Terraform

Das Praxisbuch für DevOps-Teams und Administratoren

Kapitel 2
Was ist Terraform?

Dieses Kapitel soll Ihnen Terraform und seinen Hintergrund näherbringen. Es soll erklären, wieso Terraform so beliebt geworden ist und was die *Cloud* und Schlagwörter wie *Infrastructure as Code* damit zu tun haben. Es geht nicht darum, möglichst viele der gerade angesagten *Buzzwords* in einem Kapitel anzusprechen. Vielmehr sollen Sie ein solides Fundament haben, um zu verstehen, welche Vorteile diese Herangehensweise hat, aber auch, welche Nachteile damit einhergehen. Sie sollen genügend Informationen erhalten, um für Sie und für Ihren Anwendungsfall oder Ihr Unternehmen den besten Weg zu finden. Auch wenn jeder Haustürschlüssel eine Haustür aufsperrt, passt doch nur ein bestimmter Schlüssel zu Ihrer Haustür. Und je nachdem, wie Ihre Erfahrungen, Ihre Ziele, Ihre Gegebenheiten und Ihre Vorstellungen sind, kann der eine oder andere Weg Ihnen das Leben erleichtern oder für zusätzliche Arbeit sorgen. Das gilt nicht nur für den Umgang mit Terraform, sondern auch für Terraform selbst. Eventuell ist ein anderes Tool sogar besser geeignet. Ziel ist auch hier, Sie zu befähigen, fundierte Entscheidungen über eben diese Fragen zu treffen.

2.1 Cloud Computing und seine Auswirkungen

Können Sie sich noch an die Zeiten vor der Cloud erinnern? Als jedes Unternehmen und jede Firma ihre eigene IT-Infrastruktur in Hardware betreiben musste? Als ein Serverraum oder Rechenzentrum zum guten Ton gehörten?

Seit damals ist noch nicht viel Zeit vergangen, doch kommt es einem manchmal so vor. Die Erfindung der Cloud hat die IT so massiv durcheinandergewirbelt wie viele andere Bereiche des täglichen Lebens auch. Konnte der Turnschuh-Admin vergangener Zeiten die Zahl der Server noch auswendig aufsagen, so ist dies heute nicht mehr möglich. Unternehmen wie Spotify, Netflix oder Dropbox, die IT-Dienstleistungen an Endkunden anbieten, verzichten auf eigene Rechenzentren und betreiben Tausende oder Hunderttausende von Servern. Niemand kann und will diese Server per Hand installieren und sich Hostnamen ausdenken.

Doch wie schaffen Sie es, Infrastruktur in dieser Größenordnung zu verwalten? Und was hat Terraform damit zu tun?

Der größte Vorteil des Cloud Computing ist die Flexibilität. Sie können auf Kapazitäten aus dem Fundus Ihres Cloud-Anbieters zugreifen und in kurzer Zeit Infrastruktur aufbauen, umbauen oder wieder abreißen. Um die Bestellung von Hardware müssen Sie sich nicht kümmern, hier ist Ihr Cloud-Anbieter in der Pflicht.

Doch ist es utopisch, anzunehmen, jemand könne manuell Ressourcen in dieser Größenordnung aufbauen, einrichten und pflegen.

2.2 Das Prinzip Infrastructure as Code

Ihre Aufgabe ist es, den Überblick über Ihre Infrastruktur zu behalten. Hier kommt das Prinzip des *Infrastructure as Code* zum Einsatz. In der Softwareentwicklung ist es mittlerweile längst Stand der Technik, den Quelltext (engl. *Code*) in einer Versionsverwaltung wie z. B. Git zu lagern. Änderungen werden nicht mehr »einfach so« vorgenommen, sondern über Entwicklungszweige (engl. *Branches*) und Funktionen wie *Pull Requests* kontrolliert eingebaut.

Der Quelltext wird nach allen Regeln der Kunst automatisiert geprüft und validiert, getestet und anschließend einem Peer Review unterzogen, d. h. von weiteren Personen begutachtet und freigegeben. Verfahren wie *Continuous Integration* und *Continuous Deployment* sorgen für kurze Entwicklungszyklen bei gleichzeitig geringerer Fehlerrate und höherer Codequalität.

Eng verbunden damit ist das Konzept der *Single Source of Truth*, also dem Prinzip, dass Informationen und Anweisungen in einem zentralen Repository vorgehalten werden. Was dort steht, gilt.

Die Idee hinter Infrastructure as Code ist es, alle diese Werkzeuge und Prozesse auch auf den Code anzuwenden, der Ihre Infrastruktur beschreibt. Anstelle des Quelltexts bei der Softwareentwicklung handelt es sich um Code, der nicht kompiliert wird, sondern der genutzt wird, um Ihre Infrastruktur zu verwalten. Auch dieser wird mittels einer Versionsverwaltung verwaltet, kann geprüft und kontrolliert werden, kann begutachtet und per Pull Request geändert werden.

Eine wichtige Voraussetzung ist, dass die Beschreibung Ihrer Infrastruktur in maschinenlesbarer Form erfolgt. Eine Checkliste (»Es werden dafür fünf Webserver, eine Datenbank und ein NFS-Server benötigt. Der NFS-Server braucht 100 GB Platz.«), die dann von Mitarbeitenden händisch umgesetzt wird, ist an dieser Stelle der falsche Weg.

Maschinenlesbarer Code bedeutet, dass Programme wie Terraform, AWS CloudFormation oder OpenStack Heat den Code lesen und in fertige Ressourcen umsetzen können. Dies muss reproduzierbar erfolgen können, es sollte also bei gleichem Code das gleiche Endergebnis erzielt werden. Zusätzlich sollte dies *deklarativ* erfolgen, was bedeutet, dass der Zielzustand angegeben wird und nicht die Schritte dahin. So werden bei mehrfachen Aufrufen hintereinander nur einmal Ressourcen aufgebaut und nicht bei jedem Aufruf (siehe dazu auch Abschnitt 3.2.6, »Deklarative Beschreibung des Zustands«).

Terraform bietet an dieser Stelle Vorteile, weil es anders als AWS CloudFormation oder OpenStack Heat nicht auf einen Cloud-Anbieter oder eine Technologie festgelegt ist. Es bietet zwar keine hundertprozentige Abstraktion, jedoch können Sie alle Cloud-Anbieter über das gleiche Werkzeug ansprechen.

2.3 Terraform, seine Geschichte und seine Funktionsweise

Terraform ist ein noch relativ junges, aber dennoch ausgereiftes Programm. Entwickelt wird Terraform von der in Amerika ansässigen Firma HashiCorp. Seit der Gründung 2012 entwickelt sie eine Vielzahl von Cloud-affinen Werkzeugen und Dienstleistungen. Dazu gehören neben Terraform (Erscheinungsjahr 2014) auch weitere, wie z. B.:

- Vagrant (2010) – Bereitstellung und Pflege reproduzierbarer Softwareentwicklungsumgebungen durch Virtualisierungstechnologie.
- Packer (2013) – ein Werkzeug zur Erstellung von Betriebssystemabbildern für virtuelle Maschinen.
- Consul (2014) – Consul bietet ein Dienstnetz, DNS-basierte Dienstsuche (*Service Discovery*) sowie die verteilte Speicherung von Schlüssel-Wert-Paaren (*Key/Value Pairs*).
- Vault (2015) – Vault bietet die Verwaltung von Zugangsdaten und schützungswerten Daten, den identitätsbasierten Zugriff, die Verschlüsselung von Anwendungsdaten und die Prüfung für Anwendungen, Systeme und Benutzer.
- Nomad (2015) – Orchestrationssoftware zur Unterstützung bei der Planung und Verteilung von Aufgaben auf die Arbeitsknoten eines Clusters.
- Sentinel (2017) – ein Sicherheits- bzw. *Policy*-Framework für HashiCorp-Produkte.
- Boundary (2020) – ein Werkzeug für den sicheren Zugriff auf Systeme auf der Grundlage einer vertrauenswürdigen Identität.
- Waypoint (2020) – Waypoint organisiert die plattformübergreifende Erstellung, Bereitstellung und Freigabe von Ressourcen.

Terraform selbst ist ein Werkzeug zum Erstellen, Verwalten und Betreuen von *Ressourcen*. Diese schwammige Definition ist der Knackpunkt beim Arbeiten mit Terraform. Terraform selbst bietet nur eine eingeschränkte Anzahl an Funktionen. So gut wie alle Ressourcen werden von *Providern* bereitgestellt.

Kapitel 4 erläutert Provider, deren Funktionsweise und deren Verwendung eingehend, an dieser Stelle genügt es, zu wissen, dass Provider Terraform ermöglichen, Ressourcen zu verstehen und zu verwalten. Als Vergleich müssen hier ein Schraubendreher und ein Akkuschrauber herhalten. Während ein Schraubendreher genau

eine Art von Schrauben festziehen kann, ist ein Akkuschrauber von sich aus zu gar nichts in der Lage. Setzen Sie jedoch ein Schraub-Bit ein, können Sie Schrauben passend zu diesem Bit festziehen. Je nachdem, ob Sie ein Kreuz-, Schlitz-, Torx- oder Inbus-Bit verwenden, können Sie unterschiedliche Schrauben befestigen. Setzen Sie einen Bohrer ein, können Sie sogar Löcher bohren.

Abhängig davon, bei welchem Cloud-Anbieter Sie Ihre Ressourcen aufbauen wollen, müssen Sie den zugehörigen Provider verwenden. Es gibt Provider für so gut wie alle großen Cloud-Anbieter (AWS, Google Cloud Platform, Microsoft Azure, Digital Ocean, Oracle, Alibaba, Tencent etc.). Terraform unterstützt je nach Provider Ressourcen wie virtuelle Maschinen, Netzwerke, Speicherlösungen, Anwendungen wie z. B. Datenbanken, Sicherheitsregeln und vieles mehr. Sogar eine Pizza lässt sich mit Terraform definieren und bestellen, auch wenn dies nicht direkt eine Cloud-Ressource ist (*https://registry.terraform.io/providers/MNThomson/dominos*).

Neben Cloud-Ressourcen lassen sich mit Terraform auch andere Bereiche abdecken, zum Beispiel die Erzeugung und Bearbeitung von Dateien oder die Ansteuerung von Hypervisoren mithilfe von libvirt. Näheres zu libvirt erfahren Sie in Abschnitt 4.3.5.

Daneben gibt es viele weitere Provider, die lokale Ressourcen wie Dateien verwalten. Es gibt Provider, um Organisationen bei GitHub oder GitLab zu verwalten, um z. B. Benutzer hinzuzufügen oder ein Repository anzulegen. Sie können DNS-Einträge verwalten, Loadbalancer und Firewalls per Terraform verwalten und vieles mehr.

Auch wenn dem Einsatz von Terraform scheinbar keine Grenzen gesetzt sind, so ist Terraform doch ein Werkzeug für eine bestimmte Aufgabe: die Verwaltung von Ressourcen. Eventuell kennen Sie bereits einige andere infrastrukturnahe Programme wie zum Beispiel Ansible oder AWS CloudFormation. Wie genau sich Terraform von anderen Werkzeugen abgrenzt und wann Sie Terraform einsetzen oder nicht einsetzen sollten, besprechen wir kurz in Abschnitt 2.5.

2.4 Wie funktioniert die Cloud?

Nicht technikaffine Personen verstehen unter dem Begriff Cloud im Allgemeinen Dienste wie z. B. Dropbox oder die Apple iCloud. Somit wird die Cloud meist als Datenspeicher bei einem Anbieter verstanden. Das amerikanische *National Institute for Standards and Technology* (NIST) hat eine deutlich detailliertere Definition für den Begriff Cloud ausgearbeitet: *https://csrc.nist.gov/publications/detail/sp/800-145/final*.

Gemäß der NIST-Definition definieren die folgenden fünf Eigenschaften einen Cloud-Service:

1. Die Bereitstellung der Ressourcen wird vom User ohne Interaktion mit dem Dienstleister (Provider) abgewickelt.
2. Die Services werden mit Standardzugriffsmöglichkeiten über das Netz verfügbar gemacht und sind nicht an bestimmte Clientsoftware gebunden.
3. Die verfügbaren Kapazitäten des Providers sind in einem Pool verfügbar, aus dem die Nutzerinnen die Ressourcen erstellen können (Multi-Tenant-Modell). Das bedeutet, die Nutzenden haben kein Wissen darüber, wo genau sich diese Ressourcen befinden. Sie können jedoch den Speicherort vertraglich festlegen oder für unterschiedliche Ressourcen z. B. das Land oder das Rechenzentrum festlegen.
4. Die gebuchten Serviceleistungen können schnell und flexibel bereitgestellt werden, manchmal auch automatisch. Sie erscheinen Ihnen als unendliche Ressource, die genutzt werden kann, ohne dass Sie sich um die Auslastung Gedanken machen müssen – mal abgesehen von den Kosten.
5. Die Nutzung kann gemessen und überwacht werden und dem User zur Verfügung gestellt werden.

Im Kontext des Buchs sehen wir die *Cloud* oder auch *Cloud Computing* als Dienstleistungen jeder Art. Dabei können diese Dienstleistungen auch mehrfach und von unterschiedlichen Personen, Programmen und Unternehmen konsumiert werden.

Die Grundvoraussetzung für Cloud Computing ist, dass der Zugriff mithilfe der serviceorientierten Architektur (SOA) über eine sogenannte *Representational State Transfer*-Schnittstelle (REST-API) möglich ist. Außerdem werden nur die Ressourcen bezahlt, die auch tatsächlich in Anspruch genommen wurden.

Die Anzahl der auf dem Markt verfügbaren Produkte und Anbieter für Cloud-Lösungen hat mittlerweile ein beeindruckendes Ausmaß angenommen. Dementsprechend schwierig ist es, einen Vergleich der Produkte anzustellen, da viele Funktionalitäten in gleicher oder leicht unterschiedlicher Form, aber unter anderem Namen verfügbar sind.

Eines haben alle Anbieter jedoch gemeinsam:

Alle großen Anbieter von Cloud-Lösungen stellen heutzutage APIs zur Verfügung, um darüber die Cloud anzusprechen und darin Infrastruktur aufzubauen. Jedoch liefert jeder Anbieter eigene Werkzeuge aus, um mit seinen APIs zu interagieren. Grund dafür ist, dass sich die APIs in den Funktionen unterscheiden.

Terraform abstrahiert alle diese APIs und macht sie über eine deklarative Konfigurationssprache namens HCL (siehe Kapitel 7) zugänglich. Es löst dabei (bisher) *nicht* das Problem, dass jede Cloud-Infrastruktur anders funktioniert. Terraform-Code für verschiedene Cloud-Lösungen muss weiterhin separat geschrieben werden.

2.5 Ansible, Chef, Salt, Puppet, Terraform – welches Werkzeug für welche Aufgabe?

Im Kanon der zahlreichen Werkzeuge, die Ihnen mittlerweile zur Verfügung stehen, ist es nicht immer leicht, das richtige zu verwenden. Soll eher zu Puppet oder eher zu Terraform gegriffen werden? Was unterscheidet Konfigurationsmanagement von Orchestrierung? Und wo genau ist die Abgrenzung zu *Infrastructure as Code* zu sehen? Diesen Fragen soll im folgenden Abschnitt nachgegangen werden.

Betreiben Sie Ihre Infrastruktur selbst im eigenen Serverraum oder Rechenzentrum, sind mehrere Teams nötig. Sie müssen die Infrastruktur vollständig aufbauen, in Betrieb nehmen und am Laufen halten. Nach der Installation des physikalischen Servers muss vom Netzwerkteam die Verkabelung und die Integration in die vorhandene Netzwerkinfrastruktur vorgenommen werden. Anschließend kann das Betriebssystem von den Administratorinnen bereitgestellt werden, bevor die Entwicklungsabteilung die eigentliche Applikation installiert und konfiguriert.

In der Cloud sind viele dieser Schritte nur noch einen Mausklick entfernt. Neue Netze, Firewall-Regeln (alias *Security Groups*) oder eine weitere virtuelle Maschine erstellen? Alle diese Dinge sind kein Hexenwerk mehr und können per Weboberfläche schnell erledigt werden. Zusätzlich sind viele Schritte eng verzahnt, sodass die Benutzerin beim Anlegen einer virtuellen Maschine bereits das Betriebssystem auswählen und der Maschine die grundlegende Konfiguration schon mitteilen kann. Dies kann z. B. die Konfiguration von Netzwerkadressen, Netzwerkrouten oder DNS-Servern sein. Alle diese Schritte können über Terraform abgedeckt werden.

An der Stelle, an der eine neu erstellte virtuelle Maschine erfolgreich installiert wurde und das Betriebssystem das Anmelden per SSH erlaubt, endet streng genommen der Aufgabenbereich von Terraform. Die Infrastruktur ist an dieser Stelle bereitgestellt und kann genutzt werden. Wie genau die Nutzung aussieht und was dazu noch an Vorarbeiten erledigt werden muss, ist Aufgabe des sogenannten *Konfigurationsmanagements*.

Werkzeuge wie Puppet (*https://puppet.com*), Chef (*https://www.chef.io*), Ansible (*https://www.ansible.com*) oder Salt (*https://docs.saltproject.io/en/latest*) entwickeln mittlerweile zwar auch Funktionen wie Orchestrierung und Bereitstellung von Infrastruktur, ihr Kerngebiet ist jedoch, ein neu erstelltes System in den gewünschten Zustand zu bringen. Dabei ist es egal, ob es sich um eine virtuelle Maschine, einen physikalischen Server oder eine Netzwerk-Appliance (Firewall, Router etc.) handelt.

Sobald die neue Maschine grundlegend erreichbar ist, kann die Konfiguration gemäß dem definierten Endzustand angepasst werden. Programme werden installiert, Konfigurationsdateien angepasst, Dienste gestartet oder gestoppt, NFS-Freigaben eingebunden – und zwar so lange, bis die Maschine dem Soll-Zustand entspricht.

Es ist möglich, aus Terraform heraus alle genannten Werkzeuge zu starten. Mehr dazu erfahren Sie in Abschnitt 7.2. Kapitel 8, »Updates und Day 2 Operations«, beleuchtet die Probleme, die aus der Trennung von Ressourcenverwaltung und Konfigurationsmanagement entstehen können. Dort werden auch Lösungswege aufgezeigt, um später einen reibungslosen Betrieb gewährleisten zu können.

In der Softwareentwicklung ist das Prinzip des Pair Programming schon seit Jahrzehnten ein gängiger Standard, um Fehler zu vermeiden und Wissen weiterzugeben. Auch sind Code Reviews fester Bestandteil der professionellen Entwicklungsarbeit.

Eine gute DevOps-Kultur bedeutet auch, dass Admins von diesen Ansätzen lernen. Das heißt nicht, dass alle Methoden der Entwicklungsarbeit direkt kopiert werden müssen, aber die zusätzliche Kontrolle durch die gemeinsame Arbeit kann Ihnen besonders bei komplizierten und kritischen Deployments so manches graue Haar ersparen.

Einfach umzusetzen ist dies zum Beispiel mit einer Versionsverwaltung. So können z. B. auf GitHub sogenannte *Branch Protection Rules* angelegt werden, die es verbieten, dass Codeänderungen direkt übernommen werden können. Andere Anbieter bieten ähnliche Funktionen unter anderen Namen. Hier können auch noch weitere Schritte (Stichwort CI/CD-Pipelines) unternommen werden, die automatisch den Befehl `terraform plan` ausführen und dessen Ausgabe als Kommentar im Pull Request hinterlassen.

Je nach Bedarf kann dieses Konstrukt noch weiter ausgebaut werden. Sie müssen für sich und Ihr Team ein gutes Mittelmaß zwischen Komfort und Absicherung wählen.

8.3 Mit Terraform arbeiten

Wenn Sie sich entschließen, Terraform zur Pflege Ihrer Infrastruktur einzusetzen, sollte dies umfassend kommuniziert werden.

Es ist nicht nur die Einführung eines neuen Werkzeugs, sondern gleichzeitig die Abschaffung manueller Arbeitsschritte. Das bedeutet häufig auch, dass diese manuellen Arbeitsschritte untersagt werden. Die Zugriffsrechte werden neu sortiert, gegebenenfalls auch entzogen. Gerade der Entzug kann für Unmut in der Kollegschaft sorgen. Nehmen Sie sich also Zeit für die Planungsphase, sodass alle Abteilungen informiert sind und etwas beitragen können. Dazu kommt, dass meist auch bereits existierende Infrastruktur erst in Terraform-Code übersetzt werden muss.

Wenn Sie mit einem neuen Projekt ohne vorher bestehende Strukturen starten (also »auf der grünen Wiese« loslegen), ist das ganze Vorhaben natürlich etwas einfacher.

Zusammenfassend lässt sich sagen: Sobald Sie Terraform nutzen, sollten Sie *ausschließlich* Terraform nutzen.

8.4 Überwachung (Monitoring) der Umgebung

Ein Teil der operativen Aufgaben nach der Inbetriebnahme neuer Infrastruktur ist immer, den Zustand der Umgebung im Blick zu haben. Reicht die Anzahl der Webser-

ver, um den anfallenden Traffic zu bearbeiten? Sind die virtuellen Maschinen ausreichend dimensioniert, oder kommt es zu Engpässen in Bezug auf CPU-Auslastung, RAM etc.?

Bei der eigentlichen Überwachung der Umgebung kann Terraform nicht helfen. Hier sind andere, speziell darauf ausgelegte Lösungen wie Icinga2 oder Prometheus nötig, um die zwei bekanntesten Vertreter alten und neuen Monitorings zu erwähnen. Wohl aber kann Terraform bei den auszuführenden Aktionen helfen, die aufgrund des Monitorings anstehen.

8.4.1 Horizontale Skalierung bei Lastspitzen

Stellen Sie sich ein Szenario vor, in dem mehrere Webserver hinter einem Loadbalancer stehen, der die Anfragen auf die Webserver verteilt. Technische Feinheiten wie die Persistenz von Benutzersitzungen bleiben in der folgenden Betrachtung außen vor, das Grundprinzip bleibt aber auch in derartigen Fällen gleich.

Das Monitoring zeigt, dass die Last der Webserver auf einem hohen Niveau liegt, aber bisher noch nicht zu Beeinträchtigungen der User führt. Bevor es so weit kommt, ist das Ops-Team gefragt und muss eine Lösung für das Problem suchen.

Eine Lösung ist, die Anzahl der Webserver zu erhöhen. Hierdurch werden die eingehenden Anfragen auf mehrere Schultern verteilt, sodass die einzelnen Webserver weniger Arbeit haben und die Last hoffentlich auf ein akzeptables Niveau sinkt. Dieses Vorgehen, bei dem die Anzahl an Instanzen erhöht wird, die Spezifikation der Instanzen jedoch gleich bleibt, wird gemeinhin als *horizontale Skalierung* bezeichnet. Ein Beispiel für die *horizontale Skalierung* zeigt Abbildung 8.1.

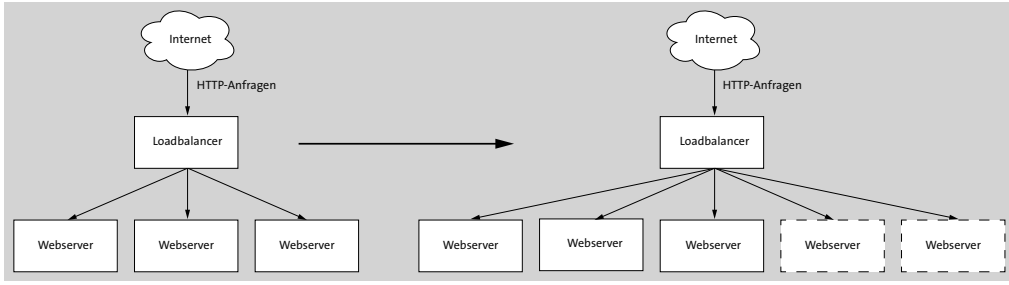

Abbildung 8.1 Horizontale Skalierung durch Erzeugung neuer Ressourcen

Wurde die Infrastruktur mit Terraform aufgebaut, kann Terraform auch zur Lösung dieses Problems verwendet werden. Im besten Fall wird der Wert einer Variablen in Terraform erhöht, die die Anzahl der Webserver enthält. Nach einem Aufruf von `terraform apply` stehen weitere virtuelle Maschinen zur Nutzung als Webserver zur Verfügung. Die Konfiguration der Webserver, d. h die Installation und Konfiguration von

Apache, Nginx oder Ähnlichen und des Loadbalancers davor, kann wie bereits beschrieben auf viele Arten erfolgen. Im gängigsten Fall wird einfach das verwendete Konfigurationsmanagement durch das Monitoring gestartet. Je nach Umgebung kann es auch möglich sein, die Konfiguration des Loadbalancers direkt per Terraform anpassen zu lassen. Die IP-Adressen der Webserver-VMs kennt Terraform ja.

Sobald der Loadbalancer die Webserver als funktional erkennt (Stichwort *Healthchecks*), nimmt er sie in Betrieb und verteilt Anfragen auf die neuen Webserver.

Das Ops-Team kann die Situation anschließend beobachten und anhand des Monitorings entscheiden, ob die Last in Ordnung oder weiterhin zu hoch ist. In diesem Fall wird die eben geschilderte Prozedur nochmals durchgeführt, die Variable angepasst, neue Webserver-VMs werden erstellt und konfiguriert, und anschließend wird abermals abgewartet.

Sinkt die Arbeitslast, weil sie z. B. saisonbedingt schwankt (Weihnachtsgeschäft, Schulbeginn etc.), kann Terraform auch hier die nicht mehr benötigten Ressourcen aufräumen. Hier muss die Loadbalancer-Konfiguration ebenfalls angepasst werden, sodass nicht mehr existierende Webserver keine Anfragen mehr erhalten.

Wichtig in diesem Beispiel ist, dass Terraform hier vollkommen losgelöst vom Monitoring verwendet wird, d. h., es existiert standardmäßig keine Kopplung zwischen Monitoring und Terraform. Je nach Umgebung kann eine solche Kopplung gewünscht sein, um manuelle Interaktion zu vermeiden oder zu vermindern. In anderen Umgebungen ist es gewünscht, dass die letzte Entscheidungsinstanz immer ein Mensch sein soll, der die Aktion genehmigt.

Zu erwähnen sei hier, dass viele Cloud-Anbieter spezielle Funktionalitäten bieten, um ohne Terraform direkt innerhalb der Cloud auf Lastspitzen zu reagieren und zusätzliche Ressourcen bereitzustellen. Inwiefern derartige Funktionen bei Ihrem Cloud-Anbieter verfügbar, für Ihren Anwendungsfall technisch geeignet und überdies wirtschaftlich sinnvoll sind, muss im Einzelfall entschieden werden.

8.4.2 Vertikale Skalierung von virtuellen Maschinen

Ein anderes Fehlerbild ist die falsche Dimensionierung von virtuellen Maschinen. Typischerweise fehlen der VM virtuelle CPUs oder RAM, die in der VM laufende Applikation verweigert daher ihren Dienst. Auch der starke Anstieg der Nutzung des Swap-Speichers bei Speichermangel fällt in diese Kategorie und führt zu Beeinträchtigungen bis hin zum Ausfall des Diensts.

Derartige Fehler müssen über das Monitoring der Umgebung bemerkt werden – im besten Fall, bevor auch nur ein einziger Kunde davon beeinträchtigt wird. Aber wie jeder erfahrene Mitarbeiter schmerzhaft lernen musste, gibt es Fehler, die trotz inten-

siven Testens erst im produktiven Einsatz auftreten. Wie heißt das Sprichwort? »Kein Plan übersteht den ersten Feindkontakt.«

Wie im vorherigen Abschnitt beschrieben, kann Terraform beim eigentlichen Monitoring, der Suche nach dem zugrunde liegenden Fehler und einer Lösung hierfür, nicht helfen. Es kann nur ein Werkzeug sein, das bei der Beseitigung des Problems hilft.

Zeigt sich, dass die Dimensionierung der VM der Grund für das Fehlverhalten ist, kann Terraform nach Anpassung der entsprechenden Angaben im Terraform-Code eine neue VM ausreichender Größe erstellen. Dies nennt sich *vertikale Skalierung* (siehe Abbildung 8.2).

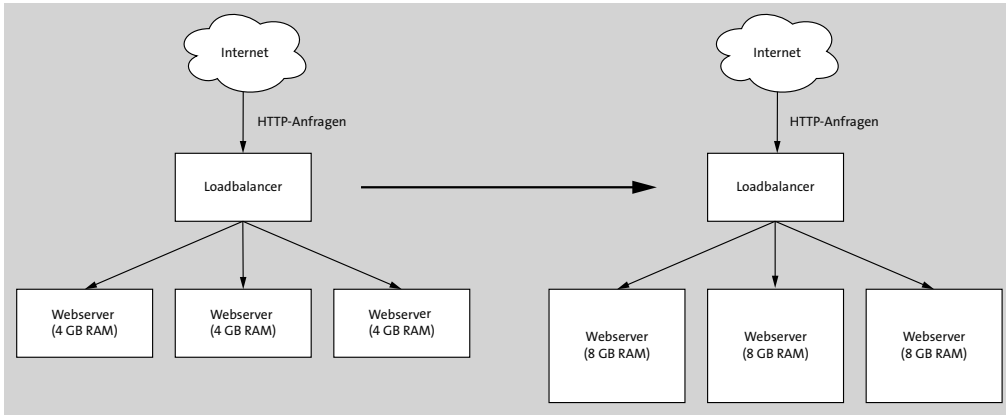

Abbildung 8.2 Vertikale Skalierung von Webservern durch Vergrößerung des Arbeitsspeichers

Der Teufel steckt hierbei jedoch im Detail.

In der Standardeinstellung führt die Anpassung der Spezifikationen einer virtuellen Maschine dazu, dass Terraform korrekt erkennt, dass der tatsächliche Zustand der Umgebung nicht dem deklarativ beschriebenen Wunschzustand entspricht. Terraform führt daher die nötigen Schritte aus, um den Wunschzustand herzustellen. Je nach verwendetem Provider führt dies dazu, dass die bestehende VM-Ressource vergrößert oder aber abgerissen wird, um anschließend eine neue VM-Ressource mit den angepassten Spezifikationen zu erstellen. Dies kann zu einem Ausfall des Diensts führen, während die alte VM entfernt, die neue VM bereitgestellt, hochgefahren und gegebenenfalls durch ein Konfigurationsmanagementsystem eingerichtet wird.

Wie in Abschnitt 7.5, »Deployments versionieren«, bereits angerissen wurde und später in diesem Kapitel detaillierter beschrieben wird, kann das Verhalten von Terraform durch die sogenannten `lifecycle`-Parameter bedingt angepasst werden. So kann mittels des `create_before_destroy`-Parameters die neue VM erstellt werden, be-

vor die alte VM entfernt wird. Dies führt jedoch zum einen zu größerem Ressourcenverbrauch während des Parallelbetriebs beider virtueller Maschinen, und zum anderen wartet Terraform nur, bis die neue Ressource aus dessen Sicht verfügbar ist, bevor es die alte Ressource entfernt. Dass nach dem Erstellen der virtuellen Maschine noch das Konfigurationsmanagement die Konfiguration anpasst und gegebenenfalls ein Backup von Daten zurückgespielt werden muss, ist Terraform standardmäßig nicht bekannt und wird daher nicht berücksichtigt. Sprich, auch in diesem Fall kann es zu einer Unterbrechung des Diensts kommen, sofern die Anwenderin nicht Vorkehrungen z. B. durch Provisioner mit Healthchecks trifft.

In den meisten Fällen ist es sinnvoller, schon bei der Planung den Fokus auf die horizontale Skalierung zu setzen. Das heißt nicht, dass sich vertikale Skalierung immer vermeiden lässt, vielmehr soll stets darauf geachtet werden, dass eine horizontale Skalierung ermöglicht wird.

8.4.3 Wenn horizontale Skalierung nicht mehr ausreicht

Es sei nur ganz kurz angerissen, dass horizontale Skalierung nicht das Allheilmittel ist. Auch hier können Sie an technische oder gar physikalische Grenzen stoßen.

Wenn ein Dienst eine absurd hohe Anzahl an Anfragen handhaben muss, kann es sich lohnen, ihn auf mehrere Regionen zu verteilen. Der Terraform-Code hierfür bleibt in den meisten Fällen nahezu identisch, muss jedoch auf unterschiedliche (geografische) Regionen angewandt werden. Um dennoch ein Zusammenspiel der Regionen zu erreichen, helfen Techniken wie DNS Loadbalancing und Tools wie Apache Kafka, CockroachDB und einige weitere Geo-Replikationswerkzeuge von den Cloud-Anbietern selbst.

8.5 Updates und Änderungen einspielen

Ein wichtiger Bestandteil der Day 2 Operations ist der Betrieb und die Instandhaltung der bestehenden Infrastruktur. Dies betrifft gegebenenfalls auch die physikalische Hardware, sofern Sie ein eigenes Rechenzentrum oder eigene Hardware in *Colocation* betreiben.

Bei Hardware kann Ihnen Terraform nur bedingt Unterstützung anbieten.

Das Aktualisieren (*Patching*) ist wichtiger Bestandteil der Day 2 Operations. Dies umfasst sowohl das Bereitstellen und Einspielen von (Sicherheits-)Updates des Betriebssystems als auch das Einspielen von Updates für die von Ihnen betriebenen Anwendungen (Webserver, Datenbanken, eigene Anwendungen etc.). Auch was Patch-Prozesse in bestehenden Maschinen angeht, ist Terraform jedoch das falsche Werkzeug.

Sofern Sie Terraform als Werkzeug frühzeitig in Ihre Planung einfließen lassen, können Sie jedoch Konzepte wie *Immutable Infrastructure* umsetzen. Und dabei kann Ihnen Terraform eine große Hilfe sein.

8.5.1 Immutable Infrastructure

Hinter dem Begriff *Immutable Infrastructure* verbirgt sich ein aus dem Cloud-native-Umfeld stammendes Konzept, bei dem eine einmal aufgebaute Ressource nicht mehr verändert wird. Stattdessen wird eine neue Ressource aufgebaut, die dann die alte Ressource ersetzt. In der Dokumentation finden Sie dazu weitere Infos: *https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure*.

Um dieses Konzept zu erläutern und die Vor- und Nachteile herauszuarbeiten, wird wieder auf das Beispiel mit einem Loadbalancer sowie mehreren Webservern zurückgegriffen: Der Loadbalancer ist den Webservern vorgeschaltet und verteilt die eingehenden Anfragen an die Webserver. Dazu prüft er, ob die Webserver erreichbar sind.

Bisherige Vorgehensweise

Die bisher gängige Vorgehensweise sieht ungefähr wie folgt aus:

Ein Server wird aus der Loadbalancer-Konfiguration genommen, um ihn per `zypper`, `yum`, `dnf`, `apt` oder per Windows-Update auf den aktuellen Stand zu bringen und die Sicherheitsupdates einzuspielen. Nach einem Neustart kann der Server wieder in die Loadbalancer-Konfiguration eingetragen werden, woraufhin sich das Spiel mit dem nächsten Server wiederholt. Die genannten Schritte werden so lange wiederholt, bis alle Webserver aktualisiert wurden.

Dieser Prozess kann zwar automatisiert werden, hat jedoch einige Unzulänglichkeiten. Was passiert beispielsweise, wenn nach dem Ausrollen der Updates auf einigen Servern ein Fehler auftritt, der scheinbar durch die Updates verursacht wurde, jedoch in den vorher stattfindenden Tests nicht aufgefallen ist? Was passiert, wenn der Server nach einem Update nicht mehr startet? Genauso kann es passieren, dass erst im Monitoring auffällt, dass sich die Perfomancemetriken seit dem Update signifikant verschlechtert haben.

In diesem Fall muss jeder einzelne Server durch Deinstallation oder Zurückrollen der Updates auf den alten und funktionierenden Zustand zurückgebracht werden. Dies ist mit viel manuellem Einsatz verbunden und skaliert daher sehr schlecht, wenn es um eine Vielzahl an Maschinen geht. Auch kann es durch Updates zu nicht abwärtskompatiblen Änderungen gekommen sein, die sich zwischen den Versionen unterscheiden. Das könnte z. B. durch Änderungen an den Datenbankschemata geschehen.

Vorgehensweise bei Immutable Infrastructure

Immutable Infrastructure versucht, diese Probleme zu umgehen, indem von Anfang an anders an das Thema Updates herangegangen wird.

Anstatt die bestehende Ressource anzufassen und zu ändern, wird eine weitere, neue Ressource aufgebaut und konfiguriert. Dann kann die neue Ressource die tatsächlichen Anfragen bearbeiten, und die alte Ressource kann nach einer gewissen Karenzzeit entfernt werden. Über die Konfiguration des Loadbalancers lässt sich jederzeit steuern, welche Webserver Anfragen erhalten.

Zu beachten ist, dass das Erstellen und Konfigurieren der neuen Ressource auch das Bereitstellen etwaig benötigter Daten umfasst. Dies sollte über das Konfigurationsmanagement automatisiert abgebildet werden.

Um auf das Webserverbeispiel zurückzukommen:

Anstatt die Webserver einzeln anzufassen und zu aktualisieren, werden *neue* Webserver-VMs mit einem aktuellen Betriebssystemabbild installiert. Dadurch sind in diesen Maschinen alle Sicherheitsupdates bereits enthalten. Dann können diese neuen Webserver in die Loadbalancer-Konfiguration eingetragen werden und erhalten Anfragen. Treten Probleme auf, werden die neuen Webserver wieder aus der Konfiguration geworfen, sodass nur die alten Webserver Anfragen beantworten.

Erst wenn sichergestellt ist, dass die neuen Webserver einwandfrei funktionieren, werden die alten Webserver entfernt.

So weit die grundsätzlichen Erläuterungen zu Immutable Infrastructure. Für die genaue Vorgehensweise beim Ersetzen sei auf die noch folgenden Ausführungen zum Thema Blue-Green-Deployments, Canary-Deployments & Co. im Verlauf des Kapitels verwiesen.

Im Zusammenhang mit Immutable Infrastructure sei noch eine Anmerkung zur ebenfalls aus der Cloud-native-Welt stammenden Redewendung »Pets vs. Cattle«, zu Deutsch »Haustiere im Gegensatz zur Viehherde«, erlaubt. Auch wenn sie aus ethischer Sicht ein fragwürdiges Bild entwirft, besteht der Hauptunterschied darin, dass Sie Ihre Server nicht als Haustiere betrachten sollten. Anstatt sie zu hegen und zu pflegen, sollen sie als austauschbar und leicht zu ersetzen betrachtet werden. In die gleiche Richtung geht das Bilder der »Snowflake«-Server, die alle manuell erstellt und konfiguriert werden. Dies führt leider am Ende dazu, dass jeder Server so einzigartig ist wie eine Schneeflocke. Stattdessen ist es besser, die Serverkonfiguration komplett zu automatisieren, sodass das Ersetzen eines Servers ohne Handarbeit ablaufen kann.

Die gesamte Infrastruktur ist das Hauptaugenmerk. Anstatt händisch erstellte Server mühsam wieder instand zu setzen und zu hoffen, dass alle gleich konfiguriert wurden, werden neue (und hoffentlich) funktionierende Server als Ersatz erstellt. Die alten Server werden entfernt. Diese Arbeitsweise funktioniert nur bei massivem Ein-

satz von Automatisierung und Infrastructure as Code, führt aber aufgrund der bereits dargelegten Vorteile zu einem einfacheren und weniger aufwendigen Betrieb der Infrastruktur.

8.5.2 Koordiniertes Ausrollen neuer Versionen

Der vorangegangene Abschnitt hat die grundsätzliche Vorgehensweise beim Update mittels Immutable Infrastructure dargelegt, jedoch zu Vereinfachungszwecken einige nicht unwichtige Details ausgelassen. Wichtigster Punkt ist die genaue Abfolge der Aktionen, wenn neue Webserver erstellt und alte Webserver entfernt werden. Hier haben sich in der Praxis mehrere Philosophien und Konzepte entwickelt. Diese sollen im Folgenden genauer erläutert werden.

Der Vorschlaghammer

In der Standardeinstellung wird Terraform bei Unterschieden in der Definition einer Ressource zwischen Ist- und Soll-Zustand versuchen, diese so zu ändern, dass der Soll-Zustand erreicht wird. Das setzt jedoch voraus, dass die gewünschte Änderung vom Provider ohne Entfernen und Neuaufbauen der Ressource möglich ist.

Ein gängiges Beispiel ist die bei Erstellung einer virtuellen Maschine verwendbare `cloud-init`-Konfiguration. `cloud-init` ist ein Mechanismus, der es erlaubt, in beschränktem Maß Konfigurationsanweisungen von außen in eine VM hineinzureichen. So können grundlegende Dienste wie DNS und Netzwerk konfiguriert und z. B. ein weiteres Konfigurationsmanagement wie Chef, Puppet, Ansible oder Salt kann gestartet werden. Terraform bietet je nach Provider die Möglichkeit, diese Konfigurationsdatei vorzubereiten und bei der Kommunikation mit der API des Cloud-Anbieters mitzuschicken.

Was jedoch gern übersehen wird: Ändert die Administratorin die Definition dieser `cloud-init`-Konfigurationsdatei, und sei es auch nur aufgrund eines Tippfehlers in einem Kommentar, betrachtet Terraform dies beim nächsten Aufruf von `terraform apply` als Änderung an der Definition der virtuellen Maschine. Terraform wird versuchen, den gewünschten Soll-Zustand herzustellen. Leider heißt dies, dass die virtuelle Maschine neu installiert werden muss, um die geänderte `cloud-init`-Konfiguration anzuwenden.

Ein anderes gängiges Beispiel ist die Änderung des Betriebssystemabbilds, die zu einer Neuinstallation der virtuellen Maschine führt. Abschnitt 8.5.4, »Ausrollen neuer Betriebssystemabbilder«, geht auf dieses Szenario ein.

Langer Rede, kurzer Sinn: Wenn die Anwenderin ihr Augenmerk nicht auf die Update-Strategie legt, wird ein Update der Konfiguration zu einer Unterbrechung des Diensts, also zu Downtime führen. Dies kann je nach Einsatzszenario akzeptabel sein,

z. B. wenn der Befehl `terraform apply` automatisiert nur in wöchentlichen Wartungszeiträumen ausgeführt wird. In den meisten Fällen wird die Infrastruktur aber kontinuierlich in weitaus kürzeren Intervallen konfiguriert. Hier ist eine solche Unterbrechung des Diensts nicht wünschenswert.

Wichtig ist, dass Sie die Risiken kennen und entscheiden können, ob eine Unterbrechung des Diensts für Sie akzeptabel ist oder nicht.

Das Ausführen des Befehls `terraform plan` zeigt Ihnen jedoch an, ob eine Ersetzung und damit auch eine Unterbrechung auftreten wird.

Zero-Downtime-Deployments

Ein weiterer Trumpf im Buzzword-Bingo ist das sogenannte *Zero-Downtime-Deployment*, das HashiCorp selbst in einem Tutorial anhand des DigitalOcean-Providers beschreibt. Die Grundlagen sollten jedoch auch für andere Provider gelten: *https://www.hashicorp.com/blog/zero-downtime-updates-with-terraform*.

Sie wissen, dass Terraform Ressourcen löschen und neu erstellen wird, wenn eine Änderung der Ressource nicht möglich ist. Wie in Abschnitt 7.5 beschrieben, kann die Reihenfolge von Entfernen und Neuerstellen durch Verwendung des `create_before_destroy`-Metaarguments umgedreht werden. Dadurch würde zuerst die neue Ressource erstellt und für einen gewissen Zeitraum parallel mit der alten Ressource existieren, sofern dies vom verwendeten Provider unterstützt wird. Bei einer solchen Vorgehensweise ist es wichtig, dass während des Deployment-Prozesses eine Prüfung stattfindet, ob die erstellte Resource bzw. der Dienst auch wie erwartet verfügbar ist. Ist das der Fall, kann sie genutzt werden, und die alte Ressource kann entfernt werden. Würde ein solcher Kontrollmechanismus fehlen, würde im schlimmsten Fall das Ersetzen der Ressource eine Unterbrechung des Diensts verursachen.

Die ideale Vorgehensweise beim Zero-Downtime-Deployment ist es, eine Prüfung des Diensts einzurichten, z. B. über einen `local-exec`-Provisioner (siehe Abschnitt 7.2) in der Definition der Ressource. Hierdurch wird während des Deployments sichergestellt, dass die neue Ressource erstellt wurde *und* verfügbar ist, bevor die alte Ressource entfernt wird.

Dazu bedarf es eines kleinen Exkurses in die Interna von Terraform: Je nach Provider wird jeder Auftrag, den Terraform an die API des Providers schickt, asynchron abgearbeitet. Das bedeutet, dass Terraform periodisch nachfragt, ob der Auftrag bereits vollständig ausgeführt wurde. Im Normalfall wartet Terraform hier jedoch nur die Rückmeldung des Providers ab, um die neue Ressource als fertig zu deklarieren. Bei virtuellen Maschinen kann dies bedeuten, dass die Maschine noch vom Betriebssystemabbild bootet, das Konfigurationsmanagement (falls vorhanden) noch Anpassungen vornimmt und somit die Einrichtung der VM aus Sicht der Anwenderin noch

nicht abgeschlossen ist. Für Terraform ist die Ressource jedoch bereits fertig, daher beginnt Terraform damit, die alte Ressource zu entfernen.

Woher sollte Terraform auch wissen, was genau die Anwenderin unter fertig versteht?

Die Nutzung eines `local-exec`-Provisioner ist eine Möglichkeit, Terraform genau diese Information zur Verfügung zu stellen und zu prüfen. Das genannte Tutorial enthält den folgenden Codeblock:

```
provisioner "local-exec" {
  command = "./check_health.sh ${self.ipv4_address}"
}
```

Hierdurch würde Terraform das Skript */check_health.sh* aufrufen, das die IP-Adresse der virtuellen Maschine als Argument enthält. Dieses Skript sollte periodisch den Zustand der VM testen und so lange laufen, wie die Rückmeldung nicht zufriedenstellend ist. Am Beispiel eines Webservers würden Sie z. B. per `curl` oder `wget` versuchen, eine HTTP-Verbindung zu öffnen. Kommt die Verbindung zustande, wird anschließend die Ausgabe gegen die Erwartungen validiert. Dies kann über die Kontrolle des Inhalts geschehen (z. B. soll die Webseite die Phrase `Willkommen bei terraformbuch.de` enthalten) oder über den Statuscode (erhalten Sie einen HTTP-Statuscode 200 zurück?) erfolgen.

Erst wenn die Kontrolle erfolgreich ist, beendet sich dieses Skript erfolgreich. Und erst dann betrachtet Terraform die virtuelle Maschine als fertig und würde die alte VM-Ressource entfernen.

Classic-, Blue-Green-, Canary-Deployment und Rolling Releases

Abbildung 8.3 zeigt ein Deployment mit einer Unterbrechung des Diensts, wie es beim Ausführen von `terraform apply` nach z. B. der Änderung des Betriebssystemabbilds passiert. Alle Server werden gleichzeitig aktualisiert, in dieser Zeit steht der Dienst nicht zur Verfügung.

Um dieses Szenario zu vermeiden, haben Sie mehrere Möglichkeiten, die im Folgenden erläutert werden.

Begrifflichkeiten

Die Terraform-Dokumentation bietet ein Tutorial zu *Blue-Green-Deployments*, *Canary-Deployments* und *Rolling Releases*: *https://learn.hashicorp.com/tutorials/terraform/blue-green-canary-tests-deployments*.

Beachten Sie aber, dass die Begrifflichkeiten gerne beliebiger verwendet werden und die eigentliche Definition unscharf ist.

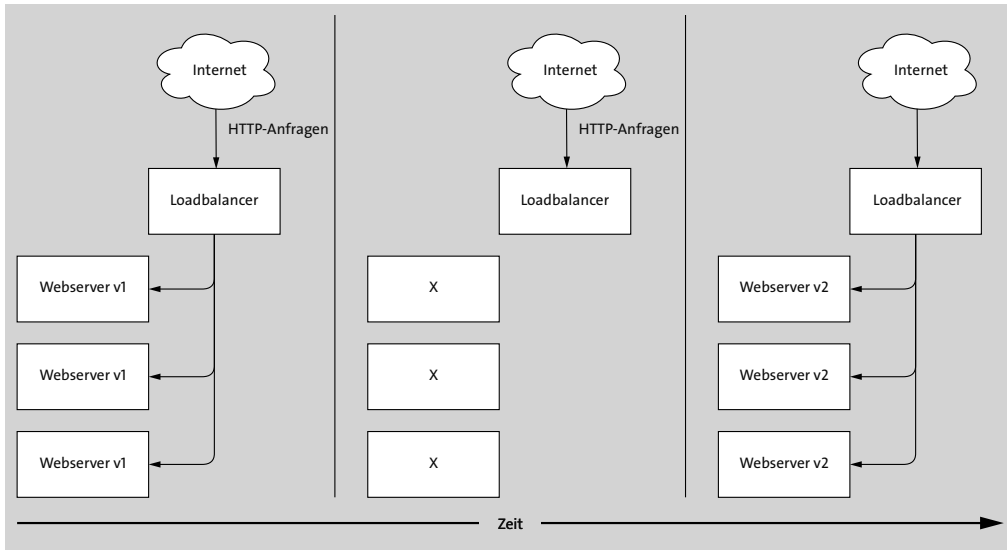

Abbildung 8.3 Unterbrechung des Diensts beim Ändern z. B. des Betriebssystemabbilds

Das Blue-Green-Deployment

Zumeist wird unter einem *Blue-Green-Deployment* der Prozess verstanden, bei dem neben einer bisherigen Infrastruktur (*Blue Deployment*) parallel eine neue Version (*Green Deployment*) aufgebaut wird. Im Beispiel mit einer Gruppe von Webservern ist ein vor den Servern konfigurierter Loadbalancer der Verteiler für die Webserveranfragen. Er leitet die Anfragen auf das eine oder andere Deployment um.

Beim eigentlichen Blue-Green-Deployment wird einfach der komplette Netzwerkverkehr (100 %) auf die neue Version umgeleitet. Sofern keine Fehler auftreten, kann die alte Version, gegebenenfalls nach einer Karenzzeit, abgebaut werden. Damit existiert nur noch die neue Version.

Die grüne Version wird beim nächsten Update dann als alte Version gehandhabt, und eine neue Version (*Blue Deployment*) wird parallel dazu aufgebaut. Dieses Spiel wird im Wechsel bei jeder Änderung durchgeführt.

Der Mechanismus des Blue-Green-Deployments ist fast transparent für eine Endanwenderin wie z. B. den Besucher einer Webseite. Er bekommt gar nicht mit, dass die Infrastruktur einen Versionswechsel erfahren hat.

Abbildung 8.4 zeigt das Schema eines Blue-Green-Deployments.

Eine alternative Umsetzung des Blue-Green-Deployments ist es, die alte Version nicht zu entfernen. Stattdessen wird diese für die Weiterentwicklung genutzt. Diese Alternative ist im Kontext der Immutable Infrastructure jedoch weniger gängig.

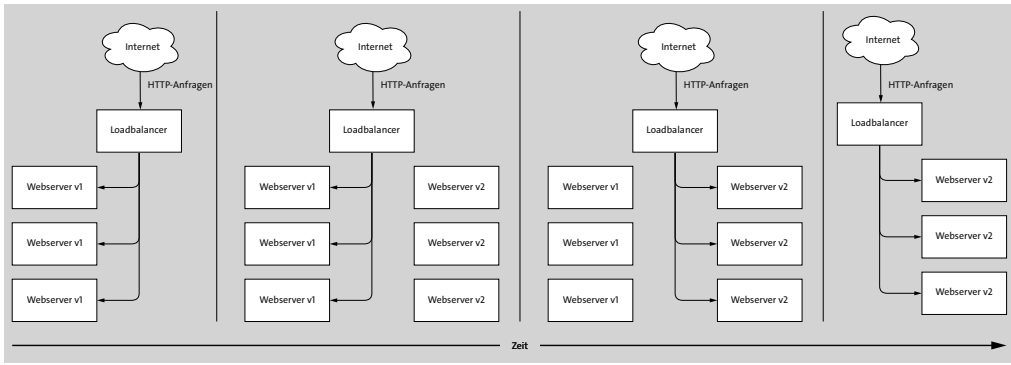

Abbildung 8.4 Schematische Darstellung eines Blue-Green-Deployments

Grue und bleen

Auch hier gibt es Unschärfen, was den zweiten Durchgang angeht. Seien Sie also nicht verwundert, wenn manche das grüne Deployment als »blau« deklarieren, sobald es stabil ist, während andere beim zweiten Durchgang von einem Green-Blue-Deployment sprechen.

Das Canary-Deployment

Ein *Canary-Deployment* baut auf dem soeben beschriebenen reinen Blue-Green-Deployment auf, unterscheidet sich jedoch von diesem. Der Unterschied ist, dass nicht auf einmal von alt (blau) nach neu (grün) umgeschaltet wird.

Der Ursprung des Begriffs kommt aus dem Bergbau, wo Kanarienvögel eingesetzt wurden, um frühzeitig auf austretendes Grubengas aufmerksam gemacht zu werden. Die Kanarienvögel starben zuerst an Grubengasvergiftung, bevor die Arbeiter gefährdet waren. Durch aufmerksame Beobachtung der Kanarienvögel konnte der Schutz der Arbeiter gewährleistet oder es konnte zumindest das Risiko minimiert werden.

Auf diesen Anwendungsfall bezogen, würden einige User als Kanarienvögel dienen und die neue Version testen. Das heißt auch, dass ein gewisser Bruchteil der Anwendung defekt sein kann. Treten Probleme auf (und werden diese auch bemerkt), können alle Anfragen wieder an die alte Version umgeleitet werden, sodass der Weiterbetrieb der Anwendung nicht gefährdet ist. Wie genau die Umsetzung dieses Mechanismus realisiert wird, ist stark vom Anwendungsfall abhängig. Beispielhaft kann es so aussehen:

1. 100 % der Zugriffe auf das alte Deployment
2. Beginn des *Canary-Deployments*: 10 % der Zugriffe auf das neue Deployment, 90 % weiterhin auf das alte Deployment

3. 50 % der Zugriffe auf jedes Deployment
4. 90 % der Zugriffe auf das neue Deployment, nur noch 10 % auf das alte Deployment
5. 100 % der Zugriffe auf das neue Deployment
6. Abbau des alten Deployments

Selbstverständlich wird jeder Schritt aufmerksam beobachtet, und nach jedem Schritt wird ein Zeitlang getestet, ob es zu Problemen kommt. Erst wenn keine Auffälligkeiten bemerkt wurden, wird der nächste Schritt angegangen.

Auch in diesem Fall ist eine erhöhte Ressourcennutzung notwendig, da große Teile der Infrastruktur doppelt vorhanden sind. Dies kann zu hohen Mehrkosten führen, was mit den Kosten beim Ausfall des Diensts in Relation gesetzt werden muss. Der Umsatzausfall eines großen Webshops kann schnell astronomische Höhen erreichen, sodass die Aufwendungen für derartige Deployments wirtschaftlich die bessere Alternative sind. Wie immer ist hier keine generelle Aussage möglich, und Sie müssen für Ihren Anwendungsfall entscheiden, wie Sie verfahren wollen oder können.

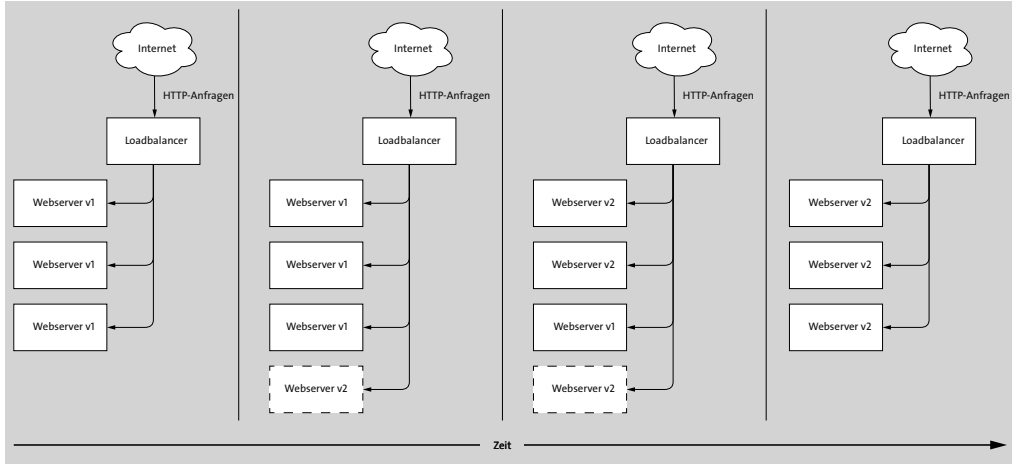

Abbildung 8.5 Schematische Darstellung eines Canary-Deployments

Das Rolling Release bzw. Rolling Deployment

Das *Rolling Release* oder *Rolling Deployment* ist (je nach Definition) ein Spezialfall des Canary-Deployments.

Beim Canary-Deployment werden alle Server, sowohl die alten als auch die neuen, parallel betrieben, bis aller Netzwerkverkehr auf die neue Version umgeleitet ist. Hingegen werden beim Rolling Release existierende Server direkt durch neue Server ersetzt, also die Gesamtanzahl nicht geändert. Je nach Gesamtanzahl der Server sind auch andere Schritte denkbar. Bei Hunderten von Servern kann mit einer Schritt-

weite von einem Server gestartet und die Schrittweite kann auf fünf und später zehn Server erhöht werden.

Vorteil ist, dass nicht die doppelte Anzahl an Servern für den gesamten Update-Zeitraum in Betrieb ist, sondern nur ein oder fünf oder zehn zusätzliche Server benötigt werden, je nach Schrittweite.

Nachteilig ist, dass ältere Server abgerissen werden, sobald der neue Server in Betrieb genommen und getestet wurde. Hierdurch wird das Zurückkehren auf einen älteren Stand beim Auftreten von Fehlern spät im Update-Prozess unmöglich.

8.5.3 Umsetzung der Release-Strategien mit Terraform

Unabhängig davon, für welche der genannten Strategien Sie sich entscheiden: Terraform kann Sie dabei unterstützen, falls der verwendete Provider eine Funktionalität dafür bereithält. Terraform selbst bietet keine derartige Funktionalität an.

Viele der verfügbaren Terraform-Deployments verwenden beispielsweise eine AWS Auto Scaling Group, die logischerweise nur bei AWS unterstützt wird. Andere nutzen die von den Providern bereitgestellten Loadbalancer und konfigurieren diese per Terraform. Hierdurch können Sie die Verteilung des Netzwerkverkehrs per Terraform steuern.

Die offizielle Dokumentation zu Terraform setzt das *Blue-Green-Deployment* mit zwei unterschiedlichen Versionen einer Ressource in Form eines Webservers um: *https://learn.hashicorp.com/tutorials/terraform/blue-green-canary-tests-deployments*.

Hierbei erstellt die Anwenderin neuen Code parallel zum alten, um das grüne Deployment zu erstellen. Anschließend wird die Update-Strategie in Form einer Loadbalancer-Konfiguration definiert und dem Loadbalancer übergeben. Durch Anpassung einer einzigen Zeile Code kann die Verteilung von »100 % blau« zu »10 % grün und 90 % blau« etc. geändert werden.

Wenngleich funktional, ist dies keine schöne Lösung, da immer noch sehr viel Handarbeit gefragt ist. Tatsächlich ist hier Ihre Kreativität gefragt, da vieles zu stark vom einzelnen Anwendungsfall abhängig ist. Terraform kann unterstützen, allerdings auch nicht immer.

8.5.4 Ausrollen neuer Betriebssystemabbilder

Alle Linux-Distributionen, sowohl aus dem Enterprise-Umfeld als auch kostenlos verfügbare, bieten periodisch aktualisierte Installationsmedien an, um bei der Installation direkt einen aktuelleren Stand an Paketen zu installieren. Dies erspart das aufwendige Nachinstallieren aller verfügbaren Updates direkt nach der Installation und kann so die Installationszeiten verkürzen. Bei Microsoft-Windows-Abbildern ist das leider nicht immer der Fall.

Unabhängig davon bietet es sich an, die neuesten Installationsmedien auch für alle neu erstellten virtuellen Maschinen in Ihrer Infrastruktur zu verwenden. Leider jedoch gehört das Betriebssystemabbild zu den Eigenschaften einer virtuellen Maschine, die sich nicht ohne das Neuerstellen der Ressource ändern lassen.

```
resource "openstack_compute_instance_v2" "webserver" {
  name        = "webserver"
  flavor_name = "4C-4GB-40GB"
  image_name  = "openSUSE Leap 15.3"
[…]
}
```

Im Beispiel verwendet Terraform zum Erstellen einer virtuellen Maschine das Image mit dem Namen `openSUSE Leap 15.3`. Ändern Sie diese Angabe auf eine andere Version wie `openSUSE Leap 15.3.1`, führt das zum Neuaufbau der Maschine.

Je nach verwendetem Cloud-Anbieter gibt es zwei Möglichkeiten, auf Betriebssystemabbilder zuzugreifen. Sie können entweder die eindeutige ID verwenden oder wie im Beispiel über einen Namen darauf zugreifen. Dies bietet Ihnen gegebenenfalls die Möglichkeit, den Namen des alten Abbilds zu ändern und anschließend das aktualisierte Betriebssystemabbild unter dem alten und bisher verwendeten Namen bereitzustellen.

Auf diese Weise sieht Terraform nicht, dass sich hinter dem gleichen Namen ein anderes Abbild verbirgt, und würde bestehende Maschinen nicht entfernen und neu aufbauen. Durch Nutzung von `terraform apply -replace` (siehe Kapitel 12) können Sie einzelne Maschinen dann mit der neueren Version neu bereitstellen lassen, sofern dies gewünscht ist.

8.6 Lifecycle-Management mit Terraform

Wie bereits in den vorangegangenen Abschnitten gezeigt, versucht Terraform beim Aufruf von `terraform apply`, den Ist-Zustand und den gewünschten Soll-Zustand in Einklang zu bringen. Alles, was nicht passt, wird passend gemacht. Falls die Änderung aufgrund von Beschränkungen des Providers etc. nicht möglich ist, wird abgerissen und neu gebaut.

Dieses Verhalten kann in produktiven Umgebungen potenziell zu Datenverlust und Unterbrechungen führen. Daher sollten Sie sich dessen bewusst sein und diese Informationen in Ihren Arbeitsablauf einbauen.

Terraform bietet mit dem sogenannten `lifecycle`-Parameter (den die Dokumentation gern als `meta-argument` bezeichnet) in bedingtem Maße Möglichkeiten, Einfluss auf das Verhalten von Terraform zu nehmen. Inwiefern das sinnvoll ist und ob diese