Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Erstellen, Bereitstellen und Betreiben von Serverless Cloud-Anwendungen mit TOSCA

Tobias Mathony

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Michael Wurster, M.Sc.
Beginn am:	3. November 2017
Beendet am:	3. Mai 2018

Kurzfassung

Trotz der relativ kurzen Existenz des Serverless Computing Ansatzes, haben sich bereits mehrere Serverless Plattformen verschiedener Cloud Provider etabliert. Aktuell setzen verschiedene Serverless Plattformen das Serverless Computing Paradigma allerdings proprietär um, was Unternehmen vor der Nutzung von Serverless Computing aufgrund des dadurch entstehenden Vendor-Lock-Ins zurückhalt. Daher ist die plattformübergreifende Kompatibilität von Serverless Funktionen und Events erstrebenswert.

TOSCA, ein Standard der OASIS-Gruppe, adressiert Portabilität und automatisiertes Management von Cloud Anwendungen und beschreibt diese unabhängig von Cloud Providern, deren Infrastrukturen und Technologien. Mittels der Konzepte von TOSCA erstellt diese Arbeit einen Ansatz, bei dem Serverless Funktionen und Events provider-unabhängig beschrieben werden. Dadurch wird die Kompatibilität dieser Funktionen und Events zwischen verschiedenen Serverless Plattformen in TOSCA ermöglicht. Außerdem wird durch das Implementieren von provider-spezifischen Management-Operationen via TOSCA das automatisierte Management von Serverless Anwendungen in einer TOSCA-konformen Laufzeitumgebung ermöglicht.

Abstract

Despite the short existence of the Serverless Computing approach, already several Serverless Platforms of different Cloud Providers have established themselves in the market. Currently, different Serverless Platforms implement the Serverless Computing paradigm proprietary. Therefore, enterprises are held back from the use of Serverless Computing due to the resulting vendor lock-in. Hence, the cross-platform compatibility of Serverless Functions and Events is desirable.

TOSCA, a standard from OASIS, addresses portability and automated management of Cloud Applications and describes these Cloud Applications in a provider-agnostic manner.
By the means of the concepts of TOSCA, this thesis proposes a concept, which describes Serverless Functions and Events abstracted from specific Serverless Platforms. As a result, Serverless Functions and Events are cross-platform compatible to any Serverless Platform. Furthermore, by the means of implementing platform-specific management operations via TOSCA, this concept enables automated management of these Serverless Applications in a TOSCA runtime, including automated provisioning and deprovisioning.

Inhaltsverzeichnis

1 Einleitung **13**

2 Grundlagen und verwandte Arbeiten **15**

2.1 Serverless Computing . . . 15

2.2 TOSCA . . . 27

3 Ansatz **33**

3.1 Motivation . . . 33

3.2 Identifikation der Komponenten einer Serverless Anwendung . . . 34

3.3 Serverless Anwendungen in TOSCA . . . 35

3.4 Programmiermodell für Serverless Anwendungen in TOSCA . . . 46

4 Implementierung **49**

4.1 Prototypische Implementierung des Konzepts via OpenTOSCA . . . 49

4.2 Validierung der Implementierung . . . 55

5 Zusammenfassung und Ausblick **59**

Literaturverzeichnis **61**

Abbildungsverzeichnis

2.1 Beispiel einer traditionellen Drei-Schichten-Architektur nach [Fow16] 16
2.2 Beispielhafte Architektur einer Serverless Anwendung nach [Fow16] 17
2.3 Inkompatibilitätsgründe von Serverless Funktionen und Events zwischen verschiedenen Serverless Plattformen 18
2.4 High-Level-Architektur von OpenWhisk nach [Kro16] 21
2.5 OpenWhisk Deployment Topologie nach [Apa18d] 22
2.6 Beispielhafter Ablauf einer Interaktion mit dem Serverless Framework 25
2.7 Projektstruktur einer beispielhaften, provider-unabhängigen Serverless Anwendung nach [Sti18] 26
2.8 Aufbau eines TOSCA Service Templates nach [OAS13a] 28
2.9 Metamodell des TOSCA Typsystems nach [BBKL14] 29
2.10 Beispielhaftes TOSCA Topology Template einer simplen Anwendung 30

3.1 Übersicht über die Vorgehensweise des Ansatzes 33
3.2 Metamodell der Anatomie einer Serverless Anwendung 34
3.3 Abbildung der Serverless Komponenten auf das TOSCA Topologie Metamodell . 36
3.4 Übersicht über die Kategorisierung von Events in Serverless Anwendungen . . 37
3.5 Vereinfachtes, beispielhaftes Topology Template einer Serverless Anwendung in TOSCA 38
3.6 Beziehungen zwischen Node Type, Interface und Management-Operationen . . 45
3.7 Plattformübergreifende Kompatibilität von Serverless Funktionen dank TOSCA 46

4.1 Vereinfachte Darstellung des OpenTOSCA Ökosystems 53
4.2 Topologie unserer beispielhaften Serverless Anwendung in Winery 54
4.3 Visuelle Repräsentation des für die Validierung der Implementierung erstellten Topology Templates 56

Verzeichnis der Listings

2.1 Hello-World Funktion in JavaScript für OpenWhisk 19
2.2 Interaktion mit OpenWhisk via CLI um eine Action hochzuladen und aufzurufen 22
2.3 Serverless Framework Service für Apache OpenWhisk nach [Ser18] 24
2.4 Code der provider-unabhängigen index.js Datei nach [Sti18] 27
2.5 Code der Provider.js Datei in AWS Lambda nach [Sti18] 27

3.1 Properties Definition des ServerlessFunktion Node Types 40
3.2 Properties Definition des OpenWhiskPlattform Node Types 44

4.1 Vereinfachte Java-Klasse mit den zu implementierenden Management-Operationen 50
4.2 Implementierung der deployFunction Management-Operation in Java 51
4.3 Implementierung der Verknüpfung von Event und Serverless Funktion 52
4.4 Antwort des CLI auf die Abfrage nach der Aktivierung unserer Serverless Funktion 57

Tabellenverzeichnis

2.1 Vergleich des Funktionsaufbaus verschiedener Serverless Plattformen 18

3.1 Vergleich der benötigten Parameter, um eine Serverless Funktion auf verschiedenen Serverless Plattformen programmatisch zu deployen 39
3.2 Vergleich der für die verschiedenen Serverless Plattformen benötigten Parameter, um einen HTTP Event zu deployen 41
3.3 Vergleich der für die verschiedenen Serverless Plattformen benötigten Parameter, um einen Publish-Subscribe-Messaging Event zu deployen 43

1 Einleitung

Serverless Computing bezeichnet ein aufstrebendes Cloud Computing Paradigma, bei dem der Cloud Provider eine Laufzeitumgebung für die Ausführung serverseitiger Logik bereitstellt [BCC+17]. Diese serverseitige Logik wird mittels kurzlebiger, zustandsloser Funktionen implementiert, welche Serverless Funktionen genannt werden. Die Architektur einer Serverless Anwendung basiert dabei auf Events, wie beispielsweise der zeitplanmäßigen Ausführung einer Funktion oder die Ausführung einer Funktion als Reaktion auf eine Änderung in einer Datenbank [MB17]. Wird eine Serverless Funktion durch einen Event ausgelöst, so allokiert der Cloud Provider dynamisch die benötigten Rechenkapazitäten für die Ausführung der Funktion. Dabei werden nur die für den Zeitraum der Funktionsausführung genutzten Rechenkapazitäten in Rechnung gestellt [Eiv17].

Trotz der relativ kurzen Existenz des Serverless Computing Ansatzes haben sich bereits mehrere Serverless Plattformen verschiedener Cloud Provider etabliert. Aktuell setzen verschiedene Serverless Plattform das Serverless Computing Paradigma jedoch proprietär um [Clo18]. Unter anderem unterscheidet sich das Angebot und die Anbindung von Events, sowie die Funktionssignaturen und das Deployment von Serverless Funktionen [Clo18]. Dies resultiert in einer Inkompatibilität von Serverless Funktionen und Events zwischen verschiedenen Serverless Plattformen. Dies hat zur Folge, dass sich Unternehmen für die Nutzung des Serverless Computing Ansatzes auf eine Serverless Plattform festlegen müssen, woraus ein Vendor-Lock-In bei der jeweiligen Serverless Plattform entsteht. Diesen gilt es, aufgrund der dadurch entstehenden Ungeschütztheit vor Preisanhebungen und der Machtlosigkeit bei Zuverlässigkeitsproblemen des Cloud Providers, zu vermeiden [AFG+10].

TOSCA, ein Standard der OASIS-Gruppe, adressiert Portabilität und automatisiertes Management von Cloud Anwendungen [OAS13a]. Dafür wird die Struktur einer Cloud Anwendung in TOSCA als Topologie dargestellt. Um das automatisierte, portable Management einer Cloud Anwendung zu ermöglichen, werden sogenannte Management-Pläne in standardisierten Workflow-Sprachen erstellt. Dabei sind grundsätzlich Management-Pläne für das Provisionieren, Verwalten und Deprovisionieren einer Cloud Anwendung vorgesehen [OAS13a]. Die Management-Pläne bestehen dabei aus dem Aufrufen von Management-Operationen der einzelnen Komponenten der beschriebenen Cloud Anwendung, welche ebenfalls in standardisierten Technologien implementiert werden [BBLS12] [OAS13a].

Diese Arbeit nutzt TOSCA zur Erstellung eines Konzepts, bei dem Serverless Funktionen und Events einer Serverless Anwendung provider-unabhängig und abstrahiert von Serverless Plattformen beschrieben werden. Dadurch werden Serverless Funktionen und Events zwischen Serverless Plattformen kompatibel. Dabei werden die Komponenten einer Serverless Anwendung auf TOSCA Elemente abgebildet. Das Erstellen der Serverless Anwendung wird dabei mittels einer TOSCA Topologie ermöglicht, welche die Komponenten und deren Beziehungen zueinander repräsentiert.

Durch die Implementierung der Management-Operationen in den jeweils provider-spezifischen Serverless Plattform Komponenten wird erwirkt, dass jede Serverless Plattform Komponente in TOSCA die Logik besitzt, um Serverless Funktionen und Events programmatisch deployen zu können. Dies ermöglicht, dass eine Serverless Funktion oder ein Event via TOSCA je nach Belieben und den Anforderungen entsprechend auf verschiedenen Serverless Plattformen automatisiert deployed werden kann. Das erstellte Konzept wird abschließend prototypisch implementiert, um dies zu validieren.

Gliederung

Diese Arbeit ist wie folgt gegliedert:

Kapitel 2 – Grundlagen und verwandte Arbeiten: Hier werden die Grundlagen dieser Arbeit beschrieben. Dies umfasst die grundlegenden Konzepte des Serverless Computing Ansatzes und TOSCA. Außerdem beinhaltet dieses Kapitel im Bezug auf diese Thesis verwandte Arbeiten.

Kapitel 3 – Ansatz: Dieses Kapitel umfasst den Ansatz, welcher mittels der Konzepte von TOSCA eine plattformübergreifende Kompatibilität von Serverless Funktionen und Events erwirkt. Außerdem wird das automatisierte Management von Serverless Anwendungen via TOSCA ermöglicht.

Kapitel 4 – Implementierung: Dieses Kapitel beschreibt die prototypische Implementierung des in Kapitel 3 erstellten Konzeptes. Des Weiteren wird die Implementierung anhand einer beispielhaften Serverless Anwendung und dem automatisierten Management dieser Anwendung in OpenTOSCA validiert.

Kapitel 5 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Grundlagen und verwandte Arbeiten

Dieses Kapitel beinhaltet die für das Verständnis dieser Arbeit elementaren Grundlagen. Zunächst werden in Abschnitt 2.1 die Grundlagen des Serverless Computing Ansatzes aufgezeigt. Dies beinhaltet außerdem eine Übersicht über die für die Implementierung des Konzepts genutzte Serverless Plattform Apache OpenWhisk[1] und die Darlegung verwandter Arbeiten dieser Thesis. Anschließend werden in Abschnitt 2.2 die für diese Arbeit relevanten Konzepte von TOSCA erläutert.

2.1 Serverless Computing

Serverless Computing, oft einfach Serverless genannt, bezeichnet ein Cloud Computing Paradigma, bei dem der Cloud Provider eine Laufzeitumgebung für das Ausführen serverseitiger Logik bereitstellt. Die serverseitige Logik wird dabei mittels kurzlebiger, zustandsloser Funktionen implementiert, welche Serverless Funktionen genannt werden. Die Ausführungsumgebung für diese Funktionen wird dabei als Function-as-a-Service bezeichnet [SMM18]. Serverless Computing verfolgt das Ideal einer event-basierten Anwendungsarchitektur. Diese Anwendungsarchitektur entspringt dem aktuellen Trend von Architekturen, bestehend aus Containern und Microservices, welche zunehmend Akzeptanz in Unternehmen finden [BCC+17]. Aufgrund dessen werden Funktionen in einer Serverless Anwendung als Reaktion auf das Auslösen bestimmter Events ausgeführt. Wird eine Serverless Funktion ausgelöst, allokiert der Cloud Provider dynamisch die für die Ausführung der Funktion benötigten Rechenkapazitäten. Nach Beendigung der Ausführung werden die Rechenkapazitäten wieder freigegeben.

Dabei müssen, im Gegensatz zu traditionellen Cloud Computing Angeboten, keine Rechenkapazitäten im Voraus reserviert werden. Bisherige Cloud Computing Angebote erforderten dies, wodurch oftmals aufgrund der Fehleinschätzung der Serverauslastung Probleme entstanden. Reservierte man zu viele Rechenkapazitäten, so entstanden dadurch hohe Kosten für ungenutzte Ressourcen. Bei der Unterschätzung der Serverauslastung, und der dadurch verbundenen Reservierung von zu wenigen Rechenkapazitäten, entstanden hohe Antwortzeiten [FLR+14].

Auch muss der Zeitraum, in der keine Rechenkapazität zur Ausführung der Serverless Funktionen genutzt wird, nicht bezahlt werden. Der Entwickler, beziehungsweise Operator der Serverless Anwendung, hat dabei keine Kontrolle über die Rechenkapazitäten, deren Allokation und Skalierung. Serverless Computing schafft im Vergleich zu bisherigen Cloud Computing Paradigmen eine noch höhere Abstraktionsebene. Der Entwickler muss lediglich die Serverless Funktionen

[1] http://openwhisk.incubator.apache.org

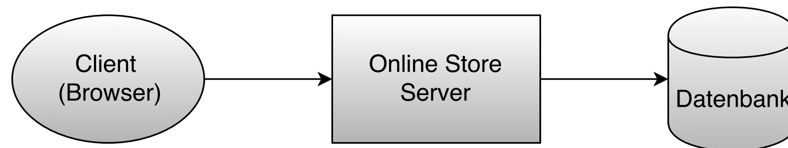

Abbildung 2.1: Beispiel einer traditionellen Drei-Schichten-Architektur nach [Fow16]

implementieren. Der Cloud Provider ist verantwortlich für das Provisionieren, Überwachen, Warten, Skalieren und die Fehlertoleranz der Ressourcen [BCC+17].

Durch die höhere Abstraktionsebene und ohne die Notwendigkeit, Rechenkapazitäten für die Ausführung der Funktionen allokieren zu müssen, wird das Erstellen von Prototypen deutlich beschleunigt [SMM18]. Durch die event-basierte Architektur einer Serverless Anwendung und der Abrechnung nach tatsächlich genutzter Rechenkapazität bringt der Serverless Computing Ansatz in verschiedenen Bereichen Vorteile gegenüber herkömmlichen Cloud Computing Paradigmen mit [SMM18]. Dies gilt unter anderem für Cloud Anwendungen im Bereich Internet der Dinge, bei welchen die verbundenen Geräte, wie Sensoren und Aktuatoren, sporadisch Daten ausgeben [SMM18]. Als Reaktion darauf können Serverless Funktionen ausgeführt werden. Außerdem eignet sich der Serverless Computing Ansatz für Web Anwendungen mit leichtgewichtigen Backend-Funktionen [SMM18]. Des Weiteren eignen sich Serverless Funktionen als Konnektor zwischen verschiedenen Services eines Cloud Providers, welche als Event dienen [SMM18]. Da die Kontrolle über die Zuteilung von Rechenkapazitäten für die Ausführung der Serverless Funktionen bei der Serverless Plattform liegt, garantiert diese auch eine den Anfragen entsprechende Skalierung.

2.1.1 Architektur von Serverless Anwendungen

Die Architektur einer Serverless Anwendung unterscheidet sich maßgeblich von traditionellen Anwendungsarchitekturen. Im Gegensatz zu der in Abbildung 2.1 dargestellten, traditionellen Drei-Schichten-Architektur wird die Logik einer Serverless Anwendung auf möglichst viele, kleine, kurzlebige Funktionen aufgeteilt. Jede Serverless Funktion soll dabei bestenfalls nur eine Funktionalität erfüllen [Cui18]. Eine beispielhafte Umwandlung der in Abbildung 2.1 gezeigten, traditionellen Drei-Schichten-Architektur zu einer Serverless Architektur ist in Abbildung 2.2 zu sehen. Charakteristisch ist hierbei die Aufteilung der Logik auf mehrere, kleine Funktionen, welche nur noch eine Funktionalität erfüllen, verglichen zu der Drei-Schichten-Architektur, bei der sich der Großteil der Logik in einer Komponente befindet. Die Architektur einer Serverless Anwendung orientiert sich an der ereignisgesteuerten Architektur, bei welcher die einzelnen Komponenten mittels Ereignissen ausgelöst und gesteuert werden [MSJL06] [MB17]. Konkret bedeutet dies, dass Serverless Funktionen mit Events verbunden werden und als Reaktion auf die Auslösung dieser Events ausgeführt werden. Beispielsweise kann eine Serverless Funktion als Reaktion auf einen neuen Eintrag einer Datenbank oder auf das Hochladen einer Datei auf einen Blobspeicher ausgeführt werden [MB17]. Für das Erledigen wiederkehrender Aufgaben, wie beispielsweise das Erstellen von Backups, können Serverless Funktionen außerdem mit Events verbunden werden, welche auf Basis eines Zeitplans ausgelöst werden. Die Serverless Architektur orientiert sich, mit dem Kompositum von vielen, kleinen Funktionen und der Auslösung dieser durch Events, an einem modernen Architekturstil. Dadurch sind Serverless Anwendungen sehr feingranular und lassen sich effizienter skalieren als monolithische Anwendungen [Eiv17]. Dies

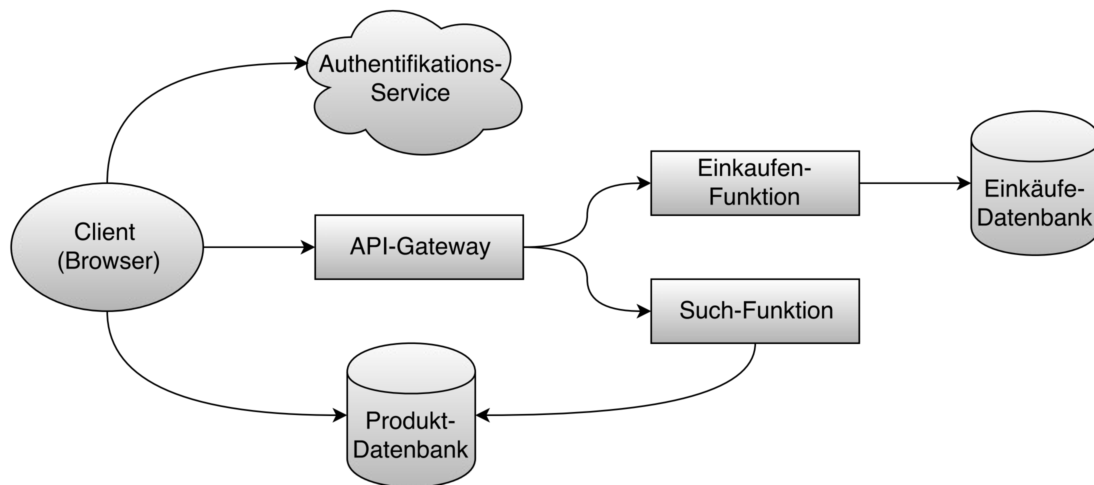

Abbildung 2.2: Beispielhafte Architektur einer Serverless Anwendung nach [Fow16]

wird anhand der in Abbildung 2.1 dargestellten Anwendungsarchitektur deutlich. Die Anwendung, welche vereinfacht dargestellt ist, beinhaltet jegliche serverseitige Logik, wie beispielsweise das Suchen oder Kaufen eines Produkts, in der Komponente des Online Store Servers. Verzeichnet nun die Suchfunktion einen extrem hohen Ansturm, ohne dass die Einkauffunktion genutzt wird, muss dennoch die gesamte Komponente des Online Store Servers hochskaliert werden, um die Anfragen prozessieren zu können [VGC+15]. Bei einer Serverless Architektur, wie in Abbildung 2.2, ist diese Logik auf verschiedene Serverless Funktionen aufgeteilt. Erfährt nun die Suchfunktion einen hohen Workload, so wird lediglich die Serverless Funktion, welche die Logik der Suchfunktion implementiert, hochskaliert. Dies ermöglicht eine effiziente Skalierung auf Funktionsebene und resultiert in einem Kostenvorteil gegenüber monolithischen Cloud Anwendungen. Des Weiteren bieten Serverless Anwendungen, aufgrund der Kapselung der Logik auf viele Funktionen kleinere Angriffsflächen für Attacken auf die Anwendung [Cui18]. Außerdem erleichtert die feingranulare Architektur einer Serverless Anwendung das Austauschen von Funktionen [VGC+15]. Im Gegensatz bedeutet dies jedoch auch, dass Serverless Anwendungen, welche aus vielen Serverless Funktionen bestehen, schwer zu verwalten sind. Da jede Serverless Funktion im Bestfall nur eine Funktionalität implementiert, besteht eine Serverless Anwendung grundsätzlich aus vielen Komponenten, wodurch Serverless Anwendungen schnell komplex werden und dadurch schwer zu verwalten sind [Cui18]. Dies liegt unter anderem daran, dass im Normalfall mehrere Entwickler an einer Serverless Anwendung arbeiten, wodurch das Entdecken der bereits implementierten Funktionalitäten ohne ein durchdachtes Kommunikations- oder Katalogsystem schwerfällt [Cui18]. Dies kann außerdem in Duplikaten von Serverless Funktionen resultieren [Cui18].

2.1.2 Serverless Funktionen

Eine Serverless Funktion ist eine kleine, kurzlebige, zustandslose Funktion welche auf einer Serverless Plattform gehosted ist und auf dieser als Reaktion auf Events ausgeführt wird. Serverless Funktionen implementieren serverseitige Logik. Aktuell unterscheiden sich Serverless Funktionen verschiedener Serverless Plattformen beispielsweise in der Funktionssignatur und der Übergabe

OpenWhisk	AWS Lambda	Google Cloud Functions

function main(params){
return{message: "..."}
}
	exports.myHandler = function
	(event, context, callback) => {
	callback(null, "...");
	}
		exports.myHandler =
		(req, res) => {
		res.send("...");
		};

Tabelle 2.1: Vergleich des Funktionsaufbaus verschiedener Serverless Plattformen

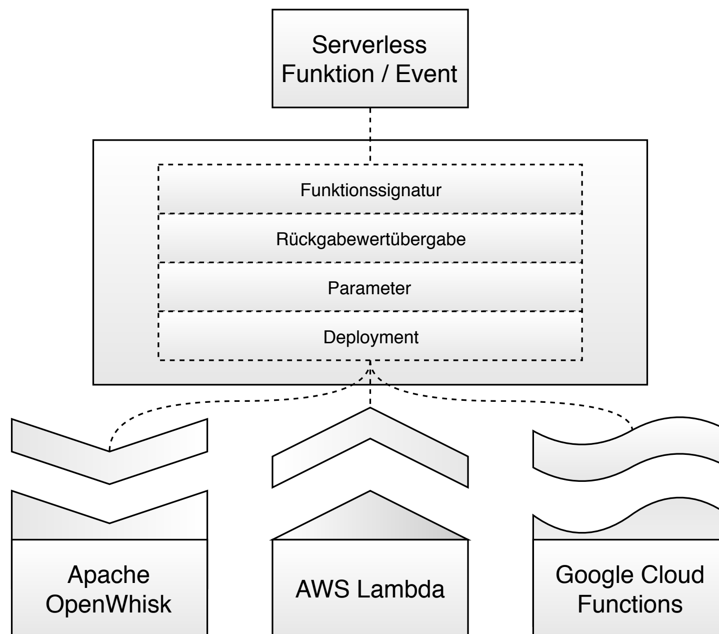

Abbildung 2.3: Inkompatibilitätsgründe von Serverless Funktionen und Events zwischen verschiedenen Serverless Plattformen

des Rückgabewerts. Tabelle 2.1 vergleicht den Funktionsaufbau einer JavaScript Serverless Funktion zwischen drei bekannten Implementierungen einer Serverless Plattform: Apache OpenWhisk, Google Cloud Functions[2] und Amazon Web Services (AWS) Lambda[3]. AWS Lambda und Google Cloud Functions haben dabei einen ähnlichen Aufbau, wobei mit dem Schlüsselwort *exports* die Einstiegsmethode bestimmt wird, welche bei dem Aufruf einer Serverless Funktion gestartet wird. OpenWhisk hat dahingegen eine *main*-Methode, welche den Einstiegspunkt bei dem Aufruf einer Serverless Funktion repräsentiert. Neben der Funktionssignatur und der Übergabe des Rückgabewerts unterscheidet sich außerdem das Deployment von Serverless Funktionen und Events je nach Serverless Plattform. Abbildung 2.3 zeigt die aktuellen Inkompatibilitäten von Serverless Funktionen und Events verschiedener Serverless Plattformen auf. Das Deployment unterscheidet sich je nach Serverless Plattform unter anderem aufgrund der provider-spezifischen Parameter, die je nach Serverless Plattform für das Deployment benötigt werden.

[2] https://cloud.google.com/functions/
[3] https://aws.amazon.com/de/lambda/

Listing 2.1 Hello-World Funktion in JavaScript für OpenWhisk

```
/**
 * Hello world as an OpenWhisk action.
 */
function main(params) {
   var name = params.name || 'World';
   return {payload: 'Hello, ' + name + '!'};
}
```

Ein Aufruf einer Serverless Funktion erfolgt in einer Serverless Plattform durch das Auslösen eines mit dieser Funktion verbundenen Events. Ein Event kann beispielsweise eine HTTP[4] Anfrage, eine zeitplanmäßige Ausführung oder die Änderung in einem Speicherservice des Cloud Providers sein [KY17] [Eiv17].

Eine Serverless Funktion soll so klein wie möglich sein und bestenfalls nur eine Funktionalität erfüllen. Die Ausführung, sowie die Skalierung von Serverless Funktionen erfolgt auf Abruf [FIMS17]. Da die Skalierung vom Cloud Provider verwaltet wird und auf Abruf erfolgt, können Kunden ihre Ressourcen nicht über- oder unterprovisionieren [Ama16]. Wird eine Serverless Funktion ausgelöst, allokiert der Cloud Provider dynamisch die für die Ausführung benötigten Rechenkapazitäten. Nach Beendigung der Ausführung werden die Rechenkapazitäten wieder freigegeben. Dabei hat der Entwickler, beziehungsweise Operator keinerlei Einsicht in die Allokation der Rechenkapazitäten für die Serverless Funktionen. Dies bedeutet, dass für jede weitere Ausführung einer bereits ausgeführten Funktion eine andere Rechenkapazität zugeteilt werden kann. Daher kann sich eine Serverless Funktion nicht darauf verlassen, den Zustand einer vorher ausgeführten Funktion zu kennen und muss demnach zustandslos sein. Das Bezahlmodell für Serverless Funktionen basiert auf millisekunden-genauer Abrechnung der tatsächlichen Ausführungsdauer [BCF+17]. Dies bedeutet, dass keine Kosten anfallen, wenn eine Funktion nicht ausgeführt wird und stellt einen Vorteil im Vergleich zu Bezahlmodellen traditioneller Cloud Computing Angebote dar. Die Ausführungsdauer einer Serverless Funktion ist typischerweise auf fünf Minuten limitiert. Wird diese Ausführungsdauer überschritten, erfolgt die Beendigung der Funktion mit einer Fehlermeldung [Fow16]. Als erste Serverless Plattform hat die Apache OpenWhisk Plattform die maximal erlaubte Ausführungsdauer einer Serverless Funktion von fünf auf zehn Minuten erhöht. Listing 2.1 zeigt den Programmcode einer beispielhaften Serverless Funktion in JavaScript[5] auf der OpenWhisk Plattform, welche „Hello World“ zurückgibt, falls kein Eingabeparameter mitgegeben wird. Existiert ein Eingabeparameter, wird dieser anstatt „World“ zurückgegeben.

Momentan von den gängigen Serverless Plattformen unterstützte Programmiersprachen für Serverless Funktionen sind unter anderem JavaScript (Node.js[6]), Java[7], Python[8] und PHP[9].

[4] https://www.rfc-editor.org/rfc/rfc2616.txt
[5] https://www.javascript.com
[6] https://nodejs.org/en/
[7] https://java.com/en/
[8] https://www.python.org
[9] http://www.php.net

Um in einer Serverless Funktion Third-Party-Libraries zu nutzen, können Serverless Funktionen mit den Quelldateien der Third-Party-Libraries und einer Konfigurationsdatei zu einem ZIP-Archiv gebündelt werden. Im Folgenden nennen wir dies eine ZIP-Funktion. Dies ist beispielsweise gängig für Node.js Serverless Funktionen, welche externe NPM[10] Module nutzen. Die konkrete Bündelung der Serverless Funktion und deren Abhängigkeiten zu einer ZIP-Funktion ist abhängig von der Serverless Plattform und der Laufzeitumgebung der Serverless Funktion.

2.1.3 Serverless Plattformen

Für die Ausführung der Serverless Funktionen stellt der Cloud Provider eine Laufzeitumgebung, nachfolgend Serverless Plattform genannt, bereit. Für die Ausführung muss eine Serverless Funktion jedoch zunächst auf die Serverless Plattform deployed werden. Die Server werden vor dem Entwickler, beziehungsweise Operator versteckt, die Allokation und Skalierung der Ressourcen wird dabei dynamisch und nicht einsehbar vom Cloud Provider durchgeführt. Dadurch wird der Cloud eine zusätzliche Abstraktionsebene verliehen [Kno16].

Serverless Plattformen ermöglichen demnach die Ausführung von Funktionen ohne die Administration oder Provisionierung von Servern durch den Entwickler [Fow16]. Auch die Skalierung der Funktionen, sowie die Wartung der Serverinfrastruktur, werden vom Cloud Provider übernommen. Das umfasst auch die Installation von Sicherheitspatches [For17]. Dies erlaubt dem Entwickler, sich vollumfänglich auf die pure Entwicklung des Funktionscodes zu fokussieren. Dadurch kann die Produkteinführungszeit von Anwendungen signifikant verkürzt werden. Laut einer Studie von Microsoft[11] ist sogar eine Verkürzung von bis zu 67% möglich [Mic17].

Im Folgenden wird anhand Apache OpenWhisk die Funktionsweise und das Konzept einer bekannten Implementierung einer Serverless Plattform erläutert. Apache OpenWhisk wird schließlich für die prototypische Implementierung des im Zuge dieser Arbeit erstellten Konzepts genutzt. Apache OpenWhisk ist eine Open-Source Serverless Plattform der Apache Software Foundation[12]. Unter anderem basiert die IBM Cloud Functions[13] Serverless Plattform auf Apache OpenWhisk. Da der gesamte OpenWhisk-Stack Open-Source ist, können OpenWhisk-Instanzen sowohl in der privaten, als auch öffentlichen Cloud und damit auch in der Hybrid-Cloud ausgeführt werden [BCC+16] [Apa17a].

OpenWhisk führt Funktionen als Reaktion auf Events oder direkte Auslösung via HTTP Anfragen aus [Apa17a]. Eine Serverless Funktion wird in OpenWhisk als *Action* und der Auslöser dieser Funktion als *Trigger* bezeichnet [Apa18a]. *Rules* werden in OpenWhisk die Assoziationen zwischen einem Trigger und einer Action genannt, die definieren, welche Action als Reaktion auf einen Trigger ausgeführt werden soll [Apa18a]. Des Weiteren unterstützt OpenWhisk das Kompositum mehrerer Actions als geordnete Menge in Form einer sogenannten *Sequence*. Diese Sequences ermöglichen, dass eine Action den Rückgabewert einer vorherigen Action der Sequence

[10] https://www.npmjs.com
[11] https://www.microsoft.com/de-de/
[12] https://www.apache.org
[13] https://www.ibm.com/cloud/functions

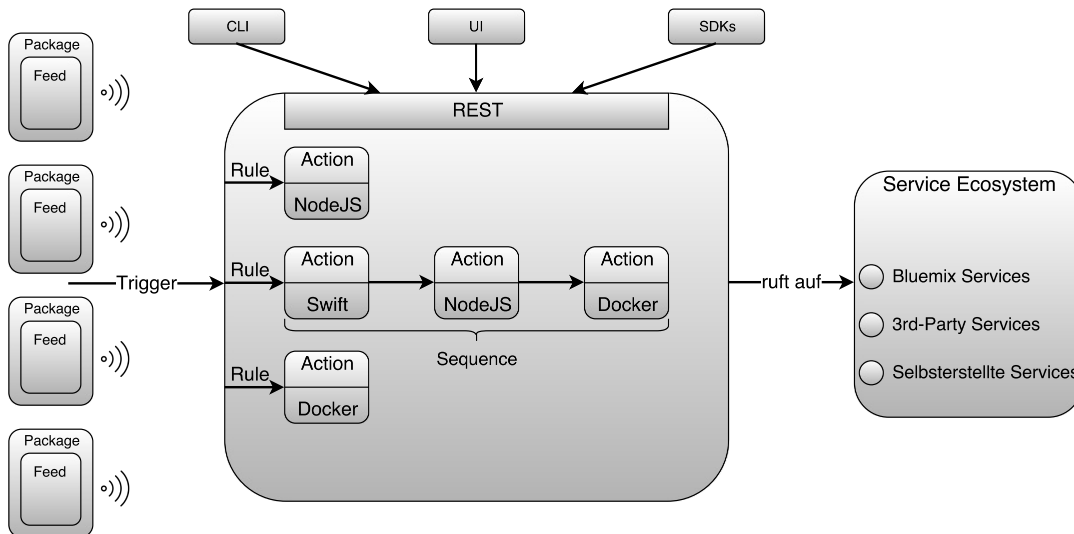

Abbildung 2.4: High-Level-Architektur von OpenWhisk nach [Kro16]

als Eingabeparameter nutzt [BCC+16]. Außerdem können Actions zu sogenannten *Packages* gebündelt werden, um ein Service-Ökosystem zu erstellen, welches dann von anderen Actions aufgerufen werden kann [YCCI16]. Die beschriebenen Beziehungen zwischen Actions, Triggers, Packages, Sequences und Rules ist der High-Level-Architektur von OpenWhisk in Abbildung 2.4 zu entnehmen. OpenWhisk unterstützt derzeit Actions in JavaScript, Swift[14], Python, Java, PHP, Docker[15], Go[16] und nativen Binaries [Apa18b]. Entwickler können mit OpenWhisk via *Representational State Transfer*[17] (REST) Application Programming Interface (API) oder verschiedenen Software Development Kits (SDKs), beispielsweise für JavaScript oder Swift, interagieren [YCCI16]. Außerdem existiert zur Interaktion mit OpenWhisk ein Command-Line-Interface (CLI), das *OpenWhisk CLI* [BCF+17]. Eine beispielhafte Interaktion, bei der mittels CLI eine Action deployed und aufgerufen wird, ist Listing 2.2 zu entnehmen. Vor der Ausführung dieser Befehle wurde die Funktion aus Listing 2.1 lokal zu einer Datei namens „hello.js“ gespeichert.

Was bei der Ausführung der in Listing 2.2 dargestellen Befehle in OpenWhisk passiert, genauer gesagt bei dem Aufrufen einer Action (*wsk action invoke*), ist im Folgenden erläutert und mittels Abbildung 2.5 visuell geschildert. Da die API von OpenWhisk auf dem *Hypertext Transfer Protocol* (HTTP) basiert, ist der Befehl um eine Action zu erstellen lediglich eine HTTP Anfrage an des OpenWhisk System [Tho16]. Diese wird mit NGINX[18], einer Webserver-Software, verarbeitet (Schritt 1), welche die HTTP Anfrage an den *Controller* weiterleitet (Schritt 2), der die Implementierung der OpenWhisk REST API beinhaltet [Tho16]. Basierend auf der Methode der HTTP Anfrage unterscheidet der Controller, was die Anfrage bedeutet und was getan werden muss, um diese zu verarbeiten [Tho16]. Das Aufrufen einer Action ist eine HTTP POST Anfrage an eine

[14]https://developer.apple.com/swift/
[15]https://www.docker.com
[16]https://golang.org
[17]http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[18]https://www.nginx.com

Listing 2.2 Interaktion mit OpenWhisk via CLI um eine Action hochzuladen und aufzurufen

```
$ wsk action create hello hello.js

$ wsk action invoke hello --result --param name Tobias
{
   "payload": "Hello, Tobias!"
}
```

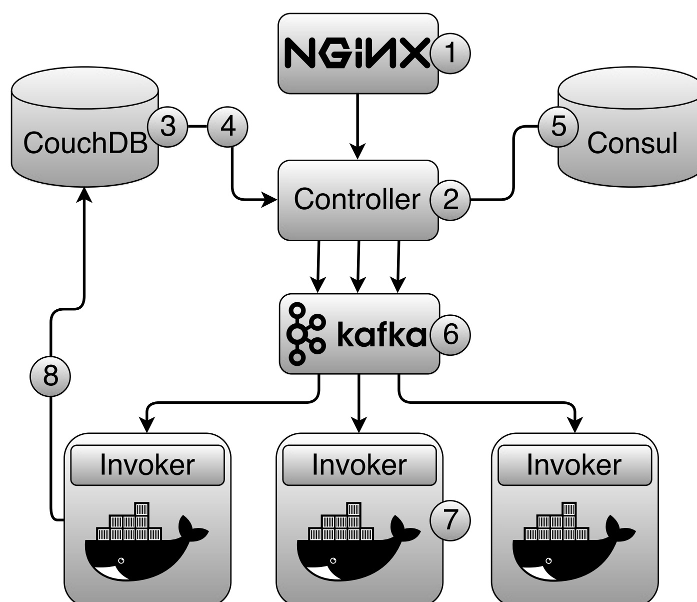

Abbildung 2.5: OpenWhisk Deployment Topologie nach [Apa18d]

bereits bestehende Action. Anschließend prüft der Controller die Authentifikation und Authorisierung des Anfragenstellers, beziehungsweise der Anfrage: die mitgegebenen Anmeldedaten werden mit der Datenbank abgeglichen (Schritt 3), danach wird geprüft, ob der Nutzer die entsprechenden Rechte besitzt, um die Action aufrufen zu dürfen [Tho16]. Ist die Prüfung erfolgreich, wird der Funktionscode und die mitgegebenen Parameter der Action aus der Datenbank geladen (Schritt 4). Um die Action nun tatsächlich auszuführen, werden *Invokers* benötigt, welche den Code ausführen können [Tho16]. Mittels *Consul*[19], welches ein Diensteverzeichnis bereitstellt, wird eine Übersicht über die verfügbaren Invoker und deren Zustand gehalten (Schritt 5)[Tho16]. Der Controller wählt nun einen der verfügbaren Invoker, um die Action auszuführen (Schritt 6). Controller und Invoker kommunizieren dabei ausschließlich über Kafka[20]. OpenWhisk nutzt für die Ausführung der Actions Docker Container [Apa18d] [Rab18]. Dies ermöglicht die isolierte und kontrollierte Ausführung einer Action. Für jede aufgerufene Action wird ein Docker Container erstellt, die Action inklusive der mitgegebenen Parameter eingefügt, und diese schlussendlich ausgeführt (Schritt 7) [Apa18d] [Rab18]. Nach Beendigung der Ausführung wird der Docker Container wieder zerstört [Rab18]. Die Ergebnisse der Ausführung der Action werden in einer Datenbank gespeichert (Schritt 8).

[19] https://www.consul.io
[20] https://kafka.apache.org/documentation/

Da die Initialisierung eines neuen Containers zeitaufwändig ist, werden Container in OpenWhisk wiederverwendet [Tho17]. Dies verkürzt zudem die Latenzzeit der Ausführung einer Serverless Funktion. Wird eine Action beispielsweise doppelt ausgeführt und eine der beiden Ausführungen ist bereits beendet, wird der selbe Container der ersten Action nochmal verwendet [Tho17]. Eine weitere, in OpenWhisk verwendete Methode zur Verbesserung der Performanz ist das sogenannte Vorwärmen von Containern. Hierfür werden anhand von erwartetem Workload vorausschauend Container erstellt. Angenommen, der Großteil der in OpenWhisk ausgeführten Actions würden eine Node.js Laufzeitumgebung benötigen, so würden stets bereits gestartete Container für die Ausführung dieser Actions bereitgehalten werden [Tho17].

Da Serverless Computing den Ansatz einer event-basierten Architektur verfolgt, gibt es auch in OpenWhisk verschiedene Arten von Events, beziehungsweise in OpenWhisk Trigger genannt, welche eine Action auslösen können. Der Katalog von OpenWhisk umfasst aktuell sechs Trigger: (1) *Alarm*, (2) *Cloudant*, (3) *GitHub*, (4) *Messaging*, (5) *Push Notifications* und (6) *Custom* Trigger. Das Alarm Package bietet die Option, Actions zeitplanmäßig, wie beispielsweise periodisch oder einmalig an einem spezifizierten Zeitpunkt, auszuführen. Dies ist unter anderem nützlich für wiederkehrende Aufgaben, wie beispielsweise stündlich durchzuführende Backups [Apa18e]. Mit dem Cloudant Package lassen sich Actions als Reaktion auf Änderungen in einer Cloudant[21] Datenbank ausführen [Apa17b]. Cloudant ist eine Datenbank-as-a-Service (DBaaS) von IBM[22]. Actions können, mittels des GitHub Packages, auch als Reaktion auf Änderungen in einem GitHub-Repository ausgeführt werden [Apa18f]. Das Messaging Package erlaubt es, Actions als Reaktion auf Nachrichten einer Kafka oder MessageHub[23] Instanz auszuführen [Apa18c]. Ein weiterer Trigger des OpenWhisk Katalogs ist der Push Notifications Trigger, welcher ermöglicht, Actions als Reaktion auf Push-Benachrichtigungen des IBM Push Notifications Service[24] auszuführen [Apa18g]. OpenWhisk erlaubt auch sogenannte Custom Trigger, bei denen der Trigger ein HTTP Endpunkt darstellt, der ausgelöst wird, wenn eine POST Anfrage an diesen Endpunkt gestellt wird. Wie bereits zuvor beschrieben, werden Trigger in OpenWhisk mittels Rules mit Actions verknüpft.

2.1.4 Serverless Framework

Das Serverless Framework ist ein MIT Open-Source[25] Projekt. Es stellt ein CLI zur Verfügung, welches den Nutzer dabei unterstützen soll, Serverless Funktionen auf verschiedenen Serverless Plattformen zu deployen. Unter anderem unterstützt das Serverless Framework AWS Lambda, Microsoft Azure Functions[26], OpenWhisk, Google Cloud Functions, Kubeless[27], Spotinst[28] und Webtask[29].

[21] https://www.ibm.com/cloud/cloudant
[22] https://www.ibm.com/us-en/
[23] https://console.bluemix.net/catalog/services/message-hub
[24] https://www.ibm.com/cloud/push-notifications
[25] https://opensource.org/licenses/MIT
[26] https://azure.microsoft.com/en-us/services/functions/
[27] http://kubeless.io
[28] https://spotinst.com
[29] https://webtask.io

Listing 2.3 Serverless Framework Service für Apache OpenWhisk nach [Ser18]

```
# serverless.yml

service: myService

frameworkVersion: ">=1.0.0 <2.0.0"

provider:
  name: openwhisk
  runtime: nodejs:default
  memory: 256 # Overwrite default memory size. Default is 512.
  timeout: 10 # The default is 60
  overwrite: true # Can we overwrite deployed functions? default is true
  namespace: 'custom' # use custom namespace, defaults to '_'
  ignore_certs: true # ignore ssl verification issues - used for local deploys

functions:
  usersCreate: # A Function
    handler: users.create # The file and module for this specific function.
    memory: 256 # memory size for this specific function.
    timeout: 10 # Timeout for this specific function. Overrides the default set above.
    runtime: nodejs:6
    overwrite: false # Can we overwrite deployed function?
    namespace: 'custom' # use custom namespace, defaults to '_'
      ...
```

Das Serverless Framework lässt sich via NPM installieren. Mit Hilfe des CLIs können Funktionen und Events auf den unterstützten Serverless Plattformen deployed werden. Um die Funktionen und zugehörigen Events zu koordinieren, nutzt das Serverless Framework sogenannte *Services*. Services sind vergleichbar mit einer Projektdatei und werden als *serverless.yml* Datei gespeichert. Diese Datei beinhaltet die Definitionen aller Funktionen, Events und Ressourcen einer Serverless Anwendung.

Als Zentrale fungierend ermöglicht das Serverless Framework das Deployment von Serverless Funktionen auf den unterstützten Serverless Plattformen. Jedoch erlaubt das Serverless Framework nicht, Funktionen verschiedener Serverless Plattformen innerhalb eines Services zu definieren. Es ist demnach nicht möglich, ein Service aufzusetzen, bei dem beispielsweise eine Funktion auf AWS Lambda deployed ist, und eine andere Funktion auf Apache OpenWhisk. Um einen Service mittels des Serverless Frameworks zu definieren, muss zunächst also eine einzige Serverless Plattform ausgewählt werden. Ein beispielhafter, jedoch nicht vollständiger Service des Serverless Frameworks für Apache OpenWhisk ist in Listing 2.3 dargestellt. Dabei wird unter *provider* unter anderem die Zielplattform und die Laufzeitumgebung der in des Services enthaltenen Serverless Funktionen definiert. Unter *functions* werden die in dem Service enthaltenen Serverless Funktionen aufgelistet. Da man sich für die Nutzung des Serverless Frameworks pro Service auf eine Serverless Plattform festlegen muss, löst dies nicht die Problematik des Vendor-Lock-Ins. Es ist also nicht möglich, Serverless Funktionen und Events plattformübergreifend zu deployen. Legt man sich nun auf eine Serverless Plattform fest, so ist man in deren Ökosystem gefangen. Das Serverless Framework bezeichnet sich selbst als provider-agnostisch, was zum einen stimmt, da ausgehend vom Serverless Framework Services für jeglichen unterstützten Cloud Provider

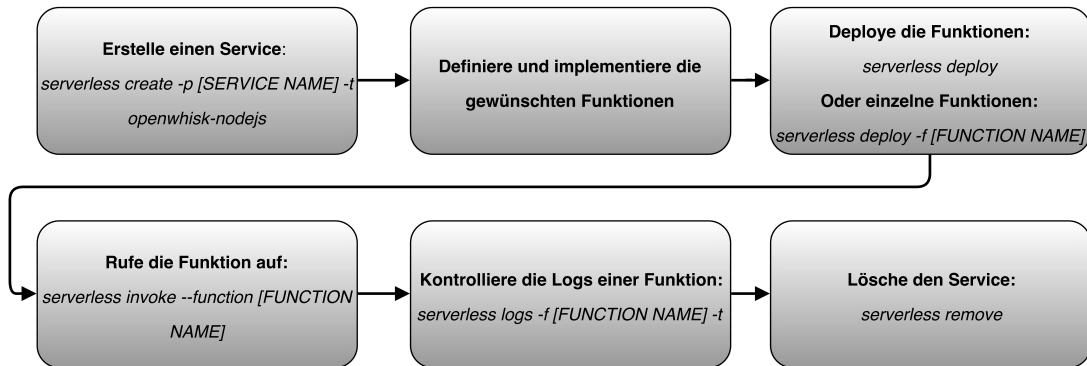

Abbildung 2.6: Beispielhafter Ablauf einer Interaktion mit dem Serverless Framework

definiert werden können, jedoch ist der Wechsel des Cloud Providers innerhalb eines bestehenden Services nicht ohne Weiteres möglich.

Jedoch erleichtert das Serverless Framework den Weg zum Deployment einer Funktion auf einer Serverless Plattform. Es erlaubt mittels simpler CLI-Befehle unter Angabe des Cloud Providers, als auch der Laufzeitumgebung der zu deployenden Serverless Funktionen, das Erstellen eines Services. Dieser Service beinhaltet bereits alles, was nötig ist, um eine Serverless Anwendung zu deployen. Da allerdings keine graphische Benutzeroberfläche zur Modellierung einer Anwendungsarchitektur existiert, ist das Erstellen komplexer Serverless Anwendungen und die Nachvollziehbarkeit der Beziehungen der Funktionen und Events gegebenenfalls kompliziert. Der beispielhafte Ablauf einer Interaktion mit dem Serverless Framework CLI ist Abbildung 2.6 zu entnehmen. Zunächst wird ein Service für OpenWhisk erstellt, anschließend werden die Serverless Funktionen den Anforderungen entsprechend implementiert. Nach der Implementierung aller gewünschten Funktionen werden diese auf der spezifizierten Plattform deployed. Das CLI bietet auch die Möglichkeit, die Funktionen aufzurufen und die Logs der Funktion für Debugging-Zwecke zu betrachten.

Um das Problem des Vendor-Lock-Ins und der Inkompatibilität von Serverless Funktionen und Events zwischen verschiedenen Serverless Plattformen zu adressieren, beherbergt das Serverless Framework das Subprojekt *Event Gateway*, welches als *Event Router* zwischen Serverless Funktionen verschiedener Serverless Plattformen dient. Das Event Gateway bildet Events providerunabhängig, beziehungsweise -übergreifend auf Funktionen ab [Mün18]. Jede Form von Daten sollen dabei als Event dargestellt werden, um dem event-basierten Programmiermodell des Serverless Computings gerecht zu werden. Daten werden dazu mittels HTTP Anfragen empfangen und als Event interpretiert [Mün18]. Aktuell werden mit AWS Lambda, Microsoft Azure, IBM OpenWhisk und Google Cloud Functions vier Serverless Plattformen unterstützt. Mit Hilfe des Event Gateways können jegliche Events auf jegliche Serverless Funktion jeglicher Serverless Plattform abgebildet werden. Unter anderem soll das Event Gateway unterstützend sein für das Kompositum großer Serverless Architekturen [Mün18].

Das Event Gateway basiert auf einem API-Gateway und dem Publish-Subscribe-Prinzip. Um die Funktionalität des Event Gateways nutzen zu können, registriert man eine Serverless Funktion mittels CLI bei dem Event Gateway. Anschließend wird die Serverless Funktion mittels des Event

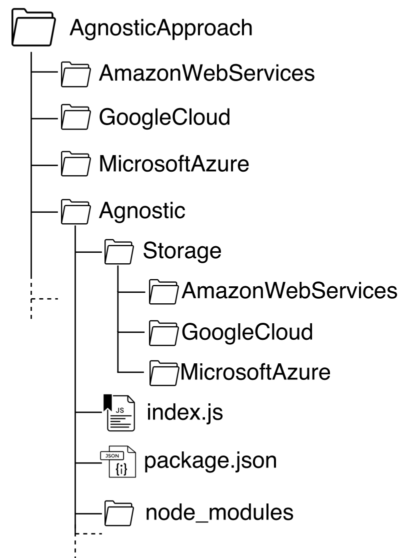

Abbildung 2.7: Projektstruktur einer beispielhaften, provider-unabhängigen Serverless Anwendung nach [Sti18]

Gateways als HTTP Endpunkt angeboten. Nun verhält sich das Event Gateway wie ein API Gateway und die Funktion kann anhand dieses API Gateways aufgerufen werden. Anschließend kann ein Event erstellt werden und dieses via CLI mit jeglicher, innerhalb des Event Gateways registrierten Funktion verknüpft werden.

Zum Zeitpunkt des Verfassens dieser Thesis befindet sich das Event Gateway jedoch noch in einem sehr frühen Stadium. Einen weiteren Ansatz, um Serverless Anwendungen provider-agnostisch zu beschreiben, präsentiert Maddie Stigler [Sti18] in ihrem Buch „Beginning Serverless Computing“ unter Zuhilfenahme des Serverless Frameworks. Am Beispiel der Softwarearchitektur eines der größten Versicherungsunternehmen der Welt untermauert Stigler den Bedarf nach einer provider-agnostischen Lösung für die Migration in die Cloud. Das Unternehmen, welches aktuell noch den größten Teil der IT-Infrastruktur vor Ort hat, möchte die gesamte Infrastruktur in die Cloud verlagern, fürchtet jedoch den Vendor-Lock-In bei einem Cloud Provider [Sti18].

Stigler schlägt daher vor, eine provider-unabhängige Anwendung, beziehungsweise Architektur, zu entwickeln [Sti18]. Da sich die verschiedenen Serverless Plattformen typischerweise nur in den angebundenen Services, sowie der Funktionssignatur unterscheiden, der zugrundeliegende Code der Serverless Funktionen jedoch identisch ist, eignet sich der Serverless Computing Ansatz laut Stigler bestens für die Erstellung einer weitestgehend provider-unabhängigen Cloud Anwendung [Sti18].

Stigler erstellt ein Konzept mittels des Serverless Frameworks, bei dem die provider-spezifische Logik seperat von der provider-agnostischen Logik gehalten wird, um eine Anwendung maximal unabhängig von Cloud Providern zu halten. Für den Proof-of-Concept dieses Ansatzes erstellt Stigler via Serverless Framework eine Hello-World Anwendung, deren Logik provider-agnostisch enkapsuliert wird und die, ohne großen Zeitaufwand für den Wechsel des Cloud Providers, je nach Belieben auf den Serverless Plattformen AWS Lambda, Microsoft Azure Functions oder Google

Listing 2.4 Code der provider-unabhängigen index.js Datei nach [Sti18]

```
// dependencies
var provider = 'aws';
var Provider = require('./storage/' + provider + '/provider');

exports.handler = function (event, context, callback) {
   console.info(event);
   Provider.printProvider("Hello World");
}
```

Listing 2.5 Code der Provider.js Datei in AWS Lambda nach [Sti18]

```
// dependencies

module.exports = {
   printProvider: function(message) {
      console.log('Message: ' + message + ' from AWS!');
   }
}
```

Cloud Functions deployed werden kann [Sti18]. Abbildung 2.7 zeigt die von Stigler gewählte Projektstruktur zur Durchführung des Proof-of-Concepts, bei dem jegliche provider-spezifische Logik in jeweils einem provider-spezifischen Ordner liegt. Jeder provider-spezifische Ordner hat eine eigene serverless.yml Datei. Der provider-unabhängige Code der Anwendung befindet sich in der Datei *index.js*, dargestellt in Listing 2.4. In der Variable *provider* wird der Cloud Provider, auf dem die Anwendung deployed werden soll, festgelegt. Jeder provider-spezifische Ordner wird nun als eigener Service im Serverless Framework aufgesetzt und erhält demach eine eigene, provider-spezifische serverless.yml Datei, sowie eine Provider.js Datei, welche die *printProvider* Methode provider-abhängig implementiert. Diese Methode gibt in diesem Beispiel lediglich den Namen des Cloud Providers zurück, auf welchem die Anwendung deployed ist. Die Implementierung der Methode als AWS Lambda Funktion ist in Listing 2.5 dargestellt. Möchte man nun den Cloud Provider ändern, muss die Variable in der index.js Datei angepasst werden [Sti18].

2.2 TOSCA

Die *Topology and Orchestration Specification for Cloud Applications* [OAS13a], kurz TOSCA, ist ein Standard der *Organization for the Advancement of Structured Information Standards* (OASIS) für die interoperable sowie portable Beschreibung von Cloud Anwendungen, deren Management und Orchestration [OAS13a] [BBKL14]. TOSCA liefert eine portable Sprache, sowohl für die Beschreibung der Struktur einer Cloud Anwendung, ihrer Komponenten und deren Beziehung zueinander, als auch deren verfügbaren Management-Funktionen [BBKL14]. Die Motivation hinter TOSCA ist sowohl die Beseitigung des Vendor-Lock-Ins, als auch automatisiertes Management von Cloud Anwendungen. Aktuell befindet sich TOSCA in Version 1.0 [OAS13a]. Die Spezifikation in Version 1.0 erfolgte zunächst in der Extensible Markup Language[30] (XML) [OAS13a] . 2015 veröffentlichte OASIS das TOSCA Simple Profile Version 1.0 in der Yet Another Markup Language

[30] https://www.w3.org/XML/

[31] (YAML) aufgrund der im Vergleich zu XML zugänglicheren Syntax. Ende 2017 veröffentlichte OASIS die Version 1.2 des TOSCA Simple Profile in YAML [OAS17].

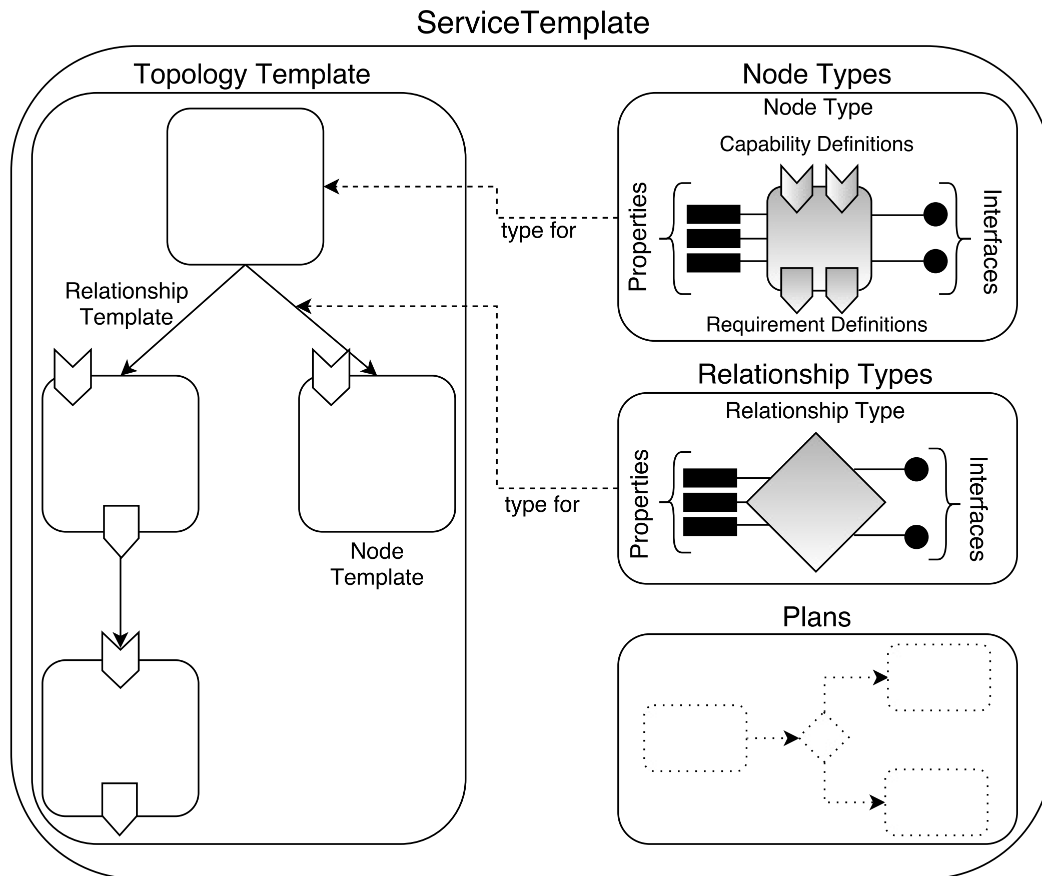

Abbildung 2.8: Aufbau eines TOSCA Service Templates nach [OAS13a]

Für diese Thesis wird jedoch im weiteren Verlauf die 2013 veröffentlichte Spezifikation in XML genutzt.

Um automatisiertes Management, sowie die Portabilität von Cloud Anwendungen zu ermöglichen, setzt TOSCA auf zwei Hauptkonzepte: (1) Anwendungstopologien, welche in TOSCA *Topology Templates* genannt werden und (2) Management-Pläne [BBKL14] [OAS13a]. TOSCA vereint diese beiden Hauptkonzepte in einem sogenannten *Service Template*. Im Folgenden wird zunächst ein Service Template als Gesamtheit beschrieben. Anschließend werden die einzelnen Bestandteile eines Service Templates erläutert.

Ein Cloud Service, beziehungsweise eine Cloud Anwendung, wird in TOSCA via eines Service Templates repräsentiert. Anhand Abbildung 2.8, welches den Aufbau eines Service Templates darstellt, ist erkennbar, dass das Service Template die beiden Konzepte des Topology Templates und der Management-Pläne vereint. Ein Topology Template repräsentiert dabei die Struktur einer

[31] http://yaml.org

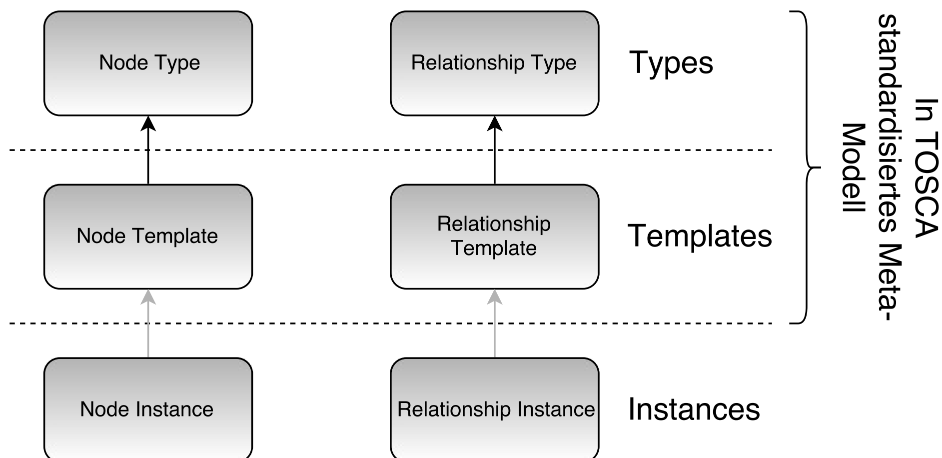

Abbildung 2.9: Metamodell des TOSCA Typsystems nach [BBKL14]

Cloud Anwendung und besteht aus Node und Relationship Templates. Wie die Abbildung 2.8 zeigt, dienen Node und Relationship Types als Typen für die Node und Relationship Templates. Management-Pläne werden, um die Portabilität dieser zu gewährleisten, in standardisierten Workflow-Sprachen erstellt. Alle notwendigen Elemente zur Definition eines Service Templates sind in einem *TOSCA Definitions* Dokument enthalten [OAS13a]. Für die Instanziierung eines Service Templates wird eine TOSCA-konforme Laufzeitumgebung benötigt. Eine TOSCA-konforme Laufzeitumgebung interpretiert Service Templates und bietet die Möglichkeit, Service Templates zu provisionieren, verwalten oder deprovisionieren [BBLS12].

Topology Templates werden in TOSCA als gerichteter Graph dargestellt, wobei die Komponenten der Cloud Anwendung die Knoten des Graphs sind. Die Beziehungen zwischen den jeweiligen Komponenten werden in dem Graph mittels gerichteter Kanten dargestellt. Abbildung 2.10 zeigt ein beispielhaftes Topology Template in TOSCA. Die Knoten eines Topology Templates werden als *Node Template* bezeichnet. Dies entspricht in dem beispielhaften Topology Template den Node Templates *PHP WebApp*, *PHP Container*, *Apache Web Server*, *Ubuntu* und *OpenStack*. Die Node Templates müssen eines *Node Types* abstammen. Ein Node Type ist eine wiederverwendbare Entität, welche den Typ eines oder mehrerer Node Templates definiert. Die Node Types der Node Templates des beispielhaften Topology Templates sind *Apache PHP-5 Module, Apache-2.4, Ubuntu-14.04-VM* und *OpenStack-Liberty12*. Node Types können Eigenschaften mittels einer *Properties Definition* definieren, welche die Node Templates erben und mit konkreten Werten belegen können, beziehungsweise müssen, falls sie nicht optional sind. In dem abgebildeten, beispielhaften Topology Template hat die Properties Definition des Ubuntu-14.04-VM Node Types unter anderem die Elemente *RAM*, *IP* und *SSHCredentials*. Node Templates können außerdem *Deployment Artifacts* haben. Diese sind ein Anhang jeglicher Dateien, die zur Provisionierung eines Node Templates in TOSCA benötigt werden. In dem beispielhaften Topology Template aus Abbildung 2.10 beinhaltet das Node Template PHP WebApp das Deployment Artifact *WebApp.php*, welches die PHP-Datei zur Provisionierung der PHP WebApp ist. Node Types können außerdem Management-Operationen anbieten, welche innerhalb *Interfaces* definiert werden. Die Beziehung zwischen den Komponenten eines Topology Templates referenziert TOSCA als *Relationship Template*. Das beispielhafte Topology Template aus Abbildung 2.10 beinhaltet vier Relationship Templates, welche die Node Templates verbinden. Analog zu Node Templates müssen Relationship

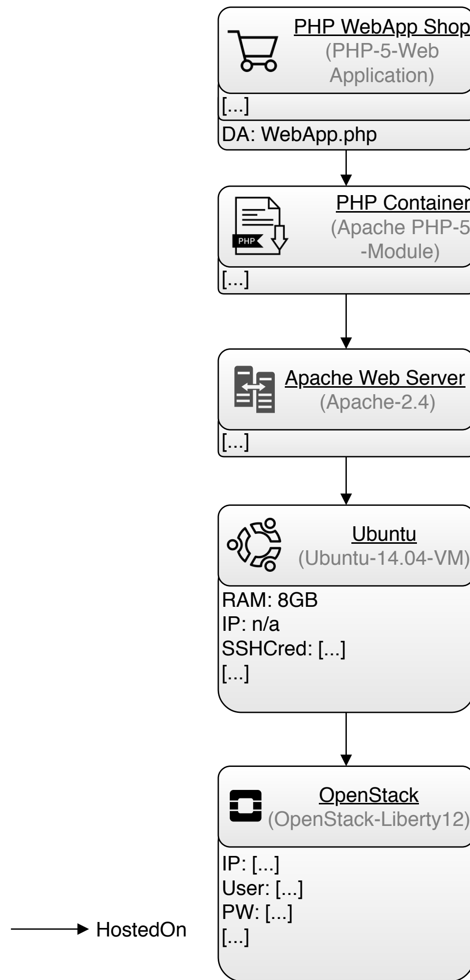

Abbildung 2.10: Beispielhaftes TOSCA Topology Template einer simplen Anwendung

Templates eines *Relationship Types* abstammen. Die Relationship Templates des Beispiel Topology Templates sind alle des *HostedOn* Relationship Types. Das Metamodell des TOSCA Typsystems in Abbildung 2.9 stellt die beschriebenen Beziehungen zwischen Tosca Types und Templates visuell dar. Bei der Instanziierung eines Service Templates werden die Templates zu Instances. Relationship Types sind, wie Node Types, wiederverwendbare Entitäten [OAS13a]. Da Topology Templates gerichtete Graphen sind, müssen die Relationship Types ein Quell- und Zielelement spezifizieren, um die Richtung der Beziehung darstellen zu können.

Die zuvor beschriebenen Topology Templates werden in TOSCA mittels Management-Plänen verwaltet. Management-Pläne sind in TOSCA als *Process Models* definiert [OAS13a]. Die Spezifikation dieser Management-Pläne basiert auf existierenden Workflow-Sprachen wie die Business Process Model and Notation[32] (BPMN) oder der Business Process Execution Language[33] (BPEL) [OAS13a].

[32]http://www.bpmn.org/
[33]https://www.eclipse.org/bpel/

Jedoch kann jede existierende Sprache zur Definition von Process Models verwendet werden [OAS13a]. Management-Pläne in TOSCA sind Sequenzen von Management-Operationen von Node Types. Pläne in TOSCA beschreiben, wie Instanzen einer Cloud Anwendung automatisiert erstellt, verwaltet oder beendet werden [OAS13a]. Demnach beschreiben Management-Pläne bestimmte Zustände des Lebenszyklus einer Cloud Anwendung. TOSCA unterscheidet daher zwischen drei verschiedenen Plänen für die Repräsentation der verschiedenen Zustände einer Cloud Anwendung: (1) Build-, (2) Verwaltungs- und (3) Termination-Pläne [OAS13a] [BBLS12]. Build-Pläne sind Workflows, welche die Provisionierung einer Cloud Anwendung beschreiben, wohingegen Termination-Pläne die Deprovisionierung dieser darlegen. Verwaltungs-Pläne stellen Workflows dar, die zur Verwaltung einer bereits provisionierten Cloud Anwendung genutzt werden. Durch die Nutzung von standardisierten Workflow-Sprachen stellt TOSCA die Portabilität sowie Interoperabilität dieser sicher. Auch die Implementierung der von Node Types angebotenen Management-Operationen ist durch die Verwendung von standardisierten Technologien wie Web Services[34] oder bekannten Skriptsprachen sichergestellt [BBLS12].

Um die beschriebenen Management-Aspekte einer Cloud Anwendung via TOSCA automatisieren zu können, werden Artefakte benötigt. Artefakte in TOSCA stellen Inhalte dar, die benötigt werden, um die Provisionierung einer in TOSCA definierten Cloud Anwendung zu realisieren [OAS13a]. Dabei wird zwischen *Implementation Artifacts* und *Deployment Artifacts* unterschieden. Implementation Artifacts beinhalten die ausführbaren Dateien der Implementierung der Management-Operationen eines Node Types und sind an Node Types gebunden [BBKL14]. Die Implementierung erfolgt via REST, WSDL [35] oder ausführbarer Skripte. Analog zu den Management-Plänen werden auch bei den Implementation Artifacts etablierte Standards genutzt, um die Portabilität und Interoperabilität dieser sicherzustellen. Die für die Provisionierung eines in TOSCA definierten Cloud Service benötigten Dateien werden *Deployment Artifacts* genannt [OAS13a] [BBLS12]. Deployment Artifacts können jegliche Dateien sein, die zur Provisionierung eines Cloud Services benötigt werden, wie beispielsweise die ausführbaren Web Application Resource[36] (WAR) Dateien eines Web Shops. Sowohl Deployment Artifacts, als auch Implementation Artifacts müssen eines *Artifact Types* abstammen [OAS13a]. Ein beispielhafter Artifact Type für ein Implementation Artifact ist das WAR Format, welches als Typ für Implementation Artifacts in Form eines Java Web Services[37] dient. Ein Artifact Type eines Deployment Artifacts ist beispielsweise ein *DockerImageArtifact*, welches den Typ für Deployment Artifacts in Form eines Docker Images darstellt.

Um die beschriebenen TOSCA Elemente zu bündeln, spezifiziert TOSCA das Dateiformat *Cloud Service Archive*[38] (CSAR) [OAS13a]. Ein CSAR verpackt jegliche benötigte Informationen, Definitionen, Typen und Dateien eines in TOSCA definierten Cloud Services. Eine CSAR-Datei ist ein ZIP-Archiv[39] mit der Dateiendung *.csar*. In einem CSAR befinden sich das Service Template, sowie alle Definitionen der darin genutzten TOSCA Typen wie beispielsweise Node Types, Relationship Types sowie Artifact Types. Des Weiteren beinhaltet ein CSAR die Management-Pläne des Cloud

[34] https://www.w3.org/2002/ws/
[35] https://www.w3.org/TR/2001/NOTE-wsdl-20010315
[36] https://docs.oracle.com/cd/E19199-01/816-6774-10/a_war.html
[37] https://docs.oracle.com/javaee/6/tutorial/doc/giqsx.html
[38] http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html#_Toc356403711
[39] https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

Services, sowie alle benötigten Implementation und Deployment Artifacts zur Ausführung dieser. Dadurch ist ein CSAR eine in sich geschlossene Datei [OAS13a] [BBKL14].

Um nun das automatisierte Management einer TOSCA Anwendung zu ermöglichen, muss ein CSAR in einer TOSCA-konformen Laufzeitumgebung, welche im Folgenden *TOSCA Runtime* genannt wird, instanziiert werden [BBKL14] [OAS13b]. Eine TOSCA Runtime kann CSARs interpretieren und anhand der in ihr enthaltenen Definitionen, Artefakte und Pläne TOSCA Anwendungen automatisiert verwalten [BBKL14]. Für das Verarbeiten eines CSARs in einer TOSCA Runtime wird zwischen zwei Prozessierungsarten unterschieden: (1) imperatives oder (2) deklaratives Prozessieren [BBH+13] [BBK+14]. Imperatives Prozessieren setzt voraus, dass die gesamte Logik für das Management einer TOSCA Anwendung im CSAR enthalten ist [BBK+14]. Demnach muss ein CSAR für das imperative Prozessieren vollautomatisiert ausführbare Management-Pläne beinhalten [BBK+14]. Im Gegensatz dazu steht das deklarative Prozessieren eines CSARs: die TOSCA Runtime interpretiert das Topology Templates der Anwendung und leitet sich anhand dieser Interpretation die benötigte Logik für das Management her [BBK+14]. Für das deklarative Prozessieren ist eine sehr genaue Definition der Node und Relationship Types notwendig [BBK+14].

3 Ansatz

Dieses Kapitel beschreibt ein Konzept, um anhand TOSCA die plattformübergreifende Kompatiblität von Serverless Funktionen und Events zu ermöglichen. Abbildung 3.1 illustriert die Vorgehensweise des Ansatzes. In Abschnitt 3.2 identifizieren wir für die Erstellung eines Metamodells der Struktur einer Serverless Anwendung zunächst die verschiedenen Komponten dieser. Anschließend abstrahieren wir in Abschnitt 3.3 die identifizierten Komponenten für die Erstellung eines plattform-agnostischen Metamodells für die Abbildung auf TOSCA Elemente. Darauffolgend wird die provider-spezifische Implementierung der Management-Operationen in TOSCA beschrieben, um plattformübergreifendes Deployment von Serverless Funktionen und Events, sowie automatisiertes Management einer Serverless Anwendung via TOSCA zu ermöglichen. Abschnitt 3.4 beschreibt abschließend ein Programmiermodell zur Erstellung und dem Betreiben von Severless Anwendungen in TOSCA.

3.1 Motivation

Wie bereits in der Einleitung und den Grundlagen beschrieben, setzen verschiedene Serverless Plattformen das Serverless Computing Paradigma proprietär um, wodurch sich Unternehmen für das Nutzen des Serverless Computing Ansatzes auf eine Serverless Plattform festlegen müssen. Dies resultiert jedoch in einem Vendor-Lock-In, welchen es zu vermeiden gilt. Daher wollen wir Serverless Funktionen und Events plattformunabhängig und abstrahiert von Serverless Plattformen beschreiben. Dadurch sollen Serverless Funktionen und Events kompatibel zwischen verschiedenen Serverless Plattformen werden. Aktuell ist es, aufgrund der in Abschnitt 2.1.2 skizzierten Unterschiede, wie beispielsweise der ungleichen Funktionssignatur oder der Übergabe des

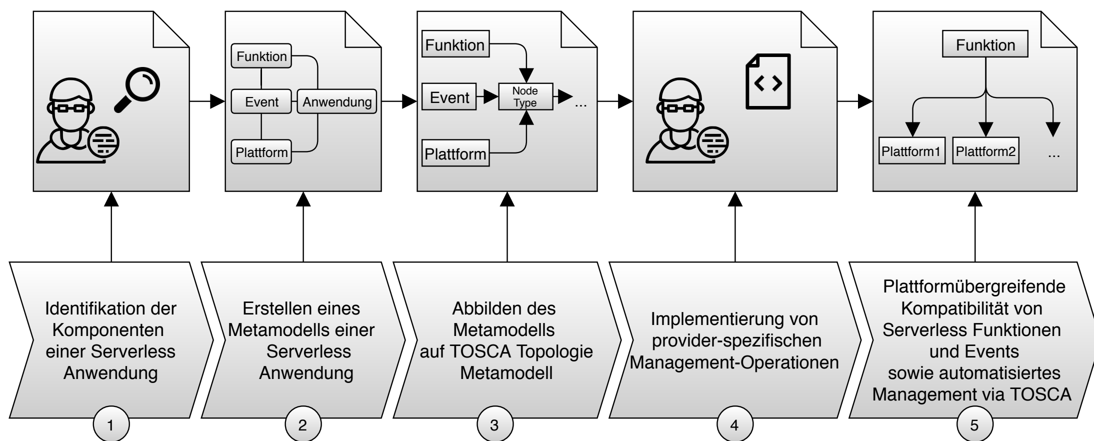

Abbildung 3.1: Übersicht über die Vorgehensweise des Ansatzes

Rückgabewertes, nicht möglich, Serverless Funktionen und Events plattformunabhängig zu nutzen. Im Folgenden wird ein Konzept beschrieben, um Serverless Funktionen und Events abstrahiert und plattformunabhängig in TOSCA zu repräsentieren. Dadurch wird eine plattformübergreifende Kompatibilität von Serverless Funktionen und Events erwirkt. Zunächst identifizieren wir dazu die verschiedenen Komponenten einer Serverless Anwendung in Abschnitt 3.2.

3.2 Identifikation der Komponenten einer Serverless Anwendung

Um eine Serverless Anwendung auf TOSCA abbilden zu können, müssen wir zunächst die einzelnen Komponenten einer Serverless Anwendung identifizieren. Abbildung 3.2 beinhaltet die Darstellung eines Metamodells, welches die Anatomie einer Serverless Anwendung darstellt. Wir sehen, dass eine Serverless Anwendung aus mindestens einer Serverless Funktion und einer Serverless Plattform besteht. Außerdem können Serverless Funktionen mit Events verbunden sein, welche diese Serverless Funktion auslösen. Serverless Funktionen und Events werden auf einer Serverless Plattform deployed. Nachfolgend werden jeweils die identifizierten Komponenten einer Serverless Anwendung beschrieben, beginnend mit der Komponente der Serverless Funktion.

Eine Serverless Funktion ist, wie bereits in Kapitel 2.1.3 beschrieben, eine kurzlebige, zustandslose Funktion, welche auf einer Serverless Plattform gehosted ist. Je nach Serverless Plattform unterscheiden sich die angebotenen Laufzeitumgebungen für die Serverless Funktionen. Aus Skalierungsgründen und der gewünschten Feingranularität einer Serverless Anwendung, soll eine Serverless Funktion möglichst nur eine Funktionalität implementieren. Wie in der Abbildung 3.2 erkennbar ist, hat eine Serverless Funktion grundlegend die Attribute des Funktionsnamen, der Laufzeitumgebung der Funktion und dem Funktionscode. Außerdem illustiert die Abbildung die Verbindung zwischen einer Serverless Funktion und einem Event innerhalb einer Serverless Anwendung.

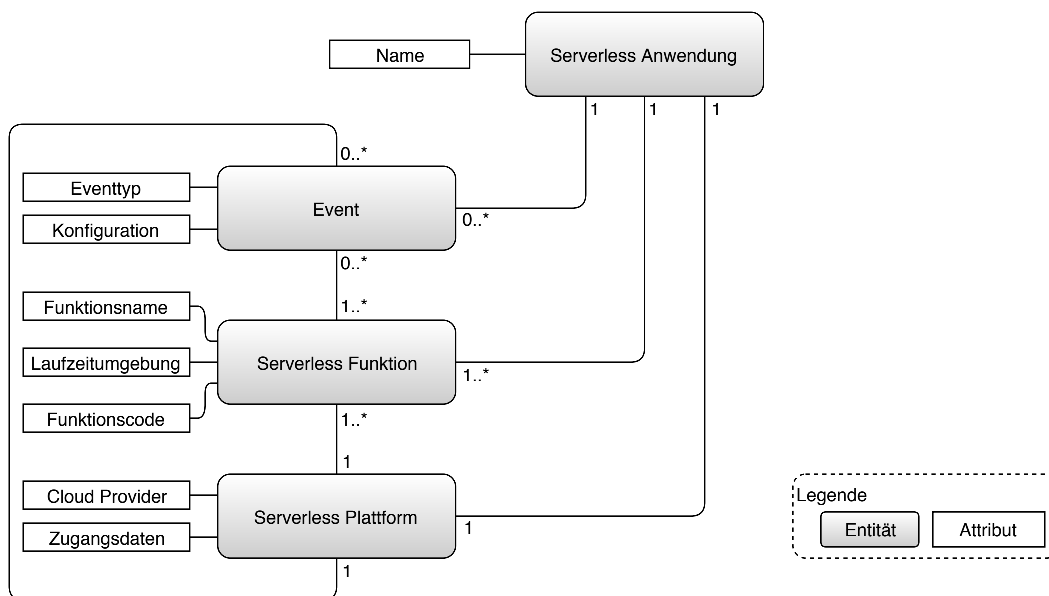

Abbildung 3.2: Metamodell der Anatomie einer Serverless Anwendung

Der Ansatz des Serverless Computing verfolgt das Ideal einer event-basierten Anwendung. Serverless Funktionen sollen in einer Serverless Anwendung von Events ausgelöst werden. Events werden, wie eine Serverless Funktion, auf einer Serverless Plattform deployed und mit mindestens einer Serverless Funktion verbunden. Dabei unterscheidet sich ein Event einer Serverless Anwendung durch das Attribut Eventtyp und die Konfiguration des Events. Die verschiedenen Events kategorisieren wir für den weiteren Verlauf der Arbeit wie folgt: (1) HTTP-Event, (2) Timer-Event und (3) Publish-Subscribe-Event, wobei sich der Publish-Subscribe-Event in weitere Sub-Events aufteilt. Die verschiedenen Events, sowie die Sub-Kategorien des Publish-Subscribe-Events sind Abbildung 3.4 zu entnehmen. Die Kategorisierung der Events basiert dabei auf den verschiedenen Arten von Anwendungsfällen, in welchen Serverless Funktionen ausgeführt werden. Ein HTTP-Event stellt dabei einen HTTP Endpunkt dar, der eine Serverless Funktion auslöst, wenn eine HTTP Anfrage an diesen Endpunkt gestellt wird. Dies ist beispielsweise ein Anwendungsfall für das Nutzen von Serverless Funktionen für das Implementieren von leichtgewichtigen Backend-Funktionen. Mittels eines Timer-Events können Serverless Funktionen auf Basis eines festgelegten Zeitplans ausgeführt werden, beispielsweise für das Erledigen wiederkehrender Aufgaben, wie das Erstellen eines Backups. Die Kategorie des Publish-Subscribe-Events umfasst jegliche Arten von Events, welche auf einem Publish-Subscribe-Prinzip basieren. Dies ist beispielsweise nützlich für Cloud Anwendungen im Bereich Internet der Dinge: Serverless Funktionen können sich mittels eines Message Brokers auf Sensoren abonnieren und bei der Veröffentlichung neuer Sensordaten ausgelöst werden. Unter die Kategorie des Publish-Subscribe-Events fällt außerdem der Datenbank-Event, welcher die Ausführung von Serverless Funktionen als Reaktion auf die Änderung in einer Datenbank ermöglicht. Des Weiteren können Serverless Funktionen mittels eines Objektspeicher-Events auf das Hinzufügen oder Entfernen von Dateien in einem Objektspeicher ausgelöst werden. AWS Lambda bietet beispielsweise den Objektspeicher AWS S3[1] als Event an, um Serverless Funktionen als Reaktion auf die Änderung innerhalb eines Objektspeichers auszuführen.

Sowohl Serverless Funktionen als auch die damit verbundenen Events werden auf einer Serverless Plattform deployed. Eine Serverless Plattform bietet eine Laufzeitumgebung zur Ausführung von Serverless Funktionen und hosted jene Funktionen. Außerdem bietet eine Serverless Plattform üblicherweise einen Katalog an Events, welche eine Serverless Funktion auslösen können. Die angebotenen Programmiersprachen für die Implementierung der Serverless Funktionen variieren je nach Provider der Serverless Plattform. Als generelle Attribute einer Serverless Plattform legen wir den Cloud Provider der Serverless Plattform, sowie die Zugangsdaten für die jeweilige Plattform fest.

3.3 Serverless Anwendungen in TOSCA

Im Folgenden bilden wir die Entitäten des Metamodells der Anatomie einer Serverless Anwendung aus Abbildung 3.2 auf Elemente der TOSCA Spezifikation ab. Zunächst möchten wir jedoch eine Serverless Anwendung als Gesamtheit auf TOSCA abbilden, bevor wir uns anschließend den darin enthaltenen Komponenten und Management-Aspekten widmen.

[1] http://aws.amazon.com/s3

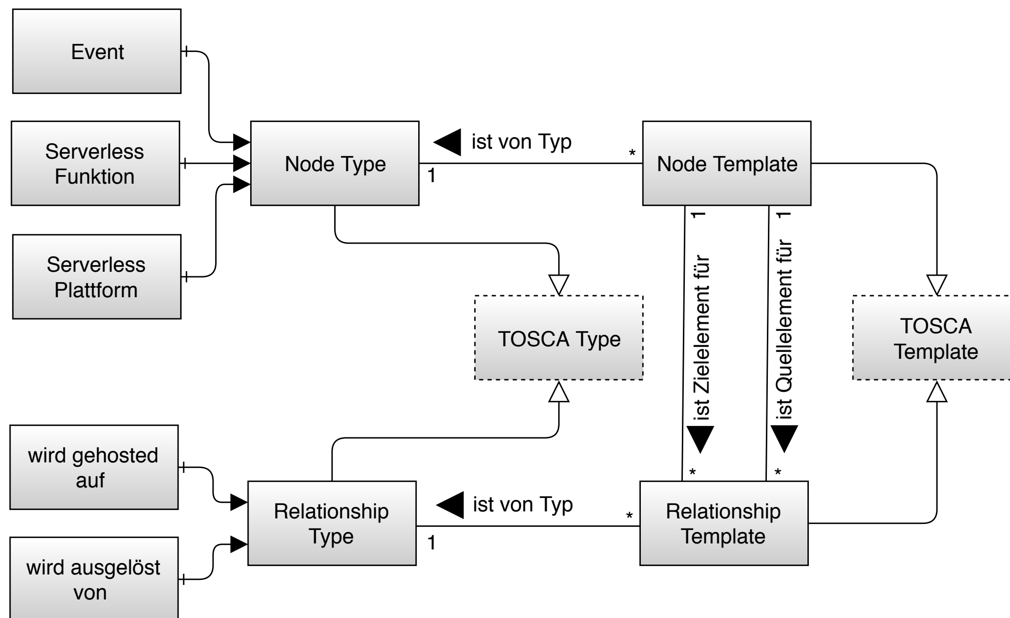

Abbildung 3.3: Abbildung der Serverless Komponenten auf das TOSCA Topologie Metamodell

Als Repräsentation einer Serverless Anwendung in TOSCA wählen wir das Service Template. Ein Service Template setzt sich, wie in Abschnitt 2.2 erläutert, aus der Struktur einer Anwendung in Form eines Topology Templates und Management-Plänen zur automatisierten Provisionierung, Verwaltung und Deprovisionierung des Services zusammen. Da Serverless Anwendungen event-basiert sind, lässt sich die Struktur solch einer Anwendung optimal als gerichteter Graph darstellen. Für das Betreiben einer Serverless Anwendung in einer TOSCA Laufzeitumgebung werden Management-Pläne benötigt, welche neben dem Topology Template in einem Service Template enthalten sind. Das Topology Template eines Service Templates setzt sich aus Node und Relationship Templates zusammen, welche nach dem TOSCA Typsystem jeweils von Node und Relationship Types abstammen. Im Folgenden widmen wir uns nun zunächst dem ersten Teil eines Service Templates, dem Topology Template. Anschließend folgt die Beschreibung der zu implementierenden Management-Operationen für das automatisierte Management von Serverless Anwendungen in TOSCA.

3.3.1 Struktur einer Serverless Anwendung in TOSCA

Die Struktur einer Serverless Anwendung in TOSCA stellen wir mithilfe eines Topology Templates dar. Dies dient auch der Erstellung einer Serverless Anwendung in TOSCA. Nun erstellen wir ein Metamodell, um die Entitäten des Metamodells der Anatomie einer Serverless Anwendung aus Abbildung 3.2 auf das TOSCA Topology Metamodell abzubilden. Dieses Metamodell ist Abbildung 3.3 zu entnehmen. Wie zu sehen ist, stellen wir Serverless Funktionen, Events und Serverless Plattformen als Node Types in TOSCA dar. Um die Kompatibilität von Serverless Funktionen und Events verschiedener Serverless Plattformen via TOSCA ermöglichen zu können, möchten wir eine maximal abstrahierte und plattformunabhängige Darstellung der verschiedenen Entitäten erreichen. Dazu betrachten wir im Folgenden die Serverless Plattformen OpenWhisk, AWS Lambda und Google Cloud Functions. Ausgehend von diesen Serverless Plattformen ermitteln wir

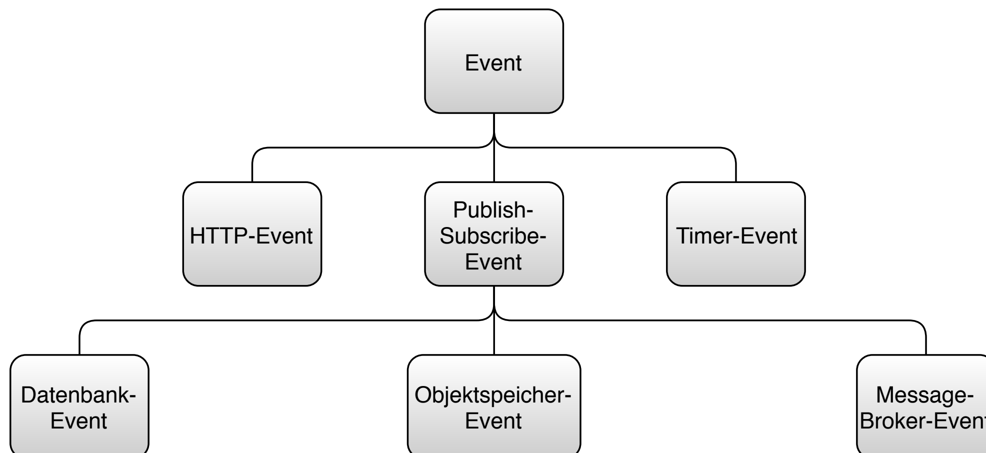

Abbildung 3.4: Übersicht über die Kategorisierung von Events in Serverless Anwendungen

Gemeinsamkeiten zwischen den Serverless Funktionen und Events dieser Serverless Plattformen, um eine abstrahierte Darstellung dieser in TOSCA zu ermöglichen. Wie bereits in Abschnitt 3.2 beschrieben, bilden wir drei Kategorien von Events: (1) HTTP-Event, (2) Timer-Event und (3) Publish-Subscribe-Event. Publish-Subscribe-Event ist dabei ein Oberbegriff für eine Vielzahl von Events. Abbildung 3.4 zeigt die verschiedenen Unterkategorien des Publish-Subscribe-Events. Dazu gehören der Datenbank-Event, Objektspeicher-Event und Publish-Subscribe-Messaging-Event. Der Datenbank-Event löst eine Serverless Funktion aus, wenn eine bestimmte Datenbank aktualisiert wird. Durch den Objektspeicher-Event werden Serverless Funktionen als Reaktion auf die Änderung eines Cloud Objektspeichers, wie beispielsweise das Hinzufügen einer neuen Datei, ausgeführt. Der Objektspeicher-Event existiert jedoch aktuell nicht für OpenWhisk. Der dritte Event des Publish-Subscribe-Event Oberbegriffs ist der Message-Broker-Event, wobei Serverless Funktionen sich auf einen Message Broker abonnieren können und bei neuen Nachrichten eines spezifizierten Topics ausgelöst werden. Da wir die Serverless Funktion, sowie die verschiedenen Events abstrahiert und provider-unabhängig repräsentieren wollen, müssen wir die provider-spezifischen Informationen der Komponenten in die jeweilige Serverless Plattform verschieben, welche wir ebenfalls via Node Types repräsentieren. Nun haben wir, wie in Abbildung 3.3 erkennbar, die Komponenten einer Serverless Anwendung auf folgende abstrahierte Node Types abgebildet: (1) *ServerlessFunktion*, (2) *Timer-Event*, (3) *HTTP-Event*, (4) *Publish-Subscribe-Event* sowie (5) mindestens einen provider-spezifischen *ServerlessPlattform* Node Type. Diese dienen, analog der Abbildung, als Typ für ein oder mehrere Node Templates.

Abbildung 3.5 zeigt die visuelle Darstellung eines Topology Templates einer vereinfachten Serverless Anwendung in TOSCA. Dies enthält zwei Node Templates des Node Types ServerlessFunktion und jeweils ein Node Template des Node Types Timer-Event und ein Node Template des Node Types Objektspeicher-Event. Die Beziehungen zwischen den Komponenten einer Serverless Anwendung werden mittels Relationship Types abgebildet. TOSCA definiert im TOSCA Simple Profile verschiedene Relationship Types, darunter unter anderem den *HostedOn* Relationship Type. Für die Abbildung der Beziehung zwischen einer Serverless Funktion und einer Serverless Plattform, sowie eines Events und einer Serverless Plattform, nutzen wir diesen HostedOn Relationship Type. Dieser repräsentiert, dass sowohl eine Serverless Funktion als auch ein Event auf einer Serverless Plattform gehosted werden. Wie in Abbildung 3.5 erkennbar ist, dienen sowohl

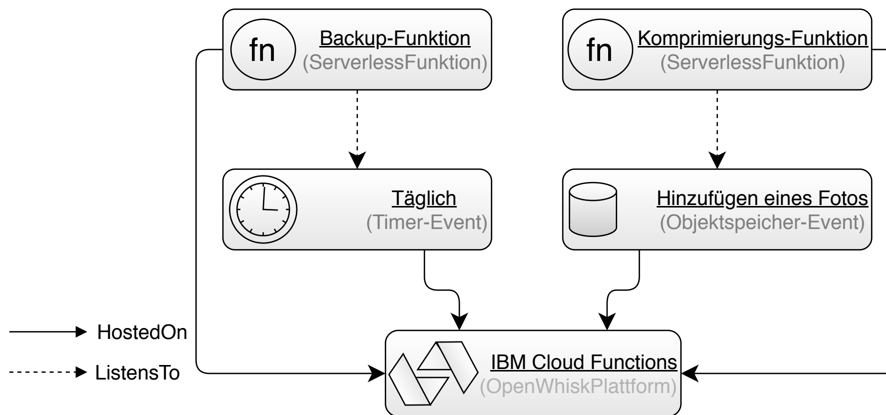

Abbildung 3.5: Vereinfachtes, beispielhaftes Topology Template einer Serverless Anwendung in TOSCA

Node Templates des Node Types ServerlessFunktion, als auch Node Templates der verschiedenen Event Node Types, als Quellelement für diesen Relationship Type. Das Zielelement dieses Relationship Types in einer Serverless Anwendung ist ein Node Templates eines provider-spezifischen ServerlessPlattform Node Types, was auch der Abbildung 3.3 des Metamodells zu entnehmen ist. Um die Beziehung zwischen Serverless Funktion und Events in einer event-basierten Serverless Anwendung in TOSCA darzustellen, erstellen wir einen neuen Relationship Type und nennen diesen *ListensTo*. Der ListensTo Relationship Type bildet das Ausführen einer Serverless Funktion durch das Auslösen eines Event in TOSCA ab. In dem Topology Template unserer beispielhaften Anwendung repräsentiert ein Relationship Templates dieses ListensTo Relationship Types, dass beispielsweise die Komprimierungs-Funktion auf das Hinzufügen eines Fotos wartet, um ausgelöst zu werden. Das Quellelement ist daher ein Node Template eines ServerlessFunktion Node Types, während das Zielelement ein Node Template eines Event Node Types ist.

Im weiteren Verlauf möchten wir das automatisierte Management einer Serverless Anwendung in TOSCA realisieren. Dadurch nutzen wir das Konzept der Management-Pläne in TOSCA, welche sich aus Management-Operationen einzelner Node Types zusammensetzen. Die Management-Operationen sollen das automatisierte, programmatische Deployment sowie Entfernen von Serverless Funktionen und Events auf Serverless Plattformen ermöglichen. Um dies zu ermöglichen, brauchen die Management-Operationen Eingabeparameter, wie beispielsweise den Namen der Serverless Funktion, die deployed werden soll. Diese benötigten Eingabeparameter nehmen wir aus den Properties Definitions der Node Templates unserer Node Types, welche nun im Folgenden beschrieben werden.

Serverless Funktionen in TOSCA

Da wir den ServerlessFunktion Node Type abstrahieren wollen, identifizieren wir zunächst, wie Serverless Funktionen auf gängigen Serverless Plattformen, wie Apache OpenWhisk, AWS Lambda oder Google Cloud Functions, programmatisch deployed und entfernt werden. Genauer gesagt betrachten wir die benötigten Parameter, um eine Serverless Funktion programmatisch mittels API Anfrage oder durch das Nutzen eines Client SDKs auf den verschiedenen Serverless Plattformen zu deployen, beziehungsweise zu entfernen. Anschließend bestimmen wir aus der Menge dieser

OpenWhisk	**AWS Lambda**	**Google Cloud Functions**
Funktionsname	*Funktionsname*	*Funktionsname*
Laufzeitumgebung	*Laufzeitumgebung*	*Einganspunkt*
Zeitlimit	*Zeitlimit*	*Zeitlimit*
Speicherlimit	*Speicherlimit*	*Speicherlimit*
API-Schlüssel	*Eingangspunkt*	Projektort
OpenWhisk Endpoint	AWS Zugangsdaten	Google Cloud Zugangsdaten
OpenWhisk Namespace	AmazonRessourcenNummer Rolle	

Tabelle 3.1: Vergleich der benötigten Parameter, um eine Serverless Funktion auf verschiedenen Serverless Plattformen programmatisch zu deployen

Parameter die Schnittmenge: die gemeinsamen Parameter, die alle von uns betrachteten Serverless Plattformen zum programmatischen Deployment einer Serverless Funktion, beziehungsweise eines Events, benötigen.

Diese Schnittmenge repräsentiert die Properties Definition des ServerlessFunktion Node Types. Durch Analyse der Dokumentation der verschiedenen Serverless Plattformen ergibt sich Tabelle 3.1, welche die verschiedenen Parameter der genannten Serverless Plattformen für das programmatische Deployment, beziehungsweise Entfernen einer Serverless Funktion auf der jeweiligen Serverless Plattform gegenüberstellt. Daraus ergibt sich also folgende Schnittmenge, beziehungsweise Properties Definition für den ServerlessFunktion Node Type: (1) Funktionsname, (2) Laufzeitumgebung, (3) Zeitlimit, (4) Speicherlimit und (5) Eingangspunkt. Diese TOSCA Properties Definition in XML kann Listing 3.1 entnommen werden. Außerdem ist die Darstellung repräsentativ für die im weiteren Verlauf dieses Kapitels erstellten Properties Definitions. Da OpenWhisk für das Deployment einer Serverless Funktion keinen Eingangspunkt benötigt, ist dieses Element der Properties Definition optional. Alle anderen provider-spezifischen Parameter, wie beispielsweise die Zugangsdaten zu einer bestimmten Serverless Plattform, schieben wir in die entsprechende Properties Definition des provider-spezifischen Serverless Plattform Node Types. Daraus ergibt sich ein maximal abstrahierter ServerlessFunktion Node Type.

Um eine Serverless Funktion zu deployen, benötigen wir nun noch die Datei, welche den Funktionscode der Serverless Funktion enthält. Um dies auf TOSCA abzubilden, nutzen wir das TOSCA Element Deployment Artifact, welches nach dem TOSCA Typsystem ebenfalls eines Artifact Types abstammen muss. Der TOSCA Primer definiert innerhalb der *Base Artifact Types* den *ArchiveArtifact* Type, welcher jegliche Deployment Artifacts repräsentiert, die eine Archivdatei darstellen [OAS13b]. Damit Serverless Funktionen externe Abhängigkeiten, wie beispielsweise Third-Party-Libraries nutzen können, werden sie mit den ausführbaren Dateien dieser Libraries und einer Konfigurationsdatei zu einem ZIP-Archiv gebündelt. Um ein einheitliches Deployment einer Serverless Funktion zu erlangen, legen wir fest, dass auch Serverless Funktionen ohne externe Abhängigkeiten als ZIP-Archiv deployed werden. Das Deployment Artifact wird bei der Erstellung des Topology Templates mit dem jeweils entsprechenden Node Template verknüpft.

Da ein Event immer in Kombination mit einer Serverless Funktion auf einer Serverless Plattform deployed wird, gehen wir im Folgenden für die Properties Definition der Event Node Types davon aus, dass wir uns durch die Elemente der Properties Definition des ServerlessFunktion Node

Listing 3.1 Properties Definition des ServerlessFunktion Node Types

```
<PropertiesDefinition elementname="properties"
    namespace="http:/example.com/nodetypes/propertiesdefinition">
        <properties>
           <key>Funktionsname</key>
           <type>xsd:string</type>
        </properties>
        <properties>
           <key>Laufzeitumgebung</key>
           <type>xsd:string</type>
        </properties>
        <properties>
           <key>Zeitlimit</key>
           <type>xsd:string</type>
        </properties>
        <properties>
           <key>Speicherlimit</key>
           <type>xsd:string</type>
        </properties>
            <properties>
           <key>Eingangspunkt</key>
           <type>xsd:string</type>
        </properties>
</PropertiesDefinition>
```

Types bereits erfolgreich für die Serverless Plattform authentifizieren können. Daher werden die für die Authentifizierung benötigten Informationen im Folgenden vernachlässigt. Ein Event ist außerdem immer mit einer Serverless Funktion verbunden, die als Reaktion auf die Auslösung des Events ausgeführt wird. Da wir die Struktur einer Serverless Anwendung in TOSCA mittels eines Topology Templates repräsentieren, gibt dies zur Laufzeit des Service Templates Aufschluss darüber, welcher Event welche Serverless Funktion auslösen soll. Daher wird der Parameter des Namens der Serverless Funktion, welche der Event auslösen soll, für die Properties Definition der Event Node Types an dieser Stelle ebenfalls vernachlässigt.

Timer-Event in TOSCA

Analog zu dem ServerlessFunktion Node Type identifizieren wir die benötigten Parameter, um den Timer-Event Node Type programmatisch auf den verschiedenen Serverless Plattformen zu deployen. Aus der Schnittmenge der Parameter ergibt sich die Properties Definition unseres abstrahierten Timer-Event Node Types. Aus der Analyse der Dokumentation der verschiedenen Serverless Plattformen geht folgende Schnittmenge der benötigten Parameter zum programmatischen Deployment, beziehungsweise Entfernen hervor: (1) Eventname und (2) Cron[2]. Ein Cron Ausdruck ist eine Zeichenkette, bestehend aus fünf bis sechs Werten, im Einzelnen geordnet nach: (1) Minuten (0-59), (2) Stunden (0-23), (3) Tag des Monats (0-31), (4) Monat (0-12), (5) Wochentag (0-7), (6) Jahr (optional). Cron ermöglicht den Ausdruck jeglicher Zeitpläne.

[2]http://crontab.org/cron.8.html

OpenWhisk	AWS Lambda	Google Cloud Functions
Eventname	*Eventname*	*Angabe, ob HTTP Endpunkt erstellt werden soll*
	API-ID	
	Resource-ID	
	HTTP-Methode	
	Authorisierungstyp	
	URI der Funktion	

Tabelle 3.2: Vergleich der für die verschiedenen Serverless Plattformen benötigten Parameter, um einen HTTP Event zu deployen

HTTP-Event in TOSCA

Um die Properties Definition eines abstrahierten HTTP-Event Node Types festzulegen, ermitteln wir zunächst die benötigten Parameter, um den HTTP-Event Node Type programmatisch auf den verschiedenen Serverless Plattformen zu deployen. Anhand dem Vergleich dieser Parameter ermitteln wir die Properties Definition für einen maximal abstrahierten, provider-unabhängigen HTTP-Event Node Type. An dieser Stelle setzten wir voraus, dass für das Deployment eines HTTP-Events auf AWS Lambda bereits ein konfiguriertes API-Gateway vorliegt. Der Vergleich der verschiedenen Parameter ist der Tabelle 3.2 zu entnehmen. Dabei fällt auf, dass sowohl OpenWhisk als auch Google Cloud Functions wenige Parameter voraussetzen, um einen HTTP Event zu deployen. Bei Google Cloud Functions muss lediglich während dem Deployment der Serverless Funktion angegeben werden, ob für diese Funktion ein HTTP Endpunkt erstellt werden soll. Da die Parameter *Resource-ID*, *HTTP-Methode* sowie *URI der Funktion* für das Deployment eines HTTP Events auf AWS Lambda je nach Serverless Funktion variieren, müssen diese Parameter dennoch in die Properties Definition des HTTP-Event Node Types, auch wenn sie nur für AWS Lambda zutreffend sind. Ansonsten wäre es nicht möglich, verschiedene HTTP Events auf AWS Lambda innerhalb eines Topology Templates in TOSCA zu deployen. Nun erhalten wir folgende Elemente der Properties Definition des HTTP-Event Node Types in TOSCA: (1) *Eventname*, (2) *ErstelleHTTPEndpunkt*, (3) *Resource-ID*, (4) *HTTP-Methode* und (5) *Funktions-URI*. Die provider-spezifischen Elemente der Properties Definition setzten wir dabei auf optional. Die übrigen, provider-spezifischen Parameter der Tabelle verschieben wir dabei erneut in die jeweilige Properties Definition der provider-spezifischen Serverless Plattform Node Types.

Publish-Subscribe-Events in TOSCA

Abbildung 3.4 zeigt die bereits beschriebene Aufteilung des Publish-Subscribe-Events in mehrere Unterkategorien. Wir erstellen, um diese Aufteilung auf TOSCA abzubilden, zunächst einen abstrakten Node Type *Publish-Subscribe-Event*. Abstrakte Node Types in TOSCA können nicht instanziiert werden. Um sie zu instanziieren, muss ein konkreter Node Type von dem abstrakten Node Type erben. Daher erstellen wir, analog zu Abbildung 3.4 die drei Node Types (1) *Datenbank-Event*, *Objektspeicher-Event* und (3) *Message-Broker-Event*.

Datenbank-Event in TOSCA Für den Event, welcher als Reaktion auf die Änderung in einer Datenbank ausgeführt wird, gehen wir davon aus, dass diese Datenbank bereits existiert. Um den Node Type maximal abstrahiert von den verschiedenen Serverless Plattformen zu beschreiben, identifizieren wir erneut die verschiedenen Parameter, die benötigt werden, um den Event, welcher als Reaktion auf die Änderung einer Datenbank ausgelöst wird, programmatisch deployen zu können. Wir betrachten dazu die Datenbanken-as-a-Service (DBaaS) von AWS, der Google Cloud und OpenWhisk, beziehungsweise IBM. Aus den gemeinsamen Parametern identifizieren wir die Properties Definition des Datenbank-Event Node Types. *DynamoDB*, eine DBaaS von AWS, veröffentlicht Aktualisierungen der Datenbank in einem Stream, welchen AWS Lambda abfragt und die darauf abonnierte Funktion bei neuen Datensätzen entsprechend ausführt. Um diesen Datenbank-Event programmatisch zu deployen, benötigt AWS Lambda zwei Parameter: (1) ARN der Eventquelle und (2) Startposition, ab welcher der Stream gelesen werden soll. *Cloudant*, eine DBaaS von IBM, benötigt folgende Parameter, um diesen Event deployen: (1) Name der Datenbank, (2) Instanz der Datenbank, (3) Zugangsdaten zu der Datenbank (Benutzername und Passwort) und (4) Host-URL der Datenbank. Google Cloud Functions führt die *Firebase Realtime Database* als Datenbank-Eventquelle für die Auslösung von Serverless Funktionen. Um diesen Event programmatisch zu erstellen, werden folgendende Parameter benötigt: (1) Typ der Änderung, auf die reagiert werden soll, (2) Name der Datenbank-Instanz und (3) Datenbank-Host. Der Typ der Änderung variiert zwischen der Auslösung bei jeder Änderung der Datenbank, nur beim Hinzufügen neuer Daten, nur beim Aktualisieren bestehender Daten oder nur bei dem Entfernen von bestehenden Daten. Aus den gemeinsamen Parametern der verschiedenen Serverless Plattformen bilden wir nun die Properties Definitions unseres abstrahierten Datenbank-Event Node Types wie folgt: (1) *Datenbankname*, (2) *Änderungstyp*, (3) *Datenbank-Nutzername*,(4) *Datenbank-Passwort*, (5) *Datenbank-Host*, (6) *Datenbank-Instanz* und (7) *Startposition*. Da die Elemente dieser Properties Definition teilweise immer noch provider-spezifisch sind, setzen wir alle Elemente, außer den Datenbank-Host, auf optional. Das Datenbank-Host Element ist demnach für Google Cloud Functions der Datenbank-Pfad, für OpenWhisk die Datenbank-Host-URL und für AWS Lambda die ARN der Eventquelle. Die übrigen, provider-spezifischen Parameter verschieben wir in die Properties Definition des jeweiligen provider-spezifischen Serverless Plattform Node Type.

Objektspeicher-Event in TOSCA Der Objektspeicher-Event ist, wie bereits erwähnt, nicht für OpenWhisk verfügbar. Für AWS Lambda dient AWS S3, der Cloud Objektspeicher von AWS, als Eventquelle. Um Dateien hochzuladen, muss jedoch zunächst ein sogenannter *Bucket* in AWS S3 erstellt werden. Verknüpft man nun den Objektspeicher-Event mit der Serverless Funktion in AWS Lambda, wird diese Serverless Funktion immer ausgeführt, wenn sich der Zustand des Buckets durch das Hinzufügen oder Entfernen von Dateien ändert. Um diesen Event programmatisch deployen zu können, benötigen wir für AWS Lambda folgende Parameter: (1) Name des Buckets und (2) Ereignistyp. Der Ereignistyp legt fest, ob der Event nur bei Hinzufügen, Entfernen oder allen Änderungen von Daten in dem entsprechenden Bucket ausgeführt werden soll. *Google Cloud Storage*, der Cloud Objektspeicher der Google Cloud, nennt die Behälter für die Speicherung von Dateien ebenfalls Bucket. Um den Objektspeicher-Event in Google Cloud Functions programmatisch zu deployen werden exakt dieselben Parameter wie für AWS Lambda benötigt: (1) Name des Buckets und (2) Ereignistyp. Daraus ergibt sich die Properties Definition unseres Objektspeicher-Event Node Types wie folgt: (1) *Bucketname* und (2) *Ereignistyp*.

OpenWhisk	AWS Lambda	Google Cloud Functions
Topic	Topic (ARN)	Topic
Name der MessageHub-Instanz		

Tabelle 3.3: Vergleich der für die verschiedenen Serverless Plattformen benötigten Parameter, um einen Publish-Subscribe-Messaging Event zu deployen

Message-Broker-Event in TOSCA Der letzte Sub-Event des Publish-Subscribe-Event Node Types ist der Message-Broker-Event Node Type. Dieser Event wird ausgelöst, wenn eine neue Nachricht auf einem Topic eines Message-Brokers veröffentlicht wird, auf den sich die Serverless Funktion abonniert hat. Um die Properties Definition dieses Node Types möglichst abstrahiert spezifizieren zu können, betrachten wir erneut die benötigten Parameter, um ein Message-Broker Event auf den verschiedenen Serverless Plattformen programmatisch deployen zu können. Im Folgenden gehen wir davon aus, dass bereits eine Instanz des Message-Brokers existiert. Die zu betrachtenden Message-Broker Services lauten bei IBM *MessageHub*[3], bei der Google Cloud *Pub/Sub*[4] und bei AWS *SNS*[5]. Der Vergleich der benötigten Parameter für das Deployment dieses Events auf den verschiedenen Serverless Plattformen ist Tabelle 3.3 zu entnehmen. Die Tabelle zeigt, dass die drei Serverless Plattformen nahezu identische Parameter benötigen, um einen Message-Broker Event zu deployen. Wir befüllen daher die Properties Definition des Message-Broker-Event Node Types lediglich mit den Elementen *Topic* und *MessageHub-Instanz*

Serverless Plattformen in TOSCA

Durch die bisher beschriebenen Node Types von Serverless Funktionen und Events in TOSCA haben wir eine abstrahierte und plattformunabhängige Beschreibung dieser Komponenten in TOSCA erwirkt. Dementsprechend müssen nun sämtliche provider-spezifische Parameter für das programmatische Deployment von Serverless Funktionen und Events in die Properties Definition des jeweiligen provider-spezifischen Serverless Plattform Node Types gelegt werden. Für jede Serverless Plattform, die wir abbilden möchten, erstellen wir daher einen eigenen, provider-spezifischen Node Type mit dementsprechender Properties Definition.

Um beispielsweise die OpenWhisk Serverless Plattform auf TOSCA abzubilden, erstellen wir den Node Type *OpenWhiskPlattform*. Anschließend betrachten wir jegliche provider-spezifische Parameter für das programmatische Deployment von Serverless Funktionen und Events auf OpenWhisk. Daraus ergibt sich die in Listing 3.2 dargelegte Properties Definition des OpenWhiskPlattform Node Types. Im Folgenden wird nun anhand der provider-spezifischen Serverless Plattform Node Types das plattformübergreifende Deployment von Serverless Funktionen und Events anhand Management-Operationen in TOSCA beschrieben.

[3] https://developer.ibm.com/messaging/message-hub/
[4] https://cloud.google.com/pubsub/docs/overview?hl=de
[5] https://aws.amazon.com/sns

Listing 3.2 Properties Definition des OpenWhiskPlattform Node Types

```
<PropertiesDefinition elementname="properties"
    namespace="http:/example.com/nodetypes/propertiesdefinition">
        <properties>
           <key>OpenWhiskNamespace</key>
           <type>xsd:string</type>
        </properties>
        <properties>
           <key>OpenWhiskEndpoint</key>
           <type>xsd:string</type>
        </properties>
        <properties>
           <key>API-Key</key>
           <type>xsd:string</type>
        </properties>
</PropertiesDefinition>
```

3.3.2 Management einer Serverless Anwendung in TOSCA

An dieser Stelle sind nun alle Node Types spezifiziert, um eine Serverless Anwendung in TOSCA mittels eines Topology Templates zu erstellen. Nun möchten wir das plattformübergreifende Deployment der abstrahierten ServerlessFunktion und Event Node Types anhand der Implementierung von provider-spezifischen Management-Operationen ermöglichen. Außerdem möchten wir das automatisierte Management einer Serverless Anwendung in TOSCA durch Management-Pläne ermöglichen. Insbesondere möchten wir das Deployment einer Serverless Anwendung via Management-Pläne ermöglichen. Die Management-Pläne werden, wie in Abschnitt 2.2 beschrieben, in standardisierten Workflow-Sprachen erstellt, um die Portabilität dieser gewährleisten zu können.

Um die in Abschnitt 3.3.1 erstellten, provider-unabhängigen Node Types plattformübergreifend und automatisiert via TOSCA deployen zu können, muss jeder provider-spezifische Serverless Plattform Node Type die Logik beinhalten, um Serverless Funktionen und die verschiedenen Events deployen zu können. Um Management-Operationen anbieten zu können, benötigt ein Serverless Plattform Node Type zunächst ein Interface. Abbildung 3.6 legt die Beziehung zwischen Node Type, Interface und Management-Operationen dar. Repräsentativ für einen provider-spezifischen Serverless Plattform Node Type ist in der Abbildung der OpenWhiskPlattform Node Type dargestellt. Dieser bietet ein *ServerlessInterface* an, welches Management-Operationen für das Erlangen der verschiedenen Zustände einer Serverless Anwendung beinhaltet. Dabei werden für die Provisionierung einer Serverless Anwendung Management-Operationen angeboten, welche Serverless Funktionen oder Events auf einer Serverless Plattform deployen. Für die Deprovisionierung einer Serverless Anwendung werden Management-Operationen angeboten, welche eine Serverless Funktion oder ein Event von einer Severless Plattform entfernen.

Management-Operationen benötigen, um die jeweilige Management-Operation ausführen, beziehungsweise implementieren zu können, Eingabeparameter. Alternativ könnte die Implementierung des programmatischen Deployments der verschiedenen Events auch innerhalb einer Management-Operation erfolgen. Da sich die für das programmatische Deployment der verschiedenen Events benötigten Properties Definitions allerdings je nach Event stark unterscheiden,

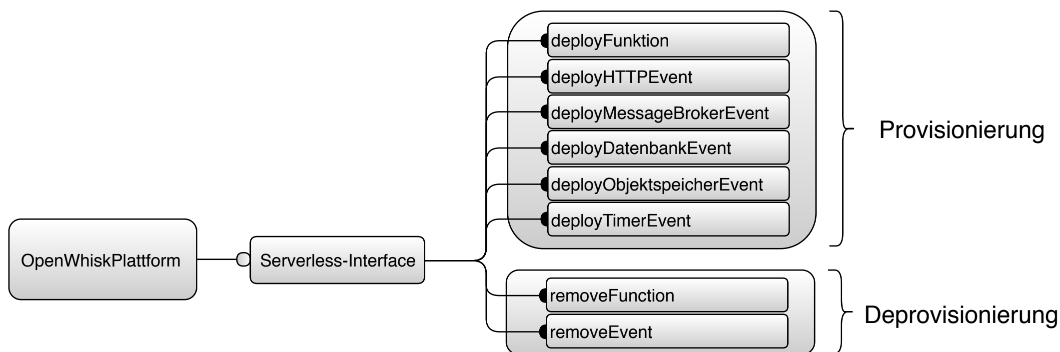

Abbildung 3.6: Beziehungen zwischen Node Type, Interface und Management-Operationen

resultiert dies in einer schwer zu überblickenden Menge an Eingabeparametern. Da das programmatische Entfernen eines Events neben den Zugangsdaten lediglich den Namen des Events benötigt, genügt eine Management-Operation für das Entfernen von Events für alle verschiedenen Arten von Events. Die deployFunktion Management-Operation aus Abbildung 3.6 benötigt beispielsweise folgende Eingabeparameter, um eine Serverless Funktion auf OpenWhisk zu deployen: (1) Funktionsname, (2) Laufzeitumgebung, (3) Zeitlimit, (4) Speicherlimit, (5) OpenWhisk-Namespace, (6) OpenWhisk-Endpoint und (7) den API-Schlüssel. Die Eingabeparameter können zur Laufzeit des Service Templates eingegeben werden, oder, je nach Implementierung der TOSCA Ziellaufzeitumgebung, aus den Properties Definitions der in dem Topology Template enthaltenen Node Templates auf die Eingabeparameter abgebildet werden.

Um die dargelegten Management-Operationen des ServerlessInterfaces zu implementieren, muss zudem noch eine Node Type Implementation des Node Types erstellt werden. Die Implementierung der Management-Operationen soll nach TOSCA Spezifikation in standardisierten Technologien erfolgen, um die Portabilität dieser gewährleisten zu können [BBLS12]. Für die Implementierung der jeweiligen Management-Operationen der provider-spezifischen Serverless Plattform Node Types empfiehlt es sich, die Programmiersprache, beziehungsweise Technologie, der Implementierung davon abhängig zu machen, ob für die Interaktion mit der Serverless Plattform ein SDK für eine bestimmte Programmiersprache existiert. Jedoch muss dabei beachtet werden, ob die TOSCA Ziellaufzeitumgebung die ausführbare Datei der Implementierung der jeweils gewählten Programmiersprache ausführen kann. Die ausführbaren Dateien der Implementierung der Management-Operation befindet sich schließlich in dem Implementation Artifact der jeweiligen Node Type Implementation. Nun müssen den Anforderungen entsprechend die Management-Operationen implementiert werden. Um beispielsweise die deployFunction Management-Operation für den OpenWhiskPlattform Node Type zu implementieren, muss eine HTTP Anfrage für das Erstellen einer Serverless Fuktion an OpenWhisk gestellt werden.

Dadurch, dass ein Serverless Plattform Node Type die genannten Management-Operationen anbietet und implementiert, kann schließlich jeder provider-spezifische Serverless Plattform Node Type die provider-spezifische Logik mit sich bringen, um einen ServerlessFunktion und die verschiedenen Event Node Types automatisiert und programmatisch via TOSCA deployen zu können. Dies ermöglicht erst die abstrahierte Repräsentation von Serverless Funktionen und Events in TOSCA. Dies bedeutet, dass unsere abstrahierten Serverless Funktion und Event Node Types nun plattformübergreifend via TOSCA auf verschiedenen Serverless Plattformen deployed

werden können, da jede Serverless Plattform die provider-spezifische Logik besitzt, um Serverless Funktionen und Events deployen zu können. Dadurch sind Serverless Funktionen und Events dank TOSCA plattformübergreifend und kompatibel auf verschiedensten Serverless Plattformen deploybar. Einzig die provider-spezifische Funktionssignatur, beziehungsweise Übergabe des Rückgabewerts einer Serverless Funktion muss je nach Zielplattform weiterhin geändert werden. Aus den Inkompatibilitäten aus Abbildung 2.3 ergibt sich dadurch Abbildung 3.7. Um eine Serverless Anwendung nun automatisiert via TOSCA zu managen, müssen schlussendlich Management-Pläne erstellt werden, welche die Provisionierung und die Deprovisionierung einer Serverless Anwendung wiederspiegeln. Diese werden, wie bereits beschrieben, in standardisierten Workflow-Sprachen wie beispielsweise BPMN oder BPEL erstellt.

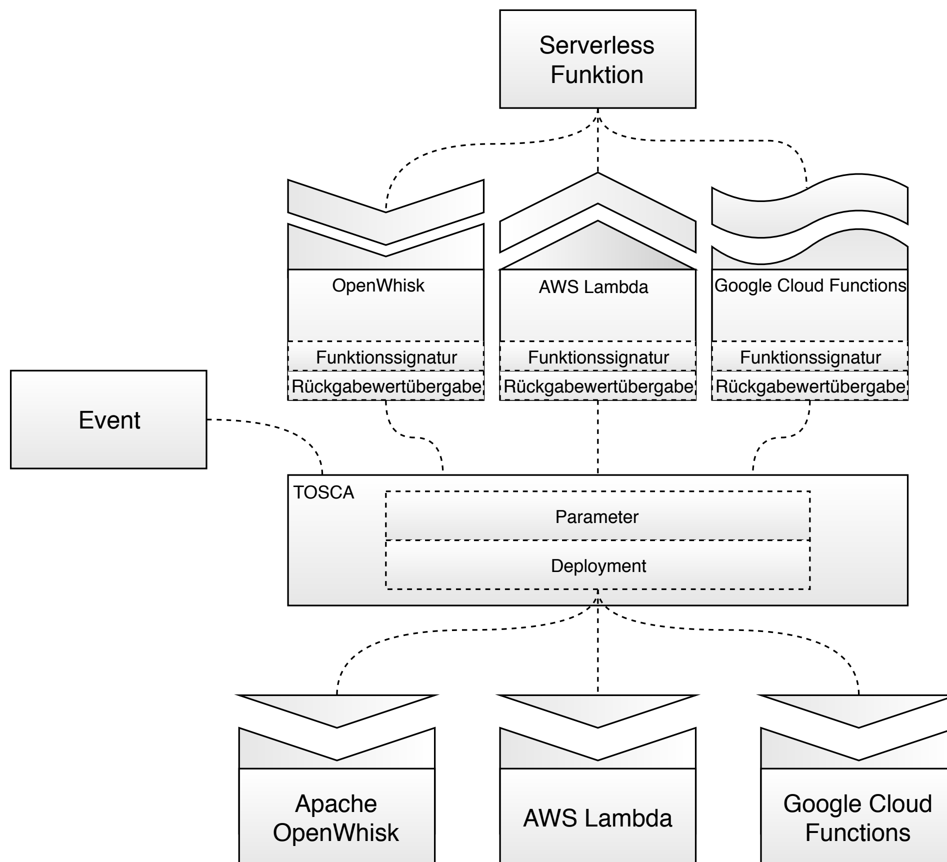

Abbildung 3.7: Plattformübergreifende Kompatibilität von Serverless Funktionen dank TOSCA

3.4 Programmiermodell für Serverless Anwendungen in TOSCA

In Abschnitt 3.3 haben wir die von uns identifizierten Komponenten einer Serverless Anwendung auf verschiendene TOSCA Elemente abgebildet. Nun möchten wir eine Serverless Anwendung anhand der definierten Elemente in TOSCA erstellen und anschließend bereitstellen. Wir nehmen an, dass die benötigten Serverless Funktionen und Events, sowie deren Zusammenspiel, bereits identifiziert wurden.

Zunächst erstellen wir ein TOSCA Service Template zur Repräsentation der Serverless Anwendung, welches nun in Topology Template und Management-Pläne unterteilt wird. Um die Struktur unserer Serverless Anwendung abzubilden, erstellen wir das Topology Template, in dem wir die von uns gewünschte Architektur der Anwendung mittels Node und Relationship Templates darstellen. Anschließend befüllen wir die Elemente der Properties Definition aller genutzten Node Templates mit den gewünschten, beziehungsweise benötigten Werten. Für das Deployment von Serverless Funktionen müssen diese nun als Deployment Artifact des Artifact Types ArchiveArtifact an das entsprechende Node Template angehangen werden.

Ist das Erstellen des Topology Templates abgeschlossen, werden die Management-Pläne für die Anwendung erstellt. Dafür ist es wichtig, dass die Management-Pläne in einer standardisierten Workflow-Sprache erstellt werden, um die Portabilität dieser sicherzustellen. TOSCA empfiehlt hier die Nutzung von BPEL oder BPMN. TOSCA sieht Management-Pläne für das Provisionieren, Verwalten und Deprovisionieren einer Anwendung vor. Der Management-Plan zum Provisionieren setzt sich aus dem Ausführen, beziehungsweise Aufrufen der Management-Operationen für das Deployment der in dem Topology Template enthaltenen Node Templates, zusammen. Hierbei muss besonders auf die Reihenfolge des Aufrufens der Management-Operationen geachtet werden; um beispielsweise einen Event mit einer Serverless Funktion zu verbinden, muss die Serverless Funktion vor dem Event deployed werden. Analog zu der Provisionierung erfolgt das Erstellen des Management-Plans zur Deprovisonierung. Allerdings setzt sich dieser aus den Management-Operationen für das Entfernen der einzelnen Node Templates des Topology Templates zusammen. Management-Pläne zur Verwaltung werden an dieser Stelle vernachlässigt.

Sind die Management-Pläne und das Topology Template erstellt, können wir das Service Template als CSAR exportieren. Um nun von den TOSCA Konzepten profitieren zu können, benötigen wir eine TOSCA-konforme Laufzeitumgebung, welche unser CSAR interpretieren kann. Diese Laufzeitumgebung stellt das automatisierte Management sicher, indem sie eine Komponente zur Ausführung der Management-Plänen zur Verfügung stellt. Nun importieren wir das CSAR in die TOSCA-konforme Laufzeitumgebung. Anschließend kann die Serverless Anwendung den Management-Plänen entsprechend beliebig provisioniert oder deprovisioniert werden.

4 Implementierung

Dieses Kapitel beinhaltet eine prototypische Implementierung des in Kapitel 3 vorgestellten Ansatzes, mit Fokus auf der Implementierung der Management-Operationen eines provider-spezifischen Serverless Plattform Node Types. Die Implementierung basiert dabei auf der Serverless Plattform OpenWhisk und der TOSCA-konformen Laufzeitumgebung OpenTOSCA.

4.1 Prototypische Implementierung des Konzepts via OpenTOSCA

Um das in Kapitel 3 erstellte Konzept für die provider-unabhängige Abbildung von Serverless Funktionen und Events in TOSCA zu validieren, nutzen wir im Folgenden das OpenTOSCA Ökosystem. OpenTOSCA ist ein an der Universität Stuttgart entwickeltes Open-Source-Projekt, welches den TOSCA Standard implementiert. Hauptsächlich besteht OpenTOSCA aus drei Kernkomponenten: (1) Winery[1], einer Modellierungsumgebung für das Erstellen von jeglichen TOSCA Types und Topology, beziehungsweise Service Templates, (2) OpenTOSCA Container, einer TOSCA-konformen Laufzeitumgebung für das Prozessieren von CSARs, sowie (3) OpenTOSCA UI, ein Self-Service Portal zur Provisionierung und Deprovisionierung von in TOSCA beschriebenen Anwendung.

In Winery legen wir zunächst die in Abschnitt 3.3 spezifizierten Node, Relationship und Artifact Types an. Da für die prototypische Implementierung des Konzepts die OpenWhisk Plattform genutzt wird, erstellen wir zudem den Node Type OpenWhiskPlattform. Zudem legen wir für den OpenWhiskPlattform Node Type das in Abschnitt 3.3.2 beschriebene ServerlessInterface, welches die in Abbildung 3.6 dargelegten Management-Operationen enthält, an. Nun gilt es, diese Management-Operationen für den OpenWhiskPlattform Node Type zu implementieren, um das programmatische Deployment der in Abschnitt 3.3 spezifizierten, abstrahierten Node Types zu ermöglichen. Die Implementierung soll dabei nach TOSCA in einer standardisierten Technologie erfolgen, um die Portabilität des Management-Aspekts einer Anwendung in TOSCA zu garantieren. Für die Wahl der Technologie der Implementierung muss allerdings die Ziellaufzeitumgebung des Service Templates betrachtet werden. Der OpenTOSCA Container, die TOSCA-konforme Laufzeitumgebung des OpenTOSCA Ökosystems, bietet aktuell eine Laufzeitumgebung für Management-Operationen in Form eines Java Web Services und eines Shell-Skripts. Um die Management-Operationen implementieren zu können, legen wir eine Node Type Implementation für das ServerlessInterface des OpenWhiskPlattform Node Types in Winery an. Für die Implementierung der Management-Operationen wählen wir Java Web Services. Unter Angabe des Artifact Types und der zu implementierenden Node Type Implementation generiert

[1] https://projects.eclipse.org/projects/soa.winery

Listing 4.1 Vereinfachte Java-Klasse mit den zu implementierenden Management-Operationen

```
package org.opentosca.nodetypeimplementations;

import ...

@WebService
public class org_opentosca_nodetypes_OpenWhiskPlattform__ServerlessInterface
      extends AbstractIAService {

   @WebMethod
   @SOAPBinding
   @Oneway
   public void deployFunction(
         @WebParam(name = "OpenWhiskEndpoint", targetNamespace =
             "http://nodetypeimplementations.opentosca.org/") final String OpenWhiskEndpoint,
      ...
       ) {
      // This HashMap holds the return parameters of this operation.
      final HashMap<String, String> returnParameters = new HashMap<>();

      // implement your operation here

      // send the response
      sendResponse(returnParameters);
   }
   ...
}
```

Winery ein Implementation Artifact, welches ein Java Web Service Projekt mit Maven[2] enthält. Der Artifact Type des Implementation Artifacts ist daher WAR. Das generierte Maven Projekt ist dabei auf das OpenTOSCA Ökosystem zugeschnitten und beinhaltet eine Hilfsklasse *AbstractIAService.java*, um mit dem OpenTOSCA Container und den darin enthaltenen Komponenten interagieren zu können. Nun exportieren wir das Java Web Service Projekt, um die einzelnen Management-Operationen implementieren zu können. Winery generiert dabei Java Methoden, welche die Management-Operationen des zu implementierenden Interfaces inklusive deren Eingabeparameter repräsentieren. Die zu implementierenden Methoden befinden sich dabei in der Java Klasse *org_opentosca_nodetypes_OpenWhiskPlattform_ServerlessInterface.java*. Listing 4.1 zeigt einen vereinfachten Auszug dieser Klasse.

4.1.1 Implementierung der Management-Operation für das Deployment einer Serverless Funktion via TOSCA

Um das Deployment einer Serverless Funktion auf der OpenWhisk Plattform zu ermöglichen, muss nun die Methode deployFunction mit der entsprechenden, provider-spezifischen Logik implementiert werden. Dazu muss mit der OpenWhisk API interagiert werden. Da zum Zeitpunkt des Verfassens dieser Thesis kein Java Client SDK für die Interaktion mit OpenWhisk existiert, werden die Management-Operationen mittels HTTP Anfragen an die OpenWhisk Plattform implementiert. Um eine Serverless Funktion programmatisch auf OpenWhisk deployen zu können,

[2] https://maven.apache.org

Listing 4.2 Implementierung der deployFunction Management-Operation in Java

```
// create the payload to send with the request
final String payLoad = "{\"namespace\":\"" + OpenWhiskNamespace + "\", \"name\":\"" +
    Funktionsname + "\", \"exec\":{\"kind\":\"" + Laufzeitumgebung + "\", \"code\":\"" +
    Funktionscode + "\"}}";
// openwhisk endpoint to create a function
 final String requestUrl = "https://" + APIKey + "@" + OpenWhiskEndpoint +
     "/api/v1/namespaces/" + OpenWhiskNamespace + "/actions/" + Funktionsname +
     "?overwrite=true";

final StringEntity entity = new StringEntity(payLoad, ContentType.APPLICATION_JSON);

// buildup of http client
final HttpClient httpClient = HttpClientBuilder.create().build();
final HttpPut request = new HttpPut(requestUrl);
// set payload and header
request.setEntity(entity);
request.setHeader("Accept", "application/json");
// execute the request and get the response
final HttpResponse response = httpClient.execute(request);
```

muss unter Angabe des Funktionsnamen, der Laufzeitumgebung, des OpenWhisk Namespaces sowie Endpunkts und des zugehörigen API-Schlüssels eine *HTTP PUT* Anfrage an die OpenWhisk API gestellt werden. Die HTTP PUT Anfrage ermöglicht das Hochladen einer Ressource auf einen Webserver [FGM+99]. Die Variable *payLoad* beinhaltet dabei die mitzugebenden Parameter für die Anfrage an die OpenWhisk API, wie beispielsweise den Funktionsnamen oder die Laufzeitumgebung der zu deployenden Serverless Funktion. Anschließend wird aus dem jeweiligen OpenWhisk Endpunkt und dem Namespace, sowie dem API-Schlüssel in der Variable *requestUrl* die URL gebildet, an welche die Anfrage gestellt wird. Schließlich wird die Anfrage mittels eines HTTP Clients ausgeführt.

4.1.2 Implementierung der Management-Operation für das Deployment eines Events via TOSCA

Um einen Event auf der OpenWhisk Plattform zu deployen, muss, analog zu dem Deployment einer Serverless Funktion, mittels HTTP Anfragen unter Angabe der jeweiligen Parameter mit der OpenWhisk API interagiert werden. Da Events in OpenWhisk mittels Rules mit Serverless Funktionen verknüpft werden, besteht die Implementierung der Management-Operation, um einen Event Node Type in TOSCA deployen zu können, aus mehreren HTTP Anfragen. Zunächst wird mittels einer HTTP PUT Anfrage unter Angabe des Namens des Events und des Eventtyps der Event in OpenWhisk erstellt. Anschließend wird der erstellte Event mittels einer HTTP POST Anfrage unter Angabe der event-spezifischen Parameter, wie beispielsweise ein Cron-Ausdruck für das Deployment eines Timer-Events, konfiguriert. Um nun den Event mit einer Serverless Funktion zu verknüpfen, wird mittels einer HTTP PUT Anfrage an die OpenWhisk API unter Angabe des Event-, sowie Funktionsnamens die entsprechende Assoziation zwischen Serverless Funktion und Event hergestellt. Listing 4.3 zeigt die Implementierung der Verknüpfung von Funktion und Event. Analog dazu werden die verschiedenen Management-Operationen für das Deployment der weiteren Events implementiert.

Listing 4.3 Implementierung der Verknüpfung von Event und Serverless Funktion

```
// create rule
final String payLoadCreateRule = "{\"name\":\"" + Eventname + "Rule" + "\", \"status\":\"\",
    \"trigger\":\"/" + OpenWhiskNamespace + "/" + Eventname + "\", \"action\":\"/" +
    OpenWhiskNamespace + "/" + Funktionsname + "\"}";
final String requestUrlCreateRule = "https://" + APIKey + "@" + OpenWhiskEndpoint +
    "/api/v1/namespaces/" + OpenWhiskNamespace + "/rules/" + Eventname + "_" + Funktionsname
    + "?overwrite=true";
final StringEntity entityRule = new StringEntity(payLoadCreateRule,
    ContentType.APPLICATION_JSON);

final HttpClient httpClient = HttpClientBuilder.create().build();
final HttpPut requestRule = new HttpPut(requestUrlCreateRule);
requestRule.setEntity(entityRule);
requestRule.setHeader("Accept", "application/json");
final HttpResponse responseRule = httpClient.execute(requestRule);
```

4.1.3 Implementierung der Management-Operation für das Entfernen von Serverless Funktionen und Events via TOSCA

Um den gesamten Lebenszyklus einer Serverless Anwendung abdecken zu können, implementieren wir Management-Operationen, welche das Entfernen von Serverless Funktionen und Events erwirken. Um dies zu realisieren, stellen wir unter Angabe des Funktions-, beziehungsweise Eventnamens eine HTTP DELETE Anfrage an die OpenWhisk API. Da das Entfernen eines Events über den Parameter des Eventnamens erfolgt, genügt dabei die Implementierung einer Management-Operation, um alle verschiedenen Eventtypen entfernen zu können.

Sind alle Management-Operationen implementiert, so bauen wir das Java Web Service Projekt als eine ausführbare WAR Datei. Anschließend importieren wir die ausführbare WAR Datei, welche die Implementierung der Management-Operationen enthält, wieder in das Implementation Artifact in Winery, aus welchem es exportiert wurde. Dadurch ist der OpenWhiskPlattform Node Type fähig, Serverless Funktionen und Events programmatisch deployen zu können. Analog dazu kann jeder benötigte provider-spezifische Serverless Plattform Node Type implementiert werden.

4.1.4 Management einer Serverless Anwendung in OpenTOSCA

Der OpenTOSCA Container, welcher gemeinsam mit den weiteren Komponenten des OpenTOSCA Ökosystems in Abbildung 4.1 simplifiziert dargestellt ist, stellt die TOSCA-konforme Laufzeitumgebung des OpenTOSCA Ökosystems dar. Wie bereits in Abschnitt 2.2 erläutert, existieren die deklarative und imperative Verarbeitungsmethode für CSARs in TOSCA. Der OpenTOSCA Container kombiniert jedoch diese beiden Ansätze zu einem Hybrid des deklarativen und imperativen Prozessierens. Hierbei werden im OpenTOSCA Container, basierend auf dem Topology Template eines CSARs, ein ausführbarer BPEL Plan erstellt, welcher im Nachhinein adaptiert werden kann [BBK+14].

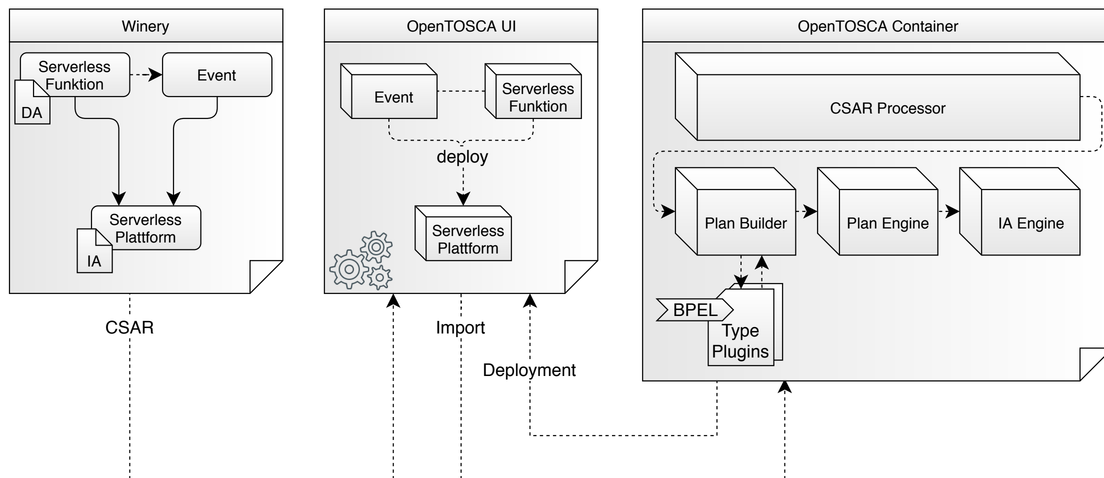

Abbildung 4.1: Vereinfachte Darstellung des OpenTOSCA Ökosystems

Mittels der OpenTOSCA UI, einem Self-Service-Portal für das Management von Cloud Anwendungen in OpenTOSCA, können CSARs in den OpenTOSCA Container importiert werden. Dabei interpretiert der *Plan Builder* des OpenTOSCA Containers das Topology Template der Anwendung und erstellt darauf basierend einen BPEL Management-Plan, sowohl für die Provisionierung als auch für die Deprovisionierung einer Anwendung in TOSCA. Dabei konkateniert der Plan Builder das Aufrufen der einzelnen, angebotenen Management-Operationen der in dem Topology Template enthaltenen Node Templates. Die Reihenfolge der Konkatenation basiert dabei auf der Struktur des Topology Templates. So werden Node Templates, welche nicht Quellelement eines Relationship Templates sind, zuerst prozessiert. Ist ein Node Template abgearbeitet, so werden die mit diesem Node Template verbundenen Node Templates prozessiert. Dadurch wird die korrekte Reihenfolge der Ausführung der Management-Operationen sichergestellt. Sind beispielsweise zwei Node Templates mit einem Relationship Templates des Relationship Types HostedOn verbunden, so wird das Zielelement des Relationship Templates vor dem Quellelement prozessiert. Dabei werden die Properties Definitions der in dem Topology Template enthaltenen Node Templates auf die jeweiligen Eingabeparameter der Management-Operationen abgebildet und in den Management-Plan eingefügt. Allerdings können die Management-Operationen eines Node Templates nicht auf Properties Definitions zugreifen, zu denen sie keine ausgehende Kante haben.

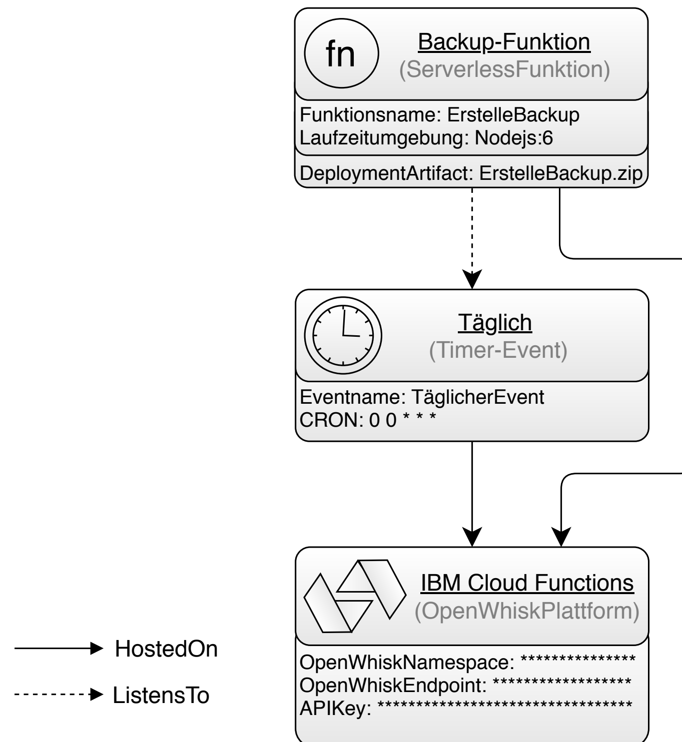

Abbildung 4.2: Topologie unserer beispielhaften Serverless Anwendung in Winery

Abbildung 4.2 zeigt die visuelle Darstellung eines Topology Templates, welches wir in Winery erstellen. Das Topology Template besteht aus einem Node Template des ServerlessFunktion Node Types, einem Node Template des OpenWhiskPlattform Node Types und einem Node Template des Timer-Event Node Types. Da die in dem Node Template des OpenWhiskPlattform Node Types enthaltenen Management-Operationen allerdings die Werte von Properties Definitions von Node Templates benötigen, zu welchen sie keine ausgehenden Kanten haben, muss der Plan Builder um ein sogenanntes *Type Plugin* erweitert werden, welches dies ermöglicht. Da das innerhalb des OpenWhiskPlattform erstellte ServerlessInterface und dessen Management-Operationen außerdem von dem TOSCA Lifecycle Interface abweicht, muss das Type Plugin mit der Logik ausgestattet werden, welche Management-Operation als Reaktion auf welche verbundenen Node Templates ausgeführt werden sollen. Dazu erstellen wir innerhalb des OpenTOSCA Containers ein *ServerlessTypePlugin* und regestrieren es als Type Plugin für den Plan Builder. Der Plan Builder prüft vor der Erstellung eines Management-Plans, ob für die Node Templates des zu prozessierenden Topology Templates Plugins existieren. Daher implementieren wir das ServerlessTypePlugin so, dass es Node Templates des ServerlessFunktion Node Types identifizieren kann und daraufhin aufgerufen wird. Ausgehend davon prüft das ServerlessTypePlugin, ob das erkannte Node Template eines ServerlessFunktion Node Types mit einem Serverless Plattform Node Type verbunden ist. Ist dies der Fall, wird die Properties Definition des Node Templates des ServerlessFunktion Node Types auf die Eingabeparameter der deployFunction Management-Operation des zugrundeliegenden Serverless Plattform Node Types abgebildet. Anschließend wird diese Management-Operation inklusive abgebildeter Eingabeparameter zu dem BPEL Management-Plan für das Provisionieren einer Anwendung in OpenTOSCA hinzugefügt. Ist dies erfolgt, wird geprüft, ob das Node

Template des ServerlessFunktion Node Types über ausgehende Kanten zu einem Node Template eines unterstützen Event Node Types verfügt. Ist dies der Fall, wird die Properties Definition des Node Templates des Event Node Types auf die Eingabeparameter der für den jeweiligen Eventtyp passenden Management-Operation abgebildet. Anschließend wird die Management-Operation inklusive der abgebildeten Eingabeparameter zu dem Management-Plan für die Provisionierung hinzugefügt. Für die beispielhafte Anwendung aus 4.2 bedeutet dies, dass der Management-Plan für das Provisionieren der Anwendung zunächst die Management-Operation für das Deployment der Serverless Funktion aufruft. Anschließend wird die Management-Operation deployTimerEvent für das Deployment des Timer-Events aufgerufen.

Da die OpenWhisk Plattform kein Java Client SDK anbietet und Serverless Funktionen, welche als ZIP-Archiv deployed werden, intern als Base64[3]-Zeichenkette enkodiert, enthält das ServerlessTypePlugin noch weitere Hilfsmethoden. Diese extrahieren das Deployment Artifact eines Node Templates eines ServerlessFunktion Node Types, welches das ZIP-Archiv einer Serverless Funktion beinhaltet. Anschließend wird das ZIP-Archiv als Base64-Zeichenkette enkodiert, um es der deployFunction Management-Operation als Eingabeparameter mitzugeben. Dieser weitere Eingabeparameter wird daher innerhalb des Kontexts des Provisionierungsplans erstellt und mit dem Wert der Base64-Zeichenkette belegt.

Importiert man nun via OpenTOSCA UI die in Winery erstellte, beispielhafte Serverless Anwendung aus Abbildung 4.2 in den OpenTOSCA Container, so erstellt die Komponente des Build Plans anhand des ServerlessTypePlugins einen adäquaten Management-Plan für die Provisionierung dieser Anwendung. Abbildung 4.1 illustriert die weiteren Schritte innerhalb des OpenTOSCA Ökosystems. Wird ein CSAR in der OpenTOSCA UI instanziiert, so wird die *Plan Engine* aufgerufen, welche daraufhin den Management-Plan für die Provisionierung ausführt. Im Zuge des Ausführens des Management-Plans werden die Management-Operationen für das Deployment der einzelnen Node Templates innerhalb der *IA Engine* ausgeführt, welche die Laufzeitumgebung für das Ausführen der Implementation Artifacts bereitstellt.

4.2 Validierung der Implementierung

Um die in Abschnitt 4.1 beschriebene, prototypische Implementierung des Konzepts zu validieren, erstellen wir eine beispielhafte Serverless Anwendung in Winery und importieren diese anschließend via OpenTOSCA UI in den OpenTOSCA Container. Schließlich wird die Anwendung mittels der OpenTOSCA UI instanziiert.

Nach der Erstellung eines Service Templates in Winery modellieren wir zunächst eine beispielhafte Serverless Anwendung anhand der erstellten Node und Relationship Types. Wir erstellen zur Validierung der Implementierung eine simple Serverless Anwendung, bei der eine Serverless Funktion als Reaktion auf die Änderung innerhalb einer Datenbank ausgeführt werden soll. Als Rückgabewert der Serverless Funktion legen wir eine Nachricht fest, welche das Ausführen bestätigt.

[3]https://tools.ietf.org/html/rfc4648

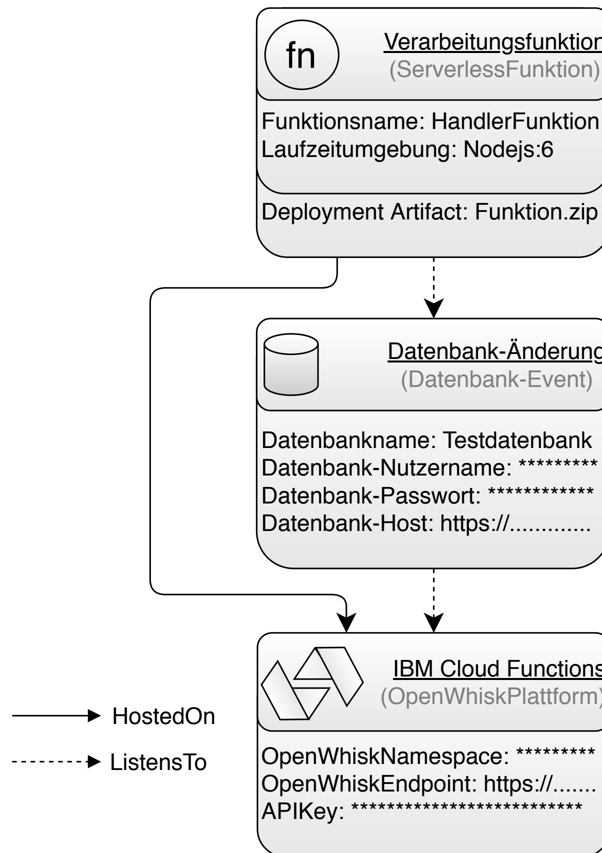

Abbildung 4.3: Visuelle Repräsentation des für die Validierung der Implementierung erstellten Topology Templates

Die in Winery modellierte Anwendung ist Abbildung 4.3 zu entnehmen. Wir belegen nun die Elemente der jeweiligen Properties Definitions der Node Templates mit den entsprechenden Werten. Dabei werden optionale Elemente der Properties Definition vernachlässigt. Anschließend wird das erstellte Service Template als CSAR exportiert.

Nun importieren wir das CSAR via OpenTOSCA UI in den OpenTOSCA Container. Dort wird durch den Plan Builder und das ServerlessTypePlugin anhand der Interpretation des Topology Templates ein BPEL Management-Plan für die Provisionierung unserer Anwendung erstellt. Dieser Management-Plan beinhaltet das Aufrufen der Management-Operationen deployFunction und deployTimerEvent. Ist das Prozessieren des CSARs durch den OpenTOSCA Container abgeschlossen, können wir via OpenTOSCA UI eine neue Instanz der Anwendung provisionieren. Nun führen die Komponenten Plan Engine, beziehungsweise IA Engine, den Management-Plan und die darin enthaltenen Management-Operationen, welche wir als Java Web Service implementiert hatten, aus. Ist die Ausführung abgeschlossen, wird in der OpenTOSCA UI die erfolgreiche Provisionierung bestätigt.

Um dies zu validieren, fügen wir einen neuen Datensatz in die mit der Serverless Funktion verknüpften Datenbank ein. Mittels des OpenWhisk CLIs prüfen wir nun die zuletzt aufgerufene Serverless Funktion und erhalten die in Listing 4.4 abgebildete Antwort.

Listing 4.4 Antwort des CLI auf die Abfrage nach der Aktivierung unserer Serverless Funktion


```
ok: got activation c60173c65f544f018173c65f544f0114
{
   "namespace": "mathony.tobias@gmail.com_dev",
   "name": "HandlerFunktion",
   "version": "0.0.1",
   "subject": "mathony.tobias@gmail.com",
   "activationId": "c60173c65f544f018173c65f544f0114",
   "start": 1522838504902,
   "end": 1522838504972,
   "duration": 70,
   "response": {
      "status": "success",
      "statusCode": 0,
      "success": true,
      "result": {
         "message": "The connected database was updated!"
      }
   }
}
```


Die Antwort des CLI bestätigt die Ausführung der Serverless Funktion als Reaktion auf die Aktualisierung der Datenbank und damit die korrekte Provisionierung der beispielhaften Anwendung.

5 Zusammenfassung und Ausblick

Dieses Kapitel beinhaltet neben einer Zusammenfassung dieser Arbeit einen Ausblick auf die mögliche Erweiterung des erstellten Konzepts und dessen Implementierung.

Zusammenfassung

Ziel dieser Arbeit war die Erstellung eines Konzepts, um Serverless Funktionen und Events provider-agnostisch in TOSCA zwischen verschiedenen Serverless Plattformen kompatibel zu beschreiben.

Zunächst wurden mit der Beschreibung des Serverless Computing Ansatzes und der Funktionsweise der, zur Implementierung des Konzepts genutzten Serverless Plattform, Apache OpenWhisk die Grundlagen des Serverless Computing Paradigmas erläutert. Außerdem wurden im Bezug auf diese Thesis verwandte Arbeiten, wie das provider-agnostische Serverless Framework und ein darauf basierendes Konzept einer provider-agnostischen Serverless Anwendungsarchitektur beschrieben. Anschließend wurden die für diese Thesis relevanten Konzepte von TOSCA dargelegt.

Für die Erstellung des Konzepts wurde zunächst ein Metamodell der Anatomie einer Serverless Anwendung erstellt. Aufbauend auf den Konzepten TOSCAs wurde daraufhin eine Abbildung des Metamodells einer Serverless Anwendung auf TOSCA Elemente erzielt. Dafür war es elementar, eine Serverless Funktion, sowie Events einer Serverless Anwendung abstrahiert von spezifischen Serverless Plattformen in TOSCA darzustellen. Um die plattformübergreifende Kompatibilität von Serverless Funktionen und Events zu erwirken, wurde die plattform-spezifische Logik für das Deployment von Serverless Funktionen und Events innerhalb provider-spezifischer Management-Operationen implementiert. Unter der Voraussetzung, dass jede Darstellung einer provider-spezifischen Serverless Plattform in TOSCA die Logik mitliefert, um Serverless Funktionen und Events zu deployen, werden diese dadurch plattformübergreifend kompatibel. Außerdem wird durch das Implementieren der Management-Operationen das automatisierte Management einer Serverless Anwendung in TOSCA ermöglicht.

Dieses Konzept wurde anschließend, basierend auf dem OpenTOSCA Ökosystem und der OpenWhisk Serverless Plattform, prototypisch implementiert. Dazu wurden die Management-Operationen für das Deployment von Serverless Funktionen und Events provider-spezifisch für die OpenWhisk Plattform als Java Web Service implementiert. Um die Implementierung zu validieren, wurde eine beispielhafte Serverless Anwendung in Winery, der Modellierungsumgebung des OpenTOSCA Ökosystems, erstellt. Anschließend wurde der OpenTOSCA Container um ein Plugin erweitert, um das automatisierte Erstellen von Management-Plänen für die Provisionierung von Serverless Anwendungen in OpenTOSCA zu ermöglichen. Schließlich wurde die in Winery erstellte Anwendung mittels der OpenTOSCA UI erfolgreich provisioniert.

Ausblick

Aufgrund der zu diesem Zeitpunkt relativ kurzen Existenz des Serverless Computings, ist zu erwarten, dass sich sowohl der Ansatz des Serverless Computing als auch die Serverless Plattformen verändern, beziehungsweise weiterentwickeln. Außerdem arbeitet die *Cloud Native Computing Foundation* (CNCF) aktuell an der Homogenisierung der Umsetzung des Serverless Computing Paradigmas. Dadurch entstehende Änderungen müssten innerhalb des erstellten Konzepts dementsprechend angepasst werden. Des Weiteren könnte eine Art kanonisches Modell erstellt werden, welches den Funktionscode einer Serverless Funktion abstrahiert von Funktionssignatur und Rückgabewertübergabe verpackt und auf das Modell der jeweiligen, provider-spezifischen Serverless Plattform übersetzt. Außerdem könnte das erstellte Konzept um die Integration des Event Gateways erweitert werden.

Da Serverless Funktionen, wie beispielsweise in der OpenWhisk Plattform implementiert, ebenfalls als Event für Serverless Funktionen dienen können, könnte die Abbildung dieses Szenarios in das Konzept integriert werden. Denkbar wäre die Erweiterung von Winery um eine Entwicklungsumgebung für das Erstellen von daraus entstehenden Workflows von Funktionen.

Aufgrund des Fehlens eines OpenWhisk Java Client SDKs wurde das Deployment von Serverless Funktionen in Form eines ZIP-Archivs via TOSCA mittels verschiedener Hilfsmethoden implementiert. Nach Veröffentlichung eines entsprechenden Java Client SDKs für die Interaktion mit OpenWhisk könnte die Implementierung dahingehend angepasst werden. Außerdem könnten die provider-spezifischen Management-Operationen für weitere Serverless Plattformen, wie beispielsweise AWS Lambda oder Google Cloud Functions implementiert werden.

Literaturverzeichnis

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia. „A View of Cloud Computing“. In: *Commun. ACM* 53.4 (Apr. 2010), S. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: http://doi.acm.org/10.1145/1721654.1721672 (zitiert auf S. 13).

[Ama16] Amazon Web Services. 2016. URL: https://de.slideshare.net/AmazonWebServices/getting-started-with-aws-lambda-and-the-serverless-cloud/29 (zitiert auf S. 19).

[Apa17a] Apache OpenWhisk. *Getting started with OpenWhisk*. 2017. URL: https://github.com/apache/incubator-openwhisk/tree/master/docs#readme (zitiert auf S. 20).

[Apa17b] Apache OpenWhisk. *Using the Cloudant package*. 2017. URL: https://github.com/apache/incubator-openwhisk-package-cloudant/blob/master/README.md (zitiert auf S. 23).

[Apa18a] Apache OpenWhisk. *About OpenWhisk*. 2018. URL: http://openwhisk.incubator.apache.org/about.html (zitiert auf S. 20).

[Apa18b] Apache OpenWhisk. *Creating and invoking OpenWhisk actions*. 2018. URL: https://github.com/apache/incubator-openwhisk/blob/master/docs/actions.md (zitiert auf S. 21).

[Apa18c] Apache OpenWhisk. *OpenWhisk package for communication with Kafka or IBM Message Hub*. 2018. URL: https://github.com/apache/incubator-openwhisk-package-kafka/blob/master/README.md (zitiert auf S. 23).

[Apa18d] Apache OpenWhisk. *System overview*. 2018. URL: https://github.com/apache/incubator-openwhisk/blob/master/docs/about.md (zitiert auf S. 22).

[Apa18e] Apache OpenWhisk. *Using the Alarm package*. 2018. URL: https://github.com/apache/incubator-openwhisk-package-alarms/blob/master/README.md (zitiert auf S. 23).

[Apa18f] Apache OpenWhisk. *Using the Github package*. 2018. URL: https://github.com/apache/incubator-openwhisk-catalog/blob/master/packages/github/README.md (zitiert auf S. 23).

[Apa18g] Apache OpenWhisk. *Using the Push package*. 2018. URL: https://github.com/apache/incubator-openwhisk-package-pushnotifications/blob/master/README.md (zitiert auf S. 23).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. „OpenTOSCA – A Runtime for TOSCA-based Cloud Applications“. In: *11th International Conference on Service-Oriented Computing*. LNCS. Springer, 2013 (zitiert auf S. 32).

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. „Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA“. In: *Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014)*. IEEE Computer Society, März 2014, S. 87–96. DOI: 10.1109/IC2E.2014.56 (zitiert auf S. 32, 52).

[BBKL14] T. Binz, U. Breitenbüecher, O. Kopp, F. Leymann. In: *Advanced Web Services*. New York: Springer, Jan. 2014. Kap. TOSCA: Portable Automated Deployment and Management of Cloud Applications, S. 527–549. ISBN: 978-1-4614-7534-7. DOI: 10.1007/978-1-4614-7535-4_22 (zitiert auf S. 27–29, 31, 32).

[BBLS12] T. Binz, G. Breiter, F. Leymann, T. Spatzier. „Portable Cloud Services Using TOSCA“. In: *IEEE Internet Computing* 16.3 (2012), S. 80–85 (zitiert auf S. 13, 29, 31, 45).

[BCC+16] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter. „Cloud-Native, Event-Based Programming for Mobile Applications“. In: *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Mai 2016, S. 287–288. DOI: 10.1145/2897073.2897713 (zitiert auf S. 20, 21).

[BCC+17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski et al. „Serverless Computing: Current Trends and Open Problems“. In: *arXiv preprint arXiv:1706.03178* (2017). URL: https://arxiv.org/abs/1706.03178 (zitiert auf S. 13, 15, 16).

[BCF+17] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, O. Tardieu. „The Serverless Trilemma: Function Composition for Serverless Computing“. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. Vancouver, BC, Canada: ACM, 2017, S. 89–103. ISBN: 978-1-4503-5530-8. DOI: 10.1145/3133850.3133855 (zitiert auf S. 19, 21).

[Clo18] Cloud Native Computing Foundation. *CNCF Serverless/FaaS Notes*. 2018. URL: https://docs.google.com/document/d/1L9n9tkGuGtj7Ap9dVRes9RVscSoXeKsF3k-d2hJcDlg (zitiert auf S. 13).

[Cui18] Y. Cui. *Why you should apply the single responsibility principle to serverless*. 2018. URL: https://medium.freecodecamp.org/why-you-should-apply-the-single-responsibility-principle-to-serverless-77810a24bd49 (zitiert auf S. 16, 17).

[Eiv17] A. Eivy. „Be Wary of the Economics of Serverless and Cloud Computing“. In: *IEEE Cloud Computing* 4.2 (März 2017), S. 6–12. ISSN: 2325-6095. DOI: 10.1109/MCC.2017.32 (zitiert auf S. 13, 16, 19).

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: https://www.rfc-editor.org/rfc/rfc2616.txt (zitiert auf S. 51).

[FIMS17] G. C. Fox, V. Ishakian, V. Muthusamy, A. Slominski. „Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research“. In: *CoRR* abs/1708.08028 (2017). arXiv: 1708.08028. URL: http://arxiv.org/abs/1708.08028 (zitiert auf S. 19).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. „Composite Cloud Application Patterns“. In: *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Vienna: Springer Vienna, 2014, S. 287–330. ISBN: 978-3-7091-1568-8. DOI: 10.1007/978-3-7091-1568-8_6 (zitiert auf S. 15).

[For17] E. Forbes. *How Serverless Computing will Change the World in 2018.* 2017. URL: https://hackernoon.com/how-serverless-computing-will-change-the-world-in-2018-7818fc06b447 (zitiert auf S. 20).

[Fow16] M. Fowler. *Serverless Architectures.* 2016. URL: https://martinfowler.com/articles/serverless.html (zitiert auf S. 16, 17, 19, 20).

[Kno16] E. Knorr. *What serverless computing really means.* 2016. URL: https://www.infoworld.com/article/3093508/cloud-computing/what-serverless-computing-really-means.html (zitiert auf S. 20).

[Kro16] D. Krook. *What makes serverless architectures so attractive?* 2016. URL: https://developer.ibm.com/opentech/2016/09/06/what-makes-serverless-attractive/ (zitiert auf S. 21).

[KY17] A. Kanso, A. Youssef. „Serverless: Beyond the Cloud“. In: *Proceedings of the 2Nd International Workshop on Serverless Computing.* WoSC ’17. Las Vegas, Nevada: ACM, 2017, S. 6–10. ISBN: 978-1-4503-5434-9. DOI: 10.1145/3154847.3154854 (zitiert auf S. 19).

[MB17] G. McGrath, P. R. Brenner. „Serverless Computing: Design, Implementation, and Performance“. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW).* Juni 2017, S. 405–410. DOI: 10.1109/ICDCSW.2017.36 (zitiert auf S. 13, 16).

[Mic17] Microsoft. *Serverless platform cuts time-to-market by two-thirds.* 2017. URL: http://customers.microsoft.com/en-us/story/quest?utm_content=buffer0d02a%5C&utm_medium=social%5C&utm_source=twitter.com%5C&utm_campaign=buffer (zitiert auf S. 20).

[MSJL06] J. McGovern, O. Sims, A. Jain, M. Little. „Event-Driven Architecture“. In: *Enterprise Service Oriented Architectures: Concepts, Challenges, Recommendations.* Dordrecht: Springer Netherlands, 2006, S. 317–355. ISBN: 978-1-4020-3705-4. DOI: 10.1007/1-4020-3705-8_8 (zitiert auf S. 16).

[Mün18] P. Müns. *Using the Event Gateway to build Multicloud Serverless Applications.* 2018. URL: https://www.slideshare.net/pmuens/using-the-event-gateway-to-build-multicloud-serverless-applications-jeffconf-hamburg-2018-88277475 (zitiert auf S. 25).

[OAS13a] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0.* 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html (zitiert auf S. 13, 27–32).

[OAS13b] OASIS. *Topology and Orchestration Specification for Cloud Applications Version (TOSCA) Primer Version 1.0.* 2013. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.pdf (zitiert auf S. 32, 39).

[OAS17] OASIS. *TOSCA Simple Profile in YAML Version 1.2*. 2017. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/csprd01/TOSCA-Simple-Profile-YAML-v1.2-csprd01.pdf (zitiert auf S. 28).

[Rab18] R. Rabbah. *Serverless functions in your favorite language with OpenWhisk*. 2018. URL: https://medium.com/openwhisk/serverless-functions-in-your-favorite-language-with-openwhisk-f7c447558f42 (zitiert auf S. 22).

[Ser18] Serverless Framework. *OpenWhisk - serverless.yml Reference*. 2018. URL: https://serverless.com/framework/docs/providers/openwhisk/guide/serverless.yml/ (zitiert auf S. 24).

[SMM18] J. Spillner, C. Mateos, D. A. Monge. „FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC“. In: *High Performance Computing: 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers*. Hrsg. von E. Mocskos, S. Nesmachnow. Cham: Springer International Publishing, 2018, S. 154–168. ISBN: 978-3-319-73353-1. DOI: 10.1007/978-3-319-73353-1_11 (zitiert auf S. 15, 16).

[Sti18] M. Stigler. „An Agnostic Approach“. In: *Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*. Berkeley, CA: Apress, 2018, S. 175–195. ISBN: 978-1-4842-3084-8. DOI: 10.1007/978-1-4842-3084-8_6 (zitiert auf S. 26, 27).

[Tho16] M. Thoemmes. *Uncovering the magic: How serverless platforms really work!* 2016. URL: https://medium.com/openwhisk/uncovering-the-magic-how-serverless-platforms-really-work-3cb127b05f71 (zitiert auf S. 21, 22).

[Tho17] M. Thoemmes. *Squeezing the milliseconds: How to make serverless platforms blazing fast!* 2017. URL: https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0 (zitiert auf S. 23).

[VGC+15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil. „Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud“. In: *2015 10th Computing Colombian Conference (10CCC)*. Sep. 2015, S. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476 (zitiert auf S. 17).

[YCCI16] M. Yan, P. Castro, P. Cheng, V. Ishakian. „Building a Chatbot with Serverless Computing“. In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs*. MOTA ’16. Trento, Italy: ACM, 2016, 5:1–5:4. ISBN: 978-1-4503-4669-6. DOI: 10.1145/3007203.3007217 (zitiert auf S. 21).

Alle URLs wurden zuletzt am 30. 04. 2018 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