Technische Hochschule **Rosenheim**

Fakultät für Informatik

Studiengang: Informatik Master

CI/CD – Verfahren im Serverless Cloud Computing

Master Thesis

von

Dominik Ampletzer

Datum der Abgabe: 01.03.2021
Erstprüfer: Prof. Dr. Hüttl
Zweitprüfer: Prof. Dr. Tilly

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Rosenheim, den 01.03.2021

Dominik Ampletzer

Kurzfassung

In dieser Masterarbeit werden Grundlagen, Vorteile und Best-Practices des DevOps sowie dessen CI/CD-Werkzeuge und Infrastructure-as-Code (IaC) behandelt. Des Weiteren werden Cloud-Computing-Grundlagen für CI/CD erläutert, mit Fokus auf Amazon Web Services (AWS) und den Serverless-Ansatz.

Hauptfokus der Arbeit ist das Framework NTT-Serverless-CI/CD, welches auf einfache Weise Serverless-CI/CD in AWS ermöglicht. NTT-Serverless-CI/CD stammt von NTT DATA Deutschland GmbH und wird voraussichtlich im ersten Halbjahr 2021 veröffentlicht. Inhalte und Architektur von NTT-Serverless-CI/CD werden erläutert. Anhand der Beispiele Microservice und Angular-Applikation wird gezeigt, wie NTT-Serverless-CI/CD angewendet werden kann und welche Besonderheiten berücksichtigt werden müssen. Außerdem geht aus den Beispielen hervor, was es macht und was nicht.

Es initialisiert in AWS alle benötigten Rollen, Rechte, Lambda-Funktionen und weitere Infrastruktur, um einen automatisierten CI/CD-Prozess nutzen zu können. Es nimmt Entwickler/innen nicht die Erstellung der Pipeline und der für Software benötigten Infrastruktur, durch CloudFormation dem IaC-Werkzeug von AWS, ab.

Weiter wird auf Sicherheitsrisiken durch NTT-Serverless-CI/CD eingegangen, welche ebenfalls auf allgemeine Sicherheitsrisiken der Cloud zurückzuführen sind. Hierfür werden Best-Practices und Vorschläge gezeigt, welche das Sicherheitsrisiko verringern können.

Es wird gezeigt, welche Kosten durch NTT-Serverless-CI/CD anfallen. Hierfür wurden drei Szenarien vorgestellt, ein Monolith, ein Microservice und eine Web-Applikation. Für diese wird mittels eines Werkzeugs in Form einer Excel-Datei unter diversen Messungen, qualifizierten Annahmen und Schätzungen je zwei Kostenkalkulationen erstellt. Eine Kostenkalkulation für ein Build-Instanzen-Äquivalent zu dem tatsächlichen CI/CD-Server und eine Kostenkalkulation für eine günstigere kleinere Server-Instanz.

Aus den Kostenkalkulationen geht hervor, dass die Mehrkosten von NTT-Serverless-CI/CD gegenüber dem regulären Serverless-CI/CD-Ansatz von AWS durch die Vorteile von NTT-Serverless-CI/CD mehr als ausgeglichen werden.

Die ermittelten Kosten werden mit Kosten für EC2-Instanzen verglichen, da die echten Kosten einer CI/CD-Infrastruktur aus der Praxis nicht verfügbar sind bzw. nicht veröffentlicht werden dürfen. Aus dem Vergleich geht hervor, dass NTT-Serverless-CI/CD für Software, welche eine niedrige bis mittlere Build-Dauer, -Häufigkeit und Hardware-Anforderung haben, geringere Kosten verursacht und verwendet werden sollte.

Für große Software, welche eine hohe Build-Dauer, -Häufigkeit und hohe Hardware-Anforderung hat, ist NTT-Serverless-CI/CD im Kostenvergleich ungeeignet. In diesem Fall sollten Alternativen betrachtet werden, bzw. die Kosten genauer analysiert werden.

Traditionelle Server könne NTT-Serverless-CI/CD in wirtschaftlicher Hinsicht auch überlegen sein, wenn diese viele kleine bis mittlere Software-Build-Prozesse unterstützen. Dies liegt daran, dass die Kosten der CI/CD-Infrastruktur verteilt werden können. Jedoch mit den Nachteilen traditioneller Server und ohne Vorteile von NTT-Serverless-CI/CD.

Schlagworte: DevOps, Serverless, CI/CD, AWS, Infrastructure-as-Code, CloudFormation

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Code Verzeichnis

1 Einleitung

Cloud-Computing, Serverless und DevOps sind aktuelle Schlagworte und Begriffe, welche in der Wirtschaft stark verbreitet sind. Nach [Cap20] nutzen bzw. implementieren von den befragten Unternehmen aus der DACH-Region derzeit 28,6% Serverless Computing, 45,5% DevOps und 9,7% Infrastructure-as-Code. Daraus kann geschlossen werden, dass diese Themen einen wirtschaftlichen Vorteil versprechen oder zumindest eine Lösung bestehender Probleme darstellen.

In dieser Arbeit werden Grundlagen für DevOps und dem Serverless-Ansatz für CI/CD im Cloud-Computing behandelt. Dies beinhaltet die Werkzeuge von DevOps und allgemeine Cloud-Computing-Grundlagen, jedoch stark gekürzt und mit dem Schwerpunkt auf Amazon Web Services (AWS). Es wird das neue Framework NTT-Serverless-CI/CD vorgestellt und analysiert.

Dieses Framework wird von NTT DATA Deutschland GmbH entwickelt und voraussichtlich im ersten Halbjahr 2021 veröffentlicht. NTT-Serverless-CI/CD, mit großem Fokus auf Automatisierung, vereinfacht die Implementierung und den Betrieb eines Serverless-CI/CD-Ansatzes in AWS stark.

Um dies zu zeigen, wird die Architektur und das Vorgehen von NTT-Serverless-CI/CD erörtert. Außerdem wird anhand der Beispiele Microservice und Angular-Applikation gezeigt, wie NTT-Serverless-CI/CD zu verwenden ist, welche Besonderheiten, Vorteile und Nachteile es bietet.

Des Weiteren werden die Kosten von NTT-Serverless-CI/CD gegenüber reservierten AWS-EC2-Instanzen geprüft. Für die Kostenberechnung wird Excel als Hilfsmittel genutzt. Die hierfür genutzte Excel-Datei kann verwendet werden um die exakte Kostenberechnung nachzuvollziehen, und auch für eine eigene Kostenberechnung.

Als Grundlage für die Kostenberechnung werden drei Szenarien vorgestellt. Ein klassischer Monolith mit einer großen Menge an Quell-Code, ein kleiner Microservice und eine durchschnittliche Web-Applikation nach 1,5 Jahren Entwicklung.

Diese Arbeit klärt folgende Fragen:

- Was sind DevOps, CI/CD, Infrastructure-as-Code und Serverless?
- Welche Vorteile bieten DevOps und der Serverless-Ansatz gegenüber traditionellen OnPremise-Lösungen?
- Was ist NTT-Serverless-CI/CD und welche Vorteile bietet es?
- Welche Kosten verursacht NTT-Serverless-CI/CD?
- Wann sollte NTT-Serverless-CI/CD verwendet werden?

1.1 Motivation

Nach Zambrano [Zam18][1] ist eine Funktion bzw. eine Abfolge von Aktionen dann geeignet Serverless umgesetzt zu werden, wenn sie folgendes erfüllt:

- Zustandslosigkeit
- Gering an Rechenaufwand und vorhersehbar
- Begrenzte Lebenszeit

Weiter lassen sich nach einer Umfrage des O'Reilly-Verlags große Vorteile mit dem Serverless-Ansatz erzielen. So gaben die Befragten folgende Vorteile an [Mag19]:

60% Reduzierte operative Kosten

58% Automatische Skalierung

55% Keine Wartungsaufwand

32% Reduzierte Entwicklungskosten

30% Erhöhte Entwicklerproduktivität

Da der CI/CD-Prozess die Anforderungen nach Zambrano erfüllt und der Serverless-Ansatz einige Vorteile bietet, stellt sich die Frage: *Wie kann eine Serverless-CI/CD-Prozess aussehen?*

Dieser Frage stellen sich Fuad Ibrahimov und Michael Loibl von NTT DATA Deutschland GmbH. Ihre Antwort ist das Framework NTT-Serverless-CI/CD. NTT-Serverless-CI/CD verspricht auf effiziente Weise Serverless-CI/CD in AWS automatisiert betreiben zu können.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, grundlegendes Verständnis für DevOps und dessen Werkzeuge CI/CD und Infrastructure-as-Code zu schaffen. Außerdem wird gekürztes Grundwissen für Cloud-Computing mit dem Schwerpunkt AWS vermittelt. Dabei sind besonders die Vor- und Nachteile von DevOps, dessen Werkzeuge und der Serverless-Ansatzes von Bedeutung.

Es wird aufgezeigt, welche Vor- und Nachteile NTT-Serverless-CI/CD im Vergleich zu traditionellen CI/CD-Lösungen und den AWS-CI/CD-Werkzeugen ohne NTT-Serverless-CI/CD aufweist.

Für NTT-Serverless-CI/CD wird ein Leitfaden für die Verwendung anhand zweier Beispiele, einen Microservice und einer Angular-Applikation aufgezeigt. Außerdem werden Code-Vorlagen und zur einfachen Verwendung eine Check-Liste C.2 erstellt.

Weiter wird eine Hilfe für die Kostenkalkulation in Form einer Excel-Datei erstellt, um vor der Verwendung von NTT-Serverless-CI/CD die Kosten abschätzen zu können. Mit dieser Hilfe wird für drei mögliche Szenarien: Monolith, Microservice und Web-Applikation eine Kostenkalkulation erstellt.

1 Kapitel: The sweet spot

1.3 Rahmenbedingungen und Abgrenzung

Diese Arbeit ist folgenden Rahmenbedingungen unterworfen:

- Als Cloud-Provider ist AWS zu verwenden.
- Erzeugte Beispiele müssen aus einem Back-End und Front-End bestehen, welche unabhängig voneinander erzeugt werden, jedoch zusammen ein System bilden und miteinander interagieren können.
- Der Fokus liegt auf dem Build-Prozess, da dieser durch NTT-Serverless-CI/CD abgedeckt ist. Die zu erzeugende Infrastruktur und das Veröffentlichen von Software wird nur am Rande betrachtet.

Hieraus sind folgende Abgrenzungen festzuhalten:

Es werden keine Cloud-Provider oder Serverless-Ansätze unterschiedlicher Cloud-Provider verglichen. Fokus der Arbeit ist der Vergleich von Serverless-CI/CD gegenüber traditionellen CI/CD-Servern.

Ebenfalls findet kein Vergleich unterschiedlicher IaC-Werkzeuge statt. Der zu verwendende Cloud-Provider ist AWS. In diesem Zug wird das IaC-Werkzeug CloudFormation vorgestellt. Dieses ist in AWS integriert und wird von NTT-Serverless-CI/CD verwendet.

Diese Arbeit beschäftigt sich nicht mit den Vor- oder Nachteilen, welche Software-Architekturen für Serverless-Ansätze mit sich bringen. Auch wenn diese von großer Bedeutung für die Software und die Kosten sind. Für den Build-Prozess spielt es eine untergeordnete Rolle, ob das erzeugte Artefakt, z.B. eine WAR-, JAR-, JS-Datei oder eine Docker-Image ist. Dass die Architektur eine Auswirkung auf die Build-Häufigkeit und Build-Dauer hat und dadurch den Build-Prozess stark beeinflusst, steht in dieser Arbeit außer Frage.

Auf das Veröffentlichen von Software wird nur theoretisch eingegangen, da AWS alle gängigen Veröffentlichungsstrategien unterstützt und die Veröffentlichung stark von der jeweiligen Software abhängig ist. Die Veröffentlichung ist im Generellen sehr wichtig, spielt jedoch für NTT-Serverless-CI/CD eine untergeordnete Rolle, da es die Möglichkeiten von AWS nutzt. Dies bezieht sich auch auf die Infrastruktur der zu betreibenden Software. NTT-Serverless-CI/CD bietet über AWS die Möglichkeit unterschiedlichste Infrastrukturen zu erzeugen, alle jedoch durch die gleiche Methode. Da die Infrastruktur abhängig von der Software ist und die Methode sich dadurch nicht verändert, wird nur am Rande darauf eingegangen.

2 DevOps und dessen Werkzeuge

In diesem Kapitel werden die grundlegende Begriffe und Konzepte des DevOps erläutert. Hierzu gehören unter anderem Continuous-Integration, Continuous-Delivery, Continuous-Deployment und wie diese zusammen zu CI/CD zusammengefasst werden. Weiter wird noch auf Infrastructure-as-Code sowie Synergien und Vorteile eingegangen, die unter DevOps entstehen.

2.1 Continuous-Integration

Der Begriff Continuous-Integration (CI) wurde erstmalig von Beck [Bec99][1] eingeführt. CI stammt ursprünglich aus der Entwicklungsmethode *Extreme Programming* und somit auch aus der agilen Software-Entwicklung.

Folgende Schritte sind nach Fowler [Fow06] nötig um CI zu betreiben:

1. Lokal den *Code* vom Source Code Management (SCM) *beziehen*
2. *Aufgabe abschließen* inklusive Testfälle
3. Lokal Code *testen und kompilieren*/bauen
4. Nach erfolgreichem Testen und Kompilieren den *Code* dem SCM *hinzufügen*
5. *Konflikte* durch das Hinzufügen des neuen Codes (sog. Merge) umgehend *beheben*
6. *Integrationsmaschine testet und baut* den Quell-Code
7. Im Fehlerfall werden *Fehler umgehend behoben* und der nun neue Code dem SCM hinzugefügt

Der zweite Schritt *Aufgabe abschließen* kann wie bei vielen Definitionen agiler Software-Entwicklung sehr weit gefasst werden. Die Aufgabe sollte jedoch klein sein, auch wenn nicht geklärt ist, wie groß genau eine Aufgabe zu sein hat. Um das Risiko von Merge-Konflikten gering zu halten, sollten möglichst mehrmals am Tag die beschriebenen Schritte durchgeführt werden. Nach Soni [Son15][S.1] soll jede/r Entwickler/in mindestens einmal am Tag den Code integrieren, wobei eine häufigere Code-Integrierung dazu beiträgt Merge-Konflikte zu vermeiden bzw. unproblematisch zu halten. Heutige SCM sind i.d.R. in der Lage einfache Merge-Konflikte selbstständig zu beheben, was die Problematik von Merge-Konflikten verringert.

Für die erfolgreiche Anwendung von CI ist es außerdem wichtig, dass Tests schnell auszuführen sind [Bec99][2]. Dass Tests eine kurze Laufzeit haben sollen, sowie schnell und einfach auszuführen sind, ist auch bei Martin [Mar09][S.171] zu finden, da Entwickler/innen dazu neigen, diese Tests sonst nicht auszuführen. Dies würde dazu führen, dass Tests erst spät

1 Kapitel: 11. How Could This Work?
2 Kapitel: 11. How Could This Work?

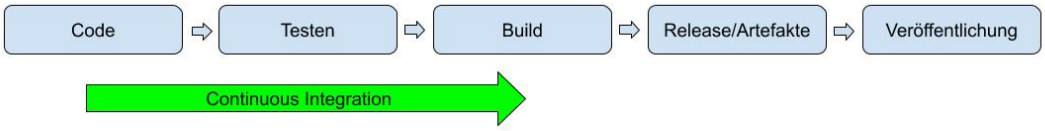


Abbildung 2.1 CI-Prozess

im Prozess fehlschlagen und somit erst spät erkannt und entsprechend spät behoben werden können. Dieses Prinzip des frühen Scheiterns ist nach Soni [Son15][S.1] die Grundlage von DevOps.

Um dieses frühe Scheitern zu ermöglichen ist es nötigt, dass Tests eine hohe Qualität haben und dass das Richtige getestet wird. Dies schließt Grenzfälle mit ein. Tests sind daher genauso wichtig wie Produktions-Code und sollen der gleichen sorgfältigen Entwicklung unterliegen [Mar09][S.162].

Um den CI-Prozess möglich zu machen, ist es nötig, dass die Integrationsmaschine (CI-Server) das Testen und Kompilieren automatisiert durchführt [Fow06]. Dies wird auch Build genannt. Der CI/CD-Server, hat nach Rossel [Ros17][3] die Aufgaben die Software zu bauen, zu testen, über Scheitern und Gelingen zu benachrichtigen und am Ende ein Artefakt zu erstellen. Hier kann Rossel jedoch nicht voll zugestimmt werden, da dies sich nicht mit Beck und anderen Autoren deckt. In 2.1 ist aufgezeigt, welche Bereiche dem CI zuzuschreiben sind. Dies schließt das Erstellen eines Artefaktes nicht mit ein, auch wenn diese Abgrenzung in der Praxis keine Rolle spielt.

Weiter ist anzumerken, dass Fowler [Fow06] auf den Umstand hinweist, dass für einen automatisierten Build alle benötigten Abhängigkeiten dem Build-Prozess zur Verfügung stehen müssen. Dies beinhaltet auch Datenbank-Schemata und gegebenenfalls Ziel- bzw. Testumgebungen.

Beim Scheitern der Tests oder der Kompilierung muss das Entwicklungs-Team und/oder - falls dies technisch möglich ist - der/die Entwickler/in, welche/r das Scheitern verursacht hat/haben umgehend durch den CI-Server informiert werden. Hierbei ist der Fokus auf das Beheben des Fehlers zu legen, um einen fehlerfreien Build zu erreichen [Kan17][4]. Dies setzt folglich voraus, dass die Arbeitskultur CI auch verinnerlicht ist. So sollte ein/e Entwickler/in keinen Code integrieren, ohne das Ergebnis über eine erfolgreiche Integration des CI-Server abzuwarten. Dies beinhaltet bei einer Fehlermeldung auch, dass der/die Entwickler/in nicht, wie möglicherweise beabsichtigt, den Arbeitsplatz verlassen kann und stattdessen den Fehler beheben muss.

Die grundlegenden Herausforderungen [Sta17][S.6] des CI sind zu gleich folgende Best-Practices [Mis18][5]:

- Regelmäßige Code-Integration
- Verwenden eines SCM
- Build- und Test-Automatisierung
- Jede Integration muss automatisiert den Build-Prozess durchlaufen

3 Kapitel: Continuous Integration
4 Kapitel: 4. Build, Test, and Release Faster with Continuous Integration
5 Kapitel: Continuous Integration best practices

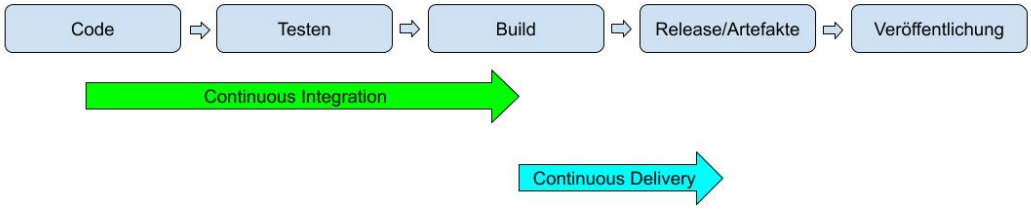


Abbildung 2.2 CD-Prozess

Stacy beschreibt die Ziele welche durch CI verfolgt werden als: frühzeitiges Erkennen von Fehlern, Erhöhen der Software Qualität und Verringern der Zeit um neue Software-Versionen zu erzeugen [Sta17][S.6]. Dies zeigt, wie auch bei Rossel, wie ungenau die Abgrenzung zu Continuous-Delivery ist, da das Erzeugen von Software-Versionen die Hauptaufgabe von Continuous-Delivery ist. Fowler beschreibt das Ziel von CI als das Verringern von Risiken [Fow06], was wiederum weniger Fehler und höhere Software-Qualität beinhaltet.

2.2 Continuous-Delivery

Stacy [Sta17][S.6-7] beschreibt Continuous-Delivery (CD) als eine Erweiterung der Continuous-Integration. Diese Erweiterung ist in Abbildung 2.2 zu sehen, wodurch auch klar wird, warum CI mit CD einhergeht. Da CD zum Ziel hat, immer release-fähige Software zu erzeugen, was die Prozessschritte von CI mit einschließt. Durch den Build-Schritt des CI wird i.d.R ein Artefakt erzeugt. Dieses Artefakt ist jedoch oft nur bedingt release-fähig. Der Fokus von CD liegt auf der Release-Fähigkeit der Artefakte. Dies bedeutet, dass der ausgeführte Build auch für die Zielumgebung erzeugt werden muss bzw. alle nötigen Konfigurationen für diese beinhalten muss. Abhängig von der Software kann dies der gleiche Build-Schritt wie bei CI sein. Dies ist bei Software der Fall, welche die Konfiguration zur Laufzeit erhalten oder wo es nur eine Umgebung gibt. Andernfalls benötigt der Build für die Zielumgebung einen eigenen Prozessschritt.

Die Automatisierung und Vereinfachung des Release-Prozesses ist der nächste logische Schritt, nach der Einführung von CI. Da CI zu mehreren kleineren Änderungen führt, welche in mehrere kleinere Releases überführt werden können [Ros17][6].

Fowler [Fow13] beschreibt den CD-Prozess als Disziplin, welche jederzeit dafür sorgt, dass eine Produktions-Release jederzeit stattfinden **kann**. Dies bedeutet, dass die Software während des ganzen Lebenszyklus release-fähig ist, auch wenn an neuen Funktionalitäten gearbeitet wird. Dadurch wird jeder Build automatisiert den Zustand der Release-Fähigkeit erhalten können und jederzeit **kann** per Knopfdruck jede Version veröffentlicht werden. Das hat zur Folge, dass sich CD mit der Erzeugung der Releases und auch der Verwaltung der jeweiligen Releases beschäftigt. Es ist auf das **kann** hinzuweisen. Da Continuous-Delivery nicht die Notwendigkeit beinhaltet jede Änderung zu veröffentlichen. Es muss nur diese Möglichkeit bestehen, jederzeit zu veröffentlichen. Diesen ist ebenfalls bei Stacy [Sta17][S.3] zu finden. Das Ziel von CD ist es somit jederzeit die Möglichkeit zu haben, die aktuelle Software zu veröffentlichen, aber nicht dies zu tun!

CD bringt den großen Vorteil mit sich, dass es das Veröffentlichungsrisiko erheblich

6 Kapitel: Continuous Delivery

reduziert. Mit CD ist es möglich, jederzeit Releases zu erstellen. Dies mindert nicht das Risiko eines fehlgeschlagenen Releases, jedoch wird die Auswirkung des fehlgeschlagenen Releases gemindert, da nachdem die Ursache für den Fehlschlag behoben ist, automatisiert ein neues Software-Release erzeugt werden kann, welches zur Veröffentlichung bereit steht. CD verzichtet auf einen langwierigen Release-Zyklus und macht i.d.R. größere Absprachen innerhalb eines Unternehmens unnötig, um einen Bugfix/Hotfix zu erstellen [Fow13]. Ob und wann die Veröffentlichung stattfindet, ist wiederum vom Projekt abhängig. Wichtig hierbei ist auch, dass die Möglichkeit schnelle Releases zu veröffentlichen, die Tests nicht überflüssig macht. CD ist eine Erweiterung von CI und Tests hoher Qualität sind nötig um CI erfolgreich zu nutzen.

Stacy [Sta17][S.3-4] führt folgende weitere Vorteile von CD auf:

Automatisierung des Software-Release-Prozesses Diese Automatisierung führt zu Effizienz. Der Release-Prozess ist wiederholbar, schnell und sicher.

Erhöhung der Produktivität Durch CD werden Aufgaben, welche zuvor manuell ausgeführt werden mussten, automatisiert. Dies hat zur Folge, dass das Entwicklungs-Team sich um weniger manuelle Aufgaben kümmern muss, welche von Natur aus fehleranfällig sind und bei Releases auch eine gewisse Angst vor Fehlern verursachen. Stattdessen kann sich das Entwicklungs-Team um seine eigentliche Aufgabe, dem Entwickeln von Software, kümmern.

Höhere Code-Qualität Die höhere Qualität beruht auf dem CI-Prozess, da Fehler durch regelmäßige Integration, was Testen und Kompilieren einschließt, früher gefunden werden können. Dabei muss CI jedoch um mögliche Release-Builds und Tests erweitert werden.

Schnellere Lieferung von Updates Dies ist ebenfalls auf die Automatisierung zurückzuführen, da z.B. für einen Bugfix die Ursache behoben und getestet werden kann, ohne einen Release-Zyklus abzuwarten.

Bei all diesen Vorteilen ist jedoch wichtig zu verstehen, dass es sich um Möglichkeiten handelt. Es geht nicht automatisch mit der Einführung von CI und CD einher, dass diese Vorteile erzielt werden. Dies liegt daran, dass das Entwicklungs-Team für die Integrationsrate, die Qualität von Quell-Code und Tests sowie für das „Leben“ von CI und CD im Arbeitsalltag verantwortlich ist.

2.3 Continuous-Deployment

Continuous-Deployment ist das letzte Konzept, welches benötigt wird, um die komplette CI/CD-Pipeline abbilden zu können. Continuous-Deployment ist für den Prozessschritt des Veröffentlichens von erzeugten Artefakten verantwortlich. Dieser Prozessschritt ist in Abbildung 2.3 zu sehen. Das Continuous-Deployment soll in der Lage sein, automatisiert jedes durch CD erzeugte Artefakt zu veröffentlichen. Die Methode bzw. Strategie des Veröffentlichens ist stark von der jeweiligen Software oder deren Anforderungen abhängig. So kann Veröffentlichen z.B. bedeuten, dass auf eine Cloud-Instanz ein neues Release aufgespielt wird, ein Update für eine Applikation zur Verfügung gestellt wird, eine Web-Seite aktualisiert wird oder das erzeugte Release auf einer Web-Seite zum Download angeboten wird. Genau wie bei CD in Kapitel 2.2 ist bei Continuous-Deployment auf das **Kann**

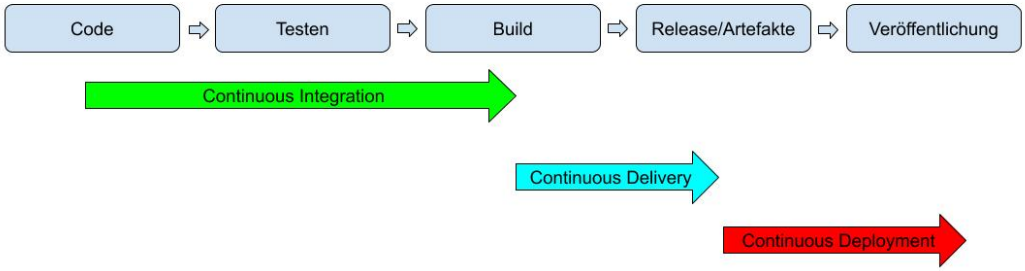


Abbildung 2.3 CI/CD-Prozess

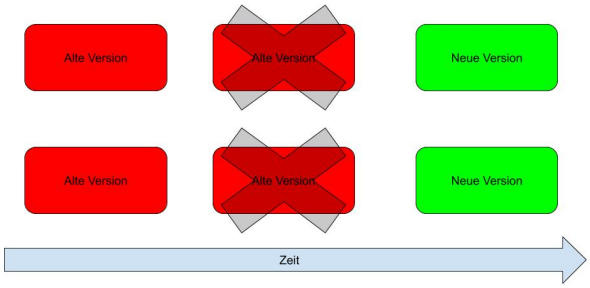


Abbildung 2.4 „Alles auf einmal"-Strategie

hinzuweisen. Continuous-Deployment bedeutet nicht, dass jede Änderung unverzüglich veröffentlicht wird oder in Produktion geht [Las20], sondern die Möglichkeit dazu besteht.

Die Vorteile, welche das Continuous-Deployment bietet, sind [Fow13] [Mis18][7]:

Minimierung des Deployment Risikos: Regelmäßig ausgeführte Releases tragen dazu bei, Probleme frühzeitig zu erkennen.

Nachverfolgung des Fortschrittes: Der Fortschritt der Software kann durch Stakeholder auf der Produktiv- oder Integrationsumgebung über die Zeit beobachtet werden und, im Falle von Integrationsumgebungen, auch ausführlich getestet werden, bevor die Änderungen in die Produktivumgebungen übernommen werden.

Rückmeldung der Nutzer: Das größte Risiko jeglicher Software ist, dass das Falsche entwickelt wird und die Nutzer die Software nicht akzeptieren. Da es durch Continuous-Deployment möglich ist, regelmäßig kleine Änderungen zu veröffentlichen, ist es möglich, Rückmeldung von Nutzern für kleine Software-Änderungen und Funktionalitäten einzuholen. Hierdurch kann frühzeitig auf Rückmeldungen eingegangen werden und auch im Falle negativer Rückmeldung gegengesteuert werden.

Das Continuous-Deployment kann mit verschiedenen Strategien umgesetzt werden. So ist z.B. für das Erzeugen einer ISO-Datei, welche eine neue Software-Version beinhaltet, ein anderes Vorgehen nötig, als für das Ausrollen einer neuen Version eines Docker-Images in einem Netzwerk-Cluster. Diese Strategien werden nach Shipley wie folgt beschrieben [Shi16][S.48-50]:

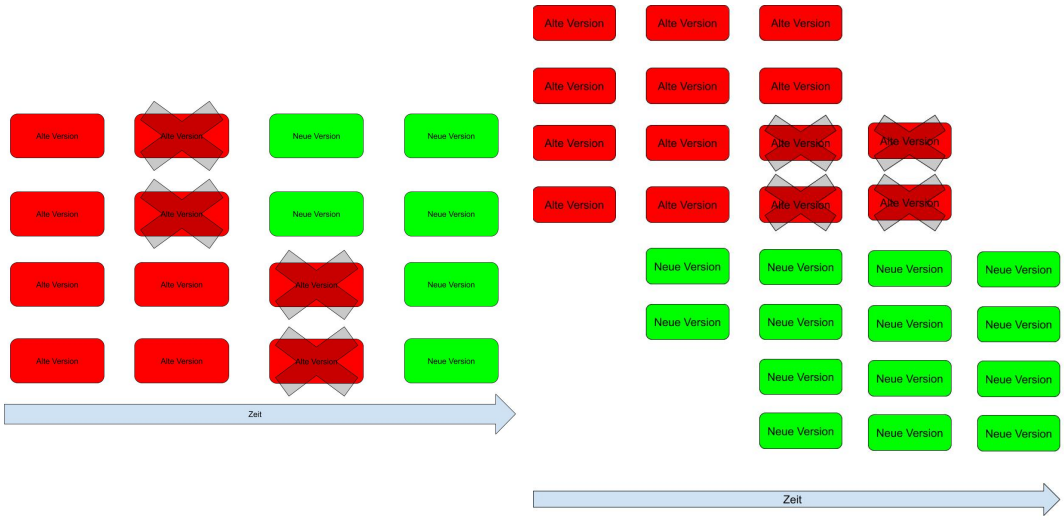
7 Kapitel: Continuous Delivery benefits

Alles auf einmal Hierbei wird die neue Software-Version auf allen Instanzen gleichzeitig veröffentlicht. Bei diesen Instanzen kann es sich z.B. um Docker-Container oder Applikations-Server handeln. Dies ermöglicht ein schnelles Veröffentlichen, hat jedoch i.d.R. zur Folge, dass die Instanzen eine gewisse Zeit nicht zur Verfügung stehen. In diesem Fall wird von Down-Time gesprochen. Dieser Ablauf wird in Abbildung 2.4 dargestellt. So ist zu sehen, dass zuerst die alte Software-Version abgeschaltet wird und erst nach einer gewissen Zeit die neue Software-Version verfügbar ist. Diese Strategie wird oft für die Entwicklung verwendet, da es die schnellste und kostengünstigste Strategie ist.

Ausrollen Das Ausrollen ist nur dann möglich, wenn es überhaupt mehrere Instanzen gibt, auf die ausgerollt werden kann. Es wird wie Abbildung 2.5a zeigt, ein Teil der Instanzen mit der alten Software-Version abgeschaltet, darauf werden Instanzen mit der neuen Software-Version gestartet bzw. aufgespielt. Sind diese neuen Instanzen verfügbar, wird der nächste Teil der alten Software-Instanzen abgeschaltet usw. bis alle Instanzen mit der neuen Software-Version verfügbar sind. Hierbei muss nicht, wie in der Abbildung zu sehen, die Hälfte der Instanzen abgeschaltet werden. Es ist von der jeweiligen Veröffentlichung abhängig, wie viele Instanzen abgeschaltet werden. Jedoch ist zu bedenken, je weniger Instanzen gleichzeitig abgeschaltet werden bzw. die neue Software-Version aufgespielt bekommen, desto länger benötigt das Ausrollen bis alle Instanzen die neue Software-Version zur Verfügung stellen. Diese Strategie hat den Vorteil, dass die Software immer zur Verfügung steht, jedoch mit geringer Kapazität bis die neue Software-Version komplett ausgerollt ist.

Ausrollen mit zusätzlichen Instanzen Diese Strategie beruht auf dem Ausrollen. Es werden jedoch, um dem Nachteil der geringeren Kapazität entgegenzuwirken, zusätzliche Instanzen verwendet. Abbildung 2.5b zeigt, dass anstatt Instanzen abzuschalten bzw. die neue Software-Version aufzuspielen, neue zusätzliche Instanzen verwendet werden und diese die neue Software-Version erhalten. Sind dann diese zusätzlichen Instanzen verfügbar, werden Instanzen mit der alten Software-Version abgeschaltet. Dies wird ebenfalls wiederholt, bis nur noch Instanzen mit der neuen Software-Version verfügbar sind. Die Geschwindigkeit, mit der diese Strategie das Veröffentlichen durchführt, ist stark von der Anzahl zusätzlicher Instanzen abhängig und mit entsprechenden zusätzlichen Kosten verbunden.

Unveränderbar Blue/Green Die unveränderbare oder auch „Blue/Green"-Strategie könnte als extreme Variante des „Ausrollen mit zusätzlichen Instanzen"-Strategie betrachtet werden. Hierbei wird die neue Software-Version auf zusätzlichen Instanzen durchgeführt, welche in der Kapazität der ursprünglichen Instanzenzahl identisch sind. Auf diese zusätzlichen Instanzen wird mittels „Alles auf einmal" die neue Software-Version aufgespielt und bildet so ein neues unabhängiges System. Ist dies abgeschlossen und steht das System mit der neuen Software-Version zur Verfügung, werden alle Anfragen auf dieses neue System umgeleitet. Anschließend kann das System mit der alten Software-Version abgeschaltet werden bzw. steht für das nächste Software-Release als zusätzliche Instanzen zur Verfügung. Da es verwirrend sein kann, welches System die neue/alte Software-Version nutzt, werden diese Systeme zur Unterscheidung oft als **Blue** und **Green** bezeichnet. Bei der „Blue/Green"-Strategie handelt es sich um die kostenintensivste Strategie, jedoch vereint Sie die Vorteile der „Alles auf einmal"-Strategie mit keiner oder nur einer geringen Down-Time.



(a) „Ausrollen“-Strategie

(b) „Ausrollen mit zusätzlichen Instanzen“-Strategie

Abbildung 2.5 Varianten der „Ausrollen“-Strategie

Anfragen aufteilend Hierbei handelt es sich um eine Variante der „Blue/Green“-Strategie, welche auch Canary genannt wird. Wie bei der „Blue/Green“-Strategie wird ein zweites System benötigt, welches die neue Software-Version veröffentlicht. Jedoch werden nicht alle Anfragen auf das System mit der neuen Software-Version umgeleitet. Es wird, um die Stabilität und Funktionalität des Systems zu prüfen, nur ein Teil der Anfragen umgeleitet. Sollte das System stabil sein und wie gewünscht funktionieren, wird nach und nach der Rest der Anfragen auf das neue System umgeleitet. Wenn alle Anfragen umgeleitet sind, kann wie bei der „Blue/Green“-Strategie das alte System abgeschaltet werden oder es steht als Canary für eine neue Software-Version zur Verfügung. Bei dieser Strategie dauert die komplette Veröffentlichung länger als bei der „Blue/Green“-Strategie, sie ist jedoch sicherer, da bei einer fehlerhaften neuen Version schnell auf die alte Version zurück gewechselt werden kann. Außerdem ist bei einer fehlerhaften Version nur ein Teil der Nutzer betroffen, was zu weniger Unmut bei der Gesamtmenge der Nutzer führt.

Diese Ansätze sind so oder mit kleinen Abweichungen auch bei anderen Autoren zu finden und werden von Cloud-Providern wie AWS bereits als Konfigurationsmöglichkeiten für Veröffentlichungen zur Verfügung gestellt. Diese unterschiedlichen Strategien können nicht nur von Software zu Software unterschiedlich angewendet werden. Es ist auch möglich und sinnvoll, für unterschiedliche Zielumgebungen unterschiedliche Veröffentlichungsstrategien zu verwenden. So ist die „Alles auf einmal“-Strategie oft für Produktivumgebungen ungeeignet, da sie eine hohe Down-Time mit sich bringt, jedoch für eine Entwicklungsumgebung die vermutlich beste Wahl, da sie schnell durchzuführen ist und die Kosten am geringsten sind.

2.4 CI/CD

Die eingeführten Begriffe Continuous-Integration, Continuous-Delivery und Continuous-Deployment werden unter der Abkürzung CI/CD zusammengefasst. Die Verwendung von CI/CD ermöglicht durch die Automatisierung das Erfüllen von Prinzipien des Agilen-Manifests [Bec01] wie „...frühe und kontinuierliche Auslieferung...“ und „Liefern funktionierender Software regelmäßig innerhalb weniger Wochen...“.

CI/CD kann i.d.R. komplett automatisiert werden, jedoch gibt es Fälle in denen manuelle Interaktion nötig ist. Diese manuelle Interaktion ist nötig bei folgenden Fällen [Las20]:

- Bei menschlicher Validierung, wie z.B. Akzeptanztests durch Nutzer oder Fachabteilungen, Veröffentlichung auf Produktivumgebung
- Wenn der CI/CD-Prozess geändert oder erweitert werden muss
- Wenn der automatisierte Prozess nicht funktioniert

Generell kann festgehalten werden, dass alle Aufgaben welche durch Software automatisiert werden können, auch durch Software automatisiert werden sollten, auch wenn noch immer eine menschliche Validierung nötig ist [Dob20][S.112].

Bei Mistry [Mis18][8] sind folgende Vorteile von CI/CD zu finden:

- Verbesserung in der Zusammenarbeit (von Entwicklung und Betrieb)
- Reduzierung der Kosten
- Höhere Qualität
- Schnelleres Umsetzen von Änderungen
- Regelmäßigeres Veröffentlichen von Funktionalitäten
- Stabilität und Widerstandsfähigkeit des Systems
- Weniger manueller Aufwand und weniger Zeitaufwand
- Unabhängigkeit von einzelnen Personen (Urlaub, Ausscheiden). Da das benötigte Wissen für Veröffentlichungen in der Automatisierung vorhanden ist, ist jeder in einem Entwicklungs-Team in der Lage zu veröffentlichen [Ros17][9].

Um CI/CD umzusetzen, können viele verschiedene Werkzeuge genutzt werden. Im Open-Source-Bereich ist das vermutlich bekannteste Jenkins[10]. Neben Jenkins gibt es noch eine vielfältige Auswahl an anderen Werkzeugen. Dies betrifft auch proprietärer Software. Eines dieser Werkzeuge ist TeamCity[11]. Es gibt auch Werkzeuge, welche SCM und CI/CD kombinieren, z.B. GitLab[12].

All diese Werkzeuge verfolgen das gleiche Ziel: Einen hohen Grad an Automatisierung durch eine CI/CD-Pipeline. Abbildung 2.7 zeigt eine vereinfachte CI/CD-Pipeline. Diese ist in vier Hauptbereiche, auch *Stages* genannt, unterteilt:

8 Kapitel: Continuous Delivery benefits
9 Kapitel: Continuous Integration, Delivery, and Deployment Foundations
10 `https://www.jenkins.io/`
11 `https://www.jetbrains.com/teamcity/`
12 `https://about.gitlab.com/`

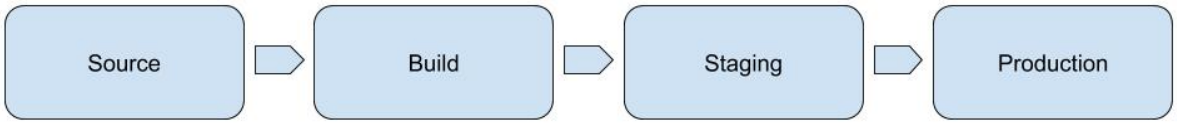


Abbildung 2.7 Vereinfachte CI/CD-Pipeline

Source Beziehen des Quell-Codes

Build Kompilieren des Quell-Codes und Erzeugen von Artefakten. Oft mit vor- und nachgelagerten Tests

Staging Veröffentlichen der Artefakte auf Test-/Integrationsumgebungen

Production Veröffentlichen auf der Produktivumgebung

Abbildung 2.7 ist auch zu entnehmen, dass die Abgrenzung zwischen Continuous-Integration, Continuous-Delivery und Continuous-Deployment nicht mehr klar ist. Die Konzepte jedoch sind innerhalb der CI/CD-Pipeline bzw. der Stages vorhanden, auch wenn diese sich über unterschiedliche Stages erstrecken.

Die Stages müssen auf den jeweilig benötigten CI/CD-Prozess angepasst werden, so kann z.B. das Staging entfernt werden, sollte es weder eine Test- noch eine Integrationsumgebung geben. Auch Szenarien ohne Build-Stage sind denkbar, wenn auch ungewöhnlich, falls die Software z.B. nur aus HTML und JavaScript besteht.

Für Tests kann es ebenfalls sinnvoll sein, diese nicht in der Build-Stage mit abzubilden, sondern stattdessen Unit-Tests, Integrations-Tests und End-to-End-Tests in einer oder mehreren Stages abzubilden.

Stacy beschreibt eine Reihe an Best-Practices, welche für CI/CD in AWS gelten [Sta17][S.27ff]. Diese können, von Entwicklungsmethoden und Cloud-Kontext bereinigt, als allgemeine CI/CD-Best-Practices wie folgt verwendet werden:

- Code-Änderungen sollen regelmäßig in das SCM integriert werden. Es sollen keine lang andauernden Feature-Branches verwendet werden.
- Testabdeckung sollte mind. 70% des Codes betragen, da die Tests zur Sicherheit im Code beitragen.
- Unit-Tests müssen aktuell gehalten und nicht vernachlässigt werden. Auch sollen fehlgeschlagene Tests behoben und nicht umgangen werden.
- Die Konfiguration von CI/CD muss als Code behandelt werden, z.B. Jenkins-Dateien werden mit im SCM gepflegt.
- Für jeden Code-Branch und jedes Entwicklungs-Team sollen unterschiedliche CI/CD-Pipelines genutzt werden.
- Die CI/CD-Pipeline muss überwacht werden und Warnungen müssen versendet werden.
- Metriken und Kontrolle sollen in den Lebenszyklus der Software und der Entwicklung eingeplant werden.

- Mit Hilfe der Metriken und den Erfahrungen aus dem Prozesses muss stetige Verbesserungen wachsen.

2.5 Infrastructure-as-Code

Infrastructure-as-Code (IaC) ist das Konzept, welches dafür steht, Wissen, Konfiguration und Verwalten von Infrastruktur in wiederverwendbare Skripte in Form von Code umzusetzen [Gue19][S.1]. Dies ermöglicht es, Infrastruktur wie regulären Quell-Code in einem SCM zu verwalten. Mit IaC können Vorteile erzielt werden, wie sie z.B. bei Borsa Istanbul (Türkische Börse) erreicht wurden. Durch den Einsatz des IaC-Werkzeuges Puppet wurde die Veröffentlichungsdauer von 10 Tagen auf eine Stunde reduziert [Rah19][S.2]. Intercontinental Exchange (ein Unternehmen aus den Fortune 500) konnte ebenfalls große Vorteile aus der Umstellung auf IaC erzielen. Intercontinental Exchange verwaltet 75% ihrer 20.000 Server mittels IaC und ist nun in der in der Lage Entwicklungsumgebung statt zuvor in 12 Tagen nun in 21 Minuten zur Verfügung zu stellen [Rah19][S.1].

Diese Beispiele zeigen, welches Potential IaC birgt. Dennoch ist IaC keine neue Erfindung. IaC wird schon seit langem von Administratoren mit durch die Verwendung von Shell-Skripten betrieben, jedoch nicht werkzeuggestützt und nicht in dem Umfang, wie es für Infrastruktur durch moderne Virtualisierungstechnologien möglich ist.

Werkzeuge wie Puppet, Ansible, Terraform oder die für diese Arbeit verwendete CloudFormation (Kapitel 3.3.3) bieten Lösungen für die von Cheung beschriebenen Probleme, welche durch manuell erzeugte Infrastruktur entstehen. Zu diesen Problemen zählen folgende [Che17][S.5]:

Hohe Kosten Es werden menschliche Ressourcen verwendet, um Infrastruktur zur Verfügung zu stellen. Diese sind teurer als automatisierte Vorgänge.

Inkonsistenzen Menschen machen Fehler, insbesondere wenn von Standard-Konfigurationen abgewichen wird.

Langsam Manuelle Infrastrukturen sind schwerfällig, da es lange dauert, bis Änderungen manuell eingepflegt bzw. umgesetzt sind. Dies kann neue Veröffentlichungen verlangsamen und den Marktgegebenheiten hinterherhängen.

Schlechte Wartbarkeit Durch meist schlechte Dokumentation und gebündeltes Wissen bei einzelnen Personen wird die Wartbarkeit erschwert. Auch Wiederholbarkeit ist hierdurch nicht gegeben. Da nicht exakt bekannt ist, wie eine existierende Infrastruktur entstanden ist.

Ein weiteres Problem stellt die Skalierbarkeit dar. Es ist durchaus möglich, Software zu entwickeln, welche horizontal skaliert und wodurch zusätzliche Instanzen die Kapazität erhöhen. Jedoch können zusätzliche Instanzen zur Erhöhung der Kapazität eines Web-Servers auch folgendes bedeuten: Eine ausgebildete Person des Operations-Teams muss manuell neue Virtuelle Maschinen (VM) starten. Diesen VMs müssen eindeutige IP-Adressen zugewiesen werden. Den IP-Adressen wiederum müssen Ports zugewiesen werden, auf welche der Web-Server reagiert. Zusätzlich muss ein Load-Balancer erweitert werden, um die neuen gestarteten VMs als Endpunkte zu führen. Dies benötigt Zeit, in welcher die Person des Operations-Teams nicht für andere Tätigkeiten zur Verfügung steht. Zudem ist dieser

Prozess fehleranfällig. Gerade bei sehr großen Applikationen ist das manuelle Hinzufügen von Instanzen kaum noch effektiv [Dob20][S.112-113].

IaC löst diese Probleme damit, dass das Wissen und die komplette Konfiguration der Infrastruktur in Code festgehalten wird. Dies führt dazu, dass der Code der „Single-Point-Of-Truth“ ist. Hierdurch kann der Code als Dokumentation dienen. Außerdem ermöglicht das Wissen um die Infrastruktur als Code die Automatisierung, welche die aus vorherigen Kapiteln bekannten Vorteile: Zeitersparnis, Fehlerreduzierung und Wiederholbarkeit mit sich bringt [Kan17][13].

Des Weiteren ist es durch unterschiedliche Konfigurationen möglich, verschiedene Umgebungen in kurzer Zeit zu erzeugen. Es ist durch unterschiedliche Konfigurationen möglich, dass Testumgebungen Produktivumgebungen entsprechen bis auf z.B. die Größe und Anzahl der verwendeten Applikationsserver. Da Testumgebungen möglichst Produktivumgebungen widerspiegeln, sollen jedoch aus Kostengründen nicht die gleiche Rechenleistung, Speicher- oder Datenbankkapazität benötigen. Auch Canaries und der Wechsel der Systeme kann durch IaC-Werkzeuge auf automatisierte Weise geschehen [Art17][S.2].

IaC ermöglicht auch das Testen neuer Infrastrukturen auf Feature-Branches. Dadurch besteht die Möglichkeit neue Infrastruktur zu testen ohne andere Umgebungen zu beeinflussen. Nach einem zufriedenstellenden Ergebnis kann der Feature-Branch in den Master-Branch über das SCM integrieren werden, wodurch bei entsprechendem CI/CD-Prozess die im IaC festgehaltene neue Infrastruktur zur Anwendung kommt [Spi12][S.2].

Vorteile, welche durch die Verwendung von IaC entstehen, werden von Kantsev wie folgt zusammengefasst, lediglich der Kostenaspekt wurde nicht berücksichtigt [Kan17][14]:

- Verwendung eines SCM für Infrastruktur ist möglich. Dies ermöglicht einfache Versionierung, Zurückrollen, Experimentieren und Nachvollziehbarkeit von Änderungen.
- Infrastrukturen können auf einfach Weise wiederverwendet, erweitert oder reproduziert werden.
- IaC dient als Dokumentation, was und wie exakt konfiguriert wurde.
- Das Bereitstellen von Cloud-Umgebungen ist trivial (durch Werkzeuge wie Terraform oder CloudFormation).
- Moderne Konfigurationsverwaltung wird ermöglicht.

Diese Vorteile sind jedoch mit nicht ganz geringen Einstiegshürden verbunden. So ist das Wissen darum, was genau passiert und das Verwenden unterschiedlicher Programmiersprachen und Beschreibungstechniken nötig. Es können durchaus Software-Konfigurationen durch Chef-Recipes oder Puppet-Code geschrieben werden. Hierfür werden keine unterschiedlichen Programmiersprachen benötigt, jedoch sind für Monitoring, Load-Balancer-Konfiguration oder automatische Skalierung weitere Programmiersprachen und Werkzeuge nötig [Art18][S.1].

Es ist anzunehmen, dass sich diese Einstiegshürden mit der Zeit verringern. So benötigt z.B. CloudFormation nur noch die Kenntnis von JSON und YAML. Dies ist ein Indiz dafür, dass Guerriero bei der Aussage, dass die IaC-Technologien noch nicht ausgereift sind und weiterer Aufmerksamkeit bedürfen [Gue19][S.10], Recht hat. Ein weiterer Hinweis hierauf ist, dass aktuelle IaC-Werkzeuge wie CloudFormation anscheinend nicht den Bedürfnissen eines

13 Kapitel: 1. What is DevOps and Should You Care?
14 Kapitel: 1. What is DevOps and Should You Care?

Teils der Nutzer/innen gerecht werden. Dies ist daran zu erkennen, dass es einige Projekte gibt, welche CloudFormation-Templates mittels anderer Programmiersprachen erzeugen, da CloudFormation nicht die gewünschte Funktionalität bietet.

Rahman [Rah19][S.3-5] konnte feststellen, dass IaC einige Sicherheitsrisiken durch falsche Implementierung birgt. Hierzu zählen unter anderem: Leere Strings als Passwörter, fest codierte Passwörter, falsche IP-Adressen, HTTP ohne TLS und schwache Verschlüsselungsalgorithmen. Rahman zeigt weiter, dass Fehler, welche aus der Software-Entwicklung bekannt sind, ebenfalls bei der IaC-Entwicklung auftreten. Dies ist kaum verwunderlich, da gerade das Ziel von IaC ist, Infrastruktur wie Quell-Code zu behandeln.

Im Gegensatz zur Software-Entwicklung ist die Auswahl an Werkzeugen für IaC, um bekannte Sicherheitslücken zu schließen oder einfache Fehler zu verhindern, gering. Auch die Funktionalität dieser Werkzeuge ist oft nicht ausreichend. Um dennoch IaC mit möglichst geringen Sicherheitsrisiken und erfolgreich einsetzen zu können, werden bekannte Lösungen aus der Software-Entwicklung verwendet. Hierzu zählen Code-Reviews, Linting-Werkzeuge, Infrastrukturtests, Fehlerverfolgungssysteme, Code-Pattern und Design-Regeln [Sta17][S.32].

2.6 DevOps

Der Begriff DevOps bezeichnet für die Vereinigung der zwei Disziplinen Software-Entwicklung und Betrieb von Software (engl. Development and Operations) [Art18][S.1]. Die Vereinigung der Namen steht ebenfalls für die Vereinigung von zuvor unabhängigen Teams zu einem Team. Diese Vereinigung ist naheliegend, da immer mehr Praktiken und Konzepte von der Software-Entwicklung in den Betrieb einfließen [Art17][S.1].

DevOps kann als eine Erweiterung der Agilen-Entwicklungsmethoden betrachtet werden, da die Zuständigkeiten einzelner Teams zusammengelegt werden, um geteilte Verantwortung und Kooperation zu erzielen. So ist z.B. das Software-Release nicht mehr Aufgabe des Betriebs, sondern des DevOps-Teams. Dies bedeutet, dass auch das Team, welches zuvor für die Software-Entwicklung zuständig war, nun als Teil des DevOps-Teams verantwortlich ist, das Software-Release erfolgreich durchzuführen. Dies ist nur möglich, wenn beide zuvor unabhängigen Teams zu einem Team verschmelzen und miteinander kommunizieren. Dafür ist es nötig, dass lange interdisziplinäre Prozesse verschlankt und extrem beschleunigt werden, die gleichen Werkzeuge verwendet werden und das Selbstverständnis als DevOps-Team erzeugt wird.[Kan17][15]

Durch DevOps können Unternehmen 30-mal öfter Software veröffentlichen und dies mit 50% weniger Fehlern bei neuen Software-Releases [Mis18][16]. Diese Häufigkeit an Software-Releases ist jedoch nur durch Automatisierung und effiziente Kommunikation innerhalb des DevOps-Teams möglich [Son15][S.3].

In der Literatur ist oft zu finden, dass DevOps aus Continuous-Integration, Continuous-Delivery und Continuous-Deployment besteht [Mis18][17], manchmal wird noch IaC hinzugefügt [Gue19][S.2]. Hiervon darf und sollte man sich nicht verwirren lassen. DevOps nutzt diese Werkzeuge intensiv, um Automatisierung zu betreiben und im Falle von IaC einfache Änderungen der Infrastruktur zu ermöglichen. DevOps ist jedoch mehr.

Abbildung 2.8 zeigt den DevOps-Prozess. Dieser geht iterativ über die Prozessschritte: Planung, Code entwickeln, Bauen, Testen, Erstellung von Software-Releases, Veröffentlichung,

15 Kapitel: 1 What is DevOps and Should You Care?
16 Kapitel: An overview of DevOps
17 Kapitel: An overview of DevOps

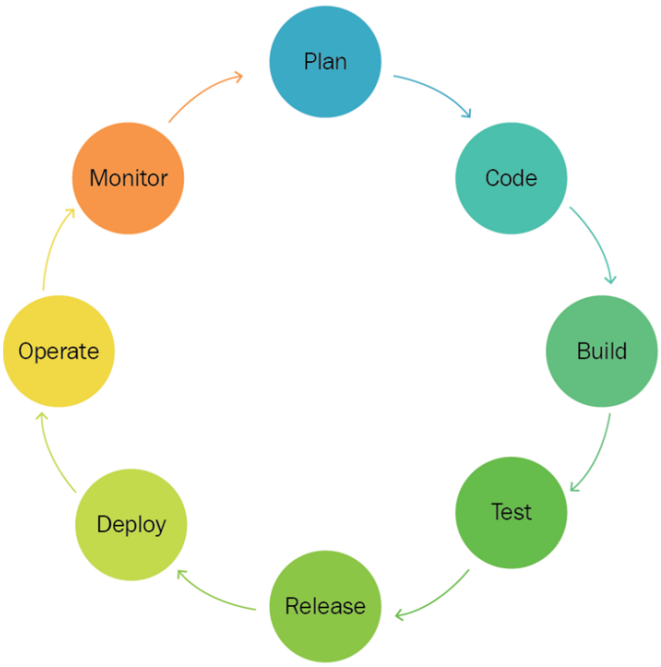


Abbildung 2.8 DevOps-Prozess [Mis18][Kapitel: The benefits of DevOps]

Betrieb und Überwachung. Hieran ist zu sehen, dass DevOps die zwei Disziplinen Software-Entwicklung und Betrieb abdeckt, was natürlich mehr ist, als mit CI/CD und IaC möglich ist. An dieser Stelle wird jedoch nicht weiter auf die Prozessschritte eingegangen, da sich diese Arbeit nur mit den DevOps Werkzeugen CI/CD und IaC beschäftigt, welche in den Prozessschritten Code bis Veröffentlichung Verwendung finden.

Die Ziele, welche DevOps verfolgt, beruhen zu großen Teilen auf denen von CI/CD und beinhalten folgendes [Mis18][18]:

- Häufigere Veröffentlichungen mit besserer Qualität
- Weniger Fehler bei neuen Software-Releases
- Geringer Dauer für die Veröffentlichung von Bugfixes
- Geringere Ausfallzeit im Fehlerfall bzw. bis zur Wiederherstellung eines Systems
- Schnellere Veröffentlichung von Software (Time-to-Market)

Diese Ziele decken sich mit der folgenden Reihe an Vorteilen, welche auf der Synergie bzw. den Vorteilen der verwendeten Werkzeuge CI/CD und IaC beruhen [Mis18][19] [Las20] [Son15][S.5]:

Verbesserte Zusammenarbeit Software-Entwicklung und Betrieb müssen zusammenarbeiten, die Verantwortung teilen und zum DevOps-Team verschmelzen. Dieses DevOps-Team ist effektiver durch geteilte Kompetenzen, kürzere Kommunikationswege, schlankere Prozesse und verringerte Übergabezeiten.

18 Kapitel: The goal of DevOps
19 Kapitel: The benefits of DevOps

Einfachere Skalierung und Reproduktion Durch Automatisierung und IaC ist es mit geringem Risiko möglich und effektiv, komplexe Systeme zu verwalten, zu ändern, zu reproduzieren und die Kapazität anzupassen.

Erhöhte Geschwindigkeit Durch die Automatisierung und den schlanken DevOps-Prozessen ist es möglich, die Time-to-Market zu verringern. Dies bezieht sich auch auf die Zeit, welche zwischen zwei Software-Releases liegt. Dies verringert das Risiko von Software-Releases.

Zuverlässigkeit Durch die Automatisierung werden menschliche Fehler vermieden. Zusätzlich trägt das kontinuierliches Testen zur Qualität der Software bei.

Zeitersparnis Durch die Automatisierung wird Zeitersparnis dadurch erreicht, dass das DevOps-Team den Automatisierungsprozess implementiert und verwaltet, anstatt diese automatisierten Aufgaben manuell auszuführen.

Erhöhte Erfolgswahrscheinlichkeit Durch das häufigere Software-Release ist es möglich, den Zyklus, in dem Endnutzer Rückmeldung über die Software und neue Funktionalitäten geben, zu beschleunigen. Dadurch ist es möglich bessere Qualität und das vom Markt Benötigte zu entwickeln. Dies erhöht die Erfolgswahrscheinlichkeit der Software.

Kontinuierliche Innovationen Durch den kurzen Rückmeldezyklus von Endnutzer und die häufigen Veröffentlichungen ist es möglich, neue Ideen zu testen und gegebenenfalls zu verwerfen.

3 Cloud-Computing-Grundlagen

Dieses Kapitel legt einige Grundlagen um das Thema Cloud-Computing dar. Schwerpunkt hierbei ist das Thema Serverless. Die behandelten Grundlagen treffen vorwiegend auf AWS, jedoch auch auf andere Cloud-Provider zu.

3.1 Einführung in Cloud-Computing

Cloud-Computing oder oft abgekürzt „die Cloud“ ist im Grunde keine neue Erfindung oder Technologie. Bei der Cloud handelt es sich, einfach ausgedrückt, um ein Netzwerk aus vielen Rechenzentren. Was die Cloud so erfolgreich macht, ist der hohe Grad an Virtualisierung bzw. moderne Virtualisierungstechnologien. Diese ermöglichen es, Cloud-Ressourcen zu nutzen ohne langwierige und umständliche Einrichtung von Servern, virtuellen Laufwerken oder Diensten.

Cloud-Provider bieten nicht nur Rechenleistung oder Speicherplatz an, auch wenn dies zum Angebot der Cloud-Provider gehört. Es werden Speicher, Rechenleistung, virtuelle Maschinen, Compliance und viele weitere, teils serverlose, Dienste angeboten.

Die Preisgestaltung in der Cloud unterscheidet sich stark von traditionellen Modellen. Traditionell wird Infrastruktur vorab bezahlt bzw. finanziert und in einem Unternehmen aufgebaut und betrieben. Dies schließt auch Infrastruktur ein, welche nicht oder nur teilweise benötigt wird [Dab20][1].

Cloud-Provider wie AWS bezeichnen ihre Preisgestaltung als **pay-as-you-go**. Unter **pay-as-you-go** ist zu verstehen, dass nur die Ressourcen, welche genutzt werden, bezahlt werden. Z.B. wird ein TiB Speicher lediglich eine Woche belegt, so wird nur dies abgerechnet. Oder wenn eine virtuelle Maschine 2,5 Stunden benötigt wird und danach abgeschaltet wird, berechnen Cloud-Provider sekundengenau die Nutzung und stellen diese in Rechnung. Auch wenn AWS seine Preisgestaltung als **pay-as-you-go** bewirbt, kommen auch noch andere Preismodelle zum Einsatz. Das klassische Mietmodel ist in Form von reservierten Instanzen zu finden und auch Modelle, welche für jede angefangene Minute u.ä. aufrunden. Es ist auch in der Cloud möglich, dedizierte Rechner in verschiedenen Abstufungen zu mieten.

Vereinfacht zeigt Abbildung 3.1, welche finanziellen Möglichkeiten **pay-as-you-go** bietet. So werden Kosten linear durch die Skalierung erzeugt. Bei traditioneller Infrastruktur entstehen sprungfixe Kosten bereits bei der Anschaffung, unabhängig von der Nutzung.

pay-as-you-go bzw. Cloud-Computing bietet die Möglichkeit einer Kostenersparnis, es ist aber nicht automatisch gegeben, dass sich die Infrastrukturkosten durch einen Wechsel in die Cloud verringern. Um eine Kostenersparnis zu erhalten, ist die Anwendung von DevOps und eine passende Software-Architektur zu empfehlen.

Villamizar zeigt, wie die Kostenersparnis von der jeweiligen Architektur abhängig ist. So liegt die Kostenersparnis eines Microservices gegenüber einem Monolithen zwischen

1 Kapitel: 1. Full Stack Development in the Era of Serverless Computing]

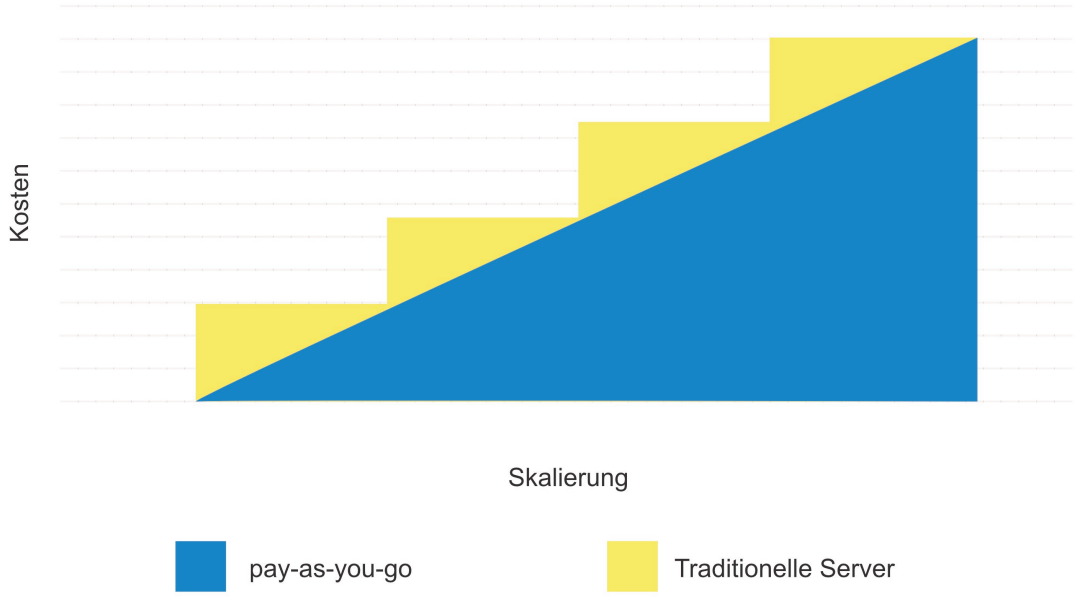


Abbildung 3.1 pay-as-you-go vs. Traditionelle Server Kosten

9,50 % und 13,42%. Mit dem AWS-Dienst Lambda konnten gegenüber einer Mircoservice-Architektur zwischen 50,43% und 57,01% eingespart werden. Lambda konnte gegenüber einem Monolithen sogar zwischen 55,14% und 62,78% einsparen [Vil16][S.4].

Vorteile, welche die Cloud nicht von selbst bietet, sondern solche, die in der Kombination von DevOps entstehen, sind die nochmalige Beschleunigung von Innovationszyklen und eine kürzere Time-to-Market [Wei17][S.3]. Diese Vorteile werden bereits durch DevOps erreicht, jedoch durch die Synergie der Cloud noch einmal gesteigert.

Weitere Vorteile sind Ausfallsicherheit und Wiederherstellbarkeit, sowie Elastizität und Skalierung. Gerade Elastizität und Skalierung stellen eine große Innovation im Vergleich zu traditionellen Servern dar. Traditionell werden Server gekauft und provisioniert, erst dann stehen diese zur Verfügung. Somit ist die Erhöhung der Serverkapazität nur durch den Erwerb und die Integration neuer Hardware möglich. Das Verkleinern von Serverkapazität ist nur durch das Entfernen bzw. Abschalten von Hardware möglich.

Durch Cloud-Provider und Virtualisierung ist es möglich, Kapazitäten sowohl manuell als auch automatisiert in eine bestehende Umgebung zu integrieren und zu entfernen, quasi ohne Zeitverzögerung und per „Knopfdruck“.

3.2 Serverless

Der Begriff Serverless ist weit verbreitet, jedoch irreführend. Es ist nicht möglich Rechenleistung ohne einen Server bzw. Hardware zu erzielen. Jedoch bedeutet Serverless in diesem Zusammenhang, dass es zwar einen Server gibt, dieser aber nicht selbst verwaltet wird [Eiv17][S.2]. Die Verwaltung der Server geschieht komplett durch den Cloud-Provider.

Es eignet sich nicht jede Software für den Serverless-Ansatz. Die moderne Microservice-Architektur ist wie geschaffen für die Serverless-Dienste der Cloud-Provider. So können z.B. Microservices in einer VM oder einem Container im Grunde ohne Down-Time veröffentlicht

bzw. erneuert werden [Sin19][S.1].

Der Serverless-Ansatz beinhaltet nach einer Umfrage noch weitere Vorteile. So gaben die Befragten folgende Erwartungen an [Mag19]:

60% Reduzierung der operativen Kosten

58% Automatische Skalierung nach Bedarf

55% Wegfallen der Server-Wartung

32% Reduzierung der Entwicklungskosten

30% Erhöhung der Entwicklungsproduktivität

Aber nicht nur der Bereich der Software-Entwicklung profitiert vom Serverless-Ansatz. Auch der CI/CD-Prozess profitiert hiervon, wodurch die Produktivität des DevOps-Teams positiv beeinflusst wird [Zam18][2]. Ein oft auftretender Trugschluss ist die Annahme, dass durch die Serverless-Nutzung auf Betriebspersonal verzichtet werden kann und dadurch operative Kosten eingespart werden. Dies trifft zum Teil zu, jedoch ist der Betrieb weiterhin nötig, aber in anderer Form. Auf das Betriebs-Team kann nicht verzichtet werden, jedoch kann es reduziert werden bzw. mit dem Entwicklungs-Team zu einem DevOps-Team verschmolzen werden. Das nun entstandene DevOps-Team verursacht durch den Serverless-Ansatz weniger operative Kosten und kann sich auf andere Aufgaben konzentrieren [Dab20][3].

Mistry fasst die Vorteile des Serverless-Ansatzes wie folgt zusammen [Mis18][4]:

Keine Administration Es ist möglich, Code ohne Provisionierung und Verwaltung zu veröffentlichen. Auch die Konzepte von Instanzen, Container-Flotten und Betriebssystemen müssen nicht beachtet werden. Und es gibt keine Notwendigkeit für eine Betriebsabteilung. (Dieser Punkt wurde bereits geklärt. Der Betrieb wird weiterhin benötigt, jedoch in Form des DevOps-Teams)

Automatische Skalierung Automatisierte ausgelöste Skripte ermöglichen es, abhängig von Metriken oder Zeiten, zusätzliche Ressourcen zur Verfügung zu stellen. Dies wird von Cloud-Providern durch Skalierungsrichtlinien verwaltet.

pay-as-you-go Es werden nur verwendete Ressourcen berechnet. Dies bezieht sich i.d.R. auf Rechenleistung, Arbeitsspeicher und Datentransfer. Dies kann auch bedeuten, falls der Code nicht verwendet wird, dass keine Kosten entstehen.

Erhöhte Geschwindigkeit Die Planungsdauer, Veröffentlichungsdauer und die Provisionierung von Betriebsinstanzen kann drastisch reduziert werden. Fragen bezüglich Infrastruktur bei der Verwendung des Serverless-Ansatzes entfallen. Und es fallen auch keine zeitlichen Aufwände für Bereitstellung und Verwaltung anfallen.

Diese Vorteile können alle durch den Serverless-Ansatz erreicht werden, jedoch hängen die wirtschaftlichen Vorteile stark vom Laufzeitverhalten und vom Arbeitsumfang der Software ab. Es kann auch für geeigneter Software-Architekturen der Fall eintreten, dass die Wirtschaftlichkeit nicht gegeben ist und traditionelle Lösungen die bessere Wahl sind [Eiv17][S.1].

2 Kapitel: Setting up CI with CircleCI
3 Kapitel: 1. Full Stack Development in the Era of Serverless Computing
4 Kapitel: Serverless applications benefits

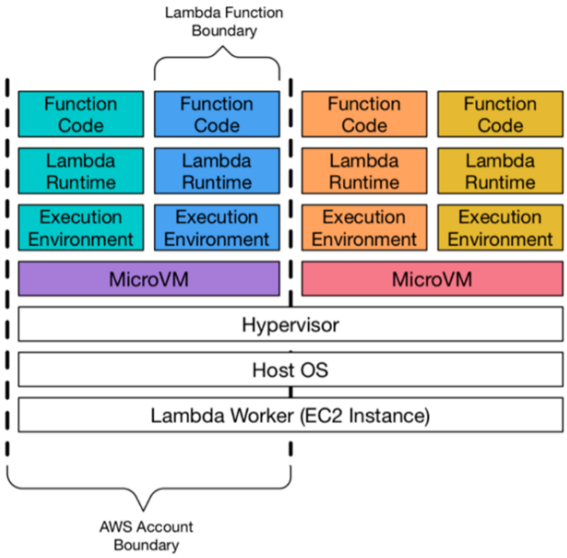


Abbildung 3.2 Lambda: Isolationsmodel [Tha19][S.5]

3.3 Verwendete AWS-Dienste

NTT-Serverless-CI/CD nutzt mehrere AWS-Dienste. In diesem Kapitel werden ausgewählte AWS-Dienste näher beschrieben.

3.3.1 Lambda

Bei Lambda handelt es sich um einen Dienst, welcher Code auf Bedarf ausführt und für das Ausführen die Verwaltung und Bereitstellung von Recheninstanzen übernimmt. Dieser Code kann in unterschiedlichen Programmiersprachen wie: Java, Python, JavaScript, C# u.a. geschrieben werden. Dies ermöglicht die Entwicklung moderner Serverless-Architekturen mit Fokus auf Funktionalität anstatt auf die Infrastruktur der Software. Nach Mistry ist hierdurch eine Kostenreduktion von bis zu 90% möglich [Mis18][5].

Abbildung 3.2 zeigt die Architektur von Lambda. Wie bei anderer Virtualisierung ist auf einem Host-System (in diesem Fall Lambda Worker bzw. EC2) ein Betriebssystem. Dieses Betriebssystem betreibt einen Hypervisor, welcher mehrere MicroVMs bedient. Diese MicroVMs sind auf ein AWS-Konto beschränkt, werden aber innerhalb dieses AWS-Kontos wiederverwendet. Die Ausführungsumgebungen werden wiederverwendet, wenn eine Funktion wiederholt ausgelöst wird. Dies ist jedoch nicht für andere Funktionen möglich. Die Lambda-Laufzeit wird nicht wiederverwendet, sie wird lediglich für das Ausführen einer Lambda-Funktion erzeugt und anschließend zerstört [Tha19][S.5].

Der Lambda-Dienst ist in eine Kontrolleinheit und eine Dateneinheit geteilt. Die Kontrolleinheit kontrolliert und verwaltet die APIs *CreateFunction* und *UpdateFunctionCode* und übernimmt die Interaktion mit anderen AWS-Diensten. Die Dateneinheit reagiert auf die

5 Kapitel: Serverless applications benefits

Invoke-API, welche Lambda-Funktionen auslöst. Für eine ausgelöste Funktion erzeugt die Dateneinheit eine Ausführungsumgebung oder verwendet eine bestehende Ausführungsumgebung wieder [Tha19][S.5].

Der Lambda-Dienst wurde durch das Ausführen von Code, ohne Aspekte wie Server oder Container zu berücksichtigen, zum populärsten AWS-Dienst für Serverless-Architekturen [Pur17][S.1].

3.3.2 CI/CD-Dienste

AWS bietet, um den CI/CD-Prozess abzubilden, eigene Dienste und Werkzeuge an. Es kann mittels dieser Dienste der CI/CD-Prozess ohne zusätzliche Frameworks und Werkzeuge abgebildet werden. Dieser Abschnitt deckt nicht alle Möglichkeiten ab, welche AWS für CI/CD bietet, jedoch werden diejenigen, welche Berührungspunkte mit NTT-Serverless-CI/CD haben aufgezeigt. Bei den aufgezeigten Diensten handelt es sich um von AWS verwaltete Serverless-Dienste, welche keine Wartung benötigten und eine redundante und hoch skalierbare Architektur aufweisen. Dadurch ist es bei Verwendung nicht nötig, sich über Parallelität oder Verwaltung zu kümmern.

CodeCommit Bei CodeCommit handelt es sich um einen SCM-Dienst. Es ist ein hoch skalierbares Werkzeug, welches keinen Speicherrestriktion unterworfen ist. Er nutzt das Rollen- und Rechtekonzepte von AWS und verschlüsselt standardmäßig die Übertragung von Daten [Mis18][6]. CodeCommit basiert auf Git und kann mit privaten Repositories auf GitHub verglichen werden, welche in AWS betrieben werden. CodeCommit wird mit existierenden Git-Werkzeugen betrieben und wie jede andere Git-Software integriert.

CodePipeline CodePipeline erzeugt die CI/CD-Pipeline in AWS. CodePipeline entspricht jedoch keiner kompletten CI/CD-Pipeline. CodePipeline ist als Dienst zu verstehen, welcher die Stages orchestriert. Z.B. wird die Ausführung des Builds an CodeBuild oder das Veröffentlichen an CodeDeploy ausgelagert. Neben der Orchestrierung der Stages und der zuständigen Dienste können durch CodePipeline Benachrichtigungen ausgelöst werden. Weiter bietet es eine grafische Oberfläche für die CI/CD-Pipeline und persistiert Zustände zwischen den Stages. Dieses Persistieren ermöglicht es einzelne Stages, sollten diese scheitern, zu wiederholen anstatt die komplette CI/CD-Pipeline von Anfang zu starten.

CodeBuild CodeBuild ist wie der Build-Dienst von AWS. Er wurde als ein hoch verfügbarer Build-Prozess mit fast unendlicher Skalierung konzipiert [Sta17][S.22]. Hierfür stellt AWS Maschinen zu Verfügung, welche mit Docker-Images als Build-Umgebung verwenden. AWS stellt hierfür verschiedene Docker-Images zur Verfügung. Es können aber auch selbst entwickelte Docker-Images verwendet werden. In den Docker-Containern werden, wie in anderen Build-Umgebungen auch Installationen, Tests und Kompilierung durchgeführt sowie Artefakte erzeugt und transferiert. Die hierfür benötigte Konfiguration findet entweder über die AWS-Managementkonsole statt oder kann als Datei im YAML-Format verwendet werden.

6 Kapitel: AWS CodeCommit – maintaining code repository

AWS bietet jedoch nur Docker-Images für folgende Build-Umgebungen an: .Net, Golang, NodeJS, Java, PHP, Powershell, Python, Ruby und Android[7]. Andere Build-Umgebungen können jedoch durch selbst erzeugte bzw. öffentliche Docker-Images verwendet werden. So ist z.B. auf Dockerhub eine Docker-Image für Swift verfügbar. Umgebungen, welche auf MacOS basieren, wie XCode können aktuell nicht verwendet werden, da auf AWS noch keine Maschinen für CodeBuild mit MacOS angeboten werden. Dies könnte sich jedoch ändern, da bereits in einigen Regionen EC2-Mac-Instanzen angeboten werden.

CodeDeploy CodeDeploy orchestriert die Veröffentlichung. Es ermöglicht die Komplexität von Veröffentlichungen zu handhaben, schnell neue Funktionalitäten zu veröffentlichen und Down-Time gering zu halten bis hin zu verhindern [Mis18][8]. Dabei kann CodeDeploy alle Veröffentlichungsstrategien, wie in Kapitel 2.3 beschrieben, nutzen. CodeDeploy ermöglicht das Veröffentlichen in der AWS-Cloud z.B. EC2-Instanzen, ECS und Lambda, als auch außerhalb von AWS auf dedizierten Servern und durch andere Cloud-Provider.

Neben den reinen CI/CD-Diensten bietet AWS noch einige weitere Werkzeuge und Dienste an, welche für DevOps genutzt werden können. Diese sind z.B. CodeStar, X-Ray und Cloud9. Ganzheitlich eingesetzt ist es in AWS möglich, DevOps zu betreiben und die bereits bekannten Vorteile zu erzielen. [Mis18][9]

3.3.3 CloudFormation

Der IaC-Dienst von AWS wird CloudFormation genannt. Bei CloudFormation handelt es sich um einen Dienst, welcher durch sogenannte „Stacks“ automatisiert Infrastruktur erstellt, ändert und zerstört. Stacks dienen dabei als Repräsentation von einer beliebigen Kombination bzw. Sammlung an AWS-Ressourcen. Diese Stacks können auf andere Stacks referenzieren (Cross-Stack-Reference) oder andere Stacks beinhalten (Nested-Stacks). Dies ermöglicht die Wiederverwendbarkeit und unabhängige Entwicklung von Stacks.

Stacks werden durch CloudFormation-Templates beschrieben. Hierbei handelt es sich um Dateien im JSON- oder YAML-Format. Bei diesen Formaten handelt es sich um statische Beschreibungssprachen, welche durch CloudFormation nur bedingt dynamische und programmatische Funktionen erhalten [Che17][S.4ff]. Dennoch ist es möglich durch Konfigurationsverwaltung und der Integration anderer AWS-Dienste dynamisch unterschiedliche VPCs, Regionen, Umgebungen und AWS-Konten durch CloudFormation zu unterstützen.

Die Änderungsfunktionalität von CloudFormation ermöglicht es, falls die verwendete AWS-Ressource es ermöglicht, Änderungen ohne neue Erzeugung dieser AWS-Ressource durchzuführen. So kann z.B. einer IAM-Rolle eine IAM-Richtlinie hinzugefügt werden, ohne diese neu erzeugen zu müssen. Dies ist jedoch nicht für jede AWS-Ressource in jeder Änderung möglich. Z.B. muss für die Umbenennung einer IAM-Rolle die Ressource zerstört und neu erzeugt werden.

CloudFormation bietet noch eine Reihe weiterer Funktionalitäten, einführend werden nur ein paar davon hier erwähnt:

7 `https://docs.aws.amazon.com/codebuild/latest/userguide/build-env-ref-available.html` Stand 13.11.2020
8 Kapitel: AWS CodeDeploy
9 Kapitel: A brief overview of AWS for DevOps

Change-Sets CloudFormation ermöglicht die Verwendung von Change-Sets. Hierbei handelt es sich um Vorversionen der Änderungen. Dadurch kann eine Änderung überprüft und die Konsequenz der Änderung bedacht werden, bevor sie umgesetzt wird. Über die grafische Oberfläche ist die Änderung vor dem Bestätigen einer Änderung ebenfalls sehr gut zu sehen.

Automatisches Zurückrollen Die Funktionalität des Zurückrollens wird bei CloudFormation während der Erzeugung und Änderungen verwendet, falls es nicht möglich ist, AWS-Ressourcen auf die gewünschte Weise zu erzeugen.

Das Löschen eines Stacks ist davon nicht betroffen. Treten während des Entfernens eines Stacks Probleme auf, werden nur die AWS-Ressourcen entfernt, welche keine Probleme erzeugen. Hieraus folgt, dass kein Zurückrollen stattfindet und auch ein inkonsistenter Zustand der Infrastruktur entstehen kann, welcher manuell behoben werden muss.

AWS-Ressourcen werden in CloudFormation-Templates durch eine Meta-Sprache beschrieben. Dies ist daran zu erkennen, dass in den CloudFormation-Templates auf einer hohen Abstraktionsebene beschrieben wird, welche AWS-Ressourcen erzeugt werden sollen. Daraus folgt, dass CloudFormation die Reihenfolge, in der die AWS-Ressourcen erzeugt werden, unter der Berücksichtigung von Abhängigkeiten und Optimierung, selbst bestimmt.

Durch diese Meta-Sprache können CloudFormation-Templates als lebende Dokumentation betrachtet werden, welche alles Wissen über die Infrastruktur enthält. Diese lebenden ausführbaren Dokumente sind als essentielle Werkzeuge der Entwicklung bzw. des DevOps zu betrachten [Spi12][S.2].

Wie bereits in Kapitel 2.5 beschrieben, bezeichnet Guerriero die IaC-Werkzeuge als noch nicht ganz ausgereift [Gue19][S.10]. Hierauf deuten auch die Werkzeuge hin, welche durch unterschiedliche Programmiersprachen CloudFormation-Templates generieren. Diese Projekte zeigen, an welche Grenzen CloudFormation stößt. Stattdessen werden Programmiersprachen verwendet, welche von Funktionalität und Dynamik flexibler sind. So wurde z.B. 2013 bereits Troposphere[10] veröffentlicht, welches es ermöglicht CloudFormation-Templates mit Python zu erzeugen. Ein weiteres Werkzeug, welches 2017 von den AWS Labs veröffentlich wurde und Golang nutzt, ist GoFormation[11]. Anschließend veröffentlichte AWS ein komplettes Framework, welches die volle IaC-Erfahrung mit verschiedenen Programmiersprachen bieten soll. Das Framework unterstützt dabei folgende Programmiersprachen: TypeScript, JavaScript, Python, C# und Java [Tov20][S.202].

Dennoch ist CloudFormation ein Werkzeug, welches seine Berechtigung hat und auch sehr gut zu verwenden ist, jedoch mit Abstrichen. Um die Entwicklung mit zu vereinfachen, gibt es einige Linting-Werkzeuge, so z.B. der auf Python basierenden AWS CloudFormation Linter (kurz cfn-lint)[12].

Für CloudFormation beschreibt Cheung einige Best-Practices, welche sich auf die drei Bereiche Planung & Organisation, Erstellung und Verwaltung aufteilen: [Che17][S.8-9]

- Planung & Organisation
  - Organisiere Stacks nach Lebenszyklus und Eigentümer
  - Verwende IAM für Berechtigungen

10 `https://github.com/cloudtools/troposphere`
11 `https://github.com/awslabs/goformation`
12 `https://github.com/aws-cloudformation/cfn-python-lint`

  - Verwende CloudFormation-Templates für mehrere Umgebungen wieder
  - Verwende Nested-Stacks für Standard-Infrastruktur und wiederkehrende Stacks
  - Verwende Cross-Stack-Referenzen für geteilte Ressourcen
- CloudFormation-Template erstellen
  - Nie Zugangsinformationen hart codieren
  - Wenn möglich AWS-Parameter-Typen verwenden
  - Parameter Constraints verwenden
  - AWS::CloudFormation::Init für EC2-Instanzen verwenden
  - Validiere CloudFormation-Templates, bevor sie verwendet werden (z.B. cfn-lint)
  - Nutze eine zentrale Parameterverwaltung (z.B. AWS SystemsManager Parameter Store)
- Stacks verwalten
  - Nutze Change-Sets vor Änderungen
  - Nutze Stack-Richtlinien
  - Verwende CloudTrail für Governance und Logging von CloudFormation
  - Nutze SCM und Code-Reviews für CloudFormation-Templates

3.4 IT-Sicherheit in AWS

IT-Sicherheit ist aktuell ein großes Thema und einer der großen Bedenken, was den Einstieg bzw. Umstieg von traditionellen IT-Lösungen in die Cloud betrifft. Wie wichtig IT-Sicherheit ist, kann an der Steigerung der Ausgaben für IT-Sicherheit in Deutschland abgelesen werden. 2017 betragen die Ausgaben 3,7 Milliarden Euro. 2021 betragen die Kosten prognostiziert 5,7 Milliarden Euro. Dies stellt eine Steigerung von ca. 65% da [Bit20].

Auch die große Anzahl und neuer Schadprogramm-Varianten von 117,4 Millionen, welche vom 01.06.2019 bis 31.05.2020 registriert wurde, lässt darauf schließen, wie wichtig es ist eine gute und aktuelle IT-Sicherheit zu unterhalten [BSI20].

Die hohen Ausgaben und die Anzahl an Schadprogrammen bestätigen die Umfrage von Bitkom, nach welcher IT-Sicherheit als wichtigstes Thema mit 67% noch vor Cloud-Computing mit 61% gewählt wurde [Bit18].

Die IT-Sicherheit muss im Cloud Umfeld einige Aspekte berücksichtigen und abdecken, welche es bei traditionellen Server-Lösungen nicht gibt [Wei17][S.2]. Des Weiteren ist es bei der Verwendung von Cloud-Infrastruktur oder Cloud-Services gar nicht möglich, die Hardware oder die zu Grunde liegenden Systeme durch die eigene IT-Sicherheit zu schützen. Um die Verantwortlichkeiten für die IT-Sicherheit klar abzugrenzen, hat AWS ein Verantwortungsmodell (engl. Responsibility Model) entworfen. Dieses Verantwortungsmodell, welches in verschiedenen Varianten für verschiedene Bereiche in AWS existiert, zeigt die Verantwortungsbereiche für den Nutzer und AWS auf.

Abbildung 3.3 zeigt die grundlegenden Sicherheitsverantwortungsbereiche von AWS und vom Nutzer. So ist AWS für die Cloud selbst zuständig. Dies umfasst einfach ausgedrückt den physischen Teil der Cloud. Dazu zählt die globale AWS-Infrastruktur (Regions, Availability Zones, Edge Locations), sowie die Sicherheit von Rechnern, Speicher, Datenbanken und

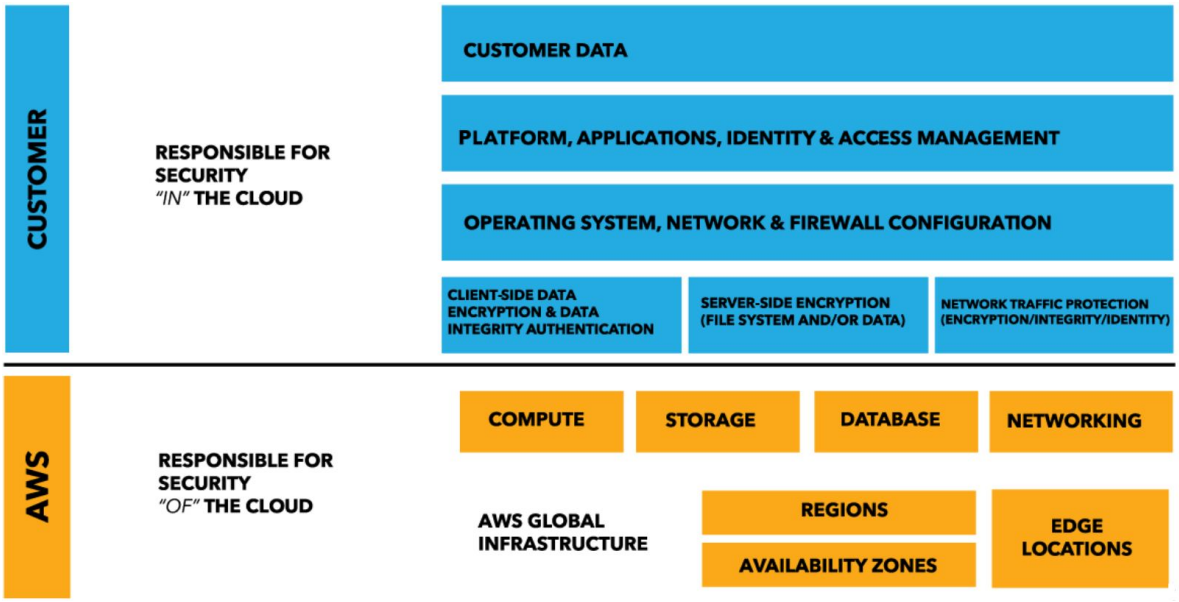


Abbildung 3.3 AWS shared security responsibility model [Ant17][Kapitel: AWS shared security responsibility model]

Netzwerk. Der Nutzer ist für alles verantwortlich was sich innerhalb der Cloud befindet. Dazu zählen Daten, Applikationen, Identifizierungs- & Zugriffsverwaltung, Betriebssysteme, Netzwerk- & Firewall-Einstellungen,Verschlüsselung (Client & Server) und der Netzwerkverkehr.

Da der Nutzer nicht die Möglichkeit hat die IT-Sicherheit der Cloud-Infrastruktur zu überprüfen und AWS aus Sicherheitsgründen nicht offen legt, welche Sicherheitsmaßnahmen ergriffen werden, ist eine unabhängige Sicherheitsüberprüfung nötig. AWS führt regelmäßige Audits durch und hat eine große Menge an Zertifizierungen, Regularien und Vorschriften, welche erfüllt werden. AWS erfüllt folgende Zertifizierungen: ASIP HDS, C5, CMMC, Cyber Essentials Plus, DoD SRG, ENS High, FedRAMP, FIPS, IRAP, ISO 9001, ISO 27001, ISO 27017, ISO 27018, K-ISMS, MTCS Tier 3, OSPAR, PCI DSS Level 1, SEC Rule 17-a-4(f), SOC 1, SOC 2, SOC 3 und TISAX[13]. Schlussfolgernd aus dieser Menge an Zertifizierungen, betreibt AWS erhebliche Sicherheitsmaßnahmen, um die IT-Sicherheit in der AWS-Cloud zu gewährleisten. Unter der Annahme, dass andere Cloud-Provider ähnliche Sicherheitsmaßnahmen ergreifen, ist es nicht verwunderlich, dass nach Anthony 95% aller Sicherheitsvorfälle in der Cloud vom Nutzer und nicht vom Cloud-Provider verursacht werden [Ant18][14].

Das Prinzip hinter dem Verantwortungsmodel ist simpel und verständlich. Der Nutzer ist für das verantwortlich, das er selbst beeinflussen kann, AWS für das, was AWS beeinflussen kann. So ist z.B. der Nutzer nicht in der Lage die Hardware eines Servers zu schützen, was unter den Verantwortungsbereich von AWS fällt. Und AWS ist nicht in der Lage, Sicherheitslücken in der Applikation eines Nutzers zu beheben, folglich fällt dies unter den Verantwortungsbereich des Nutzers.

Das Verantwortungsmodell aus Abbildung 3.3 ist für Serverless-Ansätze nicht anwendbar, da es, was die technischen Schichten betrifft, zu weit unten ansetzt. Serverless-Ansätze

13 `https://aws.amazon.com/compliance/programs/` Stand:19.11.2020
14 Kapitel: 5. AWS Security Best Practices

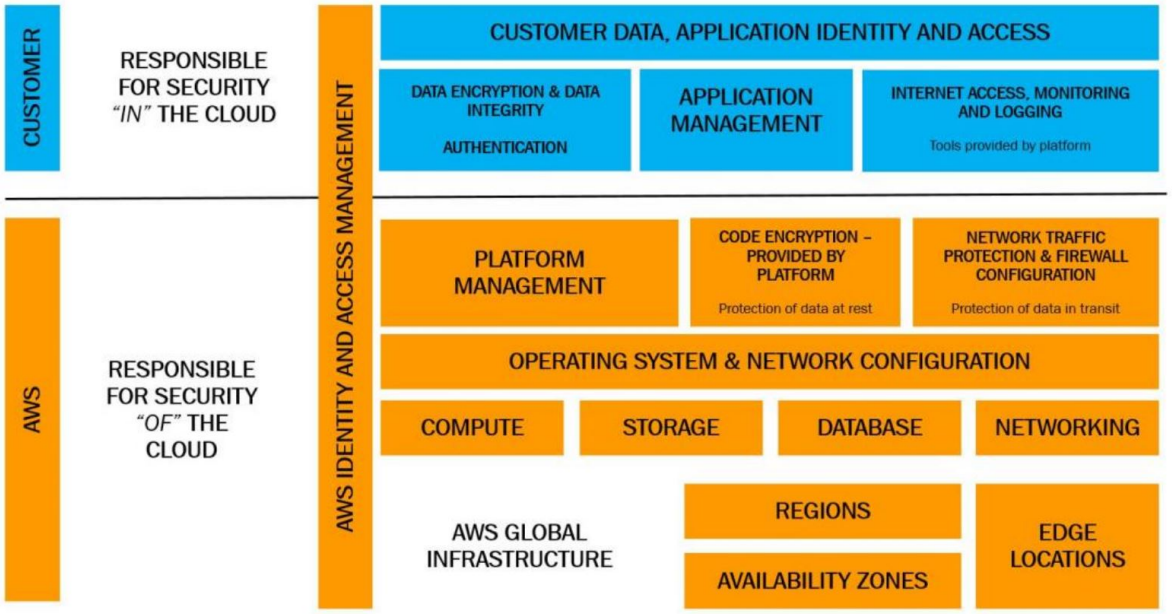


Abbildung 3.4 Lambda Responsibility Model [Tha19][S.4]

nutzen verwaltete Dienste, welche nicht vom Nutzer gesichert werden können und nach dem Prinzip des Verantwortungsmodells in den Verantwortungsbereich von AWS fallen. Hierfür hat AWS zwei weitere Varianten des Verantwortungsmodells entworfen. Abbildung 3.4 zeigt das Verantwortungsmodell für Lambda. Nach diesem ist der Nutzer für die Identifizierungs- und Zugriffsverwaltung, für die Daten und deren Verschlüsselung, für die Applikationsverwaltung bzw. Lambdaverwaltung, für den Internet-Zugang sowie für Logging und Monitoring zuständig. Im Gegensatz zum grundlegenden Verantwortungsmodell erweitert das Verantwortungsmodell für Lambda die Verantwortung von AWS um Betriebssystem, Netzwerkkonfiguration, Schutz des Netzwerkverkehrs, Plattformverwaltung sowie Verschlüsselung des vom Nutzer erstellten Lambda-Codes.

Für den Nutzer bedeutet dies, dass sich der Verantwortungsbereich bei der Verwendung von Lambda auf folgende Punkte ausstreckt:

- Daten, welche den Lambda-Funktionen übergeben werden, müssen ausreichend gesichert und nicht korrumpiert sein.
- Der auszuführende Code darf keine Sicherheitslücken aufweisen.
- Der Zugang, die Ausführung und die Berechtigungen der Lambda-Funktionen müssen angemessen und geregelt sein.
- Der Internet-Zugang der Lambda-Funktionen muss geschützt und nicht willkürlich stattfinden.
- Monitoring und Logging

Damit der Nutzer diesem Verantwortungsbereich gerecht werden kann, bietet AWS diverse Dienste an. So kann für die Verschlüsselung der Daten KMS, die Berechtigungs- und Zugangsverwaltung IAM und für das Monitoring und Logging Cloud-Watch genutzt werden. Für

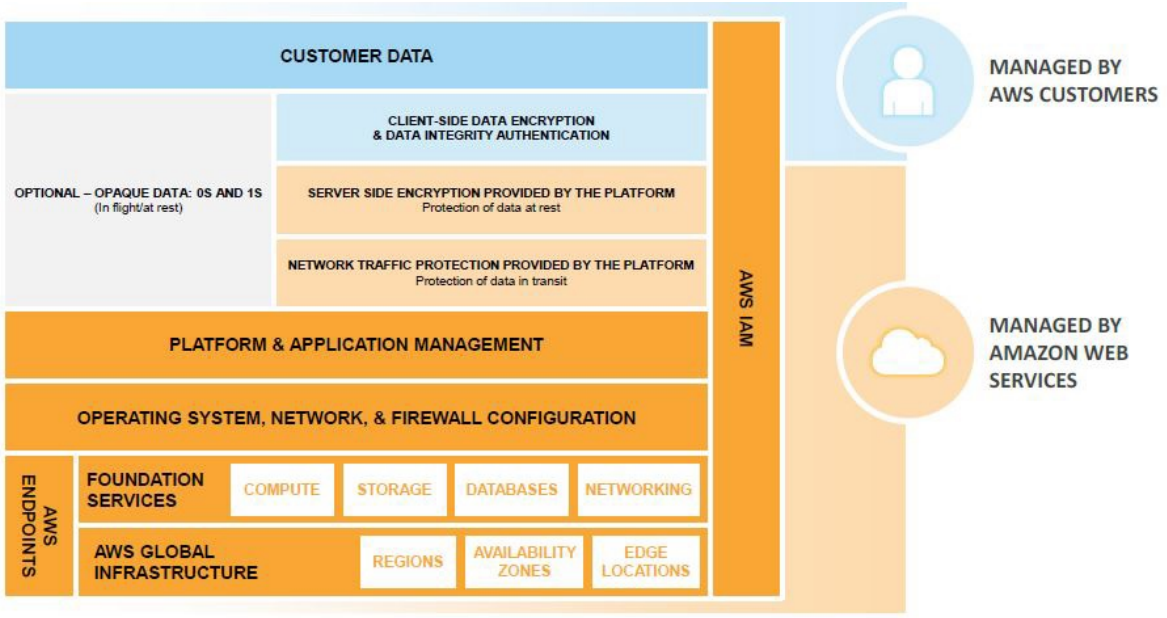


Abbildung 3.5 Shared Responsibility Model for Abstracted Services [Tod16][S.11]

die Qualität des auszuführenden Codes können bekannte Software-Entwicklungsmethoden wie Code-Analyse, Audits und Reviews genutzt werden.

Neben Lambda gibt es noch weitere Serverless Dienste, welche AWS anbietet, z.B. S3, CodePipeline und CodeBuild. Abbildung 3.5 zeigt das Verantwortungsmodell für abstrakte Dienste. Der Abbildung ist zu entnehmen, dass der Nutzer nur noch für die Berechtigung, die Verschlüsselung der Daten und die Integrität der Daten zuständig ist. Diese Aufgaben werden wie bereits erwähnt durch IAM und KMS unterstützt. Im Falle von S3 ist es zusätzlich möglich eine starke und leicht zu nutzende server-seitige Verschlüsselung zu verwenden [Kan17][15]. Weiter kann die Sicherheit mancher Ressourcen noch durch Ressourcen-Richtlinien erhöht werden.

Anthony beschreibt zwei weitere Punkte, welche für Datensicherheit berücksichtigt werden sollten. Das Rotieren der KMS-Schlüssel und das Klassifizieren der Daten, um das angemessene Maß an IT-Sicherheit für die Daten festzulegen [Ant17][16].

Des Weiteren bietet AWS den Dienst Trusted Advisor. Hierbei handelt es sich um einen Dienst, welcher ein AWS-Konto auf Kosten, Leistung, Fehlertoleranz, Service Begrenzungen und Sicherheit prüft. Der Trusted Advisor bietet jedoch keine komplette Sicherheitsüberprüfung des AWS-Kontos, insbesondere nicht von der betriebenen Software. Stattdessen wird auf Best-Practices, Berechtigungen und AWS-Sicherheitsfunktionen geprüft. Sicherheitsmängel können auch von Trusted Advisor übersehen werden und somit ist der Trusted Advisor nur als zusätzliche Überprüfungsmethode zu betrachten, nicht aber als Sicherheitsgarantie.

15 Kapitel: EC2 Security
16 Kapitel: Data security

4 NTT-Serverless-CI/CD

NTT-Serverless-CI/CD hat seinen Ursprung 2019 als Michael Loibl begann mit den CI/CD-Werkzeugen von AWS zu experimentieren. Er sah das große Potenzial, welches AWS für CI/CD birgt, war jedoch mit den Diensten, wie sie angeboten werden, nicht zufrieden.

Gleichzeitig machte Fuad Ibrahimov in Software-Projekten, in welchen er tätig war, wiederholt die Feststellung, dass die wiederholt verwendete CI/CD-Lösung: OpenShift in Kombination mit Jenkins nicht optimal ist. Ibrahimov findet die Verwaltung der Konfigurationen, des Quell-Codes und der Infrastruktur (Terraform-Skripte) in unterschiedlichen SCM-Repositories zu umständlich. Die daraus resultierenden manuellen Schritte für Veröffentlichungen und Synchronisierung sind ebenfalls eine Fehlerquelle und zeitaufwendig. Damit gehen Ad-Hoc Makefiles einher, lokal ausgeführte Skripts, welche Infrastruktur erzeugen und die ungenügende Integration in den Arbeitsablauf. Außerdem ist es aufwändig, Infrastruktur Erweiterungen umzusetzen, wie z.B. einen bestehenden Microservice um eine Lambda-Funktion oder S3-Bucket zu erweitern.

Loibl und Ibrahimov konnten sich bei NTT DATA Deutschland GmbH austauschen und zusammen haben sie das Ziel definiert, ein Framework zu erstellen, welches DevOps in AWS auf eine Weise ermöglicht, dass:

- größtmögliche Automatisierung
- einfache Wartung
- einfache Verwaltung
- ein hohes Maß an Sicherheit

erreicht werden kann. Die größtmögliche Automatisierung soll sich dabei nicht nur auf die CI/CD-Werkzeuge beziehen, welche bereits in einem hohen Grad durch AWS automatisiert sind. Auch die Implementierung der CI/CD-Werkzeuge soll automatisiert stattfinden. So ist die Vision, egal für welche Applikation (groß, klein, Web-Applikation, Microservice, Monolith, ...), durch das Ausführen eines einzigen Befehls die benötigte Infrastruktur für CI/CD inkl. der CI/CD-Pipeline in AWS zu erzeugen. Gleichzeitig soll diese CI/CD-Pipeline und die erzeugte Infrastruktur, einfach zu ändern und zu entfernen sein.

Aus der Vision wurde NTT-Serverless-CI/CD. Dieses Framework stellt verschiedene Funktionalitäten durch in Python geschriebene Lambda-Funktionen zur Verfügung. Neben Lambda verwendet NTT-Serverless-CI/CD einen Satz an CloudFormation-Templates welche in YAML geschrieben sind. Die Vision mit einem Befehl die benötigte Infrastruktur für CI/CD in AWS zu implementieren, konnte mit dem Build-Management-Werkzeug *Make* verwirklicht werden.

An dieser Stelle soll auch erwähnt werden, was NTT-Serveless-CI/CD nicht ist. NTT-Serverless-CI/CD ist nicht das Werkzeug mit dem die CI/CD-Pipeline beschrieben oder erstellt wird. Auch die Erstellung der software-spezifischen Pipeline ist unabhängig von NTT-Serverless-CI/CD. NTT-Serverless-CI/CD erzeugt die nötige Infrastruktur, Berechtigungen, Automatisierung und weitere AWS-Ressourcen, welche benötigt werden, um CI/CD in AWS

umzusetzen. Dies ist auch beabsichtigt, da die Pipeline und das Konfigurationsmanagement nicht Teil des Frameworks sein sollen, sondern der jeweiligen Software zugehörig sind. Dies ermöglicht es NTT-Serveless-CI/CD die allgemeinen Prozesse, welche bei der Software-Entwicklung in AWS für das Einrichten von CI/CD nötig sind, zu automatisieren.

Loibl und Ibrahimov stellten im zweiten Quartal 2020 firmenintern bei NTT DATA Deutschland GmbH den ersten Prototyp des NTT-Serverless-CI/CD-Frameworks vor. NTT-Serverless-CI/CD wurde nach der internen Veröffentlichung für den Enterprise-Einsatz angepasst und erfolgreich in verschiedenen Software-Projekten in der deutschen Automobilindustrie eingesetzt. Noch ist NTT-Serverless-CI/CD nicht öffentlich zugänglich. Es befindet sich jedoch im firmeninternen Prozess der Veröffentlichung und soll voraussichtlich im ersten Halbjahr 2021 auf AWS-Marketplace oder GitHub verfügbar sein.

4.1 Aufgabenteilung und Pipelines

Die Architektur von NTT-Serverless-CI/CD basiert auf der Best-Practice für unterschiedliche Aufgaben unterschiedliche AWS-Konten zu verwenden. Für die Verwaltung dieser unterschiedlichen AWS-Konten empfehlen Potter und Ma die zentrale AWS-Konto-Verwaltung *Organizations* [Pot20][S.5] [Ma19][S.8ff]. Aus der Teilung der Aufgaben in unterschiedliche AWS-Konten resultiert, dass NTT-Serverless-CI/CD vier AWS-Konten benötigt. Diese vier AWS-Konten sind ein DevOps-Konto (auch Root- oder Parent-Konto genannt) und drei Umgebungskonten (Development, Integration, Production) die sogenannten Child-Konten. Abbildung 4.1 zeigt AWS-Dienste, welche von NTT-Serverless-CI/CD im DevOps-Konto verwendet werden. Dabei übernehmen CloudFormation, CodePipeline, CodeBuild und Lambda die Erstellung und Verwaltung der CI/CD-Pipeline. Für das Konfigurationsmanagement wird SystemsManager genutzt, die Anbindung des SCM wird durch das API-Gateway in Verbindung mit dem SecretsManager und S3 geregelt. Außerdem beinhaltet S3 alle Artefakte der CI/CD-Pipeline. Für Sicherheit im System werden Key Management Service (KMS) und Identity and Access Management (IAM) verwendet.

Mit den aus Abbildung 4.1 gezeigten Diensten kann die komplette Interaktion mit dem SCM und dem Prozess der Artefakterstellung im DevOps-Konto betrieben werden. Dies stellt eine klare Trennung der Aufgaben der verschiedenen AWS-Konten dar. Die Umgebungskonten benötigen für die Erstellung der Infrastruktur CloudFormation und es werden i.d.R. weitere Dienste benötigt. Welche dies sind, ist stark von der jeweiligen Software abhängig und kann im Allgemeinen nicht festgelegt werden.

Abbildung 4.2 kann entnommen werden, wie die Infrastruktur in den Child-Konten erzeugt wird. NTT-Serverless-CI/CD unterteilt den CI/CD-Prozess in zwei unterschiedliche Bereiche. Der erste Bereich ist die sogenannte DEV-Pipeline auch DEV-Stage genannt. Wie der Name vermuten lässt, ist diese für die Entwicklung (DEVelopment) vorgesehen. Der zweite Bereich ist die DELIVERY-Pipeline auch DELIVERY-Stage genannt. Diese übernimmt die Auslieferung (DELIVERY) von Artefakten an die Umgebungen Integration und Production. Es ist nicht zwingend nötig, aber es ist zur Qualitätssicherung und für das Veröffentlichungs-Management zu empfehlen, in der DELIVERY-Pipeline manuelle Bestätigungsschritte einzubauen und diese mit IAM-Rechten zu versehen, damit nur berechtigte Personen eine Produktivveröffentlichung veranlassen können.

Beide Pipelines erreichen in einem Durchlauf einen oder mehrere CloudFormation-Stages. Diese führen durch eine IAM-Rolle auf der jeweiligen Zielumgebung CloudFormation-Stacks aus, welche die Infrastruktur für die jeweilige Umgebung erzeugen. Ob diese Erzeugung in

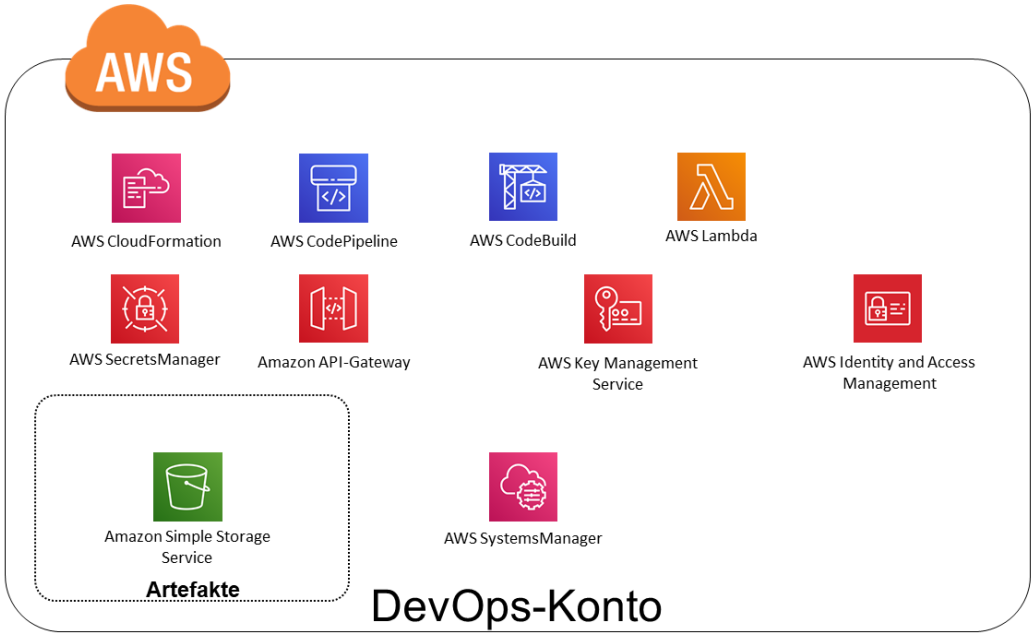


Abbildung 4.1 Dienste des DevOps-Kontos

einem, mehreren oder geschachtelten CloudFormation-Stacks bzw. -Stages geschieht, ist ganz den benötigten Anforderungen anzupassen. Mit dem NTT-Serveless-CI/CD-Framework ist die Entscheidungsfreiheit bei der Verwendung vom Nutzer zu treffen und nicht durch NTT-Serveless-CI/CD festgelegt.

In Abbildung 4.2 wurde der Veröffentlichungsschritt nicht berücksichtigt, da dieser software-spezifisch ist. Beispiele, wie die Veröffentlichung aussehen kann, sind im Kapitel 5 für einen Microservice und eine Angular-Applikation zu finden. In Abbildung 4.3 sind die jeweiligen Pipelines mit möglichen Stages zu sehen. So ist die DEV-Pipeline im Vergleich zur DELIVERY-Pipeline kürzer gehalten, da diese den Quell-Code bezieht, testet, baut, Infrastruktur erzeugt und in der Development-Umgebung veröffentlicht. Die DELIVERY-Pipeline bezieht ebenfalls den Quell-Code, führt Tests aus, baut, erzeugt Infrastruktur und veröffentlicht in die Integration-Umgebung und nach einer manuellen Bestätigung wird die Infrastruktur für die Production-Umgebung erzeugt und auch auf dieser veröffentlicht. Bei diesen Pipelines kann darüber diskutiert werden, ob die Test-Stage in die Build-Stage integriert werden oder nach der Build-Stage erfolgen soll, um z.B. End-to-End-Tests auszuführen. Die restlichen Stages folgen einer logischen Abhängigkeit. So wird zuerst der Quell-Code benötigt, ohne diesen ist weder das Testen noch das Bauen möglich. Die Infrastruktur könnte durchaus gleichzeitig mit dem Testen und dem Bauen erzeugt werden, jedoch sollte der Test oder Bau scheitern, wird die Infrastruktur nicht benötigt. Und ohne Infrastruktur ist die Deploy-Stage nicht möglich, da es kein Ziel gibt. Dies ist analog für alle drei Umgebungen der Fall.

Abbildung 4.3 ist jedoch ein Modell. Es ist durchaus möglich, Optimierungen vorzunehmen. So kann z.B. die Infrastruktur in eine Web-Server-Provisionierung und in Load-Balancer-Erzeugung geteilt werden. Dadurch ist es möglich, dass nach der Web-Server-Provisionierung die Load-Balancer-Erzeugung gleichzeitig mit der Veröffentlichung beginnt. Dadurch kann

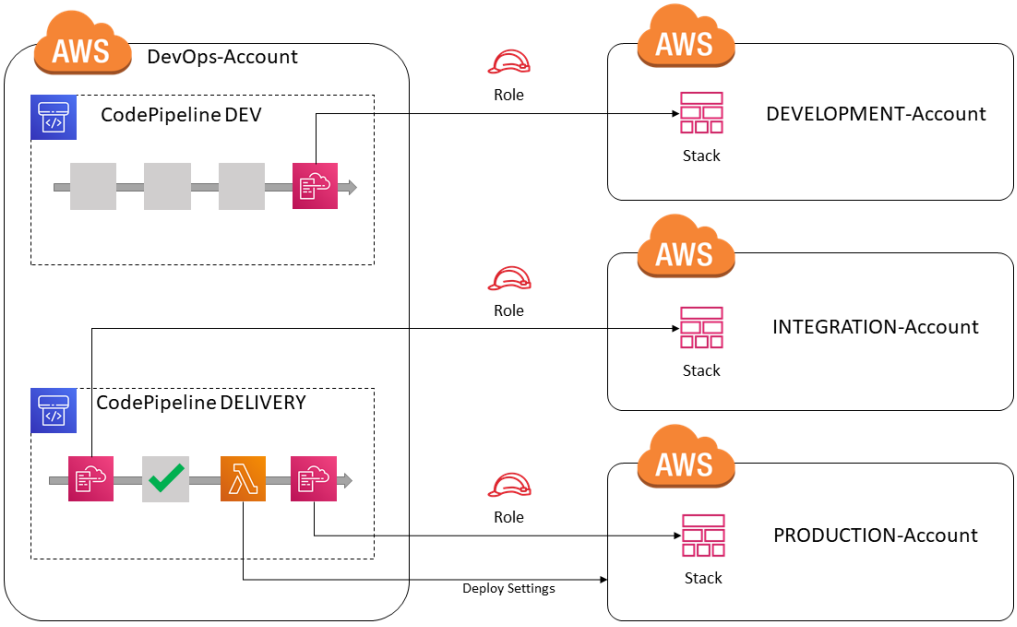


Abbildung 4.2 CodePipeline Infrastruktur [Loi20][Nachbau]

die Gesamtdauer eines Pipeline Durchlaufs verkürzt werden.

4.2 Ablauf der Entwicklung

Initial genutzt wird die von NTT-Serverless-CI/CD erstellte Pipeline durch den Framework eigenen PipelineMaker. Hierbei handelt es sich um ein CodeBuild-Projekt, welches die benötigten Informationen: Git-Referenzen, AWS-Konto-Ids, Pipeline-ARNs, Stage-Bezeichnung (DEV oder DELIVERY), S3-Artefakt-Bucket-ARNs, KMS-ARN, IAM-Rollen-ARNs und Dynamic-Pipeline-Cleanup-Lambda-ARN bezieht. Diese Informationen werden, bis auf die Git-Referenzen und die Stage-Bezeichnung automatisch bezogen, um den Initialisierungsaufwand möglichst gering zu halten und Fehler zu vermeiden.

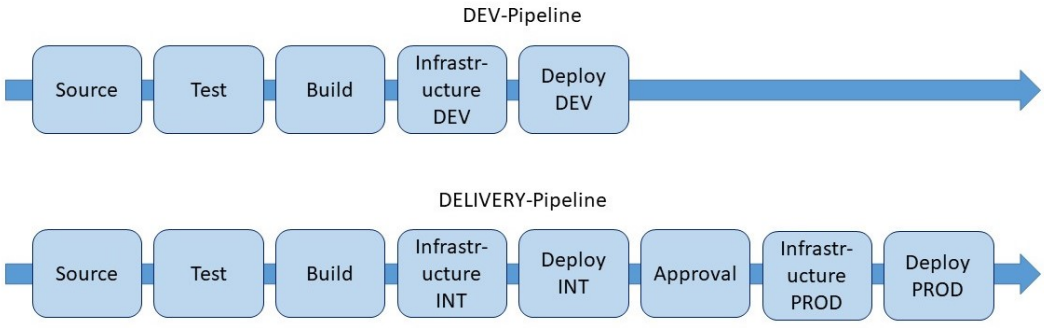


Abbildung 4.3 DEV- & DELIVERY-Pipeline

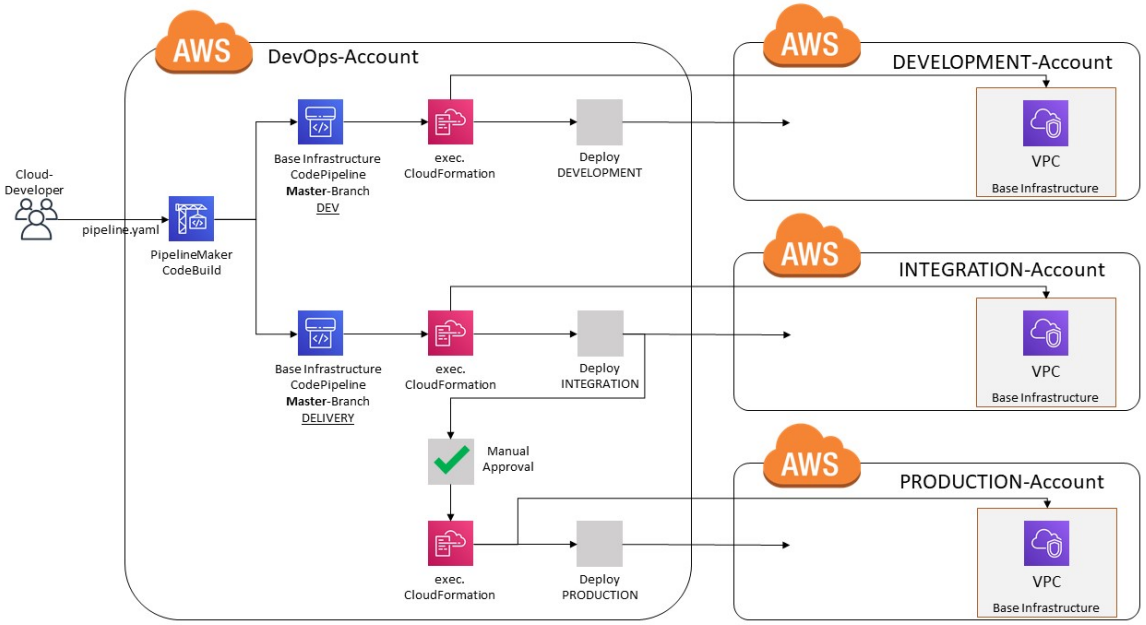


Abbildung 4.4 Nutzung NTT-Serverless-CI/CD [Loi20][Modifizierung]

Abbildung 4.4 zeigt, wie der PipelineMaker diese Informationen und die pipeline.yaml-Datei nutzt. Die pipeline.yaml-Datei sollte der AWS-Best-Practice folgend im *root*/build-Ordner liegen. Dieser Pfad kann durch den PipelineMaker konfiguriert werden, was nicht empfohlen wird. Abhängig von der Stage-Bezeichnung (DEV und DELIVERY) wird die jeweilige Pipeline erzeugt. Durch die Pipeline wiederum wird CloudFormation angestoßen, was auf der jeweiligen Umgebung ausgeführt wird, um Infrastruktur zu erzeugen. Nachdem die Infrastruktur erzeugt wurde, finden das software-spezifische Veröffentlichen statt. I.d.R. wird der Master-Branch, welcher die Basis-Infrastruktur initial enthalten soll, sowohl mit DEV als auch DELIVERY ausgeführt, um alle drei Umgebungen zu initialisieren.

Nachdem die Umgebungen initialisiert sind, kann mit der Entwicklung begonnen werden. Abbildung 4.5 zeigt ein Beispiel, in welchem die Infrastruktur um ECS (Service und Task) erweitert wurde, um mit diesem einen Container aus einem ECR zu starten. Um dies zu tun, muss das CloudFormation-Template an die Infrastruktur angepasst werden, sowie die buildspec.yaml um das Container-Abbild zu erzeugen und zu speichern. Natürlich muss der eigentliche Anwendungs-Code ebenfalls entwickelt werden, dies wird jedoch vorausgesetzt. Um nun die neue Infrastruktur und Funktionalität nutzen zu können, ist das Erzeugen eines neuen Feature-Branches durch die Git-Software (aktuell wird GitHub, GitLab und Bitbucket unterstützt) nötig. Diese Erzeugung löst einen Webhook aus, welcher wiederum den DynamicPipelineMaker auslöst (siehe Kapitel 4.3). Der DynamicPipelineMaker erzeugt auf die Erstellung eines neuen Feature-Branches immer eine DEV-Pipeline, welche CloudFormation auf der DEVELOPMENT-Umgebung auslöst. Dadurch wird die Infrastruktur um ECS erweitert. Anschließend wird in diesem ECS-Service ein Container durch ein Container-Abbild aus ECR gestartet. Dies geschieht automatisiert, durch das Erzeugen eines Feature-Branches sowie durch die Anpassung der CloudFormation-Templates.

Davon ausgehend, dass die Implementierung richtig war und alle Tests bestanden wurden,

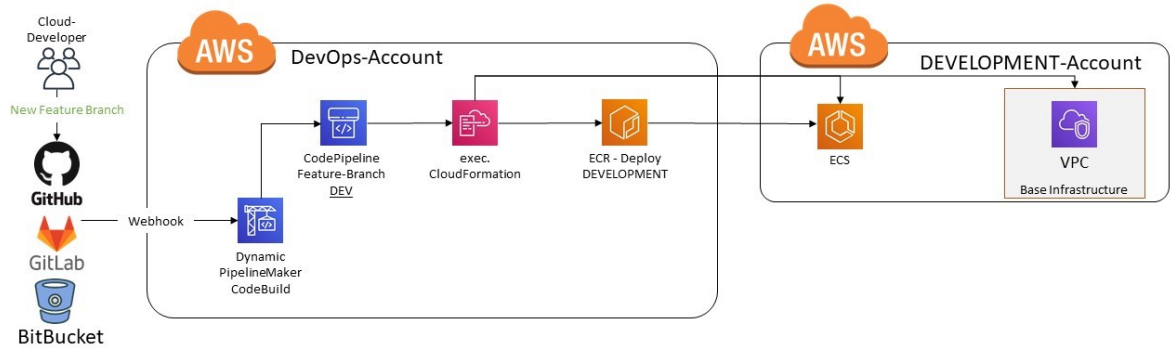


Abbildung 4.5 Bsp: Neuer Feature-Branch [Loi20][Modifizierung]

besteht der nächste Schritt darin, den Feature-Branch in den Master-Branch zu integrieren und zu entfernen. Hierfür wird ebenfalls die Git-Software genutzt. Das Integrieren und Entfernen löst zwei unterschiedliche Aktionen aus, welche auf Abbildung 4.6 zu sehen sind.

Durch das Löschen des Feature-Branches wird CloudFormation angestoßen, welches automatisiert die komplette Infrastruktur mit den dazugehörigen Artefakten (ECR) entfernt. Hierfür werden Lambda-Funktionen genutzt, welche durch NTT-Serverless-CI/CD zur Verfügung stehen. Das Entfernen der Infrastruktur und der Artefakte geschieht sowohl auf dem DevOps-Konto, als auch auf dem Development-Konto.

Das Integrieren des Feature-Branches in den Master-Branch löst die in Abbildung 4.4 erstellte DELIVERY-Pipeline aus. Dies geschieht jedoch nicht durch den Webhook. Stattdessen reagiert DELIVERY-Pipeline durch Polling auf Git-Änderungen. Dies ermöglicht es auch auf Änderungen im Master-Branch ohne die Integration von Feature-Branches zu reagieren, da in der Praxis nicht immer nur mit Feature-Branches gearbeitet wird. Durch das Polling wird der neue ECS-Service festgestellt und in der jeweiligen Umgebung (Integration und Production) in die Infrastruktur übernommen. Dies trifft ebenfalls auf ECR und das Erzeugen des Container-Abbildes zu. Auf diese Weise ist es nicht nur möglich neue Infrastrukturen hinzuzufügen., sondern es kann z.B. auch die Anzahl von Container-Instanzen verändert werden oder gar Infrastruktur komplett entfernt werden, falls diese nicht mehr benötigt wird.

Das automatisierte Anpassen der Infrastruktur betrifft lediglich Infrastruktur, welche software-spezifisch ist. Dies bedeutet, dass die CI/CD-Pipeline selbst nicht automatisiert geändert werden kann. Es kann zwar die pipeline.yaml-Datei angepasst werden, dies führt jedoch zu keiner Änderung der Pipeline. Hierfür ist ein manuelles Update der von PipelineMaker erzeugten CloudFormation-Stacks nötig. Dies schützt vor versehentlichen bzw. unbefugten Anpassungen der CI/CD-Pipeline, macht jedoch auch Änderungen an der CI/CD-Pipeline schwieriger. Da eine CI/CD-Pipeline i.d.R. selten Änderungen unterworfen ist, ist dies zu vernachlässigen.

Die CI/CD-Pipeline hat das gleiche Verhalten wie von anderen CI/CD-Werkzeugen bekannt. So lässt das Scheitern einer Stage die komplette Pipeline scheitern. Die Benachrichtigung über das Scheitern und anderen Zuständen der Pipeline kann über den AWS-Nachrichten-Service SNS oder Chatbot (Slack) weitergeleitet werden. SNS ermöglicht es neben der Benachrichtigung durch E-Mails und SMS auch Lambda-Funktionen anzusprechen. Dadurch ist es möglich die meisten Schnittstellen anderer Systeme anzusprechen, wie

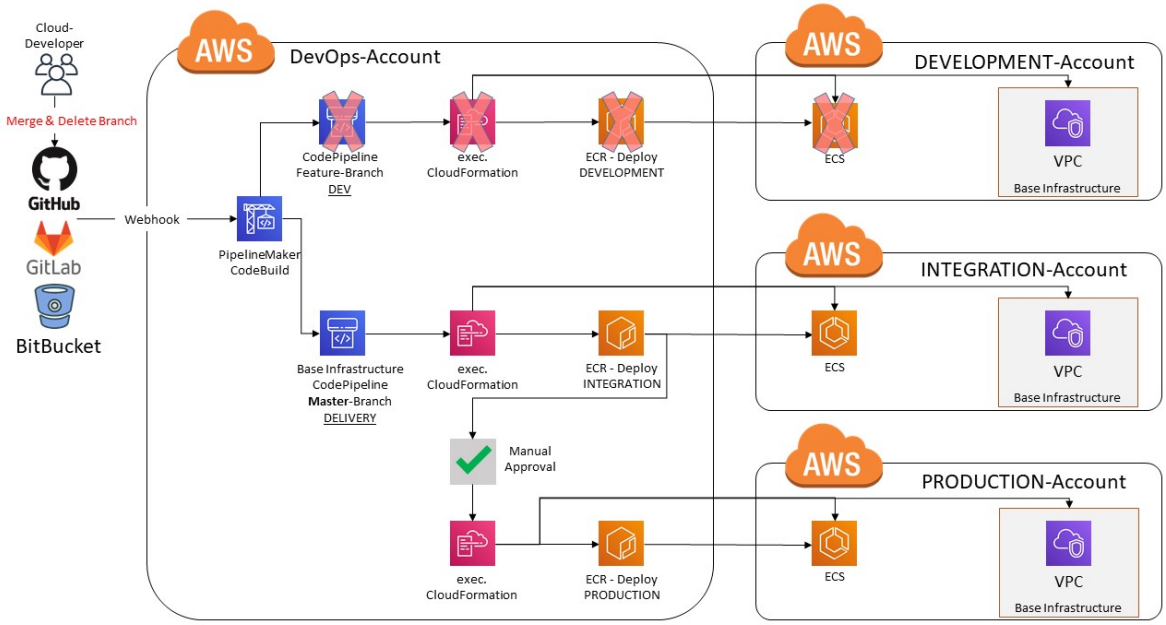


Abbildung 4.6 Integrieren und Entfernen neuer Infrastrukturen [Loi20][Modifizierung]

z.B. Meldungen über das Scheitern oder den Erfolg der Pipeline an die Git-Software.

4.3 Integration der Git-Software

NTT-Serverless-CI/CD ist aktuell nur mit den drei Git-Software Produkten GitHub, GitLab und Bitbucket kompatibel. Ob eine Erweiterung für CodeCommit in Zukunft erscheint, ist naheliegend, aber unklar.

Die Anbindung der Git-Software folgt der Referenzarchitektur von AWS, welche in Abbildung 4.7 zu sehen ist. In dieser Referenzarchitektur ist vorgesehen, dass vom Nutzer ausgelöste Git-Aktionen mittels Webhook durch die Git-Software an ein API-Gateway gesendet werden. Dieses API-Gateway wiederum leitet die Git-Aktion an eine Lambda-Funktion weiter, welche mittels KMS und SSH auf den Quell-Code zugreift und diesen in einem S3-Bucket ablegt.

Abbildung 4.8 zeigt, wie NTT-Serverless-CI/CD Git-Software integriert. Wird in der Git-Software ein Webhook ausgelöst, wird eine HTTP-Anfrage an ein API-Gateway gesendet. Das API-Gateway leitet diese HTTP-Anfrage an die GitManagementLambda weiter, welche durch NTT-Serverless-CI/CD initialisiert wurde. Diese GitManagementLambda prüft, ob es sich um eine DELETE- oder PUSH-Aktion handelt, abhängig davon löst die GitManagementLambda das Löschen der nicht mehr benötigten Daten im S3-Bucket ArtifactStore-DEV aus oder bezieht den Quell-Code mittels eines KMS-Schlüssels und eines vorab bei der Git-Software registrierten SSH-Schlüssels und legt diesen Quell-Code im S3-Bucket ArtifactStore-DEV ab.

Wie die Interaktion mit dem S3-Bucket ArtifactStore-DEV vermuten lässt, interagiert die GitManagementLambda nur mit der DEV-Pipeline bzw. mit Ressourcen, welche mit der DEV-Pipeline in Zusammenhang stehen. Dies liegt daran, dass nur für Feature-Branches Infrastrukturen dynamisch erzeugt und zerstört werden sollen. Die in Kapitel 4.2 bzw. in

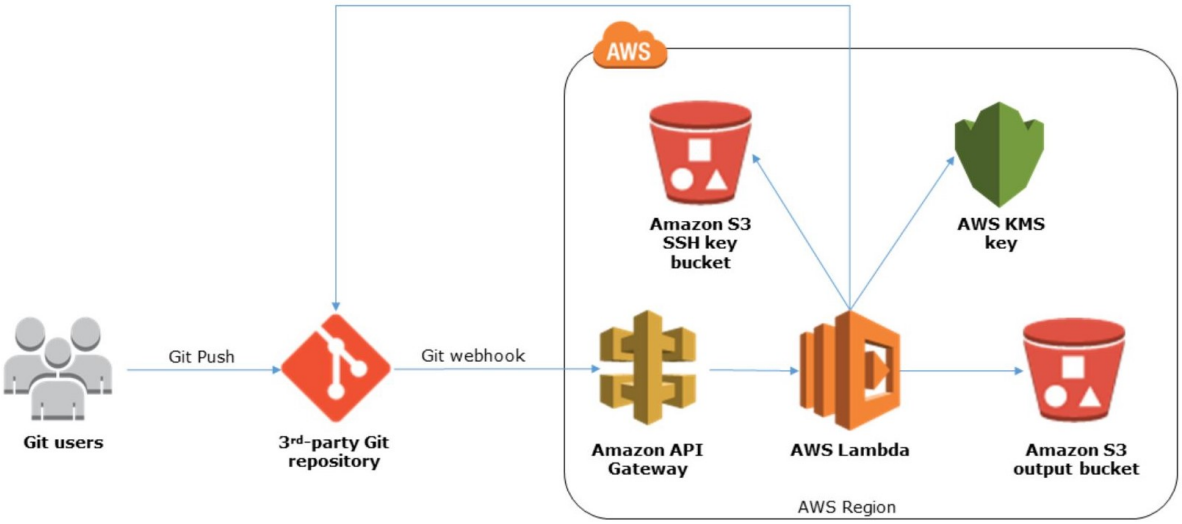


Abbildung 4.7 Webhook endpoint architecture on AWS [McC17][S.6]

Abbildung 4.4 erstellten Pipelines sind bereits initialisiert bzw. hier wird die DELIVERY-Pipeline manuell initialisiert, was das automatisierte Erzeugen überflüssig macht. Über Code-Änderungen wird die DELIVERY-Pipeline durch Polling informiert und reagiert auf diese. Die Aktivierung von Polling ist ebenfalls für die DEV-Pipeline bzw. für Feature-Branches sinnvoll.

Feature-Branches können jedoch im Gegensatz zum Master-Branch automatisiert per Webhook initialisiert werden. Das automatisierte Entfernen von Infrastruktur ist bei der DEV-Pipeline und theoretisch auch bei der DELIVERY-Pipeline möglich. Jedoch wird davon abgeraten die DELIVERY-Pipeline mit dieser Funktionalität zu versehen (siehe Kapitel 5), da ein versehentliches Entfernen von Infrastruktur in der Integration- und/oder Production-Umgebung große wirtschaftliche Folgen haben könnte.

S3 erkennt durch die Änderung im ArtifactStore, ob es sich um eine Erzeugung neuer Daten oder um das Entfernen von Daten handelt. Abhängig hiervon wird die DynamicPipeline-MakerLambda ausgelöst, welche aus Abbildung 4.5 bekannt ist und eine neue DEV-Pipeline erzeugt und ausführt. Das Entfernen von Daten löst die DynamicPipelineDeleteLambda aus, welche wiederum durch CloudFormation das Löschen der für den Branch erzeugten DEV-Pipeline auslöst. Durch erzeugte CustomRessources mittels CloudFormation kann die DEV-Pipeline die DynamicPipelineCleanupLambda auslösen, welche das Entfernen der Infrastruktur in der Development-Umgebung auslöst.

4.4 Vor- und Nachteile von NTT-Serverless-CI/CD

NTT-Serverless-CI/CD ermöglicht die schnelle und automatisierte Einrichtung von CI/CD und IaC im Kontext von AWS. Dies wiederum vereinfacht die Einführung und Arbeit mit DevOps. Die Vorteile, welche durch DevOps, CI/CD und IaC und dadurch auch durch NTT-Serverless-CI/CD gegeben sind, wurden bereits ausführlich in Kapitel 2 beschrieben. Ebenfalls wurden im Kapitel 3 Vorteile von Cloud, Serverless, AWS und insbesondere von CloudFormation beschrieben, zu welchen NTT-Serverless-CI/CD ebenfalls beiträgt. Eine genaue wirtschaftliche Berechnung ist im Kapitel 6 zu finden.

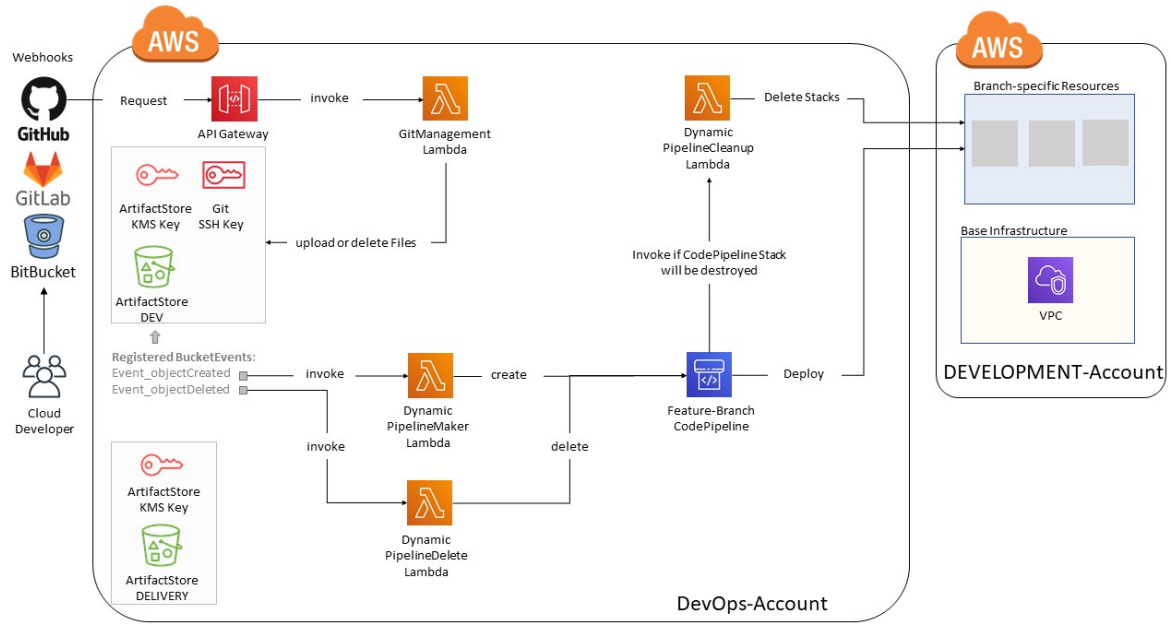


Abbildung 4.8 Webhook Architektur [Loi20][Modifizierung]

Darüber hinaus, bietet NTT-Serverless-CI/CD noch weitere Vorteile. So werden noch zusätzliche Funktionalitäten mitgeliefert, um die Pipelines zu erweitern:

Custom CodeBuild Images Diese Funktionalität ist bereits in NTT-Serverless-CI/CD eingebunden, auch wenn sie nicht für jede Software benötigt wird. Es wird dadurch eine Methode angeboten, eigene Docker-Images in CodeBuild zu integrieren. Sollte diese Funktionalität nicht benötigt werden kann die mitgelieferte DeleteECRImagesLambda dennoch hilfreich sein. Diese ermöglicht ein einfaches Entfernen von Images aus ECR auf dem DevOps-Konto, was für das Entfernen von DEV-Stacks erledigt werden muss, falls diese DEV-Stacks ECR enthalten und in Docker-Images speichern.

Settings Management In NTT-Serverless-CI/CD ist ein Mechanismus vorbereitet, welcher es ermöglicht, Konfigurationsdateien zu verwenden, anstatt vorab Konfigurationsparameter in SystemManager, SecretsManager oder in den CloudFormation-Templates zu hinterlegen. Dieser Mechanismus verwendet die übliche Konvention, Konfigurationsdateien im JSON-Format im *root*/settings-Ordner abzulegen. Diese Dateien können dann in der Pipeline durch die SettingsManagementLambda gelesen werden. Diese speichert die Parameter abhängig davon, ob diese verschlüsselt sind oder nicht, im SystemsManager oder im SecretsManager von AWS.

Einfache Updates Dadurch, dass CloudFormation verwendet wird um NTT-Serverless-CI/CD einzurichten, ist es möglich, Änderungen bzw. Updates durch CloudFormation auszurollen. Dies wiederum bedeutet, dass ähnlich wie bei der Einrichtung von NTT-Serverless-CI/CD, ein Update durch eine einfache CloudFormation Änderung durchgeführt werden kann.

Parallelität Da die erzeugten Pipelines unabhängig sind und AWS sowohl für CodePipeline

als auch für CodeBuild theoretisch unendliche Kapazitäten zur Verfügung stellt, ist es möglich, beliebig viele Pipelines gleichzeitig auszuführen. Dadurch ist es nicht mehr nötig, dass Feature- und Produktiv-Builds nacheinander ausgeführt werden. Diese werden parallel ausgeführt und Build-Warteschlangen existieren folglich nicht.

NTT-Serverless-CI/CD setzt von Design aus auf das *Least Privileged*-Konzept was IAM-Rollen und Rechte betrifft, um Sicherheitsrisiken gering zu halten [Pot20][S.3]. Und von Design aus auch auf die Trennung der Aufgaben in vier AWS-Konten. Diese bereits im Kapitel 4.1 beschriebene Trennung in vier unterschiedliche AWS-Konten setzt allerdings auch voraus, dass diese Konten vorhanden sind. Es ist somit nicht möglich, ohne manuelle Eingriffe NTT-Serverless-CI/CD mit mehr oder weniger als vier AWS-Konten zu betreiben. Dies soll sich jedoch bis zur Veröffentlichung von NTT-Serverless-CI/CD ändern, siehe 4.6.

NTT-Serverless-CI/CD bietet das Potenzial, die Qualitätssicherung zu fördern und den Prozess zu verändern. Dadurch, dass die Infrastruktur für jeden Feature-Branch erzeugt wird, ist es möglich Feature-Branches ausführlich zu testen, bevor diese in den Master-Branch integriert werden, wodurch (fachliche) Fehler bereits im Feature-Branch erkannt werden können, bevor diese in die Integration- oder Produktionsumgebung eingehen.

Ein Vorteil gegenüber anderen CI/CD-Werkzeugen, welcher durch CodePipeline und nicht direkt durch NTT-Serverless-CI/CD entsteht, ist das Wiederholen gescheiterter Stages. Die Artefakte, welche durch jede Stage erzeugt werden, werden in S3 gespeichert. Dadurch ist es möglich, diese im Fehlerfall zu analysieren, aber auch gescheiterte Stages zu wiederholen, ohne die komplette Pipeline wiederholt zu durchlaufen. Für kleine Software, ist dies wenig relevant. Sollte die Software und die Durchlaufzeit der Pipeline wachsen, kann es von relevantem zeitlichen und finanziellen Vorteil sein, einzelne gescheiterte Stages zu wiederholen.

Als Vorteil kann auch die leichte Kopplung von NTT-Serverless-CI/CD an die Software betrachtet werden. Es kann ohne große Code-Änderungen wieder aus einer Software entfernt werden, um AWS-Werkzeuge oder andere CI/CD-Ansätze zu nutzen. Die benötigten Änderungen sind jedoch nur dann gering, wenn weiterhin für Infrastruktur CloudFormation genutzt wird.

Es gibt auch einige nicht allzu schwerwiegende negative Punkte, welche angemerkt werden müssen:

Veraltete Python Version Drei Lambda-Funktionen: GitCheckoutLambda, GitManagement-Lambda und SSHKeyManagementLambda wurden in der veralteten Python 2.7 Version geschrieben. Python 2.7 ist nach python.org [Pyt] seit dem 01.01.2020 veraltet und sollte nicht mehr verwendet werden. Ein Update ist mit kleinen Code-Änderungen möglich. Ob dies vor Veröffentlichung noch in das Framework eingeht, ist unklar.

Keine automatischen Änderungen der Pipeline Automatisierte Änderungen an der Infrastruktur können nicht über die pipeline.yaml gemacht werden. Diese pipeline.yaml wird auf dem DevOps-Konto ausgeführt und Änderungen in der pipeline.yaml müssen manuell angestoßen werden. Dies ist nicht mit Änderungen an der Infrastruktur für die Umgebungskonten zu verwechseln.

Feste Pfade Pfade zu bestimmten Dateien, z.B. pipeline.yaml => *root*/build-Ordner , buildspec.yaml => *root*-Ordner, Konfigurationsdateien => *root*/settings, sind hart kodiert und können nicht trivial geändert werden.

Namenskonvention Die Bezeichnungen DEV und DELIVERY sind gewöhnungsbedürftig. Außerdem muss z.B. für die DeleteECRImagesLambda der Prefix *dev* vergeben sein, damit die entsprechenden Images entfernt werden.

AWS-Fokus Es ist nur im AWS-Umfeld einsetzbar.

Begrenzte Zeichenmenge für AWS-Ressourcen Die Begrenzung für AWS-Ressourcen-Bezeichnungen liegt nicht an NTT-Serverless-CI/CD, soll aber hier auch erwähnt werden. Um eindeutige Stack-, Bucket- und andere AWS-Ressourcen-Bezeichner zu verwenden, wird wie in Kapitel 5 oft eine Kombination aus Ressourcentyp, Projektname, Branch-Name und Pipeline-Umgebung (DEV und DELIVERY) verwendet. Dies erzeugt klar verständliche, eindeutige Bezeichner. Jedoch ist die Zeichenmenge, welche verwendet werden kann, abhängig von der Ressource durch AWS begrenzt. So ist z.B. die Grenze von S3-Bucket-Bezeichnern auf 63 Zeichen beschränkt. Dies scheint auf den ersten Blick ausreichend. Wenn jedoch eine zusammengefügte Bezeichnung verwendet wird, wie für:

Ressourcentyp Bucket

Projektname master-thesis-ampletzer

Branch-Name feature-test-new-infrastructure

Pipeline-Umgebung delivery

ergibt sich bucket-master-thesis-ampletzer-feature-test-new-infrastructure-delivery und die 63 Zeichen sind überschritten. Dies führt dazu, dass andere eindeutige Bezeichner verwendet werden müssen, sollte an diese Grenze gestoßen werden. Gute Bezeichnungen zu finden, gehört zu den schwierigen Aufgaben der Entwicklung, Näheres hierzu ist bei Martin [Mar09] zu finden. Hier soll nur davor gewarnt werden, Abkürzungen zu verwenden, da diese zur Unwartbarkeit von Code beitragen [Mar09][S.45ff].

Nur ein SSH-Schlüssel Die von McConnel beschriebene Best-Practice für jedes SCM-Repository einen eigenen SSH-Schlüssel zu verwenden, ist nur bedingt möglich [McC17][S.8]. Verwendet man NTT-Serverless-CI/CD mit nur einem SCM-Repository wird der durch NTT-Serverless-CI/CD erzeugte SSH-Schlüssel für dieses SCM-Repository verwendet und die Best-Practice ist eingehalten. Möchte man jedoch mehr als ein SCM-Respository im DevOps-Konto mit NTT-Serverless-CI/CD betreiben, ist dies möglich, indem man NTT-Serverless-CI/CD unter geänderter Stack-Bezeichnung neu initialisiert oder NTT-Serverless-CI/CD nur einmal initialisiert, dann muss jedoch der erzeugte SSH-Schlüssel mehrmals verwendet werden.

Diese Nachteile sind nicht schwerwiegend, sollten jedoch vor der Verwendung von NTT-Serverless-CI/CD zu Kenntnis genommen und berücksichtigt werden.

4.4.1 Vorteile gegenüber AWS-CI/CD

Im Kapitel 4.4 wurden Vorteile von NTT-Serverless-CI/CD aufgezeigt, welche zu großen Teilen auf DevOps, Automatisierung und die AWS-Dienste zurückzuführen sind. Da dies so ist, stellt sich unweigerlich die Frage, welche Vorteile NTT-Serverless-CI/CD gegenüber der regulären Verwendung der AWS-CI/CD-Werkzeuge bietet.

Im Grunde nutzt NTT-Serverless-CI/CD lediglich AWS-Dienste unter Zuhilfenahme von Lambda-Funktionen und CloudFormation-Templates. Daraus kann geschlossen werden,

dass mit dem nötigen Wissen die gleiche Funktionalität manuell mit den AWS-CI/CD-Werkzeugen und Lambda-Funktionen entwickelt werden kann. Da dies zutrifft und somit NTT-Serverless-CI/CD nicht benötigt wird, stellt sich die Frage: „Wozu benötigt man NTT-Serverless-CI/CD?“

NTT-Serverless-CI/CD löst Probleme und beantwortet Fragen, welche durch die Umsetzung von CI/CD in AWS auftreten. Einige dieser Probleme und Fragen sind:

- Welche Architektur soll verwendet werden?
- Welche Berechtigungen sind zu vergeben?
- Welche Best-Practices sind zu berücksichtigen und wie sind diese zu verwenden?
- Wie kann die Git-Software integriert werden?
- Wie wird Infrastruktur für Feature-Branches erzeugt?
- Wie wird die erzeugte Infrastruktur wieder entfernt, um die Kosten gering zu halten?
- Wer hat die Kapazität und das Wissen die CI/CD zu implementieren?

NTT-Serverless-CI/CD beantwortet diese Fragen bzw. gibt Hilfestellungen hierzu. Indem es die Architektur, den Prozessablauf und die Integration von Ressourcen und Git-Software vorgibt, vereinfacht NTT-Serverless-CI/CD die Verwendung von CI/CD in AWS und gibt einen Leitfaden zur Hand. Die Vorteile, welche durch NTT-Serverless-CI/CD entstehen, können wie folgt zusammengefasst werden:

Zeitersparnis Durch den erhöhten Grad an Automatisierung können Pipelines und Infrastruktur für Feature-Branches automatisiert erzeugt und komplett zerstört werden. Dies spart Zeit für manuelles Erstellen und Zerstören der Feature-Pipelines bzw. erübrigt die Implementierung einer eigenen anderweitigen Lösung für diese Probleme.

Best-Practices NTT-Serverless-CI/CD erzwingt die Einhaltung von Best-Practices. So sorgt z.B. die Aufgabentrennung in unterschiedliche AWS-Konten für klare Zuständigkeiten und Rechte, wodurch die Übersicht, Wartbarkeit und Sicherheit erhöht wird.

Einfache zeitsparende Initialisierung IAM-Rollen und -Rechte, Lambda-Funktionen, S3-Buckets usw., welche für CI/CD in AWS benötigt werden, werden automatisiert erzeugt. Ohne Kenntnis der verwendeten Technologien kann enorme Zeitersparnis bei der Einrichtung des CI/CD Prozesses sowie eine Reduzierung der Fehler erreicht werden.

Zuverlässigkeit Durch die Automatisierung werden menschliche Fehler bei der Initialisierung von NTT-Serverless-CI/CD, sowie der Erzeugung und Zerstörung von Feature-Branches vermieden.

Diese Vorteile werden auch erzielt, wenn das automatisierte Erstellen und Löschen von Infrastruktur für Feature-Branches nicht genutzt wird. Sollte diese große Stärke von NTT-Serverless-CI/CD nicht benötigt werden, ist zu empfehlen nicht benötigte Ressourcen wie das API-Gateway und diverse Lambda-Funktionen zu entfernen. NTT-Serverless-CI/CD bietet dennoch gegenüber dem manuellen Einrichten der CI/CD-Infrastruktur die beschriebenen Vorteile. Auch wenn diese bei der nicht mehr so umfassenden Infrastruktur schwächer ausfallen.

4.5 Sicherheitsrisiken durch NTT-Serverless CI/CD

NTT-Serverless-CI/CD weist neben den IT-Sicherheitsrisiken, welche durch die Verantwortungsmodelle für Lambda und abstrakte Dienste abgedeckt sind, noch weitere Sicherheitsrisiken auf. Hierbei handelt es sich um Sicherheitsrisiken, welche durch die Schnittstellen mit der Git-Software, welche außerhalb der AWS-Cloud betrieben werden kann, entstehen. Daraus folgt, dass Sicherheitsrisiken durch die Schnittstellen zur AWS-Cloud und dem Server, auf dem die Git-Software betrieben wird, entstehen. Die Sicherung dieses Git-Servers wird in dieser Arbeit nicht berücksichtigt, da dies für die Arbeit nicht relevant ist, jedoch ohne Zweifel wichtig für die IT-Sicherheit ist.

Eine Schnittstelle ist das API-Gateway, welches durch Git-Webhooks angesprochen wird und im Grunde verschiedene Sicherheitsmechaniken beinhaltet. Es kann mit IAM, Cognito, Custom Authorizer über Lambda [Ant18][1] mit, API Keys und Ressourcen-Richtlinien versehen werden.

Jedoch ist es standardmäßig nicht durch NTT-Serverless-CI/CD geschützt und öffentlich ansprechbar. Es können jedoch einige der Sicherheitsmechaniken manuell verwendet werden. Leider ist die Möglichkeit, den API-Key in die Anfrage der Git-Software bei einer Git-Aktion wie PUSH und DELETE mitzusenden, nicht gegeben, was diese einfache Möglichkeit der Sicherung ausschließt. Die IP-Adresse des Git-Servers kann mittels der Ressourcen-Richtlinie in einer Whitelist als vertrauenswürdig gekennzeichnet werden, wodurch der Git-Server die einzige IP-Adresse besitzt, welche das API-Gateway ansprechen kann. Somit werden IP-Adressen von möglichen Angreifern und anderen unbefugten Anfragen abgewiesen.

Weitere Sicherheitsmaßname bieten GitHub[2] und GitLab[3]. Sie können die Anfrage um einen Secret-Token im Anfrage-Kopf erweitern. Dieser Secret-Token kann nach der Weiterleitung durch das API-Gateway in der GitManagementLambda überprüft werden. Hierfür ist allerdings eine Erweiterung der GitManagementLambda nötig. Da eine eigenständige Erweiterung eines Frameworks in den seltensten Fällen zu empfehlen ist, ist es wünschenswert, wenn diese Funktionalität durch NTT-Serverless-CI/CD nachgereicht wird.

Eine weitere Schnittstelle zwischen AWS und der Git-Software ist das Auslösen der GitCheckoutLambda wenn das API-Gateway bzw. die GitManagementLambda eine PUSH-Aktion registriert hat. Die GitCheckoutLambda verwendet den in S3 hinterlegten SSH-Schlüssel, um den Quell-Code aus dem Git-Repository nach S3 in den ArtefactStore zu kopieren. Damit dies möglich ist, muss der öffentliche SSH-Schlüssel in der Git-Software hinterlegt werden. Hier ist ebenfalls zu beachten, dass nur der Quell-Code kopiert werden soll. Somit ist es ausreichend für NTT-Serverless-CI/CD, ein Konto in der Git-Software zu erstellen, welches ausschließlich Leserechte auf das benötigte Repository besitzt [Veh18][4]. SSH ist ein Protokoll, welches entwickelt wurde, um die Sicherheit im Internet zu erhöhen, jedoch auf eine Weise, dass es einfach zu nutzen ist, auch wenn dies auf Kosten absoluter Sicherheit geht [Ylo06]. Jedoch wurde auch SSH um komplexere Verschlüsselungen weiterentwickelt und kann als Sicherheitsstandard für Remote-Verbindungen betrachtet und als sicher angesehen werden [Bid18].

Die letzte Schnittstelle zwischen AWS und der Git-Software ist die Source-Stage. In dieser

1 Kapitel: Amazon API Gateway

2 `https://docs.github.com/en/free-pro-team@latest/developers/webhooks-and-events/securing-your-webhooks#setting-your-secret-token` Stand: 20.11.2020

3 `https://docs.gitlab.com/ee/user/project/integrations/webhooks.html#secret-token` Stand: 20.11.2020

4 Kapitel: Building a barebones DevOps pipeline & Kapitel: Security layer 4: securing the delivery pipeline

Source-Stage wird ebenfalls ein Schlüssel für die Git-Software benötigt. Hierfür kann z.B. OAuth verwendet werden. Bei der Verwendung von OAuth ist jedoch darauf zu achten, dass der Schlüssel ausreichend lang ist und nicht in Klartext in der pipeline.yaml sondern verschlüsselt über z.B. den SecretsManager verwendet wird.

4.5.1 Sicherheitsaspekte bei der Verwendung von NTT-Serverless-CI/CD

Sicherheit sollte in einer Applikation immer oberste Priorität haben, dies ändert sich nicht, wenn es sich um eine Serverless-Architektur handelt [Bai17][S.22]. Da dies auch auf NTT-Serverless-CI/CD zutrifft, gibt es eine Reihe von Best-Practices, welche bei der Entwicklung von NTT-Serverless-CI/CD berücksichtigt wurden, aber auch in der Verwendung weiter berücksichtigt werden sollten. Da es nicht möglich ist, auf jede Software einzugehen, welche durch NTT-Serverless-CI/CD unterstützt werden kann, soll hier nur exemplarisch auf allgemeine Punkte im Zusammenhang mit NTT-Serverless-CI/CD eingegangen werden.

- IAM Best-Pratices [Pri18][5]
  - Der Root-Nutzer sollte nur einmal zur Erstellung eines Administratoren-Nutzers verwendet werden und die Zugangsschlüssel für den Root-Nutzer brauchen nie aktiviert werden.
  - Zugangsschlüssel sollten regelmäßig rotieren. Dies trifft auf alle Zugangsschlüssel und Passwörter zu, auch auf vom Systemen erstellte Schlüssel wie SSH-Schlüsselpaare. KMS-Schlüssel werden bei NTT-Serverless-CI/CD automatisch einmal jährlich rotiert.
  - Multi-Faktor-Autorisierung sollte für den Root-Nutzer, aber auch für alle anderen Nutzer aktiviert werden.
  - Für die Rechtevergabe sollen Gruppen und Richtlinien genutzt werden. Keine direkten Rechte für Nutzer oder Rollen sollen vergeben werden.
  - Das **Least Privilege**-Prinzip soll berücksichtigt werden. Dies bedeutet, dass so wenig Rechte wie möglich vergeben werden und diese Rechte so eng wie möglich gefasst werden.
  - Eine starke Passwortrichtlinie sollte eingerichtet werden.
  - **Nie** Zugangsdaten teilen, auch nicht für laufende Applikationen. Für Applikationen oder kontoübergreifende Zugänge sollen IAM-Rollen verwendet werden.
  - Nicht benötigte Zugangsdaten sollen entfernt werden.
- Unterteile benötigte Dienste und Infrastruktur in technische Schichten, welche wiederum in private und öffentliche Subnetze unterteilt werden [Pot20][S.16]
- Es sollte regelmäßig auf bekannte Sicherheitslücken überprüft werden. Teile dieser Überprüfung können durch die CI/CD-Pipeline automatisiert werden. [Pot20][S.18]
- Linter wie cfn-lint gehören zur Best-Practice für CloudFormation und sollten genutzt werden [Tov20][6]. Um Sicherheitslücken schneller zu entdecken, kann neben cfn-lint, welcher CloudFormation-Templates validiert, eine Sicherheits-Linter wie cfn-nag [7]

5 Kapitel: IAM best practices
6 Kapitel: Validation, Linting, and Deployment of the Stack
7 `https://github.com/stelligent/cfn_nag`

verwendet werden. cfn-nag überprüft Muster, welche auf unsichere Infrastruktur hindeuten.

4.5.2 Weitere Risiken

Durch NTT-Serverless-CI/CD und AWS ist es sehr einfach, Infrastruktur zu erzeugen und bestehende Infrastruktur zu erweitern. Dies ist im Sinne des DevOps-Ansatzes. Jedoch birgt dies die Gefahr, insbesondere für unerfahrene Entwickler/innen unnötige bzw. **kostenintensive Infrastruktur** versehentlich zu erzeugen. Da gerade das einfache Erzeugen neuer Infrastruktur eine Stärke von NTT-Serverless-CI/CD ist, muss eine Möglichkeit gefunden werden, Fehler nicht entstehen zu lassen oder sie möglichst bald zu finden.

Für die Integration von Feature-Branches in den Master-Branch sollte ein Code-Review stattfinden. Dies wird durch die kompatible Git-Software in Form von Pull-Requests ermöglicht. Zusätzlich kann und sollte eine manuelle Bestätigung (engl. Approval) in die DELIVERY-Pipeline vor den Production-Stages integriert werden. Weitere manuelle Bestätigungen können, müssen jedoch nicht sinnvoll sein.

Diese Maßnahmen schützen jedoch Feature-Branches nicht vor unnötiger/kostenintensiver Infrastruktur. Auch bei der Initialisierung eines Feature-Branches kann die Git-Software nicht unterstützen. Um unerfahrene Entwickler/innen vor Fehlern zu bewahren, kann während der Entwicklung das 4-Augen-Prinzip z.B. in Form von Pair-Programming genutzt werden.

Unabhängig von der Erfahrung der Entwickler/innen kann der AWS Billing Alert genutzt werden um Nachrichten bei Kostenüberschreitung zu erhalten. So können hohe AWS-Kosten nicht unbedingt verhindert werden, jedoch frühzeitig gegengesteuert werden, um die Kosten nicht unkontrolliert weiter wachsen zu lassen.

Für NTT-Serverless-CI/CD besteht eine klare **Abhängigkeit von AWS**. Ohne AWS wäre NTT-Serverless-CI/CD sinnlos. Dennoch ist aufgrund der schieren Größe von AWS davon auszugehen, dass AWS lange Bestand haben wird. Die Gefahr besteht eher darin, dass es Änderungen in AWS geben könnte, welchen sich NTT-Serverless-CI/CD unterwerfen muss. Da jedoch AWS Änderungen oft schleichend einführt bzw. ältere Versionen noch unterstützt, ist zu erwarten, dass NTT-Serverless-CI/CD genügend Zeit hat, diese Änderungen umzusetzen. Somit sollten Änderungen in AWS eine geringe Gefahr darstellen.

Ein Wechsel zu einem anderen Cloud-Provider macht NTT-Serverless-CI/CD überflüssig bzw. sorgt dafür, dass es nicht mehr nutzbar ist. Dies trifft ebenfalls auf die erzeugten IaC-Skripte im CloudFormation-Format zu. Um die Möglichkeit, erzeugte IaC-Skripte bei einem Möglichen Cloud-Provider-Wechsel weiterhin (teilweise) nutzen zu können, sollte Terraform verwendet werden (siehe Kapitel 4.6). Einen Cloud-Provider zu wechseln würde bedeuten, dass die CI/CD-Werkzeuge von AWS ebenfalls nicht mehr genutzt werden könnten und eine andere CI/CD-Lösung gefunden werden müsste. Falls nur die Infrastruktur für eine Software zu einem anderen Cloud-Provider wechseln soll, bietet AWS mit CodeDeploy die Möglichkeit in andere Cloud-Umgebungen zu veröffentlichen, wodurch NTT-Serverless-CI/CD weiterhin genutzt werden könnte.

4.6 NTT-Serverless-CI/CD Ausblick

Voraussichtlich wird NTT-Serverless-CI/CD in einer Community-Version im ersten Halbjahr 2021 veröffentlicht. Es befindet sich aktuell in einem firmeninternen Veröffentlichungsprozess und wird voraussichtlich im AWS-Marketplace oder GitHub veröffentlicht. Intern bei

NTT DATA Deutschland GmbH findet es immer weitere Verbreitung insbesondere in der Enterprise-Version. Darauf beruhend, ist es sehr wahrscheinlich, dass NTT-Serverless-CI/CD noch weitere Funktionalitäten erhalten wird und zumindest teilweise weiterentwickelt wird. Ob die Weiterentwicklung nur auf Anforderungen beruhen wird, welche durch NTT DATA Deutschland GmbH erhoben werden oder ob es eine Community geben wird, welche die Entwicklung treibt, kann zum jetzigen Zeitpunkt nicht gesagt werden.

Mit der Veröffentlichung von NTT-Serverless-CI/CD werden bereits einige neue Funktionen Einzug in das Framework finden.

Terraform Terraform wird als Alternative zur CloudFormation verwendet werden können.

Dynamische Anzahl an Child-Konten Es soll möglich sein, nicht mehr auf die drei Child-Konten angewiesen zu sein. Bereits bei der Veröffentlichung soll es möglich sein, die Child-Konten-Anzahl bei der Initialisierung von NTT-Serverless-CI/CD festzulegen, wodurch es möglich sein wird, auch mehrere Integrationsumgebungen zu nutzen.

Unterschiedliche Pipelines Es soll möglich sein, unabhängige Pipelines für Integration und Produktion zu verwenden, wodurch höhere Parallelität und Varianz gegeben sein soll.

5 Verwendung von NTT-Serverless-CI/CD

Für die Erprobung von NTT-Serverless-CI/CD werden zwei GitHub-Repositories erstellt, diese beinhalten zwei Beispielprojekte. Bei diesen Beispielprojekten handelt es sich um einen Microservice und ein Web-Applikation. Beide Beispielprojekte werden unabhängig voneinander in verschiedenen Programmiersprachen entwickelt und verwenden eine unterschiedliche Infrastruktur. Dennoch sollen beide Beispielprojekte miteinander über REST interagieren und nach den Anforderungen aus Kapitel 1.3 unabhängig voneinander veröffentlicht werden. Durch diese beiden Beispielprojekte wird NTT-Serverless-CI/CD erprobt, sowie Fehler und Schwierigkeiten von NTT-Serverless-CI/CD aufgedeckt.

Grundlagen bzw. Anforderungen für beide Beispielprojekte stammen aus der Automotive-Industrie. Genauer werden die Anforderungen an einen Proof-of-Concept angelehnt und für diese Arbeit geringfügig adaptiert und erweitert. Der Fokus liegt nicht auf dem Funktionsumfang der Beispielprojekte, auch wenn diese den Funktionsumfang erfüllen, sondern auf technischer Machbarkeit, technischem Durchstich durch alle Schichten des 3-Schichten-Models und Evaluierung.

Der Quell-Code für die Infrastruktur, die Pipeline und den Build-Prozess der beiden Beispiele kann unter dem Git-Repository[1] dieser Arbeit abgerufen werden. Der Quell-Code, welcher nicht mit NTT-Serverless-CI/CD oder der Infrastruktur in Verbindung steht wird nicht veröffentlicht, da er für diese Arbeit nicht relevant ist.

5.1 Beispiel „Microservice“

Bei dem Beispiel Microservice handelt es sich um eine einfache Java-Applikation welche Java 11 und Apache Maven[2] verwendet. Die Funktionalität beschränkt sich auf CRUD-Operationen[3], welche über eine REST-Schnittstelle angesprochen werden und mit Hilfe einer Datenbank beantwortet bzw. Datenänderungen abgespeichert werden. Für die Containerisierung wurden Quarkus[4] und Docker[5] verwendet.

Die Zielarchitektur, welche mit dem Beispiel Microservice angestrebt wird, kann Abbildung 5.1 entnommen werden. So wird nach der Zielarchitektur eine PostgreSQL-Datenbank mit RDS betrieben. Diese Datenbank ist mit den Container-Instanzen verbunden, welche von Fargate in einem ECS-Cluster verwaltet wird. Die Anfragenverteilung auf die Container-Instanzen findet durch einen Application-Load-Balancer statt, welcher wiederum durch ein API-Gateway angesprochen wird.

Es werden noch weitere Adaptionen des ursprünglichen Proof-of-Concept für das Beispiel Microservice vorgenommen. So wurde die Infrastruktur in den drei Child-Umgebungen nicht erhalten, sondern zerstört, während nicht damit gearbeitet wurde. Da die Infrastruktur bei

1 https://github.com/dominikampletzer/CI-CD-Verfahren-im-Serverless-Cloud-Computing/tree/master/Beispiele
2 https://maven.apache.org/
3 CRUD Acronym für engl. **C**reate **R**ead **U**pdate **D**elete
4 https://quarkus.io/
5 https://www.docker.com/

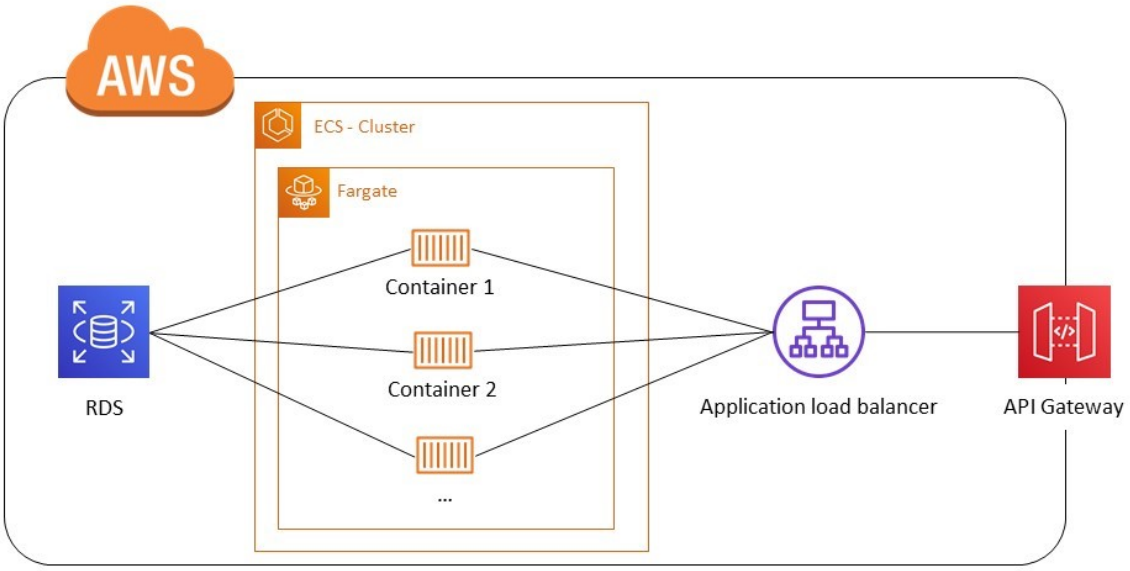


Abbildung 5.1 Architektur des Beispiels Microservice

Nichtbenutzung zerstört wird, muss für die Datenbank eine Lösung gefunden werden, wie diese einen konsistenten Zustand bei der Erstellung aufweist. Hierfür wurde ein Datenbank-Snapshot erstellt, welcher eine kleine Menge Datensätze enthält. Dieser wurde über die RDS-Snapshot-Teilen-Funktion mit jedem der verwendeten AWS-Konten geteilt. Dies entspricht keinem realen Vorgehen, sollte aber für das Beispiel ausreichend sein. Außerdem wird, um die in der Praxis bestehende Basis-Infrastruktur zu simulieren und Wartezeiten gering zu halten, auf jeder Child-Umgebung bereits ein VPC-Stack und eine ECS-Cluster-Stack ausgeführt, welche Bestand haben.

Um die Infrastruktur mit NTT-Serverless-CI/CD umzusetzen wird für das Beispiel Microservice die Infrastruktur in mehrere CloudFormation-Templates unterteilt. Bei dem Beispiel Angular-Applikation wird ein CloudFormation-Template verwendet. Dadurch werden beide Vorgehen getestet und evaluiert.

Abbildung 5.2 zeigt eine detaillierte Architektur von NTT-Serverless-CI/CD in Interaktion mit dem Beispiel Microservice. Unterschiede in den Zielumgebungen Development, Integration und Production gibt es für das Beispiel Microservice nicht, sind jedoch in der Praxis neben unterschiedlichen Konfigurationen möglich.

Die komplette Pipeline mit Infrastruktur wird, wie von AWS als Best-Practice bezeichnet, in der pipeline.yaml definiert und befindet sich im *root*/build-Ordner. Diese pipeline.yaml beinhaltet, neben der Pipeline im CloudFormation-Format, ein ECR in welchem das in der Build-Stage erzeugte Docker-Image gespeichert wird. Außerdem wird in der pipeline.yaml ein CodeBuild-Projekt, welches den Build des Microservice beinhaltet, definiert. Zusätzlich werden noch CustomRessources definiert, welche die DynamicCleanUpLambda anstoßen. Diese DynamicCleanUpLambda ist dafür verantwortlich, dass nach dem Entfernen des Feature-Branches, die Development-Infrastruktur entfernt wird.

Zusätzlich gibt es noch eine CustomResource, um die erstellten Docker-Images im ECR

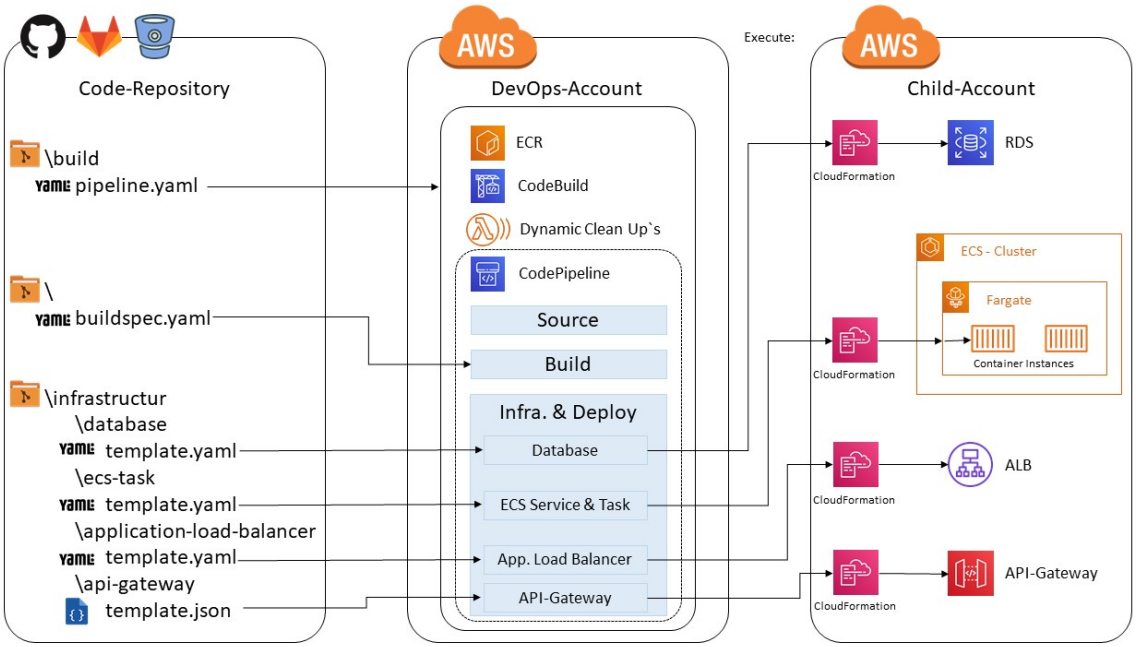


Abbildung 5.2 Microservice NTT-Serverless-CI/CD Architektur

für die DEV-Stage zu entfernen. Dies wird ebenfalls beim Entfernen des Feature-Branches ausgelöst. Bei den CleanUp-Custom-Resources ist die Reihenfolge in der die Ressourcen entfernt werden wichtig. So kann z.B. das ECR nicht gelöscht werden, wenn es noch Docker-Images enthält. Daraus folgt, um das ECR zu löschen, müssen zuerst die beinhalteten Docker-Images gelöscht werden. Um für das Entfernen des Stacks eine Abhängigkeit zu implementieren, muss die *DependsOn*-Eigenschaft in CloudFormation verwendet werden. Diese *DependsOn*-Eigenschaft ermöglicht es, eine oder mehrere Ressourcen anzugeben, welche vor der Erstellung der eigentlichen Ressource erstellt werden müssen, was beim Entfernen des Stacks die umgekehrte Reihenfolge zur Folge hat.

5.1.1 Source & Build

Über die Source-Stage gibt es nichts Spezielles zu erwähnen. Lediglich die Integration mit GitHub findet über OAuth statt und hierfür werden die Zugangsdaten über den SecretsManager bezogen bzw. sind in diesem gespeichert.

```
env:
  variables:
    ...
phases:
  install:
    runtime-versions:
      java: openjdk11
  pre_build:
    commands:
      ...
      - aws ecr get-login-password --region eu-central-1 | docker login --username
    AWS --password-stdin ${DEVOPS_ACCOUNT_ID}.dkr.ecr.eu-central-1.amazonaws.com
```

```
12      - REPOSITORY_URI=${DEVOPS_ACCOUNT_ID}.dkr.ecr.eu-central-1.amazonaws.com/${
    ECR_NAME}
13    build:
14      commands:
15        - mvn clean install
16      ...
17        - |
18            make -f locator/Makefile build-to-jvm-docker-aws
19            docker tag ${REPOSITORY}-${SUFFIX}-preview-image:${SUFFIX}
    $REPOSITORY_URI:${SUFFIX}
20            docker push $REPOSITORY_URI:${SUFFIX}
21  artifacts:
22    ...
```

Code 5.1 Microservice buildspec.yaml gekürzt

Die Build-Stage bezieht die Konfiguration aus *root*/buildspec.yaml und ist in Code 5.1 als gekürzte Version abgebildet. In Code 5.1 sind folgende Zeilen erwähnenswert:

Zeile 11 Bevor das erzeugte Docker-Image im ECR gespeichert werden kann, muss eine Anmeldung stattfinden.

Zeile 18 Das Build-Management-Werkzeug *make* befindet sich auf den regulären AWS-Linux-Build-Docker-Images und kann verwendet werden, um die Build-Komplexität in der buildspec.yaml gering zu halten und durch das Auslagern in die Makefile-Datei die *make*-Befehle auch für die Entwicklung verfügbar zu machen.

Zeile 20 Das erzeugte und mit einem Tag versehene Docker-Image wird in das ECR gespeichert, für welches bereits eine Anmeldung erfolgte.

CodeBuild und die buildspec.yaml bieten noch weitere Möglichkeiten und Phasen. Hierfür kann und sollte die sehr gute Dokumentation von AWS[6] herangezogen werden. Ein Beispiel wie Unit-Tests in die buildspec.yaml integriert werden, ist in Kapitel 5.2 zu finden. Es ist auch möglich eine buildspec.yaml nur für Tests zu verwenden und hierfür eine eigene Test-Stage in die Pipeline zu integrieren.

5.1.2 Infrastructure & Deploy

Im Beispiel Microservice ist die Unterscheidung zwischen den Stages, welche die Infrastruktur erzeugen und der Stage, welche für das Veröffentlichen zuständig ist, nicht klar getrennt. Dies und weitere interessante Punkte sind in Code 5.2 und Code 5.3 zu entnehmen.

```
1  Parameters:
2    ...
3  Conditions:
4    ShouldCreateForDelivery: !Equals [ !Ref Stage, "delivery" ]
5    ShouldCreateForDev: !Equals [ !Ref Stage, "dev" ]
6  Resources:
7    ECRRepository:
8      ...
9    CodeBuild:
10     ...
11   CleanupDB:
12     Condition: ShouldCreateForDev
```

6 https://docs.aws.amazon.com/codebuild/latest/userguide/build-spec-ref.html

```
13      Properties:
14      ServiceToken: !Ref DynamicPipelineCleanupLambdaArn
15        StackName: !Sub "postgres-db-${Stage}-${Suffix}"
16  ...
```

Code 5.2 Microservice pipeline.yaml gekürzt Teil 1

Code 5.2 zeigt folgendes:

Zeile 3 Die Conditions werden für die Unterscheidung der zwei Pipelines DEV und DELIVERY benötigt. Dies ermöglicht eine verständliche Unterscheidung innerhalb des CloudFormation-Templates

Zeile 12 Anwendungsbeispiel der Conditions. Zu sehen ist, wie die Ressourceneigenschaft Conditions verwendet werden kann. Hier wird die Ressource nur erstellt, wenn die Condition TRUE/Wahr ist. Es ist nicht möglich mit der Condition in der Condition-Ressourceneigenschaft eine IF-ELSE-Struktur umzusetzen. Soll für den IF-Zweig die Ressource erstellt werden und im ELSE-Zweig nicht, wird eine Verneinung der Condition benötigt, z.B. mit einer isDEV- und einer isNotDEV-Condition.

Zeile 11-15 Beinhaltet ein Beispiel wie die DynamicCleanupLambda nur für die DEV-Pipeline mit einer Ressource verbunden werden kann. In diesem Fall mit der Datenbank, welche durch StackName erkannt wird. Dies zeigt auch, wie fragil die DynamicCleanupLambda mit den bestehenden Stacks verbunden ist. Es wird lediglich die Bezeichnung des Stacks übergeben. Aus diesem Grund sollte eine lesbare, verständliche und eindeutige Konvention für die Bezeichnung von Stacks gewählt werden

```
1  ...
2    Pipeline:
3      Type: AWS::CodePipeline::Pipeline
4      Properties:
5        ...
6        Stages:
7          - Name: "Checkout-SourceCode"
8            ...
9          - Name: "Build"
10           ...
11         - Name: !If
12             - ShouldCreateForDelivery
13             - 'DeployInfrastructureToInt'
14             - 'DeployInfrastructureToDev'
15           Actions:
16             - Name: DeployALB
17               ActionTypeId:
18                 ...
19               Configuration:
20               RunOrder: 1
21               RoleArn: !Sub "arn:aws:iam::${RemotePreviewAccount}:role/
     CodePipelineServiceRole-${AWS::Region}-${RemotePreviewAccount}"
22                 StackName: !Sub elb-${Stage}-${Suffix}
23                 TemplatePath: "SourceArtifact::infrastructure/application-load-
     balancer/template.yaml"
24                 ParameterOverrides:
25                   Fn::Sub: |
26                     {
27                       "Project": "${Project}",
```

```
28                      "Repository": "${Repository}",
29                      "Stage": "${Stage}",
30                      "Suffix" : "${Suffix}",
31                      "VpcStackName": "base-vpc-${Stage}-master"
32                    }
33              ...
34            - Name: DeployDB
35              RunOrder: 1
36              ...
37            - Name: DeployEcsTaskAndService
38              RunOrder: 2
39              ...
40            - Name: DeployAPIGateway
41              RunOrder: 2
42              ...
43        - !If
44          - ShouldCreateForDelivery
45          - Name: "Prod-Approve"
46            ...
47        - !If
48          - ShouldCreateForDelivery
49          - Name: 'DeployInfrastructureToProd'
50 ...
```

Code 5.3 Microservice pipeline.yaml gekürzt Teil 2

Die AWS-Ressource von CodePipeline ist in Code 5.3 zu finden. Der Code zeigt unter anderem Folgendes auf:

Zeile 11-14 Anwendung der Condition in einer IF-Schleife.

Zeile 21 Zeigt auf wie die Rollen für den jeweiligen Dienst übergeben werden. Hier ist der Parameter *RemotePreviewAccount* zu sehen, welcher abhängig von der Stage die Account-ID des Development- oder des Infrastructure-Kontos beinhaltet.

Zeile 22 StackName beschreibt den Namen des Stacks, wie er auf den Child-Konten erzeugt und ausgeführt wird. Es sollte hier auf verständliche und lesbare Bezeichnungen geachtet werden, da diese Bezeichner für die DynamicCleanupLambda benötigt wird. Siehe Code 5.2 Zeile 15.

Zeile 23-31 Beschreibt den Pfad, auf welchen im Quell-Code das benötigte CloudFormation-Template gefunden werden kann und beinhaltet die Parameter die an das CloudFormation-Template übergeben werden. Hierbei ist auf die zu verwenden StackNamen zu achten, da diese für Referenzen benötigt werden können. Man sieht hier ebenfalls eine Einschränkung, die dem YAML unterliegt. Es gibt Funktionalitäten, welche nicht in YAML ausgedrückt werden können bzw. im YAML-Format von CloudFormation nicht akzeptiert werden. Dieser Fall ist z.B. ParameterOverrides. Es ist nötig für ParameterOverrides einen String zu übergeben, welcher das JSON-Format erfüllt, was die Verständlichkeit stark erschwert. *RemoteDeliveryAccount* wird nur für das Production-Konto verwendet und ist im DEV-Pipeline Fall nicht belegt.

Zeile 20, 35, 38 & 41 Diese geben die Reihenfolge an, in der die Aktionen/Aufgaben innerhalb einer Stage ablaufen. Damit ist es möglich, Abhängigkeiten abzubilden oder Parallelität zu ermöglichen.

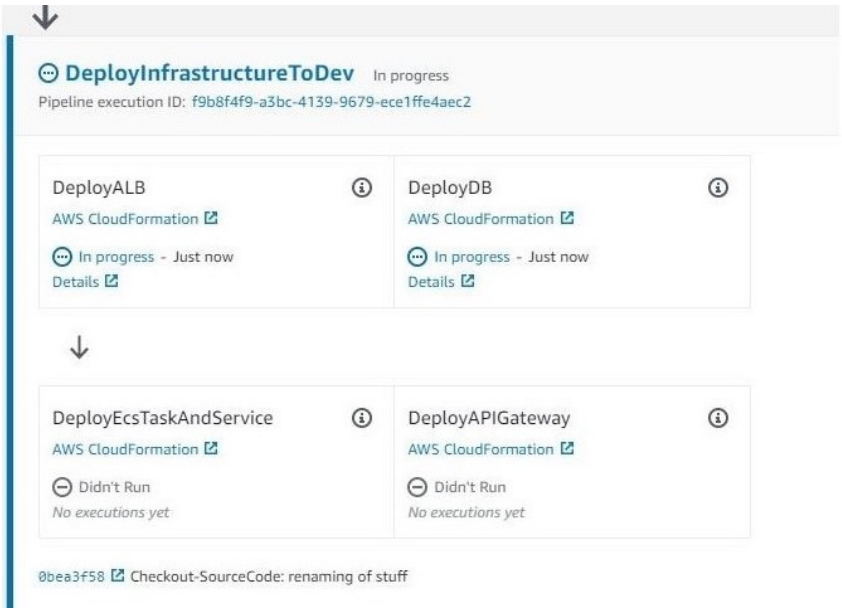


Abbildung 5.3 CodePipeline: Deploy und Infrastruktur in DEV

Zeile 39-49 Beinhaltet neben den manuellen Bestätigungsschritt im Grunde den gleichen Code wie bereits für die Infrastruktur von Development und Integration, nur mit der Konfiguration für Production. So wird z.B. für die RoleArn anstatt dem Parameter *RemotePreviewAccount* der Parameter für Production *RemoteDeliveryAccount* verwendet.

Um das Zusammenspiel von Code 5.3 und Abbildung 5.2, was die Infrastruktur betrifft zu verdeutlichen, muss Code 5.3 Zeile 23 betrachtet werden. Dieser ist zu entnehmen, dass die Infrastruktur aus dem CloudFormation-Template im SourceArtifact\infrastructure\application-load-balancer-Ordner liegt. Hierbei beinhaltet das SourceArtifact den ganzen Quell-Code, wie er aus der Source-Stage erhalten wurde, wodurch das SourceArtefact als *root* interpretiert werden kann. Dieses CloudFormation-Template wird durch das Dev-Ops-Konto auf dem jeweiligen Child-Konto ausgeführt und erzeugt die Infrastruktur wie in Abbildung 5.2 dargestellt.

Abhängigkeiten und Synchronität der Infrastruktur-Erzeugung können durch die *RunOrder* in der pipeline.yaml durch einen numerischen Wert festgelegt werden. Somit können logische Abhängigkeiten zwischen Infrastruktur-Ressourcen, wie im Code 5.3 zu sehen, abgebildet werden. Es muss z.B. der Application-Load-Balancer erzeugt werden, bevor das API-Gateway erzeugt wird, welches an den Application-Load-Balancer Anfragen weiterleitet. Dies liegt daran, dass die IP-Adresse des Application-Load-Balancers dynamisch vergeben wird. Würde das API-Gateway vor oder zeitgleich mit dem Application-Load-Balancer erzeugt werden, könnte nicht bestimmt werden, an welche IP-Adresse das API-Gateway die Anfragen weiterleiten soll. Das gleiche Prinzip trifft auch auf die Datenbank und die Container-Instanzen zu. Ohne Datenbank, ist nicht klar, wie die Container-Instanzen Datenbankverbindungen aufbauen können.

Abbildung 5.3 zeigt die Darstellung der Synchronität und der Abhängigkeit. Diese Darstellung ist der AWS-Management-Konsole für die DEV-Pipeline entnommen. Die Synchronität ist daran zu erkennen, dass die Aufgaben nebeneinander dargestellt sind, die Abhängig-

keit daran, dass die Aufgaben untereinander mit einem Pfeil verbunden sind. Weiter ist dem Bild zu entnehmen, dass die Stage mit dem Namen DeployInfrastructureToDev sowie die Unterpunkte DeployALB und DeployDB ausgeführt werden. Die beiden Unterpunkte DeployEcsTaskAndService und DeployAPIGateway jedoch noch nicht.

5.1.3 Weitere Besonderheiten

Auf einige Besonderheiten des Beispiels wird in diesem Abschnitt eingegangen, da dies für unerfahrene Entwickler/innen von Relevanz sein kann, welche sich am Beispiel Microservice orientieren. Es wird jedoch nur exemplarisch aufgezeigt, worauf zu achten ist, da auf die Anforderungen der jeweiligen umzusetzenden Software eingegangen werden muss und diese sich vom Beispiel Microservice unterscheiden können. Der zu Grunde liegende Quell-Code wurde der Übersicht halber nicht in der Arbeit hinzugefügt, kann jedoch öffentlich im Git-Repository [7] gefunden werden.

Dem ECR wurde eine Lifecycle-Richtlinie hinterlegt. Durch diese können nicht benötigte Container-Abbildungen entfernt werden um Kosten gering zu halten. Die verwendete Lifecycle-Richtlinie ist im Beispiel Microservice sehr kurz gewählt und für die Development-Umgebung im Grunde obsolet, da die DeleteECRImages-Lambda verwendet wird, um Infrastruktur inkl. Container-Abbildungen zu entfernen. Für Integration und Production hingegen ist die Lifecycle-Richtlinie zu empfehlen, da ohne diese das ECR mit jedem erfolgreichen Build wächst, wodurch immer mehr Speicher benötigt wird und somit immer mehr Kosten entstehen.

Um das Beispiel Microservice einfach zu halten, wird der Sicherheitsaspekt, so wenig wie möglich öffentlich zugänglich zu machen, nicht berücksichtigt. Die Container-Instanzen werden anstatt in einem privaten VPC in einem öffentlichen VPC betrieben. Dies ermöglicht es ECS der Child-Konten direkt auf ECR des DevOps-Kontos zuzugreifen, natürlich erst, nachdem die benötigten Rechte vergeben wurden. Es wäre sogar möglich, nicht nur die Container-Instanzen, sondern auch den Application-Load-Balancer in einem privaten VPC zu betreiben, da Anfragen über das öffentliche API-Gateway zu einem privaten Application-Load-Balancer an private Container-Instanzen weitergeleitet werden könnten.

Um diese Architektur zu implementieren, kann nach Kantsev VPC Peering genutzt werden, wodurch VPCs direkt kommunizieren ohne Dienste oder Endpunkte öffentlich zugänglich zu machen [Kan17][8]. Alternativ hierzu ist es auch möglich, mittels eines NAT-Gateways und eines VPC-Endpoints aus einem privaten VPC Internet-Zugang zu erhalten und private Dienste aufzurufen.

Weiter kann die Architektur geändert werden, um dieses Problem nicht entstehen zu lassen. Anstatt die Container-Abbildungen im ECR des DevOps-Kontos abzulegen, könnten diese auch in den Child-Konten abgelegt werden. Dies hätte jedoch zur Folge, falls die Konfiguration des Containers zur Laufzeit erfolgen kann, dass Redundanzen zumindest in Integration und Production entstehen. Dies kann, abhängig von den Anforderungen, durchaus eine sinnvolle Lösung darstellen.

Um einen statischen Endpunkt für aufrufende Applikationen zu schaffen, sollte dem durch die Pipeline erstellten API-Gateway ein Custom-Domain-Name zugewiesen werden. Durch Route53 ist es möglich, Anfragen auf diesen Custom-Domain-Namen weiterzuleiten. Im Beispiel Microservice wurde auf diesen statischen Endpunkt verzichtet, sowie komplett

7 `https://github.com/dominikampletzer/CI-CD-Verfahren-im-Serverless-Cloud-Computing/tree/master/Beispiele/Microservice`

8 Kapitel: VPC security

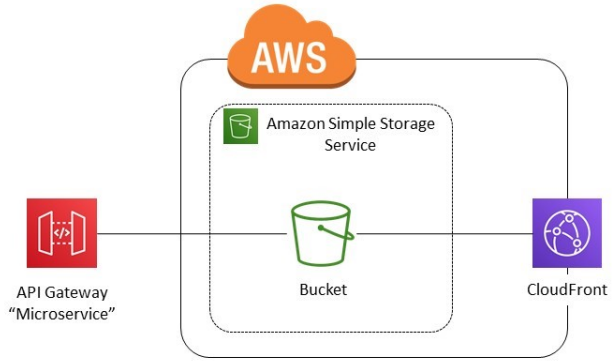


Abbildung 5.4 Architektur Beispiel Angular-Applikation

auf die Integration von Route53, da es sich um ein gelöstes Problem handelt, welches NTT-Serverless-CI/CD nicht beeinflusst. Eine ausführliche Dokumentation ist bei AWS [9] zu finden

5.2 Beispiel „Angular-Applikation“

Im Gegensatz zum Beispiel Microservice, ist die Architektur des Beispiel Angular-Applikation einfach gehalten. Abbildung 5.4 zeigt diese Architektur. Die kompilierten Artefakte befinden sich in einem S3-Bucket und werden mittels einer CloudFront-Distribution ausgeliefert. Dadurch ist der S3-Bucket vor dem Zugriff aus dem Internet über CloudFront geschützt. Die Artefakte wiederum verweisen auf das API-Gateway des Beispiel Microservice. Aus Kostengründen wurde für dieses Beispiel keine Domain erworben. In der Praxis sollte jedoch Route53 verwendet werden und mit CloudFront verbunden werden, um eine statische und ansprechende Domäne im Internet zu besitzen. Es ist möglich, eine Domäne bzw. Unterdomänen für die einzelnen Child-Konten zu verwenden, deshalb muss nicht für jede Umgebung eine eigene Domäne erworben werden.

Da es sich im Beispiel Angular-Applikation auch um die Umsetzung von NTT-Serverless-CI/CD handelt, ist es nicht verwunderlich, dass die Architektur ähnlich dem Beispiel Microservice ist. Abbildung 5.5 zeigt jedoch keinen feingranularen Ansatz zum Erstellen der Infrastruktur. Die Reihenfolge, in der die Infrastruktur erstellt wird, ist CloudFormation überlassen. Dies ist möglich, indem die benötigte Infrastruktur in einem CloudFormation-Template erzeugt wird und auf einmal auf dem Child-Konto ausgeführt wird.

5.2.1 Source & Build

Die Source-Stage unterscheidet sich nicht von der im Beispiel Microservice.

Der Build wurde auf zwei CodeBuild-Projekte aufgeteilt, wie Code 5.4 Zeile 18 & 35 zeigen. Dies hat den Hintergrund, dass Angular-Applikationen zur Build-Zeit Konfigurationen übergeben werden. Dies betrifft auch den Endpunkt des API-Gateways. So ist in Zeile 3-9 eine Map zu sehen, welche verwendet wird, um die unterschiedlichen Konfigurationen zu

9 `https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-to-api-gateway.html`

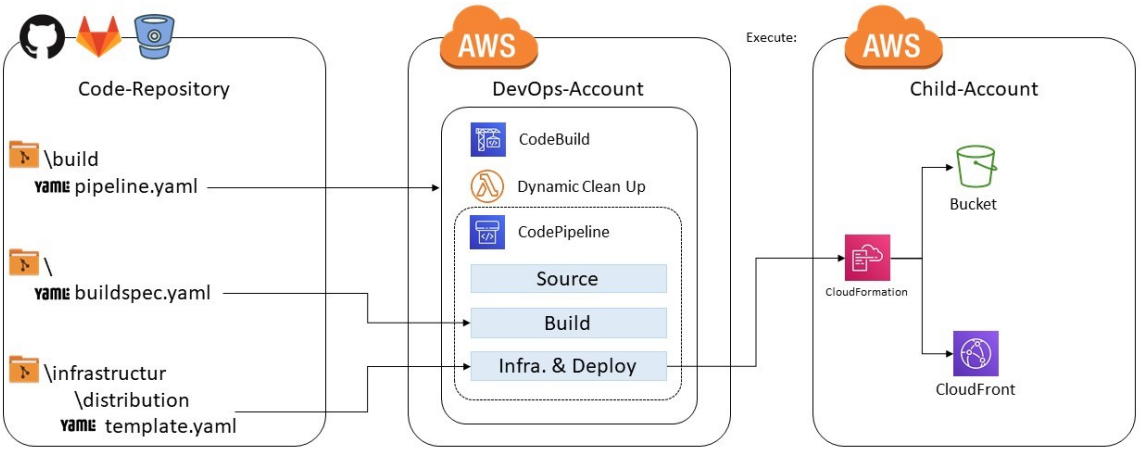


Abbildung 5.5 Angular-Applikation NTT-Serverless-CI/CD Architektur

verwenden. Die Zeichenketten *dev*, *int* und *production* können dabei frei gewählt werden, müssen jedoch mit der Bezeichnung in der Angular-Applikation übereinstimmen.

```
...
Mappings:
  StageToConfigMap:
    dev:
      config: "dev"
    int:
      config: "int"
    prod:
      config: "production"
  ApiUrlMap:
    dev:
      url: "http://<some-development-url>"
    int:
      url: "http://<some-integration-url>"
    prod:
      url: "http://<some-production-url>"
Resources:
  CodeBuild:
    Type: AWS::CodeBuild::Project
    Properties:
      Name: !Ref AWS::StackName
      ...
        EnvironmentVariables:
          - Name: CONFIGURATION
            Value: !If
              - ShouldCreateForDelivery
              - !FindInMap [ StageToConfigMap, "int", "config" ]
              - !FindInMap [ StageToConfigMap, "dev", "config" ]
          - Name: APIDNS
            Value: !If
              - ShouldCreateForDelivery
              - !FindInMap [ ApiUrlMap, "int", "url" ]
              - !FindInMap [ ApiUrlMap, "dev", "url" ]
      ...
  CodeBuildProd:
```

```
36      Condition: ShouldCreateForDelivery
37      Type: AWS::CodeBuild::Project
38      Properties:
39        Name: !Join [ "-", [ !Ref AWS::StackName, "prod" ] ]
40        ...
41          EnvironmentVariables:
42            - Name: CONFIGURATION
43              Value: !FindInMap [ StageToConfigMap, "prod", "config" ]
44            - Name: APIDNS
45              Value: !FindInMap [ ApiUrlMap, "prod", "url" ]
46  ...
```

Code 5.4 Angular-Applikation pipeline.yaml gekürzt Teil 1

Für die Übergabe der API-Gateway-URL wurden statische Zeichenketten verwendet, da diese für das Beispiel ausreichend sind und in einer Praxis-Software mit einer statischen Domäne über Route53 auch verfahren werden kann. Alternativ kann, um die URL dynamisch zu beziehen, ein exportierter Wert vom zugehörigen Back-End-Stack verwendet werden. In diesem Fall müsste im Beispiel Microservice im CloudFormation-Template für das API-Gateway ein Output definiert werden und auf dieses von der pipeline.yaml des Beispiels Angular-Applikation referenziert werden. Dies hat den Seiteneffekt, dass eine Kopplung zwischen den Beispielen entsteht.

```
1  ...
2  env:
3    variables:
4      CONFIGURATION: "get value from pipeline-cloudformation"
5      APIDNS: "get value from pipeline-cloudformation"
6  phases:
7    ...
8    build:
9      commands:
10       ...
11       - ./node_modules/.bin/ng test --watch=false --browsers
       ChromeHeadlessNoSandbox
12       - echo {\"basepath\":\"${APIDNS}\"} > src/environments/basepath.json
13       - ./node_modules/.bin/ng build --configuration=${CONFIGURATION}
14 ...
```

Code 5.5 Angular-Applikation buildspec.yaml gekürzt

Code 5.5 zeigt die gekürzte buildspec.yaml. Diese buildspec.yaml hat folgende interessante Stellen:

Zeile 11 Beispiel, wie Unit-Tests integriert werden können. Die Konfiguration ist angular-spezifisch und der Browser befindet sich nicht im Build-Docker-Image, sondern wurde über die Dev-Dependencies installiert.

Zeile 12 Zeigt eine Möglichkeit wie dynamisch eine Konfigurationsdatei im JSON-Format erstellt wird. In diesem Fall wird die URL des API-Gateways gesetzt und von der Angular-Applikation für alle REST-Aufrufe verwendet.

Zeile 13 Zeigt wie die zuvor in der pipeline.yaml festgelegte Konfigurationsbezeichnung der Angular-Applikation übergeben wird. Hierbei handelt es sich ebenfalls um angular-spezifisches Verhalten.

5.2.2 Infrastructure & Deploy

Wie die Abbildung 5.4 zeigt, ist die Architektur des Beispiels Angular-Applikation recht einfach. Demnach ist auch die Infrastructure-Stage einfach gehalten. Code 5.6 Zeile 11 zeigt wie ein CloudFormation-Template verwendet wird, um die Infrastruktur auf dem Child-Konto zu erzeugen. Dieses CloudFormation-Template ist, wie bei dem Beispiel Microservice, unter *root*/infrastructure zu finden. Bei dieser Ordnerstruktur handelt es sich um eine Konvention, welche für die Beispiele eingeführt wurde. Es ist empfehlenswert, eine ähnliche Konvention zu verwenden. Das CloudFormation-Template erzeugt bei Verwendung einen S3-Bucket, welcher die kompilierte Angular-Applikation und eine CloudFront-Distribution enthält. CloudFront ist der Content-Delivery-Service (CDN) von AWS und bietet Dienste, wie z.B. Caching und DDOS-Schutz.

```
1  ...
2    Pipeline:
3      Type: AWS::CodePipeline::Pipeline
4      Properties:
5        ...
6              - Name: !If
7                  - ShouldCreateForDelivery
8                  - 'DeployS3AndCloudFrontToInt'
9                  - 'DeployS3AndCloudFrontToDev'
10               ...
11                 TemplatePath: "SourceArtifact::infrastructure/distribution/
     template.yaml"
12             - ...
13               ActionTypeId:
14                 Category: "Deploy"
15                 ...
16               Configuration:
17                 BucketName: !Join [ '-', [ !Ref Stage, !Ref Repository, !Ref
     Suffix, !Ref RemotePreviewAccount ] ]
18                 Extract: true
19               ..
20     ...
21   CleanupS3Bucket:
22     Condition: ShouldCreateForDev
23     DependsOn: CleanupS3StackAndCloudFrontStack
24     Type: "AWS::CloudFormation::CustomResource"
25     Properties:
26       ServiceToken: !ImportValue S3CleanupLambdaStack:Arn
27       BucketName: !Sub "${Stage}-${Repository}-${Suffix}-${RemotePreviewAccount}"
28 ...
```

Code 5.6 Angular-Applikation pipeline.yaml gekürzt Teil 2

Die Veröffentlichung besteht aus dem Schritt, das erzeugte Artefakt zu entpacken und in einem Bucket abzulegen. Dies ist in Zeile 17 und 18 zu sehen. Das Ablegen des entpackten Artefakts in einem Bucket ist ausreichend und der direkte Zugriff aus dem Internet ist durch CloudFront geschützt. Somit kann der Bucket privat sein und benötigt keinen öffentlichen Zugang.

Das Beispiel Angular-Applikation besitzt noch eine CloudFront-CustomRessource, welche in Abbildung 5.4 nicht erscheint. Diese CustomRessource ist in Zeile 21-27 zu sehen und wird benötigt, um eine Lambda anzubinden, welche bei Entfernen der Infrastruktur den Inhalt des Buckets entfernt. Ohne das Entfernen des Inhalts eines Bucket, ist es nicht möglich, diesen zu entfernen. Hierfür wurde, ähnlich wie bereits bei der Basis-Infrastruktur, ein CloudFormation-

Stack ausgeführt, welcher eine Lambda erzeugt, die Bucket-Inhalte entfernt. Außerdem wurde eine IAM-Richtlinie erstellt, welche das Ausführen dieser Bucket-CleanUp-Lambda ermöglicht, sowie den IAM-Rollen *CI-CD-Bootstrap-PipelineM-PipelineMakerCodeBuildRo** und *CI-CD-Bootstrap-DevOpsLam-DynamicPipelineMaker-Lamb** zugewiesen ist. Der Bucket-CleanUp-Stack wurde nur auf dem Development-Konto ausgeführt, da die Infrastruktur auf dem Development-Konto regelmäßig zerstört wird. Integration und Production sollen Bestand haben und werden aus Sicherheitsgründen nicht automatisiert zerstört.

Die Bindung der Bucket-CleanUp-Lambda und des Buckets an die CustomRessource ist ebenfalls hervorzuheben. Die Bucket-CleanUp-Lambda wird mithilfe der CloudFormation Import-Funktionalität an die CustomRessource gebunden. Dies hat zur Folge, dass der Cleanup-Lambda-Stack eng an den Pipeline-Stack gebunden ist. Dies Bindung sorgt dafür, dass es nicht möglich ist, den Cleanup-Lambda-Stack zu entfernen oder den Zugriff auf den importieren Wert zu ändern, ohne vorher den Pipeline-Stack zu entfernen. Zum Bucket hingegen ist die Bindung sehr fragil. Diese wird durch den Bucket-Namen in Form einer dynamisch erzeugten Zeichenkette hergestellt.

5.2.3 Weitere Besonderheiten

Ein Optimierungsmöglichkeit, welche in beiden Beispielen nicht berücksichtigt ist, ist die Verwendung eines eigenen CodeBuild-Docker-Images. Dieses eigene Docker-Image wird i.d.R. nur in speziellen Fällen benötigt, z.B. wenn die benötigte Build-Umgebung nicht in AWS verfügbar ist oder um Software bereits in der Build-Umgebung zur Verfügung zu haben. Dies kann bei Web-Applikationen der Fall sein. Um Unit-Tests oder End-to-End-Tests ausführen zu können, wird ein Browser benötigt. Server bzw. CodeBuild-Docker-Images besitzen i.d.R. keine Oberflächen und keinen Browser [Ros17][10]. Um dieses Problem zu lösen, kann ein eigenes CodeBuild-Docker-Image verwendet werden, welches die benötigten Browser beinhaltet.

Alternativ kann im Falle des fehlenden Browsers dieser während des Builds installiert werden, falls dies für die benötigten Browser möglich ist. Hierfür bieten sich, abhängig vom Browser, verschiedene Möglichkeiten. So kann z.B. über das Packet-Management-Werkzeug yarn der Chromium-Browser installiert werden oder dieser direkt als Dev-Dependency hinterlegt werden.

Die reine Installation der Browser führt jedoch nicht dazu, dass im Build-Prozess eine Oberfläche zur Verfügung steht. Moderne Browser verfügen jedoch über einen Headless-Mode, welcher es ermöglicht den Browser ohne Oberfläche zu starten und Tests auszuführen. Es sollte prinzipiell immer auf dem Zielbrowser und nur falls es nicht anders geht auf veraltetete JavaScript-Bibliotheken wie PhantomJS [11] zurückgegriffen werden. Dies liegt daran, dass das Verhalten von Browser zu Browser leicht unterschiedlich sein kann. Dies trifft verstärkt auf Simulationen von Browsern wie durch PhantomJS zu.

Das selbst erstellte Docker-Image, welches benötigte Software enthält, kann Zeit für die Installation und dadurch Kosten während des Build-Prozesses sparen. Jedoch dauert die Installation von z.B. Chromium, wie die Messung im Kapitel 7.2 Tabelle 7.1 zeigt, ~5 Sekunden. Ob dies den Aufwand eines Custom-Docker-Images rechtfertigt, ist fraglich. Abhängig vom/n Zielbrowser/n kann es jedoch sinnvoll sein und im Falle längerer Installationszeiten Kosten sparen.

10Kapiel: Headless browser testing
11 `https://github.com/ariya/phantomjs`

5.3 Integration

Dieses Kapitel beschreibt die einzelnen Schritte, welche nötig sind, um das NTT-Serverless-CI/CD-Framework zu verwenden. Das Kapitel kann als Erweiterung der noch unvollständigen Dokumentation von NTT-Serverless-CI/CD betrachtet werden. NTT-Serverless-CI/CD befindet sich noch in der Entwicklung und die vorhandene *ReadMe*-Datei ist noch nicht ausreichend für die Verwendung, was sich aber bis zur Veröffentlichung ändern wird. Es ist nicht möglich eine genaue Anleitung für jede Software, welche NTT-Serverless-CI/CD nutzen wird, zur Verfügung zu stellen. Als Referenz können jedoch die Beispielprojekte aus Kapitel 5 herangezogen werden. Dieses Kapitel ist als Leitfaden für die Integration des NTT-Serverless-CI/CD zu betrachten, jedoch nicht als umfassendes Kompendium für die Integration in jedweder Software, welche NTT-Serverless-CI/CD nutzen könnte. Im Anhang C.1 ist eine Check-Liste zu finden, welche in gekürzter Form alle nötigen Schritte enthält, um NTT-Serverless-CI/CD zu integrieren. Diese Check-Liste enthält zusätzliche Punkte, welche nicht in diesem Kapitel zu finden sind, jedoch aus dieser Arbeit hervorgehen.

Um das NTT-Serverless-CI/CD zu verwenden, ist die Grundvoraussetzungen, dass der CI/CD-Prozess mit AWS umgesetzt wird und der Best-Practice gefolgt wird, dass für jede Umgebung ein eigenes AWS-Konto verwendet wird. Daraus folgt, dass vier Konten anzulegen sind (Kapitel 4.6 ist zu entnehmen, dass eine unterschiedliche Anzahl an AWS-Konten möglich sein wird). Diese sind DevOps, Development, Integration und Production. Die Benennung ist hierbei nicht wichtig, jedoch der spätere Verwendungszweck. Weiter wird Administratorenberechtigung für diese AWS-Konten benötigt.

Rossel beschreibt die Kommandozeile (CLI) als wichtiges Werkzeug für CI/CD [Ros17][12]. Dies trifft nicht ganz auf NTT-Serverless-CI/CD zu, da als technische Voraussetzung nur das CLI-Werkzeug AWS-CLI[13] benötigt wird. Es ist jedoch auch das Build-Management-Tool *Make* zu empfehlen, da dieses die Integration von NTT-Serverless-CI/CD mit einem Befehl ermöglicht. Ohne *Make* muss jeder Befehl, der sonst mit *Make* ausgeführt wird, manuell über die CLI ausgeführt werden. Wie dies geschieht, wird hier nicht beschrieben, es kann und sollte hierfür jedoch das Makefile als Referenz herangezogen werden, da dieses alle nötigen Befehle beinhaltet.

Sind die Voraussetzungen von AWS und der CLI-Werkzeuge erfüllt, kann NTT-Serverless-CI/CD bezogen werden. Dies wird voraussichtlich durch den AWS-Marketplace oder GitHub möglich sein. Um NTT-Serverless-CI/CD in AWS einzurichten, sind folgende Schritte nötig:

1. Die AWS-Konten müssen mit Profilen versehen und die jeweiligen Zugangsdaten hinterlegt werden.[14]

2. In der Datei Makefile im Root-Verzeichnis muss die gewünschte AWS-Region, die AWS-Konto-IDs der erstellten AWS-Konten und die angelegten Profile, wie in Code 5.7 zu sehen, hinterlegt werden.

3. Nachdem die Datei Makefile angepasst wurde, kann diese im Root-Verzeichnis mit dem CLI-Befehl *make deployServelessCiCd* ausgeführt werden.

```
1 SHELL = /bin/bash
2
```

12 Kapitel: Automation
13 https://aws.amazon.com/de/cli
14 https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html

```
REGION = <AWS-REGION> # Bsp. eu-central-1

DEVOPS_ACCOUNT_ID = <AWS_DEVOPS_ACCOUNT_ID> # Bsp. 123456789987
DEV_ACCOUNT_ID = <AWS_DEVELOPMENT_ACCOUNT_ID>
INT_ACCOUNT_ID = <AWS_INTEGRATION_ACCOUNT_ID>
PROD_ACCOUNT_ID = <AWS_PRODUCTION_ACCOUNT_ID>

DEVOPS_PROFILE = <DEVOPS_PROFILE> # Bsp. MY_DEV_OPS
DEV_PROFILE = <DEVELOPMENT_PROFILE>
INT_PROFILE = <INTEGRATION_PROFILE>
PROD_PROFILE = <PRODUCTION_PROFILE>
```

Code 5.7 Makefile vorbereiten

Der in Schritt 3 ausgeführte *make deployServerlessCiCd* CLI-Befehl erzeugt in den AWS-Konten durch CloudFormation alle benötigten Ressourcen. Diese sind:

- IAM-Rollen und -Richtlinien (kontoübergreifend).
- S3-Buckets für CloudFormation-Templates, Lambda-Code und erzeugte Pipelines.
- CodeBuild-Projekt, um die Pipeline zu erzeugen => PipelineMaker.
- Ein API-Gateway für das Git-Management.
- Ein SSH-Schlüsselpaar, ebenfalls für das Git-Management.
- Schlüssel in KMS für sichere Datenübertragungen.

Die hierfür benötigten CloudFormation-Templates werden mit dem gepackten Lambda-Code nach S3 kopiert und ausgeführt bzw. eingerichtet.

Um in der Pipeline den Quell-Code aus der jeweiligen Git-Software zu erhalten, muss in dieser Git-Software ein Authorisierungs-Token hinterlegt/erzeugt werden. Dies ist ausführlich in Kapitel 4.3 beschrieben. Der in der Git-Software hinterlegte Token sollte im SecurityManager hinterlegt werden, um auf sichere Weise in der Pipeline Zugriff darauf zu haben. Des Weiteren sollte eine IAM-Richtlinie erstellt werden, welche nur Zugriff auf den SecurityManager-Token hat. Diese erzeugte IAM-Richtlinie muss den beiden Pipeline-Service-Rollen(*CI-CD-Bootstrap-PipelineRolesD-PipelineServiceRole**) und der Dynamischen-Pipeline-Maker-Rolle (*CI-CD-Bootstrap-DevOpsLam-DynamicPipelineMakerLamb**), welche durch NTT-Serverless-CI/CD erzeugt wurden hinzugefügt werden. Dadurch wird ein sicherer Zugriff des AWS-DevOps-Kontos auf die Git-Software ermöglicht.

Leider wurde während dieser Arbeit festgestellt, dass die Laufzeit der GitManagementLambda mit 3 Sekunden gelegentlich zu kurz bemessen ist. Dies kann dadurch behoben werden, dass der Time-Out-Wert erhöht wird. Hierfür muss in AWS-Lambda *CI-CD-Bootstrap-DevOpsGitManag-GitManagementLambda** der Time-Out-Wert auf z.B. 10 Sekunden geändert werden. Es ist allgemein bei der Verwendung von Time-Out darauf zu achten keine stark überhöhten Werte zu verwenden, da Prozesse bei zu langer Laufzeit in einem Fehlerzustand kommen sollen, um Warnungen zu versenden und Kosten gering zu halten.

Um den CI/CD-Prozess dynamisch mit der Git-Software zu verbinden, muss das API-Gateway über einen Webhook mit der Git-Software verbunden werden. Dies ist ebenfalls ausführlich in Kapitel 4.3 beschrieben. Die zu verwendende URL hat die Form `https://<zufälligerSchlüsselundAWS-Region>.amazonaws.com/WebhookApiProdStage/webhook-actions`. Es ist ausreichend die Git-Ereignisse *PUSH und DELETE* an das API-Gateway zu übermitteln.

Um auch den Quell-Code dynamisch für Feature-Branches zu erhalten, muss der während der Einrichtung von NTT-Serverless-CI/CD erzeugte öffentliche SSH-Schlüssel in der Git-Software hinterlegt werden. Dieser ist im Bucket *ci-cd-bootstrap-devopsgitmanagement-1lc-keybucket** zu finden und hat die Bezeichnung *pub_key*.

Soll die Pipeline dem Verhalten folgen, dass die Artefakte auf dem DevOps-Konto erzeugt werden, muss die IAM-Richtlinie hierfür ebenfalls noch erstellt werden. Diese IAM-Richtline muss die *CreateBuildProject*-Berechtigung enthalten und beiden Pipeline-Service-Rollen (*CI-CD-Bootstrap-PipelineRolesD-PipelineServiceRole**) hinzugefügt werden.

Eine Vorlage für die Pipeline bzw. die *pipeline.yaml* ist im Anhang C.1 oder im Git-Repository[15]zu finden. Wie bereits wiederholt beschrieben, muss die *pipeline.yaml* im Verzeichnis *root\build* erzeugt werden.

Die buildspec.yaml welche von CodeBuild verwendet wird, sollte im Verzeichnis *root* hinterlegt sein. Wie diese buildspec.yaml zu strukturieren ist, kann in der AWS-Dokumentation[16] nachgeschlagen werden. Beispiele können ebenfalls im Git-Repository oder in verkürzter Form in den Kapiteln 5.2.1 und 5.1.1 gefunden werden.

Wie in diesem kurzen Kapitel beschrieben, ist die Einbindung von NTT-Serverless-CI/CD in wenigen Schritten möglich. Das Entwickeln der pipeline.yaml mit CloudFormation und das Erstellen der buildspec.yaml sowie das hierfür benötigte Verständnis von AWS stellt die größte Hürde dar. Da für Infrastruktur und Build-Prozesse mehr Möglichkeiten bestehen, als durch diese Arbeit abgedeckt werden können, wird auf die AWS-Dokumentation, die Beispiele inklusive Quell-Code aus Kapitel 5 und die Check-Liste im Anhang C.2 verwiesen.

15 `https://github.com/dominikampletzer/CI-CD-Verfahren-im-Serverless-Cloud-Computing/tree/master/Vorlagen`

16 `https://docs.aws.amazon.com/codebuild/latest/userguide/build-spec-ref.html`

6 Kostenkalkulation

Um die Kosten welche durch NTT-Serverless-CI/CD entstehen aufzuzeigen, wird ein möglichst breites Spektrum an unterschiedlicher Software als Grundlage herangezogen. Die unterschiedliche Software, welche als Grundlage für die Kostenszenarien dienen, wurden durch Interviews mit fünf unterschiedlichen Unternehmen bestimmt. Diese Unternehmen erstrecken sich vom kleinen IT-Unternehmen mit weniger als 10 Angestellten bis zum großen IT-Unternehmen mit über 3.000 Mitarbeitern. Aus den erhaltenen Informationen werden drei Szenarien verwendet: Monolith, Microservice und Web-Applikation, welche in diesem Kapitel vorgestellt werden.

Wie in der Praxis üblich, werden alle drei Szenarien auf dem gleichen CI/CD-Server erzeugt. Von diesem können zuverlässige Informationen über den Build-Prozess bezogen werden, jedoch ist dieser CI/CD-Server, z.B. für das Szenario Microservice überdimensioniert und nur geringfügig ausgelastet. Deshalb wird als Kostenvergleich ebenfalls mit einer lokalen Maschine die Build-Dauer ermittelt und beruhend auf diesen Informationen und den Erfahrungen aus dem Beispiel Microservice wird für jedes Szenario eine zweite alternative Kostenkalkulation erstellt mit angepasster Build-Dauer und CodeBuild-Instanz.

Weiter werden in diesem Kapitel die Kosten für eine traditionelle CI/CD-Pipeline ermittelt, falls diese auf reservierten AWS-EC2-Instanzen betrieben wird. Diese Kosten dienen als Vergleichsgrundlage zu den Kosten, die durch NTT-Serverless-CI/CD für die Szenarien verursacht werden. Anstatt die Kosten für traditionelle CI/CD-Server zu verwenden, werden EC2-Instanzen verwendet. Dies liegt darin begründet, dass diese Kosteninformationen nicht bzw. nicht ausreichend exakt bestimmt werden können oder nicht veröffentlicht werden dürfen.

6.1 Verwendete Szenarien

6.1.1 Szenario Monolith

Das Szenario Monolith beschreibt einen klassischen Software-Monolithen. Bei diesem handelt es sich um eine Software, welche ca. 20 Jahre besteht und stetig weiterentwickelt wird. Diese Entwicklung wurde von verschiedenen Firmen mit unterschiedlichen Technologien durchgeführt. Der Quell-Code umfasst für das Front- und Back-End ca. 2.500.000 Lines-of-Code (LoC) mit einem Speicherbedarf im SCM von ca. 2 GB. Die Größe des Quell-Codes komprimiert durch *7-Zip* mit dem LZMA2-Verfahren und einer durchschnittlichen Kompressionsstärke ergibt eine Größe von ca. 1.478 MB, diese Information wird für die Kostenkalkulation benötigt.

Der verwendete CI/CD-Server ist mit der AWS-Instanz *r6g.8xlarge* vergleichbar bzw. der CodeBuild-Instanz *gpu1.large*. Die AWS-Instanz *r6g.8xlarge* besitzt die Rechenleistung von 32 virtuellen CPUs (vCPU) und 256 GiB Arbeitsspeicher (RAM). Ein vCPU entspricht einem Thread eines Prozessors, welcher abhängig von der EC2-Instanz unterschiedlich ist. Für die

AWS-Instanz *r6g.8xlarge* wird der AWS-Graviton2-Prozessor verwendet[1]. Der Prozessor für die CodeBuild-Instanz *gpu1.large* (32 vCPUs, 244 GiB RAM) konnte nicht ermittelt werden.

Der Monolith wird mit der Veröffentlichungsstrategie Blue/Green betrieben und hat eine Build-Dauer auf dem CI/CD-Server von ca. 9:30 Minuten. Da CodeBuild für die Kostenermittlung auf volle Minuten aufrundet, wird dies für die Kostenberechnung ebenfalls getan. Anstatt 9:30 Minuten wird der Wert 10 Minuten in der weiteren Arbeit verwendet.

Wird das Szenario Monolith lokal gebaut, beträgt die Build-Dauer ~23 Minuten. Die lokale Maschine besitzt einen Intel i7-8650U Prozessor (4 Kerne) und 16 GB RAM sowie eine SSD-Festplatte des Typs SK hynix SC401. Diese wird für spätere Berechnungen mit der EC2-Instanz *t4g.xlarge* (4 vCPUs, 16 GB RAM) bzw. der CodeBuild-Instanz *general1.large* (8 vCPUs, 15 GB RAM) äquivalent gesetzt.

Alle drei Szenarien verwenden den gleichen CI/CD-Server und die gleiche lokale Maschine, weshalb die Leistungsinformationen in den folgenden Abschnitten nicht mehr wiederholt werden.

Der Build bzw. die CI/CD-Pipeline wird pro Monat ca. 570 mal durchgeführt, dies beinhaltet tägliche-Builds, welche auf der Integrationsumgebung veröffentlicht werden, als auch Feature-Builds, welche nicht veröffentlicht werden.

Das Ergebnis des Builds sind mehrere WAR-Dateien, welche zusammen eine Größe von ca. 600 MB aufweisen.

6.1.2 Szenario Microservice

Das Szenario Microservice ist die Grundlage für das Beispiel Microservice. Hierbei handelt es sich um einen technischen Durchstich, welcher bis auf CRUD-Funktionen, einen geringen Funktionsumfang aufweist. Vielmehr geht es um einen technischen Durchstich, welcher die Anwendungsschicht und die Datenschicht der 3-Schichten-Architektur abdeckt und durch REST-Schnittstellen von einer möglichen Präsentationsschicht angesprochen wird.

Da es sich um einen technischen Durchstich und nicht um eine Produktiv-Software handelt, ist die Build-Anzahl mit ca. 50 Builds pro Monat eher gering. Jedoch wurde dieser Wert mit nur einem/er Entwickler/in erreicht.

Das Szenario Microservice hat eine Quell-Code Größe von ca. 7 MB, dies ergibt ebenfalls mit *7-Zip* komprimiert eine Größe von ca. 3 MB.

Trotz der geringen Quell-Code Größe benötigt der CI/CD-Server für die Build-Stage 57 Sekunden. Zusätzlich werden noch 27 Sekunden benötigt, um ein Docker-Image der Applikation zu erstellen. Zusammen ergeben sich hieraus 1:30 Minuten, welche wiederum für die spätere Berechnung auf 2 Minuten aufgerundet werden. Die lokale Maschine benötigt für den gleichen Prozess durchschnittlich 1:59 Minuten, was ebenfalls auf 2 Minuten aufgerundet wird.

Das erzeugte Docker-Image, welches das erzeugte Artefakt der CI/CD-Pipeline darstellt, hat eine Größe von ca. 255 MB.

6.1.3 Szenario Web-Applikation

Beim Szenario Web-Applikation handelt es sich um eine Software, welche ca. 1,5 Jahre betrieben wird und mit einem modernen Web-Framework entwickelt wird. Es besteht aus ca. 35.000 LoC und umfasst Quell-Code von ca. 83 MB, welcher mit *7-Zip* komprimiert ca. 7,5 MB ergibt. Das erzeugte Artefakt hat eine Größe von ca. 9 MB.

1 `https://aws.amazon.com/de/ec2/instance-types/`

Tabelle 6.1 Kostenkalkulation Variablen

	Monolith	Microservice	Web-Applikation
Build-Anzahl/Monat	570	50	140
Erzeugte Artefakte in MB	600	255	9
Feature-Branches/Monat	130	10	40
Pushes pro Feature-Branch	4	4	3
Quell-Code komprimiert in MB	1478	3	7,5
CF-Template Infrastruktur in MB	1	0,3	0,1
Build-Dauer (Praxis) in Min	10	2	5
CodeBuild-Instanz (Praxis)	gpu1.large	gpu1.large	gpu1.large
Build-Dauer (Alternativ) in Min	23	3	7
CodeBuild-Instanz (Alternativ)	general1.large	general1.medium	general1.large

Im Szenario Web-Applikation sind durchschnittlich 4 Entwickler/innen beschäftigt und lösen ca. 140 Build-Prozesse auf dem CI/CD-Server pro Monat aus.

Ein Build auf dem CI/CD-Server dauert ca. 4:52 Minuten, was auf 5 Minuten aufgerundet werden muss. Die lokale Maschine benötigt durchschnittlich 6:57 Minuten, hier beträgt die aufgerundete Build-Dauer 7 Minuten.

Für Szenario Web-Applikation wird ebenfalls der gleiche CI/CD-Server wie für die anderen Szenarien verwendet, welcher mit der AWS-Instanz *r6g.8xlarge* vergleichbar ist und demnach eine Rechenleistung von 32 vCPUs und 256 GiB RAM hat.

6.2 Verwendetes Werkzeug und Metriken

Um die Kostenkalkulation transparent zu machen, ist im Git-Repository[2] eine Excel-Tabelle zu finden, welche als Grundlage für die Kostenkalkulation dient. Diese kann für die eigene Kalkulation verwendet werden, sowie für die Berechnung der Kosten. Die Kosten bzw. Preise von AWS sind in USD und beziehen sich auf die Region Frankfurt. Sollen die Kosten für andere Regionen oder zu einem anderen Zeitpunkt ermittelt werden, ist die Kostenkalkulation dem entsprechend anzupassen. Die Preise bzw. Kosten sind `https://aws.amazon.com/de/pricing/`[3] entnommen. Metriken und Annahmen beruhen auf den drei Szenarien, fehlende Angaben wurden aufgrund der Erfahrungen mit den Beispielen Microservice und Angular-Applikation geschätzt.

Es ist nötig, eine eigene Kostenkalkulation in Form einer Excel-Tabelle zu erstellen, da der Kosten-Kalkulator[4] von AWS zwar sehr gut ist um Schätzungen für die Praxis zu erstellen, jedoch nicht feingranular genug ist bzw. nicht mit so geringen Werten rechnet, wie für eine Kostenkalkulation mit NTT-Serverless-CI/CD benötigt wird.

Tabelle 6.1 zeigt die Metriken, welche auf den Szenarien beruhen und für die Kostenkalkulation berücksichtigt werden. Diese Metriken unterliegen einigen Annahmen, welche neben anderen Annahmen im nächsten Kapitel 6.3 aufgeführt sind.

Folgendes ist unter den jeweiligen Metriken zu verstehen:

Build-Anzahl/Monat Die Anzahl der Build-Prozesse, welche in einem Monat ausgeführt werden. Beruhend auf dreimonatiger Messung in einem der interviewten Unternehmen.

2 `https://github.com/dominikampletzer/CI-CD-Verfahren-im-Serverless-Cloud-Computing/tree/master/Kosten-Kalkulation`

3 Stand: 15.12.2020

4 `https://calculator.aws/`

Erzeugte Artefakte in MB Hierbei handelt es sich um die Größe der durch die CI/CD-Pipeline erzeugten Artefakte in MB.

Feature-Branches/Monat Dies beschreibt die Anzahl an Feature-Branches, welche in einem Monat erzeugt werden.

Pushes pro Feature-Branch Durchschnittliche Anzahl der Push-Aktionen, welche während der Entwicklung in der Git-Software verursacht werden. Dieser Wert beruht auf der Anzahl von Feature-Branches in Kombination mit den ausgeführten Build-Prozessen unter Berücksichtigung der täglichen Builds und Entwickler/innen-Anzahl im jeweiligen Szenario.

Quell-Code komprimiert in MB Größe des Quell-Cods in MB. Der komprimierte Wert ist zu verwenden, da die Speicherung des Quell-Codes in S3 ebenfalls komprimiert stattfindet.

CF-Template Infrastruktur in MB CloudFormation löst selbst keine Kosten aus, jedoch die erzeugte Infrastruktur. Das verwendete CloudFormation-Template wird, um auf den Child-Konten ausgeführt zu werden, zu diesen übertragen. Dies löst Daten-Transfer-Kosten aus. Hierfür wird die Größe der verwendeten CloudFormation-Templates in MB benötigt. Für das Szenario Microservice und Web-Applikation kann der Wert aus den Beispielen Microservice und Angular-Applikation verwendet werden, da diese fast identisch sind. Für das Szenario Monolith wurde ein Schätzwert verwendet.

Build-Dauer Die auf die nächste volle Minute aufgerundete Build-Dauer. Die alternative Build-Dauer für das Szenario Microservice wurde um eine Minute erhöht. Dafür wird eine kleine CodeBuild-Instanz verwendet, siehe Kapitel 6.3.

CodeBuild-Instanz Instanz bzw. vergleichbare Instanz, welche für CodeBuild verwendet wird. Diese Instanzen beruhen auf Linux-Maschinen und wurden Anhand von Kernen und RAM gewählt. Da die vergleichbare EC2-Instanz nicht für CodeBuild verwendet werden kann, wird eine äquivalent als CodeBuild-Instanz verwendet.

6.3 Annahmen

Für die Kostenkalkulation wurden einige Annahmen und Schätzungen getroffen, diese sind im Folgenden ungeordnet aufgeführt:

- Aufrufe des API-Gateways durch die Git-Software wurden nur mit GitHub gemacht. Und es wird davon ausgegangen, dass die Größe der Aufrufe durch BitBucket und GitLab ähnlich sind. Die Aufrufgröße ist mit 15 KB festgesetzt, welche sich aus der gemessenen Inhaltsgröße von 10 KB und großzügig geschätzten 5 KB für Header-Parameter zusammensetzt.
- Aufrufe der Lambda-Funktionen und durch diese verursachte Aufrufe wurden mit Ausnahme der GitManagementLambda großzügig mit 5 KB geschätzt. Dies liegt daran, dass die verwendeten Lambda-Funktionen nur wenige Informationen übertragen und i.d.R. nur CloudFormation-Stacks anstoßen.
- AWS stellt nur aktive Pipelines in CodePipeline[5] in Rechnung. Dies bedeutet, da Feature-Branches und dadurch DEV-Pipelines nach wenigen Tagen entfernt werden,

5 https://aws.amazon.com/de/codepipeline/pricing

dass diese nicht als aktiv gelten. Hieraus folgt, dass nur die DELIVERY-Pipeline als aktiv gezählt wird, welche es nur einmal pro Szenario gibt. Sollte es, aus welchen Gründen auch immer, verschiedene aktive Pipelines geben oder sollten Feature-Branches entgegen den CI/CD-Prinzipien über eine lange Zeit betrieben werden, entstehen auch für diese Pipelines Kosten.

- KMS Aufrufe wurden für jeden Build mit dem Faktor drei multipliziert, da KMS durch CodePipeline, GitManagementLambda und GitCheckoutLambda verwendet wird. Dies sind vermutlich zu viele KMS-Aufrufe, da z.B. die GitCheckoutLambda nicht jedes Mal benötigt wird. Da die KMS-Aufrufe jedoch kaum Kosten verursachen, kann hier großzügig geschätzt werden.
- Die Größe des CloudFormation-Templates wurde, wie bereits erwähnt, von den Beispielen abgeleitet und für das Szenario Monolith geschätzt.
- Laufzeiten in Millisekunden aller Lambda-Funktionen wurden anhand der Beispiele Microservice und Angular-Applikation ermittelt. Diese können sich jedoch abhängig vom erzeugten bzw. zu löschenden CloudFormation-Stack stark unterscheiden.
- Als AWS-Region wird Frankfurt verwendet und alle Preise sind in USD ohne Steuer.
- Die Free-Tier-Nutzung (kostenlose Nutzung) von AWS wird nicht berücksichtigt.
- Preisrabatte werden nicht berücksichtigt.
- Die Kosten für den Speicherbedarf wurden mit S3-Standard-Nutzung berechnet. Günstigere Preise für größere Nutzung sowie günstigere Speicheroptionen werden nicht berücksichtigt, können und sollten in der Praxis jedoch berücksichtigt bzw. für die Archivierung genutzt werden.
- Speicherbedarf wird nur auf einmaliges Erzeugen ohne Archivierung der Artefakte/Quell-Codes berücksichtigt. Dies betrifft sowohl S3 als auch ECR.
- AWS-Kosten für die Initialisierung von NTT-Serverless-CI/CD werden ignoriert, da diese zu gering und unbedeutend sind, um die Kostenkalkulation zu beeinflussen.
- Der vermutlich größte Kostenpunkt, die erzeugte Infrastruktur, sowie der Transfer der Artefakte in die Child-Konten wurde nicht berücksichtigt. Die Infrastrukturkosten der Software stellen i.d.R. den größten Teil der Cloud-Kosten dar, sind jedoch für CI/CD von geringer Bedeutung.
- CloudWatch wird außer Acht gelassen, da die Intensität des Monitorings stark von den Anforderungen der zu unterstützenden Software abhängig ist und bei den aufgezeigten Beispielen zu keinerlei Kosten geführt hat. In der Praxis muss jedoch bedacht werden, was überwacht wird und wie lange diese Daten gespeichert werden.Das Erheben der Daten verursacht i.d.R. kaum Kosten, jedoch kann häufiges Erheben von Daten mit einer langen Speicherdauer zu großen Kosten führen.
- Kompressionsverfahren von AWS und das lokale verwendete mit *7-Zip* werden äquivalent gesetzt, auch wenn es geringe Unterschiede gibt. Diese liegen jedoch bei den gegebenen Datenmengen im vernachlässigbaren MB-Bereich.

Tabelle 6.2 AWS-Kosten pro Dienst in USD

Dienst - $ / Monat	Monolith	Microservice	Web-Applikation
KMS	5,000513	5,000045	5,000126
S3	0,050888	0,007343	0,001565
SecretsManager	0,402850	0,400250	0,400700
CodePipeline	1,000000	1,000000	1,000000
API-Gateway	0,008436	0,000740	0,001554
CloudFormation	0,005567	0,000146	0,000137
Lambda	12,257301	0,944719	3,750321
CodeBuild (Praxis)	4.571,566992	80,125977	560,022559
SUMME (Praxis)	4.590,290000	87,480000	570,180000
SUMME $ / Jahr (Praxis)	55.083,480000	1.049,760000	6.842,160000
CodeBuild (Alternativ)	273,766992	1,625977	19,622559
SUMME (Alternativ)	292,490000	8,980000	29,780000
SUMME $ / Jahr (Alternativ)	3.509,880000	107,760000	357,360000

- Beruhend auf den Erfahrungen mit Beispiel Microservice, wurde für die Berechnung der Kosten für den alternativen Build-Prozess die Build-Dauer erhöht und eine kleinere CodeBuild-Instanz verwendet.

- Die Build-Dauer des CI/CD-Servers wird in CodeBuild auf äquivalenten Maschinen als identisch betrachtet. Vermutlich ist die Build-Dauer mit CodeBuild verkürzt, siehe Kapitel 7.

Diese Auflistung von Annahmen ist gleichzeitig auch als Kritik zu betrachten. Da jede Annahme die Genauigkeit des Endergebnisses negativ beeinflusst. Besonders die Annahme der Laufzeit der Lambda-Funktionen kann starke Auswirkungen auf die Kosten haben. Dies geht auch auch bei Eivy hervor. So ist es nach Eivy unerlässlich, um die Kosten für Lambda-Funktionen zu ermitteln, diese anhand der echten Software zu testen. Da unterschiedliche Aufrufhäufigkeit, Laufzeiten und die Menge an zu übertragenden Daten bei Lambda-Funktionen einen starken Einfluss auf die Kosten haben können [Eiv17][S.4].

Ein großer Kritikpunkt ist, dass die Szenarien nicht direkt mit NTT-Serverless-CI/CD umgesetzt werden, um die echte Build-Dauer zu ermitteln. Aus zeitlichen und vor allem rechtlichen Gründen wurde hierauf verzichtet und mit den Erfahrungen aus den Beispielen und der Annahme gearbeitet, dass bei ähnlichen Maschinen die Build-Dauer ähnlich ist.

6.4 Ergebnis der Kostenkalkulation

Tabelle 6.2 zeigt auf, welche Kosten für jeden verwendeten AWS-Dienst pro Monat für jedes Szenario entstehen würden. Weiter ist werden unterschiedliche Kostenkalkulationen - *Praxis* und *Alternativ* - unteschieden. *Praxis* steht hier für die Kostenkalkulation, in der die Build-Zeit des echten CI/CD-Server sowie ein Äquivalent als CodeBuild-Instanz verwendet wird. *Alternativ* hingegen zeigt die Kostenkalkulation, für welche ein CodeBuild-Instanz äquivalent zur lokalen Maschine verwendet wird. Ausnahme hiervon bildet das Szenario Microservice, da hier die Erfahrungen aus dem Beispiel Microservice verwendet wurden, und eine kleinere CodeBuild-Instanz verwendet wurde. Weiter ist die Summe für die einzelnen Kostenkalkulationen der AWS-Dienste sowohl pro Monat als auch pro Jahr zu entnehmen.

Aus der Tabelle können die Erkenntnisse gewonnen werden, dass die Kosten für KMS und CodePipeline keinen bis nur einen geringen variablen Anteil beinhalten. Weiter, was ebenfalls nicht verwunderlich ist, dass die Kostentreiber die rechenintensiven Dienste CodeBuild und Lambda sind. Ein in der Tabelle nicht offensichtlicher Kostentreiber ist S3. In den Annahmen von Kapitel 6.3 ist zu finden, dass davon ausgegangen wird, dass keine Archivierung stattfindet bzw. nicht benötigte Artefakte entfernt werden.

Ein weiterer verdeckter Kostenfaktor, welcher im Vergleich zu den Build-Kosten aus CodeBuild gering, aber vorhanden ist, stellt der Datentransfer dar. Diese Kosten werden dem jeweiligen Dienst zugeordnet, sollten jedoch auch separat bedacht werden. So ist es in AWS i.d.R. kostenpflichtig, Daten von einem Dienst zu einem anderen Dienst zu übertragen. Regionsübergreifender Datentransfer erzeugt außerdem, im Vergleich zum Datentransfer innerhalb einer Region, erhöhte Kosten. Dieser Datentransfer bedeutet, dass z.B. im Falle von CodeBuild der Quell-Code zuerst in den Dienst transferiert werden muss. Nach dem Erzeugen der Artefakte müssen diese Artefakt ebenfalls weiter transferiert werden. Hierfür entstehen im Verhältnis zu rechenintensiven Diensten geringe Kosten, diese können jedoch für große bis sehr große Software relevant sein.

Der Unterschied zwischen den zwei Kostenkalkulationen für *Praxis* und *Alternativ* ist ganz klar ersichtlich, da der Unterschied bei Faktor 10 bis 20 liegt. Hieraus wird ersichtlich, dass die verwendete CodeBuild-Instanz und die Build-Dauer den größten Einfluss auf die Kosten von NTT-Serverless-CI/CD haben. Dies lässt den Schluss zu, dass Kostenoptimierungen durch die Wahl einer weniger leistungsstarken CodeBuild-Instanz bei Verlängerung der Build-Dauer möglich sind.

6.4.1 Kosten durch AWS-CI/CD

Aus der Tabelle 6.2 kann abgeleitet werden, wie hoch die Kosten für CI/CD in AWS ohne NTT-Serverless-CI/CD sind bzw. wie hoch die Differenz zwischen CI/CD in AWS mit und ohne NTT-Serverless-CI/CD sind. Es kann davon ausgegangen werden, dass ebenfalls KMS-Schlüssel, CodePipeline, CodeBuild, S3 und der SecretsManager genutzt werden. Zusätzlich entstehen jedoch noch Kosten unbekannter Höhe für die Entwicklung und die Lösung des Problems, wie Feature-Branches bzw. die Erzeugung und das Entfernen der zugehörigen Infrastruktur entstehen.

Davon ausgehend, dass für eine CI/CD-Lösung mit AWS-Werkzeugen, ungeachtet der ungelösten Probleme, folgende stark vereinfachte Kosten bzw. Ressourcen benötigt werden, kann Tabelle 6.3 erzeugt werden:

- mindestens zwei KMS-Schlüssel
- Eine aktive Pipeline
- Gleiche CodeBuild-Kosten
- Gleiche Kosten für S3 und SecretsManager

Wird Tabelle 6.3 mit Tabelle 6.2 verglichen, ist festzustellen, dass pro Monat NTT-Serverless-CI/CD Mehrkosten im Vergleich zu einer CI/CD-Lösung nur mit AWS-Werkzeugen für Szenario Monolith von ~15 $ verursacht; für Szenario Microservice Mehrkosten von ~4 $ und für Szenario Web-Applikation Mehrkosten von ~7 $. Dabei ist es natürlich egal, mit welcher CodeBuild-Instanz die Kalkulation gemacht wird, lediglich die prozentualen Kosten

Tabelle 6.3 Kosten pro Dienst ohne NTT-Serverless-CI/CD in USD

Dienst - $ / Monat	Monolith	Microservice	Web-Applikation
KMS	2,00	2,00	2,00
S3	0,05	0,01	0,01
SecretsManager	0,40	0,40	0,40
CodePipeline	1,00	1,00	1,00
CodeBuild (Praxis)	4.571,57	80,13	560,02
SUMME (Praxis)	4.575,02	83,54	563,43
SUMME $ / Jahr (Praxis)	54.900,24	1.043,88	6.804,36
CodeBuild (Alternativ)	273,77	1,63	19,62
SUMME (Alternativ)	277,22	5,04	23,03
SUMME $ / Jahr (Alternativ)	3.326,64	60,48	276,36

für NTT-Serveless-CI/CD verändern sich. Mit diesen Mehrkosten können die Vorteile aus Kapitel 4.4.1 erzielt werden.

Nicht berücksichtigt sind die Kosten für die Infrastruktur-Integration der Feature-Branches und für die Implementierung bzw. Entwicklung einer Lösung dieses Problems. Insbesondere die Lösung dieses Problems wird voraussichtlich ähnlich der verwendeten Lösung von NTT-Serverless-CI/CD sein, wodurch ähnliche zusätzliche Kosten, wie für NTT-Serverless-CI/CD, anfallen dürften.

Und sollte dieses Problem nicht gelöst werden, decken diese Mehrkosten mit Leichtigkeit die Kosten bzw. ist der Zeitaufwand als höher zu bewerten, als der den ein/e Entwickler/in benötigt, um die Infrastruktur für Feature-Branches manuell zu erstellen bzw. zu zerstören.

6.5 Vergleichsrechnung: NTT-Serverless-CI/CD vs. EC2-Instanz

Da die Kosten für NTT-Serverless-CI/CD für die drei Szenarien berechnet sind, ist der nächste Schritt, diese Kosten den Kosten eines traditionellen CI/CD-Servers entgegenzustellen. Wie bereits einführend erwähnt, war es nicht möglich die tatsächlichen CI/CD Kosten der Unternehmen durch die Interviews zu ermitteln. Dies liegt daran, dass diese Kosten nicht bzw. nicht ausreichend exakt bestimmt werden können oder nicht veröffentlicht werden dürfen.

Es sollte nicht angenommen werden, dass der Wechsel in die Cloud generell die günstigere Alternative ist. Die Annahme, dass die Cloud-Kosten, Moor´s Gesetz entsprechend, für bestimmte Leistungen einen Preisverfall unterliegen, ist ebenfalls falsch. Dies liegt daran, dass Rechenleistung nur einen kleinen Teil der Cloud-Kosten betragen [Wei17][S.2-3].

Um dennoch einen Kostenvergleich erstellen zu können, wird eine zu dem CI/CD-Server und der lokalen Maschine äquivalente EC2-Instanz bzw. CodeBuild-Instanz für jedes Szenario verwendet. Die Verwendung dieser Äquivalente beinhaltet einige Vor- und Nachteile, wie in diesem Abschnitt deutlich wird.

Die Kosten für NTT-Serverless-CI/CD werden aus den vorangegangen Kapiteln entnommen bzw. mit dem vorgestellten Excel-Werkzeug aus Kapitel 6.2 ermittelt.

Bei der Ermittlung der Kosten eines traditionellen CI/CD-Servers spielen einige Faktoren eine Rolle, auch wenn diese Kosten nicht direkt mit einem Geldwert beziffert werden können, so z.B. Ausfall. Zu diesen Kosten zählen:

- Hardware (Erweiterung, Reduzierung)

Tabelle 6.4 Kosten reservierte EC2-Instanzen vs. NTT-Serverless-CI/CD in USD

Kosten - $ / Jahr	Monolith	Microservice	Web-Applikation
EC2 (Praxis)	10.093,40	10.093,40	10.093,740
NTT-Serverless-CI/CD (Praxis)	55.083,48	1.049,76	6.842,16
EC2 (Alternativ)	862,43	601,44	862,43
NTT-Serverless-CI/CD (Alternativ)	3.509,88	107,76	357,36

- Erneuerung
- Entsorgung
- Strom
- Arbeitszeit (Aufbau, Wartung, Entsorgung)
- Ausfall
- Sicherheit
- Miete / Raumkosten

Diese Kosten existieren so im Cloud-Umfeld nicht bzw. sie sind im Preis für die Verwendung der AWS-EC2-Instanz inbegriffen. Hierdurch wird die Vergleichsrechnung stark vereinfacht, transparent, nachvollziehbar und leicht zu prüfen. In der Praxis sind die Kosten für traditionelle Server oft schwierig zu ermitteln und gegebenenfalls nur anteilig oder über Gemeinkosten zu ermitteln, so z.B. Raumkosten.

Um die Kosten für die EC2-Instanzen zu ermitteln, werden die Annahmen und die Szenarien aus den vorangegangenen Kapiteln verwendet, sowie folgende zusätzliche Annahmen getroffen:

- Es werden reservierte EC2-Instanzen verwendet, da diese günstiger als OnDemand Instanzen sind und dies am ehesten traditionellen Servern entspricht.
- Es werden nur Kosten für ein Jahr berechnet. Und demnach wird auch die reservierte EC2-Instanz lediglich für ein Jahr verwendet.
- Falls möglich, wird *Upfront payment* genutzt, da dies ebenfalls dem traditionellen Servern am nächsten kommt und günstiger ist.
- Der CI/CD-Server wird nur für das jeweilige Szenario verwendet.
- Der von AWS zur Verfügung gestellte Cost Calculator wird verwendet, um die Kosten zu ermitteln.
- EC2-Instanzen haben genügend Rechenleistung und Kapazität um die CI/CD-Software (z.B. Jenkins) zu betreiben.
- Kosten für Wartung und Integration der CI/CD-Software werden nicht berücksichtigt. Diese müssen als Aufwand zu den kalkulierten Kosten hinzugezählt werden.

Tabelle 6.4 zeigt die kalkulierten Server-Kosten sowie zur Gegenüberstellung die Kosten für NTT-Serverless-CI/CD, jeweils für ein Jahr. Es ist der *Praxis*-Fall[6] vertreten, in welchem alle drei Szenarien die gleiche Leistung haben, sowie der *Alternativ*-Fall für Szenarien Monolith, Web-Applikation[7] und Szenario Microservice[8].

Der Tabelle kann entnommen werden, dass es für das Szenario Monolith günstiger ist eine EC2-Instanz zu betreiben, als NTT-Serverless-CI/CD zu nutzen. Die Mehrkosten durch NTT-Serverless-CI/CD betragen ~445% (Praxis) bzw. ~305% (Alternativ) gegenüber der EC2-Instanz.

Die Szenarien Microservice und Web-Applikation hingegen profitieren von NTT-Serverless-CI/CD. So ist die Kostenersparnis im Szenario Microservice ~90% (Praxis) bzw. ~82% (Alternativ) und im Szenario Web-Applikation ~32% (Praxis) bzw. ~58% (Alternativ).

Hierbei sind jedoch die Kosten der Wartung wie der Annahme zu entnehmen ist, nicht berücksichtigt. Folglich ist die Kostenersparnis in den Szenarien Microservice und Web-Applikation durch NTT-Serverless-CI/CD noch höher, da die Kosten für die Wartung noch auf die Kosten der EC2-Instanz addiert werden müssen. Dies betrifft auch das Szenario Monolith, wodurch sich die Mehrkosten von NTT-Serverless-CI/CD im Vergleich verringern, jedoch immer noch sehr hoch ausfallen dürften.

6.6 Erkenntnisse

Erkenntnisse, welche aus diesem Kapitel geschlossen werden können sind:

- Kosten für NTT-Serverless-CI/CD sind stark von der Build-Häufigkeit, Build-Dauer und am stärksten von der verwendeten CodeBuild-Instanz abhängig.
- Durch NTT-Serverless-CI/CD ist es möglich für die Software CI/CD-Infrastruktur mit angemessener Leistung zu verwenden, bzw. die Leistung der CI/CD-Infrastruktur an unterschiedliche Software, Zielumgebungen und Branches anzupassen.
- Kostenoptimierungen können in gewissen Grenzen durch erhöhte Build-Dauer bei Verwendung kleiner CodeBuild-Instanzen erzielt werden. Zeit vs. Kosten.
- Die Kosten für sehr geringe bis geringe Nutzung beinhalten einen hohen fixen Anteil.
- Der Hauptkostentreiber bei NTT-Serverless-CI/CD ist CodeBuild.
- Unabhängig vom Szenario bietet NTT-Serverless-CI/CD im Vergleich zu einer CI/CD-Lösung in AWS ohne NTT-Serverless-CI/CD für geringe Mehrkosten die Vorteile aus Kapitel 4.4.1.
- Große Software mit einer hohen Build-Häufigkeit profitiert nicht von NTT-Serverless-CI/CD.
- Kleine bis mittelgroße Software profitiert von den geringeren Kosten, welche NTT-Serverless-CI/CD verursacht.

6 `https://calculator.aws/#/estimate?id=88cebdc4efaf74cf939152900be367c6afe6e36f`
7 `https://calculator.aws/#/estimate?id=8e927b2892ddec7dc00d7981d1d5bea42c264a6d`
8 `https://calculator.aws/#/estimate?id=b89c7f7548b8c1fe2f2f8e8c849f31f04b3a6884`

Die Erkenntnisse, dass kleine bis mittelgroße Software NTT-Serverless-CI/CD verwenden sollte und große Software nicht, beruhen nur auf dem Kostenaspekt. Es gibt noch weitere Vorteile von NTT-Serverless-CI/CD, welche berücksichtigt werden sollten.

Weiter wird davon ausgegangen, dass der CI/CD-Server nur für CI/CD eines Szenarios genutzt wird. Dies ist in der Praxis oft nicht der Fall. So ging aus den Interviews hervor, dass auch z.B. SCM oder Test-Umgebungen mit diesem Server betrieben werden. Dadurch dürfen die Server-Kosten nur anteilig betrachtet werden. Dies trifft auch auf den Fall zu, wenn mehr als eine Software durch den CI/CD-Server unterstützt wird. So mag es richtig sein, dass der CI/CD-Prozess durch NTT-Serverless-CI/CD für eine Software günstiger ist als die Einrichtung eines CI/CD-Servers für diese Software. Falls der benötigte CI/CD-Server jedoch bereits betrieben wird, wäre es vermutlich Unsinn diesen nicht zu nutzen, zumindest bis das Ende des Lebenszyklus dieses CI/CD-Servers erreicht ist oder dessen Kapazitäten nicht ausreichend sind.

7 Exkurs: Instanzgrößen für CodeBuild

Im Zuge dieser Arbeit ist aufgefallen, dass die Build-Dauer in AWS durch CodeBuild im Vergleich zur Build-Dauer auf einer lokalen Maschine sehr kurz ist. Etwas Unerwartetes konnte für das Szenario Microservice im Vergleich zum Beispiel Microservice festgestellt werden. Das Szenario Microservice, welches auf dem CI/CD-Server erzeugt wird und ungefähr die Leistung einer AWS-EC2-Instanz vom Typ *r6g.8xlarge* aufweist (32 vCPUs und 256 GiB RAM), ist nur geringfügig schneller, als das Beispiel Microservice, welches quasi identisch mit dem Szenario ist, jedoch lediglich eine CodeBuild-Instanz vom Typ *general1.small* (2 vCPUs und 3 GB RAM) nutzt.

Es ist wünschenswert diverse Messungen mit unterschiedlichen Szenarien auf den echten CI/CD-Servern durchzuführen und die Szenarien mit NTT-Serverless-CI/CD abzubilden und mit den CodeBuild-Zeiten zu vergleichen. Dies ist im Zuge dieser Arbeit aus unterschiedlichen Gründen nicht möglich.

Was jedoch durchgeführt werden kann, ist die Installationsdauer und die Build-Dauer der Build-Stage für die Beispiele Microservice und Angular-Applikation auf einer lokalen Maschine mit der auf AWS-CodeBuild zu vergleichen. Hiermit soll die Vermutung untermauert werden, dass bei ähnlicher Leistung des CI/CD-Servers, CodeBuild schneller ist. Oder anders ausgedrückt, dass um vergleichbare Ergebnisse, wie bei einem CI/CD-Server mit einer bestimmten Leistung zu erzielen, eine CodeBuild-Instanz mit niedrigerer Leistung verwendet werden kann.

Dieses Kapitel soll nicht als Beweis, sondern als Indiz für weitere Nachforschung interpretiert werden, da die Aussagekraft nicht ausreichend und nicht repräsentativ ist.

7.1 Testumgebungen

Um die lokale Maschine für die Messungen vorzubereiten, wird der aktuelle Quell-Code der Beispiele geladen und jedwede nicht benötigte Software beendet, um möglichst wenig Leistungsabfluss an Fremd-Software zu erhalten. Ausnahme hiervon stellt das Beispiel Microservice dar, da die Container-Software Docker für das Erzeugen des Docker-Images benötigt wird. Für die Ausführung der Build-Befehle wird ein Skript verwendet, welches vor jedem Build den Cache und mögliche zuvor erzeugten Artefakte entfernt. Dieses Skript wird durch die CLI ausgeführt.

Die lokale Maschine weist folgende Leistungsmerkmale auf:

OS Windows 10 Enterprise 64 Bit

CPU Intel Core i7-8650U CPU 1.90GHz 2.11GHz (4 Kerne)

RAM 16 GB

Festplattenspeicher SSD - SK hynix SC401 512GB

Netzwerk 48,6 Mbit/s Download-Geschwindigkeit

Tabelle 7.1 Messergebnisse - Angular-Applikation

	Lokale Maschine	CodeBuild
ng build:dev	58.466 ms	42.365 ms
ng build:int	166.277 ms	62.056 ms
npm install	138.513 ms	26.006 ms
npm install ohne Chromium	102.809 ms	18.723 ms

Für CodeBuild müssen die Beispiele durch ein CodeBuild-Projekt repräsentiert und ausgeführt werden. Dies ist durch NTT-Serverless-CI/CD bereits geschehen. Für die Messungen werden jedoch alle zuvor verwendeten CodeBuild-Projekte und Pipelines entfernt und der PipelineMaker erneut ausgeführt, um eine saubere CI/CD-Infrastruktur für die Beispiele zu verwenden. Für das Beispiel Angular-Applikation wurde zusätzlich ein neuer Feature-Branch erzeugt, da unterschiedliche Konfigurationen für DEV und INT/PROD verwendet werden und auch hierfür eine durch NTT-Serverless-CI/CD neue CI/CD-Infrastruktur erzeugt wird.

CodeBuild verwendet für beide Beispiele die gleiche Infrastruktur, diese weist folgende Leistungsmerkmale auf:

OS Ubuntu

CPU 2 vCPUs

RAM 3 GB

Docker-Image aws/codebuild/standard:3.0 Version:20.08.14

Netzwerk Unbekannt. AWS bietet EC2-Instanzen mit unterschiedlichen Übertragungsleistung, abhängig von der Instanzgröße an. Diese reichen von 25 GBit/s bis 100 GBit/s. Es ist nicht klar, welche Übertragungsleistung für die verwendeten CodeBuild-Instanzen verwendet wird.

Bezüglich der vCPUs muss angemerkt werden, dass AWS nicht angibt, welche Leistungen diese exakt haben. vCPUs auf Maschinen für allgemeine Zwecke sind meist ein Thread eines Intel Xeon-Kerns oder eines AMD EPYC-Kerns[1]. Da eine kleine CodeBuild-Instanz verwendet wird, ist es sehr wahrscheinlich, dass einer dieser Prozessoren verwendet wird.

7.2 Messergebnisse

Tabelle 7.1 zeigt die durchschnittlichen Ergebnisse der Messungen für das Beispiel Angular-Applikation in Millisekunden. Um diese zu erhalten wurde auf der lokalen Maschine folgendes Skript 7.1) verwendet:

```
1 rmdir /Q /S "node_modules" && npm cache clear --force && npm install && npm run
    build:<configuration>
```

Code 7.1 Angular-Applikation Installationsskript

Dieses Skript entfernt bereits installierte Node-Module, löscht den NPM-Cache, installiert benötigte Node-Module neu und führt den Build abhängig von der gewünschten Konfiguration aus. Dies gewährleistet, dass auf der lokalen Maschine, genau wie in CodeBuild, die Installation frisch ist und der gleiche Build-Prozess ausgeführt wird.

1 https://aws.amazon.com/de/ec2/instance-types/

Tabelle 7.2 Messergebnisse - Microservice

	Lokale Maschine	CodeBuild
mvn clean install -DskipTests=true	204.500 ms	35.117 ms
mvn clean install -DskipTests=true (cached)	24.489 ms	-
mvn clean package	23.461 ms	19.436 ms

Als Messwert wird dabei die Zeit verwendet, welche NPM und das Angular-CLI-Werkzeug als Dauer angeben. Dies kann sowohl auf der lokalen Maschine als auch bei CodeBuild in den Logs ausgelesen werden.

Tabelle 7.1 zeigt die zeitliche Überlegenheit der CodeBuild-Umgebung gegenüber der lokalen Maschine deutlich. Diese Überlegenheit ist sowohl im rechenintensiven Build-Prozess als auch im netzwerkintensiven Installations-Prozess zu sehen.

Hieraus kann gefolgert werden, dass die Rechenleistung von 2 vCPUs aus CodeBuild den 4 Kernen eines Intel Core i7-8650U überlegen ist. Dies lässt vermuten, dass Hardware-Leistung von CI/CD-Servern nicht 1:1 auf AWS-CodeBuild übertragen werden kann. Hieraus wiederum kann vermutet werden, dass für Kostenberechnungen, anstatt einer zum CI/CD-Server äquivalenten CodeBuild-Maschine, auch eine günstigere kleinere Maschine ähnliche Ergebnisse erbringen kann.

Für das Messen des Beispiel Microservice wurde folgendes Skript auf der lokalen Maschine verwendet:

```
1 rmdir /s /q "C:\Users\<USER>\.m2\repository"  && mvn clean install -DskipTests=
    true
```

Code 7.2 Microservice Installationsskript

Mit dem Skript aus Code 7.2 wird sichergestellt, dass ein zuvor installiertes Maven-Repository entfernt wird. Dadurch entsteht keine Verzerrung durch Caching, welche auf AWS nicht auftritt, da ein neuer Docker-Container für den Build verwendet wird. Als Messwerte werden jeweils die Zeiten verwendet, welche Maven bis zum Beenden des jeweiligen Prozesses angibt.

Tabelle 7.2 ist das Ergebnis der Messungen des Beispiel Microservice. Aus Tabelle 7.2 kann das gleiche Resultat, wie aus dem Beispiel Web-Applikation gezogen werden. CodeBuild ist mit vermeintlich geringerer Leistung der lokalen Maschine überlegen. Im Installationsschritt ist ganz deutlich zu erkennen, welchen Vorteil CodeBuild durch eine bessere Netzwerkverbindung aufweist. In der Zeile *mvn clean install -DskipTests=true* ist zu sehen, wie viel schneller CodeBuild ist, wenn eine saubere Neuinstallation durchgeführt wird. Wird der Wert aus der nächsten Zeile *mvn clean install -DskipTests=true (cached)*, welcher für die lokale Maschine das Cached-Maven-Respository nutzt, mit diesen Zeiten verglichen, wird deutlich, welchen Vorteil CodeBuild durch die schnelle Netzwerkverbindung erzielt. In CodeBuild kann dieses Caching, wie bei der lokalen Maschine, ebenfalls genutzt werden. Dafür muss jedoch ein eigenes Build-Docker-Image (siehe Kapitel 5.2.3) erzeugt werden, welches bereits das Cached-Maven-Respository beinhaltet.

7.3 Kritik an Erhebung

Wie einführend in diesem Kapitel bereits erwähnt, sind die hier erhobenen Messungen nur als Indiz zu werten. Dies liegt daran, dass es für eine aussagekräftige Analyse zu wenig Daten gibt.

Es fehlt an mehreren CI/CD-Servern, welche unterschiedliche Software aus der Praxis bauen. Diese Software müsste ebenfalls in CodeBuild gebaut werden um echte Aussagen treffen zu können. Weiter sind die Beispiele gut zu gebrauchen, bilden aber keine repräsentative Quell-Code-Basis. Die Beispiele sind sehr einfach gehalten und mit diesen geht auch ein großer Overhead für den geringen Quell-Code-Umfang einher.

Eine weitere Unbekannte stellt CodeBuild da, da sowohl die Geschwindigkeit der Netzwerkverbindung als auch die Leistung des Prozessors nicht genau geklärt werden konnte.

Um diesem interessanten Thema weiter auf den Grund zu gehen und aussagekräftige Informationen zu erhalten, wäre folgendes im nächsten Schritt nötig:

- Vielzahl unterschiedlicher Software auf unterschiedlichen CI/CD-Servern erzeugen und messen
- Diese Software in CodeBuild erzeugen und messen
- Leistung des CodeBuild-Prozessors und des Netzwerks feststellen

8 Fazit

DevOps und das in DevOps enthaltene CI/CD und IaC bietet enorme Vorteile wie z.B. bessere Code-Qualität, Kostenersparnisse und kürzere Time-to-Market. Dies sind Vorteile, welche sich kein Unternehmen oder IT-Projekt entgehen lassen sollte. Unabhängig davon, ob die CI/CD-Pipeline von traditionellen Servern oder Serverless betrieben wird. Jedoch ist der Wechsel von traditionellen Servern zur Cloud nicht ausreichend. Unternehmen müssen eine Balance zwischen Software-Architekturanforderungen inklusive Microservice und Cloud Funktionalitäten finden, um im Wettbewerb zu bestehen [Wei17][S.1]. Der reine Wechsel von traditionellen Servern zu AWS-EC2-Instanzen ist unter Umständen sogar mit Mehrkosten und höherem Aufwand verbunden, auch die Architektur der Software muss für die Cloud geeignet sein. Hiervon ist jedoch der Build-Prozess weitgehend unberührt.

Anhand der Beispiele Microservice und Angular-Applikation wurde das Framework NTT-Serverless-CI/CD vorgestellt und die Verwendung erklärt. NTT-Serverless-CI/CD stellt keine Technologie dar. Es kann als Vorgehensmodell beschrieben werden, welches benötigte CloudFormation-Templates und Lambda-Funktionen beinhaltet. Diese werden von AWS-Diensten genutzt, um Serverless-CI/CD automatisiert zu initialisieren und zu betreiben. NTT-Serverless-CI/CD erzeugt alle nötigen Rollen, Rechte, Infrastruktur, Funktionen und Schnittstellen, welche für Serverless-CI/CD benötigt werden. Und dies unter Berücksichtigung der Automatisierung, Einfachheit der Einrichtung und Best-Practices von AWS zu sehr geringen Mehrkosten im Vergleich zur Verwendung der CI/CD-Werkzeuge von AWS ohne NTT-Serverless-CI/CD.

Weiter kann folgendes festgehalten werden:

- NTT-Serveless-CI/CD kann bedenkenlos für Software verwendet werden, welche Serverless-CI/CD in AWS umsetzen sollen. Die Mehrkosten von NTT-Serverless-CI/CD sind sehr gering und vereinfachen, automatisieren und erweitern Serverless-CI/CD in AWS.

- Unternehmen, welche für Ihre Software keine eigene CI/CD-Infrastruktur verwenden oder wo die Ziel-CI/CD-Infrastruktur nur ungenau abgeschätzt werden kann, sollten einen Serverless-CI/CD-Ansatz in Betracht ziehen, um initiale Kosten und Kosten der Erweiterung zu sparen. Falls dieser Serverless-CI/CD-Ansatz in AWS umgesetzt werden soll, ist ebenfalls NTT-Serverless-CI/CD zu empfehlen. Allerdings sollten die Kosten hierfür regelmäßig beobachtet und verglichen werden, da viele kleine Software-Build-Prozesse möglicherweise günstiger in einem traditionellen CI/CD-Server betrieben werden können, jedoch mit Nachteilen, welche dieser mit sich bringt.

- Bei großer Software mit langer Build-Zeit und hohen Hardware-Anforderungen tritt der zuvor angeführte Punkt ein. Es ist möglich und bietet auch Vorteile NTT-Serverless-CI/CD zu verwenden, jedoch sind die Kosten, welche NTT-Serverless-CI/CD bzw. CodeBuild verursachen viel höher als die Kosten, welche eine EC2-Instanz verursacht. Für große Software ist eine genaue Kostenkalkulation aller Kostenfaktoren, wie z.B. Wartung und Ausfall nötig, um die Wirtschaftlichkeit zu analysieren.

- Um die Kosten für NTT-Serverless-CI/CD zu ermitteln, wird ein Werkzeug in Form einer Excel-Datei zur Hand gegeben.[1]
- Messungen haben gezeigt, dass Build-Zeiten auf einer lokalen Maschine länger benötigen als in CodeBuild. Dies lässt vermuten, dass dies ebenfalls für CI/CD-Server zutrifft. Somit könnte eine CodeBuild-Instanz, welche weniger Leistung als der CI/CD-Server aufweist, ähnliche Build-Zeiten erzielen. Dies beeinflusst die Kostenkalkulationen erheblich, muss jedoch im jeweiligen Fall geprüft werden.
- Um die Kosten für NTT-Serverless-CI/CD zu senken, kann in vielen Fällen eine schwächere CodeBuild-Instanz verwendet werden, bei erhöhter Build-Dauer.
- Die Einrichtung von NTT-Serverless-CI/CD in AWS ist sehr einfach, jedoch wird eine Menge Know-How und Wissen benötigt, um die CloudFormation-Templates für die Ziel-Infrastruktur zu erstellen.
- NTT-Serverless-CI/CD ist nicht geeignet, wenn die Best-Practice der Aufgabenteilung nicht mit vier AWS-Konten umgesetzt wird. Dies soll sich jedoch bei der Veröffentlichung bereits geändert haben.
- Ebenfalls ist NTT-Serverless-CI/CD nicht ohne größere Modifikationen in der Lage andere Git-Software als BitBucket, GitHub und GitLab zu unterstützen.
- Unabhängig von CI/CD bietet IaC große Vorteile, jedoch fand noch keine Marktbereinigung statt. Diese Technologien sollten weiter beobachtet werden, da sie extreme Vorteile versprechen.

Was die Zukunft von NTT-Serverless-CI/CD genau bringt, ist bis dato unklar, da es erst im ersten Halbjahr 2021 veröffentlicht werden soll. Weiter ist unklar, ob es zukünftig von einer Community oder von NTT DATA Deutschland GmbH weiterentwickelt wird. Fest steht, dass es weitere Verbreitung innerhalb des Unternehmens findet und bis zur Veröffentlichung weitere Funktionalitäten erhält. Es ist jedoch auch nicht verwunderlich, dass es als neues Framework noch Verbesserungs- und Erweiterungspotential aufweist.

1 https://github.com/dominikampletzer/CI-CD-Verfahren-im-Serverless-Cloud-Computing/tree/master/Kosten-Kalkulation

A Abkürzungsverzeichnis

ALB Application Load Balancer- Verteilt Last/Anfragen gleichmäßig an zugeordnete Applikationen/Container

AWS Amazon Web Services- Internet Service Provider

ARN Amazon Ressource Names- Eindeutiger Bezeichner für AWS-Ressource

CD Continuous Delivery- Konzept zur Auslieferung lauffähiger Code Artefakte

CDN Content Delivery Network- Netzwerk aus Servern, welches statische Daten redundant und in verschiedenen Regionen speichert um Daten mit möglichst geringen Latenzen auszuliefern.

CI Continuous Integration- Konzept zur kontinuierlichen Integration von Code

CI/CD Continuous Integration / Continuous Delivery /Continuous Deployment- Zusammenfassung aller Continuous-Konzepte

CLI Command Line Interface- Kommandozeile/Shell

CRUD Create Read Update Delete- Kurzform für grundlegenden Operationen um Datensätze zu erstellen, lesen, ändern und löschen.

DDOS Distribution Denial Of Service- Angriffsmethode der Cyber-Kriminalität. Verteilte Systeme werden genutzt um eine Infrastruktur Anzugreifen und mit einer enormen Menge an Anfragen zu überlasten.

DevOps Develop and Operations- DevOps ist eine Disziplin welche Entwicklung und Betrieb von Software zusammenfasst.

EC2 Elastic Cloud Computing- AWS-Dienst für Virtuelle Rechnerinstanzen

ECR Elastic Container Registry- AWS-Dienst für die Verwaltung von Container-Abbildungen

ECS Elastic Container Service- AWS-Dienst für die Verwaltung von Container-Instanzen, Cluster und Services

IaC Infrastructure as Code- Infrastruktur als Code

IAM Identity and Access Management- AWS-Dienst für Rollen- und Rechteverwaltung

KMS Key Management Service- AWS-Dienst für die Verwaltung von Verschlüsselungen

LoC	Lines of Code- Code-Zeilen. Kann als Maßeinheit für Quell-Code verwendet werden
RDS	Relational Database Service- AWS-Dienst für relationale Datenbanken
S3	Simple Storage Service- AWS-Dienst für die Verwaltung von Speicher
SCM	Source Code Management- Werkzeug welches den Quell-Code verwaltet. z.B. Git und SVN
SSH	Secure Shell- Standardprotokoll für sichere Remote-Verbindungen
vCPU	virtuel CPU- Virtuelle CPU, darunter wird im AWS-Kontext ein Thread eines Prozessors verstanden
VM	Virtual Machine- Virtuelle Maschine stellt eine gekapseltes Rechnersystem innerhalb eines Rechnersystems da.
VPC	Virtual Private Cloud- AWS-Dienst um logisch isolierte Bereiche zu erzeugen

B Glossar

Angular Framework für Mobile- & Web-Applikationen (Anmerkung des Autors: Hauptverwendung sind Web-Applikationen)

Bucket Korb- Ressource innerhalb von S3. Kann mit einem Festplattenlaufwerk oder Datei-Ordner verglichen werden

Build Wird oft ungenau als Überbegriff für den Build-Prozess, aber auch für den Prozessschritt des Kompilieren und Erstellen von Software verwendet

Canary Neues System, welches ein bestehendes System ablösen soll, jedoch mit geringer Nutzung startet und bei Stabilität diese Nutzung steigert.

CloudFormation AWS-Dienst und Werkzeug für die Nutzung von Infrastructure-As-Code

CloudFront CDN-Dienst von AWS

CodeBuild AWS-Dienst um die Build-Stage und/oder Test-Stage auszuführen

CodePipeline AWS-Dienst um CI/CD-Pipelines zu erstellen

Docker Container-Software

Down-Time Verwendung in dieser Arbeit: Zeit welche ein System nicht erreichbar ist.

Git Open-Source Quell-Code-Verwaltungssystem

Headless Kopflos- Headless im Zusammenhang mit Browser beschreibt Browser-Instanzen ohne grafische Oberfläche

Lambda AWS-Dienst für Serverless Funktionen

Polling Ausdruck für regelmäßige Abfragen. Oft um zu aktualisieren oder Änderungen vorzunehmen

Proof-Of-Concept Stellt i.d.R. einen Prototypen dar, welcher die Machbarkeit eines Konzepts prüfen soll.

Release Bezeichnung für Software, welche veröffentlichungsfähig ist, ausgeliefert wird/wurde und Prozess des Veröffentlichens. Oft ungenaue Verwendung.

Route53 AWS-Dienst für Domänen und DNS

SecretsManager AWS-Dienst um Passwörter/Secrets zu verwalten und zu nutzen

Single-Point-Of-Truth „Der einzigen Punkt der Wahrheit“ - Prinzip, welche verwendet wird, wenn unterschiedliche Versionen oder Redundanzen möglich sind.

Snapshot Schnappschuss- Abbildung eines Ist-Zustandes

C Vorlagen

C.1 Pipeline.yaml

```
Parameters:
  Project:
    Description: Project name
    Type: String
  Repository:
    Description: Repository name
    Type: String
  Suffix:
    Description: Sanitized suffix name, safe to use for CF resource names
    Type: String
  Branch:
    Description: The original unsanitized branch name to reference in Git
    configuration
    Type: String
  Stage:
    Description: "Pipeline stage"
    Type: String
    AllowedValues: ["dev", "delivery"]
  RemotePreviewAccount:
    Description: "Remote child account: development or integration"
    Type: String
    AllowedPattern: (\d{12}|^$)
    ConstraintDescription: must be an AWS account ID
  RemoteDeliveryAccount:
    Description: "Remote delivery child account: production"
    Type: String
    Default: "NO_VALUE"
  ArtifactBucket:
    Description: "Artifact Bucket"
    Type: String
  PipelineKmsKeyArn:
    Description: "Pipeline KMS key"
    Type: String
  PipelineServiceRoleArn:
    Description: Service role ARN to pass to pipeline
    Type: String
  DynamicPipelineCleanupLambdaArn:
    Description: CF Stack cleanup Lambda ARN
    Type: String
Conditions:
  ShouldCreateForDelivery: !Equals [ !Ref Stage, "delivery" ]
  ShouldCreateForDev: !Equals [ !Ref Stage, "dev" ]
Resources:
  CodeBuild:
    Type: AWS::CodeBuild::Project
    Properties:
      Name: !Ref AWS::StackName
      Source:
```

```
48          Type: CODEPIPELINE
49        Environment: # your Build Machine
50          Type: LINUX_CONTAINER
51          ComputeType: BUILD_GENERAL1_SMALL
52          Image: aws/codebuild/standard:4.0
53          EnvironmentVariables: {}
54        Artifacts:
55          Type: CODEPIPELINE
56        ServiceRole: !Ref PipelineServiceRoleArn
57        TimeoutInMinutes: 10 # adjust Timeout to avoid broken builds
58    Pipeline:
59      Type: AWS::CodePipeline::Pipeline
60      Properties:
61        Name: !Join [ '-', [ !Ref Stage, !Ref Repository ] ]
62        ArtifactStore:
63          EncryptionKey:
64            Id: !Ref PipelineKmsKeyArn
65            Type: "KMS"
66          Location: !Ref ArtifactBucket
67          Type: "S3"
68        RestartExecutionOnUpdate: False
69        RoleArn: !Ref PipelineServiceRoleArn
70        Stages:
71          - Name: Checkout-SourceCode
72            Actions:
73              - Name: Checkout-SourceCode
74                ActionTypeId:
75                  Category: Source
76                  Owner: ThirdParty
77                  Provider: <YourSCM> # GitHub | GitLab | BitBucket
78                  Version: "1"
79                Configuration:
80                  Owner: !Ref Project
81                  Repo: !Ref Repository
82                  Branch: !Ref Branch
83                  PollForSourceChanges: true
84                  OAuthToken: "{{resolve:secretsmanager:<YourSecret>:SecretString:<
      YourTokenName>}}"
85                OutputArtifacts:
86                  - Name: "SourceArtifact"
87                RunOrder: 1
88          - Name: "BuildStageForIntAndDev"
89            Actions:
90              - Name: "Build"
91                ActionTypeId:
92                  Category: "Build"
93                  Owner: AWS
94                  Provider: "CodeBuild"
95                  Version: "1"
96                Configuration:
97                  ProjectName: !Ref CodeBuild
98                InputArtifacts:
99                  - Name: "SourceArtifact"
100               OutputArtifacts:
101                 - Name: BuildArtifact
102               RunOrder: 1
103             # Infrastructure and Deploy for Integration and Development
104         - !If
105           - ShouldCreateForDelivery
```

```
106          - Name: "PROD-Approval"
107            Actions:
108              - Name: "ManualApproval"
109                ActionTypeId:
110                  Category: Approval
111                  Owner: AWS
112                  Provider: Manual
113                  Version: "1"
114                Configuration: { }
115                RunOrder: 1
116                Region: !Sub ${AWS::Region}
117          - !Ref AWS::NoValue
118          # Infrastructure and Deploy for Production
119  CleanupForInfrastruktureStack:
120    Type: "AWS::CloudFormation::CustomResource"
121    Condition: ShouldCreateForDev
122    Version: '1.0'
123    Properties:
124      ServiceToken: !Ref DynamicPipelineCleanupLambdaArn
125      RoleArn: !Sub arn:aws:iam::${RemotePreviewAccount}:role/
     CodePipelineServiceRole-${AWS::Region}-${RemotePreviewAccount}
126      Region: !Ref AWS::Region
127      StackName: # Stack Name from Infrastructure creation
```

Code C.1 Vorlage pipeline.yaml

C.2 Check-Liste

- ☐ 4 AWS-Konten anlegen: DevOps, Development, Integration und Production
- ☐ Nutzer mit Administratoren Rechten für alle 4 AWS-Konten mit AWS-IAM einrichten
- ☐ AWS-CLI und Make einrichten
- ☐ Profile mit Zugangsinformationen auf Rechner einrichten
- ☐ NTT-Serverless-CI/CD herunterladen
- ☐ Makefile im *root*-Verzeichnis mit AWS-Konto-Ids, Profilen und Region anpassen
- ☐ NTT-Serverless-CI/CD in AWS einrichten -> *make deployServerlessCiCd*
- ☐ SCM integrieren
  - ☐ Konto für DevOps anlegen [Optional]
  - ☐ Authorisierungs-Token erzeugen
  - ☐ Authorisierungs-Token in SecurityManager hinterlegen
  - ☐ IAM-Richtlinie für SecurityManager erzeugen
  - ☐ IAM-Richtlinie den Pipeline-Service-Rollen hinzufügen (*CI-CD-Bootstrap-PipelineRolesD-PipelineServiceRole-** und *CI-CD-Bootstrap-DevOpsLam-DynamicPipelineMakerLamb**)
  - ☐ Webhook mit Endpunkt API-Gateway erzeugen (Benötigte Nachrichten: *PUSH*, *DELETE*)
  - ☐ SSH public key in Git-Software hintelegen (S3-Bucket *ci-cd-bootstrap-devopsgitmanagement-1lc-keybucket-**)

- ☐ IAM-Richtlinie für *CreateBuildProject* erzeugen
- ☐ IAM-Richtlinie den Pipeline-Service-Rollen hinzufügen (*CI-CD-Bootstrap-PipelineRolesD-PipelineServiceRole-**)
- ☐ CloudFormation-Stacks für Basis-Infrastruktur erzeugen. Z.B. VPC, ECS-Cluster, Domain-Sub-Routes
- ☐ pipeline.yaml in root/build erstellen
  - ☐ Lifecycle-Richtlinien für z.B. S3-Bucket-Objekte, Container-Abbildungen erzeugen
  - ☐ Custom-Ressourcen mit NTT-Serverless-CI/CD-Dynamic-Cleanup-Lambdas und erzeugten Stacks verbunden (i.d.R. nur Development)
  - ☐ Port-freigaben nur über sichere Protokolle HTTPS/SSL
  - ☐ Zusätzlich erzeugte IAM-Richtlinien und IAM-Rollen nach dem Least-Privileged-Prinzip
- ☐ buildspec.yaml in root erstellen
  - ☐ Unit-/End-To-End-Tests integriert
  - ☐ Artefakte/Container-Abbildungen adäquat mit Labeln versehen
  - ☐ Optimiertes Docker-Image für Build-Umgebung erstellt (z.B. Docker-Image mit installiertem Browser)[Optional]
- ☐ AWS-Profile nach NTT-Serverless-CI/CD Integrierung von Rechner entfernt (falls nicht anderweitig benötigt) [Optional]
- ☐ Nach Integration AWS-Trusted-Advisor für zusätzliche (Sicherheits-)Optimierungen verwenden [Optional]

Icon-Quellen

- GitLab Icon von Icon Mafia unter `https://iconscout.com/contributors/icon-mafia`
- GitHub Icon von GitHub unter `https://github.com/logos`
- AWS-Icons von AWS unter `https://aws.amazon.com/de/architecture/icons`
- BitBucket-Icon von Swifticons unter `https://www.flaticon.com/free-icon/bitbucket_214496`
- YAML-Icon von John Gardner unter `https://icon-icons.com/de/symbol/YAML-Alt3/131862`
- JSON-Icon von Papirus Development Team unter `https://icon-icons.com/de/symbol/application-json/92733`
- Git-Folder-Icon von Papirus Development Team unter `https://icon-icons.com/de/symbol/Ordner-orange-git/92940`

Literaturverzeichnis

[Ant17] A. Anthony. *Mastering AWS security*. Packt Publishing, Birmingham, UK, 2017.

[Ant18] A. Anthony. *AWS: Security Best Practices on AWS*. Packt Publishing, Birmingham, UK, 2018.

[Art17] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero und D. A. Tamburri. DevOps: Introducing Infrastructure-as-Code. In *2017 IEEE ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, S. 497–498. 2017.

[Art18] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin und D. A. Tamburri. Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach. In *2018 IEEE International Conference on Software Architecture (ICSA)*, S. 156–165. 2018.

[Bai17] A. Baird, G. Huang, C. Munns und O. Weinstein. Serverless Architectures with AWS Lambda. Techn. Ber., Amazon Web Services, Inc., November 2017.

[Bec99] K. Beck. *extreme Programming explained*. Addison-Wesley, Upper Saddle River, USA, 1999.

[Bec01] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. Martin, S. Mellor, K. Schwaber, J. Sutherland und D. Thomas. Manifest für Agile Softwareentwicklung - Prinzipien hinter dem Agilen Manifest. `https://agilemanifesto.org/iso/de/principles.html`, 2001. Stand: 2020-10-30.

[Bid18] D. Bider. RFC 8308 - Extension Negotiation in the Secure Shell (SSH) Protocol. Techn. Ber., Internet Engineering Task Force (IETF), März 2018.

[Bit18] Bitkom. Welches sind die wichtigsten IT-Trends des Jahres 2018? `https://de.statista.com/statistik/daten/studie/808775/umfrage/die-wichtigsten-trends-in-der-itk-branche/`, Februar 2018. Stand: 2020-11-18.

[Bit20] Bitkom. Ausgaben für IT-Sicherheit in Deutschland in den Jahren 2017 bis 2019 und Prognose bis 2021 (in Milliarden Euro). `https://de.statista.com/statistik/daten/studie/1041736/umfrage/ausgaben-fuer-it-security-in-deutschland`, Oktober 2020. Stand: 2020-11-18.

[BSI20] Die Lage der IT-Sicherheit in Deutschland 2020. Techn. Ber., Bundesamt für Sicherheit in der Informationstechnik (BSI), September 2020.

[Cap20] Capgemini. Anteil der Unternehmen in der DACH-Region, die folgende Technologien nutzen oder derzeit implementieren. In *Studie IT-Trends 2020*, S. 33. 2020.

[Che17] H. Cheung, J. Faerman, B. Iyer und J. Levine. Infrastructure as Code. Techn. Ber., Amazon Web Services, Inc., Juli 2017.

[Dab20] N. Dabit. *Full Stack Serverless*. O'Reilly Media, Sebastopol, CA, 2020.

[Dob20] J. Dobies und J. Wood. *Kubernetes Operators*. O'Reilly Media Inc., Sebastopol, USA, 2020.

[Eiv17] A. Eivy und J. Weinman. Be Wary of the Economics of „Serverless" Cloud Computing. *IEEE Cloud Computing*, 4(2):6–12, 2017.

[Fow06] M. Fowler. Continuous Integration. `https://martinfowler.com/articles/continuousIntegration.html`, Mai 2006. Stand: 2020-10-08.

[Fow13] M. Fowler. ContinuousDelivery. `https://www.martinfowler.com/bliki/ContinuousDelivery.html`, Mai 2013. Stand: 2020-10-08.

[Gue19] M. Guerriero, M. Garriga, D. A. Tamburri und F. Palomba. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, S. 580–589. 2019.

[Kan17] V. Kantsev. *Implementing DevOps on AWS*. Packt Publishing, Birmingham, UK, 2017.

[Las20] B. Laster. *Continuous Integration vs. Continuous Delivery vs. Continuous Deployment - Second Edition*. O'Reilly Media, Sebastopol, CA, 2020.

[Loi20] M. Loibl und F. Ibrahimov. div. firmeninterne Präsentationen. unveröffentlicht, 2020. NTT DATA Deutschland GmbH.

[Ma19] R. Ma. Managing and governing multi account AWS environments using AWS Organizations. In *RE:INFORCE 2019*. 2019.

[Mag19] R. Magoulas und C. Guzikowski. O'Reilly serverless survey 2019: Concerns, what works, and what to expect. `https://www.oreilly.com/radar/oreilly-serverless-survey-2019-concerns-what-works-and-what-to-expect`, November 2019. Stand: 2020-10-08.

[Mar09] R. Martin. *Clean Code*. mitp Verlags GmbH & Co. KG, 2009.

[McC17] J. McConnell, K. Thirugnanasambandam und S. Cardenas. Git Webhooks with AWS Services. Techn. Ber., Amazon Web Services, Inc., September 2017.

[Mis18] A. Mistry. *Expert AWS Development*. Packt Publishing, Birmingham, UK, 2018.

[Pot20] B. Potter, B. Shinn, B. Johnson, B. Pogson, D. Boyd, D. Walker, P. Hawkins und S. Elmalk. Security Pillar. Techn. Ber., Amazon Web Services, Inc., Juli 2020.

[Pri18] P. Priyam. *Cloud security automation*. Packt Publishing, Birmingham, UK, 2018.

[Pur17] H. Puripunpinyo und M. H. Samadzadeh. Effect of optimizing Java deployment artifacts on AWS Lambda. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, S. 438–443. 2017.

[Pyt] Python.org. Sunsetting Python 2. `https://www.python.org/doc/sunset-python-2/`. Stand: 2020-10-13.

[Rah19] A. Rahman, C. Parnin und L. Williams. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *2019 IEEE ACM 41st International Conference on Software Engineering (ICSE)*, S. 164–175. 2019.

[Ros17] S. Rossel. *Continuous integration, delivery, and deployment*. Packt Publishing, Birmingham, UK, 2017.

[Shi16] G. Shipley und G. Dumpleton. *OpenShift for Developers*. O'Reilly Media, Sebastopol, CA, 2016.

[Sin19] C. Singh, N. S. Gaba, M. Kaur und B. Kaur. Comparison of Different CI/CD Tools Integrated with Cloud Platform. In *2019 9th International Conference on Cloud Computing, Data Science Engineering (Confluence)*, S. 7–12. 2019.

[Son15] M. Soni. End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery. In *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, S. 85–89. 2015.

[Spi12] D. Spinellis. Don't Install Software by Hand. *IEEE Software*, 29(4):86–87, 2012.

[Sta17] D. Stacy, M. Prudnikov, A. Khan und X. Shen. Practicing Continuous Integration and Continuous Delivery on AWS. Techn. Ber., Amazon Web Services, Inc., Juni 2017.

[Tha19] M. Thakkar, B. McNamara, M. Brooker und O. Surkatty. Security Overview of AWS Lambda. Techn. Ber., Amazon Web Services, Inc., März 2019.

[Tod16] D. Todorov und Y. Ozkan. AWS Security Best Practices. Techn. Ber., Amazon Web Services, Inc., August 2016.

[Tov20] K. Tovmasyan. *Mastering AWS CloudFormation*. Packt Publishing, Birmingham, UK, 2020.

[Veh18] J. Vehent. *Securing DevOps*. Manning Publications, Shelter Island, NY, 2018.

[Vil16] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano und M. Lang. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *2016 16th IEEE ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, S. 179–182. 2016.

[Wei17] J. Weinman. The Evolving Cloud. *IEEE Cloud Computing*, 4(3):4–6, 2017.

[Ylo06] T. Ylonen. RFC 4251 - The Secure Shell (SSH) Protocol Architecture. Techn. Ber., Internet Engineering Task Force (IETF), Januar 2006.

[Zam18] B. Zambrano. *Serverless design patterns and best practices*. Packt Publishing, Birmingham, UK, 2018.