Fakultät für Informatik und Mathematik 07

Bachelorarbeit

über das Thema

Sinnvolle Einsatzmöglichkeiten und Umsetzungsstrategien für serverless Webanwendungen

Meaningful Capabilities and Implementation Strategies for Serverless Web Applications

Autor: Thomas Großbeck
grossbec@hm.edu

Prüfer: Prof. Dr. Ulrike Hammerschall

Abgabedatum: 09.03.19

I Kurzfassung

Das Ziel der Arbeit ist es, Unterschiede in der Entwicklung von Serverless und klassischen Webanwendungen zu betrachten. Es soll ein Leitfaden entstehen, der Entwicklern und IT-Unternehmen die Entscheidung zwischen klassischen und Serverless Anwendungen erleichtert. Dazu wird zuerst eine Einführung in die Entwicklung des Cloud Computings und insbesondere in das Themenfeld des Serverless Computing gegeben. Im nächsten Schritt werden zwei beispielhafte Anwendungen entwickelt. Zum einen eine klassische Webanwendung mit der Verwendung des Spring Frameworks im Backend und einem Javascript basiertem Frontend und zum anderen eine Serverless Webanwendung. Hierbei werden die Besonderheiten im Entwicklungsprozess von Serverless Applikationen hervorgehoben. Abschließend werden die beiden Vorgehensweisen mittels vorher festgelegter Kriterien gegenübergestellt, sodass sinnvolle Einsatzmöglichkeiten für Serverless Anwendungen abgeleitet werden können.

II Inhaltsverzeichnis

III Abbildungsverzeichnis

IV Tabellenverzeichnis

V Listing-Verzeichnis

VI Abkürzungsverzeichnis

AWS Amazon Web Services

IaaS Infrastructure as a Service

PaaS Platform as a Service

FaaS Function as a Service

NIST National Institute of Standards and Technology

BaaS Backend as a Service

SaaS Software as a Service

SDK Software Development Kit

1 Einführung und Motivation

Durch das enorme Wachstum des Internets werden immer mehr Dienstleistungen über das Netz angeboten [Har02, S. 14]. Viele Dienste sind so als Webanwendung direkt zu erreichen und einfach zu bedienen. Mit der Einführung des Cloud Computings sind schließlich auch Rechenleistung und Serverkapazitäten über das Internet zur Verfügung gestellt worden.

Als eines der aktuell am schnellsten wachsenden Themenfeldern im Informatiksektor hat Cloud Computing eine rasante Entwicklung genommen. So ist beispielsweise der Anteil der deutschen Unternehmen, die Cloud Dienste nutzen, in den letzten Jahren stetig gestiegen. Mittlerweile sind es bereits zwei Drittel der Unternehmen. (siehe Abb. 1)

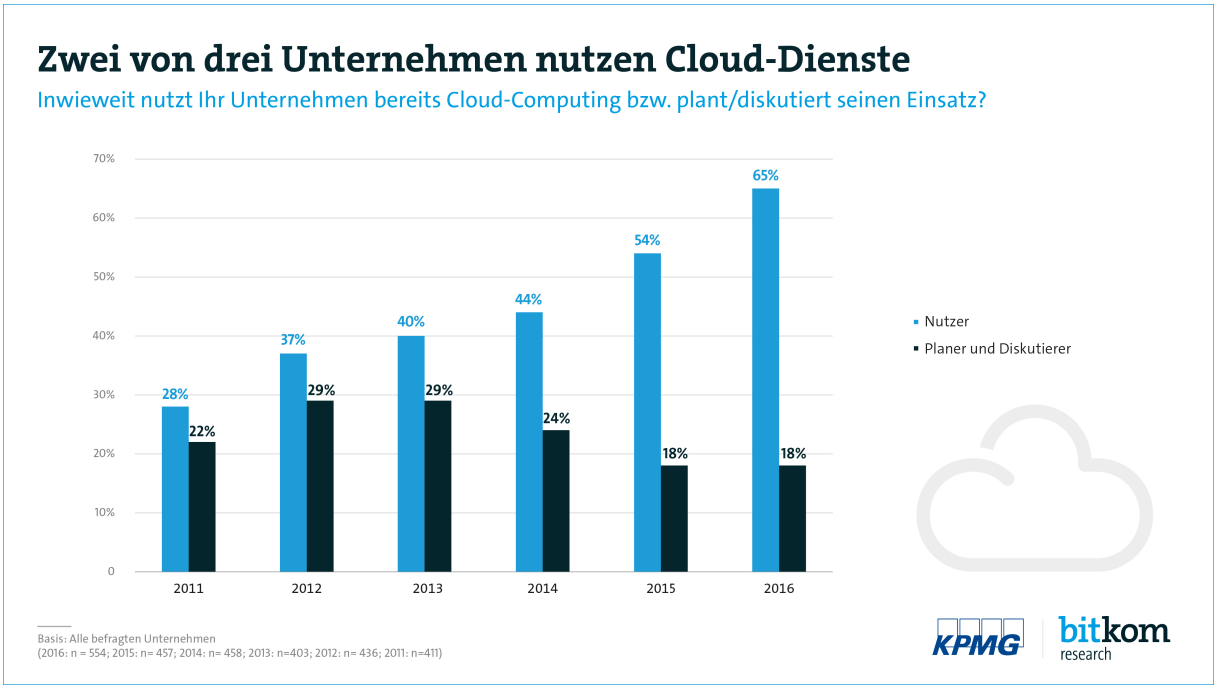

Abbildung 1: Anteil der Unternehmen, die Cloud Dienste nutzen [KS17]

Auf der Seite der Anbieter von Cloud Diensten ist ebenfalls ein großes Wachstum zu erkennen. Amazon als einer der Marktführer auf diesem Gebiet hat zum Beispiel im zweiten Quartal des Jahres 2018 55% des operativen Gewinns durch den Cloud Dienst Amazon Web Services (AWS) erzielt. (siehe Abb. 2)

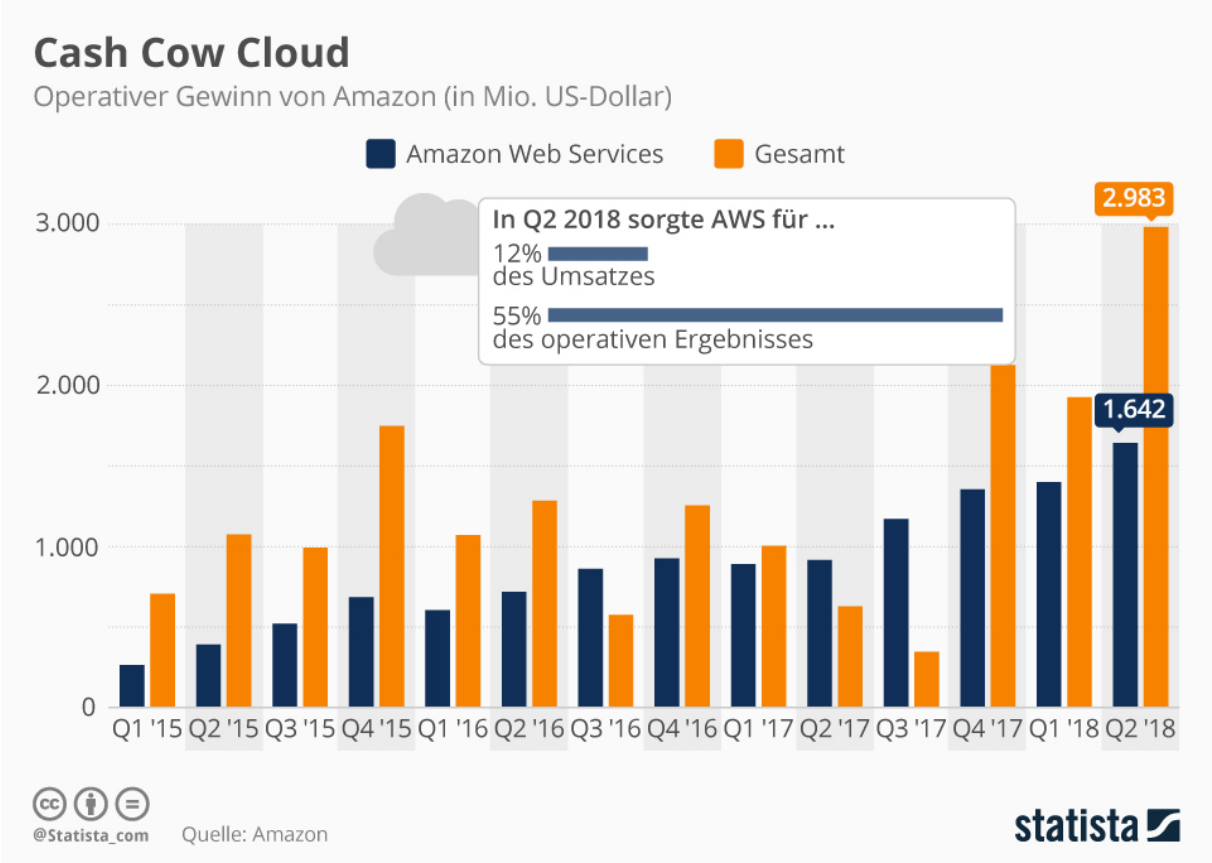

Abbildung 2: Operativer Gewinn von Amazon [Bra18]

Die neueste Stufe in der Entwicklung des Cloud Computings ist das Serverless Computing.

> *„Natürlich benötigen wir nach wie vor Server - wir kommen bloß nicht mehr mit ihnen in Berührung, weder physisch (Hardware) noch logisch (virtualisierte Serverinstanzen). [Kö17, S. 15]“*

Obwohl der Name einen serverlosen Betrieb suggeriert, müssen selbstverständlich Server bereitgestellt werden. Dies übernimmt, wie bei anderen Cloud Technologien auch üblich, der Plattform Anbieter. Allerdings muss sich nicht mehr um die Verwaltung der Server gekümmert werden. [Kö17, S. 15] Dies führt dazu, dass Serverless als sehr nützliches und mächtiges Werkzeug dienen kann. Die Tätigkeiten können dabei vom Prototyping und kleineren Hilfsaufgaben bis hin zur Entwicklung kompletter Anwendungen gehen. [Kö17, S. 11]

Da der Bereich Serverless erst vor wenigen Jahren entstanden ist und sich immer noch weiterentwickelt, gibt es bisher keine allzu große Verbreitung von Standards. Das heißt, es gibt wenige *Best Practice* Anleitungen und auch unterstützende Tools sind oftmals noch unausgereift. Somit ist es schwer für Unternehmen abzuwägen, ob es sinnvoll ist auf Serverless umzustellen bzw. Neuentwicklungen serverless umzusetzen.

Das Ziel der Arbeit ist es daher, die Unterschiede in der Entwicklung einer Serverless und einer klassischen Webanwendung anhand festgelegter Kriterien zu vergleichen, sodass hieraus sinnvolle Einsatzmöglichkeiten für Serverless Webanwendungen abgeleitet werden können, um die Vorteile des Serverless Computings ideal auszunutzen.

Um das Gebiet *Cloud Computing* besser kennenzulernen, wird zum Beginn der Arbeit die historische Entwicklung sowie Grundlagen des Themenfelds beschrieben (Kapitel 2.1). Ebenso werden Eigenschaften der Serverless Architektur erläutert (Kapitel 2.2 und Kapitel 2.3).

Im nächsten Schritt wird die prototypische Webanwendung in zweifacher Ausführung implementiert. Einmal als klassische Variante mit Hilfe des Spring Frameworks im Backend und zum anderen als Serverless Webapplikation. Hierzu werden zuerst die Kriterien sowie das Vorgehen zum Vergleich der beiden Anwendungen festgelegt (Kapitel 3.1). Nachdem die klassische Implementierung beschrieben wurde (Kapitel 3.3), wird die Serverless Umsetzung tiefer gehend betrachtet, um dem Leser einen umfangreichen Einblick in die neue Technologie zu ermöglichen (Kapitel 3.4). Abschließend werden die beiden Webanwendungen gegenüber gestellt und mittels der vorher erarbeiteten Kriterien Unterschiede in der Entwicklung herausgearbeitet (Kapitel 3.5).

Zuletzt werden anhand der Unterschiede Vor- und Nachteile einer Serverless Infrastruktur dargelegt, sodass letztendlich sinnvolle Einsatzmöglichkeiten für Serverless Webanwendungen benannt werden können (Kapitel 4).

2 Grundlagen der Serverless Architektur

2.1 Historische Entwicklung des Cloud Computings

Die Evolution des Cloud Computings begann in den sechziger Jahren. Es wurde das Konzept entwickelt Rechenleistung über das Internet anzubieten. John McCarthy beschrieb das Ganze im Jahr 1961 folgendermaßen. [Gar99, S. 1]

> *„If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as a telephone system is a public utility. [...] The computer utility could become the basis of a new and important industry.“*

McCarthy hatte also die Vision Computerkapazitäten als öffentliche Dienstleistung, wie beispielsweise das Telefon, anzubieten. Der Nutzer soll sich dabei nicht mehr selber um die Bereitstellung der Rechenleistung kümmern müssen, sondern die Ressourcen sind über das Internet verfügbar. Es wird je nach Nutzung verbrauchsorientiert abgerechnet.

Vor allen Dingen durch das Wachstum des Internets in den 1990er Jahren bekam die Entwicklung von Webtechnologien noch einmal einen Schub. Anfangs übernahmen traditionelle Rechenzentren das Hosting der Webseiten und Anwendungen. Hiermit einhergehend war allerdings eine limitierte Elastizität der Systeme. Skalierbarkeit konnte beispielsweise nur durch das Hinzufügen neuer Hardware erlangt werden. Neben der Hardware und dem Application Stack war der Entwickler außerdem für das Betriebssystem, die Daten, den Speicher und die Vernetzung seiner Applikation verantwortlich. [Inc18, S. 6]

Durch das Voranschreiten der Cloud-Technologien konnten immer mehr Teile des Entwicklungsprozesses abstrahiert werden, sodass sich der Verantwortlichkeitsbereich und auch das Anforderungsprofil an den Entwickler verschoben hat (siehe Abb. 3).

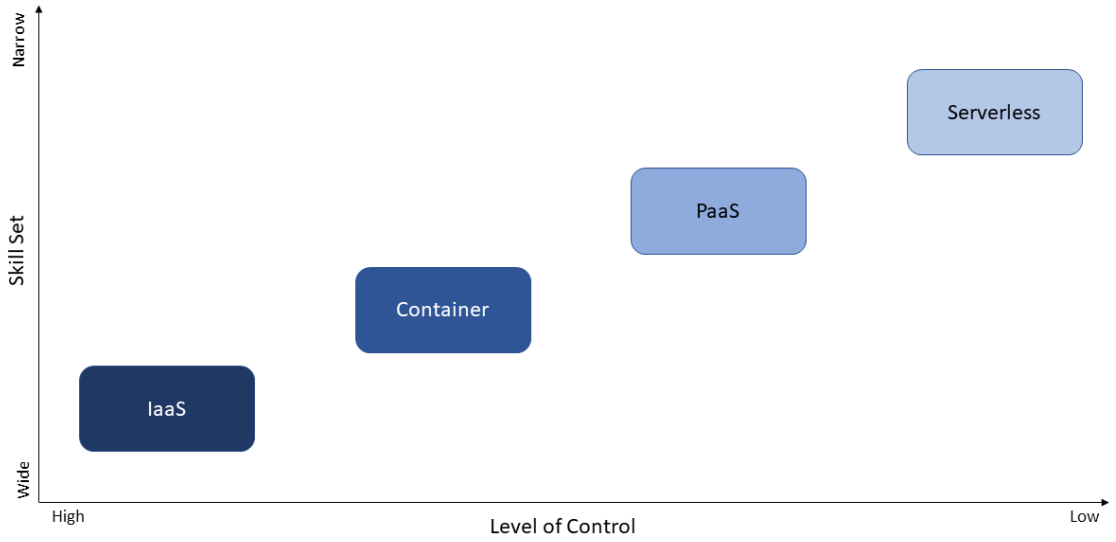

Abbildung 3: Zusammenhang Kenntnisstand und Kontroll-Level [Bü17]

Im ersten Schritt werden hierzu häufig Infrastructure as a Service (IaaS) Plattformen verwendet. Diese wurden für eine breite Masse verfügbar, als die ersten Anbieter in den frühen 2000er Jahren damit anfingen Software und Infrastruktur für Kunden bereitzustellen. Amazon beispielsweise veröffentlichte seine eigene Infrastruktur, die darauf ausgelegt war die Anforderungen an Skalierbarkeit, Verfügbarkeit und Performance abzudecken, und machte sie so 2006 als AWS für seine Kunden verfügbar. [RPMP17]

Ein weiterer Schritt in der Abstrahierung konnte durch die Einführung von Platform as a Service (PaaS) vollzogen werden. PaaS sorgt dafür, dass der Entwickler sich nur noch um die Anwendung und die Daten kümmern muss. Damit einhergehend kann eine hohe

Skalierbarkeit und Verfügbarkeit der Anwendung erreicht werden.

Auf der Virtualisierungsebene aufsetzend kamen schließlich noch Container hinzu. Diese sorgen beispielsweise für einen geringeren Ressourcenverbrauch und schnellere Bootzeiten. Bei PaaS werden Container zur Verwaltung und Orchestrierung der Anwendung verwendet. Es wird also auf die Kapselung einzelner wiederverwendbarer Funktionalitäten als Service geachtet. Dieses Schema erinnert stark an Microservices. Die genauere Abgrenzung zu Microservices wird im weiteren Verlauf der Arbeit behandelt. [Inc18, S. 6-7]

Als bisher letzter Schritt dieser Evolution entstand das Serverless Computing. Dabei werden zustandslose Funktionen in kurzlebigen Containern ausgeführt. Dies führt dazu, dass der Entwickler letztendlich nur noch für den Anwendungscode zuständig ist. Er unterteilt die Logik anhand des Function as a Service (FaaS) Paradigmas in kleine für sich selbstständige Funktionen. [Inc18, S. 7]

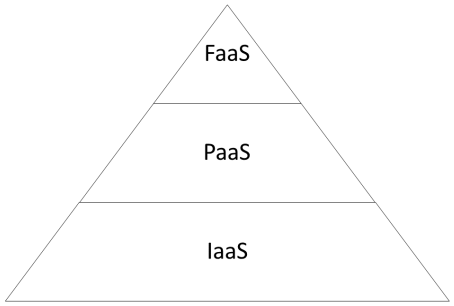

Abbildung 4: Hierarchie der Cloud Services [Kö17, S. 28]

2014 tat sich Amazon dann als Vorreiter für das Serverless Computing hervor und brachte AWS Lambda auf den Markt. Diese Plattform ermöglicht dem Nutzer Serverless Anwendungen zu betreiben. 2016 zogen Microsoft mit *Azure Function* und Google mit *Cloud Function* nach. [RPMP17]

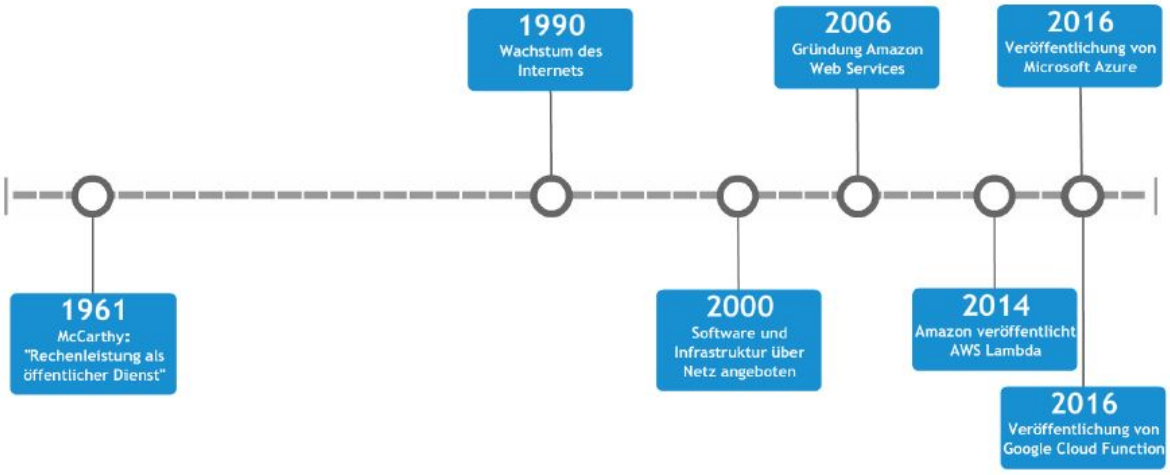

Abbildung 5: Historische Entwicklung des Cloud Computings

2.1.1 Grundlagen des Cloud Computings

„Run code, not Server [Rö17]“

Dies kann als eine der Leitlinien des Cloud Computings angesehen werden. Cloud-Angebote sollen den Entwickler entlasten, sodass die Anwendungsentwicklung mehr in den Fokus gerückt wird. Das National Institute of Standards and Technology (NIST) definiert Cloud Computing folgendermaßen. [MG11]

> *„Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.“*

Der Anwender kann also über das Internet selbstständig Ressourcen anfordern, ohne dass beim Anbieter hierfür ein Mitarbeiter eingesetzt werden muss. Der Kunde hat dabei allerdings keinen Einfluss auf die Zuordnung der Kapazitäten. Freie Ressourcen werden auch nicht für einen bestimmten Kunden vorgehalten. Dadurch kann der Anbieter schnell auf einen geänderten Bedarf reagieren und für den Anwender scheint es, als ob er unbegrenzte Kapazitäten zur Verfügung hat.

Zur Verwendung dieses Angebots stehen dem Nutzer verschieden Out-of-the-Box Dienste in unterschiedlichen Abstufungen zur Verfügung (siehe Abb. 6). Dies wären zum einen das IaaS Modell, bei dem einzelne Infrastrukturkomponenten wie Speicher, Netzwerkleistungen und Hardware durch virtuelle Maschinen verwaltet werden. Skalierung kann so zum Beispiel einfach durch allokieren weiterer Ressourcen in der virtuellen Maschinen erreicht

werden. [Sti17, S. 3]

Zum anderen das PaaS Modell. Dabei wird dem Entwickler der Softwarestack bereitgestellt und ihm werden Aufgaben wie Monitoring, Skalierung, Load Balancing und Server Restarts abgenommen. Ein typisches Beispiel hierfür ist Heroku. Ein Webservice bei dem der Nutzer seine Anwendung deployen und konfigurieren kann. [Sti17, S. 3]

Ebenfalls zu den Diensten gehört Backend as a Service (BaaS). Dieses Modell bietet modulare Services, die bereits eine standardisierte Geschäftslogik mitbringen, sodass lediglich anwendungsspezifische Logik vom Entwickler implementiert werden muss. Die einzelnen Services können dann zu einer komplexen Softwareanwendung zusammengefügt werden. [Rö17]

Die größte Abstraktion bietet SaaS. Hierbei wird dem Kunden eine konkrete Software zur Verfügung gestellt, sodass dieser nur noch als Anwender agiert. Beispiele dafür sind Dropbox und GitHub. [Sti17, S. 3]

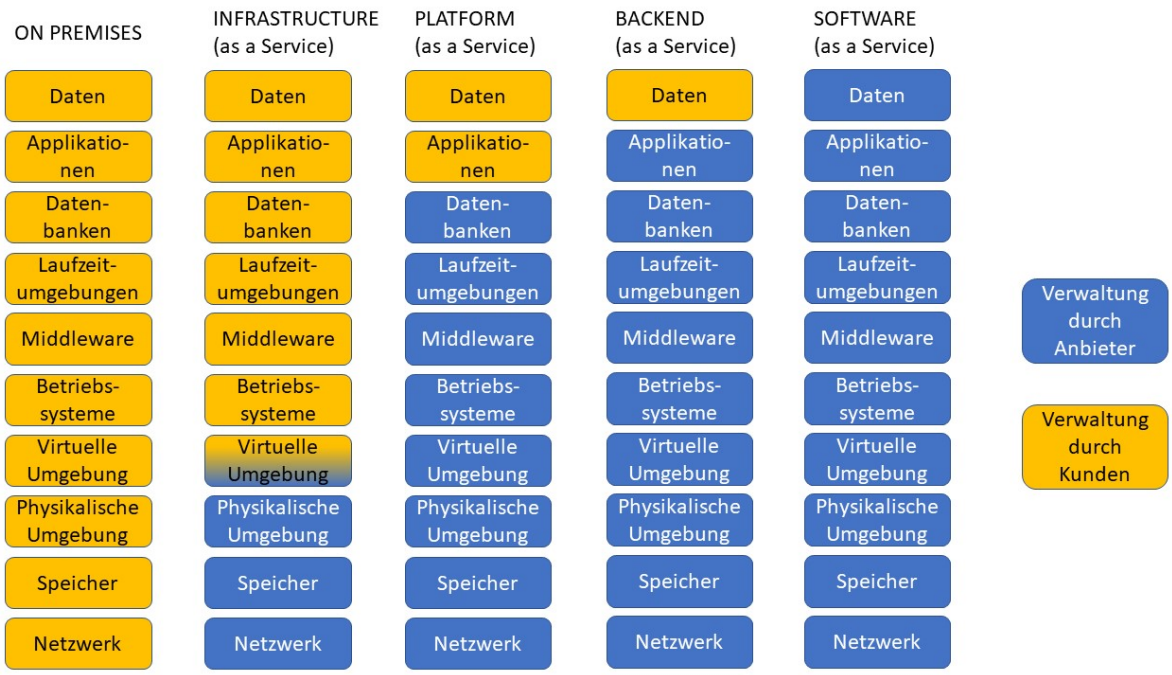

Abbildung 6: Verantwortlichkeiten der Organisation nach [Rö17]

Oftmals nutzen PaaS Anbieter ein IaaS Angebot und zahlen dafür. Nach dem gleichen Prinzip bauen SaaS Anbieter oft auf einem PaaS Angebot auf. So betreibt Heroku zum Beispiel seine Services auf Amazon Cloud Plattformen [Her18]. Ebenso ist es möglich eine Infrastruktur durch einen Mix der verschiedenen Modelle zusammenzustellen.

Letztendlich ist alles darauf ausgelegt, dass sich im Entwicklungs- und operationalen Auf-

wand so viel wie möglich einsparen lässt. Diese Weiterentwicklung wurde zum Beispiel in der Automobilindustrie bereits vollzogen. Dabei war es das Ziel die Fertigungstiefe, das heißt die Anzahl der eigenständig erbrachten Teilleistungen, zu reduzieren [Dja02, S. 8]. Nun findet diese Entwicklung auch Einzug in den Informatiksektor.

Ebenfalls von Bedeutung ist, dass die Anwendung automatisch skaliert und sich so an eine wechselnde Beanspruchung anpassen kann. Außerdem werden hohe Initialkosten für eine entsprechende Serverlandschaft bei einem Entwicklungsprojekt für den Nutzer vermieden und auch die Betriebskosten können gesenkt werden. Dem liegt das Pay-per-use-Modell zugrunde. Der Kunde zahlt aufwandsbasiert. Das heißt, er zahlt nur für die verbrauchte Rechenzeit. Leerlaufzeiten werden nicht mit einberechnet. [Rö17]

Da Cloud-Dienste dem Entwickler viele Aufgaben abnehmen und erleichtern, sodass sich die Verantwortlichkeiten für den Entwickler verschieben, ist dieser nun beispielsweise nicht mehr für den Betrieb sowie die Bereitstellung der Serverinfrastruktur zuständig. Dies führt allerdings auch dazu, dass ein gewisses Maß an Kontrolle und Entscheidungsfreiheit verloren geht.

2.1.2 Abgrenzung zu PaaS

Prinzipiell klingen PaaS und Serverless Computing aufgrund des übereinstimmenden Abstrahierungsgrades sehr ähnlich. Der Entwickler muss sich nicht mehr direkt mit der Hardware auseinandersetzen. Dies übernimmt der Cloud-Service in Form einer Blackbox, sodass lediglich der Code hochgeladen werden muss.

Jedoch gibt es auch einige grundlegenden Unterschiede. So muss der Entwickler bei einer PaaS Anwendung durch Interaktion mit der API oder Oberfläche des Anbieters eigenständig für Skalierbarkeit und Ausfallsicherheit sorgen. Bei der Serverless Infrastruktur übernimmt das Kapazitätsmanagement der Cloud-Service (siehe Abb. 7). Es gibt zwar auch PaaS Plattformen, die bereits Funktionen für das Konfigurationsmanagement bereitstellen, oft sind diese jedoch Anbieter-spezifisch, sodass der Programmierer auf weitere externe Tools zurückgreifen muss. [Bü17]

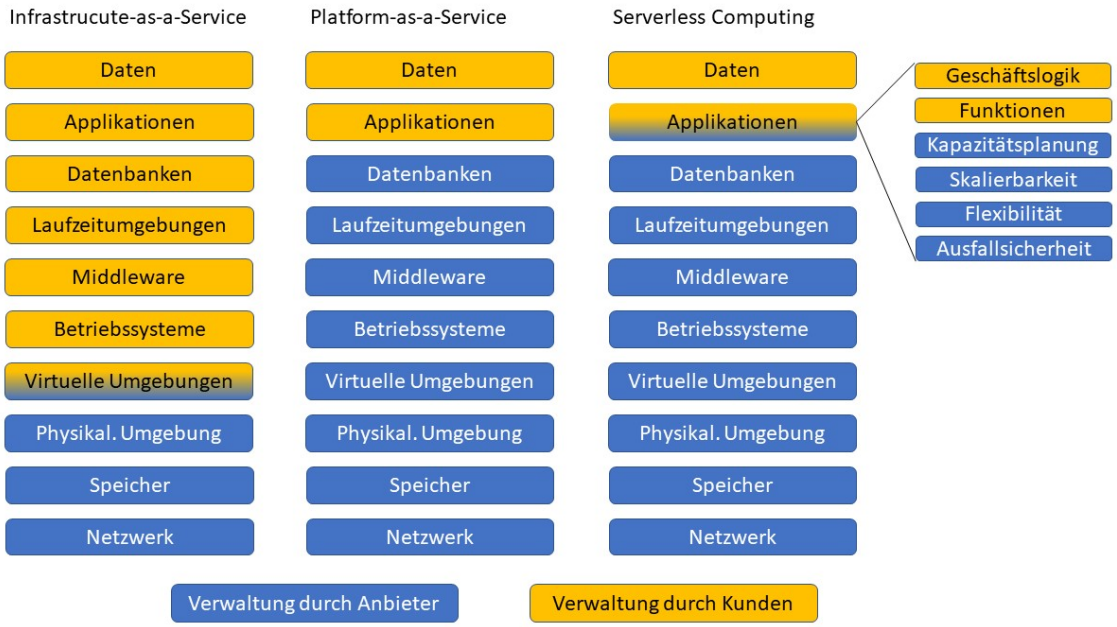

Abbildung 7: Aufgabenverteilung: IaaS vs. PaaS vs. Serverless [Bü17]

Ein weiterer Unterschied ist, dass PaaS für lange Laufzeiten konstruiert ist. Das heißt die PaaS Anwendung läuft immer. Bei Serverless hingegen wird die ganze Applikation als Reaktion auf ein Event gestartet und wieder beendet, sodass keine Ressourcen mehr verbraucht werden, wenn kein Request eintrifft. [Ash17]

Aktuell wird PaaS hauptsächlich wegen der sehr guten Toolunterstützung genutzt. Hier hat Serverless Computing den Nachteil, dass es durch den geringen Zeitraum seit der Entstehung noch nicht so ausgereift ist. [Rob18]

Final stechen als Schlüsselunterschiede zwei Punkte heraus. Dies ist zum einen wie oben bereits erwähnt die Skalierbarkeit. Sie ist zwar auch bei PaaS Applikationen erreichbar, allerdings bei weitem nicht so hochwertig und komfortabel. Zum anderen die Kosteneffizienz, da der Nutzer nicht mehr für Leerlaufzeiten aufkommen muss. Adrian Cockcroft von AWS bringt das folgendermaßen auf den Punkt. [Rob18]

> *„If your PaaS can efficiently start instances in 20ms that run for half a second, then call it serverless.“*

2.1.3 Abgrenzung zu Microservices

Bei der Entwicklung einer Anwendung kann diese in verschieden große Komponenten aufgeteilt werden. Das genaue Vorgehen wird dazu im Voraus festgelegt. Entscheidet sich das Entwicklerteam für eine große Einheit, wird von einer Monolithischen Architektur

gesprochen. Hierbei wird die komplette Applikation als ein Paket ausgeliefert. Dies hat den Nachteil, dass bei einem Problem die ganze Anwendung ausgetauscht werden muss. Auch die Einführung neuer Funktionalitäten braucht eine lange Planungsphase. [Inc18, S. 9]

Auf der anderen Seite steht die Microservice Architektur. Die Anwendung wird in kleine Services, die für sich eigenständige Funktionalitäten abbilden, aufgeteilt. Teams können nun unabhängig voneinander an einzelnen Services arbeiten. Auch der Austausch oder die Erweiterung einzelner Module erfolgt wesentlich reibungsloser. Dabei ist jedoch zu beachten, dass die Anonymität zwischen den Modulen gewahrt wird. Ansonsten kann auch bei Microservices die Einfachheit verloren gehen. Durch die Aufteilung in verschiedene Komponenten erreichen Microservice Anwendungen eine hohe Skalierbarkeit. [Bac18]

Das Konzept die Funktionalität in kleine Einheiten aufzuteilen, findet sich auch im Serverless Computing wieder. Im Gegensatz zur Microservices ist Serverless viel feingranularer. Bei Microservices wird oft das Domain-Driven Design herangezogen, um eine komplexe Domäne in sogenannte *Bounded Contexts* zu unterteilen. Diese Kontextgrenzen werden dann genutzt, damit die fachlichen Aspekte in verschiedene individuellen Services aufgeteilt werden können. [FL14] In diesem Zusammenhang wird auch oft von serviceorientierter Architektur gesprochen. Dahingegen stellt eine Serverless Funktion nicht einen kompletten Service dar, sondern eine einzelne Funktionalität. So eine Funktion kann beispielsweise gleichermaßen auch einen Event Handler darstellen. Daher handelt es sich hierbei um eine ereignisgesteuerte Architektur. [Tur18]

Ebenso ist es bei Serverless Anwendungen nicht notwendig die unterliegende Infrastruktur zur verwalten. Das heißt, dass lediglich die Geschäftslogik als Funktion implementiert werden muss. Weitere Komponenten wie beispielsweise ein Controller müssen nicht selbstständig entwickelt werden. Außerdem bietet der Cloud-Provider bereits eine automatische Skalierung als Reaktion auf sich ändernde Last an. Also auch hier werden dem Entwickler Aufgaben abgenommen. [Inc18, S. 9]

> *„The focus of application development changed from being infrastructure-centric to being code-centric. [Inc18, S. 10]“*

Im Vergleich zu Microservices rückt bei der Implementierung von Serverless Anwendungen die Funktionalität der Anwendung in den Fokus und es muss keine Rücksicht mehr auf die Infrastruktur genommen werden.

2.2 Eigenschaften von Function-as-a-Service

Wenn von Serverless Computing gesprochen wird, ist oftmals auch von FaaS die Rede. Der Serverless Provider stellt eine FaaS Plattform zur Verfügung. Die Infrastruktur des Anbieters kann dabei als BaaS gesehen werden. Eine Serverless Architektur stellt also eine Kombination aus FaaS und BaaS dar. [Rob18]

> *„FaaS entails running back-end code without the task of developing and deploying your own server applications and server systems. [Sti17, S. 3]“*

Der Fokus kann somit vollkommen auf die Geschäftslogik gelegt werden. Jede Funktionalität wird dabei in einer eigenen Function umgesetzt. [Ash17] Die Programmiersprache, in der die Anforderungen implementiert werden, hängt vom Anbieter der Plattform ab. Die geläufigen Sprachen, wie zum Beispiel Java, Python oder Javascript, werden allerdings von allen großen Providern unterstützt. [Tiw16] Jede Function stellt eine unabhängige und wiederverwendbare Einheit dar. Durch sogenannte Events kann eine Function angesprochen und aufgerufen werden. Hinter einem Event kann sich möglicherweise ein File-Upload oder ein HTTP-Request verbergen. Die dabei verwendeten Komponenten, wie zum Beispiel ein Datenbankservice, werden Ressourcen genannt. [RPMP17]

Die Functions sind alle zustandslos. Dadurch lassen sich in kürzester Zeit viele Kopien derselben Funktionalität starten, sodass eine hohe Skalierbarkeit erreicht werden kann. Alle benötigten Zusammenhänge müssen extern gespeichert und verwaltet werden, da sich prinzipiell der Zustand jeder Instanz vom Stand des vorherigen Aufrufs unterscheiden kann. Auch wenn es sich um dieselbe Function handelt. [Bü17]

Der Aufruf einer Function kann entweder synchron über das Request-/Response-Modell oder asynchron über Events erfolgen. Da der Code in kurzlebigen Containern ausgeführt wird, werden asynchrone Aufrufe bevorzugt. Dadurch kann sichergestellt werden, dass die Function bei verschachtelten Aufrufen nicht zu lange läuft. Bedingt durch die automatische Skalierung eignet sich FaaS somit besonders gut für Methoden mit einem schwankendem Lastverhalten. [Rö17]

Auch über die Verfügbarkeit muss sich der Nutzer keine Gedanken mehr machen, da der Dienstleister für die komplette Laufzeitumgebung verantwortlich ist. [Kö17, S. 28]

> *„Eine fehlerhafte Konfiguration hinsichtlich Über- oder Unterprovisionierung von (Rechen-, Speicher-, Netzwerk etc.) Kapazitäten können somit nicht passieren. [Kö17, S. 29]“*

Das heißt, dass alle Ressourcen mit bestmöglicher Effizienz genutzt werden. Die Architektur einer beispielhaften FaaS Anwendung könnte somit folgendermaßen ausschauen (siehe

Abb. 8). Hierbei nimmt das API Gateway die Anfragen des Clients entgegen und ruft die dazugehörigen Functions auf, die jeweils an einen eigenen Speicher angebunden sind. Neben der Möglichkeit HTTP-Requests über das API Gateway an die einzelnen Functions weiterzuleiten, kann auch das Hochladen einer Datei in den sogenannten *Blob Store* eine Function aufwecken. Ein Anwendungsfall in der hier aufgezeigten Beispiel-Anwendung könnte nun wie folgt ablaufen:
Ein Nutzer lädt in der Anwendung ein neues Profilbild hoch. Das API Gateway leitet den Request an die *Upload Function* weiter. Diese speichert das Bild im *Blob Store*, wodurch der Vorgang abgeschlossen sein könnte. Jedoch wird durch das Speichern in der Datenbank ein weiteres Event ausgelöst, das die *Activity Function* auslöst. Diese Function könnte nun zum Beispiel genutzt werden, um das neue Profilbild zu bearbeiten, sich die Bearbeitung in der zugehörigen Datenbank zu merken und es an den Browser zurück zu schicken. Der Vorteil dieses Vorgehens ist es, dass der Nutzer nach dem Hochladen des Bildes eine Antwort erhält und das System nicht bis zum Abschluss der Bearbeitung blockiert ist. Nebst der Möglichkeit die *Activity Function* asynchron über ein Event aufzurufen, kann sie auch über das API-Gateway erreicht werden. So könnte ein bereits bearbeitetes Bild noch einmal angepasst werden.

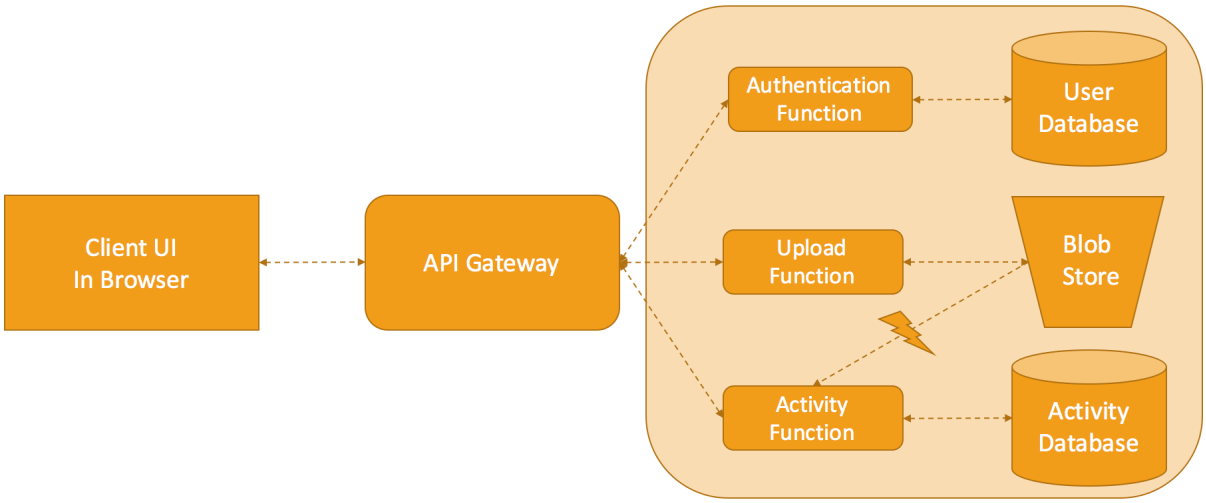

Abbildung 8: FaaS Beispiel Anwendung [Tiw16]

2.3 Allgemeine Patterns für Serverless Umsetzungen

Viele Grundsätze im Softwaresektor werden durch Manifeste festgehalten. Eines der bekanntesten Manifeste ist das *Agile Manifest*, das die agile Softwareentwicklung hervorbrachte. So ist es auch kaum verwunderlich, dass es im Bereich des Serverless Computing ebenfalls ein Manifest gibt. Das *Serverless Computing Manifesto*. [Kö17, S. 19]

Die Herkunft des Manifests kann nicht eindeutig geklärt werden. Niko Köbler äußert sich hierzu in seinem Buch *Serverless Computing in der AWS Cloud* folgendermaßen. [Kö17, S. 20]

> *„Allerdings findet sich hierfür kein dedizierter und gesicherter Ursprung, das Manifest wird aber auf mehreren Webseiten und Konferenzen einheitlich zitiert. Meine Recherche ergab eine erstmalige Nennung des Manifests und Aufzählung der Inhalte im April 2016 auf dem AWS Summit in Chicago in einer Präsentation namens "Getting Started with AWS Lambda and the Serverless Cloud"von Dr. Tim Wagner, General Manager für AWS Lambda and Amazon API Gateway.“*

Das Manifest besteht aus acht Leitsätzen, die nun genauer betrachtet werden. Einige Prinzipien wurden bereits in vorherigen Kapiteln angeschnitten oder erläutert.

Functions are the unit of deployment and scaling. Functions stellen den Kern einer Serverless Anwendung dar. Eine Function ist nur für eine spezielle Aufgabe verantwortlich und auch die Skalierung erfolgt bei Serverless Applikationen funktionsbasiert.

No machines, VMs, or containers visible in the programming model. Für den Nutzer der Plattform sind die Bestandteile der Serverinfrastruktur nicht sichtbar. Er kann mit der Implementierung nicht in die Virtualisierung oder Containerisierung eingreifen. Die Interaktion mit den Services des Providers erfolgt über bereitgestellte Software Development Kits (SDKs).

Permanent storage lives elsewhere. Serverless Functions sind zustandslos. Das heißt, dass die selbe Function beim mehrmaligen Ausführen in verschiedenen Umgebungen laufen kann, sodass der Nutzer nicht mehr auf vorherige Daten zurückgreifen kann. Zukünftig benötigte Daten müssen daher immer über einen anderen Dienst persistiert werden.

Scale per request. Users cannot over- or under-provision capacity. Die Skalierung erfolgt völlig automatisch durch den Serviceanbieter. Dieser sorgt dafür, dass die Functions parallel und unabhängig voneinander ausgeführt werden können, sodass der Kunde nicht mit diesem Aufgabenfeld in Berührung kommt. Hierzu cachen einige Anbieter die Containerumgebung, falls sie merken, dass eine Funktion in einem kurzen Zeitraum öfters aufgerufen wird, um eine bessere Performanz zu erreichen. Hierauf kann sich der User jedoch nicht verlassen.

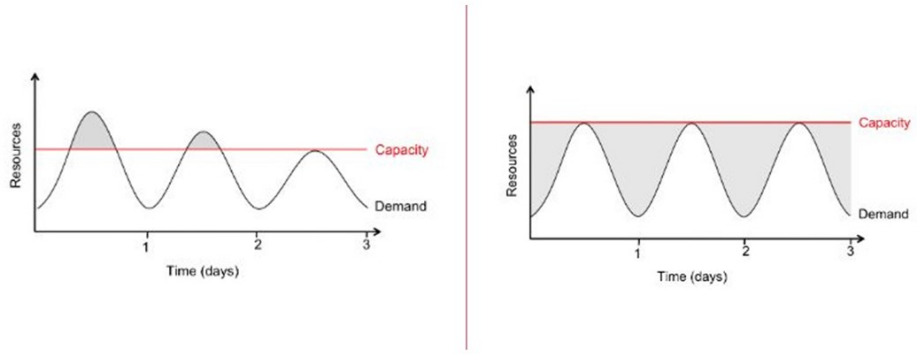

Abbildung 9: Under- und Overprovisioning [A+09, S. 11]

Im linken Diagramm ist die Auswirkung von *Underprovisioning* zu sehen. Hierbei kann es vorkommen, dass der Bedarf die gegebene Kapazität übersteigt und somit nicht mehr genug Ressourcen für alle Kunden bereitgestellt werden können. Dies führt zu unzufriedenen Nutzern und kostet schlussendlich dem Unternehmen Kunden. Bei *Overprovisioning* auf der rechten Seite hingegen ist die Kapazität gleich dem maximalen Bedarf. Hierdurch werden jedoch zu einem Großteil der Zeit mehr Ressourcen bereitgestellt als eigentlich benötigt, sodass unnötige Ausgaben entstehen.

Never pay for idle(no cold servers/containers or their cost). Der Kunde zahlt nur für die tatsächlich genutzte Rechenzeit. Die Bereitstellung der Ressourcen fällt dabei nicht ins Gewicht. Um dies dem Nutzer zu ermöglichen, sollten auf Seiten der Anbieter alle Ressourcen optimal ausgenutzt werden. So werden die Ressourcen keinem bestimmten Kunden zugeordnet, sondern stehen für viele Nutzer bereit. Je nach Bedarf können dem Anwender dynamisch benötigte Ressourcen aus einem großen Pool zugeteilt werden. Sobald die Function durchgelaufen ist, werden die Ressourcen wieder freigegeben und können von jedem anderen verwendet werden.

Implicitly fault-tolerant because functions can run anywhere. Da für den Nutzer nicht ersichtlich ist wo seine Functions beim Provider ausgeführt werden, darf in den Implementierungen auch keine Abhängigkeit diesbezüglich bestehen. Dies führt zu einer impliziten Fehlertoleranz, da der Betreiber keinen Einschränkungen unterliegt, in welchen Bereichen seiner Infrastruktur er bestimmte Functions ausführen darf.

BYOC - Bring Your Own Code. Eine Function muss alle benötigten Abhängigkeiten bereits enthalten. Der Anbieter stellt lediglich eine Ablaufumgebung zur Verfügung, sodass zur Laufzeit keine weiteren Bibliotheken nachgeladen werden können.

Metrics and logging are a universal right. Da für den Nutzer die Ausführung serverloser Services transparent abläuft und auch keinerlei Zustände in der Serverless Anwendung gespeichert werden, ist es für ihn nicht möglich Informationen über die Ausführung zu erhalten. Damit der User trotzdem Details seiner Anwendung zur Fehlersuche oder Analyse erhält, muss der Serviceprovider diese Möglichkeiten bereitstellen. So bietet er beispielsweise Logs zu einzelnen Funktionsaufrufen an. Des Weiteren werden Metriken, wie zum Beispiel Ausführungsdauer, CPU-Verwendung und Speicherallokation, zur Analyse der Applikation zur Verfügung gestellt. Das Loggen der Funktionsinhalte muss durch die Function selbst übernommen werden.

[Kö17, S. 20-23]

Neben dem *Serverless Computing Manifest* gibt es noch weitere Richtlinien, die bei der Umsetzung von Serverless Anwendungen in Betracht gezogen werden können. Sogenannte Pattern dienen dazu wiederkehrende Probleme bestmöglich und einheitlich zu lösen. Wie anhand des ausführlichen Manifests zu erkennen ist, lässt sich Serverless Computing nicht einfach in einem Pattern abarbeiten. Es spielen viele verschiedene Pattern zusammen.

Hierzu gehört zum Beispiel das Microservice Pattern. Es harmoniert hervorragend mit dem ersten Punkt des Manifests *Functions are the unit of deployment and scaling*. Jede Funktionalität wird in einer eigenen Function isoliert. Dies führt dazu, dass verschiedene Komponenten einzeln und unabhängig voneinander deployt bzw. bearbeitet werden können, ohne sich gegenseitig zu beeinflussen. Außerdem wird es einfach die Anwendung zu debuggen, da jede Function nur ein bestimmtes Event bearbeitet und somit die Aufrufe größtenteils vorhersehbar sind. Nachteilig hieran ist jedoch die Masse an Functions, die verwaltet werden müssen. Des Weiteren verlangsamt sich durch die große Anzahl an Komponenten der Deploymentprozess. [Hef16]

Der Aufruf der somit erstellten Functions führt zum nächsten Schema für Serverless Umsetzungen. Die ereignisgesteuerte Architektur sorgt dafür, dass die Functions durch Events aufgerufen werden können. Dieses Architekturmuster wird natürlich nicht nur bei Serverless Anwendungen verwendet, sondern kann auch in anderen Umfeldern zum Einsatz kommen. Es handelt sich bei Serverless Computing also lediglich um einen kleinen Bestandteil des *Event-driven computings*. (siehe Abb. 10) [Boy17]

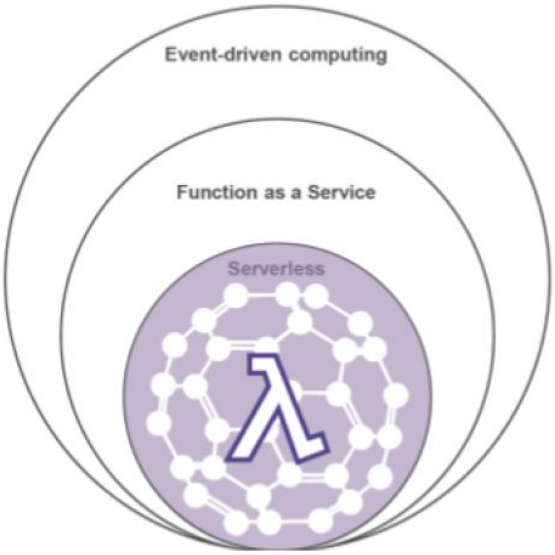

Abbildung 10: Zusammenhang zwischen Event-driven Computing, FaaS und Serverless [FIMS17, S. 5]

Neben asynchronen Events können Serverless Functions auch durch synchrone Nachrichten angesprochen werden. Hierzu kann als Einstiegspunkt einer Function ein HTTP-Endpunkt dienen. Der Aufruf folgt dann dem Request-Response Pattern, das als Basismethode zur Kommunikation zwischen zwei Systemen angesehen werden kann. Der Requester startet mit seinem Request die Kommunikation und wartet auf eine Antwort. Diese Anfrage ist der Aufruf einer Function. Der Provider auf der anderen Seite stellt die Function dar und wartet auf den Request. Nach der Abarbeitung sendet der Service seine Antwort an den Requester zurück. [Swa18]

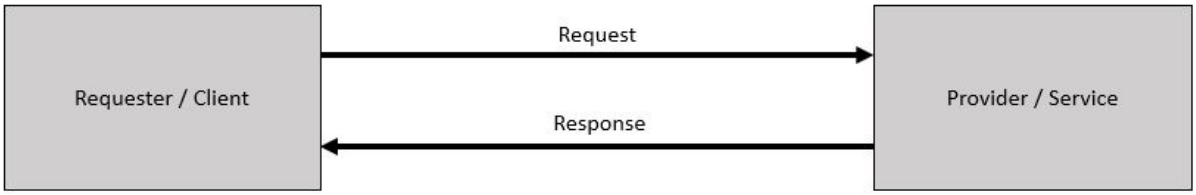

Abbildung 11: Request Response Pattern [Swa18]

3 Entwicklung einer prototypischen Anwendung

3.1 Vorgehensweise beim Vergleich der beiden Anwendungen

Zum Vergleich der beiden Anwendungen werden einige Kriterien abgearbeitet, die dabei helfen eine Aussage über die Qualität der jeweiligen Applikation zu treffen. Diese Kriterien werden nun im Folgenden genauer erläutert:

Implementierungsaufwand Es wird auf den zeitlichen Aufwand sowie auf die Codekomplexität geachtet. Das heißt, es wird untersucht, mit wie viel Einsatz einzelne Anwendungsfälle umgesetzt werden können und wie viel Overhead bei der Umsetzung möglicherweise entsteht.

Frameworkunterstützung Dabei wird analysiert inwieweit die Entwicklung durch Frameworks unterstützt werden kann. Dies gilt nicht nur für die Abbildung der Funktionalitäten, sondern auch für andere anfallende Aufgaben im Entwicklungsprozess wie zum Beispiel dem Testen und dem Deployment.

Deployment Beim Deploymentprozess sollen Änderungen an der Anwendung möglichst schnell zur produktiven Applikation hinzugefügt werden können, damit sie dem Kunden zeitnah zur Verfügung stehen. An dieser Stelle sind eine angemessene Toolunterstützung sowie die Komplexität der Prozesse ein großer Faktor. Optimal wäre in diesem Punkt eine automatische Softwareauslieferung.

Testbarkeit Hier ist zum einen ebenfalls der Implementierungsaufwand relevant und zum anderen sollte die Durchführung der Tests den Entwicklungsprozess nicht unverhältnismäßig lange aufhalten. Es ist dann auch eine effektive Einbindung der Tests in den Deploymentprozess gefragt. Im Speziellen werden mit den beiden Anwendungen Komponenten- und Integrationstests betrachtet.

Erweiterbarkeit Das Hinzufügen neuer Funktionalitäten oder Komponenten wird dabei im Besonderen überprüft. Damit einhergehend ist auch die Wiederverwendbarkeit einzelner Komponenten. Dies bedeutet, dass beleuchtet wird, ob einzelne Teile losgelöst vom restlichen System in anderen Projekten erneut einsetzbar sind.

Betriebskosten In einer theoretischen Betrachtung werden die Betriebskosten für die jeweiligen Anwendungen gegenübergestellt. So können anhand einer Hochrechnung für die Menge der benötigten Ressourcen die Kosten berechnet werden.

Performance Das Augenmerk liegt hierbei auf der Messung von Antwortzeiten einzelner Requests sowie der Reaktion des Systems auf große Last.

Sicherheit An dieser Stelle ist zum Beispiel die Unterstützung zum Anlegen einer Nutzerverwaltung von Interesse. Außerdem werden auch die Möglichkeiten bezüglich verschlüsselter Zugriffe genauer betrachtet.

Um diese Kriterien auf die beiden Applikationen anwenden zu können, werden nun für jede Kategorie Metriken aufgestellt. Die Metriken können dann genutzt werden, um die verschiedenen Eigenschaften der Anwendungen zu messen und vergleichbar zu machen.

3.2 Fachliche Beschreibung der Beispiel-Anwendung

Als Anwendungsfall für die Beispiel-Anwendung dient ein Bibliotheksservice. Der Service kann von zwei verschiedenen Anwendergruppen genutzt werden. Das wären auf der einen Seite Mitarbeiter der Bibliothek. Diese können Bücher zum Bestand hinzufügen oder löschen sowie Buchinformationen aktualisieren. Zur Vereinfachung der Anwendung gibt es zu jedem Buch nur ein Exemplar.

Auf der anderen Seite gibt es den Kunden, dem eine Übersicht aller Bücher zur Verfügung steht. Von diesen Büchern kann der Kunde beliebig viele verfügbare Bücher ausleihen, wobei eine Leihe unbegrenzt ist und somit kein Ablaufdatum besitzt. Seine ausgeliehene Bücher kann er dann auch wieder zurückgeben.

Um nutzerspezifische Informationen in der Anwendung anzeigen zu können und das System vor Fremdzugriffen zu schützen, hat jeder User einen eigenen Account. Mit diesem kann er sich an der Applikation an- und abmelden kann.

Damit der Servicebetreiber sein Angebot an die Nachfrage der Kunden anpassen kann, merkt sich das System bei jeder Ausleihe zusätzlich die Kategorie des ausgeliehenen Buches, sodass anhand der beliebten Bücherkategorien der Bestand sinnvoll erweitert werden kann.

Dieser Ablauf könnte in einem anderen Anwendungsfall beispielsweise eine Webseite sein, die den Nutzer nach der Auswahl eines Werbebanners nicht nur auf die werbetreibende Seite leitet, sondern sich gleichzeitig den Aufruf der Werbung merkt, um ihn später in Rechnung stellen zu können [Rob18].

519 ## 3.3 Implementierung der klassischen Webanwendung

520 ### 3.3.1 Architektonischer Aufbau der Applikation

521 ### 3.3.2 Implementierung der Anwendung

522 ### 3.3.3 Testen der Webanwendung

523 ## 3.4 Implementierung der Serverless Webanwendung

524 ### 3.4.1 Architektonischer Aufbau der Serverless Applikation

525 ### 3.4.2 Implementierung der Anwendung

526 ### 3.4.3 Testen von Serverless Anwendungen

527 ## 3.5 Unterschiede in der Entwicklung

528 ### 3.5.1 Implementierungsvorgehen

529 ### 3.5.2 Testen der Anwendung

530 ### 3.5.3 Deployment der Applikation

531 ### 3.5.4 Wechsel zwischen Providern

532 # 4 Vergleich der beiden Umsetzungen

533 ## 4.1 Vorteile der Serverless Infrastruktur

534 ## 4.2 Nachteile der Serverless Infrastruktur

535 ## 4.3 Abwägung sinnvoller Einsatzmöglichkeiten

536 # 5 Fazit und Ausblick

6 Quellenverzeichnis

[A+09] Armbrust, Michael u. a.: Above the Clouds: A Berkeley View of Cloud Computing. (2009). http://d1smfj0g31qzek.cloudfront.net/abovetheclouds.pdf. – Zuletzt Abgerufen am 09.01.2019

[Ash17] Ashwini, Amit: Everything You Need To Know About Serverless Architecture. (2017). https://medium.com/swlh/everything-you-need-to-know-about-serverless-architecture-5cdc97e48c09. – Zuletzt Abgerufen am 28.08.2018

[Bü17] Büst, René: Serverless Infrastructure erleichtert die Cloud-Nutzung. (2017). https://www.computerwoche.de/a/serverless-infrastructure-erleichtert-die-cloud-nutzung,3314756. – Zuletzt Abgerufen am 28.08.2018

[Bac18] Bachmann, Andreas: Wie Serverless Infrastructures mit Microservices zusammenspielen. (2018). https://blog.adacor.com/serverless-infrastructures-in-cloud_4606.html. – Zuletzt Abgerufen 09.11.2018

[Boy17] Boyd, Mark: Serverless Architectures: Five Design Patterns. (2017). https://thenewstack.io/serverless-architecture-five-design-patterns/. – Zuletzt Abgerufen am 10.01.2019

[Bra18] Brandt, Mathias: Cash Cow Cloud. (2018). https://de.statista.com/infografik/13665/amazons-operative-ergebnisse/. – Zuletzt Abgerufen am 01.12.2018

[Dal18] Daly, Jeremy: Serverless Micorservice Pattern for AWS. (2018). https://www.jeremydaly.com/serverless-microservice-patterns-for-aws/. – Zuletzt Abgerufen am 09.01.2019

[Dja02] Djabarian, Ebrahim: *Die strategische Gestaltung der Fertigungstiefe*. Deutscher Universitätsverlag, 2002. – ISBN 9783824476602

[FIMS17] Fox, Geoffrey C. ; Ishakian, Vatche ; Muthusamy, Vinod ; Slominski, Aleksander: Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research. (2017). https://arxiv.org/abs/1708.08028. – Zuletzt Abgerufen am 10.09.2018

[FL14] Fowler, Martin ; Lewis, James: Microservices. (2014). https://martinfowler.com/articles/microservices.html. – Zuletzt Abgerufen 19.11.2018

570 [Gar99] Garfinkel, Simson L.: *Architects of the Information Society: Thirty-Five*
571 *Years of the Laboratory for Computer Science at MIT*. The MIT Press, 1999.
572 – ISBN 9780262071963

573 [Har02] Hartmann, Anja K.: *Dienstleistungen im wirtschaftlichen Wan-*
574 *del: Struktur, Wachstum und Beschäftigung*. 2002 http://nbn-
575 resolving.de/urn/resolver.pl?urn=https://nbn-resolving.org/urn:
576 nbn:de:0168-ssoar-121435

577 [Hef16] Hefnawy, Eslam: Serverless Code Patterns. (2016). https://serverless.
578 com/blog/serverless-architecture-code-patterns/. – Zuletzt Abgeru-
579 fen am 10.01.2019

580 [Her18] Heroku: Heroku Security. (2018). https://www.heroku.com/policy/
581 security. – Zuletzt Abgerufen 08.11.2018

582 [Inc18] Inc., Serverless: Serverless Guide. (2018). https://github.com/
583 serverless/guide. – Zuletzt Abgerufen am 06.09.2018

584 [Kö17] Köbler, Niko: *Serverless Computing in der AWS Cloud*. entwickler.press,
585 2017. – ISBN 9783868028072

586 [KS17] Klingholz, Lukas ; Streim, Anders: Cloud Computing. (2017).
587 https://www.bitkom.org/Presse/Presseinformation/Nutzung-von-
588 Cloud-Computing-in-Unternehmen-boomt.html. – Zuletzt Abgerufen am
589 01.12.2018

590 [MG11] Mell, Peter ; Grance, Tim: The NIST Definition of Cloud Compu-
591 ting. (2011). https://csrc.nist.gov/publications/detail/sp/800-145/
592 final. – Zuletzt Abgerufen am 03.11.2018

593 [Rö17] Röwekamp, Lars: Serverless Computing, Teil 1: Theorie und Pra-
594 xis. (2017). https://www.heise.de/developer/artikel/Serverless-
595 Computing-Teil-1-Theorie-und-Praxis-3756877.html?seite=all. – Zu-
596 letzt Abgerufen am 30.08.2018

597 [Rob18] Roberts, Mike: Serverless Architectures. (2018). https://martinfowler.
598 com/articles/serverless.html. – Zuletzt Abgerufen am 30.08.2018

599 [RPMP17] Rai, Gyanendra ; Pasricha, Prashant ; Malhotra, Rakesh ; Pan-
600 dey, Santosh: Serverless Architecture: Evolution of a new para-
601 digm. (2017). https://www.globallogic.com/gl_news/serverless-

602 architecture-evolution-of-a-new-paradigm/. – Zuletzt Abgerufen am
603 30.08.2018

604 [Sti17] Stigler, Maddie: *Beginning Serverless Computing: Developing with Amazon*
605 *Web Services, Microsoft Azure, and Google Cloud*. Apress, 2017. – ISBN
606 9781484230831

607 [Swa18] Swarup, Pulkit: Microservices: Asynchronous Request Response Pat-
608 tern. (2018). https://medium.com/@pulkitswarup/microservices-
609 asynchronous-request-response-pattern-6d00ab78abb6. – Zuletzt Ab-
610 gerufen am 09.01.2019

611 [Tiw16] Tiwari, Abhishek: Stored Procedure as a Service (SPaaS). (2016). https:
612 //www.abhishek-tiwari.com/stored-procedure-as-a-service-spaas/.
613 – Zuletzt Abgerufen am 30.11.2018

614 [Tur18] Turvin, Neil: Serverless vs. Microservices: What you need to know for cloud.
615 (2018). https://www.computerweekly.com/blog/Ahead-in-the-Clouds/
616 Serverless-vs-Microservices-What-you-need-to-know-for-cloud. –
617 Zuletzt Abgerufen 15.11.2018