

Institut für Architektur von Anwendungssystemen

Universität Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Masterarbeit

**Serverless Applikationen in  
Multi-Cloud-Umgebungen:  
Architektur und Design  
Eventbasierter Kommunikation**

Daniel Hagg

**Studiengang:** Softwaretechnik

**Prüfer/in:** Prof. Dr. Dr. h. c. Frank Leymann

**Betreuer/in:** Michael Wurster, M.Sc.

**Beginn am:** 14. Dezember 2018

**Beendet am:** 14. Juni 2019



## Kurzfassung

Kurze Entwicklungszeiten, nahezu grenzenlose Skalierung und den Betrieb eines Services ohne Verwaltungs- und Managementaufwand der zugrundeliegenden Ressourcen für den Entwickler. All das verspricht der Ansatz des Serverless Computing. Aufgrund der starken Abhängigkeit des entwickelten Services zum Cloud-Provider besteht dabei allerdings stets die Gefahr des Vendor Lock-Ins. Dieser Vendor Lock-In wird beispielsweise dann erkennbar, wenn im Anwendungsfall die Nutzung mehrere Cloud-Umgebungen in einer Multi-Cloud vorgesehen ist. Eventbasierte Kommunikation, die den Serverless Applikationen zugrunde liegt, lässt sich nicht einfach auf weitere Cloud-Umgebungen erweitern. Dazu fehlen Standards und Schnittstellen.

In dieser Arbeit wird das Problem der Eventbasierten Kommunikation in Multi-Cloud-Umgebungen analysiert. Aus dieser Analyse der Anforderungen werden Konzepte entwickelt, deren Realisierbarkeit im Folgenden anhand von Prototypen gezeigt wird. Dabei zeigt sich, dass die benötigten Lösungsansätze auf die unterschiedlichen Anwendungsfälle zugeschnitten werden müssen. Insbesondere die Prototypen zeigen Probleme auf, die trotz der Nutzung von offenen Standards auftreten, wie einer HTTP-Schnittstelle und der CloudEvents Spezifikation. Dabei sticht beispielsweise die Authentifizierung der Umgebungen untereinander heraus, da dafür kein Standardvorgehen zwischen den Providern besteht. In dieser Arbeit wird Eventbasierte Kommunikation in Multi-Cloud-Umgebungen ermöglicht.

## **Abstract**

Rapid development, nearly infinite scalability and the possibility of operating a service without administration and management of the underlying resources for the developer. This is promised by Serverless Computing. Caused by the strong dependence between the developed service and the cloud provider there is a risk of a vendor lock-in. This vendor lock-in arises with examining a multi-cloud use-case. In such a use-case two or more clouds are connected to serve one application or service. Serverless applications communicate event-based. This event-based communication can not be established between two or more clouds, caused by missing specifications and interfaces.

This paper analysis the problem of event-based communication in multi-cloud environments. Concepts are being developed based on the analysis of requirements. The feasibility of these concepts is demonstrated by prototypes. They point out that the required solutions must be adjusted to different use-cases. The prototypes show problems that occur despite the use of open standards, such as HTTP interfaces and the CloudEvents specification. Authentication between different environments, for example, is difficult as there is no common specification establishing such an authentication. In this paper event-based communication in multi-cloud environments is enabled.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>11</b>
1.1	Motivation . . . . .	12
1.2	Ziel der Arbeit . . . . .	13
<b>2</b>	<b>Grundlagen und Stand der Technik</b>	<b>15</b>
2.1	Serverless Computing . . . . .	15
2.2	Serverless Framework . . . . .	20
2.3	CloudEvents . . . . .	21
2.4	Event Gateways . . . . .	22
<b>3</b>	<b>Anforderungen</b>	<b>25</b>
3.1	Referenzszenario . . . . .	25
3.2	Funktionale Anforderungen . . . . .	30
3.3	Nichtfunktionale Anforderungen . . . . .	32
<b>4</b>	<b>Konzeption</b>	<b>35</b>
4.1	Nachrichtenformat . . . . .	35
4.2	Nachrichtenkanal . . . . .	36
<b>5</b>	<b>Prototypische Implementierung</b>	<b>47</b>
5.1	Direkter Aufruf . . . . .	47
5.2	Zentrales Gateway . . . . .	54
5.3	Dezentrale Gateways . . . . .	59
5.4	Hierarchische Struktur von Gateways . . . . .	65
<b>6</b>	<b>Fazit und Ausblick</b>	<b>71</b>
	<b>Literaturverzeichnis</b>	<b>75</b>



# Abbildungsverzeichnis

2.1	Serverless Plattform Architektur nach [BCC+17, S. 5] . . . . .	16
3.1	Mockup der Benutzeroberfläche des Referenzszenarios . . . . .	26
3.2	Architektur einer Eigenentwicklung in einer Serverless Multi-Cloud-Umgebung . . . . .	26
3.3	Sequenzdiagramm des Upload eines Fotos im Referenzszenario . . . . .	27
3.4	Umsetzung von Projekten in Unternehmen mit Hilfe der Public Cloud. Auszug aus [Rig19] . . . . .	28
4.1	Model eines Multi-Cloud Events . . . . .	35
4.2	Direkter Aufruf des Empfängers über eine providerspezifische Schnittstelle . . . . .	37
4.3	Eventbasierte Kommunikation mit einem zentralen Gateway . . . . .	39
4.4	Eventbasierte Kommunikation mit dezentralen Gateways . . . . .	42
4.5	Dezentrale Gateways mit einem zentralen Gateway . . . . .	44
4.6	Dezentrale Gateways mit einem Hauptgateway als zentrale Verwaltung . . . . .	46
5.1	Prototyp Eventbasierter Kommunikation in einer Multi-Cloud-Umgebung durch direkte Aufrufe. . . . .	48
5.2	Prototyp einer Umgebung mit zentralem Gateway. . . . .	54
5.3	Prototyp einer Umgebung mit dezentralen Gateways . . . . .	60
5.4	Prototyp einer Umgebung mit hierarchischen Gateways . . . . .	66





## Verzeichnis der Listings

2.1	Ausschnitt einer Konfigurationsdatei des Serverless Framework nach [Serd] . . . .	21
2.2	Beispiel eines Events nach der CloudEvents Spezifikation im JSON Format aus [CNCd]. . . . .	22
5.1	Konfiguration unterschiedlicher Events auf einem S3 Speicher . . . . .	49
5.2	Gekürzter Mitschnitt eines von einem S3 Speicher ausgelösten Events . . . . .	50
5.3	Umwandlung des S3 Events in das CloudEvents Format . . . . .	50
5.4	Filterung der Events im Prototyp . . . . .	50
5.5	Mitschnitt des von NATS verwendeten Protokolls . . . . .	56
5.6	Gekürztes Beispiel eines SNS Events im JSON Format nach [Amab] . . . . .	61
5.7	Konfiguration zur Filterung von intern aufgetretenen Events im Serverless Framework.	61



# 1 Einleitung

Serverless Computing ermöglicht es Applikationen und Services zu erstellen und zu betreiben, ohne sich um die Provisionierung, Skalierung, Verfügbarkeit oder andere Themen der Serververwaltung kümmern zu müssen. Anders als der Name *Serverless* (dt. *Serverlos*) vermuten lässt, liegt dies nicht darin begründet, dass keine Server mehr vorhanden sind. Beim Serverless Computing werden sämtliche Verwaltungs- und Management-Aufgaben der Umgebung vom Cloud-Provider durchgeführt. Entwickler können vorhandene Komponenten nutzen und ihre selbst entwickelten Softwarekomponenten einfach in die vorhandene Umgebung einfügen und betreiben, ohne sich beispielsweise mit Skalierung oder *Quality-of-Service (QoS)* beschäftigen zu müssen. Dabei bezahlen sie nur für die tatsächlich verbrauchten Ressourcen. Ein Komponente, die nicht aufgerufen oder anderweitig genutzt wird, verursacht keine Kosten. Mit Lambda veröffentlichte Amazon im Jahre 2014 die erste FaaS-Plattform und verhalf damit dem Serverless Computing zu Popularität [BCC+17, S. 4]. Aufgrund der starken Abstraktionsebene wird Serverless Computing sogar als nächster Evolutionsschritt des Cloud-Computing bezeichnet [RC17, S. 6].

Diese strikte Auslagerung der Serververwaltung an den Cloud-Provider kann nur durch die Nutzung verschiedener Paradigmen der Softwarearchitektur erzielt werden. Eines dieser Paradigmen wird als *Function-as-a-Service (FaaS)* bezeichnet. Dabei werden einzelne Aufgaben einer Applikation in vielen kleinen und unabhängigen Funktionen implementiert, die von der Umgebung bei Bedarf aufgerufen werden. Entwickler laden diese Funktionen in die entsprechende, vom Cloud-Provider bereitgestellte Umgebung. Für diesen ist es möglich die einzelnen Funktionen bedarfsgesteuert verfügbar zu machen und zu skalieren.

Die eventbasierte Verarbeitung ist beim Serverless Computing ebenfalls ein wichtiges Paradigma, das die Auftrennung der Applikation in viele kleine Funktionen unterstützt. Durch Events wird das Eintreten eines Ereignisses signalisiert und anschließend eine oder mehrere Aktionen ausgeführt, wodurch eine vollständige Entkopplung der einzelnen Komponenten stattfinden kann. Der Auslöser eines Events benötigt keinerlei Informationen über die durch das Event ausgelösten Aktionen, die wiederum nicht wissen müssen, was zur Auslösung des Events geführt hat.

Events können dabei von unzähligen Aktionen ausgelöst werden, beispielsweise durch das Erstellen eines Datenbankeintrags, das Eintreffen eines HTTP-Requests oder durch den applikationsspezifischen Eintritt eines Ereignisses, wie einem neuen Highscore in einem Spiel. Jede Zustandsänderung und jedes andere Ereignis innerhalb des Systems kann als Event interpretiert und realisiert werden.

### 1.1 Motivation

Eines der größten Probleme des Serverless Computing ist, wie nahezu bei jeder Cloud-Technik, der sogenannte Vendor Lock-In [Sti18, S. 10]. Dabei handelt es sich um die Abhängigkeit der entwickelten Software vom gewählten Cloud-Provider. Jede Software, die in einer bestimmten Serverless Umgebung läuft, muss speziell an diese angepasst werden und ist daraufhin lediglich in dieser Umgebung fähig ausgeführt zu werden.

Das Problem des Vendor Lock-In wird auch im Bezug auf die Multi-Cloud Funktionalität des Serverless Computing deutlich. Bei einer Multi-Cloud werden zwei oder mehr Cloud-Umgebungen zusammengeschlossen, um eine Applikation oder einen Service gemeinsam in diesen Umgebungen betreiben zu können. Dies kann aus den unterschiedlichsten Gründen und auf verschiedenste Weise durchgeführt werden.

Einer der meist verwendeten Arten einer Multi-Cloud ist die Hybrid Cloud [Pet13, S. 1]. Bei ihr arbeiten zwei Cloud-Umgebungen mit unterschiedlichen Deployment Modellen zusammen, um einen Service zu betreiben. Beispielsweise können dies eine Public und eine Private Cloud sein. Anwendungsfälle, die zur Nutzung einer solchen Cloud führen, können sich stark unterscheiden. Es kann sinnvoll sein eine Applikation primär in einer Private Cloud zu betreiben, und lediglich bei Lastspitzen auf Ressourcen einer Public Cloud auszuweichen [Pet13, S. 1]. In einem anderen Anwendungsfall wird primär die Public Cloud genutzt und lediglich bei der Verarbeitung sensibler Daten auf die Private Cloud zurückgegriffen. Für Serverlose Applikationen bringt der Betrieb in einer Multi-Cloud-Umgebung aufgrund der hohen Abhängigkeit vom Provider große Herausforderungen mit sich.

Der Applikation muss es möglich sein Events aus einer Umgebung abzugreifen, in andere Cloud-Umgebungen weiterzuleiten und dort auf diese Events zu reagieren. Um dies zu verwirklichen muss es allen beteiligten Komponenten möglich sein Informationen des Events lesen und verarbeiten zu können. Für die Weiterleitung oder die endgültige Ausführung von Aktionen ist dies beispielsweise zwingend erforderlich. Dazu eignet sich ein einheitliches Event-Format, wie es die *Arbeitsgruppe Serverless der Cloud Native Computing Foundation (CNCF)* mit der *CloudEvents* Spezifikation anstrebt. Ihr Ziel ist es dabei Interoperabilität zwischen Event-Systemen herzustellen, bei denen Sender und Empfänger von Events unabhängig voneinander arbeiten und entwickelt werden können [CNCb]. Eine solche Vereinheitlichung hilft auch einer späteren Erweiterung auf zusätzliche Cloud-Umgebungen und damit der Providerunabhängigkeit. Anstatt mit jeder Umgebung über unterschiedliche Schnittstellen zu kommunizieren, kann die Kommunikation über einen identischen Kanal stattfinden.

Gleichzeitig muss es einem Entwickler möglich sein, sich auf die Vorzüge des Serverless Computing vollumfänglich verlassen zu können. Skalierbarkeit und Verfügbarkeit nehmen hierbei eine besondere Rolle ein. Genauso muss es dem Entwickler möglich sein, sich auf die Zustellung von Events verlassen zu können. Besonders durch die erhöhte Fehleranfälligkeit des Versands von Events über Cloud-Umgebungen hinweg, muss das Softwaresystem mit Nichterreichbarkeit anderer Umgebungen umgehen können.

## 1.2 Ziel der Arbeit

Im Rahmen dieser Arbeit soll Serverless Computing im Zusammenspiel mehrerer Cloud-Umgebungen im Hinblick auf Eventbasierte Kommunikation untersucht werden. Dazu werden zuerst die benötigten Grundlagen vermittelt, um anschließend mit dem Stand der Technik bisherige Lösungsansätze und Konzepte zu analysieren. Um die Anforderungen an ein Softwaresystem zur Eventbasierten Kommunikation in Multi-Cloud-Umgebungen zu erfassen und definieren zu können, soll ein Use-Case herausgearbeitet und analysiert werden. Dieser Use-Case stellt dabei keineswegs das einzige Einsatzszenario des betrachteten Softwaresystems dar.

Daher werden in dieser Arbeit verschiedene Konzepte entwickelt, die Eventbasierte Kommunikation von serverlosen Anwendungen über die Grenzen verschiedener Cloud-Umgebungen hinweg ermöglichen. Diese Konzepte werden analysiert und anschließend in prototypischen Implementierungen realisiert. Anhand dieser Prototypen wird die Umsetzbarkeit der entwickelten Konzepte demonstriert. Ebenso sollen sie auf die Erfüllung der definierten Anforderungen und auf ihre bestehenden Einschränkungen untersucht werden.

Durch diese Arbeit erlangt der Leser das Verständnis für unterschiedliche Lösungsansätze, die zur Verwirklichung Eventbasierter Kommunikation in Multi-Cloud-Umgebungen zur Verfügung stehen. Des Weiteren erfährt er, wie diese Ansätze implementiert werden können und kann daraus folgern, welche Ansätze für welchen Anwendungsfälle geeignet erscheinen.



## 2 Grundlagen und Stand der Technik

Dieses Kapitel vermittelt die benötigten Grundlagen, damit dem Leser eine Einordnung der folgenden Arbeit möglich ist. Dazu wird zuerst auf das Serverless Computing eingegangen, wobei insbesondere sowohl Vor- als auch bestehende Nachteile, beziehungsweise Herausforderungen beschrieben werden. Anschließend wird der Stand der Technik mit themenbezogenen Grundlagen dargestellt, um auf diesen in den folgenden Kapitel aufzubauen.

### 2.1 Serverless Computing

Serverless Computing ist eine Technologie im Bereich des Cloud-Computing, die es einem Entwickler ermöglicht Programmcode zu entwickeln, ihn in die Cloud zu laden und auszuführen, ohne sich um die darunterliegende Infrastruktur oder die Wartung der Umgebung zu kümmern [Sti18, S. 1]. Sämtliche dieser Aufgaben werden vom Cloud-Provider übernommen. Eine Serverlose Applikation beinhaltet, laut Baldini et al., nur noch einzelne, auseinandergenommene Funktionen [BCC+17, S. 3]. Die einzelnen Funktionen sind zustandslos und können somit vom Cloud-Provider beliebig instanziiert, ausgeführt und wieder beendet werden. Eine Kenntnis darüber, welche Funktionalität die einzelnen Funktionen bieten, benötigt er nicht. Aufgrund des aktuellen Bedarfs kann er Funktionen hoch- beziehungsweise runterskalieren.

Das Prinzip hinter diesen Funktionen wird auch *Function-as-a-Service (FaaS)* genannt. Eine Funktion ist dabei nicht vergleichbar mit einem Container, der durchgängig läuft, denn sie wird lediglich dann ausgeführt, wenn ein Aufruf der Funktion stattfindet. Dies spiegelt sich auch in der Preisgestaltung wieder. Für Funktionen wird in der Regel die tatsächliche Ausführungszeit berechnet. Wenn keine Anforderung der Funktion stattfindet, findet auch keine Abrechnung statt.

Aufgrund dieser Funktionsweise benötigen Applikationen, die als serverlose Applikationen entwickelt werden, einen eigenen Architekturstil. Diese Eventbasierte Architektur beinhaltet neben den einzelnen Funktionen eine weitere Komponente, die sogenannten Events. Events verbinden die einzelnen Funktionen untereinander und wiederum mit weiteren Komponenten der Umgebung. Diese weiteren Komponenten können beispielsweise Datenbanken, Datenspeicher, HTTP-Gateways oder auch Sensoren in einem *Internet-of-Things* Netzwerk sein. Funktionen reagieren auf Events und werden so ausgeführt. Diese Art der Kommunikation wird *Eventbasierte Kommunikation* genannt. Für die erfolgreiche Kommunikation, die dafür benötigte Infrastruktur und deren Management ist der Cloud-Provider verantwortlich.

### 2.1.1 Historie

Serverless Computing in seiner heutigen Form wurde erstmalig von Amazon auf der hauseigenen Entwicklerkonferenz re:Invent im Jahr 2014 der Öffentlichkeit vorgestellt [BCC+17, S. 4]. Mit der Veröffentlichung der Plattform Lambda [Ama19] in den *Amazon Web Services (AWS)* war es Entwicklern möglich eigene serverlose Applikationen zu betreiben. Kurze Zeit später, im Jahr 2016, drangen einige Mitbewerber Amazons mit verschiedenen eigenen Lösungen ebenfalls auf den Markt. Hierzu gehörte Google mit den Cloud Functions [Goo19] und Microsoft mit den Azure Functions [Mic19]. Neben den Public Cloud-Providern erkannten auch Anbieter von privat gehosteten Cloud-Umgebungen den Trend. Dies wurde in Applikationen wie Apache OpenWhisk [Ope19], OpenFaaS [Opeb] und Kubeless [Kuba] sichtbar.

In ihrem Buch *Research Advances in Cloud Computing* beschreiben Baldini et al. Serverless nicht als einen komplett neuen Ansatz, sondern als konsequente Weiterentwicklung aus dem Prinzip der stetig steigenden Anzahl an Abstraktionsebenen einer Applikation [BCC+17, S. 4]. Diese führen zu leichtgewichtigeren Applikationseinheiten. Leichtgewichtig steht für sie, nicht lediglich für den Verbrauch von Ressourcen, sondern auch für die Kosteneffektivität, und die Geschwindigkeit, in der Applikationen entwickelt werden können.

### 2.1.2 Funktionsweise

Anders als der Name *Serverless* (dt. *Serverlos*) impliziert, werden auch bei dieser Technologie selbstverständlich Server benötigt. Da diese aber vollständig vom Cloud-Provider verwaltet werden spricht Stigler in ihrem Buch *Beginning Serverless Computing* bei FaaS von einem Service, der sich von der Menge der vom Provider bereitgestellten Mittel zwischen *Platform-as-a-Service (PaaS)* und *Software-as-a-Service (SaaS)* befindet [Sti18, S. 2 f.] Während beim PaaS lediglich die Plattform zur Verfügung gestellt wird und diese bei Bedarf noch skaliert, konfiguriert oder angepasst werden muss, übernimmt beim FaaS all diese Aufgaben der Provider. Beim SaaS wird die komplette Applikation vom Provider bereitgestellt.

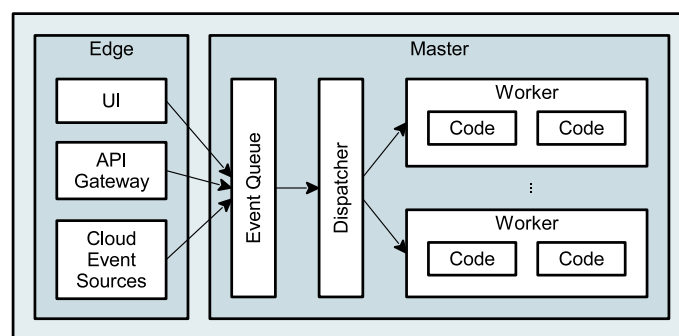


Abbildung 2.1: Serverless Plattform Architektur nach [BCC+17, S. 5]

Die grundlegende Architektur einer Serverless Plattform vergleichen Baldini et al. mit der eines eventverarbeitenden Systems [BCC+17, S. 5]. In Abbildung 2.1 wird der prinzipielle Aufbau eines solchen Systems dargestellt. Events, die von verschiedenen Quellen ausgelöst werden, werden in



eine Warteschlange geleitet. Von dort werden sie vom Dispatcher nacheinander entnommen und einem Worker zugeteilt, der den Funktionscode ausführt, der als Aktion auf das entsprechende Event konfiguriert wurde.

Etzion und Niblett differenzieren zwischen Eventbasierter Kommunikation und sogenannten *Request-Response Interaktionen* [EN10, S. 31 ff.]. Eine solche Request-Response Interaktion laufe in der Regel synchron ab. Das bedeutet, dass der Anfragende eine direkte Antwort erwartet und somit blockiert, bis die Antwort eingetroffen ist. Bei einem Request handelt es sich um eine direkte Aufforderung an den Empfänger etwas zu tun.

Bei einem Event sieht das anders aus. Ein Event ist ein Hinweis darauf, dass etwas passiert ist. Zur Unterscheidung beschreiben Etzion und Niblett ein Beispiel aus dem echten Leben [EN10, S. 33]. Beim Kauf eines Flugtickets löst der Kunde eine Request-Response Interaktion aus, sobald er die verfügbaren Flüge an einem bestimmten Tag anfragt. Das Abheben des gewünschten Flugzeugs beim Start hat dagegen den Charakter eines Events. Anstatt eines Anfragenden, kann daher bei der Eventbasierten Kommunikation von einem Event-Produzenten oder einer Event-Quelle gesprochen werden. Der Antwortende wird zu einem Empfänger.

In ihrem Artikel zum Thema *Event-driven Architecture overview* nennt Michelson die extrem lose Kopplung als eine der wichtigsten Eigenschaften einer Eventbasierten Architektur [Mic06, S. 3]. Diese ergibt sich aus dem Fakt, dass eine Quelle weder etwas über die Empfänger, noch über die daraus resultierenden Aktionen weiß. Eine Quelle erzeugt lediglich das Event und gibt es bekannt. Nach erfolgreicher Bekanntmachung sind alle weiteren Schritte für die Quelle irrelevant und nicht durchschaubar.

### 2.1.3 Vorteile

Stigler nennt in ihrem Buch fünf Gründe für die Nutzung von Serverless Computing [Sti18, S. 6 f.]. Der erste und ausschlaggebendste Grund als Entwickler auf eine serverlose Plattform zu setzen, ist die **beschleunigte Entwicklungs- und Deploymentzeit**. Da sich der Entwickler voll auf die Entwicklung der eigentlichen Applikation konzentrieren kann und keinerlei Aufwände in die Bereitstellung und den Betrieb der Umgebung stecken muss, wird die Zeit der Entwicklung minimiert, ebenso wie die Zeit in der ein Deployment durchgeführt wird. Dies ergibt völlig neue Möglichkeiten für Unternehmen, um Anwendungen schneller auf den Markt zu bringen und damit der Konkurrenz voraus zu sein.

Durch die fehlende Verantwortung des Entwicklers auf die Verwaltung der genutzten Ressourcen ergibt sich auch die **Einfachheit der Nutzung**. Insbesondere durch den eventbasierten Ansatz können Entwickler schnell und einfach Daten- und Informationsflüsse konfigurieren, ohne sich mit der darunterliegenden Technik im Detail auskennen zu müssen. Stigler nennt hier als Beispiel eine Applikation, die auf das von Amazon bereitgestellte API Gateway setzt [Sti18, S. 6]. Dieses kümmert sich um die komplette Logik des Betriebs des HTTP-Servers und der Verwaltung der Verbindungen. Der Entwickler muss sich lediglich um die Verarbeitung des aufgetretenen Events innerhalb einer Lambda Funktion kümmern und kann sich so besser auf die Businesslogik konzentrieren. Die beiden von ihr genannten Vorteile **verbesserte Skalierbarkeit** und die **fehlende Wartung der Infrastruktur** auf Seiten des Cloud-Nutzers, schließen sich dieser Argumentation an [Sti18, S. 6].

Als fünftes Argument, dass für die Nutzung einer Serverless Cloud-Plattform spricht, nennt Stigler die **Kosten** [Sti18, S. 6]. Während sich sinkende Entwicklungskosten aus dem verringerten Entwicklungsaufwand ergeben, ist bei den Betriebskosten Vorsicht geboten. Eivy führt in seinem Artikel *Be Wary of the Economics of “Serverless” Cloud Computing* das Beispiel einer API auf, deren Betrieb durch eine Lambda Funktion deutlich teurer wäre, als mit der Implementation eines Monolithen [Eiv17]. Eine genaue Kostenanalyse ist vor der Entwicklung daher dringend anzuraten.

### 2.1.4 Herausforderungen und Probleme

Ein erstes Problem, das sich beim Serverless Computing ergeben kann, resultiert aus der Abgabe der Verwaltungsaufgaben an den Cloud-Provider. Da dieser die volle Kontrolle über die verwendete Infrastruktur hat, kann der Nutzer dieser Plattform keinerlei Einfluss darauf nehmen. Der Nutzer ist daher beispielsweise auf die vom Provider bereitgestellten Laufzeitumgebungen und Konfigurationsoptionen beschränkt.

Diese Einschränkung äußert sich an mehreren Stellen negativ, beispielsweise beim sogenannten *Cold Start*. Ein Cold Start tritt immer dann ein, wenn beim Eintreffen eines Ereignisses keine freien Ausführungen einer Funktion zur Verfügung stehen, um das Ereignis abzuarbeiten. Wenn eine Funktion eine bestimmte Zeit nicht ausgeführt wurde, oder wenn durch erhöhtes Eventaufkommen alle Instanzen belegt sind, kann dies vorkommen [RC17]. Das System muss dann eine neue Instanz der Ausführungsumgebung starten.

In ihrem Paper *Serverless Computing: Design, Implementation, and Performance* [MSE+16] untersuchen McGrath und Brenner verschiedene Cloud-Provider auf die Dauer eines Cold Starts. Bei ihren Untersuchungen ergaben sich Startzeiten von teils mehreren Sekunden für einzelne Funktionen. Eine Azure Function beispielsweise benötigte, wenn mehr als fünf Minuten zwischen den Aufrufen lag, teils mehr als fünf Sekunden, um einen Funktionsaufruf erneut durchzuführen [MSE+16, S. 409]. Dem Nutzer ist es weder möglich, darauf Einfluss zu nehmen, nach welcher Zeit Funktions-Instanzen im Leerlauf beendet werden, noch kann er festlegen, wie viele dieser Instanzen im Leerlauf zur Reserve laufen. Selbst wenn einer Applikation ein geplanter Anstieg an Ressourcen bevorsteht, kann der Entwickler die Infrastruktur nicht oder nur sehr begrenzt darauf vorbereiten.

Stigler führt einen weiteren Nachteil einer Serverless Umgebung auf: das gemeinsame Nutzen von Ressourcen [Sti18, S. 10]. Insbesondere dadurch, dass Mitbewerber auf derselben Infrastruktur laufen können, haben sie identische Verfügbarkeits- und Skalierungseigenschaften. Dadurch wird verhindert, dass sich eine Applikation von denen der Konkurrenz hinsichtlich dieser Eigenschaften abheben kann. Dieser Nachteil ist nicht nur auf Serverless Umgebungen beschränkt, sondern umfasst viele der in einer Public Cloud betriebenen Dienste.

Einer der bedeutendsten Nachteile, der bei der Nutzung einer Serverless Plattform auftritt, ist der sogenannte *Vendor Lock-In*. Dieser tritt dann auf, wenn eine Applikation komplett auf die Bedürfnisse eines spezifischen Providers angepasst, beziehungsweise im Hinblick auf diesen entwickelt wurde und nicht mit Plattformen anderer Anbieter kompatibel ist. Da jeder Provider eine eigene FaaS-Plattform betreibt, die nicht auf Kompatibilität zu anderen Providern setzt, besteht beim Serverless Computing eine hohe Wahrscheinlichkeit, dass ein Vendor Lock-In stattfindet. Roberts und Chapin nennen auch die Anpassungen an die vom Provider bereitgestellten Ressourcen, wie Datenspeicher als anfällig für einen Vendor Lock-In [RC17, S. 35 f.]. Als Beispiel nennen sie eine Applikation, die als Datenspeicher ein von AWS bereitgestellten S3 Speicher nutzt. Um Lese- und Schreibzugriffe

auf diesen Speicher zu ermöglichen muss die Applikation providerspezifische Schnittstellen nutzen. Eine Anpassung an einen anderen Cloud-Provider und damit ein anderes Speichersystem bringt aufwändige Anpassungen mit sich.

### 2.1.5 Multi-Cloud

Aus unterschiedlichen Gründen kann es sinnvoll sein mehrere Cloud-Umgebungen zusammenzuschließen. Petcu nennt in seinem Paper *Multi-Cloud: Expectations and Current Approaches* zehn Gründe zur Nutzung einer solchen *Multiple Cloud* [Pet13, S. 1]. Diese unterteilt er in Anlehnung an die in [FHT+12] vorgestellte Unterscheidung in zwei Arten. In der *Federated Cloud* kooperieren zwei oder mehr Cloud-Umgebungen miteinander. Dem Nutzer einer Federated Cloud muss dabei nicht bewusst sein, dass er es nicht mit mehreren Cloud-Umgebungen zu tun hat. Anders sieht das bei der Multi-Cloud aus. Bei dieser kooperieren die Cloud-Umgebungen selber nicht, oder zumindest nur in sehr begrenztem Umfang, miteinander. Grozev und Buyya heben insbesondere hervor, dass sich die Zusammenarbeit der Cloud-Umgebungen in einer Multi-Cloud erst durch den Nutzer, beziehungsweise den Service ergibt [GB14].

Von den von Petcu genannten zehn Gründen für den Zusammenschluss von Cloud-Umgebungen ergeben einige den Use-Case für eine Multi-Cloud. Er nennt beispielsweise den Umgang mit Lastspitzen in einer Private Cloud, indem bei Bedarf Ressourcen einer Public Cloud genutzt werden [Pet13, S. 1]. In diesem Use-Case spricht man von einer Hybrid Cloud. Bei dieser werden laut der Definition des *National Institute of Standards and Technology (NIST)* mehrere Cloud-Umgebungen unterschiedlicher Deployment Models zusammengeschlossen [MG11, S. 3]. Neben Public und Private Cloud-Umgebungen kann ein solcher Zusammenschluss auch eine sogenannte *Community Cloud* beinhalten. Eine Community Cloud wird, anders als eine Private Cloud, nicht von einer einzelnen, sondern von einer geringen Anzahl an Organisationen zusammen genutzt.

Neben der Behandlung von Lastspitzen nennt Petcu auch die Einhaltung von Bestimmungen, beispielsweise dem Gesetz, als Grund für die Nutzung einer Hybrid Cloud [Pet13, S. 1]. In einigen Fällen kann es zum Beispiel aus datenschutzrechtlichen Gründen nicht erlaubt sein Daten in einer Public Cloud aufzubewahren oder zu verarbeiten. Ebenso kann es aufgrund von betrieblichen Vorgaben notwendig sein, geschäftskritische Daten nur innerhalb eines eigenen Rechenzentrums zu verarbeiten. Um in einem dieser Fälle trotzdem auf die Vorteile einer Public Cloud-Umgebung setzen zu können, können die Komponenten oder Teile eines Services, die nicht der Beschränkung unterliegen, innerhalb der Public Cloud verarbeitet werden, während alle Daten, die geschützt werden müssen, innerhalb der Private Cloud verarbeitet werden.

Um einen Multi-Cloud Betrieb zu ermöglichen, ist ein Nutzer einer Public Cloud auf die vom Provider bereitgestellten Möglichkeiten beschränkt. Insbesondere Schnittstellen, um Daten- und Ereignisflüsse zwischen den Cloud-Umgebungen zu ermöglichen, werden möglicherweise vom Provider nicht zur Verfügung gestellt. Um diese fehlenden Schnittstellen zu ersetzen und Inkompatibilitäten zwischen den Providern auszugleichen, ist daher angepasste Software notwendig. Diese muss insbesondere zwei Aufgaben erledigen. Zum einen muss sie Events aus der Cloud-Umgebung hinausleiten, damit diese auch außerhalb zur Verfügung stehen und zum anderen muss sie die Events in Form von Nachrichten an alle anderen, interessierten Cloud-Umgebungen vermitteln.

Bei einer Private Cloud Lösung kann dies anders aussehen. In ihrer Definition beschreibt die NIST eine Private Cloud lediglich als exklusiv für eine Organisation bereitgestellte Cloud-Umgebung [MG11, S. 3]. Ob diese On-Premise, also innerhalb eines eigenen Rechenzentrums der Organisation, oder Off-Premise, innerhalb des Rechenzentrums eines Providers läuft, wird nicht definiert, wie Dillon et al. hervorheben [DWC10, S. 28] Ebenso ist nicht geklärt, welchen Einfluss Entwickler, die eine Private Cloud nutzen, auf die Cloud-Umgebung ausüben können.

Der Unterschied des Einflusses der Entwickler auf die Umgebung wird insbesondere an einem Beispiel deutlich. OpenFaaS ist ein Framework zur Bereitstellung einer *Function-as-a-Service* Umgebung innerhalb einer Kubernetes oder Docker Umgebung [Opeb]. Somit kann dies beispielsweise On-Premise als Private Cloud betrieben werden. Aufgrund des direkten Einflusses des Nutzers auf diese Umgebung, kann sie von ihm direkt angepasst werden. OpenFaaS ermöglicht es beispielsweise über ein eigens dafür erstelltes SDK eigene Trigger für Funktionen zu erstellen [Opea]. Eine von einem Provider betriebene Umgebung bietet diese Möglichkeit in der Regel nicht.

## 2.2 Serverless Framework

Nicht zu verwechseln mit dem Serverless Computing ist das Serverless Framework. Das Serverless Framework ist ein Kommandozeilentool, das es ermöglicht serverlose Applikationen zu schreiben, zu deployen und anschließend zu betreiben [Sera]. Dazu nutzt das Framework keine eigene Hardware, sondern setzt auf bestehenden Cloud-Providern auf. Es unterstützt eine Vielzahl an Providern und ermöglicht es dadurch mit lediglich einem einzigen Tool Applikationen auf die Systeme unterschiedlicher Cloud-Provider zu deployen. Diese Abstraktionsschicht ermöglicht es beispielsweise Arbeitsschritte zur Erstellung oder zum Betrieb einer serverlosen Applikation providerunabhängig zu machen. Anstatt mit mehreren Oberflächen oder Tools unterschiedlicher Provider zu arbeiten, muss lediglich das Serverless Framework eingesetzt werden.

Das Serverless Framework wird als Open Source Software von der *Serverless, Inc.* entwickelt. Rund um das Framework werden von diesem Unternehmen weitere Applikationen und Services betrieben. Mittlerweile wurde das Angebot um eine kostenpflichtige Enterprise Version erweitert, die das Framework um weitere Services, wie eine verbesserte Monitoring- und Alarm-Funktionalität ergänzt.<sup>1</sup>

Der Grundgedanke hinter dem Serverless Framework ist es, sämtliche Konfigurationseinstellungen in einer zentralen Konfigurationsdatei zu speichern. Dadurch wird ermöglicht mehrere absolut identische Deployments durchzuführen, ohne, dass beispielsweise einem Administrator Konfigurationsfehler unterlaufen und die Umgebung dadurch fehleranfällig wird. Dieser Grundgedanke die Konfiguration, ähnlich wie den Quellcode einer Applikation, als versionierbare Konfigurationsdateien zu pflegen, nennt sich *Infrastructure as Code* und wird unter anderem von Morris in seinem gleichnamigen Buch beschrieben [Mor16].

In Listing 2.1 ist ein Ausschnitt einer Konfigurationsdatei des Serverless Framework abgebildet. Ein Service mit dem Namen `users` wird angelegt. Dieser enthält zwei Funktionen. Je eine zum Anlegen und eine zum Löschen eines Nutzers. Der Quellcode der Funktionen wird im selben Ordner, in anderen Dateien verwaltet.

---

<sup>1</sup><https://serverless.com/enterprise/>

---

**Listing 2.1** Ausschnitt einer Konfigurationsdatei des Serverless Framework nach [Serd]

---

```

service: users

functions: # Your "Functions"
  usersCreate:
    events: # The "Events" that trigger this function
      - http: post users/create
  usersDelete:
    events:
      - http: delete users/delete

```

---

Neben Funktionen können in der Datei weitere Informationen, wie die Konfiguration des gewählten Providers, zusätzlicher Ressourcen oder vergebener Berechtigungen abgelegt werden. Beim Deployment, durch das Kommandozeilentool des Serverless Framework, werden die gewählten Konfigurationen auf den Servern des Cloud-Providers deployed und konfiguriert.

Das Serverless Framework wird in Kapitel 5 für die prototypische Implementierung verwendet. Dieses grundlegende Wissen wird daher für das Verständnis der Arbeit benötigt.

## 2.3 CloudEvents

CloudEvents beschreibt sich selbst als Spezifikation zur Beschreibung von Event Daten in einem einheitlichen Format, um Interoperabilität zwischen Services, Plattformen und Systemen bereitzustellen [CNCb]. Der aktuelle Stand der Spezifikation befindet sich auf einem GitHub Repository.<sup>2</sup> Im Mai 2019 wurde die aktuellste Version veröffentlicht (Stand Juni 2019). Aufgrund der weiteren Verbreitung, Kompatibilität zu bestehenden Softwaresystemen und, da zu Beginn dieser Arbeit Version 0.2 noch nicht finalisiert war, wird Version 0.1 in dieser Arbeit genutzt.

Die Spezifikation CloudEvents wurde von der Arbeitsgruppe *Serverless* der *Cloud Native Computing Foundation (CNCF)* ins Leben gerufen und wird von ihr betreut. Das erklärte Endziel ist es, die finale Spezifikation der CNCF vorzulegen [CNC19].

Die CNCF ist eine Organisation, die sich das Ziel gesetzt hat Projekte zu fördern, die natives Cloud-Computing ermöglichen [Cloa]. Dazu haben sich dieser Organisation mehrere hundert Mitglieder angeschlossen, zu denen unter anderem einige der größten Public Cloud-Provider, aber auch viele kleinere Unternehmen gehören. Ihre Projekte klassifiziert die CNCF in drei unterschiedliche Stufen, die angeben, wie ausgereift ein Projekt ist und wie weit seine Entwicklung oder Erarbeitung fortgeschritten ist [Cloe]. CloudEvents befindet sich auf der *Sandbox Project* Stufe und ist somit noch in der Erarbeitung.

In Version 0.1 nennt die CloudEvents Spezifikation zehn verschiedene Attribute, aus der ein Event besteht. An dieser Stelle werden alle Attribute die zum Verständnis dieser Arbeit erforderlich sind beschrieben. Das Attribut *eventType* gibt an, welche Art eines Ereignisses aufgetreten ist. Dieses

---

<sup>2</sup><https://github.com/cloudevents/spec>

**Listing 2.2** Beispiel eines Events nach der CloudEvents Spezifikation im JSON Format aus [CNCd].

---

```
{
  "cloudEventsVersion" : "0.1",
  "eventType" : "com.example.someevent",
  "eventVersion" : "1.0",
  "source" : "/mycontext",
  "eventId" : "A234-1234-1234",
  "eventTime" : "2018-04-05T17:31:00Z",
  "extensions" : {
    "comExampleExtension" : "value"
  },
  "contentType" : "text/xml",
  "data" : "<much wow='xml' />"
}
```

---

Attribut muss in jedem Event vorhanden sein. Die Spezifikation schlägt vor den Event-Typ mit einem rückwärts geschriebenen Domainnamen zu beginnen. Dadurch wird eine eindeutige Zuordnung des aufgetretenen Events ermöglicht. Als Beispiel wird der Event-Typ `com.github.pull.create` angegeben [CNCa]. Dieser gibt an, dass er von GitHub, aus dem Bereich der Pull Requests stammt und aufgrund eines neu erstellten Pull Requests ausgelöst wurde.

Als weiteres zentrales Attribut nennt die Spezifikation die *source*. Diese gibt eine Referenz auf das Objekt oder die Ressource an, aufgrund derer das Event ausgelöst wurde. Im genannten Beispiel könnte dies die URL zum in GitHub erstellten Pull Request sein. Den genauen Aufbau dieses Attributs überlässt CloudEvents dem Erzeuger des Events und schreibt lediglich die Angabe im URI Format vor.

Neben Attributen zur Versionierung des Event-Typs, einer eindeutigen ID des Events, der Uhrzeit zum Auftritt des Events, beschreibt die Spezifikation das Attribut *data*. Dieses optionale Attribut beinhaltet die Payload des Events. Das Format, in dem der Inhalt übergeben wird, wird dafür im *contentType* Attribut angegeben. Payload eines Events können sämtliche Informationen rund um das Event sein, die für einen Empfänger nützlich sein können. Eine Vorschrift, die vorgibt welche Informationen in dieses Attribut gehören gibt die Spezifikation nicht.

Ein konkretes Datenformat, in dem ein solches Event dargestellt, serialisiert oder übertragen wird, schreibt die Spezifikation nicht vor. Stattdessen wird definiert, wie diese Attribute in unterschiedlichen Formaten dargestellt werden. In Version 0.1 beschränkt sich dies noch auf das JSON Format und ein Mapping auf das HTTP-Protokoll. Weitere Formate folgen in späteren Versionen. Die Darstellung eines CloudEvents im JSON Format ist in Listing 2.2 dargestellt.

## 2.4 Event Gateways

Sowohl Provider, als auch Softwarehersteller erkannten bereits das Problem der fehlenden Multi-Cloud Unterstützung traditioneller Serverless Umgebungen. Sie versuchen mit unterschiedlichen Gateways dieses Problem anzugehen.

Eine konkrete Implementierung eines Event Gateways steht mit dem **Serverless Event Gateway** bereit. Serverless, Inc., die neben dem Serverless Framework auch das Event Gateway entwickeln, bezeichnen es als einen Event-Router [Serb]. Das Gateway kann als Verwaltungszentrale zwischen Cloud-Umgebungen genutzt werden und setzt dabei auf die Spezifikation der CloudEvents [Ser19].

Das Serverless Event Gateway lässt sich einfach in bestehende Applikationen integrieren, die auf das Serverless Framework aufsetzen. Es wird allerdings unabhängig von diesem entwickelt und ist als quelloffenes Projekt auf GitHub gehostet. Aufgrund der einfachen Schnittstellen kann es auch außerhalb des Framework problemlos eingesetzt werden.

In der Projektbeschreibung wird davon abgeraten das Serverless Event Gateway momentan in Produktivumgebungen einzusetzen, da es sich noch in der Entwicklung befindet und es zu gravierenden Änderungen in den Programmschnittstellen kommen kann [Ser19].

Als zentrale Verwaltungseinheit bildet das Serverless Event Gateway einen Single-Point-of-Failure. Dies wird deutlich, wenn der Weg eines Events betrachtet wird. Das Gateway ermöglicht es Funktionen zu registrieren und diese nach dem Publish/Subscribe-Prinzip beim Eintritt von festgelegten Events auszuführen. Funktionen können entweder über HTTP oder eine providerspezifische Schnittstelle aufgerufen werden. Das Event muss dementsprechend erst aus der eigentlichen Umgebung herausgeleitet werden, um vom Event Gateway verarbeitet zu werden. Dort wird entschieden, welche Aktionen ausgeführt, beziehungsweise, welche Funktionen aufgerufen werden. Diese Aufrufe finden wieder in die Umgebung selbst statt.

Das Event wird also von der Cloud-Umgebung nach außen und wieder hinein geleitet. Neben einer höheren Latenz des Aufrufs bedeutet dies die Gefahr der Nichterreichbarkeit des zentralen Gateways oder der Störung des Kommunikationskanals. Bei einigen Anwendungsfällen kann dieses Vorgehen nicht akzeptabel sein.

Ein ähnliches Softwareprodukt, das das Problem der fehlenden Multi-Cloud Unterstützung angehen möchte ist **Azure Event Grid**. Dieses ist ein von Microsoft entwickelter und in der Azure Cloud-Computing-Plattform betriebener Service zum Ereignisrouting. Laut Angaben von Microsoft eignet er sich um Anwendungen mit Eventbasierten Architekturen zu erstellen [Micb]. Event Grid unterstützt nativ die Anbindung verschiedener Ereignisquellen der Azure Cloud. Ebenso kann von ihm in Form von unterschiedlichen Ereignis-Handlern auf Events reagiert werden. Zu diesen gehören auch die Azure Functions.

Es ist dabei primär als zentrales Gateway innerhalb der Azure Cloud konzipiert, um eintretende Events abzufangen und entsprechende Aktionen aufzurufen. Durch offene Schnittstellen, wie die native Unterstützung von CloudEvents, eignet sich Event Grid auch zum Einsatz in Multi-Cloud-Umgebungen. Ebenso stellt Event Grid einen HTTP-Endpoint zur Verfügung, an den Events in das System geschickt werden können.

Da es sich beim Event Grid allerdings um einen Service der Azure Cloud handelt, können mit ihm alleine keine Multi-Cloud-Umgebungen sinnvoll betrieben werden. Allen voran keine Umgebungen, in denen Azure nicht zum Einsatz kommt.

Die hier beschriebenen Gateways stellen daher alle eine Basis dar, auf denen eine Eventbasierte Kommunikation in Multi-Cloud-Umgebungen realisiert werden kann. Eine Komponente allein ist allerdings keine fertige Lösung. Dafür sind weitere Konzepte und Softwarekomponenten notwendig.





## 3 Anforderungen

Zur Konzeption und späteren Validierung des benötigten Softwarestacks, müssen in einem ersten Schritt die Anforderungen an diesen Softwarestack definiert werden. Um die grundlegenden Anforderungen an eine Multi-Cloud Serverless Umgebung abzuleiten, wird im Folgenden ein Referenzszenario herangezogen. Neben dieser Applikation leiten sich Anforderungen aus der Literatur ab. Hierzu zählen insbesondere die Anforderungen an einen Nachrichtenkanal aus dem Buch Enterprise Integration Patterns von Hohpe und Woolf [HWB04]. Einige Anforderungen, insbesondere Qualitätsanforderungen, orientieren sich dabei an denen einer providerspezifischen Serverless Umgebung. Diese werden im Folgenden anhand von Use-Cases herausgearbeitet, die Realbeispiele von Unternehmen beim Aufbau einer oder mehrerer Applikationen in einer providerspezifischen Serverless Umgebung wiedergeben.

In der Anforderungsanalyse wird zwischen funktionalen und nichtfunktionalen Anforderungen unterschieden. Die funktionalen Anforderungen geben die grundlegenden Eigenschaften des Systems an. Solche Anforderungen definieren beispielsweise Ein- und Ausgabewerte, Berechnungen oder die Nutzeroberfläche eines Systems. Nichtfunktionale Anforderungen beschreiben Eigenschaften an das System, die Qualitätsaspekte definieren. Hierzu gehören Eigenschaften, wie Performance, Stabilität oder Erreichbarkeit eines Systems.

### 3.1 Referenzszenario

Um anhand dieser die Anforderungen herauszuarbeiten, wird an dieser Stelle ein Referenzszenario in Form einer Serverless Applikation in einer Multi-Cloud-Umgebung herangezogen. Bei dieser Applikation handelt es sich um eine Webseite, die es Nutzern erlaubt Fotos, die von anderen Nutzern hochgeladen wurden zu betrachten und diese Liste durch weitere Fotos zu erweitern. Um die Bilder übersichtlicher ordnen zu können, werden ihnen automatisiert Schlagwörter zugeordnet, nach denen die Fotos gefiltert werden können.

Die Webseite der Applikation ist in einem Mockup in Abbildung 3.1 dargestellt. Im obersten Kasten kann ein neues Foto vom Nutzer hochgeladen werden. Alle bereits hochgeladenen Fotos folgen in chronologischer Reihenfolge in der Liste darunter. Neue Bilder werden an den Listenanfang gestellt. Unter jedem Foto erscheinen die zugeordneten Schlagwörter. Beim Klick auf diese Schlagwörter werden sämtliche Fotos sichtbar, denen ebenfalls dieses Schlagwort zugeordnet wurde.

Um den Nutzern eine schnellere Darstellung bei geringerem Datenverbrauch zu ermöglichen, werden die Bilder nach dem Upload zusätzlich komprimiert im Datenspeicher gehalten. Damit die Beispielanwendung möglichst kompakt bleibt, wird auf eine Benutzerverwaltung zum Upload neuer Fotos verzichtet.

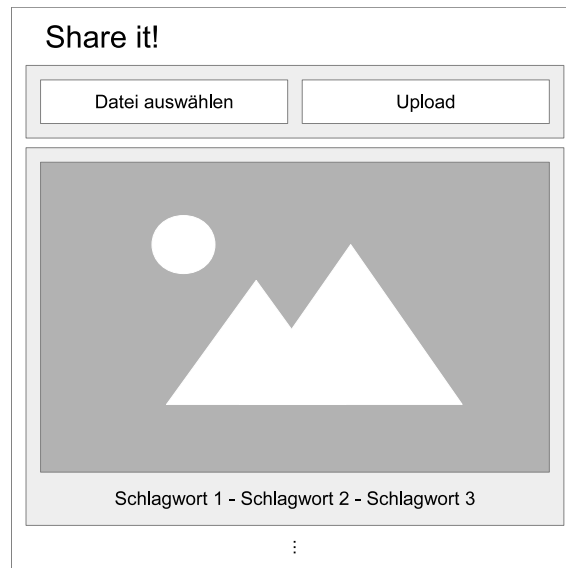


Abbildung 3.1: Mockup der Benutzeroberfläche des Referenzszenarios

Im Folgenden wird zuerst der Aufbau der Architektur und die Funktionalität der verwendeten Komponenten erklärt und anschließend die Auswahl der im Referenzszenario verwendeten Cloud-Provider.

### 3.1.1 Aufbau

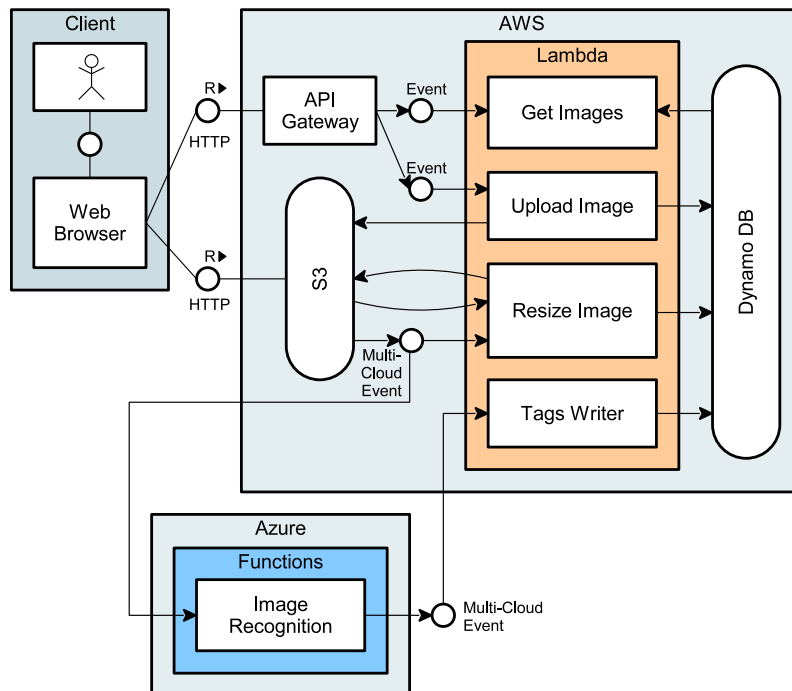
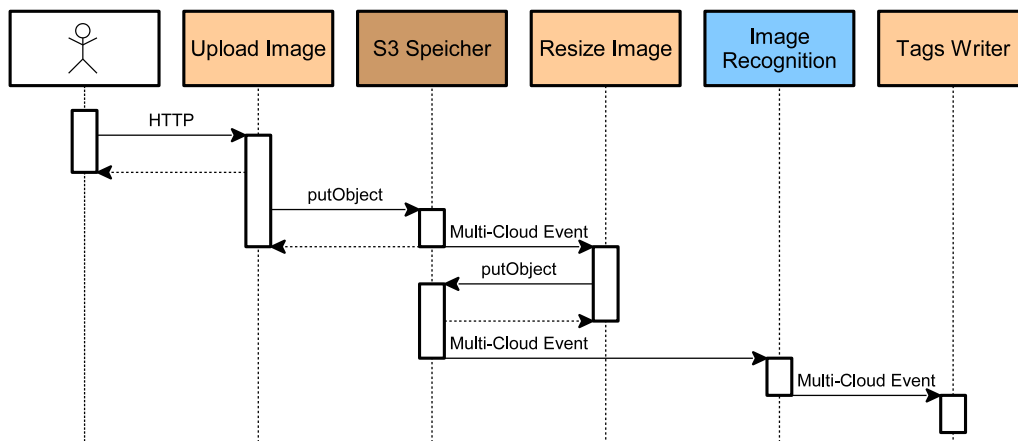


Abbildung 3.2: Architektur einer Eigenentwicklung in einer Serverless Multi-Cloud-Umgebung

Der **Client** öffnet mit einem Web Browser eine Webseite, die per JavaScript eine HTTP-Verbindung zur Lambda Funktion *Get Images* über das API Gateway herstellt. Das **API Gateway** ermöglicht es innerhalb der AWS Umgebung Lambda Funktionen per HTTP-Verbindung ausführbar zu machen. **Get Images** holt sich aus der **DynamoDB** Informationen über alle momentan auf die Plattform geladenen Bilder. Hierzu gehören die URLs zum S3 Speicher, in der die Bilddateien gespeichert sind und die Schlagworte, die den Bildern bisher zugeordnet wurden.

Sobald ein Nutzer ein eigenes Foto auf die Webseite stellen möchte, wird der Ablauf durchgeführt, der im Sequenzdiagramm in Abbildung 3.3 abgebildet ist. Zur Übersichtlichkeit und, weil diese Komponente selbst keine Aufrufe tätigt, beziehungsweise Events auslöst, wurde die DynamoDB in diesem Diagramm weggelassen. Durch einen HTTP-Aufruf an das API Gateway wird die Lambda Funktion **Upload Image** aufgerufen. Dieser wird mit dem Aufruf die Bilddatei übergeben und von dieser in den **S3 Speicher** geladen. Zusätzlich werden von der Upload-Funktion Meta-Informationen über das neu hinzugefügte Bild in die DynamoDB geschrieben. Zu diesem Zeitpunkt befindet sich lediglich die URL der Originaldatei in den Metadaten. Die komprimierte URL, sowie die Schlagwörter, sind noch nicht bekannt.



**Abbildung 3.3:** Sequenzdiagramm des Upload eines Fotos im Referenzszenario

Sollte ein Nutzer sich nun alle Bilder auf der Webseite anzeigen lassen, sieht er bereits das Bild. Dieses wird, aufgrund der fehlenden komprimierten Version, in voller Auflösung geladen. Damit dies nicht mit allen Bildern passiert, wird eine Bildkomprimierung eingesetzt.

Durch das vom S3 Speicher ausgelöste Multi-Cloud Event, wird die Funktion **Resize Image** aufgerufen. Diese bekommt mit dem Event die benötigten Informationen, um das entsprechende Bild aus dem S3 Speicher zu laden, anschließend auf eine festgelegte Größe zuschneiden und komprimieren und danach diese komprimierte Version wieder auf denselben S3 Speicher hochzuladen. Die neue Datei unterscheidet sich in der Pfadangabe von der Originaldatei. Während die Originaldatei im Unterordner `upload` gespeichert ist, liegt die neue im Unterordner `compressed`. Damit die komprimierte Version von den Web Browsern der Nutzer gefunden wird, werden die Informationen um die URL der komprimierten Datei ergänzt.

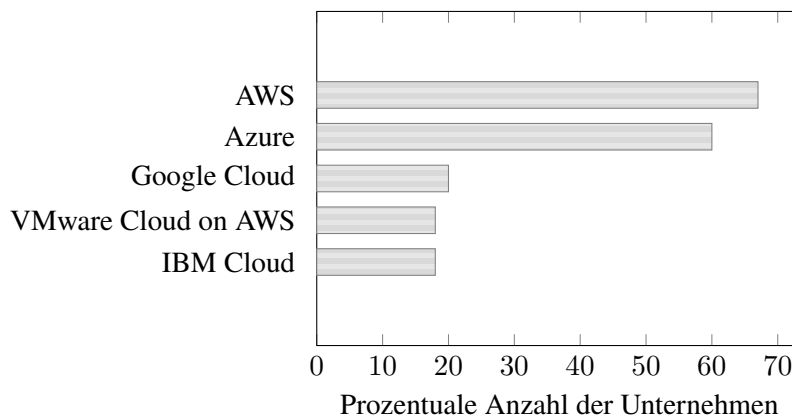
Beim Hochladen der komprimierten Datei wird erneut ein Multi-Cloud Event emittiert. Unterschied zum Event, das durch den Upload der Originaldatei ausgelöst wurde besteht in der Pfadangabe der hochgeladenen Ressource. Durch dieses Event wird die Funktion **Image Recognition** ausgelöst. Diese befindet sich, anders als alle bisher ausgelösten Funktionen, in der Azure Cloud. Innerhalb

dieser Funktion wird eine, von der Azure Cloud bereitgestellte Bilderkennung ausgeführt. Diese liefert Schlagworte zurück, die gemäß dieser Bilderkennung auf das untersuchte Bild zutreffen. Um diese Schlagworte nun zu den Metadaten des Bildes hinzuzufügen, löst die Funktion ein Multi-Cloud Event aus.

Dieses von der Funktion ausgelöste Event löst innerhalb der AWS Umgebung die Funktion **Tags Writer** aus. Die im Event enthaltenen Informationen speichert die Funktion wieder in der DynamoDB. Beim nächsten Aufruf eines Nutzers wird nun das Bild in komprimierter Form zusammen mit den Schlagworten ausgegeben.

#### 3.1.2 Cloud-Provider

Auf dem Markt der Cloud-Provider haben sich mittlerweile mehrere Anbieter positioniert. Sowohl im Bereich der Public, als auch der Private Cloud besteht eine große Anzahl an Anbietern, zwischen denen eine Auswahl getroffen werden muss. Für die entwickelten Prototypen wurde die Auswahl der Cloud-Provider anhand der Nutzungszahlen getroffen. RightScale, ein Unternehmen, das selbst Software-as-a-Service anbietet, stellt dafür die Ergebnisse einer jährlichen Umfrage von IT Unternehmen bereit.



**Abbildung 3.4:** Umsetzung von Projekten in Unternehmen mit Hilfe der Public Cloud. Auszug aus [Rig19]

Die Ergebnisse der Umfrage, welche Public Cloud-Provider in Projekten des Unternehmens eingesetzt werden, sind in Abbildung 3.4 dargestellt. In ihr wird deutlich, dass Amazon mit den *Amazon Web Services (AWS)* und Microsoft mit *Azure* mit Abstand die größte Verbreitung auf dem Markt haben. Daher sollen diese beiden Cloud-Umgebung in den Prototypen Verwendung finden. Bei der Erarbeitung der Prototypen soll allerdings auf Erweiterbarkeit – insbesondere auch auf weitere Provider – geachtet und diese dementsprechend analysiert werden.

Sowohl die Datenbank, als auch der Datenspeicher, in dem die Fotos gespeichert werden, sollen auf der AWS Plattform laufen. Die automatisierte Schlagworterkennung läuft in Microsoft Azure.

Dabei sei darauf hingewiesen, dass sich bei dieser Beispielapplikation berechtigterweise die Frage stellen lässt, ob für diese tatsächlich das Deployment in eine Multi-Cloud-Umgebung die geeignetste Form ist. Die Bildanalyse ließe sich durchaus auch in der AWS Umgebung realisieren, in der auch die restlichen Komponenten der Applikation deployed werden. Neben dem zusätzlichen Aufwand,

der durch das Aufteilen der Funktion in mehrere Cloud-Umgebungen während der Entwicklung und gegebenenfalls während des Betriebs entsteht, ließen sich auch Kosten einsparen, die durch zusätzliche benötigte Ressourcen entstehen. Der in diesem Referenzszenario verwendete Aufbau steht stellvertretend für viele andere Use-Cases.

Einen Use-Case, der zwar eher selten eintritt, die Sinnhaftigkeit einer Multi-Cloud mit zwei Public Cloud-Providern allerdings legitimiert, nennt Petcu in seinem Paper *Portability and Interoperability between Clouds: Challenges and Case Study* [Pet11, S. 63]. Bei der Migration einer Applikation zu einem anderen Cloud-Provider kann es sinnvoll sein Funktionen und Komponenten einzeln zu migrieren. Dazu wird während der Umstrukturierung eine Verbindung zwischen der alten und der neuen Cloud-Umgebung benötigt.

Wie Woo und Mirkovic in ihrem Paper *Optimal application allocation on multiple public clouds* [WM14] zeigen, besteht außerdem die Möglichkeit durch ein Deployment in eine Multi-Cloud-Umgebung sowohl die Performance, als auch die entstehenden Kosten zu senken, im Vergleich zum Deployment in einer Single-Cloud-Umgebung. Dies erreichen sie durch geschickte Verschiebung der Ressourcen in die Public Cloud-Umgebung, in der diese vom entsprechende Provider besonders performant oder günstig zur Verfügung gestellt wird. Durch diese Kombination gelingt es ihnen sowohl die Mehrkosten des zusätzlichen Aufwandes auszugleichen, als auch eine bessere Performanz zu erzielen. Da es sich bei den untersuchten Ressourcen sowohl um dedizierte Rechenressourcen in Form von virtuellen Maschinen, als auch Datenbanken und Speicher handelt, kann dieses Ergebnis für eine Serverless Umgebung nicht uneingeschränkt angenommen werden. Hierzu sind weitere Untersuchungen notwendig.

In Abschnitt 2.1.5 wurde bereits ein weit verbreiteter Einsatz eine Multi-Cloud-Umgebung beschrieben, die Hybrid Cloud. Durch den Zusammenschluss unterschiedlicher Cloud Typen lassen sich auf diese Art verschiedene Use-Cases abbilden.

Beispielsweise kann ein Online-Shop den bereits genannten Use-Case einer Multi-Cloud nutzen um Lastspitzen abzufangen, wie sie zur Weihnachtszeit häufig vorkommen, indem dann Ressourcen der Public Cloud in Anspruch genommen werden. Die Private Cloud kann mit OpenFaaS betrieben werden. Durch dieses Prinzip macht sich der Betreiber des Online-Shops unabhängiger vom Cloud-Provider und kann trotzdem weiterhin die Vorteile der nahezu grenzenlosen Skalierung der Public Cloud nutzen.

Einen weiteren Einsatz eines Systems zur Eventbasierten Kommunikation in Multi-Cloud-Umgebungen des Serverless Computing nennen Wurster et al. in ihrer Arbeit [WBK+18]. In diesem Paper beschreiben sie die Möglichkeit zur Modellierung einer serverlosen Applikation in OpenTOSCA, einem Ökosystem für den *Topology and Orchestration Specification for Cloud Applications (TOSCA)* Standard der OASIS [Ins17]. Um damit Applikationen für Multi-Cloud-Umgebungen betreiben zu können fehlt momentan die Konzeption eines Systems zur Weiterleitung von Events. Dieses Konzept kann dabei explizit modelliert, oder implizit genutzt werden. Bei der expliziten Modellierung ist der Anwender selbst dafür verantwortlich alle benötigten Komponenten im Modell zu ergänzen, während er bei der impliziten Modellierung lediglich ein Multi-Cloud Event modelliert. Die reale Umsetzung wird dann vom OpenTOSCA System deployed und verwaltet.

Insbesondere dieser implizite Use-Case unterscheidet sich insofern von den bisher genannten, da er ein sehr generisches Konzept benötigt. Anstatt lediglich einen Softwarestack für vorher festgelegte Cloud-Umgebungen zur Verfügung zu stellen, muss eine potenziell sehr große Anzahl

an Möglichkeiten berücksichtigt werden, welche Cloud-Umgebungen untereinander kommunizieren können müssen. Daher herrschen dabei besondere Ansprüche an die Erweiterbarkeit und die standardkonforme Implementation von Schnittstellen.

### 3.2 Funktionale Anforderungen

**Anforderung 1 - Multi-Cloud Event:** Dem Anwender muss es durch das zu erarbeitende Softwaresystem ermöglicht werden durch die Verbindung einer beliebigen Event-Quelle mit einem Event-Empfänger diese beiden Komponenten miteinander zu verknüpfen. Dies muss möglich sein, unabhängig davon, ob sich Quelle und Empfänger in der selben, oder in unterschiedlichen Cloud-Umgebungen befinden. Ein solches Event wird daher als *Multi-Cloud Event* bezeichnet.

Eine solche Verknüpfung findet sich in der Beispielapplikation aus Abbildung 3.2 unter anderem zwischen dem S3 Speicher als Event-Quelle und der Lambda Funktion *Resize Image*. Diese muss in der Umsetzung durch die entsprechende Konfiguration der AWS Umgebung implementiert werden. Ebenso muss dies beim Event zwischen dem genannten S3 Speicher und der Azure Function *Image Recognition* geschehen. Dabei muss beachtet werden, dass sich der S3 Speicher in der AWS Umgebung, und die *Image Recognition* in der Azure Umgebung befinden. Gegebenenfalls muss hierzu weitere Software deployt werden, um eine Weiterleitung des Events zu ermöglichen.

**Anforderung 2 - Abfangen eines Events:** Um Anforderung 1 zu erfüllen, muss das Softwaresystem in der Lage sein, beliebige Events innerhalb einer Cloud-Umgebung abfangen zu können. Dieses Abfangen kann auf unterschiedliche Art und Weise geschehen und muss der entsprechenden Umgebung und dem abzufangenden Event angepasst werden. Neben den vom S3 Speicher ausgelösten Events müssen auch weitere Ereignisse der AWS Umgebung implementiert werden können.

**Anforderung 3 - Benutzerdefiniertes Auslösen eines Events:** Einem Anwender soll es möglich sein, eigene Events innerhalb einer Funktion, oder einer anderen Applikation auszulösen. Ein solches Event unterscheidet sich lediglich in seiner Herkunft und den Nutzdaten von abgefangenen Events, nicht in seinem Aufbau oder seiner Weiterleitung.

Die *Image Recognition* kann im Beispiel ein Event auslösen, um den *Tags Writer* über das Ergebnis der Bilderkennung zu informieren. Dieses Event soll identisch zu einem abgefangenen Event behandelt und über das gleiche Softwaresystem weitergeleitet werden.

**Anforderung 4 - Weiterleiten von Events:** Nachdem ein Event abgefangen wurde, muss es vom Softwaresystem weitergeleitet werden. Dafür muss dieses System wissen, welche Events an welche Komponenten weitergeleitet werden müssen. Um dies zu erreichen muss geklärt werden, an welchen Eigenschaften Events unterschieden werden können und anhand welcher Faktoren eine Filterung zur Weiterleitung stattfinden kann.

Bei der Weiterleitung von Events zwischen Quelle und Empfänger wird ein sogenannter Nachrichtenkanal benötigt. Hohpe und Woolf beschreiben einen solchen Nachrichtenkanal als eine virtuelle Röhre, die einen Empfänger und einen Sender miteinander verbindet [HWB04, S. 75]. Für die Auswahl dieses Kanals müssen laut ihnen mehrere Faktoren beachtet werden. Es muss festgelegt sein, was passiert, wenn der Kanal nicht erreichbar ist, oder wenn der Empfänger kaputte oder nicht identifizierbare Nachrichten erhält.

Durch den Aufbau von eventbasierten Applikationen wird durch das Prinzip der Entkopplung [EN10, S. 34] [Mic06, S. 3] deutlich, dass eine Event-Quelle nicht wissen muss, ob und wie viele Empfänger auf ein Event hören. Sobald ein Event erfolgreich dem Nachrichtenkanal übergeben wurde, muss sich dieser um die Zustellung an alle aktiven Empfänger kümmern. Dabei kann es vorkommen, dass zwei oder mehr Empfänger auf ein Event hören.

Anders sieht es aus, wenn der Nachrichtenkanal momentan nicht vorhanden oder erreichbar ist. Das Event muss dann zwischengespeichert und gegebenenfalls wiederholt gesendet werden. Sollte über einen längeren Zeitraum keine Zustellung erfolgreich sein, bleibt lediglich das Schreiben einer Fehlermeldung, beziehungsweise das Alarmieren. Identisch dazu soll sich ein Nachrichtenkanal verhalten, wenn dieser einen aktiven Empfänger nicht erreicht.

Laut Hohpe und Woolf muss auch geklärt sein, wie ein Empfänger mit unbekanntem, beschädigten oder für ihn unlesbaren Nachrichten umgeht [HWB04, S. 75]. Unbekannte Nachrichten stellen in eventbasierten Kommunikationsumgebungen in der Regel keine Seltenheit dar, da jeder Teilnehmer innerhalb dieser Umgebung beliebige Nachrichtentypen versenden kann. Eine Nachricht, die einem Empfänger nicht bekannt ist, kann von ihm somit ignoriert werden, da er nicht auf dieses Ereignis reagieren muss. Bei einer offensichtlich defekten Nachricht soll der Empfänger eine Fehlermeldung loggen und, falls möglich, dem Sender eine fehlerhafte Response zurückmelden, um ein erneutes Übertragen des Ereignisses zu veranlassen.

Die Kommunikation innerhalb des Kanals findet unidirektional statt. Daten werden lediglich von der Quelle zum Empfänger weitergeleitet. Ein synchroner Aufruf mit direkter Beantwortung auf ein Event soll nicht betrachtet werden.

**Anforderung 5 - Private Events:** Es kann vorkommen, dass manche Events bestimmte Umgebungen nicht verlassen dürfen. Während es beispielsweise für die Azure Function *Image Recognition* essentiell ist, dass das entsprechende Event zum Upload eines Fotos in die Azure Cloud weitergeleitet wird, kann es für Events mit personenbezogenen Daten, wie sie beispielhaft in einer Benutzerverwaltung auftreten können, aus datenschutzrechtlichen Gründen notwendig sein, dass sie niemals in der Azure Umgebung verarbeitet werden.

Insbesondere kann es vorkommen, dass zwei Cloud-Umgebungen Events empfangen sollen, die nicht in dritte Umgebungen gelangen dürfen. Dies muss bei der Weiterleitung berücksichtigt werden.

**Anforderung 6 - Auslösen von Aktionen:** Neben dem Abfangen und Weiterleiten eines Events gehört ebenso das Auslösen einer Aktion zu den Anforderungen an das Softwaresystem. Bei diesen Aktionen handelt es sich in der Regel um Funktionen, die als FaaS vom Cloud-Provider bereitgestellt werden. Es soll allerdings auch möglich sein, eigene Software, beispielsweise als eigenständige Applikation, zu implementieren und an das System zu koppeln.

Für das Auslösen einer Aktion aufgrund eines Events stellt sich die Frage, an welcher Stelle des Systems, beziehungsweise in welcher Komponente die Entscheidung getroffen wird. Mit der Beantwortung dieser Frage wird auch festgelegt, welche Events wohin weitergeleitet werden müssen. Es ist beispielsweise möglich vor dem Versand zu entscheiden, welche Aktionen ausgelöst werden und nur die entsprechenden Aktionen weiterzuleiten. Ebenso ist es möglich alle Aktionen weiterzuleiten und beim Empfänger zu entscheiden, welche Aktionen ausgelöst werden. Gemäß der Prinzipien der eventbasierten Verarbeitung und, um eine möglichst starke Entkopplung zu erreichen, sollte die Entscheidung, welche Aktionen ausgelöst werden, möglichst erst beim Empfänger gefällt werden.

**Anforderung 7 - Mehrere Aktionen durch ein Event:** Ein einziges von einer Quelle ausgelöstes Event soll es ermöglichen mehrere Aktionen auszuführen. Diese Aktionen können in der selben oder in unterschiedlichen Cloud-Umgebungen aufgerufen werden. Ein einziges von S3 ausgelöstes Event löst im Beispiel eine Funktion in der AWS Lambda Umgebung und eine in der Azure Functions Umgebung aus.

In ihren *Message Channel Decisions* nennen Hohpe und Woolf dies eine *one-to-many* Beziehung. Dies steht im Gegensatz zu einer *one-to-one* Beziehung, bei der jeder Aufrufer mit genau einer Applikation Daten teilt. Bei einer *one-to-many* Beziehung empfehlen sie einen Kanal nach dem *Publish/Subscribe Pattern* [HWB04, S. 109].

**Anforderung 8 - Verkettung von Events:** Eine von einem Event ausgelöste Aktion kann wiederum neue Events auslösen. Eine Verkettung von Events muss daher möglich sein. So wird durch das Hochladen eines neuen Fotos in den S3 Speicher zuerst die Bilderkennung und von dort aus durch ein weiteres Event die Funktion *Tags Writer* aufgerufen.

**Anforderung 9 - Vereinheitlichung von Event-Formaten:** Damit Events zwischen den Umgebungen unterschiedlicher Cloud-Provider ausgetauscht und verarbeitet werden können, müssen Events einen einheitlichen Aufbau besitzen. Hierzu kann zwischen einem der beiden folgenden Verfahren gewählt werden. Entweder muss es möglich sein, sie aus den providerabhängigen Formaten in ein einheitliches Format übersetzen zu können, oder es muss eine direkte Übersetzung der unterschiedlichen Formate ineinander stattfinden.

### 3.3 Nichtfunktionale Anforderungen

**Anforderung 10 - Skalierbarkeit und Verfügbarkeit:** Die Skalierbarkeit ist einer der Gründe um Applikationen in Serverless Umgebungen zu entwickeln. iRobot, ein weltweit führender Hersteller von Robotern für Endnutzer, beschreibt in seiner AWS Case Study [iRo] besonders die Skalierbarkeit der Serverless Cloud als ihren größten Nutzen. An einzelnen Tagen, beispielsweise vor Feiertagen, wurden von iRobot deutlich mehr Staubsaugerroboter verkauft, die sich mit ihrem Internet of Things (IoT) Backend verbinden. Die dadurch erzielten Lastspitzen konnten einfach von der Cloud-Infrastruktur abgefangen werden.

Da sich in diesen Umgebungen die Cloud-Provider vollständig um die Skalierung und Ausfallsicherheit kümmern und Softwareentwickler sich darauf verlassen können, dass ausreichend Ressourcen zur Verfügung stehen, sollte dieses Merkmal auch bei der Erweiterung auf eine Multi Cloud-Umgebung nicht verloren gehen. Daher muss bei jeder Schicht des zu erarbeitenden Softwarestacks darauf geachtet werden, dass eine automatisierte Skalierung stattfinden kann.

Neben der Skalierbarkeit nennt beispielsweise Bustle die Verfügbarkeit einer ihrer Hauptgründe, um auf eine Serverlose Architektur umzusteigen [Bus]. Bustle betreibt eine Nachrichten-, Unterhaltungs-, Lifestyle- und Modewebseite für Frauen. Diese Verfügbarkeit darf durch eine Multi-Cloud Lösung nicht verloren gehen. Insbesondere sollte die Nichtverfügbarkeit einer einzelnen Umgebung oder Komponente möglichst geringen Einfluss auf die anderen Cloud-Umgebungen haben.

**Anforderung 11 - Providerunabhängigkeit und Erweiterbarkeit:** Neben den Anforderungen, die identisch zu denen sind, die an eine providerabhängige Serverless Umgebung gestellt werden, unterscheidet sich das geforderte Softwaresystem in erster Linie durch die Providerunabhängigkeit.



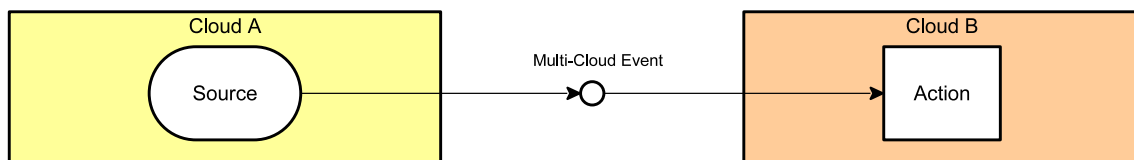
Durch die Vielzahl und Diversität der Cloud-Provider und Möglichkeiten zum Betrieb einer privaten Cloud scheint es nahezu unmöglich sämtliche Umgebungen vereinen zu können. Daher soll es in erster Linie möglich sein, durch eine gute Erweiterbarkeit des Konzeptes, weitere Cloud-Provider anbinden zu können. Diese Erweiterbarkeit kann beispielsweise durch die Nutzung von offenen und verbreiteten Standards erreicht werden.

**Anforderung 12 - Sicherheit:** In jedem Softwareprojekt sollte Sicherheit eine wichtige Rolle spielen. Gerade im Zusammenspiel mit sensiblen Daten oder sicherheitskritischen Funktionen kann der Einsatz von Software ansonsten drastische Probleme mit sich bringen. Autorisierung und Authentifizierung stellen wichtige Grundpfeiler eines Sicherheitskonzepts einer verteilten Software dar.



## 4 Konzeption

In Anforderung 1 wird als Ziel die Unterstützung eines Multi-Cloud Events genannt. Diese Events sollen vom Nutzer konfiguriert und vom Softwaresystem so ausgeführt werden, dass die Komplexität der Verarbeitung eines solchen Events vor dem Nutzer verborgen bleibt. Die Sicht eines Anwenders auf das Multi-Cloud Event ist in Abbildung 4.1 dargestellt. Die Darstellung bedient sich dafür der Block-Diagramm Notation der *Fundamental Modeling Concepts (FMC)* [Wen].



**Abbildung 4.1:** Model eines Multi-Cloud Events

Von einer Event-Quelle wird ein Event ausgelöst, das als sogenanntes *Multi-Cloud Event* an eine Funktion innerhalb einer zweiten Cloud geleitet wird. Anhand dieses Beispiels werden im Folgenden sowohl der Inhalt, beziehungsweise Aufbau eines Multi-Cloud Events, als auch der Aufbau der benötigten Infrastruktur dargestellt.

In ihrem Buch *Enterprise Integration Patterns - Designing, Building and Deploying Messaging Solutions* beschreiben Hohpe und Woolf den Aufbau und die einzelnen Komponenten eines Nachrichtensystems. Zur Definition eines Nachrichtensystems vergleichen sie diese mit einem Datenbanksystem. So, wie ein Administrator ein bestimmtes Schema für eine Datenbank vorgibt, genauso muss ein Administrator das Nachrichtensystem mit allen Nachrichtenkanälen definieren. Das Datenbanksystem kümmert sich daraufhin um die Persistenz der Daten, während das Nachrichtensystem sich um die erfolgreiche Zustellung der Nachrichten kümmert [HWB04, S. 14].

Ein solches Nachrichtensystem wird benötigt um Multi-Cloud Events von einer Cloud-Umgebung in eine oder mehrere andere Umgebungen weiterzuleiten. In den folgenden Abschnitten sollen daher anhand der von Hohpe und Woolf genannten Elemente konzeptionelle Entscheidungen zum Nachrichtensystem diskutiert und analysiert werden.

### 4.1 Nachrichtenformat

Um ein Event von einer Cloud-Umgebung in eine andere Cloud-Umgebung zu transferieren, muss es in einer sogenannten Nachricht verschickt werden. Hohpe und Woolf bezeichnen eine Nachricht als einen Datensatz, der von einem Nachrichtensystem durch einen Nachrichtenkanal übertragen werden kann [HWB04, S. 82 f.]. Dafür muss die Nachricht beim Sender in Bytecode gemarshalt und beim Empfänger wieder entmarshalt werden.

Nachrichten teilen sich nach Hohpe und Woolf in zwei Teile auf, den Header und den Body [HWB04, S. 82]. Während der Header alle wichtigen Informationen enthält, die für das Nachrichtensystem zur Übertragung wichtig sind, befinden sich im Body die eigentlichen Nutzdaten. Die Nutzdaten eines Events sind dabei schnell identifiziert. Sie bestehen aus den Informationen des providerspezifischen Event, beziehungsweise im Fall eines, wie in Anforderung 3 gefordert, vom Benutzer erzeugten Events, dessen angegebene Nutzdaten. Komplizierter wird es, die benötigten Informationen für den Header zu identifizieren.

Eine wichtige Entscheidung, die anhand der Headerinformationen getroffen werden muss, ist, welche Nachrichten an welche Komponenten oder Endpunkte weitergeleitet werden müssen oder dürfen. In Anforderung 5 werden beispielsweise Events erwähnt, die aufgrund ihres datenschutzrechtlichen Inhalts nicht an andere Cloud-Umgebungen weitergeleitet werden dürfen, während es für andere Events essentiell ist, dass sie in alle anderen an das System angebotenen Umgebungen geleitet werden.

Um zu entscheiden, welches Event in welche Umgebung geleitet werden soll und welche Aktionen aufgrund des Events ausgeführt werden, muss es möglich sein zu verstehen, um was für ein Event es sich handelt. Zur Erfüllung dieses Ziels müsste entweder jede Umgebung mit jedem Format umgehen können, oder es muss ein einheitliches Format zur Darstellung von Events verwendet werden. Providerspezifische Formate müssen dann in dieses einheitliche Format transformiert werden.

Als providerunabhängiges Format eignet sich das in Abschnitt 2.3 vorgestellte Format CloudEvents. CloudEvents nennt es selbst eines der wichtigen Design Goals in seinen Attributen ein minimales Set an Informationen zu beinhalten, mit deren Hilfe das Event an die entsprechenden Komponenten weitergeleitet werden kann [CNCb].

Eine Komponente in einem Nachrichtensystem, die eine Nachricht anhand eines bestimmten Filters entweder weiterleitet oder nicht, nennen Hohpe und Woolf einen *Message Filter* [HWB04, S. 217 f.].

## 4.2 Nachrichtenkanal

In diesem Kapitel werden verschiedene Konzepte zum Aufbau des sogenannten Nachrichtenkanals entworfen. Unter Zuhilfenahme der Anforderungen werden diese Konzepte anschließend analysiert und ihre Vor- und Nachteile untersucht. Ein Nachrichtenkanal entspricht dabei den Komponenten, die eine Verbindung zwischen Sender und Empfänger ermöglichen.

Zur übersichtlichen Darstellung der einzelnen Konzepte wird das Patternformat verwendet, wie es Fehling et. al in ihrem Buch *Cloud-Computing Patterns* beschreiben [FLR+14, S. 9 ff.]. Ein Patternformat ermöglicht eine detaillierte und gleichzeitig strukturierte Darstellung eines Lösungsansatzes auf ein Problem. Die Idee eines Pattern und der damit verbundenen Patternsprache stammt aus dem Jahr 1977 von Christopher Alexander. Er nutzte eine Patternsprache um wiederkehrende Probleme und deren Lösungsansätze in der Architektur zu beschreiben [AIS77]. Diese Patternsprache lässt sich allerdings nicht nur auf Gebäude, Städte und andere Konstruktionen anwenden, sondern ebenso auf die Softwarearchitektur.

Jedes Pattern besteht aus einem Namen und einer kurzen und prägnanten Beschreibung des Lösungsansatzes in einem grauen Kasten. Anschließend wird es in den Abschnitten *Context*, *Solution* und *Result* weiter ausgeführt. Im *Context* wird beschrieben, welche äußeren Faktoren zu diesem Konzept führten und in welchen Szenarien es eingesetzt werden kann. Die *Solution* skizziert möglichst kurz das Konzept, während im Abschnitt *Result* detaillierter auf die Vor- und Nachteile und die damit einhergehende Erfüllung der Anforderungen eingegangen wird. Leichte Abwandlungen des ursprünglichen Patterns werden, falls vorhanden, im Abschnitt *Variation* diskutiert. Am Ende eines Pattern werden gegebenenfalls existierende Ansätze und Applikationen beschrieben, mit deren Hilfe oder auf deren Grundlage das entsprechende Konzept umgesetzt werden kann.

#### 4.2.1 Direkter Aufruf

Aktionen werden von einer Funktion aufgerufen, die das Event abgefangen hat.

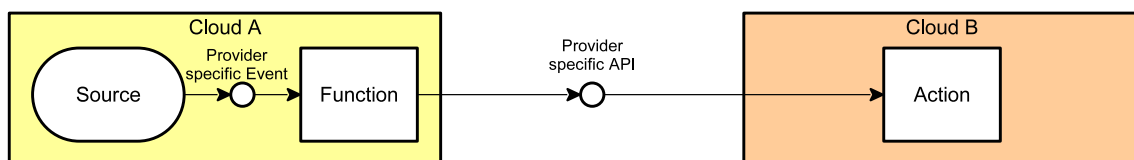


Abbildung 4.2: Direkter Aufruf des Empfängers über eine providerspezifische Schnittstelle

#### Context

Besonders in kleinen Testszenarien, oder einem kleinen Prototyp, bei dem mit geringem Aufwand Applikationen über Cloud-Umgebungen hinweg kommunizieren müssen, ist eine einfache Lösung gefragt. Daher soll auf unnötige Komponenten im System verzichtet werden und so ein einfaches Deployment ermöglicht werden.

#### Solution

Das erste Konzept zur Realisierung der Weiterleitung eines Multi-Cloud Events ist der direkte Aufruf einer entfernten Funktion. Dazu muss zuerst das Event innerhalb der Cloud-Umgebung, in der das Ereignis eintritt, abgefangen werden. Dies kann mit einer FaaS stattfinden. Innerhalb dieser Funktion kann das Event weiter verarbeitet werden, beispielsweise in ein einheitliches Format umgewandelt oder auf die individuellen Bedürfnisse der Empfänger-Cloud angepasst, um Anforderung 9 zu erfüllen.

Durch das Auslagern der Weiterleitungsfunktionalität in eine Programmbibliothek können vom Benutzer selbst erzeugte Events geworfen werden, ohne dass dieser mit den Implementierungsdetails vertraut sein muss. Damit kann Anforderung 3 an benutzerdefinierte Events erfüllt werden.

Der Aufruf einer Funktion muss durch eine vom Provider bereitgestellte Möglichkeit stattfinden. Dies kann, wie in Abbildung 4.2 durch eine providerspezifische API, oder durch Standards, wie HTTP implementiert sein. Dadurch lässt sich mit diesem Konzept auch die Auslösung von mehreren Aktionen auf ein Ereignis einfach implementieren. Statt eines einzigen Aufrufs müssen von der Funktion mehrere Aufrufe durchgeführt werden.

### Result

Bei der Weiterleitung eines Events stellt sich die prinzipielle Frage, an welcher Stelle entschieden werden soll, welche Events an welche Umgebungen, beziehungsweise Funktionen weitergeleitet werden sollen. Dabei stehen in diesem Konzept mehrere Möglichkeiten bereit. Zuerst muss durch entsprechende Konfiguration festgelegt werden, welche Events abgefangen werden sollen. Da dieses Abfangen providerspezifische Anpassungen erfordert, kann auch nicht allgemeingültig festgelegt werden, welche Möglichkeiten der Filterung hier zur Verfügung stehen. Eine Filterung kann allerdings auch in der Funktion stattfinden, mit der das Event abgefangen wird. Innerhalb dieser gibt es nahezu keine Grenzen der Filterung. Die Funktionalität muss allerdings gegebenenfalls an die Provider und ihre zur Verfügung gestellten Laufzeitumgebungen angepasst werden.

Gleichzeitig tritt durch die oben genannten Möglichkeiten ein weiteres Problem auf. Die Umgebung, in der ein Ereignis eintritt, muss wissen, welche Aktion aufgrund dessen gestartet werden muss. Dadurch wird ein Teil der Funktionslogik aus der Umgebung der Aktion gelöst und in die Umgebung des Ereignisses verschoben. Bei einer Änderung dieses Verhaltens müssen dementsprechend in mehreren Umgebungen Anpassungen durchgeführt werden. Während dies für kleine Applikationen keine allzu großen Nachteile darstellen sollte, kann dies bei komplexeren Applikationen zu einer stärkeren Kopplung führen, die durch Eventbasierte Kommunikation eigentlich verringert werden sollte. Die Entscheidung über die aus einem resultierenden Event der Quelle des Events zu überlassen widerspricht somit dem Prinzip der Entkopplung und damit den Grundlagen der eventbasierten Verarbeitung [EN10, S. 33 ff.]

Alternativ zur Verarbeitung und Filterung eines Events innerhalb der Cloud, in der das Event entstanden ist, kann es auch erst in der Ziel-Cloud verarbeitet werden. Dies impliziert, dass sämtliche Events in alle anderen Cloud-Umgebungen der Applikation weitergeleitet und dort verarbeitet werden müssen. Durch diese weitere Verarbeitung, werden zusätzliche Komponenten benötigt, die in folgenden Konzepten genauer betrachtet werden.

Als letzte Möglichkeit, ist es denkbar, dass erst die eigentlich aufgerufene Funktion für sich interessante Events herausfiltert. Beim Aufruf der Funktion bei einem Event, auf das die entsprechende Funktion nicht reagiert, bricht sie die Ausführung einfach ab. Durch dieses Vorgehen kommt es allerdings zu vielen unnötigen Aufrufen, die Overhead erzeugen.

Zudem ergibt sich durch den direkten Aufruf ein weiterer Nachteil. Da die Funktion den kompletten Messaging Channel bis zum Empfänger abbildet, ist sie auch für Konsistenz des Events im Fehlerfall zuständig, wie sie in Anforderung 4 beschrieben ist. Bei einer Funktion im Sinne der FaaS handelt es sich allerdings um recht kurzlebige und zustandslose Softwareausführungen. Falls eine Störung des Empfängers für mehrere Minuten, oder gar Stunden auftritt, müsste das Event demnach zwischengespeichert und zu einem späteren Zeitpunkt erneut verarbeitet werden. Besonders im Falle eines Events, dass an mehrere Ziele gesendet werden soll, muss zusätzlich beachtet werden, dass Events bei Nichterreichbarkeit eines Zieles an das andere Ziel nicht mehrfach weitergeleitet

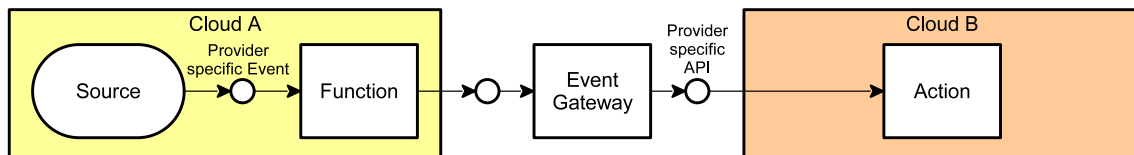
werden. Eine solche Software in einer Serverless Applikation zu implementieren wäre zwar möglich, für diesen zusätzlichen Verwaltungsaufwand stehen allerdings bereits vorgefertigte Komponenten als Middleware bereit, die in den folgenden Konzepten näher betrachtet werden.

Besonders bei den nichtfunktionalen Anforderungen spielt dieses Konzept seine Vorteile dagegen voll aus. Sowohl die Performanz, als auch die Skalierbarkeit, wie auch die Kosten der Lösung lassen sich hier durch das Einsparen von Komponenten verbessern. Durch den Einsatz von Serverless Komponenten wird die Skalierung bereits vom Cloud-Provider übernommen. Bei den Kosten schlägt lediglich der Aufruf der zusätzlichen Funktion zum Abfangen und Weiterleiten des Events zu buche.

Anforderung 11, die Providerunabhängigkeit ist durch die Verwendung von offenen Schnittstellen, wie beispielsweise HTTP, in der Regel gegeben. Da aber jeder Provider gegebenenfalls lediglich proprietäre Schnittstellen und Laufzeitumgebungen zur Verfügung stellt, kann es vorkommen, dass für jede Konstellation zwischen zwei Providern eigene Softwarekomponenten geschrieben werden müssen. Dies wird besonders im Zusammenspiel mit der Anforderung an die Sicherheit deutlich. Jeder Provider bietet eine eigene Authentifizierungsmöglichkeit zum Aufruf von Funktionen an. Je nach Provider und dessen verwendeter Schnittstelle muss diese Authentifizierung demnach anders implementiert werden.

#### 4.2.2 Zentrales Gateway

Alle Events werden an ein zentrales Gateway geleitet. Von dort wird entschieden, welche Aktionen ausgeführt werden.



**Abbildung 4.3:** Eventbasierte Kommunikation mit einem zentralen Gateway

#### Context

Durch steigende Komplexität der zu entwickelnden Applikation und die daraus resultierende höhere Anzahl der zu verarbeitenden Events, ergeben sich auch höhere Ansprüche an die zu Grunde liegende Infrastruktur. Besonders wird dies bei der Konsistenz von Events im Fehlerfall deutlich. Bei einem kurzfristigen Ausfall oder bei fehlender Erreichbarkeit durch Netzwerkfehler einer einzelnen Cloud-Umgebung müssen Events zeitverzögert erneut gesendet und somit persistent gehalten werden. Einfache FaaS-Funktionen kommen dabei an ihre Grenzen.

### **Solution**

Durch eine Erweiterung des ersten Konzepts entsteht das Konzept des zentralen Gateways, wie es in Abbildung 4.3 dargestellt ist. Identisch zum ersten Konzept wird auch in diesem das Event durch eine Funktion abgefangen. Die weitere Verarbeitung unterscheidet sich allerdings stark. Statt das Event direkt an die Empfänger-Cloud zu schicken, wird das Event an ein zentrales Gateway geleitet.

Dieses Gateway entscheidet dann darüber an welche Umgebungen das Event weitergeleitet wird, beziehungsweise welche Funktionen ausgeführt werden sollen. Dadurch werden diese Entscheidungen nicht mehr über die Umgebungen verteilt getroffen, sondern zentral an einem Ort.

### **Result**

Für die Verbindung sowohl zwischen den Event-Auslösern und dem Gateway, als auch zwischen Gateway und den Empfängern kann es hierbei je nach Cloud-Provider notwendig werden, auf unterschiedliche Protokolle und Arten des Verbindungsaufbaus zurückgreifen zu müssen. Um einen schnellen Versand eines Events zu gewährleisten bietet es sich an, eine persistente Netzwerkverbindung zwischen Gateway und Cloud-Infrastruktur aufzubauen. Besonders beim Auftreten von vielen Events in einem kurzen Zeitraum kann ein einzelner Verbindungsaufbau bei jedem Event zu großen Latenzen führen. Dieser Effekt wird dann verstärkt, wenn das Gateway und die Cloud-Infrastruktur über das Internet hinweg kommunizieren müssen.

Sollte eine persistente Verbindung zur Cloud-Infrastruktur vom Provider nicht vorgesehen sein, so bleibt weiterhin die Möglichkeit eine persistente Verbindung zu einer Komponente innerhalb der Provider-Cloud aufzubauen und von dieser Komponente aus, die meist eine geringere Latenz zur Infrastruktur hat, weitere Aktionen auszuführen.

Durch dieses Konzept ist nicht mehr die Funktion, die das Event abfängt für die erfolgreiche Zustellung verantwortlich. Lediglich bei einem fehlerhaften Zustellversuch des Events an das Gateway, muss ein Verhalten definiert werden. Dies kann beispielsweise ein Eintrag in einem Fehlerlog sein, mit dessen Hilfe eine spätere Aufarbeitung des Fehlerzustandes durchgeführt werden kann. In allen weiteren Fehlerfällen ist das Gateway für weitere Zustellversuche verantwortlich. Hierauf muss bei der Auswahl eines Gateways besonders geachtet werden.

Um Sicherheitsanforderung 12 zu erfüllen, muss sowohl die Kommunikation zwischen Event-Auslöser und Gateway, als auch die Kommunikation zwischen Gateway und Funktion abgesichert sein. Dies umfasst in der Regel eine verschlüsselte und authentifizierte Verbindung. Insbesondere bei Kommunikation zwischen mehreren Cloud-Umgebungen ist diese Anforderung von besonderer Priorität, da Verbindungen über das Internet hinweg aufgebaut werden.

Zur Bereitstellung eines Gateways bestehen mehrere Möglichkeiten. Es kann entweder von einem Provider, oder vom Applikationsersteller selbst betrieben werden. Einige Gateway-Entwickler betreiben auch Instanzen davon, die genutzt werden können. Dies ermöglicht die Nutzung der entsprechenden Software, ohne weiteren Aufwand, schränkt aber gegebenenfalls auch ein. Beispielsweise kann nur die vom Anbieter zur Verfügung gestellte Version genutzt werden.



Die Verwendung einer selbst betriebenen Instanz erhöht allerdings auch immer den Verwaltungsaufwand. Da ein zentrales Gateway einen Single-Point-of-Failure darstellt, muss diese Gateway sorgfältig ausgewählt und beim Betrieb auf eine hohe Ausfallsicherheit geachtet werden.

### Variation

Statt einem Gateway, das außerhalb der verwendeten Cloud-Umgebungen ausgeführt wird, kann dies auch innerhalb einer dieser Umgebungen betrieben werden. Insbesondere für Applikationen, bei denen ein Großteil der Ressourcen innerhalb dieser Umgebung genutzt wird und nur vereinzelt Events mit anderen Cloud-Umgebungen getauscht werden müssen, kann dies Vorteile bieten.

### Known Use

**Apache Kafka** bezeichnet sich auf der Dokumentationswebseite selbst als *Distributed Streaming Platform* [Apab]. Nachrichten werden in Kafka als ein Stream angesehen, der von mehreren Consumern verarbeitet werden kann. Besonderen Wert legt Kafka dabei zum einen auf die korrekte Abarbeitung der Nachrichten, in der Reihenfolge ihres Eingangs, als auch auf die Persistenz der Nachrichten. Diese werden für bestimmte Zeit auf dem Speicher vorgehalten und können somit, beispielsweise nach Ausfällen oder bei Fehlern, erneut vom Consumer angefragt werden.

**NATS** ist ein in der Programmiersprache Go geschriebenes, laut eigener Dokumentationswebseite leichtgewichtiges, performantes und belastbares Nachrichtensystem [Cloc]. Seit März 2018 wird es von der Cloud Native Computing Foundation (CNCF) im *Incubating-Level* als Projekt gehostet [Eva18]. Eine neutrale und providerunabhängige Weiterentwicklung sowie eine breite Unterstützung und Kompatibilität zu weiteren Projekten der CNCF scheinen daher gesichert.

Als Nachrichtenmodell unterstützt NATS sowohl Publish/Subscribe und Request Reply, als auch Queueing. Neben dem NATS Server steht auch eine Vielzahl an Client-Bibliotheken in unterschiedlichen Sprachen zur Verfügung.<sup>1</sup> Durch das einfache, text-basierte Protokoll können allerdings auch ohne großen Aufwand Clients ohne bestehende Bibliothek entwickelt werden. Durch die Erweiterung NATS Streaming arbeitet es mit *log based streaming*. Dadurch können Nachrichten auch zugestellt werden, wenn ein Client, nachdem er offline war, wieder online geht.

Synadia, die ursprüngliche Entwicklerfirma hinter NATS, stellt eine gehostete Version bereit, die mit Beschränkungen in einer kostenlosen Version oder in kostenpflichtigen Abonnements genutzt werden kann.<sup>2</sup>

Mit dem **Serverless Event Gateway** stellt auch das Serverless Framework ein eigenes Gateway zur Verfügung. Dabei hat dieses, identisch zum eigentlichen Framework, den Anspruch weitestgehend providerunabhängig zu sein. Das Serverless Event Gateway wurde bereits im Abschnitt 2.4 beschrieben.

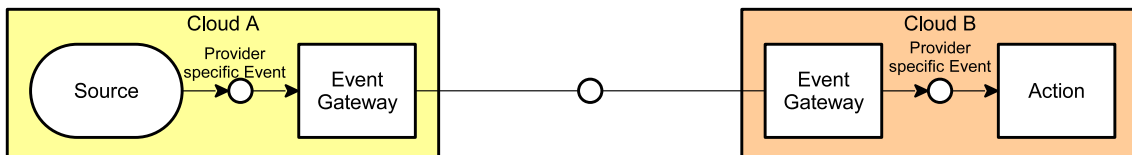
---

<sup>1</sup><https://www.nats.io/download/>

<sup>2</sup><https://www.nats.cloud/>

### 4.2.3 Dezentrale Gateways

Gateways in verschiedenen Umgebungen teilen ihre Events miteinander. Jedes Gateway entscheidet welche Aktionen innerhalb der eigenen Umgebung ausgeführt werden.



**Abbildung 4.4:** Eventbasierte Kommunikation mit dezentralen Gateways

#### Context

Ebenso wie im vorherigen Konzept nimmt sich das Konzept dezentraler Gateways auch der Probleme an, die ein direkter Aufruf von Funktionen über Providergrenzen hinweg mit sich bringt. Durch einen verteilten Ansatz kann die zentralisierte Ausführungslogik der Aktionen auf Events in die entsprechenden Cloud-Umgebungen zurückgeholt werden. Insbesondere bei Applikationen, bei denen viele Aktionen auf Events auch innerhalb der eigenen Umgebung ausgeführt werden, kann dieses Konzept einem zentralen Gateway vorgezogen werden.

#### Solution

Das Konzept der dezentralen Gateways, das in Abbildung 4.4 dargestellt ist, ist ein ebenfalls von dem Konzept der direkten Aufrufe weiterentwickeltes Konzept. Statt eines einzelnen Gateways, das für alle eingesetzten Umgebungen zuständig ist, wird in jeder Umgebung ein eigenes Gateway betrieben. Die Umgebungen spiegeln sämtliche aufgetretenen Events an alle anderen Gateways, die wiederum die Ausführungslogik ihrer Funktionen kennen. Dadurch sitzt die Ausführungslogik direkt in der entsprechenden Umgebung.

Es können providerspezifische Angebote genutzt und für jede Cloud-Umgebung ein unabhängiges Gateway entwickelt werden. Durch diese Vorgehensweise ist es möglich, für jede Umgebung eine optimierte Softwareumgebung zu betreiben, die die gewünschte Funktionalität abbildet. Damit kann eine vollständig cloud-native Umgebung aufgebaut werden, wodurch die Bereitstellung komplett vom Cloud-Provider durchgeführt wird und beispielsweise Ausfallsicherheit oder Skalierung von ihm übernommen werden. Bei diesem Ansatz kann auch eine vom Provider angedachte und bereitgestellte Möglichkeit zur eventbasierten Multi-Cloud Kommunikation verwendet werden. Insbesondere für selbst verwaltete und betriebene FaaS-Umgebungen ist dieser Ansatz gut umsetzbar.

## Result

Während die Anforderungen an das Abfangen eines Events und das Auslösen eines benutzerdefinierten Events identisch gelöst werden können wie im ersten Konzept, stellt sich bei diesem Konzept die Verteilung der Events als zentrale Schwierigkeit heraus. Hierbei ist sowohl eine standardisierte Schnittstelle, als auch ein standardisiertes Event-Format sinnvoll, damit komplizierte Umwandlungen vermieden werden können und Filterung über Umgebungsgrenzen hinweg einheitlich stattfinden kann. Außerdem muss entweder jede Cloud-Umgebung alle anderen Umgebungen kennen und ihnen die entsprechenden Events zukommen lassen, oder es muss ein Routing zwischen den Umgebungen stattfinden. Je nach Anzahl verwendeter Cloud-Umgebungen sollte es allerdings ein überschaubarer Verwaltungsaufwand sein, Events in mehrere Umgebungen zu verteilen.

Für die Datenhaltung der Events ist jede Gateway-Implementierung selbst verantwortlich. Bis zur Zustellung eines Events an jedes andere Gateway, muss dieses gespeichert werden und es müssen wiederholte Sendungszustellungen im Fehlerfall durchgeführt werden. Sollte eine Zustellung nach einer definierten Zeit nicht möglich sein, so muss eine Fehlermeldung geloggt werden und gegebenenfalls weitere Maßnahmen durchgeführt werden.

## Variation

Neben der Möglichkeit in jeder Cloud-Umgebung ein eigenes, angepasstes Gateway zu betreiben, kann auch eine Implementierung des selben Gateways in allen Cloud-Umgebungen ausgeführt werden. Durch diese Vorgehensweise kann beispielsweise die proprietäre Synchronisierung einer Gateway-Applikation genutzt werden. Mithilfe von Containertechnologie ist es vergleichsweise einfach dieses Gateway in weiteren Cloud-Umgebungen zu erweitern, solange sie diese Technologie unterstützen. Bei Cloud-Umgebungen, die keine Möglichkeit bieten die entsprechende Gateway-Implementierung zu betreiben, muss dann allerdings ein individuell angepasstes Konzept erarbeitet werden, wie diese Umgebung an die bestehende Multi-Cloud-Umgebung angeschlossen werden kann. Durch proprietäre Schnittstellen zwischen den Gateways kann dies erschwert sein.

## Known Use

**Azure Event Grid** ist ein von Microsoft entwickelter und in der Azure Cloud-Computing-Plattform betriebener Service zum Ereignisrouting. Dieser wurde bereits in Abschnitt 2.4 beschrieben.

Amazon bietet mit seinem **Simple Notification Service (SNS)** die Möglichkeit Applikationen, wie Microservices, Verteilte Systeme oder Serverlose Anwendungen zu entkoppeln [Amaa]. Dazu stellt es Nachrichtenkanäle nach dem Publish/Subscribe Prinzip zur Verfügung. Subscriber können sowohl Simple Queueing Service (SQS) Warteschlangen, Lambda Funktionen oder HTTP(S) Verbindungen sein. Auch Endbenutzer können mit SNS per E-Mail oder SMS benachrichtigt werden.

SNS stellt dabei einen Full-Managed-Service dar. Das komplette Management, einschließlich aller Ressourcen- und Softwareverwaltung wird von Amazon übernommen. Dadurch fügt sich AWS SNS perfekt in eine Serverless Umgebung ein.

Für den Betrieb in Multi-Cloud-Umgebungen und providerunabhängige Schnittstellen stellt SNS außer der bereits erwähnten HTTP-Schnittstelle bei Ankunft einer Nachricht keine offenen Schnittstellen zur Verfügung. Weder ein HTTP-Endpoint, noch eine Unterstützung von CloudEvents oder einem ähnlichen Format werden angeboten.

Auch die im vorherigen Konzept vorgestellten Gateways **NATS**, **Kafka** und das **Serverless Event Gateway** lassen sich durch einen Clusterbetrieb in eine Variation der dezentralen Gateways umwandeln. NATS beispielsweise ermöglicht es mehrere Instanzen zu einem Cluster zu verbinden [Clob]. Innerhalb dieses Clusters werden Nachrichten gespiegelt und somit in allen Instanzen zugänglich gemacht. Dadurch ist es möglich, wie in der Variation beschrieben, in jeder Cloud-Umgebung eine Instanz des selben Gateways zu betreiben. Eine separate Verteilung der Events ist dabei nicht mehr nötig. Dieser Clusterbetrieb ist in ähnlicher Form auch in Kafka [Amaa] und dem Serverless Event Gateway [Serc] vorhanden.

### 4.2.4 Hierarchische Struktur von Gateways

In jeder Umgebung wird ein eigenes Gateway betrieben. Ein zentrales Gateway verteilt alle Events an die dezentralen Gateways.

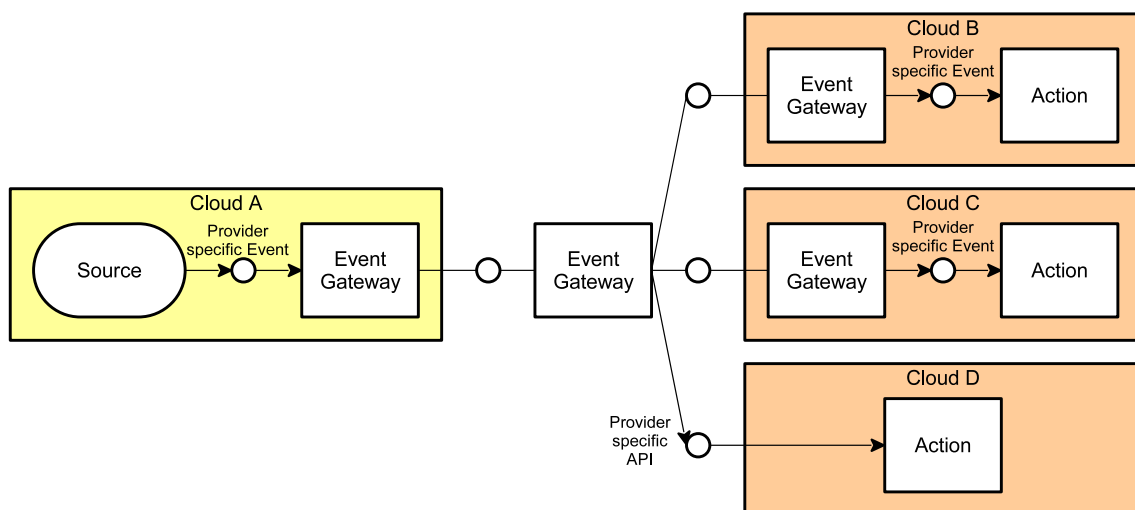


Abbildung 4.5: Dezentrale Gateways mit einem zentralen Gateway

### Context

Während der Austausch von Events in Multi-Cloud-Umgebungen mit zwei Umgebungen einen vergleichsweise einfachen Austausch von Events ermöglicht, wird dies in Multi-Cloud-Umgebungen, die aus drei oder mehr einzelnen Umgebungen bestehen deutlich komplexer. Jede Umgebung muss eine Schnittstelle zu jeder anderen Umgebung der Applikation kennen und beim Auftreten eines Events dieses weiterleiten. Im Falle einer Änderung dieser Schnittstelle, beispielsweise, wenn diese

unter einer neuen Adresse erreichbar ist, muss diese Änderung an alle Umgebungen verteilt werden. Ebenso muss jede Umgebung selbst regeln, wie sie mit nicht erreichbaren Umgebungen umgeht. Sobald eine Umgebung wieder hergestellt wurde, müssen Events von unterschiedlichen, verteilten Quellen zusammengetragen werden.

### **Solution**

Durch ein zentrales Gateway, wie es in Abbildung 4.6 dargestellt ist, kann diesen Problemen entgegengewirkt werden. Statt ein Event direkt an alle Umgebungen weiterzuleiten, wird ein aufgetretenes Event, nur an das zentrale Gateway weitergeleitet. Die Aufgabe der internen Gateways besteht weiterhin darin, auf Events mit entsprechenden Aktionen zu reagieren.

Hohpe und Wolf sprechen bei dieser Architektur von einer *Hub-and-Spoke Architecture* [HWB04, S. 288]. Das zentrale Gateway bildet den Hub, während die Verbindungen zu den dezentralen Gateways die Speichen (*englisch: Spoke*) bilden. Diese steht im Gegensatz zu einer *Point-to-Point Architecture*, bei der jeder Kommunikationspunkt mit jedem anderen verbunden ist.

### **Result**

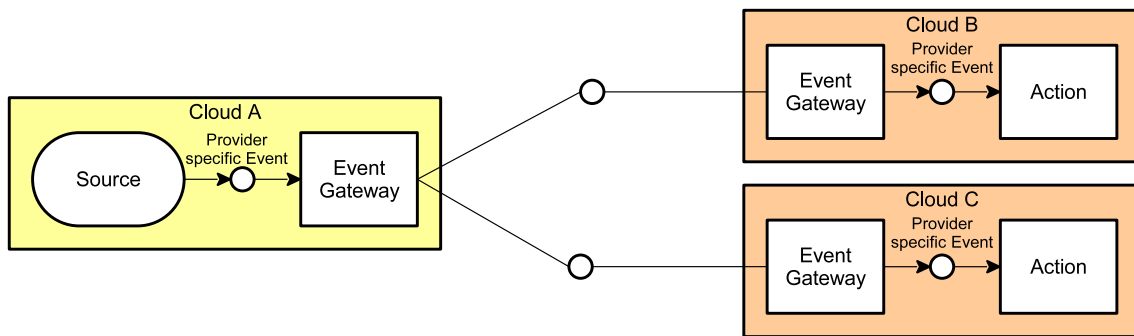
Durch dieses Konzept wird die aufwendige Verteilung der Events zentralisiert, während die Entscheidung darüber, welches Event welche Aktionen auslöst weiterhin in den jeweiligen Umgebungen liegt. Insbesondere, wenn mehrere Cloud-Umgebungen benötigt werden oder Umgebungen hinzugefügt, beziehungsweise entfernt werden, kann dieses Konzept seine Vorteile ausspielen. Fehlerbehebungen und die Wiederholung des Versands von Events wird in einer einzigen Komponente durchgeführt.

Gleichzeitig muss beachtet werden, dass die Einführung eines zentralen Gateways auch die Nachteile von diesem mit sich bringt. So kann ein Ausfall dieses Gateways den Stillstand der Kommunikation zwischen den Cloud-Umgebungen zur Folge haben. Hohpe und Wolf warnen explizit vor der Gefahr eines Bottlenecks durch einen zentralen Hub, geben aber auch Lösungsansätze [HWB04, S. 288]. Durch eine mehrfache Instanziierung des zentralen Gateways kann die Gefahr der Überlastung verringert werden. Diese mehrfache Instanziierung kann beispielsweise durch den Zusammenschluss mehrerer Instanzen eines Gateways zu einem Cluster realisiert werden.

Durch eine Erweiterung der Funktionalität des zentralen Gateways ist es auch möglich, Umgebungen, die bisher über keine Implementierung eines dezentralen Gateways verfügen, an das Gesamtsystem zu koppeln. Wie in Abbildung 4.5 dargestellt, kann ein zentrales Gateway auch einen direkten Aufruf einer Funktion durchführen. Durch die Unterstützung von Standards, wie HTTP, kann eine breite Masse an Umgebungen hiermit bereits abgedeckt werden. Die Logik der Ausführung von Aktionen bleibt mit diesem Ansatz in den meisten Fällen dezentral geregelt. Lediglich bei nicht unterstützten Umgebungen bleibt die Ausführungslogik zentral gehalten. Auf diese Weise können Vorteile eines zentralen und der dezentralen Gateways kombiniert werden.

## Variation

In jeder Umgebung wird ein eigenes Gateway betrieben. Eines dieser Gateways wird als Hauptgateway definiert und ist für die Verteilung von Events zuständig.



**Abbildung 4.6:** Dezentrale Gateways mit einem Hauptgateway als zentrale Verwaltung

Insbesondere für Applikationen, die überwiegend in einer Cloud-Umgebung implementiert wurden und nur für vereinzelte Aufrufe auf andere Umgebungen zugreifen, kann dieser Applikationsaufbau auch im Aufbau der Event-Infrastruktur wiedergegeben werden. Anstatt, dass ein externes zentrales Gateway betrieben wird, wird in dieser Variation eines der dezentralen Gateways als zentrales definiert. Dieses Hauptgateway ist, wie in Abbildung 4.6 zu sehen, für die Verteilung an alle anderen Gateways zuständig.

Durch dieses Vorgehen entsteht auch in diesem Fall eine hierarchische Struktur, durch die es wieder zu einem Single-Point-of-Failure kommen kann. Dieser kann allerdings beispielsweise dadurch umgangen werden, dass im Fehlerfall ein zweites Gateway als Hauptgateway definiert wird. Von diesem findet dann, solange das erste Gateway nicht erreichbar ist, die Event-Verteilung statt. Je nach Aufbau der Applikation muss allerdings auch beachtet werden, dass es bei einer Abgrenzung der Cloud-Umgebung, in der der überwiegende Teil der Applikationsausführung stattfindet, gegebenenfalls auch sinnvoller sein kann, eine Event-Verteilung auszusetzen, bis der Fehlerzustand wieder behoben wurde.

## 5 Prototypische Implementierung

Nachdem in den vorherigen Kapiteln die Anforderungen definiert und konzeptionelle Entwürfe erarbeitet wurden, folgen in diesem Kapitel konkrete prototypische Implementierungsmöglichkeiten. Sie stellen Möglichkeiten dar, wie Multi-Cloud Events realisiert werden können. Zur Umsetzung und zum späteren Anforderungsabgleich der Funktionalität der Prototypen soll das Referenzszenario herangezogen werden, wie es in Kapitel 3 beschrieben wird. Die Beschreibungen der einzelnen Prototypen sind im Folgenden in die Abschnitte Umsetzung und Anforderungsabgleich unterteilt. Die Umsetzung beschreibt die eingesetzten Komponenten und ihr Zusammenspiel, während im Anforderungsabgleich die Erfüllung der einzelnen Anforderungen des Prototyps überprüft werden. In Anlehnung an die Patternsprache, die in Abschnitt 4.2 Verwendung findet, werden im Anschluss Variationen vorgestellt, falls diese interessant sind. Den Abschluss bildet ein kurzes Fazit in dem auch auf die Use-Cases eingegangen wird, in denen der Prototyp geeignet ist.

Aufgrund der hohen Verbreitung bei FaaS-Plattformen, insbesondere in den beiden ausgewählten Cloud-Plattformen, wurde sowohl für die Beispielapplikation, als auch für weitere Komponenten des Softwaresystems auf JavaScript und die Laufzeitumgebung Node.js gesetzt.

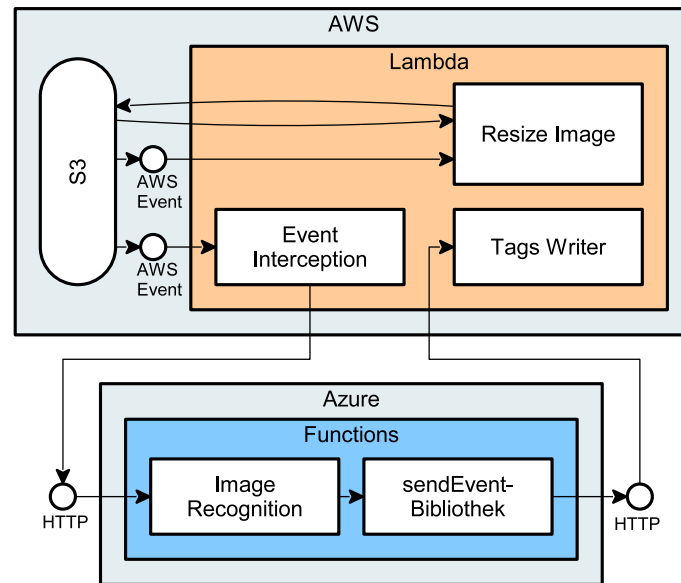
### 5.1 Direkter Aufruf

Der erste Prototyp basiert auf dem Konzept *Direkter Aufruf*. Dieser bedient sich lediglich Funktionen der FaaS-Umgebung und soll ansonsten ohne zusätzliche Software-Komponenten auskommen. Dadurch soll ein unkomplizierter Einsatz, ohne Verwaltungsaufwand ermöglicht werden.

#### 5.1.1 Umsetzung

In Abbildung 5.1 ist ein Ausschnitt des Aufbaus des Prototyps dargestellt. Alle Komponenten, die keinen direkten Einfluss auf die Multi-Cloud Events haben, wurden zur übersichtlicheren Darstellung weggelassen. Im Vergleich zur ursprünglichen Abbildung 3.2 wurde in AWS die Lambda Funktion *Event Interception* und in Azure die *sendEvent-Bibliothek* hinzugefügt. Auf diese Komponenten und deren Verhalten wird im Folgenden eingegangen.

Für die Lambda Funktion *Event Interception* wurden als Auslöser alle Events definiert, die von ihr abgefangen werden sollen. Hierbei wird eine Einschränkung der AWS Umgebung deutlich. Über die von einem S3 Speicher ausgelösten Events ist es nicht möglich mit mehreren Aktionen auf das selbe Ereignis zu reagieren. Amazon selbst empfiehlt hierfür den Einsatz von SNS zur weiteren Verteilung von Events [Nag15].



**Abbildung 5.1:** Prototyp Eventbasierter Kommunikation in einer Multi-Cloud-Umgebung durch direkte Aufrufe.

In der Beispielapplikation müssen die beiden Funktionen für die Bildkompression, als auch die Bildanalyse beim Upload eines Fotos auf ein Event reagieren. Für dieses Problem bestehen zwei Lösungsansätze. Entweder muss die Abfang-Funktion nicht nur das Event weiterleiten und die entfernte Funktion aufrufen, oder die beiden Events, durch die die Funktionen ausgelöst werden, müssen sich unterscheiden. Da die Bildkompression auf der vom Benutzer hochgeladenen Originaldatei und die Bildanalyse auf der daraus resultierenden komprimierten Datei ausgeführt wird, besteht die Möglichkeit auf unterschiedliche Events zu reagieren. Die beiden Events unterscheiden sich, wie in Listing 5.1 aufgeführt, in ihrer Präfix-Regel. Während die Bildkomprimierung auf alle Dateien, die in den Ordner `upload` geladen werden reagiert, reagiert die Abfang-Funktion auf Bilder im Ordner `compressed`. Da dieses Vorgehen die vom Provider bereitgestellten Mittel besser ausnutzt, nutzt der Prototyp diesen Weg.

Nachdem das Event mit dem Aufruf der Funktion erfolgreich abgefangen wurde, wird es zuerst in ein einheitliches Format umgewandelt, um anschließend weitergeleitet zu werden. Die Umwandlung in ein einheitliches Format ist dabei optional, da Routing und Filterung lediglich innerhalb dieser Funktion und nicht in weiteren Komponenten stattfindet. Der Endpunkt könnte auch die gemarshalten Rohdaten des Events empfangen und mit diesen weiterarbeiten. Dazu muss dieser den Aufbau der Rohdaten kennen. Sobald ein Endpunkt Empfänger von mehreren Events oder sogar Event-Formaten ist, so muss der Empfänger anhand der Rohdaten entscheiden, wie er diese Rohdaten entmarshalen muss. Mit Hilfe von Metadaten in einem einheitlichen Datenformat kann dieser Prozess vereinfacht werden. Solche Metadaten können Typ-, Datenformat- oder Versionsinformationen sein. Um diesen zusätzlichen Aufwand zu verhindern nutzt der Prototyp das CloudEvents Format in Version 0.1.



**Listing 5.1** Konfiguration unterschiedlicher Events auf einem S3 Speicher

```

# Event zum Auslösen der Bildkompression
- s3:
  bucket: imageuploadbucket
  event: s3:ObjectCreated:*
  rules:
    - prefix: upload/
# [...]
# Event zum Auslösen der Funktion, die das Event abfängt und weiterleitet
- s3:
  bucket: imageuploadbucket
  event: s3:ObjectCreated:*
  rules:
    - prefix: compressed/

```

Jedes von AWS ausgelöste Event hat einen unterschiedlichen Aufbau. Während ein von Kinesis erzeugtes Event beispielsweise das Attribut `eventName` besitzt, das den Typ des Events identifiziert, fehlt einem SNS Event dieses [CNCf]. Selbst eigentlich identische Attribute unterscheiden sich in ihrer Schreibweise. `EventVersion` beginnt im Kinesis Event mit einem Großbuchstaben, während es im SNS Event `eventVersion` bezeichnet wird.

Daher muss für jedes Event eine eigene Umwandlung definiert werden. Eine solche Umwandlung des in Listing 5.2 dargestellten S3 Events ist in Listing 5.3 zu sehen. Die Regeln der Umwandlung der einzelnen Attribute wird im Folgenden beschrieben.

Um auch über die Cloud-Umgebung hinweg Event-Typen eindeutig zu definieren, beginnen sie in allen Prototypen immer mit einem Präfix der auslösenden Umgebung. Dieses Präfix wird von weiteren Informationen mit einem Punkt getrennt. Weitere Detailstufen über den Ursprung können durch einen Punkt angehängt werden. Je weiter hinten eine Information in dieser Kette steht, umso detaillierter ist sie. Der Event-Typ `aws.s3.objectcreated.put` zeigt somit beispielsweise zuerst, dass dieses Event in der AWS Umgebung ausgelöst wurde. Nach dem Punkt folgt die Komponente, in der es ausgelöst wurde, im Beispiel ein S3 Speicher. Es folgen die Information, dass ein Objekt erzeugt wurde und anschließend, wie es erzeugt wurde. Diese Information stammt aus dem S3 Event Attribut `eventName`.

`source` ist ein zusammengesetzte Attribut aus dem S3 Bucket-ARN und dem S3 Object. In AWS werden Ressourcen über einen eindeutigen Amazon-Ressourcenname (ARN) identifiziert. Dieser Name wird um den sogenannten Key des Objects ergänzt. AWS benutzt diese Schreibweise zur Identifikation einer Datei innerhalb eines S3 Speichers selbst [Amac, Abschnitt “Amazon Simple Storage Service (Amazon S3)”].

`cloudEventsVersion` ist aufgrund der verwendeten Version auf den Wert `0.1` festgelegt. Die `eventTypeVersion` wird aus der `eventVersion` entnommen, ebenso, wie die `eventTime`, aus dem gleichnamigen Attribut. In der Payload wird erneut das komplette S3 Event verschickt. Damit wird sichergestellt, dass sämtliche Informationen übertragen werden und bestehende Programmlogik, die auf S3 Events aufbaut, beispielsweise im Bibliotheken, weiterhin verwendet werden kann.

---

### Listing 5.2 Gekürzter Mitschnitt eines von einem S3 Speicher ausgelösten Events

---

```
{
  "eventVersion":"2.1",
  "eventSource":"aws:s3",
  "awsRegion":"us-east-1",
  "eventTime":"2019-06-14T12:00:00.000Z",
  "eventName":"ObjectCreated:Put",
  "s3":{
    "bucket": {
      "arn":"arn:aws:s3:::shareit", [...]
    },
    "object":{
      "key":"compressed/abcd.jpg", [...]
    }, [...]
  }, [...]
}
```

---

---

### Listing 5.3 Umwandlung des S3 Events in das CloudEvents Format

---

```
{
  "cloudEventsVersion": "0.1",
  "eventType": "aws.s3.objectcreated.put",
  "eventVersion" : "2.1",
  "source": "arn:aws:s3:::shareit/compressed/abcd.jpg",
  "eventID": "1234-1234-1234-1234",
  "eventTime": "2019-06-14T12:00:00.000Z",
  "data": {"eventVersion":"2.1", [...]}
}
```

---

Im Anschluss an die Transformation filtert die Funktion die Endpunkte heraus, an die das Event weitergeleitet werden muss. Diese Endpunkte entsprechen den aufzurufenden Funktionen. Ein Beispiel dieser Filterung zum Aufruf der *Image Recognition* ist in Listing 5.4 dargestellt. Da die Funktion auch unterschiedliche Events abfangen und weiterleiten kann, ist diese Filterung notwendig.

---

### Listing 5.4 Filterung der Events im Prototyp

---

```
if (cloudevent.getType().startsWith("aws.s3.objectcreated")) {
  if (cloudevent.getSource().startsWith("arn:aws:s3:::shareit-direct-call/compressed/")) {
    sendToAzure(cloudEvent, config.imagerecognition_url)
  }
}
```

---

Der Prototyp nutzt für die Übertragung den offenen Standard HTTP, beziehungsweise dessen TLS verschlüsselte Version HTTPS. Dieser wird als Auslöser einer Azure Function unterstützt. Die Event-Daten werden im JSON Format als String codiert im Body der HTTP-Nachricht verschickt. Dieses Vorgehen entspricht dem *Structured Content Mode* aus dem *HTTP Transport Binding* der CloudEvents Spezifikation [CNCc].

Bei der Authentifizierung sieht dies etwas anders aus. Durch die Erstellung einer Funktion mit einem HTTP-Endpoint erzeugt Azure automatisch einen sogenannten API Schlüssel. Mit diesem Schlüssel, der als Parameter der Endpoint-URL hinzugefügt wird, ist eine Ausführung der Funktion möglich. Dieses Vorgehen empfiehlt Microsoft allerdings nur für Entwicklungsumgebungen. In Produktivumgebungen sollen Funktions-Endpoints im Zusammenspiel mit dem Azure Active Directory, dem Azure API Management oder einer App Service Environment betrieben werden [Micd]. Mit Hilfe von Azure Management Systems können beispielsweise Benutzer angelegt, deren Rechte detaillierter konfiguriert und weitere Einschränkungen festgelegt werden, wie die erlaubten eingehenden IP-Adressen. Für eine erste prototypische Implementierung ist der Schutz durch einen API Schlüssel an dieser Stelle ausreichend.

Deutlichere Nachteile bringt der Prototyp bei seinem Umgang mit Fehlern. Im Fehlerfall, also beispielsweise, wenn die HTTP-Anfrage länger als die maximal festgelegte Ausführungsdauer einer Lambda Funktion dauert, oder die Funktion eine Ausnahme auslöst, führt AWS sein Wiederholungsverhalten aus. Dieses unterscheidet sich je nach Herkunft des Events. Während synchrone Event-Auslöser, wie beispielsweise HTTP-Aufrufe über ein API-Gateway, nach einer fehlerhaften Ausführung direkt einen Fehlercode zurückgeben, wird der Aufruf der Funktion bei asynchronen Aufrufen bis zu zwei Mal wiederholt. Sollte die Funktion weiterhin nicht erfolgreich sein, gilt sie als unzustellbare Nachricht [Amad]. Diese können in eine Warteschlange geleitet und später erneut verarbeitet werden, eine sogenannte *Dead Letter Queue*, die im nächsten Prototyp vorgestellt wird. Da weitere Komponenten und sogenannte Gateways erst in den nächsten Prototypen vorgesehen sind, sollen diese an dieser Stelle noch nicht betrachtet werden.

Um nach erfolgreicher Bildanalyse das Ergebnis dieser wieder in die AWS Umgebung zu schicken, erzeugt die Beispielapplikation ein benutzerdefiniertes Event. Dieses wird von der Lambda Funktion *Tags Writer* empfangen und von dort in der Datenbank gespeichert.

Um dies zu ermöglichen enthält der Prototyp auch in der Azure Umgebung Funktionscode, um ein Event an einen von AWS bereitgestellten Funktions-Endpoint zu senden. Dieser funktioniert nach dem selben Prinzip. AWS bietet allerdings keinen direkten HTTP-Endpoint für Funktionen an, ermöglicht es aber über ein API-Gateway eine Funktion per HTTP aufzurufen. Die grundlegende Funktionsweise bleibt identisch zu der des bisher beschriebenen Aufrufs und wird daher an dieser Stelle nicht weiter ausgeführt.

### 5.1.2 Anforderungsabgleich

**Anforderung 1 - Multi-Cloud Event:** Durch den vorgestellten Prototypen ist es möglich Multi-Cloud Events zu realisieren. Sowohl interne, als auch externe Events werden von der AWS Umgebung ausgelöst und erreichen ihr Ziel, die Auslösung einer Funktion. Allerdings unterscheidet der Prototyp momentan noch zwischen internen und externen Events. Interne werden über rein über das System des Cloud-Providers geleitet, während externe abgefangen und weitergeleitet werden.

**Anforderung 2 - Abfangen eines Events:** Das Abfangen eines Events funktioniert durch die Funktion *Event Interception* wie in der Anforderung beschrieben. Mit ihr ist es möglich alle Events abzufangen, die von der FaaS-Umgebung unterstützt werden. Die Funktion muss dazu lediglich zusätzlich auf das abzufangende Event registriert werden.

**Anforderung 3 - Benutzerdefiniertes Auslösen eines Events:** Da die Funktionalität zum Auslösen eines Events in eine Programmbibliothek ausgelagert wurde, ist es möglich innerhalb einer Funktion benutzerdefinierte Events aufzurufen. Dies wird in der Funktion *Image Recognition* demonstriert. Dabei besteht die Einschränkung, dass die Programmbibliothek nur in der genutzten Laufzeitumgebung genutzt werden kann. Eine Nutzung dieser in anderen Umgebungen als Node.js ist momentan nicht möglich, da sie in JavaScript geschrieben ist und auf Ressourcen dieser Laufzeitumgebung aufbaut.

**Anforderung 4 - Weiterleiten von Events:** Nach dem Abfangen können Events erfolgreich weitergeleitet werden. Dazu enthält die Funktion zum Abfangen des Events alle benötigten Informationen, um das Event an die gewünschten Endpunkte weiterzuleiten. Events, die, wie in der Anforderung definiert, die Umgebung nicht verlassen dürfen, werden dadurch aussortiert, dass sie von Beginn an nicht abgefangen, oder von der Filterung der Abfang-Funktion herausgefiltert werden.

Während die Anforderung bisher erfüllt wurde, sieht dies mit dem Abschnitt der Zustellung im Fehlerfall schlechter aus. Bei Nichterreichbarkeit eines Endpunktes, oder bei einem anderen Fehler während des Zustellungsversuches, kann es zu Verlusten von Events kommen. Da fehlgeschlagene Ausführungen der Funktion nicht, oder nur für einen sehr kurzen Zeitraum wiederholt und anschließend nicht abgespeichert werden, gehen diese Nachrichten verloren. Für einen Einsatz in Produktivumgebungen ist der Prototyp in seiner derzeitigen Form nicht einsetzbar. Er erfüllt daher die Anforderung nicht.

**Anforderung 5 - Private Events:** Da Events direkt an einzelne Umgebungen geschickt werden und die Filterung der zu verschickenden Events in der Ursprungs-Umgebung geschieht, sind private Events möglich. Diese werden nur an die definierten Umgebungen geschickt. Eine prinzipielle Verteilung an alle angeschlossenen Umgebungen findet nicht statt.

**Anforderung 6 - Auslösen von Aktionen:** Aktionen werden, wie an den Funktionen *Image Resize* und *Image Recognition* gezeigt wurde, erfolgreich ausgelöst. Der bereits im Konzept genannte Nachteil, wird in der prototypischen Implementierung deutlich. Die Entscheidung, welche Aktionen auf welches Event folgen, wird in der Cloud-Umgebung des Event-Erzeugers getroffen. Dadurch müssen bei einer Anpassung der Programmlogik Änderungen in mehreren Umgebung stattfinden. Die Komplexität einer Änderung wird somit erhöht. Der Prototyp erfüllt diese Anforderung demnach nur eingeschränkt.

**Anforderung 7 - Mehrere Aktionen durch ein Event:** In der aktuellen Umsetzung ist es möglich beim Eintreffen eines Events mehrere Events aufzurufen. Die Aufrufe über die Funktion *Event Interception* finden allerdings alle über externe Verbindung statt. Um Anforderung 7 zu erfüllen muss es allerdings möglich sein, mehrere Aktionen aufgrund eines Events auszuführen, unabhängig davon, in welcher Umgebung Funktionen aufgerufen werden müssen. Dies ist in der aktuellen Version des Prototyps nicht der Fall. Interne Events werden dort als providerspezifische Events konfiguriert und verwaltet.

**Anforderung 8 - Verkettung von Events:** Jedes Event im System wird als eigenständige Nachricht verschickt. Dadurch ist es auch möglich ein Event aufgrund eines anderen Events auszulösen. Dies wird im Prototyp demonstriert. Das Event, das eine erfolgreiche Bildererkennung signalisiert, wird aufgrund des davor aufgerufenen Events des Uploads des komprimierten Fotos ausgelöst. Im Sequenzdiagramm des Referenzszenarios in Abbildung 3.3 wird diese Verkettung sichtbar. Die Anforderung ist somit erfüllt.

**Anforderung 9 - Vereinheitlichung von Event-Formaten:** Innerhalb der Funktion *Event Interception* wird das AWS in ein CloudEvent umgewandelt. Bei der Erzeugung des Events in der Funktion *Image Recognition* wird ebenfalls dieses Format verwendet. Diese Anforderung gilt somit gleichfalls als erfüllt.

**Anforderung 10 - Skalierbarkeit und Verfügbarkeit:** Da lediglich vom FaaS-Provider bereitgestellte Funktionen eingesetzt werden, verändert sich hinsichtlich der Skalierung und Verfügbarkeit im Vergleich zu einer Single-Cloud Lösung nichts. Alle Zusagen der Provider hinsichtlich Skalierung und Verfügbarkeit bleiben bestehen. Insbesondere bleibt die Funktionalität der Cloud-Umgebung bestehen, wenn eine andere Umgebung nicht verfügbar ist.

**Anforderung 11 - Providerunabhängigkeit und Erweiterbarkeit:** Für jede eingebundene Cloud-Umgebung werden starke Anpassungen an der Funktion, die Events abfängt, notwendig. Da nicht jeder Provider die selbe Laufzeitumgebung als FaaS anbietet, oder schon Versionsunterschiede der selben Laufzeitumgebung zu Inkompatibilität unter den Cloud-Providern führen können, müssen diese Anpassungen für jede bereitgestellte Cloud-Umgebung gemacht werden. Eine Providerunabhängigkeit und Erweiterbarkeit ist prinzipiell gegeben, der universelle Einsatz in unterschiedlichen Umgebungen wird durch die genannten Gründe allerdings erheblich erschwert.

**Anforderung 12 - Sicherheit:** Durch verschlüsselte Übertragung mit dem Standard HTTPS wird ein Aspekt dieser Anforderung berücksichtigt. Eine Verifizierung der Authentifizierung ist stark von der jeweiligen Implementation der entsprechenden Cloud-Umgebung abhängig. Durch die Nutzung der Funktion *Event Interception* kann nahezu jede von einem Provider bereitgestellte Authentifizierungsmöglichkeit genutzt werden. Nachteilig ist, dass jede dieser Implementierungen in jeder anderen Cloud-Umgebung implementiert sein müssen. Dies wirkt sich nachteilig auf die Erweiterbarkeit aus. Die Anforderung an die Sicherheit ist dadurch allerdings erfüllt.

### 5.1.3 Fazit

Der Prototyp *Direkter Aufruf* stellt eine funktionstüchtige Umgebung bereit, um Events aus einer Cloud-Umgebung in weitere Cloud-Umgebungen zu transportieren. Insbesondere durch die Vielzahl der benötigten Endpunkte, der damit verbundenen Komplexität der Authentifizierung und die limitierte Möglichkeit zur erneuten Sendung von fehlgeschlagenen Nachrichten, lässt sich eine Implementierung rein auf der Grundlage von Funktionen nicht empfehlen. Ebenso werden wichtige Prinzipien der Eventbasierten Kommunikation in diesem Prototypen verletzt [EN10, S. 30 ff.]. Durch Entkopplung der beiden Umgebungen durch weitere Komponenten können diese Herausforderungen und Probleme gelöst werden.

Mindestens der Einsatz einer Warteschlange für nicht zustellbare Nachrichten, um den Verlust von Events zu verhindern, ist selbst für die Entwicklung eines Provisoriums für Übergangszwecke unbedingt erforderlich. Neben Provisorien lässt sich das Konzept der direkten Aufrufe lediglich

in Anwendungsfällen von Multi-Cloud-Umgebungen nutzen, in denen einzelne Aufrufe über Umgebungsgrenzen hinweg getätigt werden. Sobald eine komplexere Eventbasierte Kommunikation stattfindet sollte auf dieses Konzept verzichtet werden.

## 5.2 Zentrales Gateway

Der nächste Prototyp bedient sich dem Konzept des zentralen Gateways. Er setzt dabei auf dem Prototyp *direkter Aufruf* auf und übernimmt die Funktionalität zum Abfangen und zur Umwandlung eines Events. Als zentrales Gateway dient NATS im Zusammenspiel mit je einer Schnittstelle zu den angebotenen Umgebungen.

### 5.2.1 Umsetzung

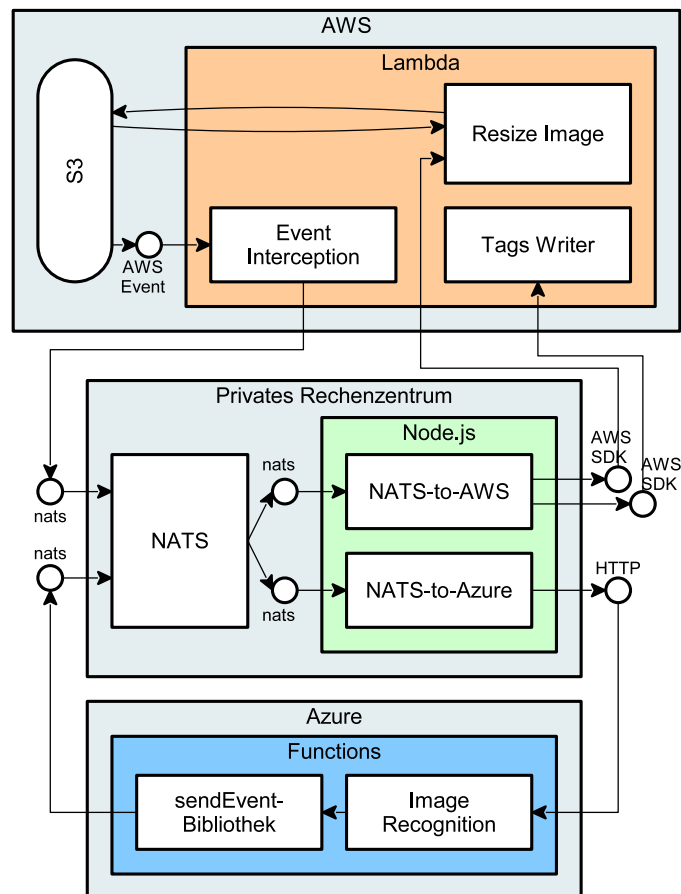


Abbildung 5.2: Prototyp einer Umgebung mit zentralem Gateway.

In dem in Abbildung 5.2 dargestellten Prototyp übernimmt NATS mit der Erweiterung *NATS Streaming* die Rolle des zentralen Gateways. Dieses wird im Prototyp ohne Clusterbetrieb auf einem Server auf Infrastruktur innerhalb eines klassischen Rechenzentrums betrieben. Alternativ dazu ist auch der Betrieb in einer Cloud-Umgebung möglich. Dazu bieten sich Container an, die

in unterstützte Umgebungen deployed werden können, ohne sie speziell anpassen zu müssen. Ein solcher Container wird auch beim Einsatz der NATS Instanz genutzt und kann somit einfach auf andere Infrastruktur verschoben werden.

Die Funktion zum Abfangen des Events wird nahezu identisch zum vorherigen Prototyp konfiguriert. Allerdings filtert dieser Prototyp nicht bereits durch Konfiguration bestimmte Events heraus. Stattdessen werden die Events aller hochgeladenen Dateien in den S3 Speicher abgefangen und innerhalb des Multi-Cloud Events Systems verarbeitet. Einen weiteren Unterschied gibt es in der anschließenden Art der Weiterleitung. Während bisher bereits in dieser Funktion eine Filterung stattfand, werden bei diesem Prototyp sämtliche Events weitergeleitet. Dazu baut die Funktion eine Verbindung zum NATS Server auf und versendet das Event als CloudEvent.

In Listing 5.5 ist ein Auszug des von NATS verwendeten Protokolls abgebildet. Mit ihm kommunizieren alle Komponenten mit dem NATS Server. In Version 0.2 führt CloudEvents ein offizielles *NATS Transport Binding* ein, in dem geregelt wird, wie ein CloudEvent über NATS verschickt werden soll [CNCE]. Dieses lässt sich, aufgrund des einfachen Aufbaus, allerdings auch auf die verwendete Version 0.1 anwenden. Alle Sender und Empfänger kommunizieren innerhalb eines *Subjects*. Das eigentliche Event wird als *Payload* verschickt. Als Event Format muss JSON unterstützt werden, weitere Formate sind laut Spezifikation optional [CNCE]. Im Prototyp wird ausschließlich per JSON-Format kommuniziert. Dies verhindert weitere Umwandlungsschritte.

Auch an dieser Stelle stellt sich der erneute Nachrichtenversand als Schwierigkeit heraus. Da für den Versand wieder auf eine Lambda Funktion zurückgegriffen wird, gelten auch für sie die Einschränkungen der Wiederholung im Fehlerfall. Sollte ein Event nicht verschickt werden können, bleibt der Funktion lediglich der Abbruch. Events können daher bei Nichterreichbarkeit des zentralen Gateways verloren gehen.

Um fehlgeschlagene Events nicht für immer verloren gehen zu lassen stellt AWS Lambda sogenannte *Dead Letter Queues* bereit. Damit ist es möglich Funktionsaufrufe bei denen Fehler auftreten in eine Warteschlange des Simple Queue Services (SQS) oder ein SNS Topic zu verschieben. Mit Hilfe dieses Speichers ist es möglich weitere Fehlerbehebungen durchzuführen. Beispielsweise kann ein Administrator informiert oder zu einem späteren Zeitpunkt eine erneute Durchführung der Funktion ausprobiert werden. Events gehen somit nicht verloren.

Neben der NATS Instanz werden auf dem Server auch die beiden Schnittstellen betrieben, die die Verbindung zwischen dem NATS Protokoll und den Cloud-Umgebungen herstellen. Dabei handelt es sich um zwei Node.js Applikationen, die sich beim Start mit der NATS Instanz verbinden und auf Events reagieren. Dies ist notwendig, da NATS nativ keinen Aufruf von FaaS-Funktionen unterstützt. Im Weiteren werden diese Schnittstellen Cloud-Adapter genannt.

Beim Start müssen sich die Cloud-Adapter mit dem NATS Server verbinden und mit dem `sub` Befehl das gewählte Subject abonnieren. Sobald ein Event eintrifft, entscheiden sie, welche Aktionen ausgeführt werden. Dabei stellt sich die Frage an welchen Attributen eines Events entschieden werden muss, welches Event welche Funktionen ausführt. Die Spezifikation der CloudEvents macht dazu keine Vorgaben. Denkbar sind hierbei sehr viele Anwendungsfälle, die eine Filterung an einer breiten Anzahl an Attributen begründen würden.

Unbestritten muss eine Filterung auf den *eventType* stattfinden. Dieses Attribut ist die Grundlage, an der entschieden wird, ob eine Aktion stattfinden soll. Ähnlich sinnvoll ist auch die Filterung auf das Attribut *source*. In der Beispielapplikation soll anhand diesem Attribut entschieden werden, ob es

### Listing 5.5 Mitschnitt des von NATS verwendeten Protokolls

---

```
# Abonnieren eines Subjects
SUB mySubject myInternSubjectName
+OK

# Veröffentlichung einer Nachricht
PUB mySubject 50
{ "eventType": "aws.s3.objectcreated.put", [...] }
+OK

# Empfang einer Nachricht
MSG mySubject myInternSubjectName 50
{ "eventType": "aws.s3.objectcreated.put", [...] }
```

---

sich bei der hochgeladenen Datei um die Originaldatei oder die komprimierte Version davon handelt. Ebenso kann anhand Dateiendung der *source* auch der Dateityp des Auslösers ermittelt werden. Die Bildkomprimierung könnte somit durch unterschiedliche Funktion für PNG- oder JPEG-Dateien durchgeführt werden.

Um dies anhand des *source* Attributes feststellen zu müssen reicht nicht nur eine Filterung auf das komplette Attribut. Eine Überprüfung, ob das Attribut einen bestimmten String am Anfang, am Ende oder irgendwo enthält, muss ebenso erlaubt sein.

Theoretisch ließe sich diese Filterung auf nahezu alle Attribute des CloudEvents begründen. So kann es auch sinnvoll sein, anhand der Versionsnummer des Event-Typs zu unterscheiden, welche Aktion ausgeführt wird, damit die Funktion ausgeführt wird, die exakt mit dieser Versionsnummer kompatibel ist. Zur Übersichtlichkeit wird in den Prototypen soweit möglich eine Filterung auf Basis des *eventType* und der *source* durchgeführt.

Hierbei unterscheiden sich die prototypischen Implementierungen zwischen dem AWS- und dem Azure-Adapter. Während der Azure-Adapter eine HTTPS-Schnittstelle nutzt, bindet der AWS-Adapter das von Amazon bereitgestellte AWS-SDK ein, um darüber Funktionen aufzurufen.

Um Anforderung 4 zu erfüllen, muss es möglich sein, fehlgeschlagene Zustellversuche zu wiederholen. Dies beinhaltet insbesondere die spätere Zustellung, wenn die Verbindung zu einem der Adapter nicht vorhanden ist. Aus diesem Grund wird die NATS Streaming Erweiterung benötigt. Diese ermöglicht *Log based Streaming*. Alle Nachrichten werden für eine konfigurierbare Zeit im Speicher vorbehalten [Clod]. Ein Client, der sich am Server mit einer ID identifiziert, kann somit alle Nachrichten, die im Speicher vorhanden sind, ab einem bestimmten Zeitpunkt abrufen. Durch ein Acknowledgement, das beim erfolgreichen Empfang, oder auch erst nach erfolgreicher Verarbeitung gesetzt werden kann, ist es ebenso möglich, alle noch nicht verarbeiteten Nachrichten abzurufen. Sollte das Acknowledgement nicht innerhalb einer definierten Zeit beim Server eintreffen, wird ein erneuter Zustellversuch unternommen. Dadurch kann auch ein Fehler zwischen dem Empfang und dem Weiterleiten der Nachricht an einen Empfänger erkannt werden. Da ein Acknowledgement für jeden Client einzeln gesetzt wird, kann jeder Adapter Events in der eigenen Geschwindigkeit verarbeiten. Beim Ausfall einer Umgebung hat dies keinen Einfluss auf die Verarbeitung oder erneute Zustellversuche in die anderen Umgebungen.



Damit keine unberechtigten Zugriffe auf den Kommunikationskanal stattfinden können, muss dieser abgesichert werden. Standardmäßig kommuniziert NATS über eine unverschlüsselte TCP Verbindung ohne Authentifizierungsmaßnahmen. Dies lässt sich allerdings umkonfigurieren. NATS bietet eine verschlüsselte Datenübertragung mit Hilfe von TLS Zertifikaten und auch Authentifizierung von Clients [Clof]. Mit Hilfe der Authentifizierung kann einzelnen Clients die Berechtigung gegeben werden, nur bestimmte Subjects zu abonnieren oder nur auf bestimmten Subjects zu schreiben. Dies kann hilfreich sein, wenn bestimmte Events nicht an alle Cloud-Umgebungen geschickt werden sollen. Dadurch kann sichergestellt werden, dass bestimmte Umgebungen keinesfalls Zugriff auf diese Events haben, indem sie nicht für Kommunikation über dieses Subject berechtigt sind.

Neben der Schnittstelle zwischen der Funktion, die Events abfängt und NATS und anschließend zwischen NATS und den Cloud-Adaptern, muss auch der anschließende Kanal zwischen Cloud-Adaptern und den Cloud-Umgebungen abgesichert werden. Da es sich bei den Cloud-Adaptern um Node.js Applikationen handelt, kann im Beispiel von AWS auf das von Amazon zur Verfügung gestellte SDK zurückgegriffen werden. Dieses erlaubt vollen Zugriff auf die von AWS bereitgestellte Funktionalitäten. Unter anderem können damit Lambda Funktionen aufgerufen werden. Durch das AWS Identity and Access Management (IAM) können Nutzer erzeugt werden, die lediglich zur Nutzung von bestimmten Ressourcen, wie beispielsweise zur Ausführung von definierten Lambda-Funktionen berechtigt sind [Amaf]. Mit Hilfe der Zugriffsschlüssel eines dieser Benutzer lässt sich dann der Zugriff auf die Lambda Funktionen aus dem AWS-Adapter authentifizieren.

Durch Duplizierung der Abfang-Funktion und des Cloud-Adapters, die Kommunikation über unterschiedliche Subjects und die Nutzung von unterschiedlichen Berechtigungsprofilen lassen sich private Events, wie in Anforderung 5 beschrieben, über das selbe Gateway bereitstellen. Diese privaten Events sind dann für möglicherweise dritte an das Gateway angeschlossene Umgebungen weder lesbar, noch lassen sich Aktionen auf diese Events aus diesen dritten Umgebungen auslösen.

### 5.2.2 Anforderungsabgleich

**Anforderung 1 - Multi-Cloud Event:** Dem Nutzer steht bei diesem Prototyp die Möglichkeit zur Verfügung Multi-Cloud Events zu realisieren. Dieser unterscheidet nicht mehr, wie im vorherigen, zwischen internen und externen Events. Alle Events werden identisch behandelt.

Da die Bereiche der Anforderungen für **Anforderung 2 - Abfangen eines Events**, **Anforderung 3 - Benutzerdefiniertes Auslösen eines Events**, **Anforderung 8 - Verkettung von Events** und **Anforderung 9 - Vereinheitlichung von Event-Formaten** im Vergleich zum ersten Prototyp nicht verändert wurden, gilt auch weiterhin der Anforderungsabgleich aus diesem Prototyp. Daher werden sie an dieser Stelle nicht erneut betrachtet.

**Anforderung 4 - Weiterleiten von Events:** Durch die Nutzung einer Dead Letter Queue für die Funktion *Event Interception* ist eine Zustellung von Events zwar nicht sichergestellt, eine Fehlerbehandlung wird allerdings ermöglicht. Sobald ein Event das zentrale Gateway NATS erreicht hat, ist eine Zustellung an alle für das Event registrierten Umgebungen sichergestellt. Dies garantiert die Verwendung der NATS Erweiterung NATS Streaming. Die Anforderung wird erfüllt.

**Anforderung 5 - Private Events:** Eine Implementierung von privaten Events ist, wie beschrieben durch ein separates Subject und eine Duplizierung mancher Komponenten, in diesem Prototyp möglich. Daher erfüllt der Prototyp diese Anforderung.

**Anforderung 6 - Auslösen von Aktionen:** Die Aktionen, die durch ein Event ausgeführt werden, werden vom zentralen Gateway aufgerufen. Anstatt, wie im vorherigen Prototyp, diese Ausführungslogik über Umgebungen zu verteilen, wird sie in diesem zentral gebündelt. Ebenso können Aktionen zur Laufzeit der Umgebungen registriert und entfernt werden, ohne das in der Umgebung in der das Event ausgelöst wird, Änderungen vorgenommen werden müssen.

Dieser Entkopplung zwischen Event-auslösender Umgebung und dem Gateway ist daher bereits der engen Kopplung aus dem ersten Prototypen vorzuziehen. Zwischen dem Gateway und der Cloud-Umgebung, in der Aktionen ausgeführt werden herrscht allerdings weiterhin eine enge Kopplung der Funktionalitäten.

**Anforderung 7 - Mehrere Aktionen durch ein Event:** Das Auslösen mehrere Aktionen auf ein Event ist möglich. Der Prototyp kann so konfiguriert werden, dass sowohl *Resize Image*, als auch *Image Recognition* auf das Event des vom Anwender hochgeladenen Fotos reagieren.

**Anforderung 10 - Skalierbarkeit und Verfügbarkeit:** Weiterhin bleibt zu beachten, dass bei diesem Prototyp das NATS Gateway momentan einen Single-Point-of-Failure darstellt, ebenso wie die daran angeschlossenen Schnittstellen. Während NATS einen nativen Clusterbetrieb ermöglicht, kann dies bei den Cloud-Adaptoren leider nicht so einfach umgangen werden. Durch das *Log based Streaming* wird Datenverlust zwar prinzipiell verhindert, allerdings muss trotzdem sichergestellt werden, dass Fehler erkannt und entweder durch manuelle Eingriffe oder automatisiert die Funktionalität wieder hergestellt wird. Die Erfüllung dieser Anforderung hängt sehr stark von der Art und Weise ab, wie der NATS Server, beziehungsweise das NATS Cluster betrieben wird.

Beim Blick auf den Durchlauf eines Events, dass in der selben Umgebung aufgerufen wird, in der es auch eine Aktion auslösen soll, wird das Problem eines zentralen Gateways besonders deutlich. Dies ist in der Beispielapplikation beim Upload einer Datei in den S3 Speicher und der dadurch ausgelösten Bildkomprimierung der Fall. Anstatt innerhalb der eigenen Cloud-Umgebung verarbeitet zu werden, wird das Event zuerst an das zentrale Gateway geleitet. Sollte dieses beispielsweise aufgrund von Netzwerkproblemen abgeschnitten sein, so findet keine Aktion auf das Event statt. Somit ist die Anforderung von diesem Prototyp nicht erfüllt.

**Anforderung 11 - Providerunabhängigkeit und Erweiterbarkeit:** Diese prototypische Implementierung zeigt auf, welche Möglichkeiten ein zentrales Gateway zur Verfügung stellen muss, um mit verschiedenen Cloud-Umgebungen kommunizieren zu können. Neben standardisierten Kommunikationskanälen, wie HTTPS, muss es auch möglich sein, providerspezifische Protokolle zu nutzen. Da durch die Node.js Umgebung in der die Cloud-Adapter ausgeführt werden, nahezu jede Schnittstelle abgebildet werden kann, erfüllt der Prototyp diese Anforderung. Sollte aufgrund von fehlenden Programmbibliotheken oder Ähnlichem keine Unterstützung von Node.js möglich sein, so kann eine beliebige andere Laufzeitumgebung verwendet werden, die eine Verbindung zu NATS herstellt.

**Anforderung 12 - Sicherheit:** Die Sicherheit ist auch bei diesem Prototyp von der konkreten Implementierung der Cloud-Adapter abhängig. Da nahezu jede Verbindung möglich ist, kann diese Verbindung auch sicher gestaltet werden.

Die Verbindung von und zu NATS lässt sich mit den von NATS zur Verfügung gestellten Mitteln gut absichern.

### 5.2.3 Variation

Da es sich bei diesem Konzept um das Konzept handelt, nachdem auch das *Serverless Event Gateway* entwickelt wurde und betrieben werden kann, kann auch dies statt der NATS Implementation genutzt werden. Anstelle des NATS Protokoll müssen die *sendEvent-Bibliothek* und die Funktion *Event Interception* dann das vom Event Gateway bereitgestellte Protokoll zur Eventauslösung nutzen [Ser19]. Da auch das Event Gateway zum Aufruf von Lambda Funktionen auf das AWS SDK zurückgreift erfolgt dieser Aufruf gleich, wie im NATS Prototyp. Gleiches gilt für die Azure Functions Implementierung und den dort verwendeten HTTP-Endpunkt.

### 5.2.4 Fazit

Verglichen mit dem ersten Prototyp erscheint dieser zuerst komplexer. Dies liegt in erster Linie an den zusätzlichen Komponenten, die benötigt werden. Durch eine klarere Struktur, zentral gelagerter Funktionslogik und einem ganzheitlichen Blick auf die Nachrichtenzustellung, erschließt sich dieser Prototyp allerdings als universeller einsetzbare Möglichkeit zur Umsetzung von Eventbasierter Kommunikation in Multi-Cloud-Umgebungen. Die einfachere Erweiterbarkeit auf weitere Cloud-Umgebungen, sowie die Möglichkeit von providerspezifischen Optimierungen ermöglicht eine optimale Grundlage um weitere Cloud-Provider an das System anzuschließen.

Aufgrund des zentralen Designs müssen sämtliche Events erst durch diese zentrale Gateway geleitet werden. Dies erhöht die Latenz zwischen Ereignis und darauf folgender Aktion und führt zu einem Komplettausfall der Systeme bei einem Ausfall des zentralen Systems.

## 5.3 Dezentrale Gateways

Der zweite Prototyp bedient sich dem Konzept der dezentralen Gateways. Um eine cloud-native Implementierung zu gewährleisten soll dazu auf vom jeweiligen Cloud-Provider bereitgestellte Ressourcen zurückgegriffen werden. Dies ermöglicht eine optimale Ausnutzung der vom Provider vorgesehene Konzepte und Komponenten.

Um eine gute Erweiterbarkeit zu erreichen sollte bei diesem Prototyp versucht werden eine einheitliche Schnittstelle zwischen den Umgebungen zu etablieren. Durch diese Schnittstelle können weitere Umgebungen durch Hinzufügen eines weiteren Endpunktes der Nachrichtenzustellung in die bisherigen Gateways in das bestehende System eingehängt werden.

Auch bei diesem Prototyp wurde das Abfangen und Umwandeln eines Events vom Prototyp des direkten Aufrufs wiederverwendet.

### 5.3.1 Umsetzung

Der in Abbildung 5.3 abgebildete Prototyp nutzt als dezentrales Gateway in der AWS Umgebung den Simple Notification Service (SNS) von Amazon und in Azure das von Microsoft entwickelte Event Grid. Diese beiden Gateways sollen im Folgenden betrachtet werden.

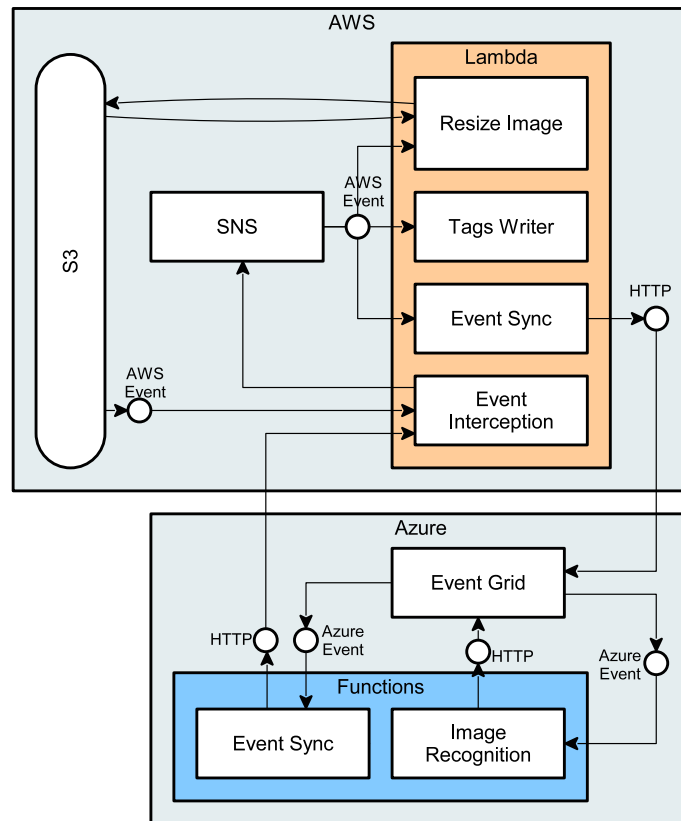


Abbildung 5.3: Prototyp einer Umgebung mit dezentralen Gateways

SNS ermöglicht es Events direkt aus verschiedenen Quellen abzufangen und diese an unterschiedliche Abonnements zu verteilen. Dazu wird das ursprüngliche Event in das *Message* Objekt eines SNS Event verpackt und weitergeleitet. Ein Beispiel eines solchen SNS Events ist in Listing 5.6 abgebildet. Zur Filterung bietet SNS eine Liste an Objekten in den *MessageAttributes* an, anhand derer Nachrichten nur an bestimmte Abonnements weitergeleitet werden können. Eine Filterung anhand von weiteren Attributen findet nicht statt. Bei der direkten Weiterleitung aus Event-Quellen werden keine Werte im *MessageAttributes* Feld gesetzt, wodurch eine Filterung nicht möglich ist.

Aus diesem Grund enthält auch dieser Prototyp die Funktion *Event Interception*. Diese wandelt das proprietäre Event-Format in das CloudEvents-Format um und leitet dieses an SNS weiter. Zur Filterung in SNS werden alle Attribute, die zur Filterung zur Verfügung stehen sollen, zusätzlich in das *MessageAttributes* Objekt übernommen. Dies erhöht zwar die Größe des Events, da diese Informationen doppelt vorhanden sind, vereinfacht dafür allerdings die Filterung und Weiterverarbeitung. Im Prototyp werden die Felder *eventType* und *source* aus dem CloudEvent kopiert.

Neben dem Abfangen von internen Events, ist die Funktion *Event Interception* auch für das Abfangen von externen Events zuständig. Dies ist nötig, da SNS keine Möglichkeit zur Verfügung stellt Nachrichten über eine offene Schnittstelle, wie einen HTTP-Endpoint entgegenzunehmen. Daher wird ein HTTP Event Trigger für die Funktion verwendet. Um diese Schnittstelle standardkonform zu halten, nimmt sie Events im CloudEvents Format entgegen und wandelt diese in das SNS Format um. Der HTTP-Endpoint muss mit einer Authentifizierung abgesichert werden. AWS erlaubt verschiedene Authentifizierungsmöglichkeiten. Um eine einheitliche Schnittstelle zu schaffen

**Listing 5.6** Gekürztes Beispiel eines SNS Events im JSON Format nach [Amab]

```

{
  "Type": "Notification",
  "MessageId": "a1b2c34d-567e-8f90-g1h2-i345j67klmn8",
  "TopicArn": "arn:aws:sns:us-east-2:123456789012:MyTopic",
  "Message": "message-body-with-transaction-details",
  [...]
  "MessageAttributes": {
    "customer_interests": {
      "Type": "String.Array",
      "Value": "[\'soccer\', \'rugby\', \'hockey\']"
    }, [...]
  }
}

```

**Listing 5.7** Konfiguration zur Filterung von intern aufgetretenen Events im Serverless Framework.

```

events:
  - sns:
      topicName: ${self:custom.snstopic}
      filterPolicy:
        type:
          - prefix: aws.

```

wurde dabei im Prototyp auf die gleiche Authentifizierung gesetzt, wie sie auch im Event Grid eingesetzt wird. Der Anfragende überträgt dafür einen HTTP-Header mit einem geheimen Schlüssel. Stimmt dieser mit dem Wert, der in der Funktion hinterlegt ist, überein, so gilt die Anfrage als authentifiziert.

Um Events an die Gateways in andere Umgebungen zu leiten, besteht prinzipiell die Möglichkeit, diese Umgebungen als HTTP-Endpoint direkt in SNS einzubinden. Da dabei allerdings Nachrichten in einem SNS-spezifischen Format versendet werden, ist die Nutzung dieser Funktionalität nicht sinnvoll. Stattdessen steht die Lambda-Funktion *Event Sync* zur Verfügung. Diese muss zwischen Events, die innerhalb der eigenen Umgebung entstanden sind und Events, die außerhalb dieser Umgebung entstanden sind, unterscheiden können, um lediglich die internen Events an andere Umgebungen weiterzuleiten. Dafür wäre es möglich ein weiteres *MessageAttribute* zu verwenden. Allerdings ist diese Information bereits im Attribut *eventType* vorhanden. Da alle Events, die innerhalb der AWS Umgebung entstanden sind mit dem *eventType* "aws." beginnen, kann diese Unterscheidung anhand diesem Präfix getroffen werden. Diese Konfiguration im Serverless Framework ist in Listing 5.7 abgebildet. Hohpe und Woolf sprechen bei dieser Art des Routing von einem *Message Filter* [HWB04, S. 217]. Dieser blockiert die Weiterleitung aller Nachrichten, die für folgende Komponenten uninteressant sind. Da alle Events, deren *eventType* nicht mit dem genannten Präfix beginnt, bereits von einem anderen Gateway an alle anderen Umgebungen verteilt werden, trifft dies in diesem Fall zu.

Fehler, die bei diesem Funktionsaufruf auftreten können, werden durch die Verschiebung des Events in eine *Dead Letter Queue* zwar nicht behoben, die fehlgeschlagenen Events können dadurch allerdings zu einem späteren Zeitpunkt erneut versandt oder manuell untersucht werden. Die Funktionsweise eines *Dead Letter Queue* wird im Prototyp *Zentrales Gateway* ausgeführt.

Für die Weiterleitung in mehr als eine weitere Umgebung, kann die *Event-Sync* Funktion für jede zusätzliche Umgebung dupliziert werden. Dies erhöht zwar prinzipiell die Anzahl der auszuführenden Funktionen, ermöglicht es aber, dass die Fehlerbehebung für jede Umgebung einzeln abläuft. Ein Event, das in Cloud-Umgebung B geschickt werden konnte, beim Versand in Cloud-Umgebung C allerdings ein Timeout auftrat, landet dann nur das Event im *Dead Letter Queue* von Umgebung C. Ein doppeltes Weiterleiten in Umgebung B bei der Fehlerbehebung wird damit verhindert.

Durch die Nutzung von komplett unterschiedlichen Gateways in jeder Umgebung muss das Event Grid unabhängig von der AWS Umgebung betrachtet werden. Mit dem Azure Event Grid stellt Microsoft eine deutlich besser auf diesen Anwendungsfall angepasste Gateway bereit. Dank der nativen Unterstützung des CloudEvents Format in Version 0.1 und der Bereitstellung von HTTP-Verbindungspunkten sowohl für die ausgehenden, als auch für die eingehenden Eventbasierte Kommunikation, lässt sich Event Grid in der Theorie ohne große Anpassungen nutzen.

Bei der Umsetzung lassen sich dann allerdings mehrere Schwierigkeiten feststellen. Es lässt sich beispielsweise im ausgehenden Webhook neben der HTTPS-URL keinerlei Konfiguration am Aufruf vornehmen. Dies unterbindet eine sinnvolle Möglichkeit zur Authentifizierung gegenüber dem HTTP-Endpunkt. Sie ließe sich lediglich durch einen Parameter, zum Beispiel in Form eines Authentifizierungs-Keys, implementieren. Diese Form der Authentifizierung ist allerdings im AWS API Gateway nicht vorgesehen und kann daher nicht genutzt werden. Ein Key zur Authentifizierung kann im API Gateway lediglich in Form eines HTTP-Headers hinterlegt werden. Aus diesem Grund muss auch im Event Grid der Umweg über eine FaaS-Funktion gemacht werden. In diesem können benutzerspezifische HTTP-Header hinterlegt werden, die einen authentifizierten Aufruf ermöglichen.

Ein weiteres Problem, das hinsichtlich der Inkompatibilität zwischen verschiedenen Providern bei der Nutzung von Event Grid auffällt, ist die Azure-spezifische Implementation des CloudEvents Formates. Zur internen Verarbeitung werden die Attribute des CloudEvents in die Attribute des Azure-spezifischen Formats abgebildet. Bei dieser Abbildung wird das CloudEvents Attribut *source* in die Azure-Event Attribute *topic* und *subject* zerlegt [Mice]. Diese müssen in der eingehenden Nachricht dazu mit einer Raute voneinander getrennt sein. Daher kann es vorkommen, dass eine eingehende Nachricht, die dem CloudEvent Standard entspricht, vom Event Grid allerdings abgelehnt wird. Sobald keine Raute im Attribut *source* vorhanden ist, ist dies der Fall.

Der Prototyp entgegnet diesem Problem, indem das Event, bevor es von AWS versendet wird, mit einer Raute vor der *source* versehen wird. Das Azure-Event Attribut *topic* bleibt damit leer. Ebenso wäre es möglich statt dem Endpunkt des Event Grid eine weitere Funktion zu benutzen und von dieser das angepasste Event ins Event Grid zu leiten. Dieses Vorgehen würde eine standardisierte Schnittstelle zur Azure Umgebung gewährleisten, erhöht aber gleichzeitig die Anzahl der Funktionsaufrufe.

Das Microsoft mit dem Event Grid eine Komponente zur Verfügung stellt, die genau für diesen Einsatzzweck konzipiert wurde, zeigt sich am Umgang mit nicht zustellbaren Nachrichten. Da sich durch die direkte Einbindung von Ereignisquellen ins Event Grid [Micb] einbinden lassen, entfällt die aus AWS bekannte Problematik der Nichterreichbarkeit des dezentralen Gateways. Um

die erfolgreiche Zustellung kümmert sich der Provider. Ebenso bestehen für die ins Event Grid eingebundenen Funktionen und Endpunkte eine konfigurierbare Sendungswiederholung [Mica] Dadurch kann kurzzeitige Nichterreichbarkeit des dezentralen Gateways überbrückt werden, ohne, dass es zum Verlust von Events kommt.

### 5.3.2 Anforderungsabgleich

Da die Bereiche der Anforderungen für **Anforderung 1 - Multi-Cloud Event**, **Anforderung 2 - Abfangen eines Events**, **Anforderung 3 - Benutzerdefiniertes Auslösen eines Events**, **Anforderung 8 - Verkettung von Events** und **Anforderung 9 - Vereinheitlichung von Event-Formaten** im Vergleich zum Prototyp *Zentrales Gateway* nicht verändert wurden, gilt auch für diesen dieser Anforderungsabgleich. Sie werden an dieser Stelle daher nicht erneut betrachtet.

**Anforderung 4 - Weiterleiten von Events:** Dieser Prototyp zeigt insbesondere, wie die Weiterleitung von Events durch providerspezifische Anpassungen optimiert werden kann. Event Grid von Microsoft spielt insbesondere in der Zustellung von Event seine Stärken aus, während SNS deutlich an den Anwendungsfall angepasst werden muss. Dies zeigt, wie unterschiedlich die Implementierungen der dezentralen Gateways in unterschiedlichen Cloud-Umgebungen sein können.

Der Unterschied der verwendeten dezentralen Gateways fällt auch im Zusammenspiel dieser auf. Beim Versuch eine einheitliche Schnittstelle zwischen den Gateways zu etablieren, vielen die Schwierigkeiten bei der Implementierung dieser auf. Obwohl beide Umgebung über HTTP miteinander kommunizieren musste zur erfolgreichen Kommunikation in beiden Umgebungen eine zusätzliche Funktionen eingebunden werden, anstatt die vorhandene HTTP-Schnittstelle zu verwenden. Im Falle des Prototyps arbeiten lediglich zwei Umgebungen miteinander, während es in anderen Use-Cases vorkommen kann, dass mehr als zwei Umgebungen miteinander agieren müssen. Durch die Point-to-Point Kommunikation dieses Ansatzes kann dies schnell zu einer *Integration Spaghetti* [HWB04, S. 287] führen.

**Anforderung 5 - Private Events:** Da jede Umgebung ihre Events direkt an andere Umgebungen schickt, sind private Events möglich. Anstatt diese an alle Umgebungen weiterzuleiten, werden die gewünschten Events lediglich an einzelne Umgebungen weitergeleitet.

**Anforderung 6 - Auslösen von Aktionen:** In beiden der implementierten Gateways zeigt sich, dass die interne und externe Verbreitung und Verarbeitung von Events unabhängig voneinander stattfinden. Damit ist dies der erste Prototyp, bei der die Ausführungslogik der Aktionen auf Events in der Cloud-Umgebung konfiguriert und ausgeführt wird, in der auch die entsprechende Aktion ausgeführt wird.

**Anforderung 7 - Mehrere Aktionen durch ein Event:** Aktionen auf Events lassen sich in den jeweiligen dezentralen Gateways definieren. Eine Beschränkung der maximalen Anzahl an Aktionen auf ein Event findet wenn überhaupt im Gateway statt. Weder in der AWS, noch in der Azure Umgebung findet eine Beschränkung statt. Daher ist diese Anforderung erfüllt.

**Anforderung 10 - Skalierbarkeit und Verfügbarkeit:** Alle verwendeten Komponenten sind cloud-native und werden vom Provider bereitgestellt. Die Anforderung ist dadurch uneingeschränkt erfüllt.

**Anforderung 11 - Providerunabhängigkeit und Erweiterbarkeit:** Da offene Standards für Schnittstellen fehlen oder schlecht umgesetzt wurden, kommt es an einigen Stellen zu unschönen Konstruktionen, die Events in verschiedene Formate umwandeln, oder providerspezifische Details hinzufügen. Dies ist sowohl in der Implementierung der AWS Umgebung, als auch der Azure Umgebung der Fall. Microsoft stellt mit Event Grid zwar eine Komponente zur Verfügung, die Eventbasierte Kommunikation auch über die eigene Cloud-Umgebung hinweg ermöglichen soll, durch Inkompatibilitäten lässt sich diese allerdings derzeit nur eingeschränkt nutzen. Es bleibt zu hoffen, dass bis zur finalen Veröffentlichung von Event Grid nachgebessert wird.

Die Verteilung von Events an unterschiedliche Cloud-Umgebungen stellt sich auch in diesem Prototyp als umständlich heraus. Dieser Umstand wird verschärft, sobald die Erweiterung auf weitere Provider betrachtet wird. Da einheitliche Schnittstellen fehlen, müssen für jede unterstützte Cloud-Umgebung in jeder anderen Umgebung Anpassungen vorgenommen werden, ähnlich wie bei den direkten Aufrufen.

**Anforderung 12 - Sicherheit:** Die Kommunikationsschnittstellen zwischen den Gateways wurden alle per verschlüsseltem HTTPS Standard implementiert. Zur Authentifizierung nutzt der Prototyp einen geheimen Schlüssel als Header in der HTTP-Anforderung, der sowohl im sendenden, als auch im empfangenden Gateway hinterlegt ist. Die Anforderung an die Sicherheit ist demnach erfüllt.

### 5.3.3 Variation

Neben dem Versuch eine einheitliche Schnittstelle zwischen den dezentralen Gateways zu etablieren, ließe sich auch die Kommunikation zwischen jedem Gateway-Paar einzeln regeln. Dadurch könnte beispielsweise das Event Grid eine Nachricht im SNS Format an die AWS Umgebung schicken, während es an weitere Umgebungen wieder ein anderes Nachrichtenformat nutzt. Dies würde die Erweiterbarkeit auf zusätzliche Umgebungen erschweren, gleichzeitig aber dafür sorgen, dass providerspezifische Schnittstellen unterstützt werden können.

### 5.3.4 Fazit

Dieser Prototyp mit dezentralen Gateways geht einige der Probleme an, die sich durch das zentrale Gateway ergeben haben. Das Prinzip der Entkopplung der Eventbasierten Kommunikation wurde vollumfänglich eingehalten. Welche Aktionen aufgrund eines Events ausgelöst werden wird erst in der Umgebung festgelegt, in der die entsprechende Aktion auch ausgeführt wird. Events, die interne Aktionen auslösen werden nicht mehr erst an das zentrale Gateway geleitet, um von dort aus wieder in die Cloud-Umgebung geschickt zu werden.

Insbesondere in einem Softwaresystem, indem mehrere dezentrale Gateways miteinander kommunizieren, besteht allerdings die Gefahr einer *Integration Spaghetti*, bei der ein verwirrendes Konstrukt aus Verbindung besteht [HWB04, S. 287]. Durch eine einheitliche Schnittstelle wird versucht diese Integration Spaghetti auf ein Minimum zu reduzieren, indem alle Umgebungen gleiche Kommunikationskanäle nutzen. Für Anwendungsfälle, in denen mehrere Umgebungen miteinander kommunizieren oder Umgebungen hinzugefügt oder entfernt werden, ist dieser Ansatz sinnvoll. Da die Hybrid Cloud einen der häufigsten Anwendungsfälle der Multi-Cloud darstellt und in dieser im Normalfall die einmal ausgewählten Cloud-Umgebungen lange Zeit beibehalten werden, ließe sich allerdings auch die Variation des Prototyp nutzen. In ihr wird keine standardisierte Schnittstelle



verwendet. Stattdessen wird eine providerspezifische Schnittstellen pro Umgebungs-Paar ausgewählt. Der konzeptionelle Aufwand, sowie der Implementierungsaufwand einer providerspezifischen Schnittstelle sollte den Aufwand zur Herstellung einer providerunabhängigen Schnittstelle in den meisten Fällen nicht übersteigen. Daher kann in den meisten Fällen einer Hybrid Cloud, in der typischerweise eine Public und eine Private Cloud vereint werden, Eventbasierte Kommunikation über dezentrale Gateways mit providerspezifischen Schnittstellen realisiert werden.

## 5.4 Hierarchische Struktur von Gateways

Im abschließenden Prototyp werden die beiden Ansätze des zentralen Gateways mit NATS mit dem Ansatz der dezentralen Gateways mit SNS und dem Event Gateway zu einem vereint. Innerhalb des zentralen Gateways wird dabei weiterhin auf das CloudEvents Format und das NATS-Protokoll gesetzt, während die einzelnen Adapter auf HTTP-Endpunkte oder providerspezifische Schnittstellen und Formate setzen können. Dadurch werden Unterschiede der Cloud-Umgebungen an einem Ort zentral abgefangen. Dezentrale Gateways müssen ihre Events lediglich an ein einzelnes Gateway weiterreichen, das sich um die weitere Verbreitung kümmert. Dies ist besonders in Applikationen mit mehreren Cloud-Umgebungen nützlich.

### 5.4.1 Umsetzung

Durch den Zusammenschluss der Prototypen *Dezentrale Gateways* und *Zentrales Gateway* entsteht ein neuer Prototyp mit hierarchischer Struktur zwischen den Gateways. NATS bildet weiterhin die zentrale Komponente des zentralen Gateways, wird allerdings um zwei weitere Adapter ergänzt. Statt wie bisher nur für den ausgehenden Datenverkehr zuständig zu sein, behandeln diese diesmal den eingehenden Datenverkehr. Dadurch ist es möglich, providerspezifische Schnittstellen für das zentrale Gateway bereitzustellen. Mit diesen können sowohl providerspezifische, als auch plattformsspezifische Protokolle unterstützt werden.

SNS bietet standardmäßig die Möglichkeit einen HTTP-Endpunkt als Abonnement hinzuzufügen. Da bei diesem Abonnement allerdings Daten im SNS-Nachrichtenformat verschickt werden, kann ein HTTP-Endpunkt diese Nachrichten nicht weiter verarbeiten, ohne dieses Nachrichtenformat zu unterstützen. Diese Aufgabe übernimmt nun der *SNS-to-NATS* Adapter.

Im Adapter *NATS-to-SNS* ist die Verbindung von NATS zu SNS geregelt. Diese Node.js Applikation baut dazu auf dem AWS-SDK auf. Mit diesem können Nachrichten direkt in einem SNS Topic veröffentlicht werden, ohne die Details der Verbindung kennen zu müssen. Lediglich die Informationen zur Authentifizierung müssen zur Verfügung gestellt werden.

Das dezentrale Gateway innerhalb der AWS Umgebung funktioniert identisch zu dem im Prototyp *Dezentrale Gateways* beschriebenen Aufbau. Durch den vereinfachten HTTP-Aufruf kann allerdings die Funktion *Event Sync* weggelassen werden. Dies hat nicht nur den Vorteil, dass diese Funktionsaufrufe wegfallen. Gleichzeitig kann bei HTTP-Abonnements auch genauer spezifiziert werden, welches Verhalten bei Fehlern durchgeführt werden soll [Amae]. AWS unterteilt die Zeit nach einer fehlgeschlagenen Zustellung in vier Phasen. Direkt nach der ersten fehlgeschlagenen Zustellung beginnt die Phase der sofortigen Wiederholung. In ihr wird, wie der Name bereits andeutet direkt,

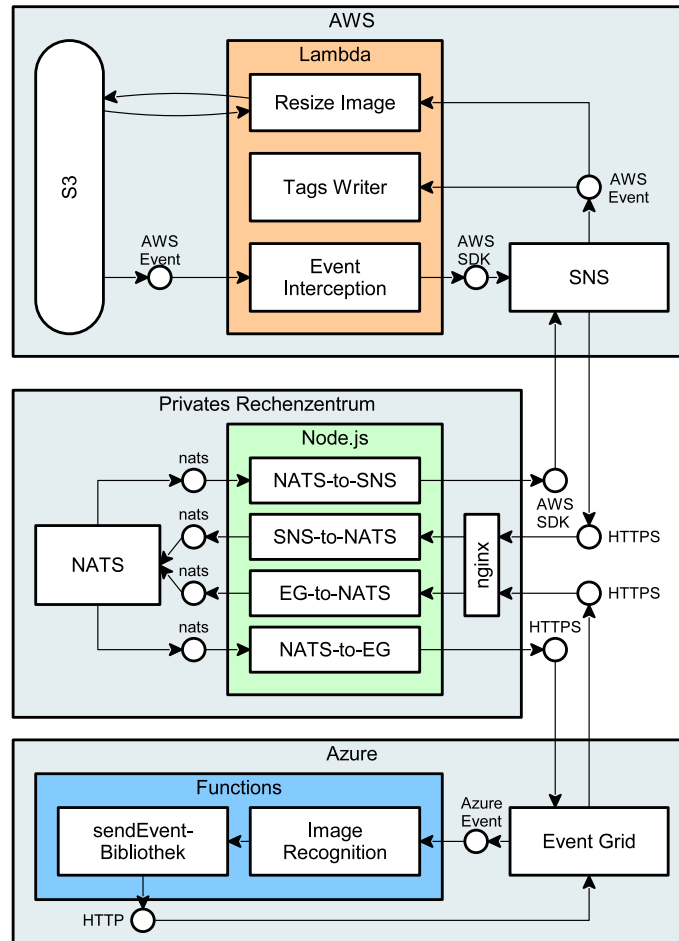


Abbildung 5.4: Prototyp einer Umgebung mit hierarchischen Gateways

ohne weitere Verzögerung ein erneuter Zustellversuch durchgeführt. Dieser Phase schließen sich drei weitere Phasen an, in denen die Verzögerung zwischen den Versuchen verlängert wird. Die Anzahl der Zustellversuche pro Phase lässt sich konfigurieren.

Da es vorkommen kann, dass das zentrale Gateway kurzzeitig ausfällt oder nicht erreichbar ist, kann eine regelmäßige Zustellwiederholung bereits zu einer erfolgreichen Zustellung führen. Bei einem längeren Ausfall ist allerdings Vorsicht geboten. Aufgrund der Vielzahl von potenziellen Ereignissen, die während des Ausfalls aufgetreten sind, kann es zu einer Überlastung des zentralen Gateways kommen, wenn diese Nachrichten in einem unkontrollierten Strom zugestellt werden. Gegebenenfalls ist es besser, die fehlgeschlagenen Nachrichten auch in diesem Aufbau nach einer geringen Anzahl an Zustellversuchen in einer *Dead Letter Queue* zu speichern und diese Warteschlange bei der Wiederherstellung geregelt abzarbeiten.

Vergleichbare Änderungen zum Prototyp *Dezentrale Gateways* ergeben sich auch für die Azure Umgebung. Das Event Grid kommuniziert nicht mehr über eine angepasste Funktion mit dem zentralen Gateway, sondern nutzt dafür die Möglichkeit des HTTPS-Endpunktes. Zur Überprüfung, ob Event Grid berechtigt ist, diesen Endpunkt zu nutzen, führt Azure erst einmal eine Validierung durch [Micc]. Dazu sendet Event Grid ein Event eines speziellen Typs an den Endpunkt. Dieses

Event enthält einen Code. Erst durch die Antwort mit diesem Code an die Anfrage sendet Event Grid alle Events an den Endpunkt. Auch diese Funktionalität muss vom *EG-to-NATS* Adapter implementiert werden.

Sowohl für die hier beschriebene Zustellwiederholung, als auch für die Validierung des HTTP-Endpunktes gibt es jeweils in AWS beziehungsweise Azure ein entsprechendes Gegenstück, das identisch funktioniert und daher nicht weiter ausgeführt wird.

Event Grid setzt beim Einsatz eines HTTPS-Endpunktes ein validiertes Zertifikat voraus. Aus diesem Grund wird *nginx* als Reverse Proxy eingesetzt [NGI]. Dieser erlaubt die Verbindung per HTTPS und tunnelt diese über HTTP an die EG-to-NATS und SNS-to-NATS Adapter. Dadurch besteht lediglich innerhalb des Servers eine unverschlüsselte Verbindung.

Weiterhin ist der Anschluss eines dezentralen Gateways einer Cloud-Umgebung an den NATS Server möglich. Dazu muss dieses Gateway selbstverständlich eine NATS Schnittstelle implementieren. Dies ist beispielsweise bei Kubeless der Fall. Kubeless ist ein auf Kubernetes basierendes Framework, dass die Bereitstellung einer Serverless Umgebung, beispielsweise in einer Private Cloud, ermöglicht [Kuba].

Über ein sogenanntes *PubSub Event* ist es möglich die Kubeless Infrastruktur direkt mit dem NATS Server zu koppeln und Funktionen per Trigger mit einem NATS Subject zu verbinden [Kubb]. Leider entspricht dieses Mapping von einer Funktion auf ein komplettes Subject nicht dem Mapping des NATS Transport Bindings von CloudEvents [CNCE]. Bei diesem werden sämtliche Events über ein Subject geschickt. Eine Filterung, welche Funktionen getriggert werden, findet erst später statt. Nichtsdestotrotz zeigt dieses Vorgehen einen möglichen prinzipiellen Aufbau, um Umgebungen ohne speziellen Cloud-Adapter an den Prototypen anzuschließen.

### 5.4.2 Anforderungsabgleich

Wie in den beiden vorherigen Prototypen werden auch hier die Anforderungen **Anforderung 1 - Multi-Cloud Event**, **Anforderung 2 - Abfangen eines Events**, **Anforderung 3 - Benutzerdefiniertes Auslösen eines Events**, **Anforderung 8 - Verkettung von Events** und **Anforderung 9 - Vereinheitlichung von Event-Formaten** nicht weiter ausgeführt, da sie bereits in einem der vorherigen Prototyp verifiziert wurden und sich dieser Anforderungsabgleich zu diesem Prototyp nicht unterscheidet.

**Anforderung 4 - Weiterleiten von Events:** Einer der größten Vorteile dieses Prototyps wird bei der Betrachtung von Abbildung 5.4 deutlich. Innerhalb der jeweiligen Cloud-Umgebungen besteht ein in sich abgeschlossenes System, mit dem Eventbasierte Kommunikation innerhalb dieser Umgebung möglich ist. Die einzelnen Umgebungen können daher bei einer Nichterreichbarkeit des zentralen Gateways unabhängig voneinander agieren. Lediglich alle externen Events und Aktionen können nicht angesprochen werden.

Durch die unterschiedlichen Adapter für jede Cloud-Umgebung ergibt sich ein weiterer Vorteil. Da sich jeder der Adapter mit einer eigenen Client-ID am NATS Server anmeldet, wird die erfolgreiche Verteilung der Events an die unterschiedlichen Umgebungen gespeichert. Sollte eine Umgebung nicht erreichbar sein, oder es zu anderen Fehlern kommen, so wird die Zustellung an diese eine Umgebung nicht als erfolgreich markiert und kann später wiederholt werden. Auf Umgebungen, in die das Event erfolgreich zugestellt werden konnte, hat dies keine Auswirkungen.

Außerdem ergibt sich durch die speziellen Cloud-Adapter die Möglichkeit providerspezifische Angebote direkt zu nutzen. Der Vorteil hieraus wird bei der SNS Implementierung deutlich. Für die HTTP-Abonnements, die bisher aufgrund des plattformspezifischen Kommunikationsprotokolls nicht genutzt werden konnten, besteht die Möglichkeit der Zustellwiederholung. Dadurch kann diese einfach genutzt werden und muss nicht mit Hilfe eines *Dead Letter Queues* implementiert werden.

**Anforderung 5 - Private Events:** Private Events sind in der momentanen Implementierung der Prototypen nicht unterstützt. Eine Erweiterung wäre allerdings einfach möglich. Die vorher definierten Events müssen entweder bereits vom Gateway oder vom Cloud-Adapter als privat markiert und über ein anderes Subject verschickt werden. Auf dieses Subject dürfen, identisch zum Vorgehen in Prototyp mit NATS als zentralem Gateway, lediglich die auf diese Events zugriffsberechtigten Umgebungen Zugriff haben.

**Anforderung 6 - Auslösen von Aktionen:** Die Entkopplung dieses Prototyp wird auch bei der Betrachtung von Änderungen an der Programmlogik deutlich. In der Beispielapplikation ist es in diesem Prototypen beispielsweise möglich, die Bilderkennung nicht mehr auf die komprimierten Bilder, sondern die Originaldateien auszuführen und lediglich Änderungen innerhalb der Azure Umgebung durchzuführen. Dies wird dadurch ermöglicht, da alle Events synchronisiert werden und die finale Entscheidung, welche Aktionen ausgeführt wird, erst in der Azure Umgebung gefällt wird.

**Anforderung 7 - Mehrere Aktionen durch ein Event:** Da jedes Gateway selbst entscheidet, welche Aktionen ausgeführt werden, entscheidet diese auch über die Anzahl der maximal auslösbaren Aktionen. Weder in SNS, noch im Event Grid bestehen dabei Limitierungen.

**Anforderung 10 - Skalierbarkeit und Verfügbarkeit:** Auch bei diesem Prototyp bleibt der schwerwiegendste Nachteil des zentralen Gateways bestehen. Das zentrale Gateway stellt weiterhin einen Single-Point-of-Failure dar. Auch wenn die Kommunikation im Fehlerfall innerhalb der Umgebungen bestehen bleibt, können die Umgebungen nicht mehr untereinander kommunizieren. Hochverfügbarkeit des zentralen Gateways ist daher zwingend erforderlich, um den Anforderungen zu genügen.

**Anforderung 11 - Providerunabhängigkeit und Erweiterbarkeit:** Die einzelnen Umgebungen müssen nichts über andere Umgebungen wissen. Weder sind spezifische Anpassungen an einzelnen Schnittstellen vorzunehmen, noch müssen die Umgebungen jede andere Umgebung kennen, die von der Applikation genutzt wird. Beim Anschluss einer weiteren Cloud-Umgebung an die Applikation müssen daher lediglich im zentralen Gateway Anpassungen vorgenommen werden. Die einzelnen Umgebungen sind weitestgehend voneinander entkoppelt.

In der Anforderung zur Providerunabhängigkeit und Erweiterbarkeit wird die Erweiterbarkeit des Systems explizit auch auf Eigenentwicklungen gerichtet. Diese Eigenentwicklungen können sowohl Applikationen, die komplett unabhängig von einer Cloud-Umgebung entwickelt wurden, als auch selbst betriebene Cloud-Umgebungen sein. Daher scheint es sinnvoll für den Betrieb solcher Eigenentwicklungen auf eine standardisierte Schnittstelle zu setzen. Ein Adapter zur Kommunikation über eine HTTP-Verbindung mit CloudEvents könnte eine solche Schnittstelle darstellen.

**Anforderung 12 - Sicherheit:** Auch in diesem Prototypen trägt die Implementation des Kommunikationskanal einen entscheidenden Faktor zur Sicherheit des Gesamtsystems bei. Da dieser Kommunikationskanal einmalig zwischen Cloud und zentralem Cloud-Adapter implementiert

werden muss, kann hierbei ohne großen Portierungsaufwand auf providerspezifische Anpassungen gesetzt werden. Das bedeutet, dass jede Schnittstelle ohne großen Aufwand eine optimierte Authentifizierung zur Cloud-Umgebungen nutzen kann.

### 5.4.3 Variation

Alternativ zum hier vorgestellten Prototyp ließe sich das zentrale Gateway auch selbst als serverlose Applikation implementieren. Statt NATS könnte beispielsweise Event Grid als zentrale Komponente verwendet werden. Verbindungen zu anderen Umgebungen könnten dann mit Funktionen hergestellt werden. Dies hätte die Vorzüge, dass sämtliche Vorteile einer serverlosen Applikation – beispielsweise die automatisierte Bereitstellung und Skalierung durch den Provider – auch vollumfänglich für das zentrale Gateway gelten.

Allerdings besteht in diesem Ansatz die Gefahr des Vendor Lock-In. Bei der Auswahl des Providers müsste bedacht werden, dass jede Applikation, die auf Basis dieses Gateways entwickelt wird, auch zwingend diesen Cloud-Provider einsetzen muss. Insbesondere für einen providerunabhängigen Ansatz stellt dies daher keine zufriedenstellende Lösung dar.

### 5.4.4 Fazit

Bei diesem Prototyp werden die Vorzüge der dezentralen Gateways und des zentralen Gateways miteinander vereint. Durch eine offene Kommunikationsspezifikation innerhalb der NATS Umgebung, besteht eine Schnittstelle, mit der alle Cloud-Umgebungen, entweder mit oder ohne Cloud-Adapter angeschlossen werden können. Eine direkte Übersetzung der Protokolle eines Cloud-Providers in die entsprechenden Protokolle eines anderen Cloud-Providers findet an keiner Stelle mehr statt. Sowohl in den Nachrichtenformaten, als auch im Nachrichtenkanal wurde mit der CloudEvents Spezifikation und dem NATS Protokoll jeweils ein Standard etabliert, mit der jede Cloud-Umgebung kommunizieren kann.

Dadurch lässt es dieser Ansatz zu selbst den in Abschnitt 2.1.5 genannten Use Case der impliziten Modellierung eines Multi-Cloud Events von Wurster et al. abbilden. Jede unterstützte Cloud-Umgebung benötigt dazu eine entsprechende Implementierung eines dezentralen Gateways und Cloud Adapter, um mit dem zentralen Gateway kommunizieren zu müssen. Eine direkte Kommunikation mit anderen Cloud-Umgebungen findet nicht statt. Demnach müssen auch keine umgebungsspezifischen Kommunikationsprotokolle unterstützt werden.



## 6 Fazit und Ausblick

In dieser Arbeit wurden die Möglichkeiten Eventbasierter Kommunikation in Multi-Cloud-Umgebungen des Serverless Computing untersucht. Die Anwendungsfälle, in denen eine Multi-Cloud-Umgebung genutzt werden kann, sind zahlreich. Neben einer provisorischen Nutzung von zwei Cloud-Umgebungen zum Umzug eines Services zu einem anderen Cloud-Provider [Pet11, S. 63], bis hin zur dauerhaften Nutzung mehrerer Cloud-Umgebungen, um deren Vorzüge, zum Beispiel besonders performante Komponenten zu einem günstigen Preis, voll auszunutzen [WM14]. Einer der häufigsten Anwendungsfälle einer Multi-Cloud dürfte in den Bereich einer Hybrid Cloud fallen, sowohl aus gesetzlichen Bestimmungen, wie datenschutzrechtlichen Bedenken, als auch um Lastspitzen auszugleichen [Pet13, S. 1].

So vielseitig, wie die Anwendungsfälle einer Multi-Cloud-Umgebung sein können, so vielseitig stellen sich auch die Lösungsansätze zur Eventbasierten Kommunikation in Multi-Cloud-Umgebungen dar. Aufgrund der Vielzahl der Anwendungsfälle lässt sich kein allgemeingültiges, zu priorisierendes Konzept festlegen. Die Anforderungen an das zu erarbeitende System wurden daher anhand eines Referenzszenarios ausformuliert. Dieses stellt allerdings lediglich die Grundlage zur Verfügung. Die Erweiterbarkeit und Übertragung in andere Anwendungsfälle wurde stets berücksichtigt.

Der direkte Aufruf einer Funktion über HTTP-Requests oder andere providerspezifische Schnittstellen aus einer Umgebung heraus kann beispielsweise bei Umstrukturierungen der Applikation als provisorischer Aufruf einer Funktion in einer anderen Umgebung völlig ausreichend sein. Als dauerhafte Lösung eignet sich dieses Vorgehen allerdings nicht. Dies liegt insbesondere an der dadurch entstehenden starken Kopplung der beiden Umgebungen und den guten Alternativen.

Sowohl eine Multi-Cloud-Umgebung mit zentralem Gateway, als auch Umgebungen mit dezentralen Gateways haben Vorteile. Der zentrale Ansatz sollte bei Applikationen gewählt werden, wenn der überwiegende Teil der Aktionen auf ein Event in einer anderen Umgebung ausgelöst wird, während der Aufruf mit dezentralen Gateways gewählt werden sollte, wenn viele Events interne Aktionen auslösen. Beim Hinzufügen oder Entfernen von Cloud-Umgebungen zeigt sich die Stärke des zentralen Gateways, da die neue Umgebung hierbei lediglich mit dem zentralen Gateway kommunizieren muss. Allerdings kommt es in den wenigsten Anwendungsfällen zu regelmäßigen Wechseln der Cloud-Umgebungen.

Da das zentrale Gateway prinzipbedingt von sämtlichen Events durchlaufen werden muss, bevor daraus resultierende Aktionen ausgeführt werden, folgt beim Ausfall dieser zentralen Komponente der Ausfall der kompletten Umgebung. Auch Aktionen die durch ein Event ausgelöst worden wären, das in derselben Umgebung entstand, werden bei Nichterreichbarkeit der zentralen Komponente nicht ausgeführt. Um dies zu umgehen, wurde das Konzept der hierarchischen Gateways vorgestellt. Dieses vereint dezentrale Gateways mit einem zentralen Gateway. Dadurch kann jede Umgebung eigenständig agieren und muss ihre ausgelösten Events nur an ein einziges Gateway weiterleiten. Dessen Aufgabe ist es das Event an alle weiteren Umgebungen zu leiten. Dadurch kann eine

vollständige Entkopplung der einzelnen Umgebungen voneinander erreicht werden. Diese wird vor allem bei Umgebungen benötigt, bei denen viele Cloud-Umgebungen miteinander kommunizieren müssen.

Aus den Konzepten wurden in dieser Arbeit Prototypen abgeleitet, mit deren Hilfe die Umsetzbarkeit der aufgestellten Konzepte erfolgreich überprüft und demonstriert werden konnte. Dabei wurden Schwächen und Vorteile der jeweiligen Implementierungen deutlich. Bei der Implementierung wurde wieder auf das Referenzszenario zurückgegriffen, indem es in allen vier Prototypen erfolgreich als Multi-Cloud Service implementiert wurde. Diese Prototypen zeigen lediglich eine der vielen Möglichkeiten auf, wie die Konzepte implementiert werden können. Da für jedes der vorgestellten Konzepte mehrere Gateways und Implementierungsansätze bestehen, gibt es auch eine große Anzahl an Möglichkeiten, die Konzepte umzusetzen.

Die Arbeit zeigt auf, wie die *CloudEvents* Spezifikation der *Serverless Working Group* eine gute Grundlage zur Providerunabhängigkeit von Eventbasierter Kommunikation bereitstellt. Insbesondere durch den Verzicht einer einheitlichen Standardisierung zur Authentifizierung über HTTP-Anfragen ist selbst bei konsequenter Umsetzung der Spezifikation noch keine einheitliche Kommunikation möglich. Diese Standardisierung zu übernehmen bleibt daher Aufgabe der jeweiligen Implementierung selbst.

Ebenfalls hervorzuheben bleiben Microsofts Bemühungen um das *Azure Event Grid*. Dieses ermöglicht eine Nutzung als dezentrales Gateway und strebt dabei eine Konformität zur *CloudEvents* Spezifikation an. An ein paar Stellen besteht bezüglich dieser Konformität noch Nachbesserungsbedarf. Dieses Vorgehen ist auf jeden Fall als fortschrittlich zu bezeichnen. Das Nachziehen mit eigenen, standardkonformen Gateways von weiteren Public Cloud-Providern würde die Implementierung von Eventbasierter Kommunikation in Multi-Cloud-Umgebungen erheblich vereinfachen.

Auf diesen Konzepten eventbasierter Multi-Cloud-Umgebungen lässt sich ein breites Spektrum an weiteren Forschungen und Untersuchungen aufbauen. Bisher nicht betrachtet wurden die Auswirkungen auf die Performanz der unterschiedlichen Konzepte bezogen auf die Eventzustellung. Zu einer solchen Betrachtung gehört insbesondere auch die Auswirkung der Nutzung unterschiedlicher Gateways. Aufgrund der breiten Masse an zur Verfügung stehenden Gateways, konnte eine solche Analyse in dieser Arbeit nicht in der Tiefe durchgeführt werden. Neben der Performance stellen auch die Kosten einen wichtigen Faktor zur Auswahl eines Gateways dar. Diese können ebenfalls analysiert werden.

Woo und Mirkovic zeigen in ihrer Arbeit [WM14] die Grundlage um mit Hilfe einer Multi-Cloud, die mehrere Public Cloud-Provider nutzt, Kosten zu sparen und dabei die zur Verfügung gestellte Performance zu verbessern. Auf Grundlage der Konzepte dieser Arbeit können weitere Untersuchungen stattfinden, inwieweit sich das Vorgehen von Woo und Mirkovic auf das Serverless Computing übertragen lässt.

Außerhalb rein konzeptioneller Betrachtungen bildet diese Arbeit eine Grundlage zur weiteren Erarbeitung konkreter Use-Cases. In weiterer Forschung kann anhand von konkreten Fallbeispielen gezeigt werden in welchen Fällen sich welches Konzept als besonders geeignet herausgestellt hat. Aus einer gewissen Anzahl an Fallbeispielen ließen sich Leitlinien entwickeln, die zur Planung einer eventbasierten Multi-Cloud-Umgebung herangezogen werden können. Dieses Vorgehen kann Softwareentwicklern und -architekten den Einstieg in die Materie erleichtern und ihnen bereits in der Planungsphase eine konkrete Richtung vorgeben.



---

Wurster et al. nutzen in ihrer Arbeit [WBK+18] das OpenTOSCA Ökosystem, um serverlose Applikationen providerunabhängig zu modellieren und anschließend zu deployen. Damit ließen sich auch Multi-Cloud-Umgebungen deployen. Aufgrund der fehlenden Unterstützung von Multi-Cloud Events ist es der Applikation nicht möglich eventbasierte Aufrufe zu tätigen. Dabei zeigt dieser Anwendungsfall, dass es möglich sein muss ein System zu erstellen, dass nahezu beliebige Umgebungen einbinden kann. Auf Grundlage dieser Arbeit lässt sich ein Konzept erschließen, mit dem es möglich ist, dieses Herausforderung anzugehen.



## Literaturverzeichnis

- [AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Aug. 1977 (zitiert auf S. 36).
- [Amaa] Amazon Web Services. *Amazon Simple Notification Service*. URL: <https://aws.amazon.com/de/sns/> (zitiert auf S. 43).
- [Amab] Amazon Web Services. *Amazon SNS-Filtrerrichtlinien für Abonnements*. URL: [https://docs.aws.amazon.com/de\\_de/sns/latest/dg/sns-subscription-filter-policies.html](https://docs.aws.amazon.com/de_de/sns/latest/dg/sns-subscription-filter-policies.html) (zitiert auf S. 61).
- [Amac] Amazon Web Services. *Amazon-Ressourcennamen (ARNs) und AWS-Service-Namespaces*. URL: [https://docs.aws.amazon.com/de\\_de/general/latest/gr/aws-arns-and-namespaces.html](https://docs.aws.amazon.com/de_de/general/latest/gr/aws-arns-and-namespaces.html) (zitiert auf S. 49).
- [Amad] Amazon Web Services. *AWS Lambda-Wiederholungsverhalten*. URL: [https://docs.aws.amazon.com/de\\_de/lambda/latest/dg/retries-on-errors.html](https://docs.aws.amazon.com/de_de/lambda/latest/dg/retries-on-errors.html) (zitiert auf S. 51).
- [Amae] Amazon Web Services. *Festlegen von Amazon SNS-Zustellungswiederholungsrichtlinien für HTTP/HTTPS-Endpunkte*. URL: [https://docs.aws.amazon.com/de\\_de/sns/latest/dg/DeliveryPolicies.html](https://docs.aws.amazon.com/de_de/sns/latest/dg/DeliveryPolicies.html) (zitiert auf S. 65).
- [Amaf] Amazon Web Services. *Identitäten (Benutzer, Gruppen und Rollen)*. URL: [https://docs.aws.amazon.com/de\\_de/IAM/latest/UserGuide/id.html](https://docs.aws.amazon.com/de_de/IAM/latest/UserGuide/id.html) (zitiert auf S. 57).
- [Ama19] Amazon Web Services. *AWS Lambda*. 2019. URL: <https://aws.amazon.com/de/lambda/> (zitiert auf S. 16).
- [Apa] Apache Software Foundation. *Apache Kafka - Documentation*. URL: <https://kafka.apache.org/documentation/> (zitiert auf S. 44).
- [Apab] Apache Software Foundation. *Apache Kafka - Introduction*. URL: <https://kafka.apache.org/intro> (zitiert auf S. 41).
- [BCC+17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, P. Suter. „Serverless Computing: Current Trends and Open Problems“. In: S. Chaudhary, G. Somani, R. Buyya. *Research Advances in Cloud Computing*. Springer Singapore, 2017, S. 1–20. ISBN: 9789811050268 (zitiert auf S. 11, 15, 16).
- [Bus] Bustle. *Bustle Case Study*. URL: <https://aws.amazon.com/de/solutions/case-studies/bustle/> (zitiert auf S. 32).
- [Cloa] Cloud Native Computing Foundation. *Cloud Native Computing Foundation*. URL: <https://www.cncf.io/> (zitiert auf S. 21).
- [Clob] Cloud Native Computing Foundation. *Cluster Upgrading*. URL: [https://nats-io.github.io/docs/nats\\_admin/upgrading\\_cluster.html](https://nats-io.github.io/docs/nats_admin/upgrading_cluster.html) (zitiert auf S. 44).

- [Cloc] Cloud Native Computing Foundation. *NATS Introduction*. URL: <https://nats-io.github.io/docs/> (zitiert auf S. 41).
- [Clod] Cloud Native Computing Foundation. *NATS Streaming Concepts*. URL: [https://nats-io.github.io/docs/nats\\_streaming/intro.html](https://nats-io.github.io/docs/nats_streaming/intro.html) (zitiert auf S. 56).
- [Cloe] Cloud Native Computing Foundation. *Sandbox Projects*. URL: <https://www.cncf.io/sandbox-projects/> (zitiert auf S. 21).
- [Clouf] Cloud Native Computing Foundation. *Securing NATS*. URL: [https://nats-io.github.io/docs/nats\\_server/securing\\_nats.html](https://nats-io.github.io/docs/nats_server/securing_nats.html) (zitiert auf S. 57).
- [CNCa] CNCF Serverless Working Group. *CloudEvents - Version 0.1*. URL: <https://github.com/cloudevents/spec/blob/v0.1/spec.md> (zitiert auf S. 22).
- [CNCb] CNCF Serverless Working Group. *CloudEvents Primer*. URL: <https://github.com/cloudevents/spec/blob/master/primer.md> (zitiert auf S. 12, 21, 36).
- [CNCc] CNCF Serverless Working Group. *HTTP Transport Binding for CloudEvents - Version 0.1*. URL: <https://github.com/cloudevents/spec/blob/v0.1/http-transport-binding.md> (zitiert auf S. 51).
- [CNCd] CNCF Serverless Working Group. *JSON Event Format for CloudEvents - Version 0.1*. URL: <https://github.com/cloudevents/spec/blob/v0.1/json-format.md> (zitiert auf S. 22).
- [CNCE] CNCF Serverless Working Group. *NATS Transport Binding for CloudEvents*. URL: <https://github.com/cloudevents/spec/blob/v0.2/nats-transport-binding.md> (zitiert auf S. 55, 67).
- [CNCf] CNCF Serverless Working Group. *References*. URL: <https://github.com/cloudevents/spec/blob/v0.1/about/references.md> (zitiert auf S. 49).
- [CNC19] CNCF Serverless Working Group. *CloudEvents*. 2019. URL: <https://cloudevents.io/> (zitiert auf S. 21).
- [DWC10] T. Dillon, C. Wu, E. Chang. „Cloud Computing: Issues and Challenges“. In: *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. Apr. 2010, S. 27–33. DOI: [10.1109/AINA.2010.187](https://doi.org/10.1109/AINA.2010.187) (zitiert auf S. 20).
- [Eiv17] A. Eivy. „Be Wary of the Economics of Serverless Cloud Computing“. In: *IEEE Cloud Computing* 4.2 (März 2017), S. 6–12. ISSN: 2325-6095. DOI: [10.1109/MCC.2017.32](https://doi.org/10.1109/MCC.2017.32) (zitiert auf S. 18).
- [EN10] O. Etzion, P. Niblett. *Event Processing in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2010. ISBN: 9781935182214 (zitiert auf S. 17, 31, 38, 53).
- [Eva18] K. Evans. *CNCF to Host NATS*. März 2018. URL: <https://www.cncf.io/blog/2018/03/15/cncf-to-host-nats/> (zitiert auf S. 41).
- [FHT+12] A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, C. Sheridan. „OPTIMIS: A holistic approach to cloud service provisioning“. In: *Future Generation Computer Systems* 28.1 (2012), S. 66–77. ISSN: 0167-739X. DOI: [10.1016/j.future.2011.05.022](https://doi.org/10.1016/j.future.2011.05.022) (zitiert auf S. 19).

- [FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Vienna, 2014. ISBN: 9783709115688 (zitiert auf S. 36).
- [GB14] N. Grozev, R. Buyya. „Inter-Cloud Architectures and Application Brokering: Taxonomy and Survey“. In: *Softw. Pract. Exper.* 44.3 (März 2014), S. 369–390. ISSN: 0038-0644. DOI: [10.1002/spe.2168](https://doi.org/10.1002/spe.2168) (zitiert auf S. 19).
- [Goo19] Google. *Google Cloud Functions*. 2019. URL: <https://cloud.google.com/functions/> (zitiert auf S. 16).
- [HWB04] G. Hohpe, B. Woolf, K. Brown. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. A Martin Fowler signature book. Addison-Wesley, 2004. ISBN: 9780321200686 (zitiert auf S. 25, 30–32, 35, 36, 45, 61, 63, 64).
- [Ins17] Institute of Architecture of Application Systems (IAAS). *OpenTOSCA - Open Source TOSCA Ecosystem*. 2017. URL: <https://www.iaas.uni-stuttgart.de/forschung/projekte/opentosca/> (zitiert auf S. 29).
- [iRo] iRobot. *iRobot Ready to Unlock the Next Generation of Smart Homes Using the AWS Cloud*. URL: <https://aws.amazon.com/de/solutions/case-studies/irobot/> (zitiert auf S. 32).
- [Kuba] Kubeless. *Kubeless - The Kubernetes Native Serverless Framework*. URL: <https://kubeless.io/> (zitiert auf S. 16, 67).
- [Kubb] Kubeless. *PubSub events*. URL: <https://kubeless.io/docs/pubsub-functions/> (zitiert auf S. 67).
- [MG11] P. M. Mell, T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Techn. Ber. Gaithersburg, MD, United States, 2011 (zitiert auf S. 19, 20).
- [Mica] Microsoft Azure. *Richtlinien für unzustellbare Nachrichten und Wiederholungen*. URL: <https://docs.microsoft.com/de-de/azure/event-grid/manage-event-delivery> (zitiert auf S. 63).
- [Micb] Microsoft Azure. *Was ist Azure Event Grid?* URL: <https://docs.microsoft.com/de-de/azure/event-grid/overview> (zitiert auf S. 23, 62).
- [Micc] Microsoft Azure Documentation. *Empfangen von Ereignissen an einem HTTP-Endpunkt*. URL: <https://docs.microsoft.com/de-de/azure/event-grid/receive-events> (zitiert auf S. 66).
- [Micd] Microsoft Azure Documentation. *HTTP-Trigger und -Bindungen in Azure Functions*. URL: <https://docs.microsoft.com/de-de/azure/azure-functions/functions-bindings-http-webhook> (zitiert auf S. 51).
- [Mice] Microsoft Azure Documentation. *Verwenden des CloudEvents-Schemas mit Event Grid*. URL: <https://docs.microsoft.com/de-de/azure/event-grid/cloudevents-schema> (zitiert auf S. 62).
- [Mic06] B. M. Michelson. „Event-driven architecture overview“. In: *Patricia Seybold Group* 2.12 (2006), S. 10–1571 (zitiert auf S. 17, 31).
- [Mic19] Microsoft. *Azure Functions*. 2019. URL: <https://azure.microsoft.com/de-de/services/functions/> (zitiert auf S. 16).

- [Mor16] K. Morris. *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media, 2016. ISBN: 9781491924396 (zitiert auf S. 20).
- [MSE+16] G. McGrath, J. Short, S. Ennis, B. Judson, P. Brenner. „Cloud Event Programming Paradigms: Applications and Analysis“. In: *Proc. IEEE 9th Int. Conf. Cloud Computing (CLOUD)*. Juni 2016, S. 400–406. DOI: [10.1109/CLOUD.2016.0060](https://doi.org/10.1109/CLOUD.2016.0060) (zitiert auf S. 18).
- [Nag15] V. Nagran. *Fanout S3 Event Notifications to Multiple Endpoints*. 24. Juni 2015. URL: <https://aws.amazon.com/de/blogs/compute/fanout-s3-event-notifications-to-multiple-endpoints/> (zitiert auf S. 47).
- [NGI] NGINX Inc. *NGINX Reverse Proxy*. URL: <https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/> (zitiert auf S. 67).
- [Opea] OpenFaaS Project. *OpenFaaS - connector-sdk*. URL: <https://github.com/openfaas-incubator/connector-sdk> (zitiert auf S. 20).
- [Opeb] OpenFaaS Project. *OpenFaaS - Introduction*. URL: <https://docs.openfaas.com/> (zitiert auf S. 16, 20).
- [Ope19] OpenWhisk. *OpenWhisk*. 2019. URL: <https://github.com/apache/incubator-openwhisk> (zitiert auf S. 16).
- [Pet11] D. Petcu. „Portability and Interoperability between Clouds: Challenges and Case Study“. In: *Towards a Service-Based Internet*. Hrsg. von W. Abramowicz, I. M. Llorente, M. Surrige, A. Zisman, J. Vayssière. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 62–74. ISBN: 978-3-642-24755-2 (zitiert auf S. 29, 71).
- [Pet13] D. Petcu. „Multi-Cloud: Expectations and Current Approaches“. In: *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds. MultiCloud '13*. Prague, Czech Republic: ACM, 2013, S. 1–6. ISBN: 978-1-4503-2050-4. DOI: [10.1145/2462326.2462328](https://doi.org/10.1145/2462326.2462328) (zitiert auf S. 12, 19, 71).
- [RC17] M. Roberts, J. Chapin. *What is Serverless?* 2017 (zitiert auf S. 11, 18).
- [Rig19] RightScale from Flexera. *State of the Cloud Report 2019. As Cloud Use Grows, Organizations Focus on Cloud Costs and Governance*. Jan. 2019 (zitiert auf S. 28).
- [Sera] Serverless, Inc. *Serverless Documentation*. URL: <https://serverless.com/framework/docs/> (zitiert auf S. 20).
- [Serb] Serverless, Inc. *Serverless Event Gateway*. URL: <https://serverless.com/event-gateway/> (zitiert auf S. 23).
- [Serc] Serverless, Inc. *Serverless Event Gateway - Clustering*. URL: <https://github.com/serverless/event-gateway/blob/master/docs/clustering.md> (zitiert auf S. 44).
- [Serd] Serverless, Inc. *Serverless.yml Reference*. URL: <https://serverless.com/framework/docs/providers/aws/guide/serverless.yml/> (zitiert auf S. 21).
- [Ser19] Serverless, Inc. *Event Gateway*. 2019. URL: <https://github.com/serverless/event-gateway> (zitiert auf S. 23, 59).
- [Sti18] M. Stigler. *Beginning Serverless Computing - Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. 2018. URL: <https://www.apress.com/us/book/9781484230831> (zitiert auf S. 12, 15–18).

- [WBK+18] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, V. Yussupov. „Modeling and Automated Deployment of Serverless Applications using TOSCA“. 2018 (zitiert auf S. 29, 73).
- [Wen] S. Wendt. *Home of Fundamental Modeling Concepts - Notation Reference*. URL: [http://www.fmc-modeling.org/notation\\_reference](http://www.fmc-modeling.org/notation_reference) (zitiert auf S. 35).
- [WM14] S. S. Woo, J. Mirkovic. „Optimal application allocation on multiple public clouds“. In: *Computer Networks* 68 (2014). Communications and Networking in the Cloud, S. 138–148. ISSN: 1389-1286. DOI: [10.1016/j.comnet.2013.12.001](https://doi.org/10.1016/j.comnet.2013.12.001) (zitiert auf S. 29, 71, 72).

Alle URLs wurden zuletzt am 12.06.2019 geprüft.





### **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.

---

Ort, Datum, Unterschrift