HAW
HAMBURG

BACHELOR THESIS
Jonah-Myron Mensah

Evaluation von Infrastructure as Code Tools für Cloud-Infrastruktur

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jonah-Myron Mensah

Evaluation von Infrastructure as Code Tools für Cloud-Infrastruktur

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 31. August 2023

Jonah-Myron Mensah

Thema der Arbeit

Evaluation von Infrastructure as Code Tools für Cloud-Infrastruktur

Stichworte

IaC, DevOps, Infrastrukturautomatisierung

Kurzzusammenfassung

Die Entwicklung der Cloud hat die Art und Weise verändert, wie Organisationen ihre IT-Infrastruktur verwalten. Mit der Fähigkeit, Ressourcen auf Abruf bereitzustellen und zu skalieren, bietet die Cloud zahlreiche Möglichkeiten, die Anforderungen moderner Software zu erfüllen. Mit diesen Möglichkeiten kommen jedoch auch neue Herausforderungen für die Infrastrukturverwaltung auf. Manuelle Ansätze reichen oft nicht mehr aus, um die große Menge an Cloud-Ressourcen effizient zu verwalten. Infrastructure as Code bietet einen Ansatz, der eine automatisierte Verwaltung dieser Infrastruktur ermöglicht. Aufgrund der Vielzahl an verschiedenen IaC-Werkzeugen ist die Auswahl des richtigen Werkzeuges eine komplexe Aufgabe.

Diese Arbeit präsentiert einen umfassenden Vergleich der verschiedenen Werkzeuge mit dem Ziel, Einblicke in ihre Eignung für die Verwaltung von Cloud-Infrastrukturen zu liefern. Die Studie vergleicht und bewertet Terraform, Pulumi, Ansible, SaltStack, Chef und Puppet, im Hinblick auf ihre zentralen Konzepte und genutzten Paradigmen. Die Vor- und Nachteile jedes Werkzeuges werden analysiert, um fundierte Schlussfolgerungen hinsichtlich ihrer Eignung für spezifische Anwendungsfälle zu ziehen.

Insgesamt dient diese Arbeit als Leitfaden für Entscheidungsträger, die das Angebot der Infrastructure as Code Werkzeuge verstehen möchten, um informierte Entscheidungen zur effektiven Verwaltung von Cloud-Infrastrukturen treffen zu können.

Jonah-Myron Mensah

Title of Thesis

Evaluation of Infrastructure as Code tools for cloud infrastructure

Keywords

IaC, DevOps, Infrastructure automation

Abstract

The evolution of the cloud has changed the way organizations manage their IT infrastructure. With the ability to provision and scale resources on demand, the cloud offers many opportunities to meet the demands of modern software. However, with these opportunities come new challenges for infrastructure management. Manual approaches are often no longer sufficient to efficiently manage a large amount of cloud resources. Infrastructure as Code offers an approach that enables automated management of this infrastructure. Due to the large number of different IaC tools, selecting the right tool is a complex task.

This paper presents a comprehensive comparison of the different tools to provide insights into their suitability for managing cloud infrastructures. The study compares and evaluates Terraform, Pulumi, Ansible, SaltStack, Chef, and Puppet, in terms of their core concepts and used paradigms. The advantages and disadvantages of each tool are analyzed to draw informed conclusions regarding their suitability for specific use cases.

Overall, this work serves as a guide for decision-makers who want to understand the range of Infrastructure as Code tools to make informed decisions on how to effectively manage cloud infrastructures.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **viii**

1 Einleitung **1**

1.1 Problemstellung . . . 1
1.2 Ziel der Arbeit . . . 2
1.3 Methodik . . . 3
1.4 Aufbau der Arbeit . . . 4
1.5 Prototypische Implementierung . . . 5

2 Infrastructure as Code - IaC **7**

2.1 Was ist Infrastructure as Code . . . 7
2.2 Ziele von Infrastructure as Code . . . 8
2.3 Herausforderungen von dynamischer Infrastruktur . . . 9
2.4 Prinzipien von Infrastructure as Code . . . 10
2.5 Dynamische Infrastruktur-Plattform . . . 12
2.6 Typen von Infrastructure as Code . . . 13
2.7 Anwendungsfälle von Infrastructure as Code . . . 14
2.8 Herausforderungen von Infrastructure as Code . . . 15

3 Vergleich der Infrastructure as Code Werkzeuge **18**

3.1 Terraform . . . 18
3.2 Pulumi . . . 21
3.3 Ansible . . . 23
3.4 SaltStack . . . 26
3.5 Chef . . . 29
3.6 Puppet . . . 31
3.7 Zwischenergebnis des Vergleiches . . . 38

3.8 Limitierungen des Vergleiches 42

4 Evaluation **43**
4.1 Master vs. Masterless und Agent vs. Agentless 44
4.2 GPL vs. DSL 45
4.3 Imperativ vs. Deklarativ 46
4.4 Veränderlich vs. Unveränderlich 50
4.5 Community 55
4.6 Schlüsselergebnisse 56

5 Fazit und Ausblick **60**
5.1 Fazit 60
5.2 Ausblick 63

Literaturverzeichnis **64**

Glossar **68**
Selbstständigkeitserklärung 70

Abbildungsverzeichnis

3.1 Wie Pulumi funktioniert 22
3.2 Anzahl der Fragen pro Schlüsselwort auf Stack Overflow (August 2023) . . 36
3.3 Stack Overflow Trends 37
3.4 Vergleichsmatrix der Werkzeuge 38
3.5 Vergleich der Initialisierung des Kubernetes-Masters 41

4.1 Vergleich Abhängigkeiten Imperativ (a) und Deklarativ (b) 48
4.2 Vergleich Imperativ (a) und Deklarativ (b) 49
4.3 Veränderliche vs. unveränderliche Infrastrukturaktualisierung 52
4.4 Kombination aus Packer, Ansible und Terraform 58
4.5 Verschiedene Ebenen der Infrastrukturverwaltung 58

Tabellenverzeichnis

3.1 Community Aktivität (August 2023) 35

1 Einleitung

1.1 Problemstellung

In der sich stetig weiterentwickelnden IT-Welt ist es wichtig, dass Unternehmen sich agil an Marktanforderungen anpassen. Um konkurrenzfähig zu bleiben, ist es erforderlich die Reaktionsfähigkeit auf Marktveränderungen zu verbessern. Für das Nutzererlebnis der Kunden gehört es mittlerweile zum Standard, dass Software reaktiv ist, möglichst keine Latenz hat, hochverfügbar ist und sich schnell weiterentwickelt. Um diese Kundenanforderungen zu ermöglichen, setzen immer mehr Unternehmen auf die Cloud.

Die Cloud hat eine grundlegende Veränderung in der Entwicklung, Bereitstellung und Nutzung von Software bewirkt. Während traditionelle Software an physische Hardware und lokale Infrastrukturen gebunden war, was zu Einschränkungen im Hinblick auf die Ressourcennutzung und Skalierbarkeit führte, ermöglicht die Cloud eine flexible Nutzung von Ressourcen. Dies ermöglicht die Anpassung an die Anforderungen moderner Softwareentwicklung.

Durch diese neu gewonnen Möglichkeiten, ergeben sich zugleich neue Herausforderungen. Die Gestaltung komplexer Softwarearchitekturen wie zum Beispiel Microservices erfordert eine Integration mit teilweise ebenso komplexen Cloud-Architekturen.

Cloud-Infrastruktur setzt sich aus einer Vielzahl von Komponenten zusammen, darunter Load Balancer, virtuelle Netzwerke, Router, Firewalls, Netzwerklaufwerke, virtuelle Maschinen, Cloud-Funktionen und weitere Dienste. Die Verwaltung dieser Komponenten gestaltet sich aufgrund ihrer Menge oft als ineffizient oder sogar unmöglich, wenn manuell vorgegangen wird.

In der Cloud wird Hardware wie Software behandelt, was zu einer Änderung der Sichtweise auf die Verwaltung von Hardware führt. Während die Verwaltung von Infrastruktur

früher zu den Aufgaben des Operations-Teams gehörte, wird heutzutage auch ein Softwarehintergrund benötigt. Daher ist für Cloud-Infrastruktur eine enge Zusammenarbeit zwischen dem Entwicklungs- und Operations-Team nötig. Diese Anforderungen hat zur Entstehung des DevOps-Konzepts beigetragen.

„DevOps ist eine Kombination von Denkweisen, Praktiken und Werkzeugen, mit denen Unternehmen schneller und einfacher Anwendungen und Services bereitstellen können. Die Weiterentwicklung und Verbesserung von Produkten gelingt damit in kürzerer Zeit als bei Unternehmen, die auf herkömmliche Prozesse für die Softwareentwicklung und Infrastrukturverwaltung setzen. Dank dieses Geschwindigkeitsvorteils können Unternehmen ihre Kunden besser bedienen und sich effektiver auf dem Markt durchsetzen“[39].

Eine der zentralen Aufgaben des DevOps-Teams ist die Infrastrukturverwaltung und deren Automatisierung. Infrastructure as Code ist für eine effiziente Bereitstellung und Orchestrierung von Cloud-Infrastruktur essenziell. Das vielfältige Angebot an Infrastrukturwerkzeugen macht die Auswahl des richtigen Werkzeuges jedoch zu einem schwierigen Prozess. Es gibt zahlreiche Faktoren, die den Entscheidungsprozess beeinflussen, da die unterschiedlichen Werkzeuge mit verschiedenen Paradigmen arbeiten.

1.2 Ziel der Arbeit

Diese Arbeit beschäftigt sich mit der Fragestellung, wie eine gezielte Auswahl geeigneter Werkzeuge erfolgen kann und welche maßgeblichen Evaluierungsaspekte bei dieser Wahl zu berücksichtigen sind. Das Ziel ist dabei, mittels eines umfassenden Vergleiches verschiedener Infrastrukturwerkzeuge, deren individuelle Stärken und Schwächen im Kontext der Cloud-Infrastrukturverwaltung zu identifizieren.

Die Arbeit soll dabei ein tiefgreifendes Verständnis für die Auswahl und den effektiven Einsatz von Infrastructure as Code Werkzeugen vermitteln. Dies wird nicht nur durch die Betrachtung von Vor- und Nachteilen der Werkzeuge erreicht, sondern auch durch eine Analyse ihrer Eignung für verschiedene Anwendungsszenarien. Dabei werden relevante Evaluierungsaspekte beleuchtet, die von zentraler Bedeutung sind, um eine fundierte Entscheidung im Kontext der Werkzeugauswahl treffen zu können.

Insgesamt strebt diese Arbeit danach, nicht nur eine umfassende Darstellung der Vor- und Nachteile verschiedener IaC-Werkzeuge zu bieten, sondern auch ein tieferes Verständnis für die kritischen Aspekte der Werkzeugauswahl zu vermitteln. Dieser Ansatz wird dazu

beitragen, dass zukünftige Entscheidungen im Hinblick auf Infrastrukturverwaltung in der Cloud auf einer soliden Basis getroffen werden können.

1.3 Methodik

Der folgende Abschnitt beschreibt den Ansatz, der zur Bearbeitung der Forschungsfrage verfolgt wurde. Zu Beginn wurde eine umfassende Literaturrecherche durchgeführt, um eine Grundlage an theoretischem Wissen zu schaffen. Hierbei wurden vor allem die offiziellen Dokumentationen der Werkzeuge genutzt, da sie den aktuellen Stand am besten wiedergeben. Durch diese Literaturrecherche wurden nicht nur die zu vergleichenden Werkzeuge identifiziert, sondern auch die relevanten Evaluierungsaspekte herausgearbeitet. Zudem wurden die unterschiedlichen Paradigmen der Werkzeuge analysiert und die daraus resultierenden Möglichkeiten und Herausforderungen bei der Infrastrukturverwaltung ermittelt.

Anschließend wurde auf Basis der Literaturrecherche eine prototypische Implementierung durchgeführt, um die in der Theorie erarbeiteten Erkenntnisse zu evaluieren. Der Aufbau der prototypischen Implementierung wird in Abschnitt 1.5 besprochen. Die Implementierung dient dazu, Erkenntnisse zu bestätigen beziehungsweise besser nachvollziehen zu können. Dadurch wurden praxisnahe Einsichten über verschiedene Aspekte der Werkzeuge gewonnen.

Die Kombination aus Literaturrecherche und prototypischer Implementierung erlaubt es, eine ganzheitliche Analyse der betrachteten Infrastrukturwerkzeuge durchzuführen. Während die Literatur die theoretische Grundlage bildet, um die Evaluierungsaspekte zu identifizieren, bietet die prototypische Implementierung die Möglichkeit, diese Aspekte anhand eines praktischen Beispiels zu bewerten. Der Vorteil an dieser Vorgehensweise ist, dass so ein Vergleich der Theorie und Praxis durchgeführt werden kann. Aufgrund der schnellen Weiterentwicklung der Werkzeuge ist dieser Vergleich hilfreich, um eventuelle Inkonsistenzen zwischen Literatur und Praxis aufzuzeigen. Dadurch soll ein umfassendes Verständnis der Werkzeuge sowie ihrer Eignung für verschiedene Anwendungsfälle geschaffen werden.

Aufgrund der Vielzahl an möglichen Vergleichskriterien wurde die Literaturrecherche hauptsächlich auf die nachfolgend aufgeführten Kriterien beschränkt:

- Installation und Konfiguration

- Kommunikation und Arbeitsmodus
- Funktionsweise
- Terminologie und Konzepte
- Community

Auswahl der Kriterien

Die Auswahl dieser Kriterien erfolgte aufgrund der Möglichkeit, aus ihnen weitere Informationen abzuleiten. Anhand des Installationsprozesses und der grundlegenden Funktionsweise lassen sich Rückschlüsse auf die Benutzerfreundlichkeit, Topologie, Skalierbarkeit, Wiederverwendbarkeit, Konfigurationssprache und andere Aspekte ziehen.

Da das Ziel dieser Arbeit darin besteht, einen umfassenden Überblick zu vermitteln, wurden bewusst einige Kriterien, die für die Evaluierung und Auswahl von Werkzeugen von Bedeutung sind, nicht in Betracht gezogen. Diese Entscheidung wurde getroffen, da die umfassende Bewertung dieser Kriterien einen erheblichen Aufwand erfordert, der nicht innerhalb des begrenzten Rahmens dieser Arbeit realisierbar ist. Zu diesen Kriterien gehören Leistungsfähigkeit, Sicherheit, Stabilität und Reife der Werkzeuge.

Ausgewählte Werkzeuge

Die Auswahl der Werkzeuge erfolgte anhand der Literaturrecherche. Während dieser wird deutlich, dass bestimmte Werkzeuge sich als weitverbreitete Lösungen etabliert haben. Dazu zählen: Terraform, Pulumi, Ansible, SaltStack, Chef und Puppet.

1.4 Aufbau der Arbeit

Im folgenden Abschnitt wird die Struktur dieser Arbeit beschrieben. In Kapitel 2 werden die grundlegenden Konzepte der Infrastrukturverwaltung sowie Infrastructure as Code erörtert. Kapitel 3 stellt zunächst die ausgewählten Werkzeuge vor und analysiert diese hinsichtlich der oben genannten Evaluierungskriterien. Anschließend erfolgt ein Vergleich der Werkzeuge. In Kapitel 4 werden die unterschiedlichen Funktionsweisen und Paradigmen der Werkzeuge erläutert. Des Weiteren wird untersucht, wie diese Paradigmen die

Infrastrukturverwaltung beeinflussen. Anhand der Informationen des Vergleiches sowie der Erläuterungen der verschiedenen Paradigmen wird schließlich eine Empfehlung für eine effektive Infrastrukturverwaltung gegeben. Kapitel 5 bildet den Abschluss dieser Arbeit, indem es die zentralen Erkenntnisse zusammenfasst und einen Ausblick auf mögliche wissenschaftliche Forschungsbereiche im Kontext von Infrastructure as Code bietet.

1.5 Prototypische Implementierung

Dieser Abschnitt beschreibt den Aufbau der prototypischen Implementierung. Das Ziel der Implementierung ist die praktische Umsetzung einer praxisnahen Infrastruktur. Aufgrund der Anzahl der Werkzeuge, mit denen diese Infrastruktur implementiert wird, ist das Szenario einfach gehalten. Das Fallbeispiel ist ein gängiger Anwendungsfall für Infrastrukturverwaltung und umfasst wiederkehrende Aufgaben wie die Installation von Paketen und das Starten und Stoppen von Diensten, die so auch in einer realen Umgebung vorkommen.

Anforderungen an die Lösung

Im folgenden werden die Anforderungen an die Infrastruktur beschrieben. Die zu erstellende Infrastruktur sollte leicht nachvollziehbar sein und bewusst auf spezielle Sonderfälle verzichten. Eine weitere wichtige Anforderung ist die Vermeidung eines Provider-Lock-Ins. Dadurch wird sichergestellt, dass die Lösung mit minimalem Aufwand auf verschiedene Cloud-Provider übertragen werden kann. Diese Anforderung ist aufgrund der Auswahl der genutzten Werkzeuge automatisch erfüllt.

Folgende Schritte sind zum Erreichen des Zielzustandes erforderlich:

1. Zwei virtuelle Maschinen provisionieren
2. Docker installieren
    - Abhängigkeiten installieren (apt-Pakete)
    - docker.io apt-Paket installieren
    - Docker über dameon-Datei konfigurieren
    - Sicherstellen, dass Service mit korrekter Konfiguration gestartet wurde

3. Kubernetes installieren
4. Kubernetes-Master initialisieren
5. Kubernetes-Worker zum Cluster hinzufügen

Dadurch wird eine minimale Infrastruktur bereitgestellt, auf der unter Verwendung von Containern Applikationen gestartet werden können.

2 Infrastructure as Code - IaC

Das folgende Kapitel ist eine Einführung in das Thema Infrastructure as Code. Es behandelt die grundlegenden Konzepte, Prinzipien und Ansätze, die essenziell sind um die Möglichkeiten und Herausforderungen der automatisierten Infrastrukturverwaltung zu verstehen. Ziel dieses Kapitels ist es, eine solide Basis für den Vergleich und die anschließende Evaluierung der Infrastrukturwerkzeuge zu schaffen.

2.1 Was ist Infrastructure as Code

Infrastructure as Code, abgekürzt IaC, ist ein Teilgebiet aus dem Bereich DevOps und wird zum automatisierten Verwalten von IT-Infrastruktur genutzt. Durch die Kombination aus verschiedenen Werkzeugen, Skripten und Praktiken, wird der zu erreichende Zustand der Infrastruktur in Form von Code beschrieben.

Um das Verständnis für den Begriff zu ergänzen, ist es hilfreich, folgende Definition zu betrachten: „Infrastructure as code is an approach to infrastructure automation based on practices from software development. It emphasizes consistent, repeatable routines for provisioning and changing systems and their configuration"[9, S. 5].

In der Cloud wird Hardware ähnlich wie Software und Daten behandelt, weshalb Cloud-Infrastruktur durch Infrastructure as Code automatisiert verwaltet werden kann. Diese Sichtweise eröffnet die Möglichkeit, die aus der Softwareentwicklung gewonnene Erfahrung auf die Infrastrukturverwaltung zu übertragen. Dadurch lassen sich Konzepte wie Versionskontrolle (VCS), Continous Integration (CI) und Continous Delivery (CD) für die Infrastrukturverwaltung nutzen.

Infrastructure as Code Konzepte können sowohl auf virtualisierte als auch physische Hardware angewendet werden. Obwohl der Fokus dieser Arbeit auf der Verwaltung von Cloud-Infrastruktur liegt, lassen sich einige der Konzepte und Erkenntnisse auch auf die Verwaltung von physischer Hardware übertragen.

2.2 Ziele von Infrastructure as Code

Seit den Anfängen netzwerkbasierter Systeme ist das Bereitstellen, Konfigurieren und Verwalten von Servern eine große Herausforderung. Hardware aufrüsten, Betriebssysteme warten, Software installieren, Sicherheitslücken schließen und Konfigurationen anpassen, sind Aufgaben, die früher manuell von Systemadministratoren oder dem Operations-Team durchgeführt wurden.

Die Anzahl der zu verwaltenden Server, unabhängig davon, ob diese in der Cloud, oder physisch vorliegen, ist heutzutage oft sehr groß. Mit der zunehmenden Nutzung von Microservices zur Zusammensetzung von Applikationen, benötigen IT-Unternehmen eine große, zuverlässige und sichere Infrastruktur. Oft umfasst diese eine Vielzahl von Ressourcen, deren manuelle Verwaltung eine komplexe Aufgabe darstellt und mit hohem Kostenaufwand verbunden ist.

Cloud-Computing ist eine Möglichkeit, die modernen Herausforderungen der IT zu bewältigen. Unternehmen wechseln von eigen betriebener Infrastruktur in die Cloud und nutzen das Angebot an verschiedenen Diensten. Zu diesen Diensten gehören unter anderem Infrastructure as a Service (IaaS), Platform as a Service (PaaS) und Software as a Service (SaaS). Infrastructure as Code hilft dabei, die Menge an genutzten Diensten in Form von Code zu verwalten.

Die Ziele, die durch die Nutzung von Infrastructure as Code angestrebt werden, lassen sich in wirtschaftliche und technische Ziele unterteilen, wobei sich die technischen Ziele aus den wirtschaftlichen ergeben. Aus einem wirtschaftlichen Kontext, geht es vor allem darum Geschäftsprozesse zu optimieren, um so Kosten zu minimieren und dadurch den Gewinn zu maximieren. Manuelle Prozesse werden ersetzt um eine höhere Geschwindigkeit und Konsistenz in der Entwicklung und Auslieferung von Produkten oder Leistungen zu erreichen.

Die technischen Ziele, die durch die Nutzung von Infrastructure as Code angestrebt werden, sind vielfältig. Reproduzierbarkeit, Hochverfügbarkeit, Skalierbarkeit und Ausfallsicherheit gehören mittlerweile zum Standard, wenn es darum geht, ein optimales Nutzererlebnis sicherzustellen. Des Weiteren wirkt sich Infrastructure as Code auf die Wartbarkeit eines Projektes aus, da der Quellcode für die Infrastruktur als *Single source of truth* gilt und eine Dokumentation nahezu vollständig ersetzen kann.

2.3 Herausforderungen von dynamischer Infrastruktur

Im nachfolgenden Abschnitt werden die Herausforderungen beschrieben, die mit dynamischer Infrastruktur einhergehen. Diese Herausforderungen können durch den Einsatz von Infrastructure as Code bewältigt werden.

Server Sprawl Durch die Nutzung von Cloud-Diensten in Kombination mit Infrastrukturwerkzeugen, entsteht oftmals eine große Anzahl an virtualisierten Servern in der Cloud. Wenn das Aufsetzen dieser Ressourcen unkoordiniert und ohne Überwachung durch eine zentrale Instanz stattfindet, entsteht *Server Sprawl*. Einige der Konsequenzen sind:

- Ineffizienz aufgrund von Unterauslastung von Servern, wenn zum Beispiel die benötigte Rechenleistung dieser Ressourcen nicht im Vorfeld geplant wurde
- Erhöhte Kosten durch Anzahl der Server
- Sicherheitsrisiken, da sich mit einer wachsenden Zahl von Servern, auch die potenzielle Angriffsfläche vergrößert
- Erhöhte Komplexität beim Verwalten der Infrastruktur und einhalten der Konsistenz zwischen den Servern

Unterschiede in Versionen und Konfigurationen der Server, können dazu führen, dass die darauf laufende Software auf einigen Maschinen korrekt funktioniert und auf anderen nicht. Diese Inkonsistenz zwischen den Servern nennt sich *Configuration Drift*.

Configuration Drift Configuration Drift wird ausgelöst durch inkonsistente Konfigurationen zwischen Servern, vor allem wenn Änderungen ad hoc durchgeführt werden und diese nicht dokumentiert werden.

Snowflake Server Snowflake Server sind Server, die eine auf ihren Anwendungsfall spezialisierte Konfiguration besitzen, die nicht mehr nachvollziehbar ist. Wenn diese Server nun ersetzt werden müssen, gibt es keine Informationen darüber, wie genau er zu konfigurieren ist. Damit so ein Server repliziert werden kann, muss also erst einmal herausgefunden werden, wie dieser konfiguriert war. In einer automatisierten Infrastrukturkonfiguration sind alle Änderungen, die manuell, also außerhalb der Ausführung eines Skriptes oder Werkzeuges stattfinden, Schritte die einen Server zum Snowflake Server machen.

Fragile Infrastruktur „A fragile infrastructure is easily disrupted and not easily fixed. This is the snowflake server problem expanded to an entire portfolio of systems“[9, S. 8]. Derartige Infrastruktur kann schwerwiegende Auswirkungen für Unternehmen haben, die auf die Infrastruktur angewiesen sind. Durch die schrittweise Umstellung der gesamten Infrastruktur auf eine zuverlässige und reproduzierbare Architektur wird diesem Problem entgegengewirkt.

2.4 Prinzipien von Infrastructure as Code

Im Folgenden werden die Prinzipien von Infrastructure as Code nach Morris [9] vorgestellt. Diese Prinzipien helfen dabei, die zuvor beschriebenen Herausforderungen von dynamischer Infrastruktur zu bewältigen.

Systeme können einfach reproduziert werden

„It should be possible to effortlessly and reliably rebuild any element of an infrastructure“[9, S. 10] Softwareversionen, Abhängigkeiten, Netzwerkeinstellungen und weitere relevante Konfigurationen werden in Skripten festgehalten, damit neue Systeme schnell, einfach und zuverlässig aufgesetzt werden können. So lassen sich Änderungen ohne Angst vor Problemen durchführen. Auf Fehler kann durch das Ersetzen von fehlerhaften Servern schnell reagiert werden.

Systeme sind wegwerfbar

Systeme können auf einfache Weise erstellt, verändert, bewegt oder auch zerstört werden. Die Fähigkeit, auf Änderungen im System einzugehen, hilft dabei moderne Systeme stabiler zu machen. Gerade für größere Systeme ist es wichtig, Änderungen und Verbesserungen am laufenden System vorzunehmen. Moderne Software wird häufig so konzipiert, dass sie auch bei Veränderungen oder einem Ausfall der Infrastruktur nicht zwangsläufig zum Ausfall der Software führt. Diese Fehlertoleranz wird insbesondere durch die Verwendung dynamischer Infrastruktur ermöglicht.

Systeme sind konsistent

Infrastructure as Code ermöglicht das Aufsetzen nahezu identischer Systeme mit geringem Aufwand. Bei Betrachtung der Skalierung von Software ist es naheliegend, dass Softwarekomponenten, die repliziert werden, in einer identischen Umgebung laufen sollten. Ohne Konsistenz ist das Eintreten des erwarteten Verhaltens eines

Systems nicht gewährleistet. Es gibt zwei Ansätze, wie Fälle behandelt werden, in denen Systeme leichte Abweichungen voneinander haben müssen:

- Änderungen möglichst für alle Systeme durchführen
- Systeme in verschiedene logische Einheiten einteilen. Beispiel: *L*-Dateiserver und *XL*-Dateiserver

Das Problem von Configuration Drift kann mit diesen Vorgehensweisen bewältigt werden.

Prozesse sind wiederholbar

Um das Prinzip von konsistenten Systemen einhalten zu können, ist es wichtig, dass Prozesse wiederholbar sind. Durch Nutzung von Infrastructure as Code im Gegensatz dazu Änderungen manuell durchzuführen, kann dies gewährleistet werden. Selbst wenn die Änderungen nicht an jedem System vorgenommen werden müssen, ist es sinnvoll, sie im Code festzuhalten, falls weitere Systeme die gleichen Änderungen benötigen. Auch bei kleineren Änderungen lohnt es sich Konfigurationsskripte anzufertigen. Größere Probleme werden nach dem Teile und Herrsche Prinzip in Teilprobleme zerlegt und ebenfalls im Code festgehalten. Dieses Vorgehen ist ein wichtiger Faktor, für das Erreichen einer automatisierten Infrastruktur.

Design verändert sich stetig

Beim Entwerfen von Systemen wird im Gegensatz zur Vergangenheit daran gedacht, dass Systeme sich sehr schnell verändern können. Systeme sollten an verschiedene Anforderungen anpassbar sein. Hierbei sollte jedoch darauf geachtet werden, dass Systeme nicht zu komplex entworfen werden, da sich das negativ auf die Wartbarkeit auswirkt. Häufige Änderungen sollten möglich sein und werden durch Nutzung der Cloud ermöglicht. Je häufiger ein System verändert wird, desto mehr Erfahrung wird im Bereich der Infrastrukturverwaltung gesammelt, was wiederum zu ausgereifteren Systemen führt. Hierbei kann auf Erfahrungen der Softwareentwicklung oder auch anderen Bereichen der IT zurückgegriffen werden.

2.5 Dynamische Infrastruktur-Plattform

„A dynamic infrastructure platform is a system that provides computing resources, particularly servers, storage, and networking, in a way that they can be programmatically allocated and managed“[9, S. 21]. Sie ist eines der wichtigsten Konzepte hinsichtlich der Verwaltung von Cloud-Infrastruktur. Zu Beginn des Kapitels wurde erwähnt, dass Hardware in der Cloud Software und Daten sind. Die dynamische Infrastruktur-Plattform bietet die Implementierung dieses Konzepts.

Es gibt drei Anforderungen an eine dynamische Infrastruktur-Plattform:

Programmierbar: Damit die Infrastructure as Code Werkzeuge mit der Plattform interagieren können, wird eine Schnittstelle benötigt. Diese wird in Form von einer API bereitgestellt [9].

On-Demand: Die Plattform sollte eine sofortige Erstellung oder Zerstörung der Ressourcen ermöglichen. Daraus folgt, dass die Abrechnung dieser Ressourcen flexibel sein muss. Während klassische Abrechnungsmodelle auf langfristigen Verpflichtungen basierten, bieten dynamische Infrastruktur-Plattformen flexible Abrechnungsmodelle, bei denen pro Stunde oder auch pro transferiertem Gigabyte an Daten abgerechnet wird [9].

Self-service: Zusätzlich zur Möglichkeit, Ressourcen innerhalb von Minuten bis Sekunden anzufordern, muss es möglich sein, die Ressourcen den eigenen Anforderungen anzupassen [9].

Dynamische Infrastruktur-Plattformen stellen folgende Ressourcen zur Verfügung:

- **Rechenleistung:** Virtuelle Maschinen, virtuelle Server, Cluster, Cloud-Funktionen (Serverless)
- **Netzwerke:** DNS Einträge, Netzwerkrouten, Load Balancer, API Gateways, Proxys, VPNs, Netzwerkadressen, Firewall Regeln
- **Speicher:** Virtuelle Speicher Volumen, Objektspeicher, Netzwerkdateisysteme, Datenbanken

2.6 Typen von Infrastructure as Code

1. **Shell-Skripte** sind die ersten Ansätze von Infrastructure as Code. Oftmals werden diese Skripte jedoch für spezielle Anwendungsfälle konzipiert und eignen sich damit nicht für eine allgemeine Automatisierung von Infrastruktur. Es ist möglich, die Skripte so zu entwerfen, dass sie den Anforderungen von dynamischer Infrastruktur entsprechen, dies ist jedoch mit einem hohen Aufwand verbunden, da Shell-Skripte keine Mechanismen bieten, um wichtige Konzepte wie zum Beispiel Idempotenz zu erreichen.

2. **Konfigurationsmanagementwerkzeuge** beschäftigen sich hauptsächlich mit dem Konfigurieren und Instandhalten von Infrastruktur. Sie übernehmen die Einrichtung von Systemen, das Ausliefern von Software und die Installation erforderlicher Abhängigkeiten. Des weiteren nehmen sie Netzwerk- und Sicherheitseinstellungen vor und sind für die Überwachung von Systemen sowie deren Fehlerbehebung zuständig. Konfigurationsmanagementwerkzeuge können sowohl in der Cloud als auch auf physischen Systemen eingesetzt werden.

3. **Provisionierungswerkzeuge** oder auch Orchestrierungswerkzeuge finden ihre Anwendung im Bereich der Cloud. Sie übernehmen das automatisierte Hochfahren von neuen Systemen und auch das Herunterfahren und Zerstören dieser Systeme. Provisionierungswerkzeuge kommunizieren mit der API von Cloud-Providern, um verschiedene Ressourcen zu verwalten. Sie können zusätzlich auch genutzt werden, um lokale Ressourcen wie virtuelle Maschinen zu verwalten.

4. **Server-Vorlagen-Werkzeuge** automatisieren den Prozess der Erstellung und Verwaltung von Serverkonfigurationen. Die Konfigurationen werden dabei in Maschinenabbildern festgehalten. Diese Abbilder können dann für eine einfache Replizierung von konsistenten Servern genutzt werden.

Einige Cloud-Provider bieten eigene Lösungen für die Konfiguration und Provisionierung ihrer Ressourcen an. Diese Lösungen stellen oftmals eine einfache Nutzung in den Vordergrund, was es Softwareentwicklern ohne bestehende Systemadministrations- oder DevOps Kenntnisse ermöglicht, diese Werkzeuge effizient zu nutzen. Der Nachteil dabei ist die Bindung an die Infrastruktur des Anbieters, was eine Migration oftmals aufwändig macht. Dem gegenüber stehen die in dieser Arbeit behandelten Werkzeuge, welche unabhängig von Cloud-Providern sind.

2.7 Anwendungsfälle von Infrastructure as Code

Infrastructure as Code kann für verschiedene Anwendungsfälle genutzt werden. Einige der häufigeren Anwendungsfälle werden im Folgenden vorgestellt:

Provisionierung und Verwaltung von Cloud-Ressourcen Die Cloud ermöglicht eine flexible Nutzung von verschiedenen Ressourcen und Diensten. Diese lassen sich mit Provisionierungswerkzeugen erstellen, verwalten und zerstören.

Notfall-Wiederherstellung Wenn die Prinzipien von Infrastructure as Code befolgt werden, kann nahezu ohne Verzögerung auf Fehlerfälle reagiert werden. Ganze Server können auf Bedarf zerstört und neu hochgefahren werden. Die Infrastruktur ist einfach replizierbar und kann daher schnell wiederhergestellt werden.

Verwaltung von Infrastruktur Die Verwaltung von Infrastruktur umfasst Aufgaben der Systemadministration. Das Installieren und Aktualisieren von Software, Konfigurieren von Nutzern oder Netzwerkeinstellungen und auch das Beheben von Sicherheitsrisiken zählt zu den Aufgaben der Infrastrukturverwaltung.

Skalieren von Infrastruktur Infrastructure as Code ermöglicht es, Infrastruktur auf Bedarf zu skalieren. Werden zu Spitzenzeiten mehr Kapazitäten benötigt, um ein positives Nutzererlebnis zu gewährleisten, können Cloud-Instanzen provisioniert werden, um diesen Bedarf abzudecken. Sobald das System merkt, dass die zusätzlichen Kapazitäten nicht mehr benötigt werden, kann es diese ohne das Eingreifen eines Administrators wieder zerstören. Diese Optimierungsmöglichkeit senkt die Kosten für die Betreiber der Infrastruktur durch effiziente Auslastung der Ressourcen.

Mehrfach-Umgebungs-Entwicklung Durch die Verwendung von mehreren Umgebungen kann gewährleistet werden, dass die Software richtig getestet ist, bevor sie den Nutzern bereitgestellt wird. Eine für die Softwareentwicklung übliche Umgebungskonfiguration besteht aus einer Entwicklungs-, Staging- und einer Produktions-Umgebung. Durch den Einsatz von Infrastructure as Code können Umgebungen ohne großen Aufwand repliziert und somit konsistent aufgesetzt werden.

Hardware einrichten Außerhalb des Cloud-Kontextes ist die Einrichtung von physischer Hardware eine große Herausforderung. Auch hier kann Infrastructure as Code verwendet werden. So lassen sich verschiedene Hardware-Komponenten, wie Server und Netzwerkgeräte einrichten.

Zu den oben genannten häufigen Anwendungsfällen gibt es noch einige speziellere Anwendungsfälle:

Entwicklungsumgebung einrichten Der Onboarding-Prozess innerhalb von Softwareprojekten beinhaltet die Einrichtung von Entwicklungsumgebungen. Mithilfe von Infrastructure as Code können Entwicklungsumgebungen automatisiert eingerichtet und repliziert werden. Dadurch wird eine konsistente Entwicklung sichergestellt.

Verwaltung von IoT-Geräten Im Kontext von *Internet of Things* wird oft eine große Anzahl von Geräten eingesetzt. Diese Geräte kommen häufig an Orten zum Einsatz, an denen physischer Zugriff schwierig oder unmöglich ist. Zusätzlich kann die manuelle Konfiguration der IoT-Geräte eine erhebliche Menge Zeit in Anspruch nehmen. Aus diesen Gründen erweist sich der Einsatz von Infrastructure as Code als äußerst vorteilhaft, da die Geräte automatisiert aus der Ferne eingerichtet und verwaltet werden können.

Wie zu sehen ist, gibt es verschiedenste Anwendungsbereiche für Infrastructure as Code. Diese reichen von wiederkehrenden Routineaufgaben bis zu spezialisierten Anwendungsfällen.

2.8 Herausforderungen von Infrastructure as Code

Die Entwicklung von IaC-Konfigurationen bringt einige Herausforderungen mit sich. Abhängig von den Anforderungen ist das Erlernen neuer Technologien wie Werkzeugen oder Programmiersprachen erforderlich. Dies beansprucht zusätzliche Ressourcen und kann mit hohem Aufwand verbunden sein. Insbesondere zu Beginn der Einführung von IaC kann es schwierig sein, sich an empfohlene Praktiken zu halten und Änderungen an der Infrastruktur ausschließlich im Code umzusetzen, anstatt manuelle Änderungen vorzunehmen. Die Einhaltung der Prinzipien von IaC erfordert daher eine Veränderung der Denkweise in Bezug auf die Verwaltung von Infrastrukturen.

Testen von Infrastruktur

Bei einer Studie über die Herausforderungen von IaC, gaben zahlreiche befragte Nutzer an, dass das Testen eine der größten Herausforderungen bei der Entwickelung von IaC ist.

„practitioners clearly perceive it very difficult to test infrastructure code“[2] Die Befragten führten das Fehlen einheitlicher Test-Frameworks und speziell für IaC entwickelter Entwicklungswerkzeuge als Grund dafür an. „the lack of an IDE specifically designed to support the development, operation, and maintenance of infrastructure code is perceived as a challenge“[2]

Das Testen von Cloud-Infrastruktur ist ein wichtiger Bestandteil des IaC-Entwicklungsprozesses. Hierdurch kann die Leistungsfähigkeit, Stabilität und Skalierbarkeit einer Infrastruktur überprüft werden. Die Netzwerkebene ist ebenso ein kritischer Aspekt der Cloud-Infrastruktur und erfordert Tests, um die reibungslose Funktionsweise des Gesamtsystems zu gewährleisten.

Das Testen einzelner Komponenten einer Infrastruktur kann isoliert durchgeführt werden. Es ist vergleichsweise einfach festzustellen, ob eine virtuelle Maschine gestartet wurde, die Firewall korrekt konfiguriert ist, die erforderliche Umgebung für eine Applikation bereitgestellt wurde und die Anwendung über das Internet erreichbar ist. Jedoch reicht ein isolierter Test oft nicht aus, insbesondere bei großen und komplexen Infrastrukturen. Schwierigkeiten können beim Integrationstest auftreten, da die Infrastruktur oft im Gesamtzusammenhang betrachtet werden muss. Je nach Art der angestrebten Infrastruktur und dem Verwendungszweck kann es notwendig sein, die gesamte Infrastruktur von Anfang bis Ende aufzubauen und zu testen, um sicherzustellen, dass alle Komponenten korrekt funktionieren. Dieser Prozess kann zeitaufwändig sein und in jedem Schritt können Fehler auftreten, die den Prozess von vorn beginnen lassen.

Bei der Betrachtung einer veränderlichen Infrastruktur müssen möglicherweise verschiedene Ausgangszustände berücksichtigt werden, die in den Zielzustand überführt werden müssen. Die Komplexität solcher Tests steigt proportional mit der Größe und Komplexität der Infrastruktur an.

Sicherheit

Eine weitere Herausforderung von Infrastructure as Code ist das Minimieren von Sicherheitsrisiken. Hierfür ist eine umsichtige Verwaltung von vertraulichen Informationen und Zugriffsrechten erforderlich. Da der Infrastruktur-Quellcode für gewöhnlich in Git-Repositorys liegt, müssen sensible Informationen separat verwaltet werden. Zur sicheren Speicherung dieser Informationen werden oftmals Schlüssel-Werte-Speicher verwendet.

Einige Infrastrukturwerkzeuge integrieren derartige Speicher, während andere zusätzliche Lösungen erfordern.

Einführung von IaC

Die Migration von klassisch betriebener Infrastruktur hin zu einer mit IaC verwalteten Infrastruktur stellt für Organisationen eine große Herausforderung dar. Zu Beginn des Migrationsprozesses ist es von entscheidender Bedeutung, den Ist-zustand so präzise wie möglich zu dokumentieren, um einen umfassenden Überblick über die bestehende Infrastruktur zu erhalten. Die Ermittlung des Ist-Zustands gestaltet sich oft als anspruchsvolle Aufgabe. Historisch gewachsene Infrastrukturen können Änderungen aufweisen, die nicht mehr nachvollzogen werden können. Es ist nicht unüblich, dass die Schritte, die zur Erreichung des aktuellen Zustandes erforderlich waren, nicht vollständig dokumentiert wurden. Bei einer solchen Migration können zudem Kompatibilitätsprobleme auftreten, etwa aufgrund von Unterschieden zwischen alten und neuen Systemen. Infolgedessen bietet sich bei der Ausarbeitung des Soll-Zustandes die Möglichkeit zur erneuten Überprüfung, ob sich die Anforderungen geändert haben oder ob es mittlerweile effizientere Lösungen für spezifische Teile der Infrastruktur gibt. Danach kann der Soll-Zustand in Form von Quellcode beschrieben werden. Dieser Prozess kann anfangs unproduktiv erscheinen, da einige Schritte zunächst als zusätzlicher Aufwand wirken. Dennoch werden die Vorteile, die IaC mit sich bringt, insbesondere auf lange Sicht spürbar.

3 Vergleich der Infrastructure as Code Werkzeuge

Nachdem im vorherigen Kapitel eine grundlegende Wissensbasis geschaffen wurde, liegt der Fokus dieses Kapitels darauf, durch einen Vergleich der ausgewählten Werkzeuge eine fundierte Entscheidungsgrundlage zu bilden. Dabei werden die Stärken und Schwächen der jeweiligen Werkzeuge erkannt, um auf Basis dieses Wissens eine umfassende Evaluation durchzuführen.

Da innerhalb dieser Arbeit nicht alle möglichen Kriterien von Infrastrukturwerkzeugen betrachteten werden können, beschränkt sich der Vergleich auf folgende Kriterien:

- Installation und Konfiguration
- Kommunikation und Arbeitsmodus
- Funktionsweise
- Terminologie und Konzepte
- Community

3.1 Terraform

Terraform, entwickelt von HashiCorp, ist ein Provisionierungswerkzeug, das sich auf die Bereitstellung von Cloud-Infrastruktur spezialisiert hat. Es bietet Unterstützung für eine Vielzahl von Cloud-Providern.

Installation und Konfiguration

Terraform kann über den Paketmanager des jeweiligen Betriebssystems installiert werden. Die unterstützten Betriebssysteme sind macOS, Windows und Linux. Auf Linux Systemen muss der HashiCorp GPG-Schlüssel installiert werden. Alternativ kann eine einzelne Terraform-Binärdatei heruntergeladen werden, die je nach Betriebssystem zum `PATH` hinzugefügt werden muss [5].

Bevor Terraform genutzt werden kann, müssen die gewünschten Cloud-Provider konfiguriert werden. Diese Konfiguration ist wichtig, damit Terraform mit der API des jeweiligen Providers kommunizieren kann. Die Konfiguration wird im Root-Modul einer Terraform-Konfiguration platziert. Die Provider-Konfiguration wird durch die Erstellung eines *provider*-Blocks realisiert. In den meisten Fällen ist eine Form der Authentifizierung gegenüber dem Cloud-Provider erforderlich, die oft durch die Angabe eines Tokens erfolgt. Die meisten Provider können dieses Token aus Umgebungsvariablen extrahieren [3].

Nachdem der Provider konfiguriert wurde, kann dieser durch den Aufruf `terraform init` heruntergeladen werden. Anschließend kann Terraform verwendet werden, um die Ressourcen der konfigurierten Provider zu verwalten.

Kommunikation und Arbeitsmodus

Terraform kommuniziert mit der API der jeweiligen Cloud-Provider. Jede Operation ist der Aufruf eines bestimmten Endpunktes für eine Ressource. Terraform nutzt den Push-Ansatz zur Verteilung der Konfigurationen.

Funktionsweise

Terraform erstellt und verwaltet Ressourcen mithilfe von Providern. Diese ermöglichen die Interaktion mit verschiedenen Plattformen oder Diensten, die eine API bereitstellen. Der Terraform Workflow besteht hauptsächlich aus drei Schritten [6]:

Write Dieser Schritt beinhaltet das Erstellen von Konfigurationen, um Ressourcen und Dienste von einem oder mehreren Cloud-Providern zu verwalten.

Plan Terraform erstellt einen Ausführungsplan. Dieser Plan beschreibt die Infrastruktur, die basierend auf der vorhandenen Infrastruktur und der Konfiguration erstellt, aktualisiert oder zerstört werden würde.

Apply Nach Bestätigung des Nutzers führt Terraform die im Plan-Schritt vorgeschlagenen Operationen in der korrekten Reihenfolge aus. Dabei werden eventuelle Ressourcenabhängigkeiten berücksichtigt.

Terraform verwendet eine Zustandsdatei *(terraform.tfstate)* um zu bestimmen, welche Änderungen erforderlich sind, um den gewünschten Zustand zu erreichen. Diese Datei wird vor der Durchführung einer Änderungen aktualisiert. Um Metadaten zu verfolgen und die Leistung zu optimieren, insbesondere bei großen Infrastrukturen, in denen mehrere Benutzer an einem Terraform Projekt arbeiten, sollte diese Datei synchronisiert werden, um Inkonsistenzen zu vermeiden. Es bietet sich außerdem an die Datei für erhöhte Sicherheit zu verschlüsseln [8].

Konfigurationssprache

Terraform nutzt die domänenspezifische Sprache HashiCorp Configuration Language, kurz HCL.

Terminologie und Konzepte

Provider Von Terraform oder der Community geschriebenes Plugin um Ressourcen und Dienste von verschiedenen Cloud-Providern zu verwalten. Provider stellen eine Sammlung an verfügbaren Ressourcen, auf Grundlage von API-Aufrufen an einen Cloud-Provider bereit [7]. Provider stehen sowohl für IaaS- als auch für PaaS- und SaaS-Anbieter zur Verfügung.

Configuration Eine Terraform-Konfiguration ist Code, der in HCL geschrieben ist und den gewünschten Zustand der Infrastruktur beschreibt [4].

Module Ist eine eigenständige Sammlung von Konfigurationen, die eine Gruppe verwandter Ressourcen verwaltet [4].

State Zwischengespeicherte Informationen über die verwaltete Infrastruktur und Konfiguration. Die State-Datei beschreibt den aktuellen Zustand der Infrastruktur. Sie ist ein Mapping zwischen den tatsächlichen Ressourcen und der Konfiguration [4].

Registry Die Terraform Registry ist eine öffentliche Sammlung von wiederverwendbaren Providern, Modulen und Bibliotheken.

3.2 Pulumi

Pulumi ist ein Open-Source Provisionierungswerkzeug. Es wird für die Bereitstellung und Verwaltung von Cloud-Infrastruktur genutzt. Pulumi bietet Unterstützung für eine Vielzahl von Cloud-Providern.

Installation und Konfiguration

Pulumi kann über einen Paketmanager, des jeweiligen Betriebssystems installiert werden. Die unterstützten Betriebssysteme sind macOS, Windows und Linux. Zusätzlich existiert ein Installationsskript, dass auf Unix Systemen verfügbar ist. Je nachdem welche Programmiersprache mit Pulumi genutzt werden soll, muss die jeweilige Laufzeit dieser Sprache installiert werden. Standardmäßig wird *Pulumi Cloud*[1] verwendet, um den Zustand der Infrastruktur zu verwalten. Optional kann der Zustand auch in einem manuell verwalteten Objektspeicher oder auf dem lokalen Dateisystem verwaltet werden. Wenn die Pulumi Cloud genutzt wird, muss ein Login über den Befehl `pulumi login` erfolgen. Der Vorteil des Cloud-Backends ist, dass es keine weitere Konfiguration benötigt.

Kommunikation und Arbeitsmodus

Pulumi kommuniziert mit der API der jeweiligen Cloud-Provider. Jede Operation ist der Aufruf eines bestimmten Endpunktes für eine Ressource. Pulumi nutzt den Push-Ansatz zur Verteilung der Konfigurationen.

[1] Pulumi Cloud ist ein Cloud-Dienst und verwaltet den Bereitstellungsstatus und vertrauliche Informationen, führt Deployments aus und setzt Richtlinien und Zugriffskontrollen durch.

Funktionsweise

Pulumi nutzt für die Verwaltung von Infrastruktur ein Modell, bei dem der gewünschte Zustand definiert wird. Ein Pulumi-Programm wird von einem *language host* ausgeführt, um den gewünschten Zustand für die Infrastruktur zu berechnen. Die *Deployment Engine* gleicht den gewünschten Zustand, mit dem aktuellen Zustand ab und ermittelt so, welche Ressourcen erstellt, aktualisiert oder gelöscht werden müssen. Die Engine verwendet eine Reihe von Ressourcenanbietern, um die einzelnen Ressourcen zu verwalten. Während des Betriebs aktualisiert die Engine den Zustand der Infrastruktur mit Informationen über alle Ressourcen, die bereitgestellt wurden, sowie allen ausstehenden Operationen [25].

Abbildung 3.1 zeigt das Zusammenspiel dieser Komponenten:

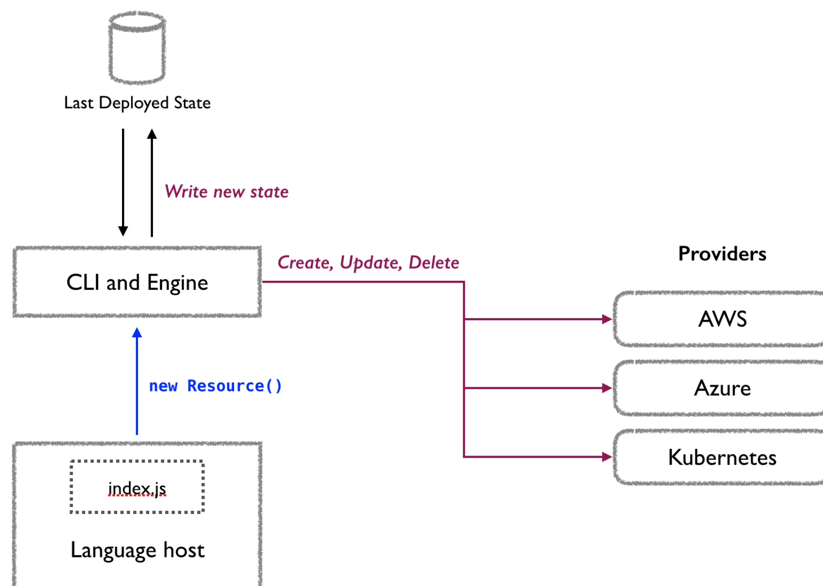

Abbildung 3.1: Wie Pulumi funktioniert

Source: https://www.pulumi.com/docs/concepts/how-pulumi-works/

Konfigurationssprache

Ein auffälliges Merkmal, das Pulumi von anderen Werkzeugen unterscheidet, besteht darin, dass Pulumi eine vielfältige Auswahl an Programmiersprachen unterstützt.

Dies ermöglicht Entwicklern, Konfigurationen in ihrer bevorzugten Sprache zu verfassen, ohne eine neue Sprache lernen zu müssen. Zusätzlich können sämtliche Konzepte und Werkzeuge, wie zum Beispiel Klassen, Pakete, integrierte Entwicklungsumgebungen, Debugger und Testwerkzeuge dieser Programmiersprache für die Konfiguration der Infrastruktur genutzt werden [26].

Terminologie und Konzepte

Ressource Provider Übernimmt die Kommunikation mit einem Cloud-Dienst, um die im Pulumi-Programm definierten Ressourcen zu erstellen, lesen, aktualisieren und löschen. Besteht aus Resource Plugin und SDK [27].

Program Pulumi-Programme die den gewünschten Zustand des Systems beschreiben. Sie können beliebige Pakete verwenden, die vom Paketmanager der Sprache unterstützt werden [27].

Component Ist eine logische Gruppe von Ressourcen. Komponenten sind eine nützliche Abstraktion, die Implementierungsdetails umschließt [27].

State Eine eigene Kopie des aktuellen Zustands der Infrastruktur. State hilft Pulumi dabei zu wissen, wann und wie Cloud-Ressourcen erstellt, gelesen, gelöscht oder aktualisiert werden sollen [27].

Pulumi Registry Ein zentrales Repository zum Teilen von Infrastrukturcode und Komponenten die mit Pulumi erstellt wurden.

3.3 Ansible

Ansible ist ein von Red Hat entwickeltes, Open-Source Konfigurationsmanagementwerkzeug. Es kann Systeme konfigurieren und komplexe Arbeitsabläufe für Softwarebereitstellung und Systemaktualisierungen orchestrieren. Ansible ist in Python geschrieben und wird hauptsächlich über die Kommandozeile genutzt. Der Fokus von Ansible liegt auf Einfachheit, Benutzerfreundlichkeit, Sicherheit und Zuverlässigkeit.

Installation und Konfiguration

Die Installation von Ansible ist unkompliziert. Auf dem Host-System, muss abhängig von der Ansible-Version, eine passende Version von Python vorhanden sein. Anschließend kann Ansible entweder über Pip installiert werden oder alternativ über den Paketmanager des verwendeten Betriebssystems, sofern Ansible dort verfügbar ist. Eine besondere Eigenschaft von Ansible ist die *agentless* Funktionalität, das bedeutet, es läuft ohne zusätzlichem Agenten auf den zu verwaltenden Knoten. Beim ersten Ausführen eines Playbooks gegenüber einem Knoten werden automatisch alle erforderlichen Abhängigkeiten gemäß den Anweisungen des Host-Systems installiert.

Kommunikation und Arbeitsmodus

Die Kommunikation bei Ansible erfolgt über das SSH-Protokoll. Standardmäßig geht Ansible davon aus, dass SSH-Schlüssel verwendet werden, um sich mit den zu verwaltenden Knoten zu verbinden. Daher hängt die Sicherheit von den Richtlinien und Methoden der Schlüsselverteilung ab. Obwohl die Verwendung von SSH-Schlüsseln die empfohlene Vorgehensweise ist, ist auch eine Authentifizierung über Passwörter möglich.

Ansible arbeitet mit dem Push-Ansatz. Der verwaltende Host sendet dabei die Instruktionen an den zu verwaltenden Knoten. Zusätzlich dazu, kann Ansible über *ansible-pull* auch im Pull-Modus agieren. Dafür muss Ansible auf dem Zielsystem installiert sein. Dann kann sich der zu verwaltende Knoten, eine gewünschte Konfiguration aus einem Git-Repository ziehen. In Kombination mit einem Cron-Job kann so ein regelmäßiges und eigenständiges Konfigurieren der zu verwaltenden Knoten automatisiert werden.

Funktionsweise

Ansible verbindet sich auf die zu verwaltenden Knoten und überträgt kleine Programme, sogenannte Ansible-Module. Diese Programme beschreiben den zu erreichenden Zustand des Systems. Ansible führt diese Module über SSH aus und entfernt sie wieder, nachdem sie durchgelaufen sind. Wenn es möglich ist, sind die Module so entworfen, dass sie Idempotent sind und Änderungen nur dann vornehmen, wenn diese nötig sind [31].

Ansible definiert die zu verwaltenden Knoten in einer Konfigurationsdatei, dem Inventory. Die zu verwaltenden Knoten können gruppiert werden, um Befehle gegen eine Teilmenge

```
[k8s-master]
192.168.100.1

[k8s-worker]
192.168.35.140
192.168.35.150

[db-server]
192.168.100.2
```

Listing 1: Statisches Ansible Inventory

dieser Gruppe auszuführen [30]. Ein Beispiel für ein statisches Inventar wird in Listing 1 gezeigt. Für die Verwaltung von Cloud-Infrastrukturen, in denen die zu verwaltenden Knoten regelmäßig hoch- und heruntergefahren werden und somit regelmäßig neue IP-Adressen zugewiesen bekommen, ist dieser statische Ansatz jedoch nicht geeignet. Aus diesem Grund bietet Ansible die Möglichkeit, ein dynamisches Inventar zu nutzen. Dieses kann Informationen über zu verwaltende Ressourcen aus verschiedenen Quellen verfolgen [33].

Sensitive Daten wie Passwörter, Schlüssel oder Benutzernamen, die innerhalb des Konfigurationsmanagements genutzt werden, können durch Ansible-Vault gesichert werden. Ansible-Vault nutzt symmetrische Verschlüsselung (AES-256).

Einige Cloud-Provider bieten außerdem die Möglichkeit, Ansible Module lokal laufen zu lassen und direkt mit der API der Cloud oder des Services zu kommunizieren [29].

Konfigurationssprache

Ansible-Playbooks werden in YAML geschrieben. Zusätzlich unterstützt Ansible Jinja2 als Templating-Sprache. Sowohl YAML als auch Jinja2 sind gut lesbar und leicht zu schreiben. Dadurch ist ein schneller Einstieg in die Nutzung möglich.

Terminologie und Konzepte

Playbooks Organisierte Sammlung von Skripten, welche die Serverkonfiguration definiert. Beschreiben eine Reihe von Schritten, die auf dem remote System ausgeführt

werden soll. Playbooks sind die Sprache, mit der Ansible Systeme orchestriert, konfiguriert, verwaltet oder bereitstellt [28].

Roles Sind Organisationsstrukturen. Sie fassen Playbooks, Dateien, Vorlagen und Variablen zusammen [28].

Facts Fakten und Informationen über einen Knoten. Können in Playbooks und Vorlagen als Variablen genutzt werden [28].

Ansible-Galaxy Ist ein Verteilungsserver zum Teilen von Ansible-Community-Inhalten. Über das gleichnamige Kommandozeilenprogramm können Nutzer Ansible-Collections installieren [28].

3.4 SaltStack

SaltStack, kurz Salt, ist ein Open-Source Konfigurationsmanagementwerkzeug, das auf Python basiert. Salt ist ereignisgesteuert und dient zur automatisierten Bereitstellung, Konfiguration und Verwaltung komplexer IT-Systeme.

Installation und Konfiguration

Die Installation von SaltStack muss in zwei Teile untergliedert werden: die Installation der Master-Knoten und die Installation der Worker-Knoten. SaltStack empfiehlt in beiden Fällen die Verwendung eines bereitgestellten Bootstrap-Skripts. Je nach bereitgestellten Parametern beim Ausführen dieses Skripts wird entweder der Salt-Master-Dienst für die Master-Knoten oder der Salt-Minion-Dienst für die Worker-Knoten installiert. Für Installationen auf Windows und macOS steht ein Installationsprogramm zur Verfügung.

Um die Kommunikation der Salt-Minions mit dem Salt-Master zu ermöglichen, ist eine vorherige Konfiguration erforderlich. Dabei muss sichergestellt werden, dass die Minions den Hostnamen *salt* zum Auflösen des Salt-Masters nutzen können. Die Auflösung des Hostnamens, der dem Master zugeordnet ist, kann in der Minion-Konfiguration angepasst werden.

Wenn der Minion zum ersten Mal gestartet wurde initiiert dieser einen Handshake und sendet seinen öffentlichen Schlüssel an den Master. Dieser Schlüssel wird auf dem Master-Server gespeichert und muss durch den Befehl `salt-key` akzeptiert werden. Ein Minion führt Befehle erst aus, nachdem sein öffentlicher Schlüssel akzeptiert wurde.

Nachdem der öffentliche Schlüssel akzeptiert wurde, antwortet der Master mit seinem eigenen öffentlichen Schlüssel, sowie einem rotierenden AES-Schlüssel. Dieser wird zum Verschlüsseln und Entschlüsseln der Nachrichten verwendet. Der zurückgegebene AES-Schlüssel wird mit dem öffentlichen Schlüssel des Minions verschlüsselt und kann daher nur von diesem entschlüsselt werden. Die gesamte weitere Kommunikation zwischen dem Master und dem Minion wird mit AES-Schlüsseln verschlüsselt [36].

Kommunikation und Arbeitsmodus

Salt nutzt ein Server-Agent-Kommunikationsmodell. Die Kommunikation zwischen dem Master und den Minions erfolgt über ein Publish-Subscribe-Muster. Das Kommunikationssystem richtet eine persistente Datenverbindung zwischen dem Salt-Master und den Minions mithilfe von ZeroMQ oder TCP ein. Die Verbindungen werden vom Minion initiiert, daher ist es nicht nötig eingehende Ports auf den Zielsystemen zu öffnen, was den Angriffsvektor verringert [36].

Salt bietet sowohl den Push- als auch Pull-Ansatz an. Der Nutzer kann Aktualisierungen über den Master an die Minions senden oder die Minions anweisen, geplant zu bestimmten Zeiten Informationen vom Master abzurufen und verfügbare Aktualisierungen zu laden.

Es besteht die Möglichkeit, Salt ohne einen Master laufen zu lassen, jedoch sind in diesem Fall Einschränkungen zu berücksichtigen. Des Weiteren bietet Salt ebenfalls die Möglichkeit, Befehle und States über SSH auszuführen, ohne einen Minion zu installieren.

Funktionsweise

Salt nutzt die *Remote-Execution-Engine*, die ein schnelles und sichere bidirektionales Kommunikationsnetz für eine Gruppe von Systemen aufbaut. Auf diesem Kommunikationsnetz aufbauend, bietet Salt einen extrem schnelles, flexibles und einfach zu nutzendes Konfigurationsmanagementsystem: die *Salt States*. Salt States oder das State-System ist die Komponente von Salt, die für das Konfigurationsmanagement zuständig ist. Das

State-System basiert auf *SLS*-Formeln. Diese Formeln liegen in Dateien auf dem Dateiserver von Salt [35].

Die Remote-Execution-Engine sendet über den Publisher-Port an alle verbundenen Minions. Die Kommunikation des Salt-Masters ist eine leichtgewichtige Menge an Instruktionen. Die Instruktionen sehen etwa wie folgt aus: Wenn du ein Minion mit diesen Eigenschaften bist, führe folgenden Befehl mit diesen Argumenten aus [38].

Die Minions stellen fest, ob sie die Eigenschaften erfüllen, wenn sie den Befehl erhalten und entscheiden so, ob der Befehl ausgeführt wird oder nicht. Jeder Salt-Minion hat bereits alle Befehle, die er benötigt, lokal gespeichert, sodass der Befehl ausgeführt werden kann und die Ergebnisse schnell an den Salt-Master zurückgegeben werden können. Alle Befehle, die vom System ausgeführt werden können, werden von Salt bereitgestellt und heißen *execution modules*. Wenn ein Minion einen Befehl erhält, sucht er das dazu passende Modul und ruft die entsprechende Funktion auf. Die Befehle, die ein Minion erhält, werden jeweils einem eigenen Worker-Thread zugewiesen. Dadurch können Minions mehrere Jobs gleichzeitig ausführen. Salt-Befehle werden plattformunabhängig ausgeführt. Salt abstrahiert die spezifischen Details der einzelnen Betriebssysteme, Hardwaretypen und Systemwerkzeuge [38].

Konfigurationssprache

SaltStack verwendet YAML als Konfigurationssprache und Jinja2 als Templating-Sprache.

Terminologie und Konzepte

Salt State File Die SLS-Datei ist eine wiederverwendbare Konfiguration, die einen spezifischen Teil eines Systems beschreibt. Sie repräsentiert den gewünschten Zustand [38].

Formula Ist eine Sammlung von SLS-Dateien. Formulas bieten eine Möglichkeit, häufig verwendete Konfigurationsmuster zu standardisieren und wiederverwendbar zu machen [34].

Grains Schlüssel-Werte-Paare, die Fakten über das System enthalten [37].

Formula Repository Eine zentrale Sammlung von Formula-Repositorys für SaltStack.

3.5 Chef

Progress Chef ist eine Sammlung von Werkzeugen, die dabei helfen Systeme durch Code zu konfigurieren. Chef Infra ist das zugehörige Open-source Konfigurationsmanagementwerkzeug. Die Konfiguration wird einmal definiert und dann auf alle Systeme angewandt, unabhängig vom Betriebssystem, dessen Version und der Architektur. Im Gegensatz zu anderen Werkzeugen verfolgt Chef einen *policy-based* Ansatz, der auf den Prinzipien von *Test Driven Development* und Idempotenz aufbaut [21].

Installation und Konfiguration

Die Installation von Chef umfasst drei Komponenten.

Chef Workstation Die Workstation kann auf Unix-Systemen über den Paketmanager des jeweiligen Systems installiert werden. Für Windows steht ein Installationsprogramm zur Verfügung. Es wird empfohlen, die Ruby Version der Chef Workstation als Standardversion auf dem System zu verwenden, auf dem die Workstation ausgeführt wird. Beim ersten Ausführen der Workstation wird das Verzeichnis `.chef` im Benutzerverzeichnis erstellt. Dies ist der Speicherort, an dem die Workstation-Konfiguration und der Client-Schlüssel abgelegt werden. Die gesamte Kommunikation zwischen der Workstation und dem Chef Server erfolgt über eine RSA-Schlüsselpaar-Authentifizierung. Dieses Schlüsselpaar wird auf dem Chef Server generiert. Der private Schlüssel muss in das `.chef` Verzeichnis kopiert werden, damit die Kommunikation funktioniert [18].

Chef Server Der Chef Server kann in drei verschiedenen Modi betrieben werden: Standalone, High Availability und Tiered. Die Installation hängt vom gewählten Modus ab. Zuerst muss das Paket für eine der unterstützten Linux-Plattformen heruntergeladen werden. Nach dem Herunterladen wird das Paket unter Verwendung von Root-Rechten installiert. Anschließend müssen alle Serverdienste gestartet werden. Danach muss ein Administrator erstellt werden, was zur Generierung eines privaten RSA-Schlüssels führt. Zuletzt muss eine Organisation erstellt werden, für die ebenfalls ein privater RSA-Schlüssel erzeugt wird. Dieser Schlüssel wird als *chef-validator key* bezeichnet und sollte sicher aufbewahrt werden [22].

Chef Client Für die Installation des Chef Clients kann auch ein passendes Paket für die Plattform des zu verwaltenden Knotens heruntergeladen und installiert werden.

> Eine einfachere und schnellere Möglichkeit besteht jedoch darin, das zusätzliche Werkzeug *knife* zu verwenden, um einen Knoten einzurichten. *knife* verbindet sich per SSH mit dem Knoten und automatisiert dessen Einrichtung [17].

Zudem gibt es ein universelles Installationsskript, welches die verschiedenen Anwendungen installieren kann. Das Skript erkennt automatisch die Plattform, Version und Architektur des Systems und installiert die erforderlichen Pakete.

Kommunikation und Arbeitsmodus

Die API des Chef Servers verwaltet die gesamte Kommunikation zwischen Client und Workstation. Diese API ist eine authentifizierte REST-API. Das bedeutet, dass für alle Anfragen, Authentifizierung und Autorisierung erforderlich sind. *knife* verwendet ebenfalls die Chef Server API [24].

Chef nutzt standardmäßig den Pull-Ansatz, um Konfigurationen zu verteilen.

Funktionsweise

Chef Infra verfolgt einen richtlinienbasierten Ansatz. Diese Richtlinien verbinden geschäftliche und betriebliche Anforderungen, Prozesse und Abläufe mit den folgenden Einstellungen und Objekten, die auf dem Chef Server gespeichert sind: Rollen, Umgebungen, Attribute, sensible Daten und Cookbooks [23].

Die Workstation ist das lokale System eines Administrators und fungiert als Schnittstelle zum Chef Server oder den Chef Clients. Auf der Workstation werden Konfigurationen in Form von Cookbooks und Policyfiles erstellt und getestet. Anschließend werden diese mithilfe von *knife* an den Chef Server übertragen. Zudem dient die Workstation zur Synchronisierung der Chef Repositorys durch die Verwendung von Versionskontrolle. Darüber hinaus ermöglicht die Workstation die Interaktion mit den zu verwaltenden Knoten, beispielsweise bei der Durchführung einer Bootstrap-Operation.

Der Chef Server ist die zentrale Einheit des Systems. Hier werden Cookbooks, Recipes und Metadaten gespeichert. Der auf den zu verwaltenden Knoten laufende Chef Client, stellt sicher, dass die Agenten in regelmäßigen Intervallen den Chef Server abfragen, notwendige Befehle ausführen und die Ergebnisse an den Server zurückliefern [23].

Der Agent aktualisiert sich selbstständig, automatisiert seine Abläufe und ist auch in Umgebungen mit begrenzter Bandbreite effektiv einsetzbar. Er stellt filterbare Echtzeitdaten für jeden verwalteten Knoten zur Verfügung und unterstützt den gesamten Arbeitsablauf von der Durchsetzung von Zuständen bis zur Datenaggregation und Validierung [21].

Konfigurationssprache

Chef verwendet die Chef Infra Language, eine auf Ruby basierende DSL. Diese Sprache unterstützt eine Vielzahl der häufigsten Szenarien für die Automatisierung von Infrastrukturen. Wenn zusätzliche Funktionalitäten benötigt werden, kann die DSL durch nativen Ruby-Code erweitert werden [20].

Terminologie und Konzepte

Cookbook Die grundlegende Einheit der Konfigurations- und Richtlinienverteilung. Beschreiben die zu verwaltenden Systemressourcen wie Dateien, Vorlagen und Softwarepakete [19].

Policyfile Bietet die Möglichkeit eine unveränderliche Sammlung von Cookbooks, Cookbook-Abhängigkeiten und Attributen in einer einzigen Datei zu definieren. Diese Datei wird einer Gruppe von Knoten zugewiesen, welche sich beim Anfragen des Chef Servers dann nach diesen Richtlinien konfigurieren [23].

Attribute Definieren umgebungsspezifische Details über einen Knoten und werden über das Werkzeug *Ohai* gesammelt [23].

Chef Supermarket Ist eine Plattform, die eine Sammlung von Community-Cookbooks zur Verfügung stellt.

3.6 Puppet

Puppet ist ein von Puppetlabs entwickeltes Open-Source Konfigurationsmanagementwerkzeug. Mit Puppet wird der gewünschte Zustand der Infrastruktur definiert und anschließend automatisch auf die Systeme angewandt, wodurch eine effiziente und konsistente Konfigurationsverwaltung ermöglicht wird.

Installation und Konfiguration

Die Installation von Puppet auf Unix-Systemen beginnt mit der Aktivierung des Puppet-Repositorys im genutzten Paketmanager. Anschließend kann entweder das Paket für den Puppet-Agent oder den Puppet-Server installiert werden. Für Windows-Systeme steht ein Installationsprogramm für den Agenten zur Verfügung. Optional kann zusätzlich die eigens für Puppet entwickelte *PuppetDB*-Datenbank installiert werden. Dies erweitert die Funktionalität um zusätzliche Features wie erweiterte Abfragen und umfassende Berichte über die Infrastruktur [12].

Um ein reibungsloses Funktionieren des Puppet-Systems zu gewährleisten, müssen sowohl der Puppet-Server als auch die Agenten eindeutige Hostnamen haben. Diese Hostnamen sollten innerhalb des Systems auflösbar sein, entweder über einen privaten DNS-Server oder mithilfe der *hosts*-Datei[2]. Ebenso ist die Erreichbarkeit des Puppet-Masters über Port 8140 essenziell. Standardmäßig geht Puppet davon aus, dass der Master den Hostnamen *puppet* hat. Sollte dies nicht der Fall sein, muss der korrekte Hostname in der Konfiguration aller Agenten angepasst werden. Das SSL-Zertifikat des Masters, das zur Unterzeichnung der Agentenzertifikate dient, muss dann entsprechend neu generiert werden [15].

Da der Puppet-Master als Zertifizierungsstelle für die Agenten fungiert, ist eine präzise Systemzeit unerlässlich. Nachdem ein Puppet-Agent erstmals gestartet wurde, sendet dieser eine Anfrage zur Unterzeichnung des Zertifikats (certificate signing request) an den Puppet-Master. Bevor eine Kommunikation zwischen Master und Agent möglich ist, muss diese Anfrage vom Master signiert werden [12].

Alternativ kann Puppet in einem *Masterless*-Modus ausgeführt werden, bei dem jeder verwaltete Knoten über eine eigene und vollständige Kopie der Konfigurationsinformationen verfügt und seinen eigenen Katalog kompiliert. In diesem Modus führen die verwalteten Knoten die Anwendung *Puppet apply* aus, normalerweise als geplante Aufgabe oder Cron-Job. Eine Ausführung auf Bedarf ist auch möglich, zum Beispiel um die anfängliche Konfiguration eines Servers oder kleinere Konfigurationsaufgaben durchzuführen [13].

[2] Die hosts-Datei ist eine lokale Konfigurationsdatei, um Hostnamen zu IP-Adressen zuzuordnen.

Kommunikation und Arbeitsmodus

Der Puppet-Master bietet eine Schnittstelle mit verschiedenen Endpunkten. Wenn der Agent etwas beim Master anfordert oder übermittelt, stellt er eine Anfrage an einen dieser Endpunkte.

Die Kommunikation zwischen Master und Agenten erfolgt über HTTPS mit Client-Verifikation. Daher ist es notwendig, dass sowohl der Master als auch die Agenten über identifizierbare SSL-Zertifikate verfügen. Die im Master integrierte Zertifizierungsstelle ermöglicht es den Agenten, durch eine Anfrage an die API ein Zertifikat anzufordern.

Puppet verwendet standardmäßig den Pull-Ansatz [13].

Funktionsweise

Beim Einsatz von Puppet wird der gewünschte Zustand der zu verwaltenden Infrastruktur definiert. Dafür werden Konfigurationen in *Puppet Code* geschrieben. Puppet Code ist deklarativ, das bedeutet, er beschreibt den gewünschten Zustand des Systems und nicht die Schritte, die erforderlich sind, um dorthin zu gelangen. Anschließend automatisiert Puppet den Prozess, die Systeme in diesen Zustand zu überführen und sie in diesem Zustand zu halten. Der Puppet-Primärserver speichert den Code, der den gewünschten Zustand definiert. Der Puppet-Agent übersetzt diesen Code in Befehle und führt diese dann auf dem ausgewählten System aus. Dieser Vorgang wird als *Puppet Run* bezeichnet [16].

Bei Verwendung der Agent-Master-Topologie fordern die verwalteten Knoten ihre eigenen Konfigurationskataloge vom Master an. Diese Knoten führen den Puppet-Agenten als Hintergrunddienst aus. Die Agenten senden in regelmäßigen Abständen Fakten an den Master und fordern ihren Konfigurationskatalog an. Der Master kompiliert daraufhin den individuellen Katalog für den anfordernden Knoten. Dieser Katalog wird aus verschiedenen Informationsquellen erstellt, auf die der Master Zugriff hat. Sobald ein Agent seinen Katalog erhält, wendet er ihn auf seinen Host an, indem er jede Ressource überprüft, die im Katalog beschrieben ist. Findet der Agent Ressourcen, die nicht im gewünschten Zustand sind, führt er die erforderlichen Änderungen durch, um dies zu korrigieren. Im *no-op*-Modus werden die Änderungen nicht ausgeführt, es wird jedoch berichtet, welche Änderungen geplant sind. Dadurch lassen sich anstehende Änderungen vor ihrer Anwendung überprüfen [13].

Konfigurationssprache

Puppet nutzt eine domänenspezifische Sprache, den sogenannten Puppet Code. Als Templating-Sprache wird Embedded Puppet (EPP) oder Embedded Ruby (ERB) genutzt.

Terminologie und Konzepte

Manifest File Beinhaltet Code, der in der Puppet-Sprache geschrieben ist. Kann Ressourcen und Klassen deklarieren, Variablen setzen, Funktionen evaluieren und Klassen, definierte Typen, Funktionen und Knoten definieren [14].

Module Eine Sammlung von Klassen, Ressourcentypen, Dateien, Funktionen und Vorlagen, für eine spezielle fachliche oder logische Aufgabe [14].

Fact Informationen über einen Knoten. Das Werkzeug *Facter* liest Informationen, wie Hostnamen, IP-Adressen oder das Betriebssystem und macht diese für Puppet verfügbar [14].

Puppet Forge Ein Katalog von Modulen, die entweder von Puppet oder der Community erstellt wurden. Diese bieten die Möglichkeit vorgefertigte Module zu nutzen und vereinfachen damit den Automatisierungsprozess [14].

Kriterium Community

Die Community eines Werkzeuges spielt besonders für unerfahrene Nutzer eine wichtige Rolle. Die Aktivität der Community beeinflusst die Verfügbarkeit von Anleitungen, die Dauer der Beantwortung von Fragen, die Vielfalt an bestehenden Lösungen für spezifische Probleme sowie die Geschwindigkeit der Entwicklung neuer Funktionen und Behebung von Fehlern.

Eine Bewertung der Community gestaltet sich als herausfordernd, da sie über verschiedene Plattformen wie Foren, Blogs und Videos verteilt ist. Aus diesem Grund wurden die Aktivitäten auf GitHub[3] und Stack Overflow[4] als Indikatoren für die Community-Aktivität herangezogen. Bei der Analyse der GitHub-Projekte wurden Kennzahlen wie

[3] https://github.com/
[4] https://stackoverflow.com/

die Anzahl der Mitwirkenden (Contributors) und Sterne (Stars) berücksichtigt. Einige Werkzeuge verfügen über mehrere Repositorys, beispielsweise für die verschiedenen Cloud-Provider. In solchen Fällen wurden relevante Repositorys zusammengefasst. Für die Auswertung der Stack Overflow Aktivität wurden sowohl die aktuelle Stack Overflow Umfrage[5] als auch Statistiken aus den Stack Overflow Trends[6] und Tags[7] herangezogen.

Die Tabelle 3.1 zeigt, dass die Aktivität von Ansible und Terraform im Hinblick auf die betrachteten Kriterien im Vergleich zu den anderen Werkzeugen vergleichsweise hoch ist.

Tabelle 3.1: Community Aktivität (August 2023)

Werkzeug	Mitwirkende	Sterne	Module
Terraform	1,752	38.400	14526[a]
Pulumi	220	17.000	143[b]
Ansible	5,494	58.300	31682[c]
SaltStack	2,430	13.500	356[d]
Chef	667	7.300	4029[e]
Puppet	577	7.100	6993[f]

[a] Module aus Terraform Registry
[b] Packages aus Pulumi Registry
[c] Rollen aus Ansible-Galaxy
[d] SaltStack-Repositorys, die Formulas repräsentieren
[e] Cookbooks aus Chef Supermarket
[f] Module aus Puppet Forge

Stack Overflow Survey 2023

Die Stack Overflow Survey von 2023 behandelt die Frage: „Mit welchen Entwicklungswerkzeugen haben Sie im vergangenen Jahr umfangreiche Entwicklungsarbeit geleistet und mit welchen wollen Sie im nächsten Jahr arbeiten?“

11,3 % der 80,249 Befragten gaben dabei Terraform und 8,62 % Ansible an. Puppet mit 1,12 %, Chef mit 0,94 % und Pulumi mit 0,82 % werden weniger genutzt. SaltStack wurde wurde von keinem der Befragten angegeben [11].

[5] https://survey.stackoverflow.co/2023/
[6] https://insights.stackoverflow.com/trends?tags=
[7] https://stackoverflow.com/tags

Anzahl Fragen per Tag Auf Stack Overflow lässt sich die Anzahl der gestellten Fragen zu einem Schlüsselwort ermitteln. Abbildung 3.2 veranschaulicht die Anzahl der Fragen zu den verschiedenen Werkzeugen. Es ist jedoch wichtig zu beachten, dass für einige Werkzeuge verschiedene Schlüsselwörter kombiniert wurden, da die Fragen nicht unter einem einheitlichen Schlüsselwort zusammengefasst sind. Die Abbildung zeigt, dass die Anzahl der Fragen für Terraform und Ansible am höchsten ist.

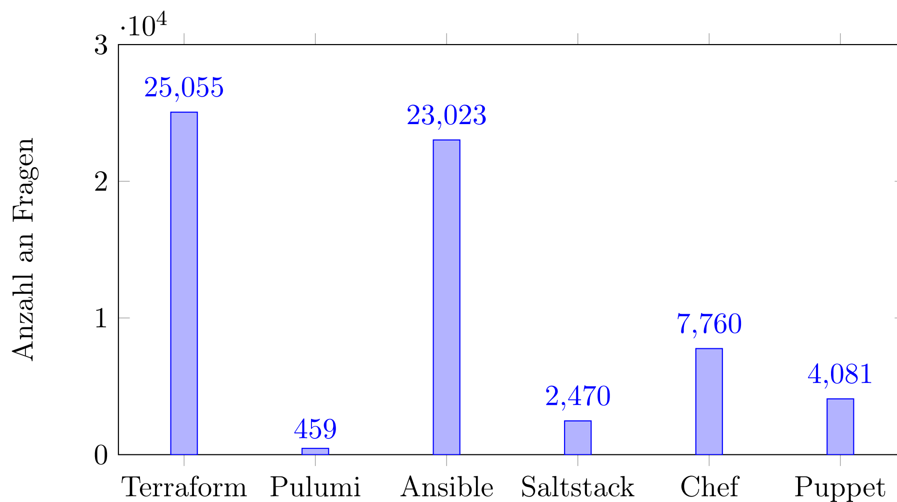

Abbildung 3.2: Anzahl der Fragen pro Schlüsselwort auf Stack Overflow (August 2023)

Stack Overflow Trends

Die Stack Overflow Trends gewähren Einblicke in die Entwicklung der Anzahl von gestellten Fragen zu verschiedenen Technologien und Werkzeugen im Laufe der Zeit. Es ist wichtig zu betonen, dass diese Daten nicht zwangsläufig auf Beliebtheit oder Qualität hinweisen. Stattdessen dienen sie als Indikator für die Aktivitätsentwicklung in den jeweiligen Communitys im Zeitverlauf.

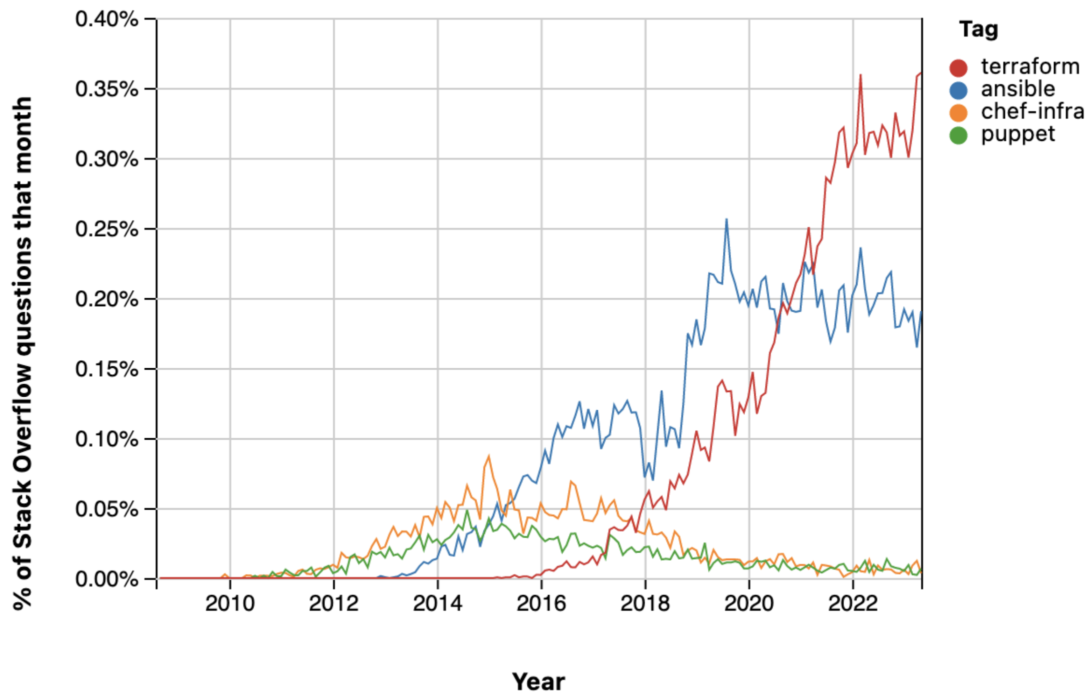

Abbildung 3.3: Stack Overflow Trends

Abbildung 3.3 ist zu entnehmen, dass Chef und Puppet rückläufige Trends aufweisen, während Ansible und Terraform steigende Trends verzeichnen.

Abschließend lässt sich festhalten, dass aus den Daten von Stack Overflow und GitHub hervorgeht, dass Ansible und Terraform derzeit als die populärsten Werkzeuge gelten. Diese Beliebtheit könnte auf ihre umfangreiche Community-Unterstützung, ihre flache Lernkurve und damit einfache Nutzung, die breite Anwendbarkeit in unterschiedlichen Szenarien sowie die kontinuierliche Weiterentwicklung zurückzuführen sein.

Vergleichsmatrix

Aus dem Vergleich der Werkzeuge ergibt sich nun folgende Vergleichsmatrix, welche die Haupterkenntnisse übersichtlich zusammenfasst.

	Terraform	Pulumi	Ansible	Saltstack	Chef	Puppet
Konfigurationssprache	HCL	GPLs	YAML	YAML	Chef-Infra-Langauge	PuppetDSL
Verteilungsmethode	Push	Push	Push/Pull	Push	Push/Pull	Pull
Konfigurationsansatz	Deklarativ	Deklarativ	Hybrid	Hybrid	Hybrid	Deklarativ
Infrastruktur	Unveränderlich	Unveränderlich	Veränderlich	Veränderlich	Veränderlich	Veränderlich
Topologie	Masterless Agentless	Masterless Agentless	Masterless Agentless	Master Agent	Master Agent	Master Agent
Installation	Sehr einfach	Einfach	Sehr einfach	Einfach	Mittel	Mittel
Unterstützte Ziel-Betriebssysteme			Windows Linux macOS	Windows Linux macOS	Windows Linux macOS	Windows Linux macOS
Community	Sehr aktiv	Weniger aktiv	Sehr aktiv	Weniger aktiv	Aktiv	Aktiv

Abbildung 3.4: Vergleichsmatrix der Werkzeuge

Mithilfe dieser Matrix, lassen sich je nach Anforderungen eines Projektes, bereits Entscheidungen darüber treffen, welches Werkzeug für ein Projekt geeignet ist.

3.7 Zwischenergebnis des Vergleiches

In diesem Abschnitt werden die Ergebnisse des Vergleiches zusammengefasst.

Die ausgewählten Werkzeuge sind alle Open-Source und bieten zudem eine Enterprise-Version an. Diese Enterprise-Versionen umfassen oft erweiterte Funktionalitäten, wie beispielsweise stärkere Sicherheitsmechanismen, grafische Benutzeroberflächen zur Verwaltung der Knoten und Konfigurationen sowie Master-Instanzen, die in der Cloud gehostet werden und nicht eigenständig verwaltet werden müssen. Die Funktionen der Enterprise-Versionen sind gerade in größeren Projekten wichtig, können aber aufgrund des Rahmens

dieser Arbeit nicht behandelt werden. Alle Werkzeuge, die einen Agenten nutzen, unterstützen Linux, Windows und macOS als Zielsysteme.

Systeme mit einer Master-Agent-Topologie wurden jeweils mit einem einzigen Master installiert, damit sich der Installationsprozess der Werkzeuge besser vergleichen lässt. In der Praxis werden jedoch meistens Systeme mit Hochverfügbarkeit, also mehreren Master-Instanzen benötigt. Die Installation dieser Systeme ist in der Regel aufwändiger. Zusätzlich ist es wichtig anzumerken, dass der Schwierigkeitsgrad der Installation, von den bestehenden Kenntnissen abhängt. Die Installation sowie Konfiguration von Systemen mit einer Master-Agent-Topologie ist komplexer als bei Systemen, die ohne Master und Agenten auskommen. Das liegt daran, dass die Installationen grundlegende Kenntnisse über Systemadministration, Netzwerke, Asymmetrische Verschlüsselungsverfahren und Zertifikate erfordert.

Ein weiterer Punkt, der beim Betrachten des Vergleiches auffällt, ist die zunehmende Eignung der Werkzeuge für die Nutzung durch Softwareentwickler. Dies zeigt sich darin, dass einige Werkzeuge Allzweck-Programmiersprachen anstelle von domänenspezifischen Sprachen als Konfigurationssprache anbieten. Dieser Trend resultiert daraus, dass gerade in kleineren Teams oft keine dedizierten Infrastrukturentwickler vorhanden sind. Es wird immer häufiger, dass Softwareentwickler für sowohl das Deployment als auch die erforderliche Infrastruktur verantwortlich sind.

Die prototypische Implementierung bestätigt einige der Erkenntnisse, liefert jedoch auch neue Erkenntnisse, die beim Betrachten der Literaturrecherche nicht so deutlich hervorstechen. Es wird vor allem deutlich, dass sich die Werkzeuge anhand ihrer Spezialisierungen in zwei Gruppen einteilen lassen. Terraform und Pulumi sind hauptsächlich für die Provisionierung und Orchestrierung von Cloud-Ressourcen vorgesehen, während Ansible, SaltStack, Chef und Puppet auf das Konfigurationsmanagement ausgerichtet sind. Es gibt Überschneidungen zwischen den Funktionalitäten dieser beiden Gruppen, diese sind jedoch in ihrem Umfang eingeschränkt. Des Weiteren ist anzumerken, dass Provisionierungswerkzeuge standardmäßig mit unveränderlicher Infrastruktur arbeiten, während Konfigurationsmanagementwerkzeuge mit veränderlicher Infrastruktur arbeiten.

Die Terminologie der verschiedenen Werkzeuge zeigt, dass grundlegende Konzepte gemeinsam genutzt werden, auch wenn diese unterschiedlich benannt sind. Jedes Werkzeug hat eine Form von Konfigurationsdatei, eine wiederverwendbare Sammlung dieser Konfigurationsdateien, Fakten über die zu verwaltenden Knoten und ein Repository, in

dem öffentliche Sammlungen von Konfigurationen heruntergeladen werden können. Provisionierungswerkzeuge teilen zusätzlich noch die Konzepte eines Zustandes (State) und Providern, die verschiedene Ressourcen anbieten.

Diese Ähnlichkeit der Konzepte und Funktionalitäten verdeutlicht, dass die Grundideen der Infrastrukturverwaltung über die verschiedenen Werkzeuge hinweg ähnlich sind, auch wenn Umsetzung und Benennung variieren können. Diese Gemeinsamkeiten ermöglichen es Entwicklern und Administratoren, ihre Kenntnisse zwischen den verschiedenen Werkzeugen zu übertragen.

Bei Betrachtung der verschiedenen Implementierungen des Fallbeispiels wird deutlich, dass sich der Quellcode der Infrastruktur trotz unterschiedlicher Werkzeuge ebenfalls sehr ähnlich ist. Dies ist auf die generelle Funktionsweise der Werkzeuge zurückzuführen. Sie abstrahieren Aufgaben aus der Systemadministration und fassen diese in Modulen zusammen. Abbildung 3.5 zeigt die Initialisierung des Kubernetes-Masters in Ansible, SaltStack, Chef und Puppet.

```
1  - name: Initialize cluster on master node
2    shell: kubeadm init --pod-network-cidr=10.244.0.0/16 >> ci.txt
3    args:
4     chdir: $HOME
5     creates: ci.txt
6
7  - name: create .kube directory
8    file:
9      path: $HOME/.kube
10     state: directory
11     mode: 0755
12
13 - name: copy admin.conf to user's kube config
14   copy:
15     src: /etc/kubernetes/admin.conf
16     dest: $HOME/.kube/config
17     remote_src: yes
18
19 - name: install Pod network
20   shell: kubectl apply -f kube-flannel.yml >> pns.txt
21   args:
22     chdir: $HOME
23     creates: pns.txt
```

(a) Ansible

```
1  init-cluster:
2    cmd.run:
3      - name: kubeadm init --pod-network-cidr=10.244.0.0/16 >> ci.txt
4      - creates: ci.txt
5
6  admin-config:
7    file.managed:
8      - source: /etc/kubernetes/admin.conf
9      - name: /root/.kube/config
10     - mode: '0755'
11     - makedirs: true
12
13 install-pod-network:
14   cmd.run:
15     - name: kubectl apply -f kube-flannel.yml >> pns.txt
16     - creates: pns.txt
```

(b) SaltStack

```
1  execute 'init_master' do
2    command "kubeadm init --pod-network-cidr=10.244.0.0/16 >> ci.txt"
3    creates "cluster_initialized.txt"
4    cwd "/root"
5  end
6
7
8  directory "/root/.kube" do
9    action :create
10   mode "0755"
11 end
12
13 remote_file "/root/.kube/config" do
14   source "file:///etc/kubernetes/admin.conf"
15 end
16
17 execute 'install pod network' do
18   command "kubectl apply -f kube-flannel.yml >> pns.txt"
19   creates "pod_network_setup.txt"
20   cwd "/root"
21 end
```

(c) Chef

```
1  class kubernetes::master {
2  require kubernetes::install
3  $path = ['/usr/bin','/usr/sbin']
4
5  exec { 'init master':
6    command   => 'kubeadm init --pod-network-cidr=10.244.0.0/16 >> ci.txt',
7    path      => $path,
8    creates   => '/root/ci.txt',
9    logoutput =>  true,
10 }
11
12 file { '/root/.kube':
13   ensure => 'directory',
14   mode   => '0755',
15 }
16
17 file { '/root/.kube/config':
18   ensure => 'present',
19   source => '/etc/kubernetes/admin.conf',
20 }
21
22 exec { 'install pod network':
23   command => 'kubectl apply -f kube-flannel.yml >> pns.txt',
24   path    => $path,
25   creates => '/root/pns.txt',
26   cwd     => '/root',
27   }
28 }
```

(d) Puppet

Abbildung 3.5: Vergleich der Initialisierung des Kubernetes-Masters

Anhand der Vergleichsmatrix wird ebenfalls deutlich, dass die Werkzeuge im Hinblick auf die betrachteten Kriterien und den daraus resultierenden Eigenschaften eine hohe Ähnlichkeit aufweisen. Die Gemeinsamkeiten der Werkzeuge erschweren die Entscheidung darüber, welches Werkzeug für den eigenen Anwendungsfall am sinnvollsten ist. Daher werden im nächsten Kapitel die verschiedenen Paradigmen und ihre Auswirkun-

gen erörtert, um die Stärken und Schwächen der Werkzeuge sowie ihre Eignung für die Verwaltung von Cloud-Infrastruktur aufzuzeigen.

3.8 Limitierungen des Vergleiches

Der vorangegangene Vergleich weist einige Einschränkungen hinsichtlich der allgemeinen Aussagekraft der Ergebnisse auf. Der Vergleich stützt sich auf die Literaturrecherche und die prototypische Implementierung eines Fallbeispiels. Die Implementierung wiederum basiert auf einem konkreten Anwendungsfall. Obwohl bei der Konzeption des Anwendungsfalls darauf geachtet wurde, verschiedene Bereiche der Systemadministration und Infrastrukturverwaltung abzudecken, kann dieser Anwendungsfall nicht alle möglichen Aufgaben und Szenarien der Infrastrukturverwaltung umfassen.

Die meisten Vergleiche von Infrastructure as Code Werkzeugen behandeln, ähnlich wie der vorangegangene Vergleich, nur grundlegende Merkmale und gehen nicht auf spezifische Kriterien ein. Ohne anwendungsspezifische Daten von Nutzern mit praktischer Erfahrung, führen die meisten Vergleiche zu dem Schluss, dass sich die Werkzeuge bei Betrachtung allgemeiner Merkmale sehr ähnlich sind. Dies muss im Hinblick auf die Gültigkeit der erhobenen Daten und der daraus resultierenden Ergebnisse, welche in die Evaluation einfließen, berücksichtigt werden.

Um noch aussagekräftigere Informationen zu erhalten, könnte eine Umfrage unter regelmäßigen Nutzern der vorgestellten Werkzeuge durchgeführt werden. Ein weiterer Vorschlag, um den Schwächen des Vergleiches entgegenzuwirken wäre es, die Werkzeuge im Hinblick auf komplexere Anwendungsfälle zu vergleichen, um eine genauere Beurteilung zu ermöglichen. Da diese Thesis jedoch einen allgemeinen Überblick bieten soll, wurde auf eine detaillierte Betrachtung spezifischer Kriterien verzichtet.

4 Evaluation

Das Ziel dieser Arbeit ist der Vergleich und die Evaluation von Infrastructure as Code Werkzeugen im Kontext der Cloud, um die Stärken und Schwächen der Werkzeuge zu identifizieren und daraus Schlussfolgerungen hinsichtlich ihrer Eignung für die Cloud ziehen zu können. Im vorherigen Kapitel wurde bereits ein umfassender Vergleich verschiedener Infrastrukturverwaltungswerkzeuge durchgeführt. In diesem Kapitel werden die gesammelten Ergebnisse dieses Vergleiches näher betrachtet, ihre Bedeutung erläutert und die Auswirkungen auf die Eignung für die Infrastrukturverwaltung in der Cloud diskutiert.

Die Auswahl eines geeigneten Werkzeuges stellt für Organisationen keine triviale Aufgabe dar und hängt von einer Vielzahl verschiedener Kriterien ab. Einige dieser Kriterien lassen sich objektiv bewerten, während andere aus projektspezifischen Anforderungen oder den internen Richtlinien einer Organisation resultieren. Die Evaluierung von Infrastrukturwerkzeugen ist wichtig, da Organisationen fundierte Entscheidungen treffen müssen, um ein effizientes Infrastrukturmanagement zu gewährleisten. Im Rahmen von DevOps werden Prozesse optimiert, um Zeit zu sparen und Kosten zu minimieren. Eine undurchdachte Werkzeugauswahl kann jedoch zu unnötigen Kosten und Aufwand führen, welche die Entwicklung einer Organisation bremsen können.

Methodik der Evaluation

Die aus dem Vergleich hervorgehenden Unterschiede der Werkzeuge werden hinsichtlich ihrer Stärken und Schwächen analysiert und es wird erörtert, wie sich diese auf entscheidende Aspekte der Infrastrukturverwaltung auswirken können. Dabei wird die Frage beantwortet, wie folgende Aspekte die Eignung der Werkzeuge beeinflussen:

- Auswirkung der verschiedenen Topologien
- Auswirkung der Konfigurationssprache

- Auswirkung der Entscheidung zwischen prozeduralem und deklarativem Ansatz
- Auswirkung der Community

Diese Herangehensweise trägt dazu bei, die Leitfrage der Arbeit zu beantworten und gibt gleichzeitig einen Ausblick darauf, wie die gewonnenen Erkenntnisse auf verschiedene Anwendungsfälle übertragen werden können.

4.1 Master vs. Masterless und Agent vs. Agentless

Der folgende Abschnitt beantwortet die Frage, wie sich die verschiedenen Topologien von Infrastrukturwerkzeugen auf die Verwaltung von Infrastruktur auswirken. Dabei wird sowohl der Unterschied zwischen Systemen mit und ohne Master betrachtet *(Master vs. Masterless)*, als auch Systemen mit und ohne Agenten *(Agent vs. Agentless)*.

Infrastrukturwerkzeuge, die einen Master erfordern, haben den Vorteil, dass die verwaltete Infrastruktur an einem zentralen Ort liegt. Master-Systeme haben die Autorität über alle verwalteten Knoten und können diese so zwingen, die vorgegebenen Konfigurationen und Richtlinien einzuhalten. Master-Systeme haben jedoch den Nachteil, dass sie ihre eigene Infrastruktur benötigen, was zusätzliche Ressourcen wie virtuelle Maschinen oder Server bedeutet. Ein weiterer Nachteil von Master-Systemen ist, dass diese installiert, konfiguriert, aktualisiert und verwaltet werden müssen, was zusätzlichen Aufwand mit sich bringt.

Für Agenten gilt ebenfalls, dass sie konfiguriert, aktualisiert und verwaltet werden müssen. Einige Verwaltungsaufgaben können zwar vom Master übernommen werden, andere hingegen müssen in der Konfiguration festgehalten werden. Es gibt verschiedene Möglichkeiten, Agenten zu installieren. Die benötigte Software kann innerhalb eines Maschinenimages bereitgestellt oder manuell über eine SSH-Verbindung installiert werden. Einige Werkzeuge und Cloud-Provider bieten Bootstrapping-Mechanismen an, mit denen Agenten automatisch installiert werden.

Ein Vorteil bei der Nutzung von Agenten besteht darin, dass sie die erforderliche Konfiguration vom Master abrufen können. Der Agent ist dann selber dafür verantwortlich, dass die in der Konfiguration beschriebenen Schritte ausgeführt werden. Dadurch wird sichergestellt, dass die Knoten korrekt konfiguriert sind und bleiben, was eine problemlose

Skalierung der Cloud-Infrastruktur ermöglicht. Infrastrukturwerkzeuge die einen Pull-basierten Ansatz verfolgen, brauchen in den meisten Fällen einen Master.

Master-Agent-Systeme haben den Vorteil, dass sie komplexere Sicherheitsmechanismen, wie asymmetrische Verschlüsselung, verwenden können. Daher kann es für Organisationen mit strengen Sicherheitsrichtlinien, wie Versicherungen, Regierungen und Banken, sinnvoll sein, ein Master-Agent-System zu nutzen.

Einige Werkzeuge bieten die Option, einen Agenten ohne dazugehörige Master-Instanz zu verwenden. In diesem Modus sind die Agenten jedoch meist auf das Anwenden von lokalen Konfigurationen beschränkt. Dieser Modus kann für kleinere Änderungen auf einzelnen Knoten sinnvoll sein, ist jedoch nicht geeignet, um eine große Anzahl von Knoten zu verwalten.

Im Allgemeinen gilt bei der Betrachtung von Infrastructure as Code Systemen, dass mehr Komponenten zu mehr Komplexität und damit auch mehr Fehlerquellen führen. Bei auftretenden Fehlern ist es oft nicht sofort ersichtlich, ob der Fehler beim Master, dem Agenten oder in der Konfiguration liegt.

4.2 GPL vs. DSL

In diesem Abschnitt werden die Unterschiede von Allzweck-Programmiersprachen (General Purpose Language - GPL) und domänenspezifischen Sprachen (Domain Specific Language - DSL) und deren Auswirkungen auf das Erstellen und Verwalten von Infrastrukturkonfigurationen erörtert.

DSLs sind einfacher zu erlernen, jedoch weisen sie gewisse Einschränkungen in ihren Möglichkeiten auf. Der Vorteil von DSLs liegt in ihrer klaren und präzisen Struktur, da sie für eine spezifische Domäne entworfen werden. Oftmals gibt es bei Werkzeugen, die eine DSL nutzen genau einen Weg, um ein Ziel zu erreichen. Dies führt zu einheitlichem Code, der leicht lesbar und verständlich ist. Werkzeuge, die eine DSL als Konfigurationssprache verwenden, sind auch für Anwender mit begrenzten Programmierkenntnissen zugänglich.

GPLs bieten den Vorteil, dass sie von Softwareentwicklern genutzt werden können die bereits Erfahrung mit der entsprechenden Sprache haben. Das ist ein großer Vorteil, da keine Ressourcen zum Erlernen einer neuen Sprache aufgebracht werden müssen. Die

Entwickler können ihre Programmierkenntnisse nutzen, um zusätzlich zur Softwareentwicklung auch Aufgaben im Bereich der Infrastrukturverwaltung zu übernehmen. Darüber hinaus ermöglichen GPLs die Nutzung aller Werkzeuge und Bibliotheken, die in dieser Sprache verfügbar sind. Die Erfahrung aus der Softwareentwicklung kann daher auf die Infrastrukturverwaltung übertragen werden. Des Weiteren bieten GPLs aufgrund ihrer Funktionalitäten, wie bedingten Anweisungen und Schleifen, die in der Softwareentwicklung üblich sind, auch für Infrastrukturkonfigurationen mehr Möglichkeiten.

Zusammenfassend lässt sich festhalten, dass sich die Auswahl vor allem auf die Zielgruppe der Nutzer auswirkt, jedoch beide Ansätze von Nutzern mit unterschiedlichen Hintergründen verwendet werden können. Dennoch ist das Erlernen einer DSL in der Regel einfacher.

4.3 Imperativ vs. Deklarativ

Die Entscheidung, ob ein Werkzeug mit imperativem oder deklarativem Ansatz genutzt wird, hat Auswirkungen auf die tägliche Nutzung und stellt somit eine der wichtigeren Entscheidungen dar, die im Kontext der betrachteten Kriterien getroffen werden muss. Imperative Konfigurationen beschreiben, *was* notwendig ist, um den gewünschten Zielzustand zu erreichen, während deklarative Konfigurationen beschreiben, *wie* der Zielzustand aussehen soll. Die klassische Softwareentwicklung folgt in der Regel dem imperativen Ansatz, weshalb dieser für Entwickler möglicherweise vertrauter wirkt.

Imperative Konfigurationen haben den Nachteil, dass der Zustand der Infrastruktur nicht immer vollständig im Quellcode abgebildet wird. Das liegt daran, dass die Reihenfolge, in der eine Konfiguration angewendet wird, bekannt sein muss, um den exakten Zustand zu ermitteln [1]. Zudem bedeutet dies, dass bei Ausführung derselben Konfiguration unterschiedliche Zielzustände entstehen können, was sich negativ auf die Wiederverwendbarkeit der Konfiguration auswirkt. Die Wiederverwendbarkeit ist jedoch für Cloud-Infrastrukturen von entscheidender Bedeutung. Besonders beim Skalieren von Anwendungen müssen Systeme regelmäßig erstellt und zerstört werden, sollten dabei jedoch konsistent bleiben. Um sicherzustellen, dass imperative Konfigurationen idempotent sind, müssen oft explizite Schritte in die Konfiguration aufgenommen werden. Nur auf diese Weise kann sichergestellt werden, dass die Konfiguration unabhängig von der Anzahl ihrer Ausführungen und des aktuellen Zustands der Infrastruktur in die gewünschte Zielinfrastruktur überführt werden kann.

Beim deklarativen Ansatz spiegelt der Code auch den tatsächlichen Zustand der Infrastruktur wider, vorausgesetzt es werden keine Änderungen außerhalb des Werkzeuges durchgeführt. Ein Nachteil von deklarativen Konfigurationen liegt in der begrenzten Kontrolle über Konfigurationsdetails aufgrund der von den Werkzeugen verwendeten Abstraktion. Die Werkzeuge treffen unter Umständen Entscheidungen, die nicht vom Nutzer beeinflusst werden können. Das bedeutet, dass das Werkzeug vorgibt, was möglich ist, weshalb die Verwaltung externer Ressourcen oft mit erheblichem Aufwand verbunden ist.

Im Hinblick auf die Provisionierung von Cloud-Ressourcen haben deklarative Konfigurationen den Vorteil, dass Abhängigkeiten nur implizit definiert werden müssen. Im Gegensatz dazu erfordern imperative Konfigurationen beim Verwalten von Cloud-Infrastruktur eine genaue Einhaltung der definierten Reihenfolge von Abhängigkeiten.

Folgendes Beispiel verdeutlicht dies: Cloud-Volumes[1], die von virtuellen Maschinen benötigt werden, müssen vor der Bereitstellung dieser virtuellen Maschinen provisioniert werden. Diese Abhängigkeit muss bei imperativen Konfigurationen in der Regel explizit angegeben werden. Deklarative Werkzeuge hingehen, erkennen diese Abhängigkeit und können so die Erstellung und Zerstörung von Ressourcen in der korrekten Reihenfolge vornehmen.

Wie in Zeile 21 von Abbildung 4.1a zu sehen ist benötigt die virtuelle Maschine eine `volume_id`. Diese existiert jedoch erst nach Ausführung der Schritte in den Zeilen 1-9. Die Reihenfolge, in der erst das Volume und dann die virtuelle Maschine erstellt wird, ist hier explizit anzugeben. Im Vergleich dazu ist in Abbildung 4.1b zu sehen, dass in Zeile 7 die `digitalocean_volume.server_volume.id` genutzt wird, bevor die Ressource dafür in Zeile 10 definiert wird. Terraform erkennt diese Abhängigkeit und führt zuerst die Erstellung des Volumes aus und anschließend die Erstellung der virtuellen Maschine.

[1] logische Cloud-Laufwerke

```yaml
- name: Create new Block Storage
  community.digitalocean.digital_ocean_block_storage:
    state: present
    oauth_token: "{{ token }}"
    command: create
    region: fra1
    block_size: 10
    volume_name: server-volume
  register: volume

- name: Create a new Droplet
  community.digitalocean.digital_ocean_droplet:
    state: present
    oauth_token: "{{ token }}"
    name: "server"
    size: "s-1vcpu-1gb"
    region: "fra1"
    image: "ubuntu-18-04-x64"
    wait_timeout: 500
    ssh_keys: ["{{ key_id }}"]
    volumes: ["{{volume.id}}"]
```

(a) Ansible

```hcl
resource "digitalocean_droplet" "server" {
  image              = "ubuntu-18-04-x64"
  name               = "server"
  region             = "fra1"
  size               = "s-1vcpu-1gb"
  ssh_keys = [data.digitalocean_ssh_key.key.id]
  volume_ids = ["${digitalocean_volume.server_volume.id}"]
}

resource "digitalocean_volume" "server_volume" {
  region                  = "fra1"
  name                    = "server-volume"
  size                    = 100
  initial_filesystem_type = "ext4"
  description             = "volume for the server"
}
```

(b) Terraform

Abbildung 4.1: Vergleich Abhängigkeiten Imperativ (a) und Deklarativ (b)

Bei der Zerstörung von Ressourcen verhält es sich ähnlich. Ob ein Cloud-Volume, das noch innerhalb einer VM eingebunden ist, gelöscht werden kann, hängt bei imperativen Konfigurationen vom zur Verwaltung der Ressourcen genutzten Modul des jeweiligen Werkzeugs ab. Hierbei gibt es zwei Möglichkeiten:

1. Das Modul erkennt die Abhängigkeit automatisch und hängt das Volumen selber aus.

2. Das Modul erkennt die Abhängigkeit nicht. Dann muss das Aushängen in die Konfiguration aufgenommen werden.

In deklarativen Konfigurationen trifft der erste Fall zu, da das Werkzeug sicherstellt, dass die nötigen Bedingungen zum Löschen des Volumes gegeben sind.

Mittlerweile verfolgen die Werkzeuge oft einen hybriden Ansatz, der die Vermischung von deklarativen und imperativen Konfigurationen ermöglicht. Ansible unterstützt diesen Ansatz. Ob eine Konfiguration imperativ oder deklarativ ist, hängt von den verwendeten Modulen ab. Folgendes Beispiel[2] verdeutlicht dies: In Abbildung 4.2a ist ersichtlich, wie

[2] Dieses Beispiel lässt sich auf jedes Konfigurationsmanagementwerkzeug übertragen, da sie alle ein Modul zur Ausführung einfacher Shell-Befehle anbieten.

der gewünschte Zielzustand beschrieben ist. In Abbildung 4.2b hingegen sind die expliziten Schritte angegeben, die zum Erreichen des Zielzustandes erforderlich sind.

```
1  - name: Install nginx web server
2    hosts: web_servers
3    tasks:
4      - name: Install web server package
5        package:
6          name: nginx
7          state: present
```

(a) Ansible - Deklarativ

```
1  - name: Check if nginx is installed
2    shell: dpkg -l nginx
3    register: nginx_installed
4
5  - name: Install nginx
6    shell: apt-get install nginx
7    when: nginx_installed.rc ≠0
```

(b) Ansible - Imperativ

Abbildung 4.2: Vergleich Imperativ (a) und Deklarativ (b)

Auch bei der Entscheidung zwischen dem deklarativen und imperativem Ansatz spielt der Anwendungsfall oft eine entscheidende Rolle. Der gewählte Ansatz kann je nach Anwendungsfall positive oder negative Auswirkungen auf die Lesbarkeit, Wiederverwendbarkeit und Komplexität der Konfigurationen haben.

Zusammenfassend lässt sich sagen, dass imperative Konfigurationen mehr Wissen erfordern, da jeder Schritt, der zum Erreichen des Zielzustandes erforderlich ist, explizit angegeben werden muss. Das bedeutet jedoch auch, dass eine erhöhte Kontrolle über die Konfiguration möglich ist. Darüber hinaus, lässt sich der Quellcode einer imperativen Konfiguration auch ohne bestehendes Wissen über die Infrastruktur gut nachvollziehen, da jeder Schritt und somit auch alle Details genau beschrieben sind. Gerade für kleinere Änderungen ist der imperative Ansatz besser geeignet, da solche Änderungen einfach umgesetzt werden können.

Deklarative Konfigurationen benötigen weniger Wissen, da der gewünschte Zielzustand beschrieben wird und sich das jeweilige Werkzeug darum kümmert, diesen Zustand zu erreichen. Daraus folgt jedoch, dass die Kontrolle über die Konfiguration eingeschränkt ist, da der Nutzer oftmals keinen Einfluss auf die Funktionsweise der genutzten Module hat. Des Weiteren sind deklarative Konfigurationen idempotent und dadurch wiederverwendbar. Einige Aufgaben können durch den deklarativen Ansatz jedoch komplizierter werden. Deklarative Konfigurationen sind in der Regel kompakter als ihr imperatives Gegenstück und wirken übersichtlicher, unter der Voraussetzung, dass bereits bestehendes Wissen über den Quellcode, Anwendungsfall und die Domäne vorhanden sind.

Die aktuellen Werkzeuge unterstützen mittlerweile beide Ansätze, oft mit gewissen Einschränkungen, die stärker in eine der beiden Richtungen tendieren. Dieser hybride Ansatz hat den Vorteil, dass beide Konzepte gemischt werden können. Daher ist die Entscheidung, welchen der beiden Ansätze eine Organisation verfolgt, heute nicht mehr so bedeutend wie früher. Vielmehr spielt die Art und Weise, wie das Werkzeug angewendet wird und in welchem Maße die Konzepte miteinander kombiniert werden, eine entscheidende Rolle. Welcher der zwei Ansätze dann konkret ausgewählt wird, kann vom Anwendungsfall abhängig gemacht werden.

4.4 Veränderlich vs. Unveränderlich

Ob bei der Verwaltung von Infrastruktur der veränderliche oder unveränderliche Ansatz gewählt wird, ist eine wichtige Entscheidung mit Auswirkung auf die Konzeption, Entwicklung und Wartung der Infrastruktur. Veränderliche Infrastruktur kann nach der Bereitstellung angepasst werden. Bei unveränderlicher Infrastruktur werden Ressourcen vollständig ersetzt, um Konfigurationsänderungen vorzunehmen. In diesem Abschnitt werden die Vor- und Nachteile dieser beiden Paradigmen gezeigt.

Veränderliche Infrastruktur ist flexibel. Änderungen können an laufenden Instanzen vorgenommen werden, was schnelle und einfache Aktualisierungen ermöglicht. Zudem bietet dieser Ansatz eine feinere Kontrolle, da Anpassungen an einzelnen Komponenten möglich sind. Ein Nachteil, der sich daraus ergibt, ist, dass solche Änderungen schneller zu Inkonsistenzen, also Configuration Drift führen. Des Weiteren besteht veränderliche Infrastruktur aus einer Historie von Änderungen. Diese Historie kann jedoch durch Fehler beeinträchtigt werden, wodurch die Ausgangsversion in verschiedene Unterversionen divergiert. Dadurch wird es schwierig, den exakten Zustand der aktuellen Infrastruktur zu bestimmen.

Unveränderliche Infrastruktur bündelt die gesamte Konfiguration innerhalb eines Maschinenabbildes. Änderungen an der Infrastruktur werden durch das Zerstören alter und Erstellen neuer Ressourcen erzielt. Dies kann zu zusätzlichen Kosten führen, da teilweise mehrere Instanzen gleichzeitig laufen müssen, um Ausfallzeit zu vermeiden. Besonders für kleinere, weniger komplexe Infrastrukturen ist veränderliche Infrastruktur oft kosteneffizienter. Dadurch, dass bei unveränderlicher Infrastruktur Maschinenabbilder genutzt werden, welche die gesamte Konfiguration beinhalten, kann mithilfe von Versionierung genau nachvollzogen werden, wie der aktuelle Stand der Infrastruktur ist.

Zusätzlich ist durch die Versionierung ein Zurücksetzen zu einer früheren Version möglich. Der unveränderliche Ansatz ermöglicht eine einfache und schnelle Replikation von Ressourcen. Ein Nachteil ist, dass die Maschinenabbilder erstellt und getestet werden müssen bevor eine Aktualisierung ausgerollt werden kann. Besonders bei großen Abbildern kann ein Deployment dadurch mehr Zeit in Anspruch nehmen. Unveränderliche Infrastruktur erfordert außerdem eine externe Datenverwaltung, um Datenverlust trotz regelmäßiger Erstellung und Zerstörung der Ressourcen zu vermeiden.

Auch bei unveränderlicher Infrastruktur kann es zu Configuration Drift kommen, da es Aspekte einer Konfiguration gibt, die sich nicht innerhalb eines Maschinenabbildes festhalten lassen. Zum Beispiel dann, wenn Konfigurationen dynamische Informationen wie die IP-Adressen von provisionierten Cloud-Ressourcen benötigen. Die Abweichung zwischen den Konfigurationen ist jedoch in der Regel geringer als bei veränderlicher Infrastruktur, da der Großteil der Konfiguration vom Maschinenabbild abhängt.

Aktualisierungsprozess

Um die Unterschiede zwischen den beiden Paradigmen zu verstehen, ist es hilfreich, den Aktualisierungsprozess und die damit verbundene Bereitstellung von Infrastrukturkomponenten zu betrachten. Das folgende Beispiel verdeutlicht die Unterschiede dieser Prozesse:

Die Version des Webservers muss aktualisiert werden. In einer veränderlichen Infrastruktur genügt es, eine Aktualisierung auf der laufenden Instanz durchzuführen. Dies kann beispielsweise mithilfe eines Konfigurationsmanagementwerkzeuges geschehen. Das Werkzeug kann den Paketmanager nutzen oder ein Artefakt mit der neuen Version auf das Zielsystem kopieren. Anschließend wird der Webserver neu gestartet.

Im Fall einer unveränderlichen Infrastruktur muss zunächst ein neues Maschinenabbild erstellt werden, das die neue Version des Webservers enthält. Danach muss die Konfiguration des Provisionierungswerkzeuges angepasst werden, damit das neue Serverabbild als Grundlage für die virtuelle Maschine dient. Das Werkzeug zerstört daraufhin die alte VM und provisioniert eine neue VM basierend auf dem neuen Maschinenabbild.

In der Regel ist eine Aktualisierung von veränderlicher Infrastruktur schneller. Der beschriebene Aktualisierungsprozess wird in Abbildung 4.3 grafisch veranschaulicht.

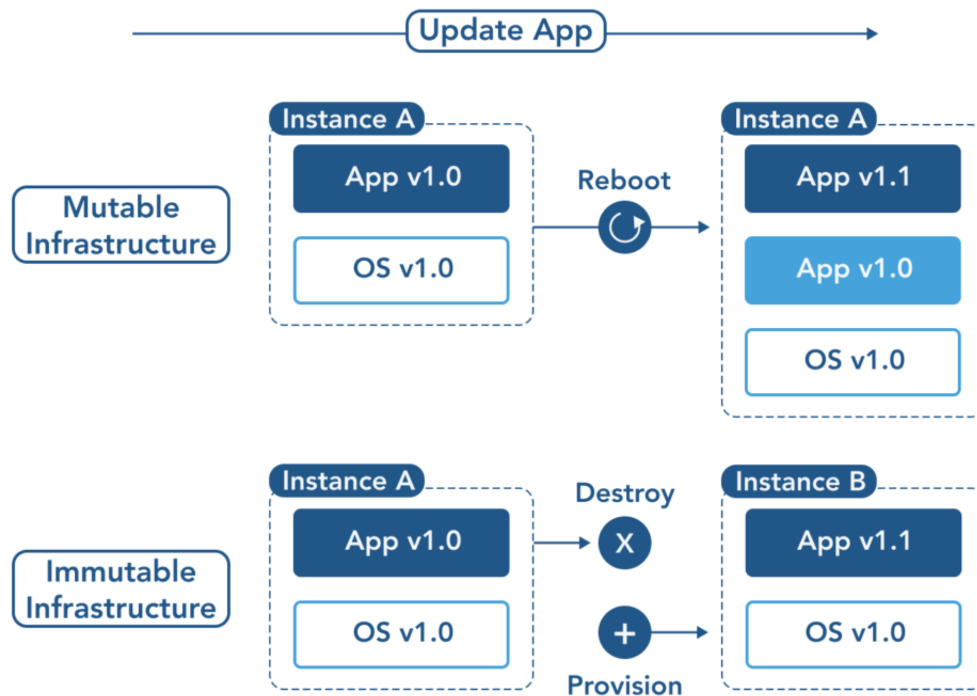

Abbildung 4.3: Veränderliche vs. unveränderliche Infrastrukturaktualisierung

Source: https://www.opsramp.com/guides/why-kubernetes/infrastructure-as-code/

Es gibt für unveränderliche Infrastruktur zwei Strategien hinsichtlich der Reihenfolge in der Ressourcen erstellt oder zerstört werden.

1. Erst zerstören dann provisionieren: Der Webserver bleibt so lange unerreichbar, bis die neue Instanz provisioniert wurde.

2. Erst provisionieren dann zerstören: Sobald die Instanz vollständig gestartet ist, wird der Verkehr auf diese umgeleitet. Dadurch entsteht zwar keine Ausfallzeit, jedoch eine erhöhte Ressourcennutzung und die damit verbundenen Kosten.

Darüber hinaus gibt es verschiedene Faktoren, die den Aktualisierungsprozess von unveränderlicher Infrastruktur stören können. Dazu gehören Netzwerkprobleme, DNS-Fehler oder auch das Ausfallen von Abhängigkeitsverzeichnissen. Diese Störungen können dazu führen, dass das System nur teilweise aktualisiert wird und sich die Infrastruktur zwischen dem aktuellen und dem gewünschten Zustand befindet [10]. Bei veränderlicher

Infrastruktur muss daher auf allen Zielsystemen überprüft werden, ob die Änderungen korrekt ausgeliefert wurden. Für kleinere Infrastrukturen mag dies keine große Herausforderung sein, jedoch steigt der Aufwand dieser Überprüfung proportional zur Anzahl der Komponenten, die von der Aktualisierung betroffen sind.

Bei unveränderlicher Infrastruktur ist eine solche Überprüfung nicht notwendig. Das liegt daran, dass die Konfiguration nicht auf einer Abfolge von Änderungen basiert, sondern auf einem Maschinenabbild, bei dem durch Tests sichergestellt wurde, dass es die Änderungen enthält. Sofern es keine Fehler bei der Provisionierung der Ressourcen gab, kann davon ausgegangen werden, dass sich die Infrastruktur so verhält, wie zum Zeitpunkt des Tests.

Testbarkeit

Die Testbarkeit der Infrastruktur ist ein weiterer wichtiger Faktor bei der Entscheidung für einen der Ansätze. Das Testen stellt sicher, dass Konfigurationsänderungen die Infrastruktur in den gewünschten Zielzustand überführen. Dabei sollte die getestete Infrastruktur möglichst genau das Produktivsystem widerspiegeln. Tests, die auf einem Testsystem durchgeführt werden, können sich unter Umständen auf einem Produktivsystem anders verhalten [1]. Daher ist es erforderlich, eine konsistente Replikation der Produktivinfrastruktur zu erstellen. In der Praxis stellt dies oft eine Herausforderung dar, insbesondere wenn es nicht dokumentierte Änderungen am Produktivsystem gegeben hat. Selbst minimale Anpassungen, können zu Inkonsistenzen führen, die erhebliche Auswirkungen auf die Testergebnisse haben können.

Bei unveränderlicher Infrastruktur ist eine konsistente Replikation in den meisten Fällen einfacher, da nach der Provisionierung nur noch kleinere Anpassung vorgenommen werden müssen. Diese Eigenschaft führt dazu, dass unveränderliche Infrastruktur in der Regel einfacher zu testen ist als veränderliche. Bei unveränderlicher Infrastruktur verhält sich die getestete Version auf allen Zielsystemen identisch, vorausgesetzt es gab keinen Fehler bei der Provisionierung.

Ein weiterer Vorteil den unveränderliche gegenüber veränderlicher Infrastruktur hat, ist die bessere Skalierbarkeit. Die Fehleranfälligkeit bei der Konfiguration von veränderlicher Infrastruktur kann die Skalierbarkeit negativ beeinflussen. Des Weiteren stellt die erforderliche Konfigurationszeit einen Nachteil im Vergleich zum unveränderlichen Ansatz dar, bei dem die Konfiguration nach der Provisionierung direkt verfügbar ist. Da

das automatische Skalieren eine häufige Anforderung für Cloud-Infrastrukturen ist, hat der unveränderliche Ansatz in dieser Hinsicht klare Vorteile.

Folgende Übersicht fasst die Vor- und Nachteile der zwei Paradigmen zusammen:

Veränderliche Infrastruktur	
Vorteile	Nachteile
+ Feine Kontrolle + Schnelle Aktualisierung + Anpassung an spezifische Anforderungen möglich + Flexibilität + Ressourceneffizient	- Konfigurationszeit - Versionierung schwierig - Inkonsistenzen durch fehlerhafte Aktualisierungen - Unvorhersehbarer Zustand
Unveränderliche Infrastruktur	
Vorteile	Nachteile
+ Versionierung einfach	- Externe Datenverwaltung notwendig
+ Weniger Configuration Drift	- Längere Aktualisierung
+ Skalierbarkeit	- Generalisierung von Problem kann zusätzliche Komplexität einführen
+ Reproduzierbarkeit	- Aktualisierungen können zu Ausfallzeit führen
+ Testbarkeit	- Aktualisierungen können zu erhöhter Ressourcennutzung führen
+ Vorhersehbarer Zustand	- Maschinenabbilder müssen gespeichert und verwaltet werden
+ Weniger Fehleranfällig beim Deployment	
+ Einfache Rollbacks	

Zwischenfazit

Abschließend lässt sich festhalten, dass beide Ansätze Vor- und Nachteile aufweisen. Je nach Anforderungen an die Infrastruktur muss entschieden werden, welcher Ansatz besser geeignet ist. Veränderliche Infrastruktur bietet aufgrund ihrer Flexibilität insbesondere dann Vorteile, wenn regelmäßig kleinere Anpassungen an der Infrastruktur vorgenommen

werden. Unveränderliche Infrastruktur erweist sich hingegen als vorteilhaft, wenn Konsistenz, Zuverlässigkeit und Skalierung von großer Bedeutung sind. Bei der Entscheidung ist auch das Design der auf der Infrastruktur laufenden Anwendungen zu berücksichtigen. Zustandsbehaftete Anwendungen, die eine persistente Datenspeicherung erfordern, können in einer unveränderlichen Infrastruktur, in der Instanzen häufig ersetzt werden, eine Herausforderung darstellen.

4.5 Community

Der Einfluss der Community und des Ökosystems[3] eines Werkzeuges auf seine Nutzung ist signifikant. Die Verfügbarkeit von Anleitungen, nutzbaren Modulen und wiederverwendbaren Konfigurationen hängt stark von der Aktivität der Community ab. Ein weit verbreitetes Werkzeug erleichtert den Erhalt von Unterstützung und beeinflusst die Geschwindigkeit, mit der Support-Anfragen beantwortet werden. Besonders in Organisationen, die bisher keine Erfahrung mit Infrastructure as Code haben, spielt die Community eine entscheidende Rolle bei der Einführung von Infrastrukturautomatisierung. Die Community kann die Zeitspanne bis zur Erreichung einer produktiven Nutzung maßgeblich beeinflussen.

Wie der Vergleich gezeigt hat, ist anzunehmen, dass Ansible und Terraform derzeit die populärsten Werkzeuge sind. Bei der Umsetzung des Anwendungsbeispiels waren Ressourcen für diese beiden Werkzeuge am leichtesten zugänglich. Es gab zahlreiche Beispielimplementierungen für den gewählten Anwendungsfall.

Stack Overflow spiegelt zwar nicht die gesamte Community wider, dennoch stellt es eine bedeutende Quelle für Unterstützung und Ressourcen dar. Je mehr Fragen zu einem Werkzeug auf Stack Overflow gestellt wurden, desto einfacher ist es, Fehlerquellen zu identifizieren oder Lösungen für häufig auftretende Probleme zu finden.

Die Anzahl der Mitwirkenden an Open-Source-Werkzeugen beeinflusst die Geschwindigkeit, mit der neue Funktionen hinzugefügt und Fehler behoben werden. In der dynamischen IT-Branche ist stetige Weiterentwicklung unerlässlich, um Probleme effektiver zu bewältigen. Besonders in vergleichsweise neuen Bereichen wie Infrastructure as Code ist die Auswahl eines Werkzeuges, das kontinuierlich weiterentwickelt wird, von großer

[3] Gesamtheit der verschiedenen Komponenten und Ressourcen, die zu einem bestimmten Werkzeug gehören.

Bedeutung. Zudem führt eine aktive Community zu einem breiten Ökosystem mit zahlreichen Werkzeugen und Integrationen, die bei der Entwicklung von Infrastrukturkonfigurationen von Nutzen sind.

Insgesamt lässt sich sagen, dass die Community eines Werkzeuges essenziell für die Weiterentwicklung, den Wissensaustausch und die Zusammenarbeit ist. In der sich ständig wandelnden IT sind diese Faktoren von großer Bedeutung und haben einen erheblichen Einfluss auf die Werkzeugnutzung. Eine aktive Community führt zu einer flacheren Lernkurve und gesteigerter Produktivität aufgrund des gemeinsamen Wissensaustauschs.

4.6 Schlüsselergebnisse

Der folgende Abschnitt fasst die Schlüsselergebnisse der Evaluation der IaC-Werkzeuge zusammen. In der Arbeit wurden verschiedene Aspekte der Werkzeuge analysiert, darunter ihre Konzepte und die daraus resultierenden Vor- und Nachteile. Die gewonnenen Erkenntnisse bieten wertvolle Einsichten in die Eigenschaften und Einsatzmöglichkeiten der Werkzeuge.

Welche der Lösungen sich am besten für eine Nutzung im Cloud-Umfeld eignet, lässt sich nicht durch die Angabe eines konkreten Werkzeuges beantworten. Die Antwort ist vielmehr ein Prozess, der durchlaufen werden muss, um eine fundierte Entscheidung zu treffen, die zu den Anforderungen einer Organisation oder eines Projektes passt. In dieser Arbeit wurden mehrere Faktoren dieser Entscheidung und deren Auswirkungen vorgestellt. Diese gesammelten Erkenntnisse helfen dabei, ein geeignetes Infrastructure as Code Werkzeug auszuwählen.

Die verglichenen Werkzeuge ähneln sich in ihrer Funktionsweise und den unterstützten Funktionen. Mittlerweile sind die Werkzeuge flexibler geworden und bieten mehrere der vorgestellten Paradigmen an. Auch wenn es nicht vorgesehen ist, lassen sich Konzepte erzwingen und vermischen. Zudem ist aufgrund der ähnlichen Funktionsweise eine Migration zwischen den verschiedenen Werkzeugen gut möglich. Die Umsetzbarkeit einer Migration hängt von der Größe der Infrastruktur sowie den Ressourcen eines Teams ab und sollte stufenweise durchgeführt werden.

Eine wichtige Erkenntnis des Vergleiches liegt darin, dass die Werkzeuge am stärksten sind, wenn sie je nach Anwendungsfall in Kombination miteinander genutzt werden, um

die jeweiligen Stärken auszunutzen und den Schwächen entgegenzuwirken. Alle Werkzeuge haben aufgrund ihrer verschiedenen Ansätze Vor- und Nachteile hinsichtlich der Nutzung für Cloud-Infrastruktur. Daher ist es sinnvoll, die Werkzeuge miteinander zu kombinieren. Die Ansätze lassen sich zu hybriden Lösungen vermischen, um so eine flexiblere Infrastrukturverwaltung zu ermöglichen. Es ist nicht länger notwendig, sich für ein einziges Paradigma zu entschieden, da es möglich ist, die verschiedenen Ansätze in Kombination zu nutzen.

Allgemeine Empfehlung

Generell ist eine Kombination aus einem Provisionierungs- und einem Konfigurationsmanagementwerkzeug eine gute Herangehensweise, um die Anforderungen der Cloud-Infrastrukturverwaltung zu bewältigen. Da Cloud-Infrastrukturen oft aus einer Vielzahl von Komponenten bestehen, sind sowohl die Zustandsverwaltung (State Management) als auch der deklarative Konfigurationsansatz von Provisionierungswerkzeugen von Vorteil. Konfigurationsmanagementwerkzeuge eignen sich besser dafür, benötigte Abhängigkeiten und Software auf diesen Ressourcen zu installieren oder Einstellungen am System vorzunehmen. Provisionierungswerkzeuge haben hierfür in der Regel nur wenige Möglichkeiten, die sich auf die Ausführung von Shell-Skripten oder einfachen Befehlen beschränken. Server-Templating-Werkzeuge können ergänzend genutzt werden, um Standardkonfigurationen in Maschinenabbildern festzuhalten. Diese Abbilder können aufgrund der Integrationsmöglichkeiten der Templating-Werkzeuge, anhand von bereits bestehenden Konfigurationen aus Konfigurationsmanagementwerkzeugen erstellt werden. Die daraus resultierenden Maschinenabbilder können zum Erstellen von neuen Instanzen genutzt werden.

Ein guter Startpunkt bei der Einführung von Infrastrukturverwaltung sind Terraform und Ansible. Aufgrund ihrer Verwendung einer DSL als Konfigurationssprache, sowie einer aktiven Community bieten beide Werkzeuge eine flache Lernkurve. Zu beiden Werkzeugen existieren zahlreiche Ressourcen für verschiedene Anwendungsfälle. Diese Ressourcen beschränken sich dabei nicht nur auf die einzelnen Werkzeuge, sondern schließen auch die Kombination der beiden ein. Dadurch ist eine schnelle und produktive Einführung von Infrastructure as Code innerhalb eines Projektes möglich. Optional kann Packer als Server-Templating-Werkzeug genutzt werden, da es dieselbe DSL wie Terraform verwendet und eine gute Integration zu Ansible bietet. Der Prozess bei dem diese drei Werkzeuge in Kombination genutzt werden ist in Abbildung 4.4 dargestellt.

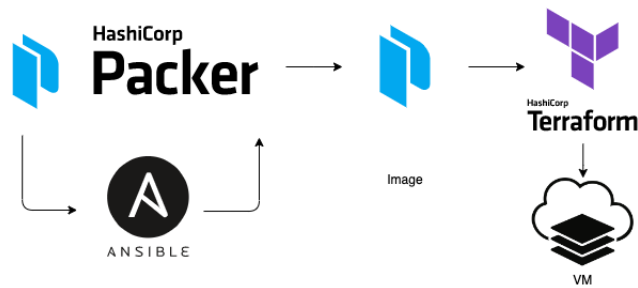

Abbildung 4.4: Kombination aus Packer, Ansible und Terraform

Zunächst wird eine Packer-Konfiguration erstellt, die anhand eines Ansible-Playbooks ein Maschinenabbild erzeugt. Dieses Abbild wird anschließend von Terraform als Basis für die Provisionierung einer virtuellen Maschine verwendet.

Wenn die vorgestellten Ansätze mit den Konzepten von Containern und der damit verbundenen Container-Orchestrierung verknüpft werden, ermöglicht dies eine effiziente und umfassende Verwaltung sowohl der Cloud-Infrastruktur als auch der Applikationsinfrastruktur durch Quellcode. Die Applikationsinfrastruktur wird dabei durch Container-Orchestrierungswerkzeuge bereitgestellt, die eine Plattform bieten, um die automatisierte Bereitstellung, Verwaltung, Skalierung und Vernetzung von Containern zu ermöglichen [32].

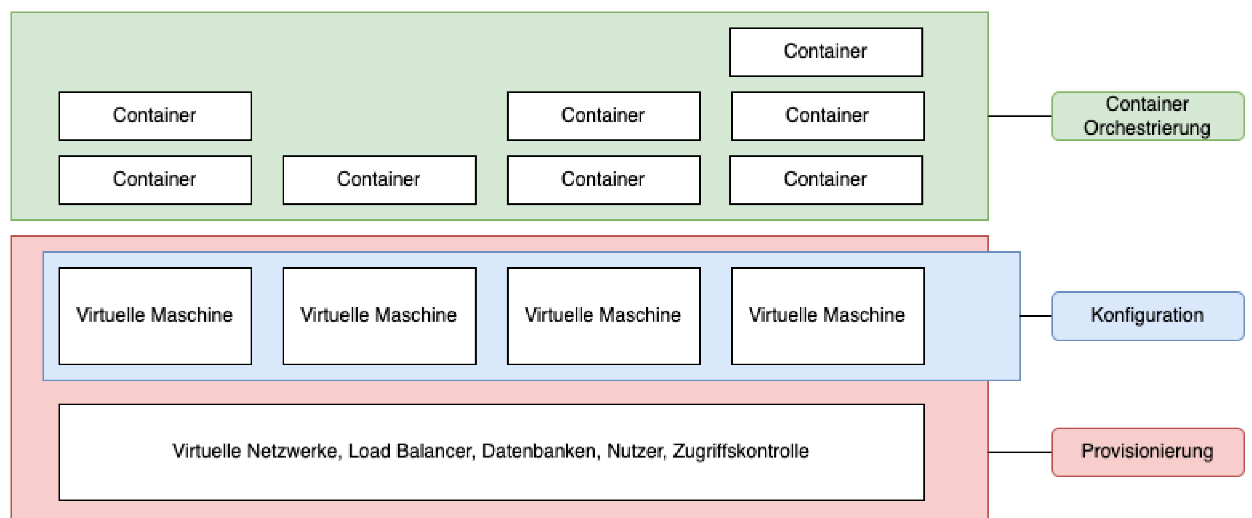

Abbildung 4.5: Verschiedene Ebenen der Infrastrukturverwaltung

Wie in Abbildung 4.5 zu sehen ist, lassen sich die verschiedenen Werkzeuge so kombinieren, dass eine Infrastrukturverwaltung von der Hardware- bis zur Applikationsebene möglich ist. Auf diese Weise können die Vorteile von Containern genutzt werden, was den Aufwand für die Infrastrukturverwaltung verringert, da Teile dieser Verwaltung von den Containern übernommen werden. Dadurch wird nicht nur die benötigte Cloud-Infrastruktur im Quellcode ersichtlich, sondern auch die Applikationen, die auf ihr ausgeführt werden.

Die Implementierung eines Prototyps ist ebenfalls ein hilfreicher Schritt bei der Entscheidung für die richtigen Werkzeuge. Dadurch lässt sich die Auswahl eines Werkzeuges oder einer Kombination von Werkzeugen validieren. Falls das ausgewählte Werkzeug beziehungsweise die ausgewählten Werkzeuge im Rahmen des Prototyps als geeignet erachtet werden, kann eine schrittweise Migration zur Infrastrukturverwaltung mit IaC durchgeführt werden. Im Idealfall können Teile des Prototyps sogar als Produktivcode wiederverwendet werden. Andernfalls ist es möglich, zu evaluieren, welche Anforderungen das ausgewählte Werkzeug nicht erfüllen kann. Basierend auf diesen Erkenntnissen kann dann eine alternative Werkzeugauswahl getroffen werden.

5 Fazit und Ausblick

5.1 Fazit

Dieses Kapitel bildet den Abschluss der Arbeit und fasst die Ergebnisse hinsichtlich der Fragestellung zusammen. Der Vergleich der IaC-Werkzeuge hat gezeigt, dass diese einen wichtigen Beitrag zur Automatisierung und Vereinfachung der Infrastrukturverwaltung leisten können. Die Wahl des richtigen Werkzeuges erfodert eine sorgfältige Analyse der Anforderungen, sowie einer Abwägung der Vor- und Nachteile.

In der Arbeit wurden verschiedene Infrastructure as Code Werkzeuge hinsichtlich ihrer Eignung für die Verwaltung von Cloud-Infrastruktur untersucht. Dafür wurden der Installationsprozess, die grundlegende Funktionsweise, die Konfigurationssprache und die wichtigsten Konzepte der Werkzeuge analysiert. Anschließend wurden die verschiedenen Paradigmen und Ansätze sowie deren Auswirkungen auf die Infrastrukturverwaltung evaluiert. Anhand dieser Evaluation kann eine fundierte Entscheidung für die Werkzeuge getroffen werden.

Die Auswahl eines geeigneten Werkzeuges hängt von einer Vielzahl von Faktoren ab, darunter die Anforderungen eines Projekts, interne Richtlinien und die vorhandenen Programmierkenntnisse der Entwickler. Nichttechnische Faktoren, wie beispielsweise die Kosten einer Enterprise-Version oder strenge Sicherheitsanforderungen, können ebenfalls wichtige Kriterien bei der Auswahl sein. Die Anforderungen müssen bei der Auswahl eines Werkzeuges mit den verschiedenen Möglichkeiten der Werkzeuge abgeglichen werden.

Im Folgenden sind die verschiedenen Paradigmen und deren Auswirkung auf die Infrastrukturverwaltung noch einmal zusammengefasst:

- Die Topologie eines Werkzeuges hat vor allem Auswirkungen auf den Installationsprozess und die verfügbaren Sicherheitsmechanismen. Außerdem hat sie Auswirkungen darauf ob der Push- oder Pull-Ansatz genutzt wird und somit auch auf die Skalierbarkeit der Infrastruktur.

- Die verwendete Konfigurationssprache dient vor allem als Indikator für die Zielgruppe eines Werkzeuges. Allzweck-Programmiersprachen (GPLs) richten sich eher an Entwickler, während domänenspezifische Sprachen (DSLs) für Anwender ohne umfassende Programmierkenntnisse geeignet sind und eine flache Lernkurve bieten.
- Die Wahl zwischen imperativem und deklarativem Ansatz für die Konfiguration beeinflusst die Kontrolle, Lesbarkeit und Einfachheit einer Lösung. Die vorgestellten Werkzeuge setzen größtenteils auf ein hybrides Modell, bei dem beide Ansätze kombiniert werden können.
- Die Entscheidung für ein Werkzeug, das eine veränderliche oder unveränderliche Infrastruktur verwaltet, hat vor allem Auswirkungen auf die Konzeption, Skalierbarkeit, Testbarkeit und Fehlerbehandlung der Infrastruktur. In der Regel verwalten Konfigurationsmanagementwerkzeuge veränderliche Infrastruktur, während Provisionierungswerkzeuge sich auf unveränderliche Infrastruktur konzentrieren.
- Die Community spielt eine entscheidende Rolle hinsichtlich der verfügbaren Ressourcen für ein Werkzeug. Die Aktivität innerhalb der Community bestimmt auch die Geschwindigkeit, mit der Fehler behoben, neue Funktionen hinzugefügt und Fragen beantwortet werden. Diese Faktoren gewinnen besonders bei der Einführung von IaC und der damit verbundenen Lernkurve an Bedeutung.

Der Vergleich der Werkzeuge hat gezeigt, dass diese oft ähnliche Eigenschaften aufweisen und sich die Unterschiede auf spezifische Details beschränken. Grundsätzlich können mit allen Werkzeugen die gleichen Ergebnisse erzielt werden. Die Präferenzen der Entwickler, hinsichtlich der genutzten Paradigmen der Infrastrukturverwaltung, sind von entscheidender Bedeutung bei der Auswahl. Daher lässt sich sagen, dass alle betrachteten Werkzeuge für den Einsatz in der Cloud geeignet sind. Aufgrund der unterschiedlichen Aufgaben der Infrastrukturverwaltung eignen sich einige Werkzeuge besser für spezifische Aufgabenbereiche.

Provisionierungswerkzeuge bieten deutliche Vorteile bei der effizienten Verwaltung von Cloud-Ressourcen. Sie ermöglichen das schnelle Bereitstellen, Skalieren und Ersetzen von Infrastrukturkomponenten, was insbesondere in dynamischen Cloud-Umgebungen von großem Nutzen ist. Jedoch ist es wichtig zu beachten, dass Konfigurationsmanagementwerkzeuge eine ebenso wichtige Rolle spielen. Sie fokussieren sich auf die Verwaltung des

Zustands der bereitgestellten Ressourcen und ermöglichen die Anwendung von Konfigurationen, das Durchsetzen von Sicherheitsrichtlinien und das Gewährleisten der Konsistenz über die gesamte Infrastruktur hinweg.

Letztlich ergänzen sich diese beiden Ansätze in der Praxis häufig. Während Provisionierungswerkzeuge die Bereitstellung und Skalierung von Ressourcen erleichtern, ermöglichen Konfigurationsmanagementwerkzeuge die präzise Anpassung und Verwaltung der Konfigurationen. Zudem kann der Einsatz von Server-Templating-Werkzeugen hilfreich sein, um häufig verwendete Konfigurationen innerhalb von Maschinenabbildern bereitzustellen. Des Weiteren ermöglicht die Kombination der vorgestellten Konzepte mit Containern und deren Orchestrierung eine vollständige Verwaltung großer Infrastrukturen sowie deren Applikationen in Quellcode.

Anhand der in dieser Arbeit vorgestellten Faktoren und Kriterien, sowie deren Auswirkungen auf die Infrastrukturverwaltung, kann eine fundierte Auswahl für eine Kombination von Werkzeugen getroffen werden. Zusätzlich wurden auf Grundlage der Ergebnisse Werkzeugempfehlungen abgegeben. Als generelle Empfehlung wurden Terraform und Ansible genannt. Zum aktuellen Stand[1] sind dies die populärsten Lösungen, hinsichtlich der Aktivität und Größe ihrer Community. Die Werkzeuge haben jeweils eine flache Lernkurve, bieten eine gute Integration zueinander an und sind ausreichend für einen Großteil der Anforderungen von Cloud-Infrastrukturen.

Abschließend lässt sich feststellen, dass die Auswahl der geeigneten Werkzeuge ein agiler Prozess ist, bei dem mehrere Schritte nötig sind um eine passende Auswahl der Werkzeuge zu treffen. Es ist wichtig sich die evaluierten Kriterien und deren Auswirkung anzugucken und je nach Anwendungsfall und Organisationsvorgaben zu entscheiden, welche Kriterien die größte Gewichtung bekommen und welche Paradigmen für die Infrastrukturverwaltung genutzt werden sollen. Durch den hybriden Ansatz einiger Werkzeuge ist das Kombinieren verschiedener Paradigmen möglich. Daher können die Präferenzen der Entwickler hinsichtlich der verwendeten Konfigurationssprache einen entscheidenden Einfluss haben.

Des Weiteren kann eine prototypische Implementierung hilfreich sein, um die Auswahl der Werkzeuge zu validieren. Dadurch kann entweder die Auswahl des Werkzeuges als passend bestätigt werden oder es kann überprüft werden, welche Anforderungen das Werkzeug nicht erfüllt und diese Erkenntnis in die Auswahl eines anderen Werkzeuges einbeziehen.

[1] August 2023

5.2 Ausblick

Infrastructure as Code ist ein vergleichsweise junges Teilgebiet der Informatik. Es ist absehbar, dass die Infrastrukturautomatisierung weiterhin an Bedeutung gewinnen wird. Diese Entwicklung ist dabei unabhängig von der Entwicklung der Cloud, da Infrastructure as Code auch außerhalb der Cloud eingesetzt wird. Durch das Aufkommen von neuen Technologien werden sich die Anforderungen an die Infrastrukturverwaltung kontinuierlich verändern. In diesem Zusammenhang ergeben sich interessante Forschungsthemen und Fragen. Ein solches Thema ist die Anpassung der IaC-Werkzeuge an diese neuen Anforderungen.

Ein Aspekt, der in dieser Arbeit nicht behandelt wurde, jedoch von großer Relevanz ist, betrifft die Sicherheit im Kontext von IaC. Aspekte wie Zugriffskontrolle und die Verschlüsselung sensibler Daten stellen bedeutende Aufgaben der Infrastrukturverwaltung dar, insbesondere im Cloud-Umfeld. Forschungsarbeiten zu diesem Thema könnten zu einem sichereren Einsatz dieser Technologien beitragen. Auch die Beobachtbarkeit (Observability) und das Protokollieren (Logging) von Infrastrukturen sind von Relevanz, sowohl für die Sicherheit als auch für die Reaktionsfähigkeit, und bieten somit spannende Ansätze für weitere Forschung.

Eine weitere interessante Forschungsfrage ergibt sich aus der zunehmenden Abstraktion der Cloud-Dienste. Können Konzepte wie *Platform as a Service (PaaS)*, *Function as a Service (FaaS)* und *Software as a Service (SaaS)* so weiterentwickelt werden, dass sie die automatisierte Infrastrukturverwaltung ersetzen? Kann der Fortschritt im Bereich der künstlichen Intelligenz eine solche Entwicklung beeinflussen? Gegenwärtig scheint eine Ablösung von Infrastructure as Code, insbesondere aufgrund der Kosten der PaaS-, FaaS- und SaaS-Dienste, noch nicht realistisch.

Literaturverzeichnis

[1] Brikman, Yevgeniy: *Terraform: Up & Running*. O'Reilly Media. – ISBN 9781491924358

[2] Guerriero, Michele ; Garriga, Martin ; Tamburri, Damian A. ; Palomba, Fabio: Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, S. 580–589

[3] HashiCorp: *Provider Configuration*. – URL https://developer.hashicorp.com/terraform/language/providers/configuration. – Zugriffsdatum: 2023-06-17

[4] HashiCorp: *Terraform Glossary*. – URL https://developer.hashicorp.com/terraform/docs/glossary. – Zugriffsdatum: 2023-06-17

[5] HashiCorp: *Terraform Installation*. – URL https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli. – Zugriffsdatum: 2023-06-19

[6] HashiCorp: *Terraform Intro*. – URL https://developer.hashicorp.com/terraform/intro. – Zugriffsdatum: 2023-06-17

[7] HashiCorp: *Terraform Providers*. – URL https://developer.hashicorp.com/terraform/language/providers. – Zugriffsdatum: 2023-05-24

[8] HashiCorp: *Terraform State*. – URL https://developer.hashicorp.com/terraform/language/state. – Zugriffsdatum: 2023-06-17

[9] Morris, Kief: *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media. – ISBN 9781491924358

[10] OpsRamp: *An Introduction to Infrastructure as Code & Immutable Architecture*. – URL https://www.opsramp.com/guides/why-kubernetes/infrastructure-as-code/. – Zugriffsdatum: 2023-07-28

[11] Overflow, Stack: *Stack Overflow Survey 2023*. – URL https://survey.stackoverflow.co/2023/#section-most-popular-technologies-other-tools. – Zugriffsdatum: 2023-07-13

[12] Perforce: *Installing Puppet*. – URL https://www.puppet.com/docs/puppet/7/install_puppet.html. – Zugriffsdatum: 2023-02-27

[13] Perforce: *Puppet Architecture*. – URL https://www.puppet.com/docs/puppet/5.5/architecture.html. – Zugriffsdatum: 2023-02-27

[14] Perforce: *Puppet Glossary*. – URL https://www.puppet.com/docs/puppet/6/glossary.html. – Zugriffsdatum: 2023-02-27

[15] Perforce: *System configuration*. – URL https://www.puppet.com/docs/pe/2019.8/system_configuration.html. – Zugriffsdatum: 2023-02-27

[16] Perforce: *What is Puppet?*. – URL https://www.puppet.com/docs/puppet/8/what_is_puppet. – Zugriffsdatum: 2023-02-27

[17] Progress: *Bootstrap a Node*. – URL https://docs.chef.io/install_bootstrap/. – Zugriffsdatum: 2023-06-11

[18] Progress: *Chef Getting started*. – URL https://docs.chef.io/workstation/getting_started/. – Zugriffsdatum: 2023-02-27

[19] Progress: *Chef Glossary*. – URL https://docs.chef.io/glossary/. – Zugriffsdatum: 2023-06-11

[20] Progress: *Chef Infra Language*. – URL https://docs.chef.io/infra_language/. – Zugriffsdatum: 2023-06-11

[21] Progress: *Chef Infrastructure Management*. – URL https://www.chef.io/products/chef-infrastructure-management. – Zugriffsdatum: 2023-06-11

[22] Progress: *Chef Installation*. – URL https://docs.chef.io/server/install_server/. – Zugriffsdatum: 2023-06-11

[23] Progress: *Chef Overview*. – URL https://docs.chef.io/chef_overview/. – Zugriffsdatum: 2023-06-11

[24] PROGRESS: *Chef Security*. – URL https://docs.chef.io/chef_client_security/. – Zugriffsdatum: 2023-06-11

[25] PULUMI: *Pulumi How It Works*. – URL https://www.pulumi.com/docs/concepts/how-pulumi-works/. – Zugriffsdatum: 2023-05-24

[26] PULUMI: *Pulumi vs. Terraform*. – URL https://www.pulumi.com/docs/concepts/vs/terraform/. – Zugriffsdatum: 2023-05-24

[27] PULUMI: *Pulumi Webseite*. – URL https://www.pulumi.com/docs/concepts/glossary/. – Zugriffsdatum: 2023-05-24

[28] REDHAT: *Ansible Glossary*. – URL https://docs.ansible.com/ansible/latest/reference_appendices/glossary.html. – Zugriffsdatum: 2023-06-10

[29] REDHAT: *How Ansible Works*. – URL https://www.ansible.com/overview/how-ansible-works. – Zugriffsdatum: 2023-06-10

[30] REDHAT: *How to build your inventory*. – URL https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html#inventory. – Zugriffsdatum: 2023-02-25

[31] REDHAT: *Learning Ansible*. – URL https://www.redhat.com/en/topics/automation/learning-ansible-tutorial. – Zugriffsdatum: 2023-06-10

[32] REDHAT: *What is container orchestration?*. – URL https://www.redhat.com/en/topics/containers/what-is-container-orchestration. – Zugriffsdatum: 2023-07-29

[33] REDHAT: *Working with dynamic inventory*. – URL https://docs.ansible.com/ansible/latest/inventory_guide/intro_dynamic_inventory.html. – Zugriffsdatum: 2023-02-25

[34] SALT: *Salt Formulas*. – URL https://salt-formulas.readthedocs.io/_/downloads/en/latest/pdf/. – Zugriffsdatum: 2023-02-10

[35] SALT: *Salt in 10 Minutes*. – URL https://docs.saltproject.io/en/master/topics/tutorials/walkthrough.html. – Zugriffsdatum: 2023-02-10

[36] SALT: *SaltStack Communication*. – URL https://docs.saltproject.io/en/getstarted/system/communication.html. – Zugriffsdatum: 2023-02-10

[37] SALT: *SaltStack Glossary*. – URL https://docs.saltproject.io/en/latest/glossary.html. – Zugriffsdatum: 2023-02-10

[38] SALT: *Understanding SaltStack*. – URL https://docs.saltproject.io/en/getstarted/system/index.html. – Zugriffsdatum: 2023-06-10

[39] SERVICES, Amazon W.: *Was ist DevOps?*. – URL https://aws.amazon.com/de/devops/what-is-devops/. – Zugriffsdatum: 2023-08-07

Glossar

Agent Agenten sind Programme die auf den zu verwaltenden Knoten installiert werden. Sie laufen üblicherweise als Hintergrunddienst. Sie sind für das Empfangen und Ausführen von Befehlen verantwortlich.

Cloud-Provider Cloud-Provider oder Cloud-Anbieter stellen Cloud-Ressourcen bereit, die Organisationen auf Bedarf nutzen können.

Continous Delivery Kontinuierliche Auslieferung bezeichnet eine Sammlung von Techniken, Prozessen und Werkzeugen, die den Software-Auslieferungsprozess verbessern.

Continous Integration Kontinuierliche Integration beschreibt den Prozess des fortlaufenden Zusammenfügens von Komponenten zu einer Anwendung.

Idempotenz Idempotenz ist die Eigenschaft einer Operation, bei der sie mehrfach ausgeführt werden kann und dennoch das gleiche Ergebnis liefert wie bei ihrer ersten Ausführung.

Internet of Things Netzwerk von Computern, die in physische Geräte eingebettet sind.

Knoten Ein Knoten ist ein zu verwaltendes System. Dieses System kann dabei eine virtuelle Maschine, ein physischer Server oder ein anderes elektronisches Gerät sein.

Maschinenabbild Ein Maschinenabbild (Machine Image) ist eine einzelne, statische Einheit, welche vorkonfigurierte Betriebsysteme und Software enthalten. Dadurch können schnell neue Maschinen erstellt werden.

Master Die Master-Instanz ist im Kontext von IaC eine zentrale Instanz. In der Regel ist diese für die Verwaltung von Agenten verantwortlich.

Microservices Microservices bezeichnen ein Architekturmuster, bei dem entkoppelte Dienste über APIs miteinander kommunizieren und so zu einem Gesamtsystem zusammengesetzt werden. Dabei ist unabhängig in welcher Programmiersprache die einzelnen Dienste geschrieben sind.

Serverless Der Begriff Serverless bezieht sich auf ein Cloud-natives Entwicklungsmodell, bei dem Entwickler Anwendungen erstellen und ausführen können, ohne Server verwalten zu müssen. Einmal bereitgestellt, werden Serverless-Applikationen auf Bedarf ausgeführt und können automatisch nach oben oder unten skaliert werden.

Test Driven Development Test Driven Development, kurz TDD, ist ein Softwareentwicklungsprozess, bei dem Anforderungen in Testfälle umgewandelt werden, bevor der tatsächliche Code für diese Anforderungen geschrieben wird.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

____________	____________	________________________
Ort	Datum	Unterschrift im Original