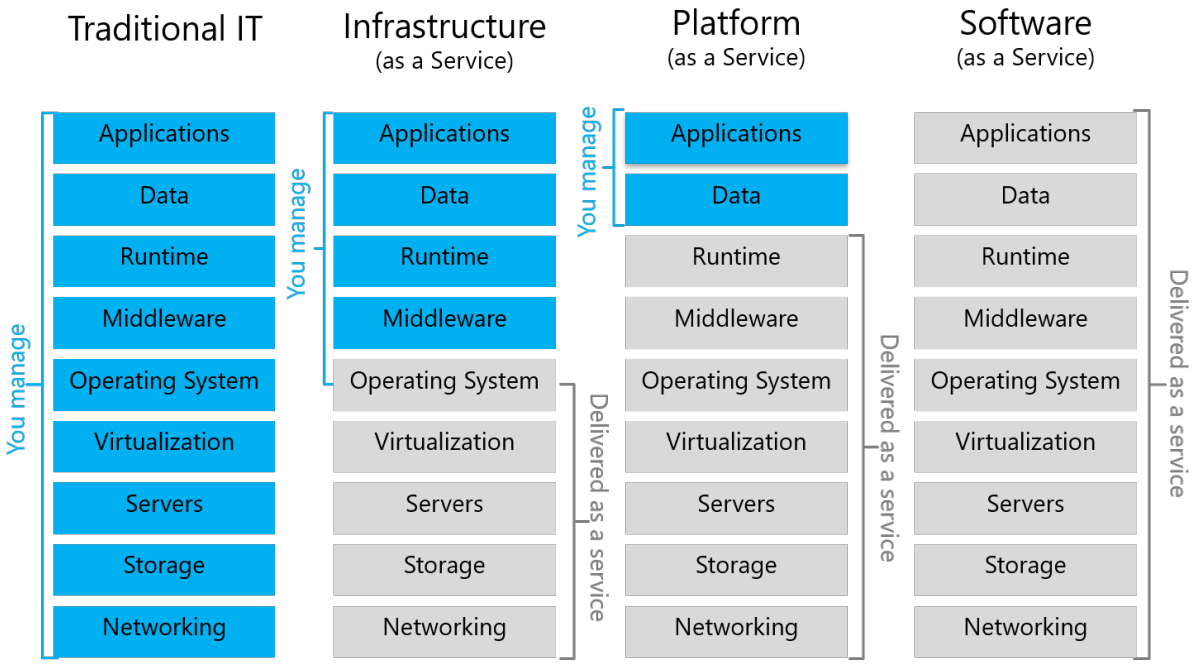

Abb. 2.1: Die Cloud Service Modelle im Überblick[3]

Infrastructure as a Service: Der Nutzer hat die Fähigkeit Rechenleistung, Datenspeicher, Netzwerke und weitere fundamentale Computing Ressourcen 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, beispielsweise Firewalls, haben.

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 Cloud Service Provider (CSP) bereitgestellt. Die darunter liegende Infrastruktur mit Netzwerken, Servern, Betriebssystemen und Speicher wird vom CSP betrieben, der Nutzer hat die Kontrolle über Anwendung und Konfiguration der Umgebung, in der die Anwendung betrieben wird.

Software as a Service: Dem Nutzer wird der Zugriff auf die vom CSP in der Cloud Infrastruktur betriebenen Softwareanwendung gewährt. Auf diese wird mithilfe eines Thin oder Fat Client zugegriffen, dabei kümmert sich der Nutzer nicht um den Betrieb und die Konfiguration der darunterliegenden Cloud Infrastruktur (Netzwerke, Server, Betriebssystem,

Speicher) und die Anwendung selbst mit Ausnahme eingeschränkter Nutzereinstellungen.

In Art der Bereitstellung eines Cloud Services werden vier grundlegende Modelle unterschieden: es existieren 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-Premises[1] 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-Premises 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-Premises 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 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 (Abb. 2.1). Potentiell hohe Kosten, die bereits vor der Inbetriebnahme eines Systems mit einem höheren

1 Hardware ist lokal vor Ort/nicht vor Ort

Risiko aufgewendet werden müssen, stellen in Form von individuell geringeren laufenden Beträgen ein deutlich reduziertes Risiko dar[4, S.23].
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[4, S.24].
Die Gesamtkostenersparnisse stehen auch im Zusammenhang mit Skaleneffekten[1], die für große Cloud Service Provider gelten. Der Betrieb eines einzelnen Servers ist im Verhältnis mit bedeutend höheren Kosten verbunden als das hinzufügen eines äquivalenten Systems zu einem Rechenzentrum im Betrieb von Mircosoft oder einem vergleichbaren Anbieter[4, S.24-25].

Skalierbarkeit: Besonders für schnell wachsende Unternehmen ist die Möglichkeit schneller 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[4, S.27].

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[4, S.28].

Security: Sicherheitsaspekte können sowohl einen Vor- als auch Nachteil von Cloud Computing darstellen. Hier sollen zuerst Vorteile dargelegt werden, potentielle Probleme sind im nächsten Abschnitt beschrieben.
Die technischen Möglichkeiten und besonders auch die Wahrnehmung des Themas Sicherheit in der Cloud unterlagen und unterliegen auch noch immer 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 ihre „natürliche“ Skalierbarkeit[4, S.25].

1 Zusammenhang zwischen produziertem Ertrag und eingesetzter Ressourcen

Herausforderungen und Risiken

Netzwerkabhängigkeit: Da der Zugriff auf Cloud Dienste über das Internet erfolgt, entsteht dadurch entsprechend auch eine hohe Abhängigkeit. Stabile und schnelle Netzwerkanbindung ist eine kritische Voraussetzung für effektives Arbeiten[5, S.737], 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 Kostenmodell, die den eignen Interessen stark entgegen steht, 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[6].

Security und Privacy: Sicherheitsrisiken sind einer der meistgenannten Gründe die gegen Cloud Computing sprechen[5, S.737], besonders im Fall der Nutzung einer Public Cloud. Die Gefahr, dass Daten in die Hände Dritter gelangen, kann zum beispielsweise nicht vollständig ausgeschlossen werden. Da die Verantwortung über die Sicherheit der Daten dem Cloud Anbieter unterliegt, kann es auch zu Problemen hinsichtlich Privatheit der Daten kommen, sollte etwa eine Regierungsorganisation Zugriff auf bestimmte Daten eines Nutzer verlangen, könnte dieser ohne dessen Einverständnis gewährt werden.

Kosten: Auch wenn die Nutzung von Cloud Computing mit dem Vorteil geringerer Kosten beworben wird, ist dies nicht zwingend garantiert. Werden die vorhandenen Systeme ungünstig verwendet, bleiben zum Beispiel viele gebuchte CPUs und IP-Adressen sowie weitere Ressourcen ungenutzt, können unnötig hohe Kosten entstehen. Auch während der Migrationsphase, in der möglicherweise beide Systeme parallel betrieben werden müssen, können höhere Kosten entstehen als in einem vergleichbaren Zeitraum davor. Nicht zuletzt stellt die Migration komplexer Systeme häufig eine große technische und dadurch auch finanzielle Herausforderung dar[7, S.321].

2.1.3 Überblick über die wichtigsten Cloud Service Provider

In diesem Abschnitt soll ein knapper Überblick über die wichtigsten Public Cloud Service Provider gegeben werden. Abbildung 2.2 stellt den Marktanteil der aktuell umsatz-

stärksten Cloud Service Provider sowie das globale Umsatzwachstum insgesamt dar. Zu beachten ist, dass in dieser Abbildung das Service Modell SaaS nicht beinhaltet ist.

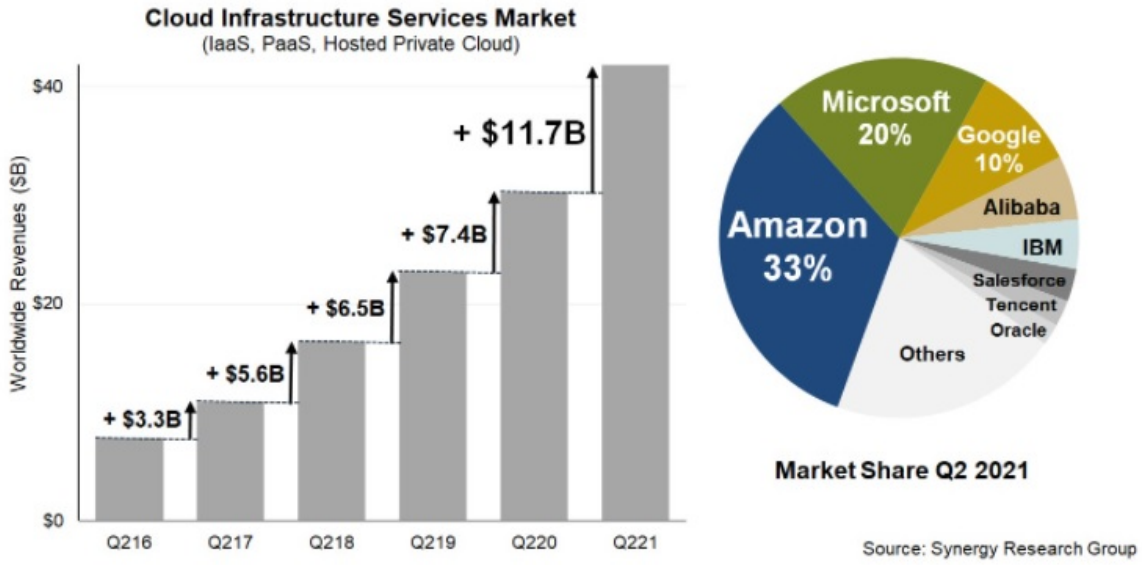

Abb. 2.2: CSP Market Share Q2 2021 nach Umsatz[8]

Aus Abb. 2.2 wird ersichtlich, dass Amazon, Microsoft und Google gemeinsam mehr als die Hälfte des Marktes beherrschen, Amazon allein stellt etwa ein Drittel aller Infrastruktur Services bereit.

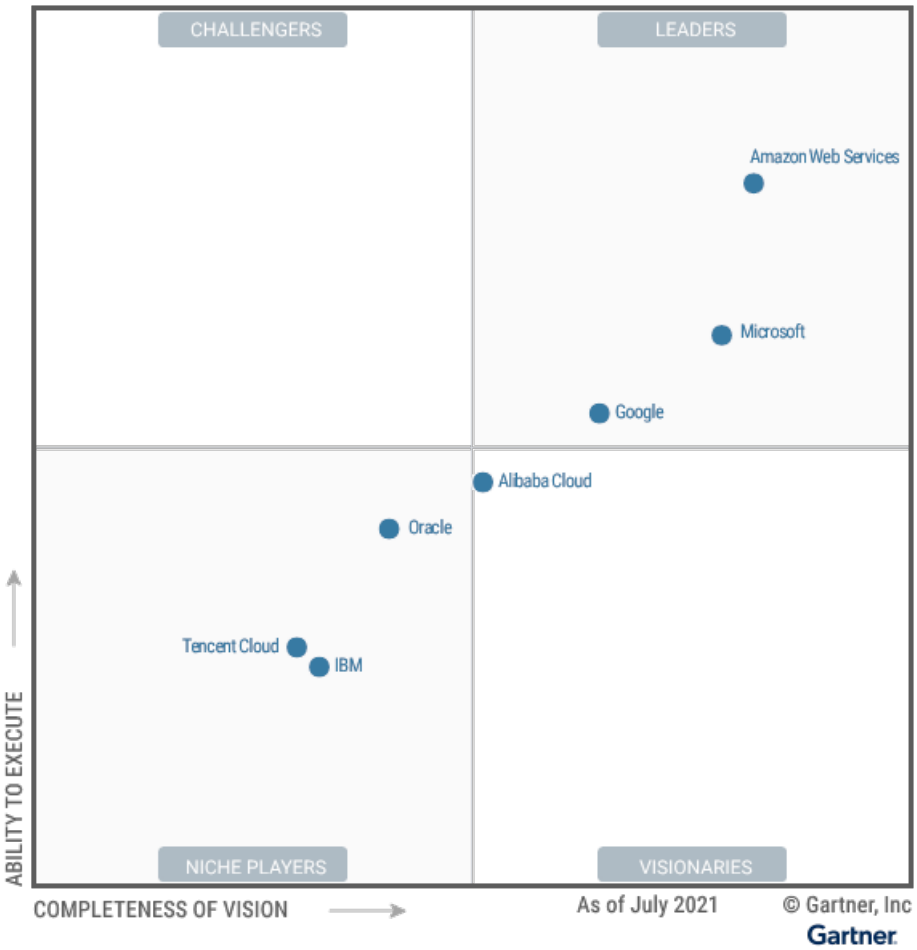

Abb. 2.3: CSP Gartner Magic Quadrant Juli 2021[9]

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 Vormachtstellung von Amazon gegenüber Microsoft und Google, auffällig ist auch die Stellung von IBM, die auf ein mögliches Problem in der Ausgereiftheit der IBM Cloud hinweisen könnte.

2.2 Infrastructure as Code

Automatisierung spielt in der modernen Softwareentwicklung eine wichtige Rolle. Automatisierte Unit- und Integration-Tests gehören inzwischen schon lange zum Standardrepertoire eines Softwareentwicklers. Die Verbreitung der DevOps-Kultur förderte in den

letzten Jahren auch die Automatisierung des Build- und Deployment-Prozesses und mit der steigenden Verbreitung Cloud-basierter Software erweitert sich diese Automatisierung nun auch auf das Provisioning der Infrastruktur.

Infrastructure as Code beschreibt im Allgemeinen einen Ansatz zur Automatisierung des Infrastruktur-Provisionings basierend auf Methoden aus der Softwareentwicklung[10, S.4].

Statt eines manuellen Aufbaus und händischer 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[11].

2.2.1 Technologischer Wandel und das Cloud Age Mindset

Durch die Technologien der Cloud ist es heute möglich eine gewünschte IT-Infrastruktur sehr viel schneller bereitzustellen als zuvor. Statt Einkauf, Anschließen und Einrichten eines physischen Servers das, je nach Szenario einen Zeitraum von mindestens mehreren Stunden oder Tagen bis hin zu Wochen dauern kann, können virtuelle Ressourcen in der Cloud in wenigen Minuten verfügbar gemacht werden. Der schnellere Ablauf wird durch die Automatisierung von Prozessen wie der Bereitstellung der Infrastruktur mithilfe von IaC Tools noch verstärkt. Mit diesen Veränderungen wird das Management und die Erweiterung der bestehenden Systeme jedoch nicht unbedingt einfacher[10, S.3], die Anwendung von Arbeitsprozesse die sich bisher bewährt haben sind oftmals ineffektiv und verhindern das Ausnutzen des vollen Potentials der Cloud. Kief Morris, Autor von *Infrastructure as Code - Dynamic Systems for the Cloud Age*, stellt die fundamentalen Unterschiede des Arbeitens mit Cloud-Technologien mithilfe der folgenden Tabelle (Tab. 2.1) 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

Tab. 2.1: „Iron vs Cloud Age“[10, S.3]

Veränderungen im „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 monolitisch, die Konfiguration erfolgt eher mithilfe von GUI-gesteuerten Programmen oder direkter physischer Interaktion, 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 das „Iron Age“ zugeschnittene Arbeitsweise für die Cloud nicht effektiv ist. Ein neues „Cloud Age Mindset“, das die rechte Spalte der Tabelle verinnerlicht ist, erforderlich um die Vorteile der Cloud wirklich effektiv und in vollem Umfang nutzen zu können.

2.2.2 Vorteile von Infrastructure as Code im Vergleich zu manuellem Infrastruktur-Provisioning

Kein Configuration Drift durch einheitliche Codebasis: 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 Umgebungen oftmals auch leicht verschiedene Anforderungen an diesen Server. Auf diese kann dann mit Optimierungen, etwa in Form spezifischer Konfigurationsdetails, reagiert werden. Wird nun das ursprüngliche Basis-System geupdated, werden individuelle und oft undokumentierte Anpassungen nicht berücksichtigt, wodurch ein Update unbequeme Konsequenzen nach sich ziehen kann. Werden alle Veränderungen in einer einheitlichen Codebasis verwaltet und Updates häufig vorgenommen, verhindert man starken Configuration Drift, der ansonsten oftmals stattfindet[10, S.17-18].

Wiederverwendbarkeit durch einheitlichen Code: Ein weiterer Vorteil, der durch die Verwendung einer einheitlichen Codebasis entsteht, ist die Wiederverwendbarkeit und Reproduzierbarkeit eines Systems. Wenn ein identisches System an einer anderen Stelle aufgebaut werden soll oder geht ein System aus unvorhergesehenen Gründen in seiner Gesamtheit verloren, 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[10, S.16].

Schnelleres Provisioning durch Cloud: Ein bereits häufig angesprochener Vorteil ist die schnelle Bereitstellung, durch diese Verkürzung bzw. Eliminierung des Hardware-Anschaffungs-Prozesses. Dadurch können sowohl frühe Developer-Builds, als auch spätere Release-Versionen schneller getestet und deployed werden, wodurch der gesamte Release-Cycle beschleunigt wird.

Schnellerer Profit: Die logische Schlussforderung aus dem vorangegangenen Punkt ist die schnellere Erzielung von Umsätzen. Den Markt als erstes mit einem neuen Produkt zu betreten, bringt wirtschaftliche Vorteile mit sich, Voraussetzung dabei ist eine adäquate Qualität, welche durch IaC ebenfalls gefördert wird.

Einheitlicheres Tooling in Dev, Ops und weiteren Beteiligten Teams: Verwendung von IaC fördert ein einheitlicheres Tooling in allen Bereichen, die mit einem Softwareprodukt in Zusammenhang stehen. Ein einfaches Beispiel hierfür ist die Verwendung eines Sourcecode Editors, der sowohl für das Schreiben von Programm- und Infrastruktur-Code gleichermaßen notwendig ist.

Stärkere Automatisierung im Arbeitsablauf: Automatisierung bedeutet immer einen gewissen initialen Zusatzaufwand, jedoch kann auf längere Sicht deutlich von automatisierten Abläufen profitiert werden. Testen, Builds und Deployment wurden bereits angesprochen und sind klassische Prozesse, deren Automatisierung längerfristig vorteilhaft ist.

Höhere Zuverlässigkeit und Sicherheit durch schnelle Updates: Veränderungen stellen die größte Gefahr für ein Produktionssystem dar, gleichzeitig sind Veränderungen aber unvermeidbar, um die Sicherheit des Systems zu gewährleisten und es zu verbessern. Durch die Fähigkeit, Veränderungen schnell und dennoch zuverlässig durchzuführen, wird die Reaktionszeit auf unvorhergesehene Ereignisse und Anforderungen wesentlich verkürzt[10, S.4-5].

Schnellere Fehlersuche und -behebung: Potentielle Fehler können durch Automatisierung und Modularität (siehe. Kapitel 2.2.4) schneller isoliert, gefunden und behoben werden. Infolgedessen profitieren Sicherheit und Stabilität von einer Struktur, die eine einfache und schnelle Integration von Veränderungen fördert.

2.2.3 Herausforderungen und Argumente gegen den Einsatz von Infrastructure as Code

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

1. „Veränderugen werden nicht oft genug durchgeführt um Automatisierung zu rechtfertigen."

Die Idee, dass ein System einmal erstellt und dann „fertig" ist, wodurch eine Automatisierung der Veränderungen überflüssig wird, entspricht sehr selten der tatsächlichen Realität. IT-Systeme und damit auch IT-Infrastruktur wird während ihres 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 zuverlässigen Betrieb zu gewährleisten. Neue Features in bestehender Software kann neue Infrastruktur, zum Beispiel in Form einer zusätzlichen Datenbank, notwendig machen oder eine veränderte Konfiguration erfordern. 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 ein System erfährt. Können die Kapazitäten automatisiert erweitert werden, wird ein Zusammenbruch des Services auf einfache Art verhindert[10, S.4-5].

„A fundamental truth of the Cloud Age is: Stablity comes from making changes."[10, S.5]

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

2. „Infrastruktur soll zuerst aufgebaut, danach automatisiert werden."

Die Umsetzung von Infrastructure as Code stellt eine durchaus große Herausforderung dar, umso mehr wenn die entsprechenden Kompetenzen noch aufgebaut werden müssen. Der Nutzen wird deshalb nicht unbedingt direkt ersichtlich wodurch, es zu Situationen kommt, in denen es attraktiv erscheint Infrastruktur zuerst aufzubauen und sich erst später um die Automatisierung zu kümmern. Mit diesem Ansatz werden viele der Vorteile von IaC jedoch verwirkt. Ein bestehendes, komplexes Infrastruktur-System zu automatisieren, ist

eine deutlich größere Herausforderung als ein System von Grund auf mit dem Gedanken an Automatisierung aufzubauen[10, S.6].

3. „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. Ein unausgeglichener Fokus auf eines der beiden Kriterien führt im Lauf der Zeit zu einem „Fragile Mess" (Abb. 2.4).

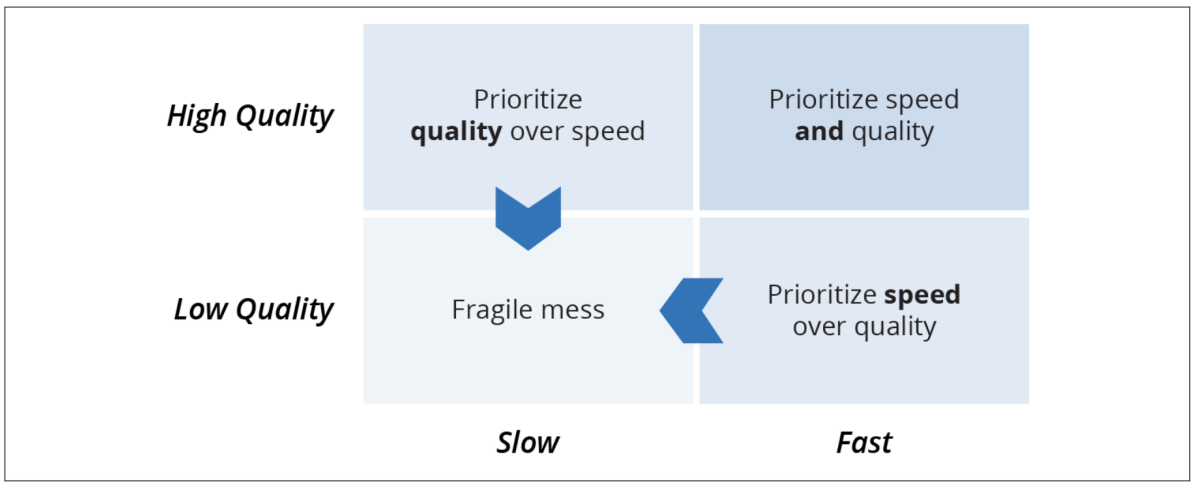

Abb. 2.4: Verhältnis von Geschwindigkeit und Qualität[10, S.8]

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 (Quadrant rechts unten), 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. Aufgrund dieser Probleme ist es wichtig, Geschwindigkeit und Qualität gleichermaßen zu priorisieren.

2.2.4 Die drei Kernverfahren von Infrastructure as Code

- **Define everything as Code**: Alle Teile eines System in Form von Code zu definieren, bringt mehrere Vorteile mit sich. Der Konfigurations-Code kann mehrfach ausgeführt werden, daher ist ein 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 vorhersehbarer, 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 entspricht und diesen damit auch dokumentiert[10, S.10].
- **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[10, S.10].
- **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 die 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 System auf eine ältere Version zurückgesetzt werden muss, um den Betrieb wieder herzustellen[10, S.11].

2.3 Funktionsprinzip und 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. Während 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 sein, Server werden hinzugefügt und abgebaut, Software wird aktualisiert und neu konfiguriert.

2.3.1 Überblick über Infrastructure as Code Tools

Infrastructre 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 eher ein Beispiel eines spezialisierten Tools. Die unten stehende Abbildung 2.5 soll einen Überblick über verschiedene IaC Tools und deren Aufgabengebiete 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.

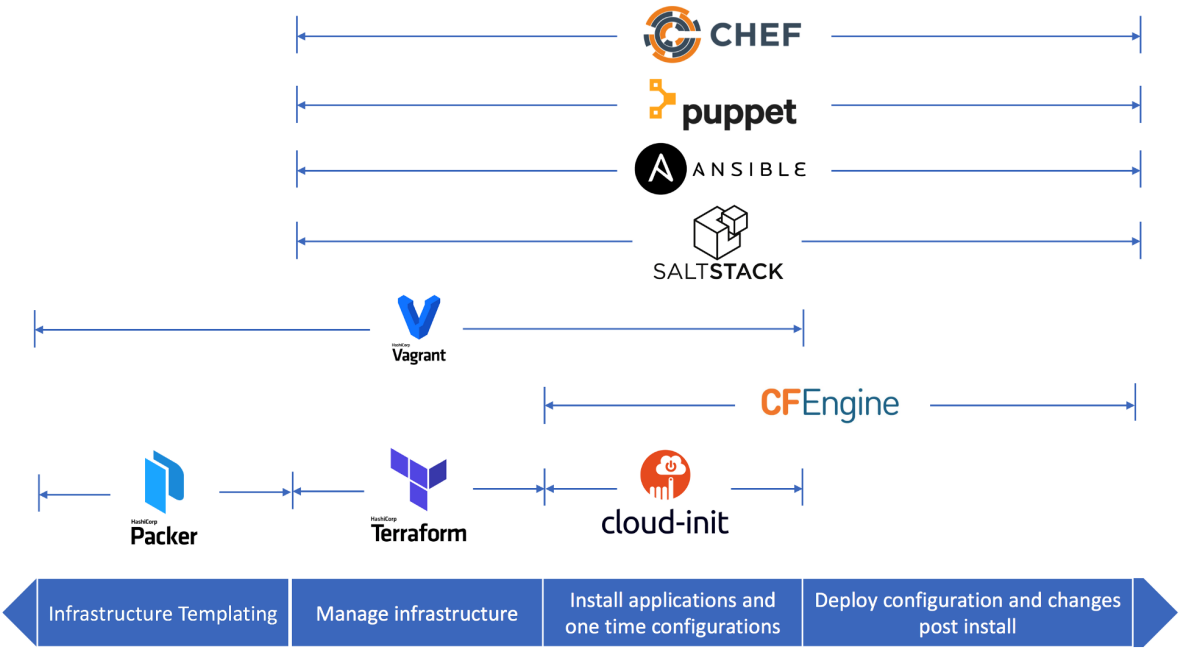

Abb. 2.5: Überblick IaC Tools[12]

Vorteile und Limiterungen 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. Neben Terraform existiert in diesem Bereich auch das Tool Pulumi, in dieser Arbeit wird Terraform jedoch aus den folgenden Gründen verwendet:

- **Modularität**: Terraform Module erlauben es ein System in mehrere klar definierte Komponenten zu strukturieren. Dadurch wird die Wiederverwendbarkeit dieser ermöglicht und gefördert, die daraus resultierenden Vorteile wurden bereits in den vorangegangenen Kapiteln angesprochen.
  Pulumi strukturiert Infrastruktur Code entweder in einem großen monolithischen Projekt oder vielen kleinen Mikroprojekten, beide Optionen sind weniger flexibel und umständlicher als die von Terraform implementierte Lösung.
- **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 Voteil 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 oder eine Liste erwartet wird.

2.3.2 Funktionsweise von Terraform

Terraform ist ein Infrastructure as Code Tool das es ermöglicht Infrastruktur sicher und effizient aufzubauen, zu verändern und zu versionieren. Die Funktionsweise von Terraform wird in Abbildung 2.6 dargestellt.

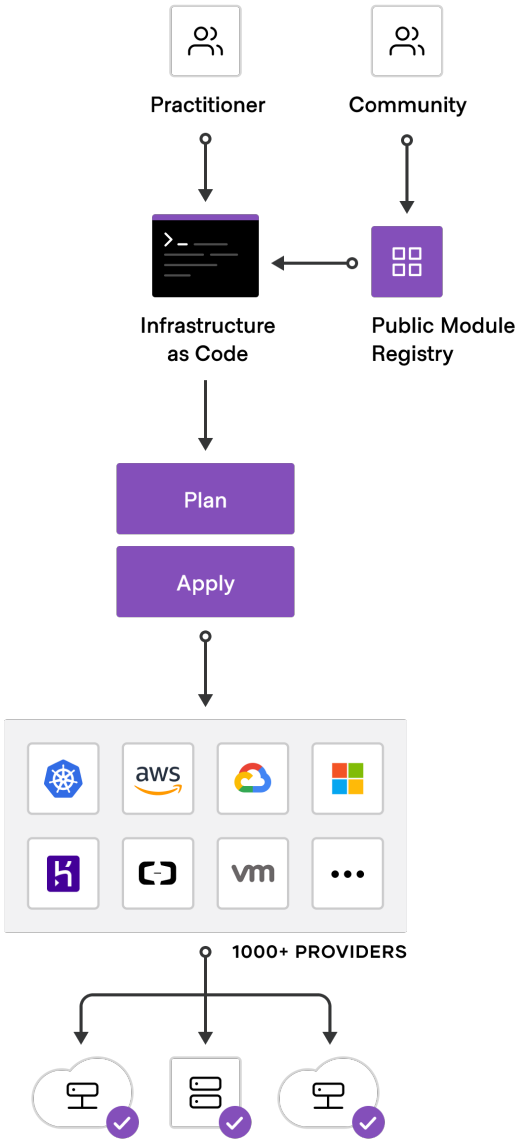

Abb. 2.6: Terraform Funktionsprinzip[13]

Terraform verwendet eine deklarative Domänenspezifische Sprache (DSL), die es erlaubt Infrastruktur in Konfigurationsdateien zu beschreiben, die versioniert, geteilt und wiederverwendet werden können. Terraform Module können auf einer Public Module Registry [1] zur allgemeinen Verfügung bereitgestellt und von der breiteren Terraform-Community wiederverwendet werden. Terraform unterstützt über Plug-Ins, die Terraform Provider, ein große Anzahl an Plattformen.

1 Vergleichbar mit der npm-Registry von Node.js

Terraform State

Das Terraform State-File ist die wichtigste Datei in einem Terraform-Projekt. Sie enthält die Informationen der aktuell existierenden und von Terraform verwalteten Ressourcen und stellt ein Mapping zwischen den Konfigurationsdateien und der realen Welt dar[14]. Zusätzlich befinden sich essentielle Metadaten und Informationen über die Abhängigkeiten der verschiedenen Ressourcen im Terraform State.

Der Terraform State kann entweder lokal oder in einem Remote Backend, zum Beispiel der Terraform Cloud, gespeichert werden. Lokaler State ist ausreichend für ein kleineres Forschungsprojekt eines einzelnen Entwicklers, für wichtige Projekte und die Kollaboration in einem Team sollte jedoch ein Remote Backend gewählt werden. Sollten zwei Entwickler gleichzeitig Änderungen an der Infrastruktur vornehmen wollen, muss ausgeschlossen werden, dass der State korrumpiert und inkonsistent wird. Sobald der erste Anpassungsvorgang begonnen wird, wird der Zugriff auf den State gesperrt und erst am Ende wieder freigegeben, auf diese Weise bleibt der State konsistent.

Terraform Workflow

Der grundlegende Terraform Workflow wird in Abbildung 2.7 dargestellt. Er besteht primär aus vier Befehlen: einem Init-Befehl zur Initialisierung des Projekts, der Erstellung des Execution Plan, Durchführung des Execution Plan und einem Destroy-Befehl zur Freigabe/Abbau aller von Terraform verwalteten Ressourcen.

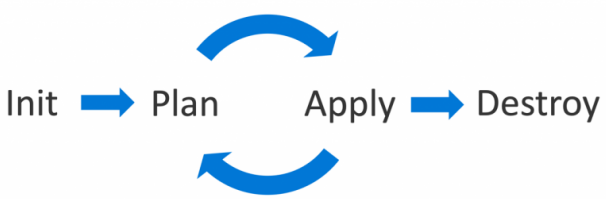

Abb. 2.7: Terraform Workflow[15]

Mit **`terraform init`** wird das Arbeitsverzeichnis initialisiert, die Provider und Module installiert sowie das Backend für den Terraform State konfiguriert.

Durch **`terraform plan`** wird der Execution Plan erstellt. Der Execution Plan definiert alle Schritte, die von Terraform durchgeführt werden, um vom aktuellen Zustand, gespeichert

im Terraform State, zum in den Konfigurationsdateien mit Terraform Code beschriebenen Zustand zu gelangen. Dieser Execution Plan kann ausgegeben und manuell reviewt werden, bevor die tatsächliche Ausführung dessen vorgenommen wird. Durch die Erstellung dieses Plans werden Abhängigkeiten betrachtet, die die Reihenfolge der Ressourcen-Erstellung bestimmen, voneinander unabhängige Ressourcen können zum Beispiel parallel erstellt werden.

Die Ausführung des Execution Plan wird durch **`terraform apply`** ausgeführt. Es gibt zwei Modi: Der **Automatic Plan Mode** erstellt automatisch einen neuen Execution Plan, der **Saved Plan Mode** verwendet einen in einer Datei gespeicherten Execution Plan.

Und zuletzt dient **`terraform destroy`** dazu alle in Terraform verwalteten Ressourcen zu „zerstören“ und damit wieder freizugeben. Dies wird in einem Produktionssystem selten Anwendung finden, während der Entwicklung und für Testprojekte ist dies jedoch extrem nützlich.

2.3.3 Einführung in die Hashicorp Configuration Language

Die von Terraform verwendete Konfigurationssprache wurde mit dem Ziel entwickelt, einen Kompromiss zwischen Maschinenfreundlichkeit und Menschenlesbarkeit zu erzielen[16]. 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 (HCL) zum Einsatz.

Grundlegende Elemente

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

```
1 blockType "label1" "label2" {
2   ...
3   /*This is a block*/
4   ...
5
6   argument = 5
7
8   arguments + can + build = expressions
```

```
}
```

List. 2.1: Grundlegende Elemente von HCL

Blöcke stellen für gewöhnlich ein Objekt, im Fall von Infrastruktur-Code meistens eine Computing Ressource, 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-Variablen 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 Input-Variablen enthält, angepasst werden, alles andere kann ohne Veränderungen wiederverwendet werden.

```
variable "project" {
  default = "some_value"
  description = "a description"
}
```

List. 2.2: Beispiel Input-Variable

Input-Variablen können entweder während der Ausführung über die Kommandozeile oder über eine Datei mit einfachen Key-Value Paaren befüllt werden.

Output-Variablen werden in der Regel verwendet, um nach dem Aufbau des Systems auf spezifische Parameter einfachen und schnellen Zugriff zu ermöglichen.

```
output "something_important" {
  value = resource-group.vm-name.ip-address
}
```

List. 2.3: Beispiel Output-Variable

Werte wie die IP einer Virtuellen Maschine auf einer Public Cloud sind oft vor der Bereitstellung dieser nicht bekannt und können erst danach ausgegeben werden. Terraform kann Output-Variablen mithilfe eines Kommandos schnell und übersichtlich ausgeben.

Terraform Module

Die Verwendung von Modulen gestaltet sich ebenfalls relativ einfach. Es wird immer eine Quelle benötigt, aus der das Modul bezogen wird, diese kann ein lokales Verzeichnis oder eine Module Registry sein. Zusätzlich benötigen die meisten Module eine Reihe von Input-Variablen wie im folgenden Listing 2.4 dargestellt.

```
module "showcase_kubernetes" {
  source = "./showcase_kubernetes"

  project   = var.project
  network   = google_compute_network.jsa-vpc_network.name
  zone      = var.zone
  num_nodes = 2
}
```

List. 2.4: Beispiel Terraform Module

Terraform Provider Konfiguration

Terraform Provider benötigen ähnlich wie Module eine Quelle und eine optionale zusätzliche Konfiguration. In Listing 2.6 wird beispielhaft der Google Cloud Platform Provider eingebunden und für ein spezifisches Projekt konfiguriert.

```
terraform {
  required_providers {
    google = {
      source  = "hashicorp/google"
      version = "4.1.0"
    }
  }

  provider "google" {^
    project = "active-woodland-324808"
```

```
11      region  = "europe-west3"
12      zone    = "europe-west3-b"
13    }
14  }
```

List. 2.5: Beispiel Terraform Provider Konfiguration

3 Evaluierungsanforderungen und Umsetzung

3.1 Evaluierungsanforderungen

3.1.1 Ziel der Evaluierung

Das Ziel dieser Evaluierung ist es, einen Vergleich zwischen den beiden Cloud Plattformen Google Cloud Platform und Microsoft Azure auf Basis von deren Unterstützung von Infrastructure as Code mit Terraform zu erstellen. Anhand der Ergebnisse sollen auch die Vorteile von Infrastructure as Code gegenüber manuellem Provisioning deutlich werden.

3.1.2 Untersuchte Komponenten der Terraform Provider

Zu diesem Zweck wird ein Beispielszenario, bestehend aus einem Kubernetes Cluster und einem Datenbankserver, für beide Plattformen in Terraform Code implementiert und auf diesen deployed. Zusätzlich wird im Vorfeld manuell eine Container Registry auf den beiden Plattformen erstellt und ein Docker Image auf dieser bereitgestellt. Diese Container Registry soll dann als Terraform Data Source im Projekt eingebunden und das darauf liegende Image via Terraform auf dem Kubernetes Cluster deployed werden.
Die Struktur, primär bestehend aus Datenbankserver und Kubernetes Cluster, ist eine einfache Repräsentation einer Infrastruktur, die für den Betrieb moderner Cloud basierter Anwendungen notwendig ist.

3.1.3 Limitierungen der Untersuchung

Primär soll damit die grundlegende Eignung der Plattform für dieses Szenario untersucht werden. Spezialisiertere Szenarien sind daher durch diesen Vergleich nicht repräsentiert,

ebenso wird die Security des Infrastruktur-Systems nur in einer minimal erforderlichen Weise umgesetzt, da sich Anforderungen an diese je nach Organisation und Anwendung stark unterscheiden können.
Weiterhin findet das Deployment des Beispielszenarios nur in einer einzelnen Availability Zone der jeweiligen Plattformen statt. Im Fall von GCP ist dies die Zone „europe-west3-b“ und bei Microsoft „westeurope“.

3.1.4 Auswahl der Evaluierungskriterien

Das Softwarequalitätsmodell der ISO 25010 wird als ein Grundstein für die Evaluierung von Software beschrieben[17]. Da die Terraform Provider statt einer ganzheitliche Software nur Plug-Ins und damit eine kleinere Komponente von Terraform darstellen, kann nicht jeder der in der ISO 25010 aufgelisteten Punkte sinnvoll angewandt 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, den man durch den Einsatz von Infrastructure as Code Tools- und Prinzipien erzielen kann.

Abb. 3.1: Überblick ISO 25010[17]

Im Folgenden sollen die einzelnen Merkmale der ISO 25010 kurz zusammengefasst und erläutert werden, ob diese in der Evaluierung der Terraform Provider von Azure und Google Cloud Platform sinnvoll betrachtet werden können.

Functional Suitability

- **Functional Completeness**: Grad der 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 Coud 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, deren Zweck es ist die Bedienung zu vereinfachen.

  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.B. während der Eingabe) schützt.

  Anwendbarkeit: Da bei der User Error Protection die selbe Lage, wie beim vorhergehenden Punkt vorliegt, wird auch diese nicht eingehender 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 trifft 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, soll dieser Aspekt nicht gezielt 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 einiger Unterbrechung oder einem Fehler betroffene Daten wiederherstellen und die Funktionalität wiederherstellen 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 Coud 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 die Diagnose 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 des 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 Testing 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 Umsetzung des Testsystems

3.2.1 Eingesetzte Software und Tools

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

Als Sourcecode Editor kommt Visual Studio Code zum Einsatz.
Die Versionierung des Sourcecodes erfolgt durch ein privates Git-Repository auf Github.
Terraform Kommandos werden in der Bash ausgeführt. Zeitmessungen werden mit dem Bash Kommando `time <command>` durchgeführt und das Ergebnis der Zeile `real` verwendet.
Die Abhängigkeitsgraphen der jeweiligen Systeme in Kapitel 3.2.3 und 3.2.4 werden mit dem in Terraform integrierten Kommando `terraform graph` erstellt und mithilfe des Tools **Terraform Graph Beautifier** visuell aufbereitet. Abb. 3.2 bzw. 3.3 stellen jeweils ein Beispiel für einen „rohen“ und den zugehörigen aufbereiteten Graphen dar.

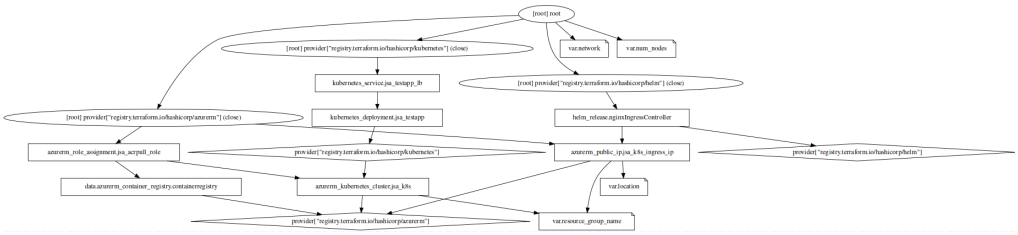

Abb. 3.2: Output Terraform Graph[Eigene Abbildung]

In der Darstellung von `terraform graph` werden sämtliche Abhängigkeiten und Elemente inklusive der Module und Provider dargestellt.

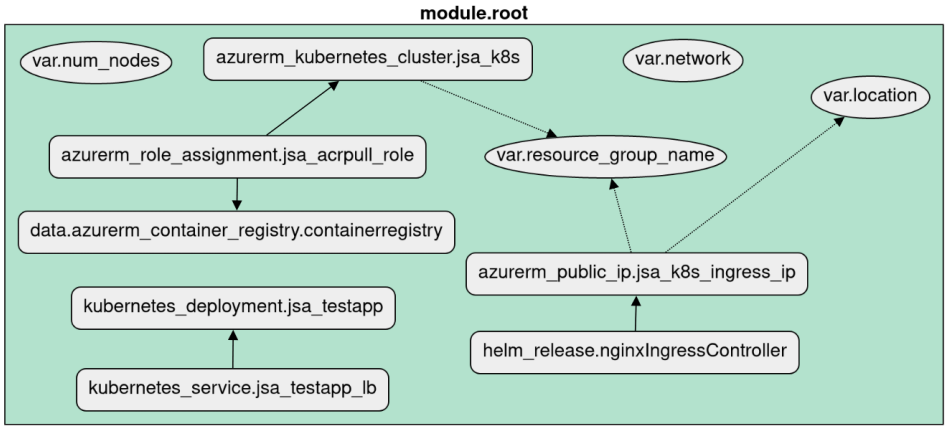

Abb. 3.3: Output Terraform Graph Beautifier[Eigene Abbildung]

Terraform Graph Beautifier versteckt einige Elemente, wie die Provider, um eine gesamtheitlich bessere Übersicht verschaffen zu können. Module werden als farblich hinterlegte Rechtecke dargestellt, innerhalb derer sich die zugehörigen Ressourcen befinden. Die generierte Datei ist dabei eine HTML-Datei, die es erlaubt alle Elemente durch Drag-and-Drop zu verschieben, um die Darstellung individuell anpassen zu können. Der Nachteil dabei ist, das dies auch erforderlich ist; die standardmäßige Ausgabe achtet nicht auf eine sinnvolle Anordnung der Elemente.

3.2.2 High-Level Aufbau des Testsystems

Als primäres Testobjekt dient ein Infrastruktur-System, 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.
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.

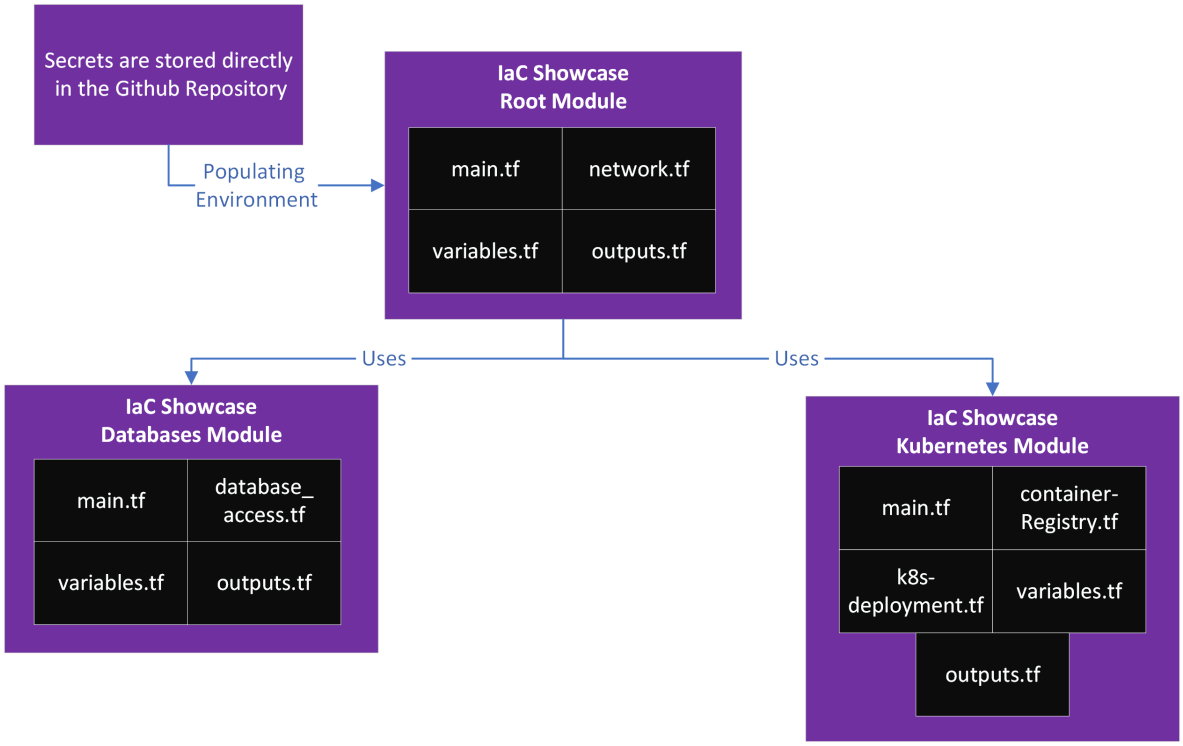

Abb. 3.4: Infrastructure Showcase High-level Struktur[Eigene Abbildung]

In der Abbildung der High-Level Struktur wird auf eine detaillierte Darstellung der Res-

sourcen verzichtet, da je nach Plattform verschiedene Funktionalitäten in unterschiedlichen Ressourcen zusammengefasst werden können. Stattdessen wird durch .tf-Dateien dargestellt, welche Funktionalität in welchem Modul vorhanden sein soll und wie der Code des Systems in Hinsicht auf Dateien aufgeteilt werden soll.

Zusätzlich wird hier die Möglichkeit eines automatisierten Deployments berücksichtigt, Voraussetzung für dieses ist das Speichern der Authentifizierungsdaten der jeweiligen Cloud Plattform im Github Repository des Terraform Codes, mithilfe von Github Actions kann dann der Terraform Workflow automatisiert werden. Da sich dieses Thema jedoch nicht im Rahmen der Forschungsfragen dieser Arbeit befindet, wurde nur ein minimales Beispiel-Deployment automatisiert, das zugehörige Listing A.1 befindet sich im Anhang dieser Arbeit und könnte mit einfachen Modifikationen auch für das Deployment der Test-Systeme verwendet werden.

3.2.3 Konkreter Aufbau in Azure

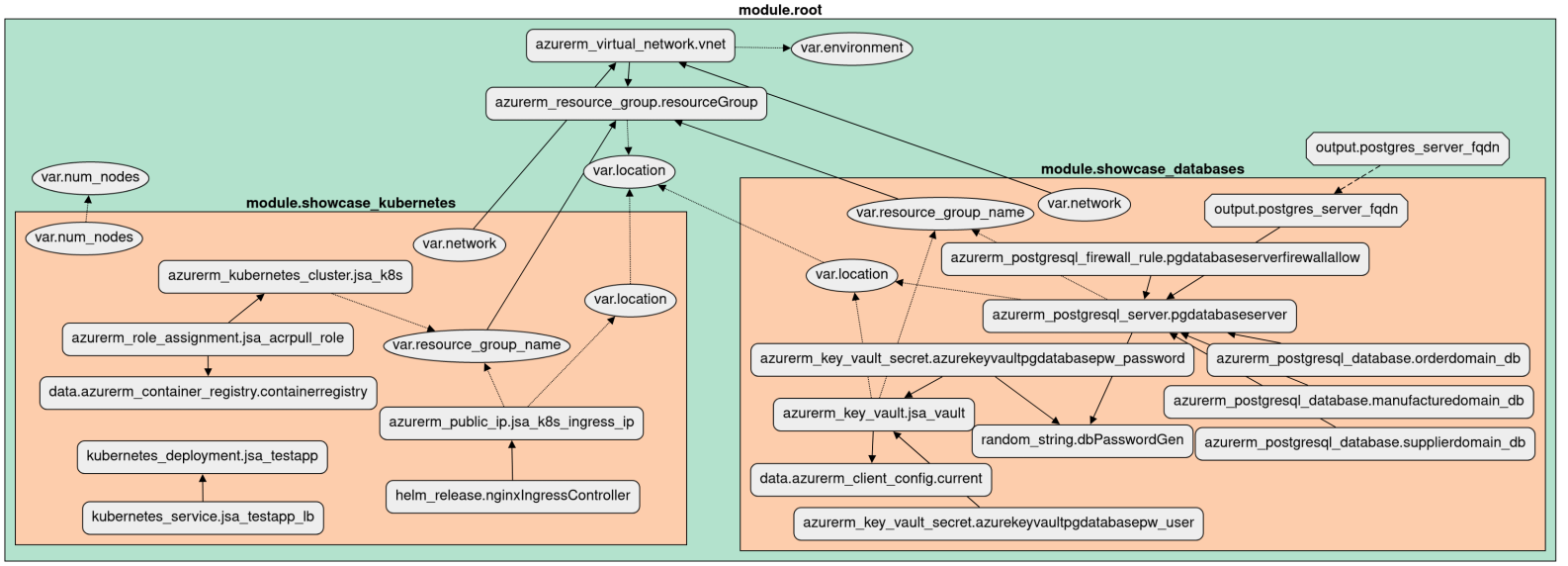

Abb. 3.5: Testsystem in Microsoft Azure[Eigene Abbildung]

Die beiden Module für den Kubernetes Cluster (`azurerm_kubernetes_cluster.jsa_k8s`) und den Datenbankserver (`azurerm_postgresql_server.pgdatabaseserver`) sowie das Root-Modul werden in Abb. 3.5 mit allen Terraform Ressourcen und deren Abhängigkeiten sowie allen Variablen und Outputs dargestellt.

Die erste Auffälligkeit dabei ist, dass nicht alle Ressourcen direkt voneinander abhängig sind. Das Kubernetes Deployment, bestehend aus der Applikation (`kubernetes_deployment.jsa_testapp`)[1] und einem Loadbalancer (`kubernetes_service.jsa_testapp_lb`), ist als Terraform Ressource zum Beispiel nicht direkt vom Kubernetes Cluster abhängig, stattdessen wird der Kubernetes Terraform Provider mit der Adresse des Clusters als Deployment-Ziel konfiguriert.

Weiterhin ist erkennbar, dass so gut wie alle Ressourcen von der `azurerm_resource_group` abhängig sind, Google Cloud Platform verwendet hier ein anderes Konzept, Näheres wird dazu im nächsten Unterkapitel beschrieben. Gleiches gilt für den Secret Storage, dieser wird hier mit der Ressource `azurerm_key_vault.jsa_vault` erstellt.

Direkter Zugriff auf den Datenbankserver ist durch dessen Fully Qualified Domain Name (`output.postgres_server_fqdn`) möglich, User und Passwort werden im Secret Storage gespeichert.

Die Container Registry wird in Form der Data Source `data.azurerm_container_registry.containerregistry` eingebunden, Berechtigungen, die notwendig, sind um das Kubernetes-Deployment durchzuführen, werden durch das `azurerm_role_assignment.jsa_acrpull_role` vergeben.

1 Die Testapp ist ein trivialer Node.js Server

3.2.4 Konkreter Aufbau in Google Cloud Platform

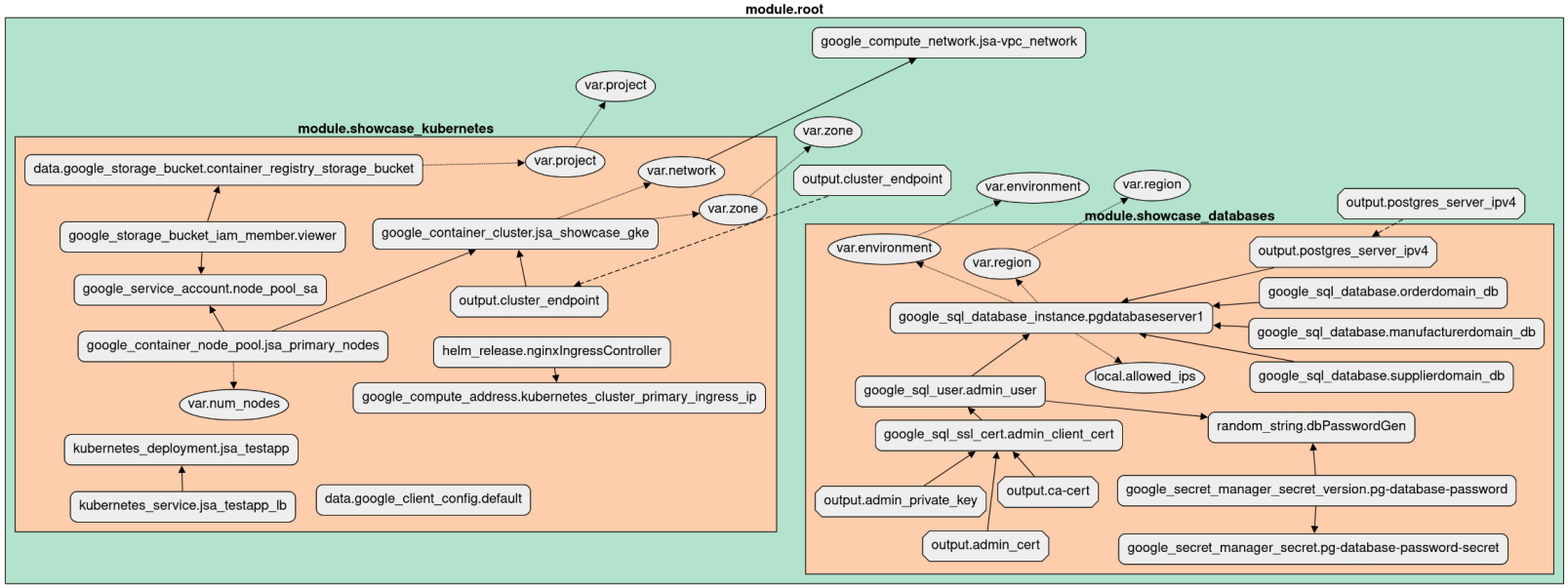

Abb. 3.6: Testsystem in Google Cloud Platform[Eigene Abbildung]

Die beiden zentralen Elemente der Module sind äquivalent zu Azure der Kubernetes Cluster `google_container_cluster.jsa_showcase_gke` und der Datenbankserver `google_sql_database_server.pgdatabaseserver1`. Anders als bei Azure existiert keine dedizierte Ressource für eine PostgreSQL-Server, die „generische“ Ressource wird per Attribut entsprechend konfiguriert.

Wie bereits im vorangegangen Unterkapitel angesprochen, existiert in diesem Aufbau für Google Cloud Platform keine zur Resourcegroup äquivalente Terraform Ressource. Stattdessen muss ein Google Cloud Platform Project mit einer einzigartige Id angelegt werden, im Anschluss wird der Google Terraform Provider für dieses Projekt konfiguriert. Gleiches gilt für den Secret Storage, dieser existiert projekt-weit und es muss lediglich der entsprechende Service im Projekt aktiviert werden.

Direkter Zugriff auf den Datenbankserver wird hier per Ip-Adresse ermöglicht (`output_postgres_server_ipv4`), anders als in Azure ist auch der Zugriff auf den Endpoint des Kubernetes Cluster sehr simpel identifizierbar.

Die Container Registry ist nicht unbedingt auf den ersten Blick als diese erkennbar. Ein einfacher Storage Bucket in Form der Data Source `data.google_storage_bucket.container_registry_storage_bucket` wird hier als Container Registry verwendet, warum dies der Fall ist, wird in der Bewertung der Functional Completeness (Kapitel 4.2) genauer erläutert.

3.2.5 Durchzuführende Messungen und Analysen

Functional Completeness

Eine essentielle Voraussetzung für den Einsatz von Terraform, ist die Vollständigkeit der Funktionalitäten, die für ein gegebenes Infrastruktur-System 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 Infrastruktur-Systems 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 Test-Infrastruktur-System
- 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 drei mal wiederholt. 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 durch das gewohnte `terraform apply` gestartet. 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
- Veränderung des Machine-Typs einer Virtuellen Maschine
- Vergrößerung eines Node Pools von ein auf zwei Nodes

4 Ergebnisse und Bewertung

4.1 Evaluierung der Einheitlichkeit der Testsysteme für Azure und Google Cloud Platform

Terraform bietet ein einheitliches Tooling und eine einheitliche Bedienung für alle von Terraform unterstützten Plattformen. Die Arbeitsprozesse und der generelle Workflow werden dadurch relativ Plattform-unabhängig, Verbesserungen in den Arbeitsabläufen können universell in allen Terraform-Projekten umgesetzt werden.
Beim Schreiben des Codes, der eigentlichen Kerntätigkeit, endet die Einheitlichkeit jedoch zum größten Teil. Der Aufbau und die Bezeichnungen aller Ressourcen hängen ausschließlich von der Implementierung des Terraform Providers und damit von der Plattform ab, die Gemeinsamkeiten liegen primär in Semantik und Syntax, welche durch die Sprache bestimmt werden.

Die wichtigste Erkenntnis ist diese: Um verschiedene Cloud Plattformen in Kombination mit Terraform einsetzen zu können, ist es weiterhin erforderlich im Umgang mit der jeweiligen Plattform geschult zu sein. Das Beherrschen von Terraform kann das Wissen über individuelle Plattformen in keiner Weise ersetzen, Terraform-Kompetenzen dienen primär dazu die Umsetzung von Infrastructure as Code zu ermöglichen.

4.2 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 `azurerm_virtual_machine` erfüllt
PostgreSQL Datenbankserver	Funktionalität wird durch Ressource `google_sql_database_instance` erfüllt	Funktionalität wird durch Ressource `azurem_postgresql_database` erfüllt
Anlegen von Datenbanken mit Charset UTF-8 und Collation English_Unites States.1252	Funktionalität wird durch die Ressource `google_sql_database` teilweise erfüllt. Collation kann bei Erstellung nicht wie erwünscht definiert werden. Default ist en_US.UTF8	Funktionalität wird durch die Ressource `azurem_postgresql_database` erfüllt. Charset und Collation werden bei Erstellung definiert
Secret Storage	Funktionalität wird durch Ressource `google_secret_manager_secret` erfüllt	Funktionalität wird durch Ressource `azurem_key_vault_secret` erfüllt
Kubernetes Cluster	Funktionalität wird durch Ressource `google_container_cluster` erfüllt	Funktionalität wird durch Ressource `azurem_kubernetes_cluster` erfüllt
Einbindung der Container Registry als Data Source	Funkltionalität wird durch Data Source `google_storage_bucket` erfüllt	Funktionalität wird durch Data Source `azurem_container_registry` erfüllt

Tab. 4.1: Anaylse der Functional Completeness

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 Projekt angelegt, Azure hingegen bietet die Möglichkeit mehrere Key Vaults innerhalb einer Resource Group anzulegen. Die Speicherung des tatsächlichen Secrets in der Ressource
`google_secret_manager_seceret_version` anstelle des zuvor anzulegenden
`google_secret_manager_seceret` fällt dadurch wenig intuitiv aus.
Die Container Registry wird in Google Cloud in Form eines normalen Google Storage Buckets angelegt. Die bevorzugte Methode zur Speicherung eines Container Images 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 verwaltet.

4.3 Ergebnisse und Bewertung der Time Behaviour Tests

		Google Cloud Platform		Microsoft Azure	
	Testlauf	Aufbau	Abbau	Aufbau	Abbau
Aufbau/Abbau Testsystem	1	15m48s	7m54s	7m7s	6m5s
	2	17m37s	8m11s	7m59	7m44s
	3	15m12s	6m52s	6m54s	6m6s
Aufbau/Abbau VM	1	0m30s	1m3s	1m12s	1m42s
	2	0m23s	1m4s	1m11s	1m37s
	3	0m25s	1m4s	1m11s	1m40s
Aufbau/Abbau PostgreSQL-Datenbankserver	1	3m59s	1m3s	2m28s	0m42s
	2	3m58s	0m52s	2m32s	0m43s
	3	3m57s	0m52s	2m28s	0m40s
Aufbau/Abbau Kubernetes- Cluster mit default Node Pool	1	3m59s	2m52s	4m36s	5m33s
	2	3m39s	3m3s	6m5s	5m32s
	3	3m31s	2m53s	5m13s	5m29s

Tab. 4.2: Ergebnisse der Time Behaviour Tests (1/2)

	Testlauf	Google Cloud Platform	Microsoft Azure
Execution Plan für Testsystem	1	1.2s	14.6s
	2	1.2s	15.2s
	3	1.6s	15.9s
Execution Plan für Datenbank-server	1	0.9s	22.2s
	2	1.0s	23.2s
	3	1.0s	23.3s
Execution Plan für Kubernetes Cluster mit default Node Pool	1	0.8s	13.5s
	2	0.9s	13.3s
	3	0.9s	12.3s

Tab. 4.3: Ergebnisse der Time Behaviour Tests (2/2)

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 Zeitdifferenz 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 Azure Location „westeurope“ 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. 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, die nicht mehr im Rahmen dieser Arbeit liegen.

4.4 Ergebnisse und Bewertung der Recoverability Tests

	Google Cloud Platform	Microsoft Azure
Längere Unterbrechung des Internets, State in Remote Backend	State korrumpiert, erneutes terraform apply führt zu Fehler, manuelles Entfernen der Ressource notwendig	State korrumpiert, 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 korrumpiert, manuelles Löschen der VM-Ressource notwendig	Terraform State korrumpiert, manuelles Löschen der VM-Ressource notwendig
Soft Cancel von Terraform apply	State korrumpiert	Shutdown sofort, state korrumpiert
Hard Cancel von Terraform apply	State korrumpiert	Nicht durchgeführt

Tab. 4.4: Ergebnisse der Recoverability Tests

Bewertung

Die Tests zur Recoverability vermitteln ein einheitliches Bild: Das Ergebnis jedes Szenarios ist die Korrumpierung 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 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 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.

4.5 Ergebnisse und Bewertung der Modifiability Tests

	Google Cloud Plattform	Microsoft Azure
Änderung des Betriebssystems einer virtuellen Maschine	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.	
Vergrößerung des default Node Pools von 1 auf 2 Nodes	Ressource wird zerstört und neu erstellt.	Ressource wird modifiziert.

Tab. 4.5: Ergebnisse der Modifiability Tests

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 Neuerstellen 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 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, das Fehlen grundlegender Funktionalitäten wurde im Rahmen dieser Evaluierung nicht festgestellt. Betrachtet man die Details, schneidet Microsoft Azure insgesamt etwas ab. Die Deployment Zeiten sind im Durchschnitt etwas schneller und es existieren einige Detail-Funktionalitäten, die von Google Cloud Platform aktuell noch nicht vollständig abgedeckt sind.

Die Dokumentation der Terraform Provider beider Cloud Plattformen fällt sehr gut und umfangreich aus, darin enthaltene Beispiele für die Konfiguration der individuellen Ressourcen erleichtern den Einstieg in Terraform bedeutend.

Weniger zufriedenstellend fällt dagegen die Einheitlichkeit der Implementierungen aus. Der geschriebene Code ist abgesehen von der generellen Struktur grundverschieden, Providerübergreifende Wiederverwendbarkeit existiert praktisch nicht. Terraform ermöglicht damit eher einen einheitlichen Workflow und die Verwendung einer einheitlichen Sprache, statt der Möglichkeit den Infrastruktur-Code einmal universell für alle Plattformen zu entwickeln. Die in Kapitel 3.2.2 erwähnte Möglichkeit der Verwendung von Github Actions oder ähnlichen Lösungen zu einer weiteren Stufe der Automatisierung, wird durch Terraform ebenfalls einheitlicher gestaltet.

5.2 Ausblick

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 wichtige 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

A.1 Github Action Terraform

```
name: 'Playground GCP CI'

on:
  push:
    paths:
      - 'playground/**'
    branches:
      - master
  pull_request:

jobs:
  terraform:
    name: 'Terraform'
    runs-on: ubuntu-latest

    defaults:
      run:
        shell: bash

    steps:
      - name: Checkout
          uses: actions/checkout@v2

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v1
        env:
          GOOGLE_CREDENTIALS: ${{ secrets.GOOGLE_CREDENTIALS }}
```

```
28
29        - name: Terraform Init
30          working-directory: ./playground/gcp
31          run: terraform init
32          env:
33            GOOGLE_CREDENTIALS: ${{ secrets.GOOGLE_CREDENTIALS }}
34
35        - name: Terraform Format
36          working-directory: ./playground/gcp
37          run: terraform fmt -check
38
39        - name: Terraform Plan
40          working-directory: ./playground/gcp
41          run: terraform plan
42          env:
43            GOOGLE_CREDENTIALS: ${{ secrets.GOOGLE_CREDENTIALS }}
44
45        - name: Terraform Apply
46          working-directory: ./playground/gcp
47          if : github.ref == 'refs/heads/master' &&
               github.event_name == 'push'
48          run: terraform apply -auto-approve
49          env:
50            GOOGLE_CREDENTIALS: ${{ secrets.GOOGLE_CREDENTIALS }}
```

List. A.1: Github Action für Automatisiertes Terraform Deployment

Literaturverzeichnis

[1] ASADI, Shahla, et al. Customers perspectives on adoption of cloud computing in banking sector. Information Technology and Management, 2017, 18. Jg., Nr. 4, S. 305-330. [Online] Verfügbar unter: https://doi.org/10.1007/s10799-016-0270-8 (Zugriff am: 23.01.2022)

[2] MELL, Peter, et al. The NIST definition of cloud computing. 2011. [Online] Verfügbar unter: http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf (Zugriff am: 28.10.2021)

[3] Chou, David. Cloud Service Models (IaaS, PaaS, SaaS) Diagram. (2018). [Online] Verfügbar unter: https://dachou.github.io/2018/09/28/cloud-service-models.html (Zugriff am: 28.10.2021)

[4] Lisdorf, Anders. Cloud computing basics a non-technical introduction. Berkeley, CA: Apress, 2021.

[5] Nandgaonkar, Suruchee V.; Raut, A. B. A comprehensive study on cloud computing. International Journal of Computer Science and Mobile Computing, 2014, 3. Jg., Nr. 4, S. 733-738. [Online] Verfügbar unter: https://d1wqtxts1xzle7.cloudfront.net/33540634/V3I4201499a71-with-cover-page-v2.pdf?Expires=1643306860&Signature=P5Q4J9nyWUI0Z1Em~hDev-I09EiyIiBLj~OrDNu9AATa4fGOvHsa~phLE3INxm6268epofaJuqxTeDo67z6ofl66tlzV5-pHUi11Ve7TnBJXtgdWZrRMP5PJ79QT84YAjLPKKnJnn-rAZtUIbhHaMew3cx2uSJ~KWj40-lLfQTWnS684WrBBdIXpwvjcFPM6pPeGghmPNp9td29p6kgNztLcW7u~~0wL-CoHYwpkx7ms~3WWIQLUOR7KDNYyjxotFB2EVeogRuqQAFEV5l516RpIb9fviHlYmrt3iw0tK7xqsOXLAgRU039L8KndyvKX5WZgrl9rjgCv6o1cafErxg__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA (Zugriff am: 27.01.2022)

[6] OPARA-MARTINS, Justice; SAHANDI, Reza; TIAN, Feng. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. Journal of Cloud Computing, 2016, 5. Jg., Nr. 1, S. 1-18. [Online] Verfügbar unter https://ieeexplore.ieee.org/abstract/document/7009018 (Zugriff am: 24.01.2022)

[7] BAI, Kun, et al. What to discover before migrating to the cloud. In: 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013). IEEE, 2013. [Online] Verfügbar unter: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6573001 (Zugriff am: 24.01.2022)

[8] Kein Autor. Enterprise spending on cloud infrastructure services grows to USD 42 billion in Q2, 2021: SRG (2021). [Online] Verfügbar unter: https://www.cioandleader.com/article/2021/07/30/enterprise-spending-cloud-infrastructure-services-grows-usd-42-billion-q2-2021 (Zugriff am: 24.01.2022)

[9] Bala, Raj; Gill, Bob; Smith, Dennis; Ji, Kevin; Wright, David. Magic Quadrant for Cloud Infrastructure and Platform Services (2021). [Online] Verfügbar unter: https://www.gartner.com/doc/reprints?id=1-271OE4VR&ct=210802&st=sb (Zugriff am: 24.01.2022)

[10] Kief Morris. Infrastructure as Code. O'Reilly Media, 2. edition, 2020.

[11] Jacobs, Mike; Kaim, Ed. What is Infrastructure as Code? (2021) [Online] Verfügbar unter: https://docs.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code (Zugriff am: 24.01.2022)

[12] Nayak, Ramnath. When to use which Infrastructure-as-code tool (2019). [Online] Verfügbar unter: https://medium.com/cloudnativeinfra/when-to-use-which-infrastructure-as-code-tool-665af289fbde (Zugriff am: 25.01.2022)

[13] Terraform Website. [Online] Verfügbar unter: https://www.terraform.io/ (Zugriff am: 25.01.2022)

[14] Terraform State Documentation. [Online] Verfügbar unter: https://www.terraform.io/language/state/purpose (Zugriff am 26.01.2022)

[15] Kumar, Rajesh. Terraform Basics Workflow loop explained!!! (2019). [Online] Verfügbar unter: https://www.devopsschool.com/blog/terraform-basics-workflow-loop-explained/ (Zugriff am: 26.01.2022)

[16] Introduction to HashiCorp Configuration Language (HCL) (2018). [Online] Verfügbar unter: https://www.linode.com/docs/guides/introduction-to-hcl/ (Zugriff am: 25.01.2022)

[17] ISO/IEC 25010. [Online] Verfügbar unter: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (Zugriff am: 26.01.2022)