

Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Masterarbeit

Automatisierte Archivierung und Wiederinbetriebnahme cloud-basierter Anwendungen

Gerd Matheis

Studiengang: Softwaretechnik
Prüfer/in: Prof. Dr. Dr. F. Leymann
Betreuer/in: Lukas Harzenetter, M.Sc.

Beginn am: 4. März 2019
Beendet am: 4. September 2019

Kurzfassung

Mit der Einführung von Cloud-Infrastrukturen, auf denen Anwendungen entwickelt und bereitgestellt werden können, hat sich die Entwicklung von Software maßgeblich verändert:

Während vor einigen Jahren noch der Entwickler oder gar der Kunde selbst die benötigte Infrastruktur für die jeweilige Software bereitstellen musste, werden heute viele web-basierte Anwendungen direkt für die Cloud entwickelt. Dies bringt den Vorteil, dass sich weder der Kunde noch der Softwarehersteller um die Wartung der Infrastruktur kümmern müssen. Darüber hinaus lassen sich Updates viel einfacher verteilen.

Diese Vorteile führen dazu, dass immer mehr und immer größere Anwendungen direkt für die Cloud entwickelt und dort betrieben werden. Dabei bestehen die Anwendungen meistens aus einem vielschichtigen und komplexen Netz so genannter *Microservices*. Dadurch wird es immer schwieriger, die kleinen Bestandteile einer großen Anwendung aufeinander abzustimmen. Um dieses Problem zu lösen, wurden Software-Systeme, wie beispielsweise *OpenTOSCA*, entwickelt, welche das Einrichten und Verwalten von Cloud-Systemen vereinfachen oder ganz automatisieren sollen.

Cloud-basierte Software wird von den Cloud-Betreibern nach Verbrauch abgerechnet. Das bedeutet, dass der Betreiber der Software nur für die Ressourcen zahlt, welche er auch in Anspruch nimmt. Wird aber eine Cloud-Anwendung nur zeitweise benötigt, fallen auch außerhalb der Nutzungszeiten Kosten an, wenn die Anwendung weiterhin in der Cloud bereit steht. Werden die Komponenten jedoch abgeschaltet, kann es bei zustand-behafteten Komponenten zu Datenverlust kommen, wodurch nur zwei Optionen bleiben: Die entsprechenden Komponenten werden weiterhin betrieben und verursachen weiterhin Kosten oder es muss ein Abschaltmechanismus in die Cloud-Anwendung einprogrammiert werden, welcher den Zustand der Komponenten konserviert und beim nächsten Start der Anwendung wieder lädt.

Es ist daher wünschenswert, dass Systeme wie *OpenTOSCA* in der Lage sind, laufende Cloud-Anwendungen zu archivieren und zu einem späteren Zeitpunkt in exakt diesem Zustand wieder auszuführen.

Ziel dieser Arbeit ist es, einen Prototypen für ein solches System zu entwickeln und den Ansatz anhand eines Minimal-Beispiels zu testen. Hierbei wird ein besonderer Fokus auf zustand-behaftete Komponenten wie *Message-Queues* gelegt, da genau diese sich bisher als problematisch darstellen. Weiterhin sollte die Archivierung durch ein transaktionales Verfahren abgesichert werden, sodass nur ein erfolgreiches Archivieren der Anwendung zur tatsächlichen Löschung aus der Cloud führt. Die entsprechenden Hindernisse und Ergebnisse werden in diesem Dokument festgehalten.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Motivation	1
1.2. Ziel der Arbeit	2
1.3. Gliederung	3
2. Grundlagen	5
2.1. Cloud / Cloud-Computing	5
2.2. Cloud-Orchestrierungs-Software	7
2.3. Grundlegende Konzepte	14
3. Verwandte Arbeiten	17
3.1. A Survey of Migration Mechanisms of Virtual Machines	17
3.2. Algorithms for automated live migration of virtual machines	19
3.3. Stateful Process Migration for Edge Computing Applications	21
3.4. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing	23
3.5. Freezing and Defrosting Cloud Applications: Automated Saving and Restoring of Running Applications	25
4. Konzept	29
4.1. Analyse bestehender Ansätze verwandter Arbeiten	29
4.2. Konzeption des Prototyps	32
5. Entwicklung des Prototyps	35
5.1. Design-Entscheidungen	35
5.2. Architektur des Prototyps	36
5.3. Erkenntnisse und Hindernisse	44
6. Zusammenfassung und Ausblick	47
Literaturverzeichnis	49
A. Anhang	55
A.1. Bildrechte	55

Abbildungsverzeichnis

2.1. Public- / Hybrid- und Private-Cloud	6
2.2. AWS Logo	6
2.3. Schematische Darstellung einer Message-Oriented Middleware	10
2.4. OpenTOSCA Module	11
2.5. Continuous Integration / Delivery / Deployment & DevOps	14
2.6. Schematische Darstellung: PM & VM	16
3.1. Anwendung der Push- / Pull-Strategie	20
3.2. Vergleich: Downtime durch Migration	21
3.3. Schematische Darstellung von SnowFlock	24
3.4. OpenTOSCA im Detail	26
4.1. CloudFridge: Archivierungs-Schema	34
4.2. CloudFridge: Reprovisionierungs-Schema	34
5.1. CloudFridge Logo	35
5.2. Abhängigkeitsgraph des Prototyps	36
5.3. Amazon Lambda und Amazon SQS	46

Verzeichnis der Algorithmen

5.1. Discovery-Algorithmus für die Suchen nach archivierbaren Elementen	39
5.2. Routine für das Serialisieren der einzelnen FreezableObjects	39
5.3. Routine für das transaktionale Herunterfahren der Komponenten	40
5.4. Routine für das Laden und die Deserialisierung der Komponenten	41
5.5. Routine für das transaktionale Starten der Komponenten	41

Abkürzungsverzeichnis

- Amazon CF** Amazon CloudFormation. 13
- Amazon EC2** Amazon Elastic Compute Cloud. 6, 48
- Amazon S3** Amazon Simple Storage Service. 6
- Amazon RDS** Amazon Relational Database Service. 2, 6, 9, 44, 45
- Amazon SQS** Amazon Simple Queue Service. vii, 2, 44, 45, 46
- API** Application Programming Interface / Programmierschnittstelle. 30, 42, 44
- AWS** Amazon Web Services. vii, 2, 5, 6, 8, 13, 29, 30, 31, 32, 33, 38, 42, 44, 45, 46, 48
- CI** Continuous Integration. 14
- CloudFridge** Prototyp. vii, 2, 3, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 47, 48
- CPU** Central Processing Unit / Prozessor. 17, 24, 30
- laaS** Infrastructure as a Service. 8
- ID** Identifikator. 22, 43, 44
- IP** Internet-Protokoll. 12, 26
- IPC** Inter Process Communication / Inter-Prozess-Kommunikation. 9, 22
- JSON** JavaScript Object Notation. 13, 43, 48
- LAN** Local Area Network / lokales Netz. 18
- MOM** Message-Oriented Middleware. vii, 9, 10, 11, 15
- OS** Operating System / Betriebssystem. 22, 26, 30, 31
- PaaS** Platform as a Service. 8
- PM** physische Maschine. vii, 15, 18, 19, 20, 24, 30
- PoC** Proof of Concept. 2, 20, 26, 33, 42, 44, 45, 48
- RAM** Random Access Memory / Arbeitsspeicher. 17, 18, 24, 25, 30
- SDK** Software Development Kit. 35, 44, 45, 46, 48
- TOSCA** Topology and Orchestration Specification for Cloud Applications. iii, vii, 1, 7, 8, 11, 12, 13, 17, 25, 26, 27, 30, 31, 32, 46, 48
- TTL** Time to live / Lebenszeit. 9

UE User Equipment / Benutzer-Equipment. 21, 22

VCP Virtueller Cloud Pool. 18, 21

VM virtuelle Maschine. vii, xii, 3, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 29, 30, 31

VMM Hypervisor / VM-Monitor. 15, 16

VPN Virtuelles Privates Netzwerk. 18

WAN Wide Area Network / Weitverkehrsnetz. 18, 29

YAML YAML Ain't Markup Language. 12, 13

1. Einleitung

1.1. Motivation

Durch das Konzept des *Cloud-Computings* werden immer mehr Anwendungen basierend auf Cloud-Architekturen entwickelt, um sich die Vorteile der neuen Technologie zu Nutze zu machen. Ein Vorteil dabei ist, dass die benötigten Ressourcen bei einem Cloud-Anbieter gemietet werden können. Dabei werden nur Ressourcen in Rechnung gestellt, die auch genutzt werden [MSS16]. Durch adaptive (*De-*) *Provisionierung* der Ressourcen können diese auch an unvorhersehbare Auslastungsschwankungen angepasst werden, sodass kaum Über- oder Unter-*Provisionierung* auftritt [Sta11] [Wag12].

Mit zunehmender Komplexität und Vielschichtigkeit der Anwendungen wird es schwierig, den Überblick über die einzelnen Komponenten zu behalten. Dieses Problem verschärft sich noch, wenn die Anwendung aus einem vielschichtigen und komplexen Netz sogenannter *Microservices* besteht [Hug15] [Ham14]. Dabei ist das Architekturmodell der *Microservices* insbesondere in der Cloud interessant: Verschiedene atomare Module können, unabhängig voneinander, *vertikal skaliert* werden [LF14]. Die einzelnen Module lassen sich zudem über verschiedene Cloud-Anbieter hinweg verteilen. Daher werden Werkzeuge wie *OpenTOSCA*¹ benötigt, um das Deployment solcher Anwendungen durch einen automatisierten Prozess zu erleichtern [WBK+18] [QCW15]. Weiterhin kann *OpenTOSCA* auch die *Provisionierung* der Ressourcen automatisieren [ZBF+17].

Für die Mehrheit der Softwaresysteme reicht ein komplett *automatisiertes Deployment* in die Cloud aus, da die Software ständig betrieben und von Konsumenten genutzt wird. Allerdings gibt es Fälle, in denen die Software nur zu bestimmten Zeiten benötigt wird [NHS+18]. Ein Beispiel dafür ist eine *Cloud*-gestützte Anwendung zur Koordinierung von Einsatzkräften bei Großschadenslagen.

Das Problem von ungenutzten Cloud-Systemen besteht hauptsächlich im Abrechnungsmodell der Cloud-Betreiber: Dieses sieht vor, dass nur die verbrauchten Ressourcen in Rechnung gestellt werden. Allerdings erzeugen auch nicht genutzte Komponenten eine Grundlast, die entsprechende Ressourcen benötigt, wodurch unnötige Kosten entstehen [AWS19b].

Darüber hinaus belegen die ungenutzten Komponenten Ressourcen, die anderweitig genutzt werden könnten und erhöhen unnötig den Stromverbrauch der Server. Das mag im Falle einer einzelnen Anwendung nur eine verschwindend geringe Auswirkung haben aber auf das gesamte Rechenzentrum gerechnet kann der Stromverbrauch sichtbar vermindert werden, wenn diese brachliegenden Komponenten abgeschaltet würden.

¹OpenTOSCA - <https://www.opentosca.org>

Das Abschalten ist allerdings nur im Fall von *zustand-freien* Komponenten eine triviale Aufgabe, wohingegen *zustand-behaftete* Komponenten wie *Message-Queues* diese Aufgabe erschweren: Im Falle der *Message-Queue* gibt es verschiedene Anforderungen, welche von dieser erfüllt werden müssen. Beispielsweise dürfen die Nachrichten nur einmal an jeden Empfänger gesendet werden (*exactly-once*). Daraus ergibt sich, dass die *Message-Queue* im Betrieb neben den eigentlichen Nutzdaten Metadaten halten muss, um diese Anforderungen sicherzustellen. Wird die Cloud-Anwendung beendet und die *Message-Queue* gelöscht, gehen genau diese Metadaten verloren und müssen beim Wiedereinschalten der Anwendung wiederhergestellt werden.

Um das Abschalten einer Cloud-Anwendung auch für große Softwaresysteme rentabel zu gestalten, muss der Prozess des Archivierens sowie die Wiederinbetriebnahme ebenfalls automatisiert werden, weswegen sich eine Integration in ein bestehendes System zum automatisierten Deployment von Cloud-Anwendungen empfiehlt.

Ziel ist es, dass sich Cloud-Anwendungen auf Knopfdruck an- und wieder abschalten lassen, wodurch sich sowohl IT-Ressourcen als auch natürliche Ressourcen schonen lassen. Darüber hinaus kann eine gute Archivierung auch weitere Möglichkeiten eröffnen: Sofern die Archivierung auf einem möglichst abstrakten Modell basiert, ist es denkbar, durch entsprechende Transformationen der archivierten Daten, die *Cloud*-Anwendung in verschiedene *Cloud*-Systeme zu deployen. Dies beinhaltet nicht nur den vollständigen Umzug von Cloud-Systemen von einem Anbieter zum Nächsten, sondern auch den partiellen Umzug. Wichtige Voraussetzung hierbei ist, dass die Cloud-Anwendung nicht auf herstellereigene Features setzt, wobei auch diese sich durch entsprechend komplexere automatisierte Transformationen auflösen lassen sollten.

1.2. Ziel der Arbeit

Ausgehend von den in Abschnitt 1.1 vorgestellten Problemen besteht das Ziel darin, eine Lösung zu schaffen, mit der es möglich ist, Cloud-Anwendungen zu archivieren und den archivierten Zustand wiederherzustellen. Dabei wird im Rahmen dieser Arbeit zunächst die Machbarkeit des Ansatzes geprüft und, sofern diese Überprüfung positiv ausfällt, anhand eines Minimal-Beispiels die Machbarkeit bewiesen.

In dieser Arbeit wird, aus oben genannten Gründen, ein Fokus darauf gelegt, für den Proof of Concept (PoC) eine Auswahl von *zustand-behafteten* Komponenten der *AWS-Cloud* zu archivieren:

1. **Datenbanken:** *Amazon DynamoDB*², *Amazon RDS*³
2. **Message-Queues:** *Amazon SQS*⁴
3. **Caching-Systeme:** *Amazon ElastiCache*⁵

Dazu werden in der *Amazon-Cloud* manuell Instanzen der genannten Systeme angelegt und mit Beispieldaten versehen. Anschließend werden diese vom *Prototyp (CloudFridge)* archiviert und wieder in Betrieb genommen.

²Amazon DynamoDB - <https://aws.amazon.com/de/dynamodb>

³Amazon Relational Database Service - <https://aws.amazon.com/de/rds>

⁴Amazon Simple Queue Service - <https://aws.amazon.com/de/sqs>

⁵Amazon ElastiCache - <https://aws.amazon.com/de/elasticache>

1.3. Gliederung

Die Arbeit ist in die folgenden Kapitel unterteilt:

- **Kapitel 2 - Grundlagen:** In diesem Kapitel werden die nötigen Grundlagen erklärt, die zum Verständnis der Arbeit benötigt werden. Die Grundlagen sind dabei bewusst breit aufgestellt, werden aber zeitgleich zu Themenblöcken zusammengefasst, sodass sich einzelne Abschnitte auch überspringen lassen.
- **Kapitel 3 - Verwandte Arbeiten:** Im Rahmen dieser Arbeit werden Konzepte genutzt, die anderen wissenschaftlichen Arbeiten entnommen sind. Diese Arbeiten werden zusammengefasst vorgestellt, um einen Überblick über die Konzepte anderer zu geben und die eigenen Konzepte damit vergleichen zu können. Im Fokus stehen dabei Arbeiten rund um das Thema Migration von *virtuellen Maschinen (VM)* und *zustand-behafteten* Komponenten.
- **Kapitel 4 - Konzept:** In diesem Kapitel wird das Konzept für den Prototyp zur Archivierung cloud-basierter Anwendungen vorgestellt. Hierzu werden zunächst die Ansätze der verwandten Arbeiten zusammengefasst und miteinander verglichen. Anschließend werden die Konzepte, die zum Anwendungsfall dieser Arbeit passen, hervorgehoben. Dabei wird auch beleuchtet wo die Unterschiede bestehen und wo die Konzepte abgewandelt werden müssen.
- **Kapitel 5 - Entwicklung des Prototyps:** Dieses Kapitel beschäftigt sich mit den Details des Prototyps (CloudFridge). Zunächst werden kurz die Design-Entscheidungen vorgestellt und begründet. Anschließend wird die Funktionsweise der CloudFridge detailliert vorgestellt. Hierbei wird darauf geachtet, die Kern-Konzepte der Implementierung sowie die Gedanken dahinter zu beleuchten. Abgeschlossen wird das Kapitel durch eine kurzes Fazit, in welchem auf die Hindernisse eingegangen wird, die während der Implementierung aufgetreten sind. Darüber hinaus wird der Ansatz der CloudFridge reflektiert und bewertet.
- **Kapitel 6 - Zusammenfassung und Ausblick:** Die Ergebnisse der Arbeit werden schließlich zusammengefasst dargestellt. Darüber hinaus zeigt dieses Kapitel auf, wie die Arbeit an der CloudFridge fortgeführt werden kann.

2. Grundlagen

Im Folgenden werden die Begriffe erklärt, die im Rahmen dieser Arbeit genutzt und deren Verständnis vorausgesetzt wird.

2.1. Cloud / Cloud-Computing

Das Wort *Cloud* wird gerne als Bezeichnung für Cloud-Speicher genutzt [Mor17]. Dabei bezeichnet die Cloud aber allgemein nur ein

"[...] Netzwerk von Servern [...], die alle eine eigene Funktion erfüllen.", [MS]

Diese Server sind miteinander verbunden und stellen ihre Ressourcen zur Verfügung. Neben dem Speicher sind das unter anderem auch die Rechenkapazität und die Möglichkeit, Anwendungen auszuführen. Diese Ressourcen werden Nutzern in Form verschiedener Cloud-Dienste zur Verfügung gestellt.

Das Cloud-Netzwerk kann in verschiedenen Formen betrieben werden, wobei die drei wichtigsten Vertreter Public-, Private- und Hybrid-Cloud sind [FLR+14] [MS] [Mat17].

- **Public Cloud¹:**
Als *öffentliche Cloud* werden IT-Ressourcen bezeichnet, die einer großen Zahl von Nutzern zur Verfügung stehen. Dabei kann grundsätzlich jeder die Ressourcen der Cloud mieten. Als Beispiel für diese Cloud dienen die Amazon Web Services oder Microsoft Azure.
- **Private Cloud²:**
Im Gegensatz zur *öffentlichen Cloud* steht die *private Cloud* nur einer beschränkten Zahl von ausgewählten Nutzern zur Verfügung. Grund dafür ist häufig, dass in dieser Cloud besonders sensible Daten verarbeitet werden, die nicht für die Allgemeinheit bestimmt sind. Ein Beispiel hierfür ist ein Cloud-Dienst, der lediglich aus dem Firmen-Netzwerk zu erreichen ist.
- **Hybrid Cloud³:**
Die *hybride Cloud* ist eine Kombination einer *privaten Cloud* mit einer *öffentlichen Cloud*. Dabei werden die öffentlichen Ressourcen an Kunden vermietet, wohingegen die privaten Ressourcen nur ausgewählten Nutzern zur Verfügung stehen.

¹Public Cloud - https://www.cloudcomputingpatterns.org/public_cloud

²Private Cloud - https://www.cloudcomputingpatterns.org/private_cloud

³Hybrid Cloud - https://www.cloudcomputingpatterns.org/hybrid_cloud



Abbildung 2.1.: Public- / Hybrid- und Private-Cloud
Grafik-Vorlage entnommen aus: [ALT]

2.1.1. Amazon Web Services (AWS)

AWS ist der Cloud-Dienst von Amazon, der im Jahre 2006 erstmals angeboten wurde [AWS19c]. Die AWS-Cloud besteht dabei aus verschiedenen Modulen, die nach Belieben zu einer Cloud-Anwendung kombiniert werden können.



Abbildung 2.2.:
AWS Logo, Grafik
entnommen aus: [AWS19c]

Um beispielsweise eine einfache Gewinnspiel-Webseite über die *Cloud* zu verwirklichen, benötigt man eine Komponente, welche die Webseite darstellt und eine Komponente, welche die übermittelten Teilnehmer-Daten speichert. Die Bereitstellung der Webseite erfolgt hierbei durch einen Amazon Simple Storage Service (Amazon S3)⁴-Bucket oder durch einen virtuellen Server wie Amazon Elastic Compute Cloud (Amazon EC2)⁵. Die Daten können beispielsweise in einer relationalen Datenbank wie Amazon Relational Database Service (Amazon RDS)⁶ gespeichert werden. Das Angebot ist inzwischen deutlich vielfältiger als relationale Datenbanken, Datenspeicher und virtuelle Webserver und stellt damit für viele Geschäftsbereiche vorgefertigte Module bereit [AWS19a].

Damit sich Anwendungen für die Nutzung der AWS authentifizieren können, stellt Amazon ein eigenes Konzept bereit: Jedes AWS-Konto verfügt über eine Haupt-Anmeldung, über die das Konto verwaltet werden kann [AWS19e]. Dieses stellt somit den administrativen Zugriff bereit. Daneben lassen sich weitere Zugänge für das Konto erstellen, bei denen die jeweiligen Rechte angepasst werden können. Für die Nutzung der AWS-Dienste aus Anwendungen heraus kann ein Zugriffsschlüssel für diese Anwendung generiert werden, welcher aus zwei Teilen besteht:

- **AWS AccessKey:**
Der *AccessKey* stellt die Anmelde-Kennung dar. Diese ist innerhalb der AWS-Dienste eindeutig und mit dem Benutzernamen einer Anmeldung vergleichbar.
- **AWS SecretKey:**
Der *SecretKey* ist das logische Gegenstück zum *AccessKey* und ist vergleichbar mit dem Passwort der Benutzeranmeldung.

⁴Amazon S3 - <https://aws.amazon.com/de/s3>

⁵Amazon EC2 - <https://aws.amazon.com/de/ec2>

⁶Amazon RDS - <https://aws.amazon.com/de/rds>

Durch das Konzept der App-Zugänge wird vermieden, dass potentielle Angreifer Zugriff auf den vollständigen Account erhalten, da nur auf die jeweils freigegebenen Ressourcen zugegriffen werden kann.

2.2. Cloud-Orchestrierungs-Software

Cloud-Anwendungen werden mit zunehmender Größe unübersichtlich. Dieses Problem wird noch gravierender, wenn die verschiedenen Komponenten bei unterschiedlichen Anbietern laufen. Darüber hinaus kommt es vor, dass von einzelnen Komponenten direkt mehrere Instanzen gestartet werden müssen, weil die Komponente bekanntermaßen stark ausgelastet wird. Entsprechend komplex und fehleranfällig wird ein manuelles Deployment einer *Cloud-Anwendung*. Daher ist das *automatisierte Deployment* und Management von *Cloud-Anwendungen* erforderlich, welches durch *Cloud-Orchestrierungs-Software*, wie beispielsweise *OpenTOSCA*, ermöglicht wird [Son15].

2.2.1. Skalierung: Horizontal / Vertikal

Wenn die Zahl der Nutzer eines Systems steigt, erhöht sich damit auch der Ressourcen-Verbrauch des Systems. Daher ist es wichtig, dass die einzelnen Komponenten sich gut an die jeweilige Auslastung anpassen lassen. Diese Anpassung wird als *Skalierung* bezeichnet [IBM14] [LF14]. Die Skalierung funktioniert dabei in beide Richtungen, d. h. die Ressourcen können hoch und runter skaliert werden.

Grundsätzlich gibt es zwei Arten der Skalierung:

- **Horizontale Skalierung:**
Bei der *horizontalen Skalierung* wird die Komponente sinnbildlich verbreitert. Das bedeutet, dass die Komponente kopiert wird und dann mehrere Instanzen davon zur Verfügung stehen. So kann die Last zwischen den einzelnen Instanzen aufgeteilt werden.
- **Vertikale Skalierung:**
Bei der *vertikalen Skalierung* werden einer Komponente mehr Ressourcen zur Verfügung gestellt. Beispielsweise kann einem Web-Server mehr RAM zur Verfügung gestellt werden. Durch die Veränderung der zugeteilten Ressourcen kann die einzelne Komponente sich an die jeweiligen Anforderungen anpassen.

2.2.2. (De-) Provisionierung

Der Begriff *Provisionierung* bezeichnet

"[...] die automatisierte Bereitstellung von IT-Ressourcen [...] [, welche] durch Deprovisionierung wieder freigegeben werden [können].", [GWL19b]

Am Beispiel eines *Cloud*-Anbieters bedeutet dies, dass sich verschiedene Module automatisch hochfahren und wieder herunterfahren lassen. Dies geschieht meist durch eine *Cloud-Orchestrierungs-Software* oder einen entsprechenden Mechanismus für *horizontale oder vertikale Skalierung*.

2.2.3. Funktionsweise der Cloud-Orchestrierungs-Software

Für die *Provisionierung und Deprovisionierung* der Komponenten benötigt die *Cloud-Orchestrierungs-Software* eine Definition, aus welchen Komponenten die Anwendung besteht und wo diese zu deployen sind. Darüber hinaus muss definiert werden, wie die einzelnen Komponenten zusammen hängen und arbeiten. Aus diesen Informationen kann die Software einen Deployment-Plan ableiten und die Software in die *Cloud* veröffentlichen [WBK+18]. Hierbei wird zwischen zwei Deployment-Modellen unterschieden[EBF+17]:

- **Deklaratives Deployment-Modell:**

Beim *deklarativen Deployment-Modell* wird die Struktur der zu deployenden Anwendung beschrieben. Das heißt, es werden alle Komponenten sowie deren Abhängigkeiten definiert.

Durch diese Definition wird festgelegt, in welchem Zustand sich die Anwendung nach dem Deployment befinden soll. Entsprechend kann die Software für das automatisierte Deployment daraus die nötigen Deployment-Schritte ableiten und die Software eigenständig deployen.

- **Imperatives Deployment-Modell:**

Das *imperative Deployment-Modell* beschreibt, welche Schritte nötig sind, um eine *Cloud-Anwendung* zu deployen. Für jede Komponente werden dabei alle nötigen Schritte in der korrekten Reihenfolge angegeben, die von der *Deployment-Software* durchgeführt werden müssen, um die jeweilige Komponente zu deployen.

Dabei muss auch angegeben werden, in welcher Reihenfolge die einzelnen Komponenten deployt werden müssen.

2.2.4. Topology and Orchestration Specification for Cloud Applications (TOSCA)

Die *Topologie- und Orchestrierungs-Spezifikation für Cloud-Anwendungen* ist ein OASIS Standard, der zur Modellierung von *Cloud-Anwendungen* geschaffen wurde [TOSCA-v1.0] [Kic19]. Dabei definiert der Standard eine Sprache, mit der sich die Komponenten einer *Cloud-Anwendungen* und deren Beziehungen untereinander als *Service-Topologie* beschreiben lassen. Darüber hinaus werden Management-Prozesse beschrieben, welche die Komponenten über Orchestrierungs-Prozesse erstellen oder ändern.

Ziel des Standards ist es, dass *Cloud-Anwendungen* unabhängig vom jeweiligen *Cloud-Anbieter* deployt werden können und damit der so genannte *Vendor LockIn* verhindert wird.

2.2.5. zustand-freie und -behaftete Komponenten

Für die Bereitstellung von Anwendungen stehen dem Software-Betreiber verschiedene Dienste zur Verfügung [Mat17] [HBKL19]. Im Rahmen von Infrastructure as a Service (IaaS) werden Infrastruktur-Komponenten wie beispielsweise virtuelle Server angeboten. Daneben gibt es aber auch Platform as a Service (PaaS), worunter Plattform-Dienste wie beispielsweise Datenbank-Instanzen fallen. Die Kombination der verschiedenen Diensten ermöglichen letztlich den Betrieb der Software.

Hierbei gibt es Komponenten, wie beispielsweise *AWS Lambda*⁷, welche **zustand-frei** sind: Das bedeutet, dass eine AWS Lambda Instanz jederzeit gestartet und gestoppt werden kann, ohne dass dadurch Daten verloren gingen. Die Komponente verarbeitet zwar Daten, speichert diese aber nicht lokal sondern gibt die Daten an andere Komponenten weiter. Entsprechend wird der Zustand einer Software nie von *zustand-freien* Komponenten beschrieben

Andere Komponenten wie beispielsweise Amazon RDS hingegen enthalten oder speichern die Daten einer Anwendung. Ohne diese Daten kann die Anwendung nicht korrekt arbeiten, weswegen die Daten den Zustand der Software beschreiben. Daher bezeichnet man diese Komponenten als **zustand-behaftet** und enthaltenen Daten müssen vor Beendigung der Komponente gespeichert werden.

Im Folgenden wird auf zwei weitere **zustand-behaftete** Komponenten eingegangen, die im Rahmen dieser Arbeit für Erklärungen und das Aufzeigen von Hindernissen herangezogen werden.

Caching-Systeme

Als Cache wird ein System bezeichnet, welches bestimmte Daten temporär speichert und dadurch einen schnellen Zugriff auf diese erlaubt [AWS19g].

Die Daten, die der Cache hält, können dabei aber veralten, da diese durch den Cache nicht aktuell gehalten werden. Entsprechend werden die Daten im Cache häufig mit einer Time to live / Lebenszeit (TTL) versehen, die angibt, wie lange die Daten gültig bleiben.

Caches finden zumeist dann Einsatz wenn Daten häufig benötigt werden oder die Abfrage der Daten zeitaufwendig ist. In beiden Fällen würden Ressourcen verbraucht, die man anderweitig nutzen könnte. Der Cache hingegen ist so optimiert, dass er möglichst schnell die zu einer Anfrage abgespeicherte Information abrufen und zurückgeben kann.

Im Falle des *ReflectionManagers* werden beispielsweise die Informationen aller im *Prototyp* enthaltenen Klassen abgespeichert, weil das Abrufen dieser Informationen sehr zeitaufwendig ist. Der Cache hält diese Informationen im Speicher und ermöglicht durch eine intelligente Datenstruktur einen sehr schnellen Zugriff. Da diese Daten nur durch eine Umprogrammierung des *Prototyps* veralten, muss der Cache diese auch nur einmal zum Programmstart abrufen.

Die Daten müssen jedoch für eine möglichst lange Zeit gültig bleiben, damit sich ein Cache lohnt: Angenommen es werden durch eine Anfrage Daten in einer Datenbank abgefragt und im Cache abgelegt, dann sollten diese Daten wenigstens für eine weitere Anfrage genutzt werden können. Verlieren die Daten aber bis dahin ihre Gültigkeit, müssen die aktuellen Daten wieder aus der Datenbank abgefragt werden und der Cache wäre überflüssig.

Message-Oriented Middleware (MOM) / Message-Queue

Eine *Message-Oriented Middleware (MOM)* ist ein System zur Inter Process Communication / Inter-Prozess-Kommunikation (IPC) [HW03] [Ora10]. Dabei dient die *MOM* sowohl als Übertragungskanal als auch als Nachrichten-Puffer.

⁷AWS Lambda - <https://aws.amazon.com/de/lambda>

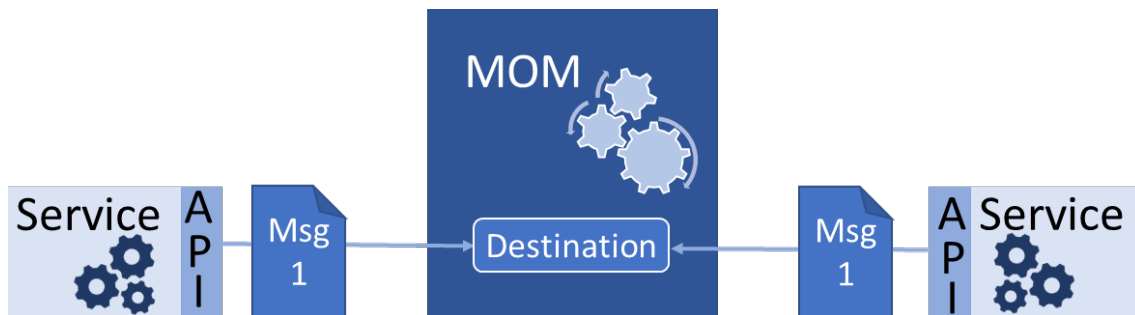


Abbildung 2.3.: Schematische Darstellung einer Message-Oriented Middleware
In Anlehnung an: [Ora10]

Bei verteilten Systemen wie z. B. einer Anwendung auf Basis der *Microservice*-Architektur müssen die einzelnen Komponenten untereinander kommunizieren. Dabei kann es vorkommen, dass einzelne Komponenten abstürzen oder aus anderen Gründen eine eingehende Nachricht nicht entgegennehmen und verarbeiten können. Weiterhin müssten die einzelnen *Microservices* exakt wissen, mit welcher Komponente sie kommunizieren müssen, um ein bestimmtes Ergebnis zu erhalten.

Eine *MOM* dient hier als allgemeiner Kanal zum Informationsaustausch, über den jede Komponente Daten an eine andere übertragen kann. Dabei sorgt die *MOM* auch für die Übertragung der Nachrichten zum jeweiligen Empfänger, wodurch die *Microservices* nicht mehr wissen müssen, wo ihr Kommunikationspartner sich befindet. Eine *MOM* nutzt hierbei verschiedene Übermittlungsmethoden, wobei die Wichtigsten im Folgenden kurz vorgestellt werden:

- **Message Passing:**
Beim *Message Passing* werden die Nachrichten direkt vom Sender zum Empfänger transportiert. Die *MOM* dient dabei nur als Übermittlungskanal.
- **Message Queuing:**
Beim *Message Queuing* werden die Nachrichten in eine Nachrichten-Warteschlange eingereiht und indirekt übertragen. Somit können die Empfangs-Komponenten die Nachrichten selbstständig abrufen und verarbeiten, sofern die Ressourcen dafür zur Verfügung stehen.
- **Publish & Subscribe:**
Der Ansatz des *Publish & Subscribe* sieht vor, dass die Nachrichten vom Sender (*Publisher*) über einen Multicast-Kanal bereitgestellt werden. Alle Abonnenten des Kanals (*Subscriber*) können die bereitgestellten Informationen abrufen und verarbeiten.

Bei der Übermittlung der Daten über eine *MOM* oder *Message Queue* gibt es verschiedene Möglichkeiten die Nachrichten an den Empfänger zu übermitteln. Je nach Anforderung stehen dabei grundsätzlich die folgenden drei Methoden zur Verfügung:

- **at-most-once delivery:**
Bei der *at-most-once delivery* werden die Datenpakete maximal einmal übertragen. Bei der Zustellung der Datenpakete wird keine Bestätigung des Empfängers abgewartet, sondern das Paket wird direkt nach dem Versenden gelöscht. Diese Methode bietet eine sehr hohe Performanz, wobei es vereinzelt zum Verlust von Nachrichten kommen kann.

- **at-least-once delivery:**

Bei der *at-least-once delivery* werden die Datenpakete mindestens einmal übertragen. Hierbei wird auf eine Rückantwort des Empfängers gewartet, die den Empfang der Daten quittiert. Sollte diese Antwort ausbleiben, wird das Datenpaket so lange neu gesendet, bis eine Antwort des Empfängers den Erhalt quittiert. Hierdurch kann es vereinzelt dazu kommen, dass eine Nachricht den Empfänger mehrfach erreicht.

- **exactly-once delivery:**

Ähnlich wie bei der *at-least-once delivery* wird auch bei der *exactly-once delivery* auf eine Rückantwort des Empfängers gewartet, bevor die Nachricht verworfen wird. Dabei führen aber sowohl die MOM als auch der Empfänger beim Auftreten eines Fehlers einen Rollback durch. Das heißt, die komplette Empfangs-Logik findet transaktional statt und wird somit beim Auftreten eines Fehlers komplett zurückgesetzt. Dies macht die *exactly-once delivery* vergleichsweise teuer.

2.2.6. OpenTOSCA

OpenTOSCA ist eine *Open Source* Implementierung einer *Cloud-Orchestrierungs-Software* nach dem *TOSCA*-Standard, woraus sich auch der Name herleitet [BEK+16] [Kic19]. Die Software besteht, wie Abbildung 2.4 zeigt, aus drei Modulen und arbeitet auf Basis eines *deklarativen Deployment-Modells*.



Abbildung 2.4.: Die verschiedenen Module von OpenTOSCA

In Anlehnung an: OpenTOSCA-Darstellung, <https://opentosca.org>

- **Winery:**

Winery ist das Modellierungs-Werkzeug von *OpenTOSCA*. Über eine Web-Plattform werden die einzelnen Komponenten von *Cloud-Anwendungen* modelliert. Die einzelnen Komponenten werden hierbei als *Nodes* angelegt, welche über *Relationship*-Definitionen miteinander in Beziehung gestellt werden. Die Topologie wird dabei als Graph visualisiert, wobei die Komponenten als Knoten und deren Beziehungen als Kanten dargestellt werden.

OpenTOSCA ist aber nicht nur für das Deployment sondern auch für das Management der *Cloud-Anwendungen* zuständig. Daher lassen sich für jede Komponente zusätzliche Parameter wie beispielsweise die benötigten Ressourcen angeben. Als Ergebnis erstellt *Winery* ein *Deployment Package* (vgl. Abbildung 3.4), welches über die *TOSCA Runtime* geladen und in der *VinoThek* bereitgestellt wird.

- **VinoThek:** Die *VinoThek* ist ein Self-Service Web-Portal, in welchem alle konfigurierten und in *OpenTOSCA* verfügbaren *Cloud-Anwendungen* angezeigt werden. Die *VinoThek* ermöglicht es, neue Instanzen der *Cloud-Anwendungen* zu starten und laufende Instanzen zu beenden und stellt dafür eine grafische Oberfläche bereit.
- **TOSCA Runtime:**
Die *TOSCA Runtime* ist die *Engine* von *OpenTOSCA*. Sie ist dafür zuständig, die *Cloud-Anwendungen* unter Zuhilfenahme des *Deployment Packages* zu starten und zu beenden. Darüber hinaus werden von ihr die laufenden Anwendungen überwacht und bei Bedarf die jeweiligen Ressourcen *skaliert*. Sie übernimmt damit alle Aufgaben der *Runtime und Management Phase*, die in Abbildung 3.4 dargestellt ist.

2.2.7. Ansible

*Ansible*⁸ ist eine IT-Automatisierungs-Engine [Hum19] [cA18]. Das bedeutet, *Ansible* ist in der Lage, verschiedenste administrative Tätigkeiten zu automatisieren. Neben der Orchestrierung und dem automatisierten Deployment von *Cloud-Anwendungen* beinhaltet dies beispielsweise auch die Automatisierung der kompletten *Continuous Delivery*-Pipeline. Selbst die Konfiguration und Administration von Systemen lässt sich durch *Ansible* automatisieren.

Intern arbeitet *Ansible* auf der Basis von Python und benötigt für die Automatisierung lediglich Zugriff auf die Zielmaschinen über OpenSSH. Die Besonderheit bei *Ansible* ist, dass auf den Zielsystemen keine Agent-Software benötigt wird. Das heißt, auf den Zielsystemen muss keine zusätzliche Software installiert werden, welche mit *Ansible* kommuniziert und die Befehle auf der Zielmaschine ausführt.

Die Software lässt sich vollständig durch YAML-Dateien konfigurieren. Für die Automatisierung von Aufgaben werden sogenannte *Playbooks* festgehalten. Die Möglichkeiten der Automatisierung reicht dabei von einfachen Konfigurationen von Systemen bis hin zu der kompletten Einrichtung inklusive der Installation von vorab definierter Software.

Um einen Prozess zu automatisieren, muss entsprechend ein *Playbook* angelegt werden, welches mit einem Benutzerhandbuch vergleichbar ist. Darin werden die einzelnen Schritte festgehalten, welche von der Automatisierungs-Software durchgeführt werden sollen. Neben den *Playbooks* existieren auch noch die Ressourcen *Inventar* und *Modul*. Das *Inventar* steht hierbei für einen Knoten, auf dem *Ansible* operieren kann. Entsprechend werden darin die Daten wie die IP-Adresse oder der vollständige Rechnername angegeben. Das *Modul* hingegen stellt ein Unterprogramm dar und wird daher für häufig wiederkehrende Aufgaben genutzt. Es stellt damit das Äquivalent einer Funktion einer Programmiersprache dar.

⁸Ansible - <https://www.ansible.com>

2.2.8. Chef

Ähnlich wie *Ansible* ist auch *Chef*⁹ eine Automatisierungs-Software [HA18]. Bei Chef handelt es sich zudem um eine *Open-Source-Software*, welche unter der *Apache License 2.0*¹⁰ veröffentlicht wurde. Das bedeutet, dass jeder sich den Quellcode der Software ansehen und diesen nach Belieben anpassen kann.

Chef besteht dabei aus den vier Haupt-Komponenten *Chef Automate*, *Chef Infra*, *Chef InSpec* und *Chef Habitat*. Damit stehen neben der Automatisierung von Aufgaben beispielsweise Module für die Überwachung und die Berichterstattung sowie ein Dashboard mit der Übersicht bereit.

Das Modul für die Automatisierung wird als *Chef Habitat* bezeichnet. Die Terminologie der Software leitet sich vom Namen ab, der übersetzt für „Chefkoch“ steht. Dabei wird ein automatisierter Vorgang als *Rezept* bezeichnet und beinhaltet alle nötigen Informationen, um die jeweilige Aufgabe auszuführen. Darunter fallen beispielsweise die zu installierenden Linux-Pakete oder die auszuführenden Dienste. Die Automatisierungen werden im *Kochbuch* gesammelt und stehen dort, visualisiert durch die *Web-Konsole*, zur Ausführung bereit.

Intern arbeitet Chef mit der Programmiersprache Ruby und nutzt das Pattern der Client-Server-Architektur [DA18]. Darüber hinaus wird eine eigenständige Version angeboten, welche nur auf dem jeweiligen System operiert. Die Rezepte selbst stellen grundsätzlich Ruby-Anwendungen dar und definieren alle nötigen Schritte für die Konfiguration eines Systems.

2.2.9. Amazon CloudFormation (Amazon CF)

Mit der *Amazon CF*¹¹ ist es möglich die Infrastruktur der AWS über textuelle Konfigurationen zu automatisieren [DA17] [LK19].

Im Gegensatz zu *Ansible*, *Chef* und *OpenTOSCA* unterstützt die Amazon CF nur das automatisierte Bereitstellen von Infrastrukturlandschaften innerhalb der AWS. Die entsprechenden Automatisierungsschritte werden hier durch sogenannte *Templates* ermöglicht.

Die *Templates* können wahlweise im JSON- oder im YAML-Format formuliert werden. Allerdings bietet die Amazon CF auch eine Weboberfläche an, in der sich die einzelnen Komponenten mittels Drag & Drop orchestrieren lassen. Auch bei der Nutzung der Weboberfläche wird im Hintergrund letztlich ein textbasierte Konfiguration erzeugt, auf deren Grundlage weiter gearbeitet werden kann. Daher ist es auch möglich, durch einen integrierten JSON- und YAML-Editor Details des *Templates* direkt im Designer anzupassen.

Wird ein *Template* der AWS übergeben, erstellt diese den zugehörigen Komponenten-*Stack*. Dabei werden automatisch die Abhängigkeiten und Datenflüsse berücksichtigt. Nach der Erstellung werden die Komponenten des *Stacks* auch in der *AWS Konsole* angezeigt und können dort wie gewohnt angepasst werden.

⁹Chef - <https://www.chef.io>

¹⁰Apache License 2.0 - <https://www.apache.org/licenses/LICENSE-2.0>

¹¹Amazon CF - <https://aws.amazon.com/de/cloudformation>

2.3. Grundlegende Konzepte

Dieser Abschnitt erläutert Techniken und Konzepte, welche in besonderem Zusammenhang mit der Cloud stehen und häufig in Kombination mit Cloud-Anwendungen auftreten.

2.3.1. Continuous Integration / Delivery / Deployment

Die Begriffe *Continuous Integration*, *Continuous Delivery* und *Continuous Deployment* hängen eng zusammen. Sie bezeichnen die verschiedenen Automatisierungs-Grade, die bei der Entwicklung und Veröffentlichung eine Rolle spielen.

Vor allem in Kombination von *Cloud-Anwendungen* mit der *Microservice-Architektur* stellt sich das *Continuous Deployment* als eine Herausforderung dar. Das *DevOps*-Konzept entwickelt sich aber mehr und mehr zum Branchen-Standard, weswegen der Einsatz von *Cloud-Orchestrierungs-Software* für *Cloud-Anwendungen* unerlässlich ist. Daher ist es wichtig, dass eine *Cloud-Orchestrierungs-Software* sich gut in eine *Continuous Deployment*-Pipeline integrieren lässt.

Die Unterschiede der „Continuous“-Begriffe werden durch Abbildung 2.5 veranschaulicht und im Folgenden kurz beschrieben.

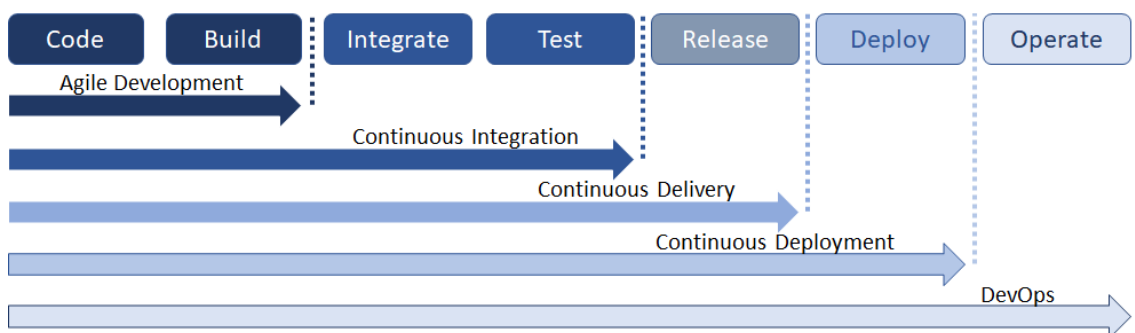


Abbildung 2.5.: Continuous Integration / Delivery / Deployment & DevOps
In Anlehnung an: [Cey18]

- **Continuous Integration (CI):** Noch vor einigen Jahren wurden große Updates von den Entwicklern (*Devs*) veröffentlicht, welche von den Betreibern der Software (*Ops*) installiert werden mussten. Die Updates wurden meist im Abstand von einem oder mehreren Monaten veröffentlicht, was dazu führte, dass die Updates umfangreich waren und deren Integration entsprechend mühsam [KBM14]. Um diesem Problem zu begegnen wurden Systeme zum automatisierten Erstellen und Testen von Software entwickelt. Dieser Grad der Automatisierung wird als *Continuous Integration* bezeichnet [HF] [Cey18]. Dabei wird jede Änderung, die in das Versionierungssystem (z. B. *Git*¹²) eingepflegt wird, von dem CI-System übernommen. Die Software wird erstellt (*build*) und anschließend durch automatisierte Tests geprüft.

¹²Git - <https://git-scm.com>

- **Continuous Delivery:** Die *Continuous Delivery* baut auf dem Konzept der *CI* auf und erweitert diese durch das automatische Veröffentlichen von Software-Versionen. Hierdurch lassen sich Software-Versionen schnell und einfach veröffentlichen, ohne dabei die Qualität negativ zu beeinflussen. Daraus folgt, dass sich die Release-Zyklen deutlich verkürzen lassen und somit kleine Bugfixes und Erweiterungen schnell veröffentlicht werden können.
- **Continuous Deployment:** Beim *Continuous Deployment* wird nicht nur die neue Software-Version veröffentlicht, sondern auch automatisiert in das Produktiv-System übernommen. Hierdurch wird das o. g. Problem der (teils aufwändigen) Integration [KBM14] der Software gelöst. Entsprechend lassen sich, verglichen mit der *Continuous Delivery*, Änderungen deutlich schneller beim Kunden anwenden.
- **DevOps:** Der Begriff *DevOps* setzt sich aus den Worten „Development“ und „Operations“ zusammen und soll damit die Brücke zwischen den beiden Bereichen bilden [Lou12]. Ziel ist eine bessere Zusammenarbeit zwischen den Entwicklern der Software und dem Team, welches den Betrieb der Software sicherstellt. Ein wichtiger Schritt hierbei ist die Umsetzung des *Continuous Deployments*, welches die Updates der Entwickler direkt in die Produktiv-Umgebung einspielt. Daneben gehört auch die automatisierte Überwachung des Systems zu einer guten Umsetzung des *DevOps*-Konzeptes, da sowohl Entwickler als auch Operator entlastet werden sollen, um sich auf die wesentlichen Aufgaben konzentrieren zu können.

2.3.2. Microservice

Der Begriff *Microservice* kommt aus dem Ansatz der *Microservice*-Architektur, welche sich in den letzten Jahren entwickelt hat. Hierbei bestehen Anwendungen aus einer Vielzahl von untereinander unabhängig deploybaren Diensten [LF14] [Völ18].

Ein großer Vorteil von *Microservices* ist, dass sie voneinander unabhängig laufen und über den Austausch von Nachrichten miteinander kommunizieren. Häufig findet dieser Austausch über eine gemeinsame *MOM* statt. Durch dieses Modell lassen sich die einzelnen Module völlig unabhängig voneinander, z. B. von mehreren Teams, entwickeln. Darüber hinaus können Anwendungen leicht abgeändert und erweitert werden, da sich die kleinen Module jederzeit anders kombinieren lassen. Die Kombination der einzelnen Module wird auch als „Komposition“ bezeichnet.

Die Schwierigkeit dieses Architektur-Modells liegt in der Kommunikation der Module untereinander: Da die Module voneinander unabhängig laufen, lässt sich eine Funktionsstörung eines Moduls nur schwer detektieren. Entsprechend müssen die einzelnen Module so entwickelt werden, dass sie trotz des Ausfalls eines anderen Moduls weiter arbeiten.

2.3.3. virtuelle Maschine (VM)

Grundlegend arbeiten viele Cloud-Systeme mit virtuellen Maschinen (VMs). Daher beziehen sich viele verwandte Arbeiten (vgl. Kapitel 3) auf VMs.

Eine *VM* bezeichnet grundsätzlich eine Maschine, die über keine reale Hardware verfügt. Entsprechend wird stets eine virtualisierte Hardware benötigt, welche durch einen sogenannten *Hypervisor* / *VM-Monitor* (*VMM*) bereitgestellt wird [GWL19a] [ITW07] [FGLI15] [Mat17].

2. Grundlagen

Der VMM wiederum muss auf irgendeiner physischen Maschine (PM) laufen, welche die physische Hardware bereitstellt. Dabei können auf einer PM mehrere VMs laufen. Anders herum können aber auch mehrere PMs die Ressourcen für eine einzelne VM stellen, wenn deren Ressourcenbedarf die verfügbaren Ressourcen einer einzelnen PM übersteigt (vgl. Abbildung 2.6).

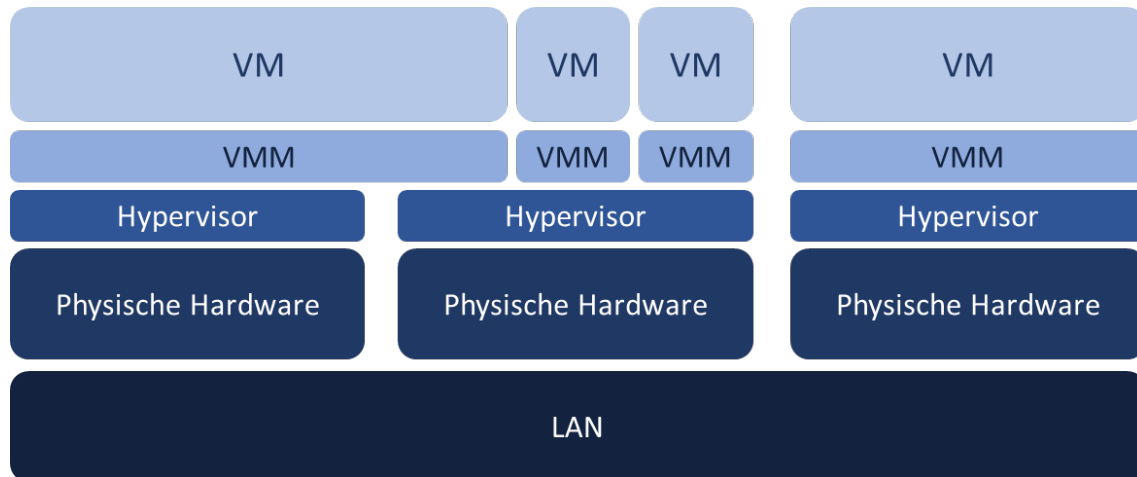


Abbildung 2.6.: Schematische Darstellung: Ein Verbund von PMs betreibt mehrere VMs
In Anlehnung an: [FGLI15]

Eine *VM* kann entsprechend nicht nur einen virtualisierten Computer sondern auch die Laufzeitumgebung (z. B. die Java Runtime *JVM*) für ein Programm darstellen.

In den meisten Fällen wird aber von einem virtualisierten Computer gesprochen, der auf einer virtualisierten Hardware läuft. Dabei teilen sich die VMs und der VMM die Ressourcen des physischen Computers [FGLI15]. Durch diese Technik ist es möglich, verschiedene, voneinander abgeschottete Systeme auf ein und derselben physischen Maschine auszuführen und so die Ressourcen unter den Nutzern aufzuteilen.

Besonders *Cloud*-Anbieter setzen auf die Möglichkeit der Virtualisierung, indem das komplette Rechenzentrum über den VMM zu einer großen VM zusammengefasst wird. Auf dieser wiederum werden kleine VMs ausgeführt, welche den Kunden die jeweiligen Dienste zur Verfügung stellen.

3. Verwandte Arbeiten

Im Folgenden werden die Arbeiten vorgestellt, die sich mit ähnlichen Aufgaben befassen wie der im Rahmen dieser Arbeit entwickelte Prototyp. Hauptsächlich finden sich dabei verschiedene Ansätze für die Replikation und Speicherung von *virtuellen Maschinen (VM)*.

Die Arbeit von Harzenetter et al. (vgl. Abschnitt 3.5) bildet hierbei die Ausnahme, da sie ein Konzept vorstellt, durch welches sich das „Einfrieren und Auftauen“ von *Cloud*-Anwendungen auf Basis der *TOSCA*-Plattform umsetzen lässt.

3.1. A Survey of Migration Mechanisms of Virtual Machines

In der Arbeit „A Survey of Migration Mechanisms of Virtual Machines“ untersuchen V. Medina und J. M. García verschiedene Migrations-Mechanismen von *VMs* [MG14]. Sie arbeiten dabei auf der Grundlage von diversen Arbeiten die Konzepte heraus und kategorisieren diese. Dabei gehen sie auch auf die Funktionsweise existierender Produkte wie *VirtualBox*¹ ein.

Ihre Untersuchung ergibt, dass sich die untersuchten Herangehensweisen in vier Kategorien einsortieren lassen:

- **Live Migration:**

Die *Live Migration* sieht vor, dass ganze *VMs* migriert werden. Hierzu müssen auch *zustand-behaftete* Komponenten wie beispielsweise Random Access Memory / Arbeitsspeicher (RAM) und Central Processing Unit / Prozessor (CPU) migriert werden, wofür unterschiedliche Techniken zur Verfügung stehen. Daneben sind auch noch der Speicher der *VM* sowie deren Netzwerk-Konnektivität zu migrieren, was eine nicht-triviale Aufgabe darstellt.

- **Precopy:** Das *Precopy*-Verfahren sieht vor, dass der RAM der Quell-*VM* über ein iteratives Verfahren migriert wird, während die Quell-*VM* weiterhin ausgeführt wird. Dabei werden in jedem iterativen Schritt nur die Veränderungen im RAM seit der letzten Iteration migriert.

Anschließend wird die Quell-*VM* pausiert, der Zustand der CPU sowie die restlichen verbliebenen Änderungen des RAM migriert und die Ziel-*VM* gestartet. Falls die Ziel-*VM* versucht, auf noch nicht übertragene RAM-Segmente zuzugreifen, werden diese noch von der Quell-*VM* übertragen.

¹VirtualBox - <https://www.virtualbox.org/>

- **Postcopy:** Beim *Postcopy*-Verfahren wird der RAM erst nach der Migration des CPU-Zustands migriert. Dazu wird die *VM* zunächst angehalten und der Zustand der CPU-Register migriert. Im Anschluss wird die Ziel-*VM* ohne die Migration des RAM ausgeführt.

Sobald die *VM* versucht auf noch nicht migrierte Segmente des RAM zuzugreifen, wird die Ziel-*VM* kurz angehalten und die entsprechenden RAM-Segmente werden migriert. Anschließend wird die Ziel-*VM* weiter ausgeführt.

Insbesondere die *WAN Live Migration* gewinnt zunehmend an Interesse, da hierdurch auch Live-Migrationen über WAN-Netzwerke ermöglicht werden. Allerdings muss dabei der komplette Zustand der *VM* einschließlich offener Netzwerkverbindungen migriert werden. Der Ansatz von *CloudNet* stellt eine Möglichkeit der *WAN Live Migration* dar, wobei auf das VPN-Konzept zurückgegriffen wird [WRSM11]. Verteilte und über WAN verbundene *Cloud*-Rechenzentren werden miteinander über ein VPN-Layer zu einem Virtuellen Cloud Pool (VCP) verbunden. Hierdurch lassen sich die offenen Netzwerkverbindungen über das VPN zum neuen Host der *VM* weiterleiten, ohne dass diese neu aufgebaut werden müssen.

- **Suspend / Resume Migration:**

Die Migration durch die *Suspend- / Resume*-Mechanik basiert darauf, dass die *VMs* während der Migration vollständig angehalten und nach der Migration wieder ausgeführt werden. Dieses Konzept wird üblicherweise für die Migration über WAN genutzt, bei der die Netzwerkverbindungen unterbrochen und später neu aufgebaut werden.

Für die WAN-Migration ist es äußerst wichtig, den kompletten persistenten Zustand der *VM* zu migrieren. Um die Übertragung des *VM*-Speichers zu optimieren, werden hierzu *Deltas* (Δ) genutzt, welche die Änderungen des Festplattenzustands darstellen. Ein *Delta* bezeichnet dabei eine Schreib-Operation auf der Festplatte und bildet sich aus dem Tripel: Geschriebene Daten, betroffene Segmente der Festplatte und Größe der geschriebenen Daten. Diese Änderungen bilden nur einen Bruchteil der kompletten Festplatte ab und lassen sich damit deutlich schneller über LAN oder WAN versenden.

- **Load Balancing / Green Computing:**

Niedrig ausgelastete Server sind vergleichsweise ineffizient hinsichtlich der nötigen Energie für den Betrieb und die Kühlung. Daher ist es erstrebenswert, alle Server möglichst gleich auszulasten und überflüssige Ressourcen vollständig abzuschalten (*Green Computing*). Hierfür bedarf es Algorithmen, welche auf den genannten Migrationstechnologien aufsetzen und durch optimierte Migrationen die Ressourcen verwalten und bei Bedarf Server herunterfahren oder wieder starten.

Das *Load Balancing* stellt damit keine eigene Migrations-Technik, aber einen erweiterten Anwendungsfall dar, bei dem es darauf ankommt, die verfügbaren Ressourcen zu verwalten und möglichst optimal auszulasten. Aufgrund der zusätzlichen Aufgabe des Ressourcen-Managements kategorisieren Medina et al. die entsprechenden Beispiele separat.

3.2. Algorithms for automated live migration of virtual machines

In der Arbeit *Algorithms for automated live migration of virtual machines* befassen sich Forsman et al. mit der Frage, wie sich die Last von Multi-VM-Systemen besser unter den einzelnen *physischen Maschinen (PM)* aufteilen lässt [FGLI15]. Um die Auslastung zu verbessern, sollen die einzelnen VMs so angelegt werden, dass möglichst alle Systeme gleichermaßen ausgelastet sind. Dabei kann es vorkommen, dass sich der Ressourcenbedarf der VMs ändert und diese für eine optimale Auslastung migriert werden müssen. Die Migration der VMs soll dabei vollautomatisiert und mit möglichst geringer Downtime ablaufen.

Forsman et al. stellen in ihrer Arbeit zwei verschiedene Strategien zur Ermittlung der Migrationen vor. Dabei nutzt das System zur automatischen Migration sowohl die *Push-* als auch die *Pull-*Strategie gleichermaßen, um eine möglichst äquivalente Auslastung des Gesamtsystems zu erzielen.

- **Push-Strategie:**

Die Push-Strategie ist so definiert, dass ausgelastete PMs ihre VMs den anderen verfügbaren PMs zur Übernahme anbieten.

Hierbei veranstaltet der überlastete *Host (PM)* eine Auktion aller VMs. Alle anderen PMs können Gebote für eine oder mehrere der angebotenen VMs abgeben. Sobald alle Gebote eingegangen sind, entscheidet der Auktionator, welches Gebot die beste Auslastung des Gesamt-Systems verspricht und migriert die entsprechende VM zu der PM, von der das Gebot stammt. Obwohl bei einer Auktion mehrere VMs angeboten werden, wird nur eine einzige VM migriert.

- **Pull-Strategie:**

Während bei der *Push-Strategie* ein Host seine VMs versteigert, bietet bei der *Pull-Strategie* ein nicht ausgelasteter Host seine Ressourcen im Netzwerk an. Auch hierbei wird eine Auktion ausgeführt um zu ermitteln, welche VM-Migration den größten Effekt bezüglich der einheitlichen Auslastung des Gesamtsystems hat. Dabei ist es den Bietern erlaubt, ein Gebot für jede VM abzugeben, die zu den Ressourcen-Vorgaben des Auktionators passt. Nach Abgabe der Gebote aller anderen PMs entscheidet der Auktionator, welcher Host welche der angebotenen VMs migrieren darf.

Für die Migration der einzelnen VMs werden ebenfalls verschiedene Herangehensweisen vorgestellt:

- **Cold Migration:**

Bei der *Cold Migration* wird zunächst die betreffende VM heruntergefahren. Im Anschluss wird die VM zum anderen Host migriert und dort wieder hochgefahren. Hierdurch ist die VM für einen vergleichsweise langen Zeitraum nicht nutzbar.

- **Hot Migration:**

Bei der *Hot Migration* wird die VM nicht heruntergefahren, sondern lediglich angehalten, damit sich der Zustand nicht verändert. Anschließend wird die VM auf den neuen Host migriert und dort weiter ausgeführt. Daraus ergibt sich der Vorteil, dass der Zustand der laufenden Programme beibehalten wird und somit die Migration schneller durchgeführt werden kann.

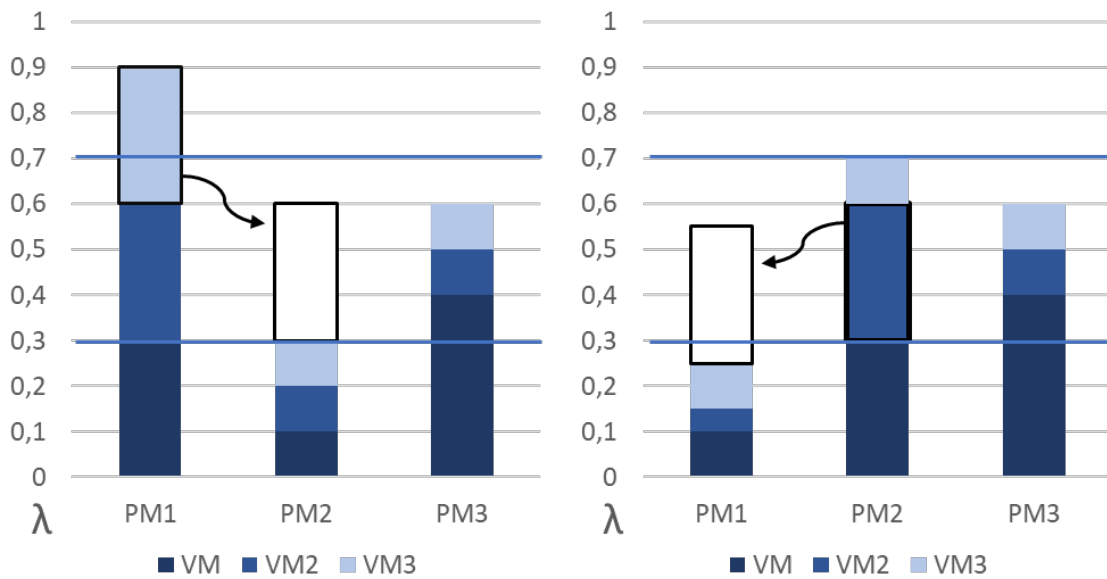


Abbildung 3.1.: Anwendung der Push- / Pull-Strategie
In Anlehnung an: [FGLI15]

- **Live Migration:**

Die dritte genannte Migrations-Form ist die *Live Migration*, bei der die *VM* im ausführenden Zustand migriert wird. Da sich der Zustand der *VM* hierbei jedoch während der Migration ändert, müssen diese Änderungen zusätzlich migriert werden.

Ziel der Arbeit ist eine automatisierte Live-Migration, welche die folgenden Kriterien erfüllt:

1. **Hotspots detektieren:**

Als *Hotspot* wird ein überlasteter Host bezeichnet. Das automatische Migrations-System muss die überlasteten *PMs* identifizieren und die Last umverteilen.

2. **Hotspots minimieren:**

Bei der Migration der *VMs* dürfen dadurch keine neuen *Hotspots* entstehen.

3. **Coldspots vermeiden:**

Die *Coldspots* bezeichnen nicht ausgelastete *PMs*. Da das Ziel ein möglichst gleich ausgelastetes Gesamt-System ist, sollen *Coldspots* vermieden werden.

4. **Migrations-Kosten minimieren:**

Trotz verschiedener Migrations-Techniken bleibt die Migration eine teure Operation. Daher sollen die Migrations-Kosten möglichst minimal gehalten werden.

Um den Ansatz auf seine grundsätzliche Tauglichkeit prüfen zu können, setzen die Autoren voraus, dass eine *VM* die kleinste zu migrierende Einheit darstellt. Außerdem wird immer nur eine *VM* pro Algorithmus-Iteration migriert und nur die Prozessorlast wird als Metrik für die Hotspot-Erkennung genutzt.

Als PoC nutzen Forsman et al. eine simulierte Testumgebung, mit welcher das genannte Verfahren getestet wird. Dabei werden *Push*- und *Pull*-Strategie simultan verwendet um auf die verschiedenen Auslastungen der einzelnen Hosts reagieren zu können. Ihre Evaluation zeigt, dass das System trotz des Hinzufügens und Entfernens „einer großen Anzahl von VMs“, innerhalb von 4 bis 15 Minuten wieder ausbalanciert ist.

3.3. Stateful Process Migration for Edge Computing Applications

Die Arbeit „Stateful Process Migration for Edge Computing Applications“ von M. Horii, Y. Kojima und K. Fukuda [HKF18] beschäftigt sich mit der Migration von Anwendungen, welche extrem niedrige Latenzen erfordern, wie zum Beispiel Navigationssysteme oder Sensoren.

Die Arbeit spricht in diesem Zusammenhang davon, dass die Anwendungen, welche mit dem so genannten User Equipment / Benutzer-Equipment (UE) kommunizieren, möglichst geringe Latenzen dort hin haben sollen. Das Konzept der Cloud sieht vor, dass die Anwendungen *irgendwo* in der Cloud ausgeführt werden. Vor allem bei einem virtuellen Cloud-Pool (VCP) kann dies zu dem Problem führen, dass die physischen Kommunikationswege zu lang und die Latenz damit zu hoch wird.

Als Beispiel wird von den Autoren das Konzept des *Connected Car* angeführt [HPS+15]. Dabei handelt es sich um ein Auto, welches mit anderen Autos kommuniziert, also verbunden ist. Die Autos teilen sich verschiedene Sensordaten mit, um auf dieser Grundlage das autonome Fahren zu ermöglichen. Besonders bei hohen Geschwindigkeiten ist es notwendig, dass diese Daten schnell verarbeitet werden. Da Autos oder auch Drohnen bewegliche Objekte in der realen Welt sind, ist es nötig, die zugehörigen Anwendungen möglichst nahe zu betreiben. Dabei meint *nah*, dass die Anwendungen auf einem Netzwerkknoten betrieben werden, dessen realer Standort dem realen Standort der UEs am nächsten kommt. Hierzu müssen die Anwendungen auf den entsprechenden Server migriert werden. Das Konzept, Anwendungen möglichst nahe bei den Konsumenten zu betreiben wird dabei als *Edge Computing* bezeichnet.

Die Lösung für das o. g. Problem sehen Horii et al. in der Migration der *zustand-behafteten* Prozesse, wohingegen die *zustand-freien* Prozesse nicht migriert werden sollen. Bei der Migration von Prozessen werden, im Vergleich zu *VMs* oder zu Containern, deutlich geringere Datenmengen übertragen (vgl. Abbildung 3.2). Der Nachteil dabei ist, dass Prozesse sehr eng an die zugrundeliegende Architektur gebunden sind und sich kaum auf andere Systeme portieren lassen. Dadurch müssen sowohl Quell- als auch Ziel-Server die exakt gleiche Architektur aufweisen, damit der Prozess überhaupt migriert werden kann.

Die Unterscheidung zwischen *zustand-behafteten* Prozessen und *zustand-freien* Prozessen sorgt ebenfalls für weitere Einsparungen bei der Migration, wodurch diese deutlich beschleunigt werden kann. Die beiden Prozess-Typen werden von den Autoren so erklärt:

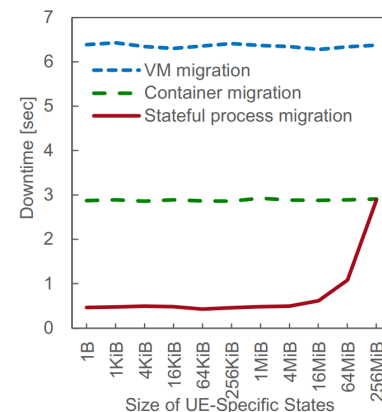


Abbildung 3.2.:
Vergleich der Downtime unterschiedlicher Migrationen.
Entnommen aus: [HKF18]

- **zustand-behafteter Prozess:**

Ein *zustand-behafteter Prozess* ist so definiert, dass dieser UE-spezifische Daten, wie beispielsweise die *ID* des Autos, beinhaltet. Entsprechend ist dieser Prozess nur für die Nutzer wichtig, deren Daten dort verarbeitet werden.

- **zustand-freier Prozess:**

Als *zustand-freier Prozess* wird ein Dienst bezeichnet, der selbst keine UE-spezifische Daten enthält, sondern seine Dienste anderen Prozessen zur Verfügung stellt. *Zustand-freie Prozesse* empfangen und verarbeiten Daten von anderen Prozessen und senden die Resultate an diese zurück. So kann ein *zustand-freier Prozess* beispielsweise aus den übermittelten Bewegungsdaten eine Vorhersage des weiteren Bewegungsverlaufs errechnen und diese Vorhersage zurücksenden.

Wichtig hierbei ist, dass jeder Prozess als Eingabe alle Informationen erhält, die zum Verarbeiten der Information nötig sind. Andernfalls kann ein *zustand-freier Prozess* aufgrund fehlender Zustandsinformationen die Eingabe nicht verarbeiten.

Die Migration wird in drei Phasen durchgeführt:

1. **Vorbereitung:**

Da der *zustand-behaftete Prozess* von den *zustand-freien Prozessen* und dem Operating System / Betriebssystem (OS) abhängt, müssen diese vor der Migration laufen. Daher werden in der *Vorbereitungs-Phase* auf der Ziel-VM das OS sowie die *zustand-freien Prozesse* gestartet.

2. **Migration:**

Zunächst gibt die Quell-VM der Ziel-VM die Anweisung zu prüfen, ob ausreichend Ressourcen zur Verfügung stehen. Falls diese Bedingung erfüllt wird, startet die eigentliche Migration des *zustand-behafteten Prozesses*.

3. **Konvertierung der Kommunikationskanäle:**

Die Kommunikation zwischen Prozessen wird als Inter Process Communication / Inter-Prozess-Kommunikation (IPC) bezeichnet. Da *zustand-behaftete Prozesse* mit anderen Prozessen kommunizieren, müssen hierfür vom OS IPC-Kanäle erstellt werden. Diese IPC-Kanäle werden vom OS erstellt und mit einer ID versehen, um die Kanäle eindeutig identifizieren zu können.

Werden die *zustand-behafteten Prozesse* migriert, referenzieren diese noch auf IPC-Kanäle des Quell-OS. Um die IPC-Kanäle ebenfalls migrieren zu können, müssen auf dem Ziel-OS zunächst IPC-Kanäle angelegt werden. Dabei werden aber vom Ziel-OS neue IDs für die IPC-Kanäle vergeben, wodurch die alten IPC-Kanal-IDs ungültig werden.

Entsprechend müssen bei der Migration von Prozessen auch die IPC-Kanäle konvertiert und auf der Ziel-VM verfügbar gemacht werden, sodass die migrierten Prozesse nur mit den lokalen Prozessen kommunizieren können. Dies geschieht über eine dynamische Konvertierung der IPC-Kanal-IDs, welche vom **Kanal-Konverter** durchgeführt wird.

Der *Kanal-Konverter* bildet eine Schicht zwischen dem OS und den Prozessen. Dadurch ist dieser in der Lage, die veralteten Kanal-IDs zu korrigieren und so die neuen Kanäle verfügbar machen. Wird also von einem Prozess ein IPC-Kanal angefragt, dessen ID sich bei der Migration geändert hat, landet die Anfrage beim *Kanal-Konverter*. Dieser führt ein

Kanal-Verzeichnis und kann anhand dessen die neue IPC-Kanal-ID herausfinden und die Anfrage mit der richtigen Kanal-ID an das OS weiterleiten. Hierdurch wird dem anfragenden Prozess vom OS der korrekte IPC-Kanal zur Verfügung gestellt.

Für den Fall, dass ein *zustand-behafteter Prozess* zum Zeitpunkt der Migration noch offene Anfragen hatte, werden die Antworten auf diese Anfragen zur Ziel-VM umgeleitet. Da pro Anfrage nur eine Antwort zurück gegeben wird, muss pro offener Anfrage auch nur diese eine Antwort umgeleitet werden. Alle weiteren Anfragen werden von den *zustand-behafteten Prozessen* auf dem neuen System ausgeführt, wodurch keine weiteren Weiterleitungen nötig werden.

3.4. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing

In der Arbeit „SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing“ stellen Lagar-Cavilla et al. ein System abgeleiteter VMs vor.

Das Konzept der *virtuellen Maschinen (VM)* wird von vielen *Cloud*-Dienstleistern als Grundlage genutzt, um ihre verschiedenen Dienste anzubieten. Grund dafür ist, dass das *VM-Konzept* Vorteile wie zum Beispiel Sicherheit, einfaches Management, Isolation der Umgebungen oder Flexibilität bietet. Darüber hinaus lassen sich die verfügbaren Ressourcen flexibel auf die laufenden VMs verteilen und es wird sichergestellt, dass keine davon mehr Ressourcen nutzt, als ihr zugeteilt werden.

Einer großer Vorteil des *Cloud-Computings* ist, dass sich die Zahl der genutzten VMs ganz den Bedürfnissen angepasst werden kann. So werden im Fall von hoher Last auf ein System viele VMs ausgeführt, um die Anfragen zu bedienen, während bei niedriger Last möglicherweise eine einzige VM ausreicht. Das Problem dabei ist, dass das Erstellen von neuen VMs eine teure Operation ist, die häufig mehrere Minuten dauert. Hinzu kommt, dass die neuen VMs von einer Basis-Installation oder einem eigenen *Template* ausgehen und somit noch nicht einmal den aktuellen Zustand des laufenden Systems kennen. Daraus folgt, dass nach dem Erstellen der VM zunächst der Zustand des Systems übertragen werden muss, was bislang programmatisch gelöst wird.

Als Lösung präsentieren Lagar-Cavilla et al. das *SnowFlock*-Verfahren, welches durch Abbildung 3.3 verdeutlicht wird.

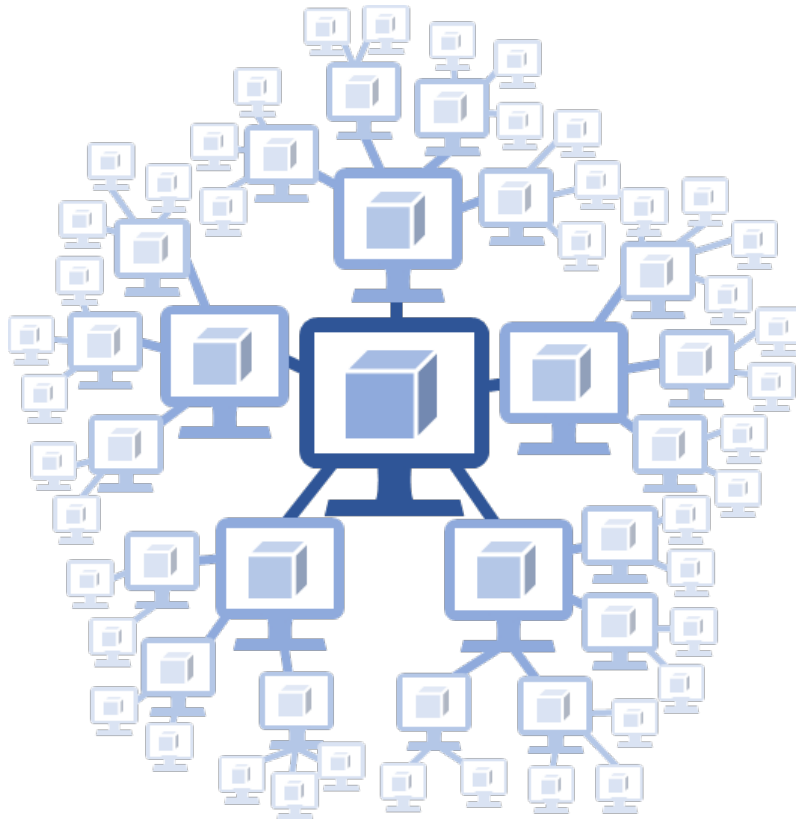


Abbildung 3.3.: Schematische Darstellung von SnowFlock, © 2019 F. Amend

Von einer Ursprungs-VM werden viele weitere VMs abgeleitet, die anschließend voneinander unabhängig auf verschiedenen Host-Systemen weiter laufen können. Durch die vielen unabhängigen VMs lassen sich die Cloud-Ressourcen optimal nutzen. Dabei reduziert *SnowFlock* den Aufwand der Erstellung dieser vielen identischen VMs erheblich, indem es sich am *UNIX process fork*² orientiert:

- Der *Fork* kann auf beliebigen PMs erstellt werden.
- Eine Quell-VM kann beliebig viele *VM-Forks* erzeugen.
- Ein *Fork* ist eine vollständige Replik aller Prozesse und Threads der Basis-VM.
- Der *Fork* ist eine unabhängige Kopie und nach der Replikation werden keine Updates mehr propagiert.

Um dies zu verwirklichen führt *SnowFlock* für jeden *Fork* die folgenden Schritte durch. Dabei werden nur die Daten migriert, die für den jeweiligen Task zwingend erforderlich sind. Lagar-Cavilla et al. haben herausgefunden, dass dies meistens nur 0.1% des Zustands der Ursprungs-VM darstellt. Darüber hinaus tendieren die abgeleiteten VMs dazu, lediglich einen Bruchteil des RAM-Abbildes der Ursprungs-VM zu nutzen.

²UNIX process fork - <http://man7.org/Linux/man-pages/man2/fork.2.html>

1. **Lazy State Replication:**

Zunächst wird die minimale Basisinstallation der VM abgeleitet. Das bedeutet, dass nur die Daten kopiert werden, welche zwingend dafür benötigt werden, die VM auszuführen. Diese werden von den Autoren als **VM Deskriptor** bezeichnet und beinhalten unter anderem das Betriebssystem und die zwingend erforderlichen Zustands-Daten wie z. B. die genutzten RAM-Segmente oder den Zustand der CPU.

2. **Vermeidungsheuristiken:**

Die *Vermeidungsheuristiken* sorgen dafür, dass möglichst nur die Daten übertragen werden, welche auch zwingend für den Betrieb des Forks erforderlich sind. Ein Teil davon ist der **Memory-On-Demand-Mechanismus**, bei dem die benötigten RAM-Segmente erst bei Bedarf über die Netzwerkverbindung angefordert und migriert werden.

3. **Multicast Distribution:**

Da die abgeleiteten VMs alle ähnliche Code-Pfade ausführen, benötigen sie ähnliche Daten. Über die Möglichkeit der *Multicast Distribution* der Daten wird die Last auf dem Netzwerk zum Zeitpunkt der Migration deutlich verringert: Anstatt die Daten jeder VM einzeln zu senden, werden diese über das Multicast-Protokoll nur einmal gesendet und allen VMs zugestellt, welche die Daten benötigen.

Die Autoren setzten bei ihrem Algorithmus auch ein Augenmerk auf die Daten-Sicherheit. Ein wichtiges Konzept hierbei ist die Isolation der VMs voneinander [Mat17], welche in diesem Fall durch separierte *virtuelle Festplatten* sichergestellt wird.

3.5. Freezing and Defrosting Cloud Applications: Automated Saving and Restoring of Running Applications

Die Arbeit *Freezing and Defrosting Cloud Applications: Automated Saving and Restoring of Running Applications* von Harzenetter et al. [HBKL19], befasst sich mit der Archivierung und Reprovisionierung *cloudbasierter* Anwendungen unter Beibehaltung des Zustandes.

Das Konzept des *Cloud Computings* erlaubt es, IT-Ressourcen bei Bedarf anzumieten und dabei nur so viel zu bezahlen, wie man auch verbraucht hat. Entsprechend lassen sich die Ressourcen auch wieder frei geben, um so unnötige Kosten zu sparen. Über die Nutzung von *Cloud-Orchestrierungs-Software*, wie beispielsweise *OpenTOSCA* lassen sich *Cloud-Anwendungen* einfach starten und auch wieder beenden.

Die Autoren zeigen, dass dieses Szenario durchaus Anwendung finden kann. Sie führen dazu das Beispiel einer Geschäftsanwendung an, die eben nur so lange benötigt wird, wie die Mitarbeiter der Firma damit auch arbeiten, also meist zwischen 06:00 Uhr und 20:00 Uhr. In der Zeit von 20:00 Uhr bis 06:00 Uhr (10 Stunden) würde die Anwendung entsprechend nicht genutzt und könnte genauso gut beendet werden. Das Problem dabei ist, dass der Zustand vorher gespeichert und beim nächsten Start wieder geladen werden muss. Allerdings sind aktuelle *Cloud-Deployment-Systeme* nur in der Lage ein Abbild einer kompletten VM zu machen, um den Zustand eines Systems abzuspeichern. Dieses Verfahren bringt jedoch die folgenden Nachteile mit sich:

3. Verwandte Arbeiten

1. *VM*-Snapshots benötigen typischerweise viel Speicherplatz. Dieses Problem steigert sich noch, wenn für die Sicherung einer Anwendung Snapshots von mehreren *VMs* angelegt werden müssen. Dabei wird häufig nur der Zustand einzelner Komponenten, die in der *VM* laufen, benötigt. Harzenetter et al. zeigen dies am Beispiel einer *MySQL*³-Datenbank:

Die *MySQL*-Datenbank wird in einer *VM* betrieben, auf der die Linux-Distribution *Ubuntu*⁴ als OS installiert ist. Ein Snapshot der *VM* würde also entsprechend neben den Daten der Datenbank den Datenbank-Server selbst sowie die *Ubuntu*-Installation enthalten.

2. Wird die *VM* wiederhergestellt, kann es dazu kommen, dass die Konfiguration der *VM* aufgrund veralteter Referenzen auf externe Ressourcen ungültig geworden ist. Eine Ursache hierfür kann sein, dass sich bei der *Reprovisionierung* die Internet-Protokoll (IP) Adresse der *VM* ändert, welche die externen Ressourcen bereit stellt.

Daher stellen Harzenetter et al. in ihrer Arbeit ein Konzept vor, mit dem sich *zustand-behaftete* Komponenten archivieren und zu einem späteren Zeitpunkt in exakt diesem Zustand wiederherstellen lassen. Als PoC wird das vorgestellte Konzept auf Basis der *OpenTOSCA*-Plattform umgesetzt. Weil *TOSCA* nur die (De-) Provisionierung von *Cloud-Anwendungen* vorsieht, wird das Modell um die Funktionen *Freeze* und *Defrost* erweitert.

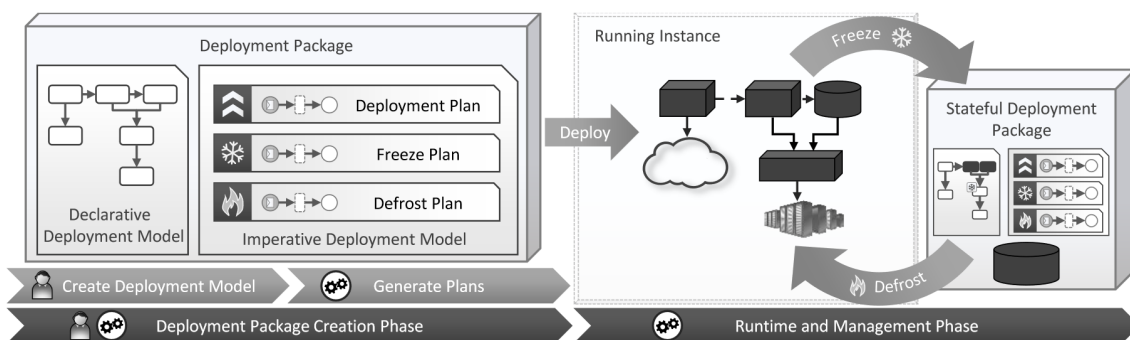


Abbildung 3.4.: OpenTOSCA im Detail
Entnommen aus: [HBKL19]

OpenTOSCA arbeitet grundsätzlich in drei Phasen, welche in Abbildung 3.4 dargestellt sind und im Folgenden näher erläutert werden. Dabei werden auch die Schritte *Freeze* und *Defrost* näher betrachtet.

1. **Deployment Package Creation Phase:**

Zunächst benötigt *OpenTOSCA* ein deklaratives *Deployment-Modell* welches angibt, wie die einzelnen Komponenten zusammen arbeiten. Dieses Modell muss manuell über das Werkzeug *winery* angelegt werden.

Aus dem *Deployment-Modell* kann *OpenTOSCA* den *Deployment*-, *Freeze*- und *Defrost*-Plan erzeugen. Diese Pläne bestimmen, in welcher Reihenfolge die jeweiligen Komponenten gestartet oder heruntergefahren werden müssen.

³MySQL - <https://www.mysql.com>

⁴Ubuntu - <https://ubuntu.com>

2. Deployment-Package:

Nach Anlegen des Deployment-Modells und der Ableitung der entsprechenden Pläne werden diese Daten zu einem *Deployment-Package* zusammengefasst. Dadurch enthält es alle Daten, die nötig sind, um die im *Deployment-Modell* angegebenen Komponenten in die Cloud zu deployen.

3. Runtime and Management Phase:

Während der *Runtime and Management Phase* überwacht die TOSCA-Runtime die laufenden *Cloud-Anwendungen*. Zudem stehen verschiedene Operationen für die verfügbaren *Deployment Packages* und die laufenden Anwendungen zur Verfügung:

- **Deploy:**

Wird eine Anwendung über *OpenTOSCA* gestartet, wird das *Deployment-Package* geladen und ausgeführt. Nachdem der Prozess abgeschlossen ist, wird die laufende Anwendung weiterhin von *OpenTOSCA* überwacht und die einzelnen Komponenten bei Bedarf skaliert.

- **Freeze:**

Um eine laufende Anwendung über *OpenTOSCA* „einzufrieren“, wird der *Freeze Plan* des *Deployment-Packages* genutzt. Die Komponenten werden entsprechend des Planes angehalten und neben dem *Deployment-Modell* und den erzeugten Plänen in einem *Stateful Deployment Package* als sogenannte *state artifacts* archiviert.

- **Defrost:**

Die archivierte Anwendung, welche als *Stateful Deployment Package* vorliegt, kann über den Prozess des „Auftauens“ unter Anwendung des *Defrost-Planes* wieder in die Cloud deployt werden. Der *Defrost-Plan* gibt hierbei vor, in welcher Reihenfolge die Komponenten in der Cloud angelegt und mit den archivierten Daten wieder in den ursprünglichen Zustand gebracht werden.

4. Konzept

Im Rahmen dieses Kapitels wird das Konzept für den *Prototyp (CloudFridge)* erarbeitet und vorgestellt. Hierzu werden zunächst die Ansätze der verwandten Arbeiten analysiert und kategorisiert, um daraus die Grundlagen für die *CloudFridge* zu bilden. Anschließend wird das Konzept erarbeitet und dargestellt.

4.1. Analyse bestehender Ansätze verwandter Arbeiten

Dieser Abschnitt reflektiert die Ansätze der verwandten Arbeiten und extrahiert daraus Lektionen, welche für die *CloudFridge* berücksichtigt werden müssen.

4.1.1. A Survey of Migration Mechanisms of Virtual Machines

Medina et al. stellen in ihrer Arbeit (vgl. Abschnitt 3.1) verschiedene Migrations-Mechanismen von *VMs* vor, welche sie auf Basis von anderen Arbeiten in verschiedene Kategorien unterteilen [MG14]. Dies ist nützlich um daraus einen Überblick möglicher Algorithmen für die *CloudFridge* zu gewinnen. Als Ergebnis präsentieren die Autoren die Kategorien *Live Migration*, *Suspend / Resume Migration* und *Load Balancing / Green Computing*.

- **Load Balancing / Green Computing:**

In die Kategorie *Load Balancing / Green Computing* fallen Verfahren, welche im Kern auf Migrationstechniken zurück greifen. Hauptziel der Algorithmen ist aber, die Ressourcen der Cloud möglichst optimal auszulasten und übrige Server gegebenenfalls herunterzufahren um dadurch Strom zu sparen und die Effizienz zu erhöhen. Die Konzepte beziehen sich dabei fast ausschließlich auf *VMs* und nutzen im Kern entweder *Live Migration* oder *Suspend / Resume Migration*.

- **Live Migration:**

Die Konzepte der *Live Migration* erläutern, wie sich *VMs* zur Laufzeit mit möglichst geringer Downtime migrieren lassen. Die Ansätze werden noch weiter in *Pre-copy*- und *Post-copy*-Verfahren unterteilt. Weiterhin werden spezielle Verfahren für die Migration über WAN genannt. In beiden Fällen wird aber hauptsächlich das Problem gelöst, wie Migrationen sich möglichst schnell und mit möglichst geringer Downtime durchführen lassen. Die genannten Konzepte arbeiten darüber hinaus ausschließlich mit *VMs* und lassen sich daher nur bedingt auf *AWS-Komponenten* übertragen.

- **Suspend / Resume Migration:**

In dieser Kategorie fassen Medina et al. Verfahren zusammen, welche zunächst die zu migrierenden Systeme anhalten und erst anschließend die Migration starten. Da die CloudFridge von einem angehaltenen *Cloud-System* ausgeht, um Zustandsänderungen des Systemes zu vermeiden, fällt der Ansatz ebenso in diese Kategorie.

Dabei fällt auf, dass alle Systeme letztlich Zugriff auf das zugrunde liegende Betriebssystem nehmen, um die Migration umzusetzen. Sei es wie bei *SoulPads*, welches im Kern einen Snapshot einer *VM* migriert oder *Capsule*, welches zur Reduktion des zu migrierenden RAM eng mit dem OS zusammenarbeitet. Damit wird klar, dass die Archivierung der *zustand-behafteten* Komponenten davon abhängt, ob die *AWS-API* die nötigen Zugriffe erlaubt. Schließlich handelt es sich hierbei um *cloudspezifische* Komponenten mit vordefinierten Funktionen. Daher ist es möglich, dass auf die *zustand-behafteten* Daten nicht zugegriffen werden kann.

Die Autoren geben in ihren verwandten Arbeiten noch den Hinweis, dass

"Logging und Replay [...] für die Wiederherstellung eines Zustands weit verbreitet [sind].", sinngemäß nach [MG14]

Dies kann eine mögliche Lösung für das Problem des unzureichenden Zugriffs darstellen (vgl. Abbildung 5.3).

4.1.2. Algorithms for automated live migration of virtual machines

In der Arbeit von Forsman et al. wird ein Verfahren vorgestellt, durch welches die Last der einzelnen *VMs* möglichst gleichmäßig auf alle *PMs* verteilt werden kann. Entsprechend der Kategorisierung von Medina et al. handelt es sich bei dem Verfahren um ein *Load Balancing*-Verfahren. Zwar finden *Push*- und *Pull*-Strategie für die *CloudFridge* keine Anwendung, aber die Autoren stellen verschiedene Herangehensweisen für die *VM*-Migration vor, welche interessante Grundlagen liefern.

- **Live Migration:**

Während Medina et al. in ihrer Arbeit den Fokus auf verschiedene Migrations-Technologien setzen, gehen Forsman et al. in dieser Arbeit näher auf allgemeine Schwierigkeiten der *Live Migration* ein. Sie zeigen, dass insbesondere der Zustand von CPU, RAM sowie die Netzwerkverbindungen bei dieser Migrationsart besondere Hürden stellen. Ursächlich dafür ist, dass bei der *Live Migration* die *VM*, wenn überhaupt, nur für ein kurzes Zeitfenster angehalten werden darf.

Für die *CloudFridge* ist insbesondere interessant, wie sich ein solches System trotz ständiger Änderungen migrieren oder archivieren lässt. Gerade *Cloud-Anwendungen* stellen dabei eine Herausforderung dar, da das System während der Archivierung aufgrund der *Microservice-Architektur* an vielen verschiedenen Stellen den Zustand ändern kann.

- **Cold Migration:**

Die *Cold Migration* ist das exakte Gegenteil der *Live Migration*: Das System wird heruntergefahren, migriert und wieder in Betrieb genommen. Das ist die Funktionalität, welche

OpenTOSCA bereits bereitstellt: Ein System kann gestartet und wieder gestoppt werden. Dabei ist es egal, bei welchem Anbieter die Anwendung wieder gestartet wird, sofern dieser den *TOSCA Standard* unterstützt.

- **Hot Migration:**

Die *Hot Migration* wird von den Autoren nur kurz angeschnitten. Dabei wird dennoch der Unterschied zur *Live Migration* deutlich: Das System wird zwar für die Dauer der Migration angehalten, wodurch sich der Zustand des Systems nicht ändern kann aber der Zustand bleibt dabei vollständig erhalten.

Für die *CloudFridge* eignet sich diese Art der Migration deshalb besonders. Dabei stellt allerdings wieder der *Cloud-Anbieter* die Hürde dar: Lassen sich die *zustand-behafteten* Komponenten pausieren? Falls nicht, muss das System anderweitig pausiert werden. Dies kann beispielsweise dadurch erzielt werden, dass die zustandsändernden Module als erstes archiviert und beendet werden.

4.1.3. Stateful Process Migration for Edge Computing Applications

Im Rahmen dieser Arbeit beschreiben Horii et al. ein System für die Migration von Prozessen. Hierbei unterscheiden die Autoren zunächst zwischen *zustand-behafteten* und *zustand-freien* Prozessen. Hintergrund dafür ist, dass das Vorgehen vorsieht, dass ein äquivalentes System inklusive aller *zustand-freien Prozesse* bereits auf dem Zielsystem existiert. Dadurch müssen ausschließlich die *zustand-behafteten Prozesse* migriert werden.

Dieser Ansatz ist daher aufgrund mehrerer Aspekte interessant:

1. Es geht bei Horii et al. nicht um die Migration von *VMs*. Die Migration von Prozessen lässt sich nicht einfach durch einen Snapshot bewerkstelligen sondern ist bedeutend aufwändiger.
2. Die *zustand-freien Prozesse* werden vollständig ignoriert. Auch bei der *CloudFridge* besteht die Herausforderung darin, die *zustand-behafteten Komponenten* zu migrieren, da die anderen Komponenten sich bereits durch *OpenTOSCA* stoppen und wieder starten lassen.
3. Anhand der Migration von Prozessen wird aufgezeigt, dass auch die Kanäle zwischen den einzelnen Komponenten migriert werden müssen. Da die einzelnen Module von *Cloud-Anwendungen* ebenfalls über Kanäle miteinander interagieren, muss dies für die Entwicklung der *CloudFridge* ebenso beachtet werden.

Für das Verfahren der eigentlichen Prozess-Migration verweisen die Autoren auf bereits existente Techniken wie beispielsweise Zap [OSSN02]. Zap stellt dabei eine leichtgewichtige Virtualisierung zwischen OS und Anwendungen zur Verfügung, wodurch sich auch Prozesse migrieren lassen. Dieses Verfahren lässt sich jedoch nicht auf die *CloudFridge* übertragen, da dort nur mit vorgefertigten Modulen des *Cloud-Anbieters* gearbeitet werden kann.

4.1.4. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing

Lagar-Cavilla et al. zeigen in ihrer Arbeit, wie sich *VMs* durch ein leichtgewichtiges Verfahren in kurzer Zeit vervielfachen lassen. Dabei ist vor allem die *Lazy State Replication* interessant:

Der Ansatz der *Lazy State Replication* besagt, dass zunächst nur Daten migriert werden, die zwingend für den Betrieb erforderlich sind. Sollte die *VMs* darüber hinaus Daten benötigen, werden diese bei Bedarf über das Netzwerk nachgeladen. Da alleine das Bereitstellen der Komponenten durch die *AWS-Cloud* je nach Komponente einige Minuten dauern kann, ist es sinnvoll, zu untersuchen, ob sich bei der Wiederherstellung des Zustands Daten sparen lassen. Möglicherweise kann hierdurch die Bereitstellung der Komponente beschleunigt werden. In jedem Fall wird jedoch die Bandbreite geschont, da weniger Daten zum *Cloud-Anbieter* hochgeladen werden müssen.

4.1.5. Freezing and Defrosting Cloud Applications: Automated Saving and Restoring of Running Applications

Harzenetter et al. zeigen in ihrer Arbeit, wie sich Anwendungen auf Grundlage von *OpenTOSCA* archivieren und wieder in Betrieb nehmen lassen. Hieraus ergeben sich diverse Konsequenzen für die *CloudFridge*.

OpenTOSCA ist in der Lage Anwendungen in die Cloud zu deployen. Dabei wird zwar der Zustand der Anwendung nicht wiederhergestellt aber zumindest die Basis-Anwendung. Die *CloudFridge* muss sich entsprechend nur noch darum kümmern, die *zustand-behafteten Komponenten* zu archivieren.

Da *OpenTOSCA* bereits die Grund-Anwendung in die Cloud deployt, ist es nicht nötig, die Kommunikationskanäle der einzelnen Komponenten zu archivieren. Die Kanäle werden beim Reinstanzieren der Anwendung automatisch korrekt angelegt. Die *CloudFridge* würde anschließend in einem zweiten Schritt die *zustand-behafteten Komponenten* wieder in den Zustand bringen, den diese vor der Archivierung hatten.

Auch die Reihenfolge, in welcher die Komponenten von der *CloudFridge* angelegt werden, ist für die Entwicklung des Prototyps nicht ausschlaggebend. Die Reihenfolge kann von *OpenTOSCA* vorgegeben werden. Wird der Prototyp entsprechend erweitert und in *OpenTOSCA* integriert, ist das Problem ebenfalls behoben. Selbiges gilt auch für die Reihenfolge, in der die Komponenten archiviert werden.

4.2. Konzeption des Prototyps

Im Folgenden wird das Konzept für den Prototyp vorgestellt. Hierbei wird auf die Lektionen zurück gegriffen, welche durch die Analyse der verwandten Arbeiten (vgl. Abschnitt 4.1) gewonnen werden konnten.

Das **Ziel des Prototyps** ist es zu zeigen, ob und wie sich *zustand-behaftete Komponenten* der *AWS-Cloud* archivieren lassen. Hierzu werden die gewonnenen Lektionen genutzt und in einen Prototyp umgesetzt. Dabei werden für die Entwicklung verschiedene Vereinfachungen angenommen:

- Der Prototyp muss keine *zustand-freien Komponenten* archivieren können.

- Die Archivierungs-Reihenfolge spielt zunächst keine Rolle.
- Bei der Archivierung von Komponenten wird angenommen, dass das System angehalten ist und sich die Daten darin nicht mehr ändern.
- Die Kommunikations-Kanäle zwischen den Komponenten werden nicht archiviert.
- Die Komponenten werden sequenziell abgearbeitet.

Damit kann der Proof of Concept (PoC) sich darauf konzentrieren, die *zustand-behafteten Komponenten* zu archivieren. Dabei ist zu untersuchen, wie sich diese in der *AWS-Cloud* verhalten und worauf bei der Archivierung zu achten ist.

Nur wenn die Archivierung der Komponenten erfolgreich war, darf die Anwendung aus der Cloud entfernt werden. Aus diesem Grund besteht sowohl die Archivierung als auch die Reprovisionierung aus mehreren Schritten:

- **Archivierung:**
Wie Abbildung 4.1 zeigt, besteht das Vorgehen aus drei Schritten, die aufeinander aufbauen.
 1. **Suche von Elementen:**
Zunächst muss die *CloudFridge* herausfinden, aus welchen Komponenten die *Cloud-Anwendung* besteht. Diese werden über die *Freezer* des jeweiligen *Cloud-Anbieters* über die Funktion `CollectFreezableObjects` abgefragt. Dabei erhält die *CloudFridge* von jedem *Freezer* eine Liste von *FreezableObjects* zurück.
 2. **Elemente archivieren:**
Bevor die Archivierung gestartet wird, kann die Liste der *FreezableObjects* von der *CloudFridge* angefordert und verändert werden. Durch den Aufruf von `SerializeObjects` wird für jede Komponente der hinterlegte Serialisierer ausgeführt. Die Daten werden sowohl in den *FreezableObject*-Instanzen als auch auf der Festplatte gespeichert. Treten während der Archivierung Fehler auf, bricht die *CloudFridge* den Vorgang ab, wodurch das System unverändert bleibt.
 3. **Sequenzielles Herunterfahren:**
Wie schon nach der Suche der Elemente, kann auch vor dem Herunterfahren die Reihenfolge der Elemente verändert werden. Durch Aufruf von `TransactiveShutdown` werden die Komponenten von der *CloudFridge* sequenziell heruntergefahren. Sollte dabei ein Fehler auftreten, werden die bereits heruntergefahrenen Komponenten wieder hochgefahren und der Zustand wiederhergestellt.

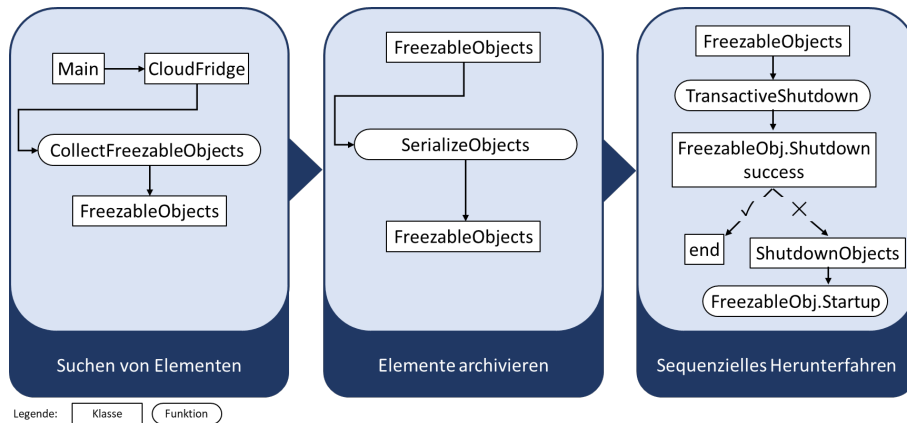


Abbildung 4.1.: CloudFridge: Archivierungs-Schema

• **Reprovisionierung:**

Die Reprovisionierung benötigt, im Gegensatz zur Archivierung, nur zwei Schritte, da die Suche nach Elementen entfällt (vgl. Abbildung 4.2):

1. **Daten laden:**

Über die Funktion `DeserializeObjects` werden die archivierten Objekte in die *Cloud-Fridge* geladen und stehen damit für weitere Schritte als Liste von *FreezableObjects* zur Verfügung.

2. **Sequenzielles Starten:**

Auch hier kann die Reihenfolge der *FreezableObjects* verändert werden, ehe das sequenzielle Starten über `TransactiveStartup` angestoßen wird. Die Methode versucht sequenziell die *FreezableObjects* zu reprovisionieren. Sollten dabei Fehler auftreten, wird der Prozess abgebrochen und die bereits provisionierten Komponenten werden wieder beendet.

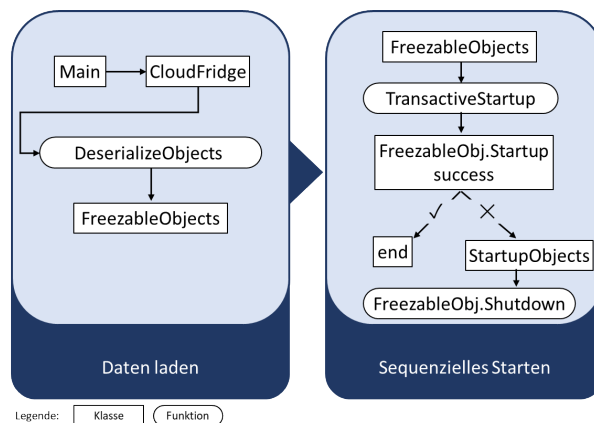


Abbildung 4.2.: CloudFridge: Reprovisionierungs-Schema

5. Entwicklung des Prototyps

Im Rahmen dieser Arbeit wird ein Prototyp (CloudFridge) für ein System zur automatisierten Archivierung und Wiederinbetriebnahme von *Cloud*-Anwendungen entwickelt. Im Folgenden sollen sowohl Design-Entscheidungen begründet als auch die Architektur des Prototyps erläutert werden. Darüber hinaus werden die verschiedenen Probleme, die bei der Entwicklung aufgetreten sind, dargelegt.

5.1. Design-Entscheidungen

Ziel der Arbeit ist es, einen Prototypen zu entwickeln, der möglichst einfach weiter entwickelt werden kann. Hierfür erscheint die Sprache Java aus mehreren Gründen als die beste Option: Laut TIOBE-Index ist Java noch immer die am meisten genutzte Programmiersprache [TIOBE19]. Hinzu kommt, dass Java plattformunabhängig ist und durch das *OpenJDK*¹ eine quelloffene Laufzeitumgebung zur Verfügung steht. Damit wird sichergestellt, dass zukünftige Arbeiten nicht aufgrund proprietärer Produkte erschwert werden.

Der Prototyp wird für die *Amazon AWS*-Plattform entwickelt. Alternativen wie *Microsoft Azure*² oder *Google Cloud Platform*³ werden aufgrund der ausreichenden Leistung der *AWS-Dienste* nicht näher betrachtet, zumal die *Amazon-Cloud* (Stand 2018) den größten Marktanteil hat und damit Plattform für die meisten *Cloud*-Anwendungen ist [SR18].

Die Implementierung des Prototyps basiert auf der *Amazon AWS-SDK*-Version 1.11. Zwar steht auch ein SDK in Version 2.0 zur Verfügung, welches sich aber nicht entsprechend der Anleitung [AWS19f] installieren lässt.



Abbildung 5.1.:
CloudFridge Logo, © 2019 F. Amend

¹OpenJDK - <https://openjdk.java.net>

²Microsoft Azure - <https://azure.microsoft.com>

³Google Cloud Platform - <https://cloud.google.com>

3. Zuletzt können die Objekte mittels `TransactiveShutdown` heruntergefahren werden. Dabei wird jede Komponente separat angewiesen herunterzufahren. Sollte dabei ein Fehler auftreten wird der Prozess beendet und die bereits heruntergefahrenen Komponenten werden wieder gestartet.

Die *Re-Provisionierung* erfolgt auf ähnliche Weise wie die Archivierung (vgl. Abbildung 4.2):

1. Die Funktion `DeserializeObjects` erwartet im angegebenen Verzeichnis die archivierten *Cloud*-Objekte. Diese werden geladen und der *CloudFridge* zur Verfügung gestellt.
2. Anschließend kann über den `TransactiveStartup` die *Cloud*-Anwendung wieder gestartet werden. Dabei werden die einzelnen Komponenten angewiesen zu starten. Sollte dabei ein Fehler auftreten, wird der Prozess beendet und die bereits gestarteten Komponenten werden wieder heruntergefahren.

Für den Fall, dass die (De-) Serialisierung der Objekte einen Fehler erzeugen sollte, wird dieser von der *CloudFridge* an die äußere Logik weitergegeben. Daher umschließt die *Main*-Klasse die Operationen mit einer try-catch-Anweisung.

5.2.2. CredentialStorage

Der *CredentialStorage* ist eine Hilfsklasse, in der die verschiedenen Anmeldeinformationen hinterlegt werden können. Über diese Klasse können alle *Freezer* (Klassen, die von *IFreezer* ableiten) sowie die einzelnen *FreezableObjects* die nötigen Anmeldeinformationen abrufen und diese für ihre Operationen in der *Cloud* nutzen. Der *CredentialStorage* ist intern als Verzeichnis aufgebaut, welches zu einem Datentyp (`Class<?>`) ein Objekt des Datentyps zuordnet. Dabei enthält das Objekt die entsprechenden Anmeldeinformationen. Grundsätzlich stellt die Klasse für diese Aufgabe zwei generische Methoden bereit:

- `Store`: Über diese Funktion können Anmeldeinformationen im *CredentialStorage* hinterlegt werden. Die Funktion erwartet dabei die Angabe des Datentyps sowie ein Objekt mit den Anmeldeinformationen.
- `Read`: Die Funktion ist das logische Gegenstück und gibt nach Angabe des Datentyps das zugehörige Objekt mit den Anmeldeinformationen aus.

Da allerdings Nutzer des *CloudFridge*-Prototyps die jeweils genutzten Objekt-Klassen nicht kennen, werden diese generischen Funktionen nur für den klassen-internen Gebrauch freigegeben. Zur einfachen Handhabung nach außen muss die Klasse um Funktionen erweitert werden, die das Speichern und Laden der jeweiligen Anmeldeinformationen ermöglichen. Dies muss dem Aufrufer auch ohne Kenntnis der intern genutzten Objekte möglich sein.

Im Falle der *AWS-Cloud* wird hierfür die Funktion `AwsCredentials(String accessKey, String secretKey)` bereit gestellt, welche als Parameter lediglich den *AccessKey* sowie den *SecretKey* benötigt. Um die Anmeldedaten abzurufen, steht die parameterfreie Funktion `AwsCredentialsProvider()` zur Verfügung, welche die Daten in Form des AWS-internen Objekts (*AWSCredentialsProvider* [AWS19d]) zurück gibt.

5.2.3. ReflectionManager

Der *ReflectionManager* dient der *CloudFridge* als Möglichkeit, neu hinzugefügte *Freezer* oder *archivierbare Elemente (FreezableObjects)* ohne weitere Code-Änderungen zu berücksichtigen. Allerdings ist in Java das Suchen von Klassen mittels Reflection nur bedingt möglich. Der Weg führt darüber, dass die verschiedenen *Packages*⁴ eines *ClassLoaders*⁵ zur Laufzeit durchsucht werden müssen. Werden demnach alle Klassen gesucht, die ein bestimmtes Interface implementieren, müssten dafür bei jeder Suche alle *Packages* durchsucht werden.

Aus diesem Grund erstellt der *ReflectionManager* über die Funktion `Refresh` einen Index der verfügbaren Klassen und deren Eigenschaften. Über diesen Index können die Klassen zur Laufzeit deutlich performanter ermittelt werden. Für den Abruf stellt der *ReflectionManager* drei Funktionen bereit (`GetIterator`, `GetList` & `GetClasses`), die sich nur im Rückgabe-Typ unterscheiden. Diese Funktionen erwarten als Eingabe-Parameter die Angabe, nach welchen Klassen gesucht wird. Dabei werden alle Klassen zurückgegeben, die sich von einer der angegebenen Klassen ableiten.

5.2.4. CloudFridge

Die Klasse *CloudFridge* stellt das Herzstück des Prototyps dar. Sie ermöglicht es die verschiedenen Elemente der *Cloud*-Anwendung in der *Cloud* zu ermitteln und diese anschließend zu serialisieren. Hierbei setzt die *CloudFridge* auf ein sehr abstraktes Datenmodell, sodass problemlos weitere *Cloud*-Objekte der *AWS* oder sogar anderer *Cloud*-Anbieter integriert werden können. Die *CloudFridge* steht dabei als Controller über den *Freezern* und den *FreezableObjects*.

- `CollectFreezableObjects`:

Bei der Initialisierung der *CloudFridge* werden die enthaltenen *Freezer* mittels Reflection gesucht und als Instanz hinterlegt. Wie Algorithmus 5.1 zeigt, iteriert `CollectFreezableObjects` über die Liste der *Freezer* und nutzt deren `GetFreezableObjects`-Methode, um die einzelnen *Cloud-Objekte* zu ermitteln. Die *Cloud-Objekte* werden für die weitere Verarbeitung als Objekt in einer Liste abgespeichert.

⁴Java *Package*-Klasse - <https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Package.html>

⁵Java *ClassLoader*-Klasse -

<https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/ClassLoader.html>

Algorithmus 5.1 Discovery-Algorithmus für die Suchen nach archivierbaren Elementen

```

function COLLECTFREEZABLEOBJECTS
  freezableObjects := List<>
  freezers ← REFLECTIONMANAGER.GETLIST(Class<IFreezer>)

  for all freezer ∈ freezers do
    for all freezObj ∈ FREEZER.GETFREEZABLEOBJECTS do
      FREEZABLEOBJECTS.ADD(freezObj)
    end for
  end for

  // FreezableObjects in der CloudFridge-Instanz abspeichern
  this::FreezableObjects ← freezableObjects
end function

```

- **SerializeObjects** :
Die Funktion *SerializeObjects* sorgt dafür, dass die einzelnen *Cloud-Objekte* serialisiert werden. Auch hier dient die *CloudFridge* nur als Controller, der alle *Cloud-Objekte* sequenziell abarbeitet und deren Methode für die Serialisierung aufruft (vgl. Algorithmus 5.2). Dabei kann die Liste der *Cloud-Objekte* vorab noch verändert werden: Es dürfen sowohl Objekte hinzugefügt oder gelöscht als auch die Reihenfolge der Objekte geändert werden.

Algorithmus 5.2 Routine für das Serialisieren der einzelnen FreezableObjects

```

function SERIALIZEOBJECTS(dir)
  // FreezableObjects aus der CloudFridge-Instanz laden
  freezableObjects ← this::FreezableObjects

  for all freezableObj ∈ freezableObjects do
    freezableObj.JSON ← FREEZABLEOBJ.SERIALIZE
    fileName ← GETFILENAME(freezableObj)
    WRITEFILE(dir, fileName, freezableObj.JSON)
  end for
end function

```

- **TransactiveShutdown** :
Wie Algorithmus 5.3 zeigt, werden nach erfolgreicher Serialisierung die einzelnen *Cloud-Objekte* sequenziell heruntergefahren. Dazu wird die **Shutdown**-Methode der *Cloud-Objekte* genutzt. Auch hier darf die Liste der *Cloud-Objekte* vorab noch verändert werden. Schlägt das Herunterfahren fehl, versucht die *CloudFridge* die bereits heruntergefahrenen Objekte wieder hoch zu fahren.

Algorithmus 5.3 Routine für das transaktionale Herunterfahren der Komponenten

```
function TRANSACTIVESHUTDOWN
  // FreezableObjects aus der CloudFridge-Instanz laden
  freezableObjects ← this::FreezableObjects
  shutdownObjects ← List<>
  success ← true

  for all freezableObj ∈ freezableObjects do
    SHUTDOWNOBJECTS.ADD(freezableObj)

    if FREEZABLEOBJ.SHUTDOWN = false then
      success ← false
      exit for
    end if
  end for

  if success = false then
    forall shutdownObjects do FREEZABLEOBJ.STARTUP(
  end if)
end function
```

- `DeserializeObjects` :
Um die archivierten *Cloud-Objekte* zu reprovisionieren müssen diese zunächst aus dem Archiv geladen werden. Diese Aufgabe übernimmt die Funktion *DeserializeObjects*, dargestellt in Algorithmus 5.4, indem alle Dateien eines Verzeichnisses eingelesen und über die entsprechenden Objekte letztlich deserialisiert werden. Welches Objekt die abgespeicherten Daten deserialisieren kann, entnimmt die *CloudFridge* der Archiv-Datei. Anhand des Klassennamens kann die *CloudFridge* durch Nutzung der Reflection eine Klassen-Instanz des *Cloud-Objekts* erzeugen und dieser die serialisierten Daten zur Deserialisierung übergeben.

Algorithmus 5.4 Routine für das Laden und die Deserialisierung der Komponenten

```

function DESERIALIZEOBJECTS(dir)
  freezableObjects := List<>

  for all file ∈ dir do
    // Datei einlesen
    content ← READFILE(file)
    freezableObj ← FREEZABLEOBJECT.FROMSTRING(content)

    // Cloud-Objekt-Repräsentation erzeugen und mit Daten füllen
    freezable ← FREEZEOBJ.CREATEFREEZABLEOBJECTINSTANCE
    FREEZABLE.DESERIALIZE(freezableObj.Data)
    FREEZABLEOBJECTS.ADD(freezable)
  end for

  // FreezableObjects in der CloudFridge-Instanz abspeichern
  this::FreezableObjects ← freezableObjects
end function

```

- **TransactiveStartup** :
TransactiveStartup ist das logische Gegenstück zu der Funktion *TransactiveShutdown* und sorgt dafür, dass die geladenen *Cloud-Objekte* reprovisioniert werden. Hierzu greift auch diese Funktion auf die Liste der *Cloud-Objekte* in der CloudFridge zurück, welche vorab bearbeitet werden darf. Schlägt die Reprovisionierung für ein *Cloud-Objekt* fehl, wird der Vorgang abgebrochen und alle bereits reprovisionierten Elemente werden gelöscht.

Algorithmus 5.5 Routine für das transaktionale Starten der Komponenten

```

function TRANSACTIVESTARTUP
  // FreezableObjects aus der CloudFridge-Instanz laden
  freezableObjects ← this::FreezableObjects
  startupObjects ← List<>
  success ← true

  for all freezableObj ∈ freezableObjects do
    STARTUPOBJECTS.ADD(freezableObj)

    if FREEZABLEOBJ.STARTUP = false then
      success ← false
      exit for
    end if
  end for

  if success = false then
    for all startupObjects do FREEZABLEOBJ.SHUTDOWN
  end if
end function

```

- `Reset` :
Die Methode *Reset* sorgt dafür, dass die Liste der *FreezableObjects* geleert wird. Dies ist nur für den Test der *CloudFridge* nötig, da diese zunächst die *Cloud-Objekte* sammelt und herunter fährt, wobei die Liste der Objekte angelegt wird. Vor der Reprovisionierung muss die *CloudFridge* wieder in ihren Initialzustand überführt werden, was durch die *Reset*-Methode ermöglicht wird.

5.2.5. Interface *IFreezer*

Freezer stellen der *CloudFridge* die Funktion `GetFreezableObjects()` zur Verfügung und bilden damit die Zwischenschicht zwischen *CloudFridge* und den *FreezableObjects*. Designtechnisch wäre es möglich gewesen, diese Funktion in die *FreezableObjects* aufzunehmen. Durch Nutzung der Reflection müsste aber zunächst eine *FreezableObjects*-Instanz angelegt werden, ehe die Funktion zum Abruf der archivierbaren Objekte aufgerufen werden kann. Aus diesem Grund wurde die Architektur so gestaltet, dass die *IFreezer* als Kontrollklasse für Operationen dienen, welche die jeweilige Cloud betreffen. Dem untergeordnet stellen *FreezableObjects* eine Instanz eines Cloud-Dienstes und damit die atomaren Elemente der Cloud-Anwendung dar.

Den *Freezern* wird von der *CloudFridge* der *CredentialStorage* zum Abruf der Anmeldeinformationen bereitgestellt. Dazu ruft die *CloudFridge* nach Erzeugen der *Freezer*-Instanz die Funktion `CredentialStorage(CredentialStorage credentialStorage)` auf und übergibt damit die Instanz des *CredentialStorages* an den *Freezer*.

Der `Name()` des *Freezers* dient lediglich zur Unterscheidung der verschiedenen *Freezer*-Instanzen bei Debug-Ausgaben.

5.2.6. *AWSFreezer*

Der *AWSFreezer* ist die Implementierung eines *IFreezers* für die *AWS-Cloud* und dient darüber hinaus als Beispiel-Implementierung.

Die *AWS-Cloud* ist in verschiedene Regionen unterteilt, welche sich, mit wenigen Ausnahmen, aus den Standorten der Amazon-Rechenzentren herleiten. Für die Bereitstellung eines Dienstes muss angegeben werden, in welcher Region dieser bereitgestellt werden soll. Im Falle der *CloudFridge* werden beispielsweise alle Komponenten für den PoC in der Region „EU_CENTRAL_1“ gehostet. Daher müssen alle Regionen durchsucht werden, um eine vollständige Liste der bereitgestellten Dienst-Instanzen zu erhalten. Dafür wird eine interne Hilfsfunktion bereitgestellt, welche über die verfügbaren Regionen iteriert und alle laufenden Instanzen des angegebenen Dienstes ermittelt.

Der Abruf der verschiedenen Cloud-Dienste wird über die Amazon-API durchgeführt, welche für die unterschiedlichen Dienste unterschiedliche Abfragen bereitstellt. Die Implementierung des Abrufs unterscheidet sich dementsprechend von Element zu Element. Da es die Aufgabe des *Freezers* ist, die verschiedenen Dienste des jeweiligen Cloud-Anbieters zu kennen, können die Abfragen als statische Methode im jeweiligen *FreezableObject* implementiert und vom *Freezer* direkt aufgerufen werden. Diese Liste der archivierbaren Elemente wird anschließend an die *CloudFridge* zurückgegeben.

5.2.7. AbstractFreezable

Die Klasse *AbstractFreezable* stellt die abstrakte Basis-Klasse für die verschiedenen *FreezableObjects*, also die atomaren Einheiten der Cloud-Anwendung dar. Sie definiert dabei für die *CloudFridge* die Funktionen und Methoden für den Zugriff und fungiert damit als eine Art Interface.

Der Konstruktor erwartet bei der Erstellung eines *FreezableObjects* die eindeutige ID der Instanz als Eingabe-Parameter. Eindeutig bedeutet dabei lediglich, dass diese ID von keiner anderen Instanz des gleichen Typs verwendet wird. Diese ID wird in einem klasseninternen Feld gespeichert und lässt sich danach nicht mehr ändern. Damit ist das *FreezableObject* hinsichtlich der ID immutable.

Da sich in Java der Datentyp eines Objektes nicht mittels Reflection abrufen lässt, stellt die Basis-Klasse die abstrakte Methode `clazz()` zur Verfügung. Abstrakte Methoden wie `clazz()` müssen von den ableitenden Klassen implementiert werden. Entsprechend ist es die Aufgabe der ableitenden Klasse, die Funktion `clazz()` zu implementieren und hierbei den eigenen Datentyp zurück zu geben. Dieses Vorgehen ermöglicht es der *CloudFridge*, unbekannte Klassen über die Methoden der Reflection zu instanzieren und mit den erzeugten Objekten zu arbeiten.

Damit die *CloudFridge* die *FreezableObjects* archivieren und wieder aufsetzen kann, muss die Basis-Klasse der *FreezableObjects* die nötigen Methoden bereitstellen. Diese werden durch die abstrakten Funktionsdefinitionen `Serialize` und `Deserialize` sowie `Startup` und `Shutdown` ermöglicht.

Wie die Namen schon vermuten lassen, bieten die ersten beiden der genannten Funktionen die Möglichkeit, das *FreezableObject* zu (De-)Serialisieren. Dies beinhaltet sowohl die Nutz-Daten als auch jedwede Meta-Daten, die nötig sind, um den Zustand der jeweiligen Komponenten zu sichern. Da die Implementierung der Funktionen ebenfalls durch die ableitende Klasse geschieht, ist das Format der serialisierten Daten der jeweiligen Implementierung überlassen. Dadurch ist sichergestellt, dass sich auch unterschiedlichste Objekte archivieren lassen, obwohl das JSON-Format nicht in allen Fällen optimal ist.

Da die ID nicht ausreicht, um ein *FreezableObject* eindeutig zu identifizieren, stellt die Basis-Klasse die Methode `UniqueName` bereit. Für den statischen Zugriff ist die Methode `static UniqueName(Class<?> clazz, String id)` ebenfalls implementiert. Diese Methode verbindet den Namen des Objekt-Typs mit der eindeutigen ID der jeweiligen Dienst-Instanz. Der *UniqueName* wird von der *CloudFridge* sowohl für den Namen der JSON-Datei als auch für lesbare Debug-Ausgaben genutzt.

5.2.8. FreezableObject

Die *FreezableObjects* stellen die atomaren, archivierbaren Elemente dar, aus denen sich die Cloud-Anwendung zusammen setzt. Sie müssen von der *Basis-Klasse AbstractFreezable* abgeleitet werden, damit die *CloudFridge* mit diesen Objekten arbeiten kann. Dabei gibt die *Basis-Klasse* bereits einige Methoden und Funktionen vor, die von dem *FreezableObject* implementiert werden müssen. Diese Methoden und Funktionen stellen den Zugriff durch die *CloudFridge* sicher und fungieren somit als eine Art Interface. Neben diesen Funktionen darf das *FreezableObject* in beliebiger Weise abgeändert werden, um an das zu archivierende Cloud-Objekt angepasst zu werden. Im Falle von AWS-Cloud-Objekten muss beispielsweise ein internes Feld zur Verfügung stehen, in dem abgespeichert wird, in welcher Region die Dienst-Instanz läuft.

5.2.9. DisposableObject

Das *DisposableObject*-Konstrukt ist eine Hilfsklasse, die sich vom Java-Interface *AutoCloseable*⁶ ableitet. Das sogenannte *Automatic Resource Management* wurde mit Java 7 eingeführt [Ble10]. Dieses sorgt dafür, dass alle Objekte innerhalb eines Try-Catch-Blocks automatisch beim Verlassen des Blocks aufgeräumt werden. So können beispielsweise Datei-Zugriffe freigegeben werden, ohne dies manuell im *finally*-Statement des Try-Catch-Blocks anzuweisen.

Das *AutoCloseable*-Interface des Java-Frameworks stellt hierfür die Definition einer zu implementierenden `close()`-Funktion zur Verfügung. Diese Funktion wird von der generischen Hilfs-Klasse *DisposableObject* implementiert. Über den Konstruktor von *DisposableObject* werden beliebige Objekte unter Angabe einer *closeAction* in einer *DisposableObject*-Instanz gekapselt, welche die oben genannten Eigenschaften hat.

Vor allem die *AmazonClientBuilder*⁷, welche für die Zugriffe auf die einzelnen *Cloud*-Objekte benötigt werden, müssen nach Gebrauch wieder durch den Aufruf von `shutdown()` beendet werden, da jeweils nur ein aktiver *AmazonClientBuilder* Zugriff auf die *Cloud*-Objekte erhält. Wird der Zugriff nicht sauber beendet, führt das Erstellen eines neuen *AmazonClientBuilders* zu einer Exception.

5.3. Erkenntnisse und Hindernisse

Die Erkenntnisse und Hindernisse, welche sich während der Implementierung und dem Test des PoC ergaben, werden in diesem Kapitel vorgestellt und im anschließenden Fazit ausgewertet.

Während der Entwicklung des Prototyps fiel auf, dass das AWS-SDK häufig Inkonsistenzen aufweist: Für die Archivierung ist eine eindeutige Unterscheidung der Elemente vonnöten. Da jedes Element der *AWS-Cloud* über einen Identifikator (ID) verfügt, sollten sich die Elemente eindeutig zuordnen lassen. Allerdings fällt dabei auf, dass die Element-IDs unterschiedlich gestaltet sind. Hierdurch wird die ID auch durch das SDK über verschiedene Wege bereitgestellt. Bei **Amazon DynamoDBs** stellt der Name der jeweiligen Tabelle die ID dar und kann über die Funktion `AmazonDynamoDB.listTables().getTableNames()` ermittelt werden. Bei der **Amazon SQS** hingegen wird die *QueueUrl* als ID genutzt was durch den folgenden API-Aufruf ermöglicht wird: `AmazonSQS.listQueues().getQueueUrls()`. Für **Amazon ElastiCache**-Instanzen wird die Funktion `getCacheClusterId()` bereitgestellt, während für **Amazon RDS**-Instanzen die Funktion `getDBInstanceIdentifier` die ID zurück gibt.

Als weiteres Problem ist zu nennen, dass die *AWS-Cloud* für die Bereitstellung oder Löschung mancher Komponenten mehrere Minuten benötigt. Dies ist insbesondere bei der Reprovisionierung ein Problem, da die CloudFridge mit dem Einspielen der archivierten Daten warten muss, bis die Instanz von der *AWS-Cloud* angelegt und bereitgestellt wurde. Hierzu bietet das AWS-SDK sogenannte *waiters* an. Jedoch nur für die Objekte **Amazon DynamoDB**, **Amazon ElastiCache** und **Amazon RDS**. Bei einer **Amazon SQS** hingegen lässt sich nicht prüfen, ob die Instanz bereitgestellt

⁶AutoCloseable - <https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/AutoCloseable.html>

⁷AmazonClientBuilder - <https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/client/builder/AwsSyncClientBuilder.html>

oder gelöscht wurde. Dabei sind diese Funktionen für die CloudFridge wichtig, um zu ermitteln, ab wann eine Komponente bereit ist, die archivierten Daten wieder aufzunehmen. Insbesondere deshalb, weil einige Komponenten wie beispielsweise Amazon RDS mehrere Minuten für die Bereitstellung durch die Cloud benötigen.

Die Archivierung der **Amazon DynamoDB** verlief problemfrei, während bei der **Amazon SQS** ein Fehler im AWS-SDK das Abrufen aller Nachrichten aus der Queue verhindert. Allerdings kann bei der **Amazon SQS** zumindest eine Nachricht korrekt abgerufen und archiviert werden, was für den PoC ausreichend ist, da die CloudFridge zumindest grundsätzlich alle Daten der Komponenten über das AWS-SDK ermitteln und zu einem späteren Zeitpunkt wieder einspielen kann. Entsprechend wurde hierfür auch nicht nach einem Workaround gesucht.

Im Falle der **Amazon SQS** tritt darüber hinaus das Problem auf, dass nicht abgerufen werden kann, welche Komponenten die Nachricht aus der Queue bereits abgerufen haben. Hinsichtlich der *exactly-Once-Delivery* ergibt sich daraus das Problem, dass im Falle der Reprovisionierung diese Daten verloren gehen.

Bei der Archivierung der Komponenten **Amazon RDS** und **Amazon ElastiCache** verlief der Test teilweise negativ: Grundsätzlich lassen sich zwar die Konfigurationen der Komponenten abrufen, allerdings müssen die Daten über eine Drittanbietersoftware abgerufen werden. Das AWS-SDK stellt hierzu keine Funktionen bereit.

Weiterhin werden manche Komponenten an fest definierte Endpunkte gebunden. Das heißt, dass den Komponenten bei der Erstellung von der AWS eine zufällig gewählte Adresse zugewiesen wird, über welche auf die Komponenten zugegriffen werden kann. Im Falle einer **Amazon RDS**-Instanz sieht dieser beispielsweise so aus: `{"address": "meine-postgresql.cykixj0rlryc.eu-central-1.rds.amazonaws.com"}` Bei der Reprovisionierung wird jedoch keine Möglichkeit bereitgestellt, diesen Endpunkt genau so wiederherzustellen. In der Konsequenz bedeutet das für die *Cloud-Anwendung*, dass sich die Zugriffspunkte auf diese Komponenten ändern. Sofern diese Endpunkte aber fest in der *Cloud-Anwendung* implementiert sind, ist eine Reprovisionierung der Anwendung nicht möglich.

5.3.1. Fazit

Auch wenn es gelingt, einige Daten der Komponenten der *Cloud-Anwendung* abzurufen und zu serialisieren, ist die CloudFridge noch weit von einem vollautomatisierten System zur Reprovisionierung entfernt. Dies ist hauptsächlich darauf zurück zu führen, dass der *Cloud-Anbieter* hier nicht die benötigten Funktionen bereitstellt. Dennoch zeigt der Prototyp, dass es in beschränktem Umfang möglich ist, die vorhandenen *Cloud-Objekte* zu archivieren und wieder in Betrieb zu nehmen.

Das Problem, dass über das AWS-SDK nicht alle Informationen abgerufen werden können, lässt sich auf zweierlei Arten lösen:

1. Eine Möglichkeit ist, dass der *Cloud-Anbieter* seine Plattform erweitert und die entsprechenden Funktionen zum Abruf der Informationen bereitstellt. Dies ist aber nicht in jedem Fall möglich: Wie bereits erwähnt werden manche Komponenten an feste Endpunkte verknüpft, wobei diese bei der Erstellung teilweise zufällig generiert werden. `{"address": "meine-postgresql.cykixj0rlryc.eu-central-1.rds.amazonaws.com"}`

5. Entwicklung des Prototyps

Am Beispiel des Amazon RDS-Endpunkts ist zu sehen, dass dieser sich aus dem Namen der Instanz und daran angeschlossen einer zufälligen Zeichenfolge zusammensetzt. Um diese Komponente nun exakt so wieder in Betrieb nehmen zu können, müsste der *Cloud-Betreiber* dafür sorgen, dass jeder Endpunkt nur einmal vergeben wird und damit zum Zeitpunkt einer Reversionierung noch zur Verfügung steht.

Darüber hinaus ist es vermutlich nicht im Interesse des Anbieters, sein *AWS-SDK* zu erweitern. Wenn das Archivieren von Anwendungen verschiedener Cloud-Anbieter möglich wäre, könnten diese auch in einer anderen Cloud wieder in Betrieb genommen werden. Die archivierten Daten müssten dazu zwar an die geänderte Cloud-Architektur angepasst werden, aber grundsätzlich ist ein solches Vorgehen denkbar. Dann wäre es möglich durch ein automatisiertes Verfahren den *Vendor LockIn* zu umgehen, was definitiv nicht im Interesse des *Cloud-Anbieters* ist.

2. Die zweite Möglichkeit besteht darin, zu analysieren, welche Cloud-Objekte sich archivieren lassen. Aus diesen Komponenten können dann Patterns abgeleitet werden, wie sich nicht-archivierbare Komponenten dennoch in den Zustand bei der Archivierung zurückversetzen lassen. Beispielsweise stellt es sich derzeit so dar, dass sich die *Amazon SQS*-Komponente nur soweit archivieren lässt, dass die Nachrichten der Queue abgerufen werden können. Um darüber hinaus zu ermitteln, welche Komponenten bereits Nachrichten aus der Queue abgerufen haben, könnte hierzu ein *Logging- und Replay*-Verfahren genutzt werden. Hierzu dürfen die Komponenten jedoch nicht direkt auf die *Amazon SQS* zugreifen, sondern müssen hierzu den Umweg über eine *Logging-Infrastruktur* gehen, wie sie durch Abbildung 5.3 dargestellt wird.



Abbildung 5.3.: Amazon Lambda und Amazon SQS

Anschließend kann ein System wie *OpenTOSCA* die geplante Cloud-Architektur so umbauen, dass nur Komponenten und Patterns genutzt werden, die sich archivieren lassen. Dieser Ansatz muss allerdings noch in weiterführenden Arbeiten auf seine Machbarkeit geprüft werden.

6. Zusammenfassung und Ausblick

Zusammenfassung

Das Ziel dieser Arbeit war es, einen Prototyp für die automatisierte Archivierung von Anwendungen und deren Reversionierung zu entwickeln, um zukünftig Cloud-Anwendungen inklusive ihres Zustands „einfrieren“ und wieder „auftauen“ zu können. Hierzu sollte erforscht werden, ob sich ein solches Vorhaben mit den aktuellen Möglichkeiten überhaupt umsetzen lässt und im Erfolgsfall ein funktionaler Prototyp entstehen, auf dessen Grundlage weitergearbeitet werden kann.

Dafür wurden die erforderlichen Grundlagen recherchiert und im Rahmen dieser Arbeit dargelegt. Anschließend wurden, basierend auf den verwandten wissenschaftlichen Arbeiten, wichtige Lektionen gewonnen, die bei der Umsetzung der Prototyps zu beachten sind. Die Arbeiten wurden dazu zuerst einzeln analysiert und anschließend miteinander verglichen. Abschließend wurden die gewonnenen Informationen auf die CloudFridge übertragen und daraus ein Konzept für den Prototyp abgeleitet.

Aus diesem Konzept heraus entstand letztlich der lauffähige Prototyp, dessen Implementierungsdetails ebenfalls in dieser Arbeit festgehalten sind und damit als Dokumentation für weitere Arbeiten an der CloudFridge dienen. Die aus der Implementierung und den Tests gewonnenen Erkenntnisse wurden festgehalten und im Kontext bewertet.

Letztlich führten die Tests auf Grundlage der CloudFridge zu dem Ergebnis, dass die Archivierung und Reversionierung sich weiterhin problematisch darstellt, wenngleich die CloudFridge einen ersten Schritt in diese Richtung führt. Hauptgrund für die Probleme ist die Tatsache, dass die CloudFridge stark davon abhängig ist, welche Zugriffe vom Cloud-Dienstleister ermöglicht werden.

Ausblick

Weitere Arbeiten können die Ergebnisse dieser Arbeit nutzen, um zunächst das System zur Archivierung von Cloud-Systemen weiter auszubauen:

Aktuell stellt die Implementierung einen PoC dar, weswegen nicht alle möglichen *AWS*-Module archiviert werden können. Sofern weitere Module unterstützt werden sollen, kann der Prototyp um die entsprechenden Module erweitert werden. Ziel sollte dabei sein, dass *CloudFridge* alle *AWS*-Module unterstützt.

Neben der Erweiterung um *AWS*-Module kann der Prototyp auch dahingehend angepasst werden, die verschiedenen Cloud-Dienste anderer Anbieter zu unterstützen. Hierzu ist, wie in Kapitel 5 beschrieben, analog zum *AWS-Freezer* ein weiterer *Freezer* anzulegen und die *CloudFridge* um die entsprechenden *FreezableObjects* zu erweitern.

Das Archivierungsformat der *CloudFridge* kann überarbeitet werden. Derzeit wird hierfür das JSON-Format genutzt. Dies ist für Komponenten wie beispielsweise Amazon EC2¹ nicht optimal, weswegen hier alternative Formate getestet werden können.

Der Prototyp basiert, als PoC, auf einer möglichst einfachen Implementierung, welche derzeit keine nebenläufige Ausführung vorsieht. Durch Nutzung des *AWS-SDK 2.0* oder höher werden asynchrone Zugriffe auf die *AWS*-Dienste ermöglicht. Darüber hinaus ließen sich auch verschiedene Komponenten durchaus zeitgleich herunterfahren. Durch diese Anpassungen ist insgesamt eine signifikante Beschleunigung der *CloudFridge* zu erwarten.

Sofern die Module mehrerer Cloud-Anbieter in die *CloudFridge* integriert wurden, ist es denkbar, die Logik des „Auftauens“ so zu erweitern, dass Module, die aus der einen Cloud stammen, transformiert und bei einem anderen Cloud-Anbieter wieder in Betrieb genommen werden.

Wie bereits erwähnt, bietet es sich an, die *CloudFridge* in eine bestehende *Cloud-Orchestrierungs-Software* zu integrieren. Gerade *OpenTOSCA* bietet hier die Möglichkeit, im Rahmen von weiteren Bachelor- und Master-Arbeiten angepasst und erweitert zu werden. Dabei ist es jedoch empfehlenswert, die *CloudFridge* zunächst um weitere Module und Cloud-Anbieter zu erweitern. Der Vorteil von einer Integration in *OpenTOSCA* (o. ä.) ist, dass diese Systeme bereits über einen Deployment-Plan verfügen, aus dem sich ein Archivierungs-Plan generieren lässt.

¹Amazon Elastic Compute Cloud (Amazon EC2) - <https://aws.amazon.com/de/ec2>

Literaturverzeichnis

- [ALT] All Lines Technology. *Using a multi-cloud strategy to achieve digital transformation*. URL: <https://all-lines-tech.com/Modern-IT/Public-Private-Hybrid-Cloud> (zitiert auf S. 6).
- [AWS19a] Amazon Web Services Inc. *AWS Produkte & Services*. 2019. URL: <https://aws.amazon.com/de/products> (zitiert auf S. 6).
- [AWS19b] Amazon Web Services Inc. *AWS-Preise - Im Überblick*. 2019. URL: <https://aws.amazon.com/de/pricing> (zitiert auf S. 1).
- [AWS19c] Amazon Web Services Inc. *Informationen zu AWS*. 2019. URL: <https://aws.amazon.com/de/about-aws> (zitiert auf S. 6).
- [AWS19d] Amazon Web Services Inc.. *Interface AWSCredentialsProvider*. 2019. URL: <https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/auth/AWSCredentialsProvider.html> (zitiert auf S. 37).
- [AWS19e] Amazon Web Services Inc. *Verwalten der Zugriffsschlüssel für IAM-Benutzer - AWS Identity and Access Management*. 2019. URL: https://docs.aws.amazon.com/de_de/IAM/latest/UserGuide/id_credentials_access-keys.html (zitiert auf S. 6).
- [AWS19f] Amazon Web Services Inc. *Verwenden des SDK mit Apache Maven*. 2019. URL: https://docs.aws.amazon.com/de_de/sdk-for-java/v2/developer-guide/setup-project-maven.html (zitiert auf S. 35).
- [AWS19g] Amazon Web Services Inc. *Was ist Caching und wie funktioniert es?* 2019. URL: <https://aws.amazon.com/de/caching> (zitiert auf S. 9).
- [BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. „The OpenTOSCA Ecosystem – Concepts & Tools“. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016* (2016), S. 112–130. DOI: [10.5220/0007903201120130](https://doi.org/10.5220/0007903201120130) (zitiert auf S. 11).
- [Ble10] A. Blewitt. *Automatic Resource Management in Java*. Aug. 2010. URL: <https://www.infoq.com/news/2010/08/arm-blocks> (zitiert auf S. 44).
- [cA18] chrissikraus, S. Augsten. „Definition „Ansible“. Was ist Ansible?“ In: *Dev Insider* (Juni 2018). URL: <https://www.dev-insider.de/was-ist-ansible-a-724411> (zitiert auf S. 12).
- [Cey18] Ö. Ceylan. *Continuous integration and deployment for your enterprise hybrid mobile apps*. März 2018. URL: <https://itnext.io/continuous-integration-and-deployment-for-your-enterprise-hybrid-mobile-apps-51a57501abf0> (zitiert auf S. 14).

- [DA17] T. Drilling, S. Augsten. „AWS CloudFormation, Teil 3. CF-Template für AWS-Ressourcen erstellen“. In: *Dev Insider* (Nov. 2017). URL: <https://www.dev-insider.de/cf-template-fuer-aws-ressourcen-erstellen-a-659367> (zitiert auf S. 13).
- [DA18] T. Drilling, S. Augsten. „AWS OpsWorks for Chef Automate, Teil 3. Chef-Cookbooks im Einsatz“. In: *Dev Insider* (Dez. 2018). URL: <https://www.dev-insider.de/chef-cookbooks-im-einsatz-a-778421> (zitiert auf S. 13).
- [EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. „Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications“. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, S. 22–27 (zitiert auf S. 8).
- [FGLI15] M. Forsman, A. Glad, L. Lundberg, D. Ilie. „Algorithms for automated live migration of virtual machines“. In: *Journal of Systems and Software* 101 (2015), S. 110–126. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2014.11.044>. URL: <http://www.sciencedirect.com/science/article/pii/S0164121214002751> (zitiert auf S. 15, 16, 19, 20).
- [FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns*. Springer, 2014. ISBN: 978-3-7091-1568-8. DOI: [10.1007/978-3-7091-1568-8](https://doi.org/10.1007/978-3-7091-1568-8). URL: <https://www.cloudcomputingpatterns.org> (zitiert auf S. 5).
- [GWL19a] P. D. R. Lackes. *Gabler Wirtschaftslexikon - virtuelle Maschine*. Feb. 2019. URL: <https://wirtschaftslexikon.gabler.de/definition/virtuelle-maschine-48523> (zitiert auf S. 15).
- [GWL19b] F. Leymann, C. Fehling. *Gabler Wirtschaftslexikon - Provisionierung*. Feb. 2019. URL: <https://wirtschaftslexikon.gabler.de/definition/provisionierung-53362/version-276455> (zitiert auf S. 7).
- [HA18] HJL, S. Augsten. „Definition „Chef (Software)“. Was ist Chef?“ In: *Dev Insider* (Juli 2018). URL: <https://www.dev-insider.de/was-ist-chef-a-728620> (zitiert auf S. 13).
- [Ham14] E. Hammer. *The Fallacy of Tiny Modules*. Mai 2014. URL: <https://hueniverse.com/the-fallacy-of-tiny-modules-920f9a14cc43> (zitiert auf S. 1).
- [HBKL19] L. Harzenetter, U. Breitenbücher, K. Képes, F. Leymann. „Freezing and Defrosting Cloud Applications. Automated Saving and Restoring of Running Applications“. 2019 (zitiert auf S. 8, 25, 26).
- [HF] J. Humble, D. Farley. *Continuous Delivery. Reliable Software Releases through Build, Test, and Deployment Automation*. URL: <https://martinfowler.com/books/continuousDelivery.html> (zitiert auf S. 14).
- [HKF18] M. Horii, Y. Kojima, K. Fukuda. „Stateful process migration for edge computing applications“. In: *2018 IEEE Wireless Communications and Networking Conference (WCNC)*. Apr. 2018, S. 1–6. DOI: [10.1109/WCNC.2018.8377072](https://doi.org/10.1109/WCNC.2018.8377072) (zitiert auf S. 21).

- [HPS+15] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, V. Young. „Mobile edge computing a key technology towards 5g“. In: (Sep. 2015). URL: https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf (zitiert auf S. 21).
- [Hug15] G. Hughson. *Microservice Mistakes – Complexity as a Service*. 2015. URL: <https://iasaglobal.org/microservice-mistakes-complexity-as-a-service> (zitiert auf S. 1).
- [Hum19] G. Hummel. *What is Ansible?* März 2019. URL: <https://cloudacademy.com/blog/what-is-ansible> (zitiert auf S. 12).
- [HW03] G. Hohpe, B. Woolf. *Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*. 1. Auflage. Addison-Wesley Professional, 2003. ISBN: 978-0-321-20068-6. URL: <https://www.enterpriseintegrationpatterns.com> (zitiert auf S. 9).
- [IBM14] D. Beaumont. *How to explain vertical and horizontal scaling in the cloud*. Apr. 2014. URL: <https://www.ibm.com/blogs/cloud-computing/2014/04/09/explain-vertical-horizontal-scaling-cloud> (zitiert auf S. 7).
- [ITW07] DATACOM Buchverlag GmbH. Aug. 2007. URL: <https://www.itwissen.info/Virtuelle-Maschine-virtual-machine-VM.html> (zitiert auf S. 15).
- [KBM14] D. V. Klein, D. M. Betser, M. G. Monroe. „Making ‚Push On Green‘ a Reality“. In: *;login: - The usenix magazine* 39.5 (Okt. 2014) (zitiert auf S. 14, 15).
- [Kic19] A. Kicherer. „Modellierung und Deployment von cyber-physischen Systemen basierend auf TOSCA“. Masterarbeit. Universität Stuttgart, 2019 (zitiert auf S. 8, 11).
- [LF14] J. Lewis, M. Fowler. *Microservices. a definition of this new architectural term*. März 2014. URL: <https://martinfowler.com/articles/microservices.html> (zitiert auf S. 1, 7, 15).
- [LK19] S. Luber, F. Karlstetter. „Definition Infrastructure as Code (IaC) auf Basis von AWS. Was ist AWS CloudFormation?“ In: *CloudComputing Insider* (Aug. 2019). URL: <https://www.cloudcomputing-insider.de/was-ist-aws-cloudformation-a-857705> (zitiert auf S. 13).
- [Lou12] M. Loukides. *What is DevOps?* Juni 2012. URL: <http://radar.oreilly.com/2012/06/what-is-devops.html> (zitiert auf S. 15).
- [Mat17] G. Matheis. *Hauptseminar: Cloud Computing. Security and Privacy*. Hauptseminar. Universität Stuttgart, Aug. 2017 (zitiert auf S. 5, 8, 15, 25).
- [MG14] V. Medina, J. M. García. „A Survey of Migration Mechanisms of Virtual Machines“. In: *ACM Comput. Surv.* 46.3 (Jan. 2014), 30:1–30:33. ISSN: 0360-0300. DOI: 10.1145/2492705. URL: <https://doi.acm.org/10.1145/2492705> (zitiert auf S. 17, 29, 30).
- [Mor17] P.-S. Morawietz. *So sichern Sie Ihre Daten in die Cloud*. März 2017. URL: https://www.pcwelt.de/ratgeber/So_sichern_Sie_Ihre_Daten_in_die_Cloud-Daten-Backups-7959653.html (zitiert auf S. 5).
- [MS] Microsoft Corporation. *Was ist die Cloud?* URL: <https://azure.microsoft.com/de-de/overview/what-is-the-cloud> (zitiert auf S. 5).

- [MSS16] A. Mazrekaj, I. Shabani, B. Sejdiu. „Pricing Schemes in Cloud Computing: An Overview“. In: *International Journal of Advanced Computer Science and Applications* 7 (Feb. 2016). doi: [10.14569/IJACSA.2016.070211](https://doi.org/10.14569/IJACSA.2016.070211) (zitiert auf S. 1).
- [NHS+18] C. Neufeind, L. Harzenetter, P. Schildkamp, U. Breitenbücher, B. Mathiak, J. Barzenand, F. Leymann. „The SustainLife Project – Living Systems in Digital Humanities“. In: *Papers From the 12th Advanced Summer School of Service-Oriented Computing (SummerSOC2018)*. IBM Research Division, Okt. 2018, S. 101–112 (zitiert auf S. 1).
- [Ora10] Oracle Corporation. *Message-Oriented Middleware (MOM)*. 2010. URL: <https://docs.oracle.com/cd/E19340-01/820-6424/eraaq/index.html> (zitiert auf S. 9, 10).
- [OSSN02] S. Osman, D. Subhraveti, G. Su, J. Nieh. „The Design and Implementation of Zap: A System for Migrating Computing Environments“. In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dez. 2002), S. 361–376. ISSN: 0163-5980. doi: [10.1145/844128.844162](https://doi.org/10.1145/844128.844162). URL: https://www.cs.cmu.edu/~sosman/publications/osdi2002/osdi2002_zap.pdf (zitiert auf S. 31).
- [QCW15] R. Qasha, J. Cala, P. Watson. „Towards Automated Workflow Deployment in the Cloud Using TOSCA“. In: *2015 IEEE 8th International Conference on Cloud Computing*. Juli 2015, S. 1037–1040. doi: [10.1109/CLOUD.2015.146](https://doi.org/10.1109/CLOUD.2015.146) (zitiert auf S. 1).
- [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)*. Nov. 2015, S. 85–89. doi: [10.1109/CCEM.2015.29](https://doi.org/10.1109/CCEM.2015.29) (zitiert auf S. 7).
- [SR18] M. Schindler, RightScale. *Cloud-Computing: Azure und Google Cloud wachsen schneller als AWS*. 2018. URL: <https://www.zdnet.de/88325995/cloud-computing-azure-und-google-cloud-wachsen-schneller-als-aws> (zitiert auf S. 35).
- [Sta11] J. Staten. *The 3 stages of cloud economics*. Mai 2011. URL: <https://www.cio.co.uk/cloud-computing/the-3-stages-of-cloud-economics-3430516> (zitiert auf S. 1).
- [TIOBE19] TIOBE Software BV. *TIOBE Index for August 2019*. 2019. URL: <https://www.tiobe.com/tiobe-index> (zitiert auf S. 35).
- [TOSCA-v1.0] OASIS Standard. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Nov. 2013. URL: <https://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf> (zitiert auf S. 8).
- [Völ18] C. Völker. „Suitability of serverless computing approaches“. Masterarbeit. Universität Stuttgart, 2018 (zitiert auf S. 15).
- [Wag12] S. Wagner. *The Great Hope of Cloud Economics and the Over-provisioning Epidemic*. März 2012. URL: https://www.cloudyn.com/wp-content/uploads/2013/06/The_Great_Hope_of_Cloud_Economics.pdf (zitiert auf S. 1).

- [WBK+18] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, V. Yussupov. „Modeling and Automated Deployment of Serverless Applications Using TOSCA“. In: *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*. Nov. 2018, S. 73–80. DOI: [10.1109/SOCA.2018.00017](https://doi.org/10.1109/SOCA.2018.00017) (zitiert auf S. 1, 8).
- [WRSM11] T. Wood, K. K. Ramakrishnan, P. Shenoy, J. van der Merwe. „CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines“. In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '11. Newport Beach, California, USA: ACM, 2011, S. 121–132. ISBN: 978-1-4503-0687-4. DOI: [10.1145/1952682.1952699](https://doi.org/10.1145/1952682.1952699). URL: <http://doi.acm.org/10.1145/1952682.1952699> (zitiert auf S. 18).
- [ZBF+17] M. Zimmermann, F. W. Baumann, M. Falkenthal, F. Leymann, U. Odefey. „Automating the Provisioning and Integration of Analytics Tools with Data Resources in Industrial Environments Using OpenTOSCA“. In: *2017 IEEE 21st International Enterprise Distributed Object Computing Workshop (EDOCW)*. Okt. 2017, S. 3–7. DOI: [10.1109/EDOCW.2017.10](https://doi.org/10.1109/EDOCW.2017.10) (zitiert auf S. 1).

Alle URLs wurden zuletzt am 31.08.2019 geprüft.

A. Anhang

A.1. Bildrechte

Sämtliche von mir, Franziska Amend, erstellten Grafiken sind urheberrechtlich geschützt. Sie dürfen ohne eine vorherige schriftliche Genehmigung weder ganz noch auszugsweise kopiert, verändert, vervielfältigt oder veröffentlicht werden.

In dieser Arbeit handelt es sich dabei um die folgenden gelisteten Grafiken:

1. Abbildung 3.3: „Schematische Darstellung von SnowFlock“ auf Seite 24
2. Abbildung 5.1: „CloudFridge Logo“ auf Seite 35

Im selben Atemzuge erteile ich hiermit Herrn Gerd Matheis die schriftliche Genehmigung diese Grafiken in seinem Sinne, insbesondere im Rahmen dieser Arbeit, zu verwenden. Jedoch ist es ihm nicht gestattet dritten Personen diese Genehmigung zu erteilen.

Stuttgart, 04.09.2019,

Ort, Datum, Unterschrift

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, 04.09.2019,

Ort, Datum, Unterschrift